Conteúdo

1. Introdução a Redes de Computadores

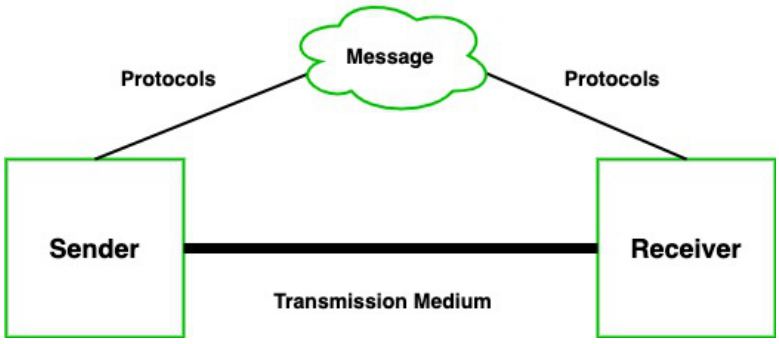

Figure 1: Cinco elementos (mensagem, transmissor, receptor, meio e protocolo) de um sistema básico de comunicação de dados, que não é restrito à internet

Redes de computadores são sistemas que conectam dois ou mais dispositivos para compartilhar dados e recursos, como arquivos, impressoras e acesso à internet. Essas conexões podem ser feitas por cabos, ondas de rádio ou outros meios, permitindo a comunicação entre máquinas de forma rápida e eficiente. As redes variam em tamanho e alcance, desde redes locais (LANs) até redes globais como a internet.

Pacotes são unidades menores em que os dados são divididos para serem transmitidos pela internet. Cada pacote contém parte da informação original, além de dados de controle, como endereços de origem e destino. Eles viajam separadamente pela rede, podendo seguir rotas diferentes, e são reorganizados no destino para reconstruir a mensagem completa. Esse processo permite uma comunicação mais eficiente e confiável, mesmo em redes complexas e congestionadas.

Protocolos são conjuntos de regras que definem como os dispositivos de uma rede devem se comunicar. Eles padronizam a forma como os dados são formatados, enviados, recebidos e interpretados, garantindo que sistemas diferentes consigam trocar informações corretamente. Sem protocolos, a comunicação na internet seria desorganizada e ineficiente.

TCP (*Transmission Control Protocol*) é responsável por garantir uma comunicação confiável entre dois dispositivos na internet. Ele estabelece uma conexão antes da troca de dados, verifica se os pacotes chegaram corretamente e na ordem certa, e retransmite os que forem perdidos. Além disso, controla o fluxo e o congestionamento da rede, assegurando que a transmissão ocorra de forma estável e segura. É usado em aplicações como navegadores, e-mails e transferências de arquivos.

As principais **topologias de redes incluem** a topologia em **barramento**, onde todos os dispositivos estão conectados a um único cabo, facilitando a instalação, mas tornando a rede vulnerável a falhas no cabo principal. Na topologia em **estrela**, os dispositivos se conectam a um ponto central, o que facilita o gerenciamento, mas torna a rede dependente desse ponto. A topologia em **anel** conecta cada dispositivo a dois outros, permitindo um desempenho consistente, mas podendo ser afetada por falhas em um único dispositivo. A topologia em **malha** oferece alta redundância e confiabilidade, pois cada dispositivo se conecta a vários outros, embora seja mais complexa e cara. Por fim, a topologia **híbrida** combina diferentes topologias, proporcionando flexibilidade e escalabilidade, mas pode ser complexa de gerenciar. A escolha da topologia depende das necessidades específicas da rede.

Os meios físicos se enquadram em duas categorias: meios guiados e meios não guiados. Nos meios guia dos, as ondas são dirigidas ao longo de um meio sólido, tal como um cabo de fibra ótica, um

par de fios de cobre trançado ou um cabo coaxial. Nos meios não guiados, as ondas se propagam na atmosfera e no espaço, como é o caso de uma LAN sem fio ou de um canal digital de satélite.

Os meios de **transmissão em redes** de computadores incluem cabos de cobre, fibra óptica e transmissão sem fio. Os **cabos de cobre**, como o **par trançado** e o **coaxial**, são amplamente utilizados devido ao seu custo acessível e facilidade de instalação, mas têm limitações em termos de distância e interferência. A **fibra óptica**, por outro lado, utiliza luz para transmitir dados, oferecendo alta velocidade e capacidade de transmissão em longas distâncias, além de ser imune a interferências eletromagnéticas, embora seu custo e complexidade de instalação sejam maiores. A **transmissão sem fio**, que inclui tecnologias como **Wi-Fi** e **Bluetooth**, proporciona mobilidade e facilidade de instalação, mas pode ser afetada por obstáculos físicos e interferências de outros dispositivos. Os meios de transmissão de dados em **redes móveis**, como 3G, 4G e 5G, são tecnologias sem fio que utilizam ondas de rádio para transmitir informações entre dispositivos móveis e torres de celular.

Mais recentemente há o Starlink é um sistema de **internet via satélite** criado pela SpaceX, que utiliza uma constelação de satélites em órbita baixa para fornecer conectividade em áreas remotas e rurais. Ao contrário das tecnologias móveis como 3G, 4G e 5G, que dependem de torres de celular, o Starlink transmite dados entre satélites e terminais na Terra, oferecendo altas velocidades de download e baixa latência.

A escolha do meio de transmissão depende das necessidades específicas da rede, como velocidade, distância e custo.

Os **tipos de redes** de computadores incluem LAN, WAN, MAN.

- A **LAN** (*Local Area Network*) é uma rede que cobre uma área geográfica limitada, como uma residência ou escritório, permitindo alta velocidade de transmissão e baixo custo; exemplos incluem redes domésticas conectadas via cabos Ethernet ou Wi-Fi.
- A **MAN** (*Metropolitan Area Network*) cobre uma área maior que uma LAN, mas menor que uma WAN, como uma cidade, e é frequentemente usada por empresas ou instituições para interconectar várias LANs; um exemplo seria uma rede que conecta diferentes filiais de uma empresa em uma área metropolitana.
- A **WAN** (*Wide Area Network*) abrange áreas geográficas extensas, conectando redes locais em diferentes cidades ou países, como a internet, que interliga diversas LANs ao redor do mundo.

1.1. Internet

Internet é a infraestrutura de uma rede global de computadores interconectados que trocam informações digitalizadas. Funciona com base em um conjunto de regras chamadas **protocolos**, que padronizam como os dados são enviados, roteados e recebidos. Dispositivos obtêm um endereço IP (Internet Protocol) que identifica unicamente cada nó na rede.[1]

- **Surface Web** é a parte da internet acessível por navegadores comuns e indexada por mecanismos de busca como Google.
- **Deep Web** engloba conteúdos que não estão indexados, como bancos de dados acadêmicos, sistemas internos de empresas, e-mails, ou páginas protegidas por login — ou seja, não são necessariamente ilegais.
- **Dark Web** é uma parte da Deep Web acessível apenas por tecnologias específicas, como a rede Tor (The Onion Router), que permite navegação anônima e criptografada. Nela, é comum encontrar fóruns, mercados ilegais, sendo muitas vezes associada a atividades ilegais justamente pela dificuldade de rastreamento e regulação (mas nem sempre, como o caso de jornalistas que

[1] Como Funciona a Internet?

a usam para comunicação de dados sensíveis). Também é comum o uso de criptomoedas, como Bitcoin, para garantir anonimato nas transações.

1.1.1. Linha do tempo da internet

- **(1969)** A ARPANET foi a primeira rede de computadores a implementar a comutação de pacotes, permitindo a comunicação entre diferentes sistemas. Criada pela DARPA, ela conectava universidades e centros de pesquisa nos EUA e tornou-se a base técnica da Internet moderna.
- **(1972)** Ray Tomlinson enviou o primeiro e-mail entre dois computadores conectados à ARPANET, utilizando o símbolo `@` para separar o nome do usuário do nome do computador. Isso revolucionou a comunicação digital.
- **(1974)** Vint Cerf e Robert Kahn desenvolveram o protocolo TCP/IP, que permitiu a interconexão de diferentes redes, formando a base da internet como conhecemos hoje.
- **(1983)** Criado para traduzir endereços IP numéricos (difíceis de memorizar) em nomes amigáveis como "exemplo.com", o Sistema de Nomes de Domínio (DNS) foi introduzido, tornando a navegação na internet mais amigável.
  - Em 1° de janeiro de 1983, ocorre a migração do Network Control Protocol (NCP) para o padrão TCP/IP, idealizado por Vint Cerf e Bob Kahn, criando um protocolo universal que permitiu a interconexão de diversas redes.
- **(1990)** Tim Berners-Lee desenvolveu a World Wide Web no CERN, criando um sistema de navegação baseado em hipertextos. Ele também criou o primeiro navegador e servidor web, facilitando o acesso à informação na internet.
- **(1993)** Browsers gráficos como Mosaic popularizam o acesso, impulsionando o crescimento comercial da rede.
- **(1994)** Realizada a primeira compra segura usando SSL: um CD da banda Sting foi comprado pela NetMarket, demonstrando a viabilidade do e-commerce seguro.
- **(2000 >)** Nos anos 2000, a Internet migra para conexões de banda larga[2], trazendo streaming, e-mail robusto, rede sociais (Wikipedia, MySpace, Facebook), smartphones e tecnologias como Wi-Fi, 3G e 5G – conectando bilhões globalmente.

[2]Banda larga é um tipo de conexão à internet que permite transmitir grandes quantidades de dados rapidamente e de forma contínua. Ela é caracterizada por ser sempre ativa (não precisa "discá-la" como as antigas) e ter alta velocidade de download e upload, permitindo, por exemplo, ver vídeos, ouvir músicas e jogar online com qualidade. Antes da banda larga, o acesso à internet era feito principalmente por discagem telefônica (conhecida como internet discada ou dial-up)

2. Microsserviços vs. aplicações monolíticas

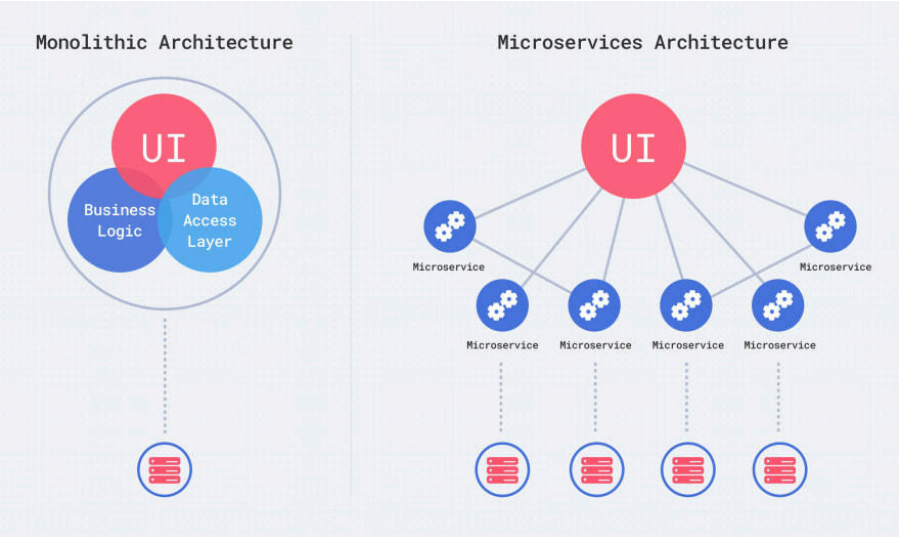

Figure 2: Diferenças entre arquiteturas monolíticas e de microsserviços.

Aplicações monolíticas é um sistema construído como uma única base de código coesa, onde todos os módulos — interface, lógica de negócios e acesso a dados — estão integrados e implantados juntos. Todas as funcionalidades, mesmo que distintas, compartilham o mesmo banco de dados, servidor e ambiente, o que torna mais simples desenvolver e testar inicialmente, mas pode dificultar a escalabilidade, manutenção e implantação à medida que o sistema cresce.[3]

Uma das principais **vantagens** desse modelo é a facilidade no desenvolvimento inicial, o que o torna ideal para projetos pequenos ou MVPs (produtos mínimos viáveis). O processo de deploy é simples, já que toda a aplicação é empacotada em um único artefato, o que também reduz a complexidade operacional.

No entanto, à medida que a aplicação cresce, começam a surgir **desvantagens**. O sistema se torna mais difícil de manter, já que os módulos estão fortemente acoplados. Escalar uma funcionalidade específica exige escalar toda a aplicação, o que é ineficiente. Além disso, qualquer mudança ou falha em uma parte pode afetar o sistema inteiro, tornando os deploys mais arriscados. Por isso, aplicações monolíticas são mais apropriadas para sistemas menores, com equipes reduzidas e requisitos simples de operação.

Microsserviços é uma arquitetura em que a aplicação é dividida em pequenos serviços independentes, cada um responsável por uma função específica do negócio. Esses serviços se comunicam geralmente por APIs (ver em seção 3) e podem ser desenvolvidos, implantados e escalados separadamente. Isso traz mais flexibilidade, resiliência e autonomia para as equipes, mas aumenta a complexidade da operação, exigindo boas práticas de monitoramento, comunicação e infraestrutura.[4] |[5]

Isso oferece **vantagens** importantes como a escalabilidade granular, permitindo que apenas partes específicas do sistema sejam escaladas conforme a demanda. Também aumenta a resiliência, pois uma falha em um serviço não compromete todo o sistema. Além disso, times podem trabalhar de forma autônoma em diferentes serviços, escolhendo tecnologias adequadas para cada contexto.

[3] Monólitos vs Microserviços | Explicado em 5 minutos

[4] Microservices Explained in 5 Minutes

[5] O que são microsserviços

Entretanto, essa abordagem também traz **desafios**. A complexidade operacional cresce, exigindo ferramentas para orquestração, deploy contínuo, monitoramento, comunicação entre serviços e gerenciamento de falhas. A troca de dados entre os microsserviços pode introduzir latência e tornar mais difícil garantir a consistência das informações. Assim, os microsserviços são mais indicados para sistemas grandes, com múltiplos domínios de negócio, equipes maiores e necessidade de alta escalabilidade e independência entre os módulos.[6]

[6] Qual é a diferença entre arquitetura monolítica e de microsserviços?

3. APIs

API (*Application Programming Interface*) é um conjunto de regras, protocolos e ferramentas que permite que diferentes peças de softwares se comuniquem e troquem dados entre si.[7]

Por exemplo, para que um app de entregas possa localizar um endereço ele faz uma requisição à API ViaCEP, enviando um CEP: `https://viacep.com.br/ws/01001000/json/`. A API acessa os dados de endereços e devolve como resposta ao app informações estruturadas em JSON do endereço solicitado. Isso ocorre sem que o desenvolvedor precise saber como a API realmente funciona por trás dos panos (caixa preta).

APIs podem ser locais (em bibliotecas ou sistemas operacionais) ou expostas na web (como REST, SOAP, RPC), permitindo integrações, reutilização de código e criação de ecossistemas conectados de forma padronizada.

A arquitetura da API geralmente é explicada em termos de cliente e servidor. A aplicação que envia a solicitação é chamada de cliente e a aplicação que envia a resposta é chamada de servidor. Então, no exemplo do clima, o ViaCEP é o servidor e o app é o cliente.

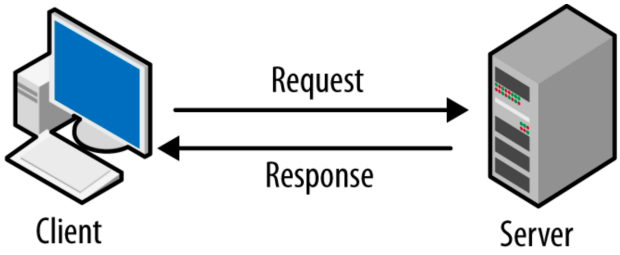

Figure 3: APIs dentro da arquitetura cliente-servidor.

3.1. REST e RESTful

REST é um estilo arquitetural definido por Roy Fielding que estabelece um conjunto de restrições para comunicação entre sistemas distribuídos via HTTP, como cliente-servidor, stateless, cache, sistema em camadas, interface uniforme e, idealmente, HATEOAS.

API RESTful (ou RESTful API) é uma interface de comunicação entre sistemas que segue os princípios do estilo arquitetural REST(*Representational State Transfer*).[8]

Os os princípios da arquitetura REST são:

- Usa o protocolo HTTP para comunicação e manipula recursos identificados por URLs, como por exemplo `https://viacep.com.br/ws/01001000/json/`.
- Ela utiliza os métodos HTTP corretos: `GET` para buscar dados, `POST` para criar, `PUT` ou `PATCH` para atualizar e `DELETE` para remover.
- É **stateless**, ou seja, cada requisição carrega todas as informações necessárias, sem depender de um estado salvo no servidor.
- As respostas geralmente retornam dados em formatos como JSON ou XML, com cabeçalhos que indicam o formato, cache, autenticação e status da operação.

As APIs REST oferecem diversos benefícios que as tornam amplamente utilizadas no desenvolvimento moderno de software. Em primeiro lugar, elas seguem padrões bem estabelecidos da web, como o protocolo HTTP, o uso de URLs para identificar recursos e formatos de dados como JSON, o que facilita tanto o consumo quanto a implementação das APIs. Além disso, são baseadas em

[7] APIs explicadas em 6 minutos!

[8] The Right Way To Build REST APIs

uma arquitetura stateless, ou seja, cada requisição é independente e carrega todas as informações necessárias, o que favorece a escalabilidade, já que o servidor não precisa manter estado entre as chamadas.

Outro benefício importante é a possibilidade de utilizar cache, o que melhora o desempenho e reduz a carga no servidor, já que os dados podem ser reutilizados pelo cliente sem necessidade de novas requisições. As APIs REST também promovem um forte desacoplamento entre cliente e servidor, permitindo que cada parte evolua de forma independente, além de permitir a inclusão de camadas intermediárias, como proxies e gateways.

A interface padronizada baseada em verbos HTTP torna o uso intuitivo e facilita a manipulação de recursos. Além disso, os dados geralmente são transmitidos em formatos leves e legíveis, como JSON, o que simplifica o desenvolvimento, a depuração e a integração com outros sistemas. Por fim, REST oferece flexibilidade para evolução contínua da API, suporte a diferentes formatos e grande interoperabilidade, sendo ideal para aplicações distribuídas, microserviços e integrações com sistemas diversos (aplicativos móveis, navegadores e etc).

Ao comparar essas característica do REST a alternativas, temos:

- SOAP: É um protocolo de comunicação rígido que utiliza XML e possui uma estrutura de mensagens estrita, com regras definidas em WSDL. Ao contrário do REST, não é stateless por padrão, o que pode resultar em maior sobrecarga, menor legibilidade e uso limitado de cache, levando a uma complexidade e desempenho inferiores em sistemas distribuídos modernos.
  - No SOAP, o cliente é fortemente acoplado ao servidor porque depende de um contrato rígido definido via WSDL, que especifica exatamente os métodos disponíveis, os parâmetros esperados e a estrutura das mensagens. Isso significa que qualquer mudança no servidor pode quebrar o funcionamento do cliente, exigindo atualizações coordenadas. Esse acoplamento reduz a flexibilidade e dificulta a evolução independente dos sistemas, ao contrário do REST, que promove um acoplamento mais fraco e maior tolerância a mudanças.
- GraphQL: uma linguagem de consulta que centraliza o acesso a dados em uma única rota, permitindo que o cliente especifique exatamente quais dados deseja. Isso proporciona flexibilidade, mas aumenta a complexidade do servidor, dificulta o uso de cache em níveis HTTP e complica o monitoramento e controle de acesso, já que não se baseia em recursos padronizados como URLs.

Endpoints um endpoint em uma API RESTful é o endereço exato (URL) onde o cliente faz requisições para acessar ou manipular recursos no servidor. Cada endpoint representa uma porta de entrada específica, por exemplo, `/usuarios` para listar todos os usuários ou `/usuarios/123` para obter, atualizar ou deletar o usuário com ID `123` . Quando o cliente envia uma requisição HTTP (como GET, POST, PUT, DELETE) para esse endereço, o servidor processa o pedido, valida e autentica (caso necessário) e responde com os dados ou o status correspondente.[9]

Endpoints são fundamentais pois fazem a conexão entre cliente e servidor.

Recursos são entidades ou objetos que representam dados manipulados pela aplicação, como usuários, produtos, posts, etc. Cada recurso é acessado por uma URL única (endpoint) e pode ser manipulado por métodos HTTP de forma padronizada.

Requisição ou solicitação REST é uma mensagem enviada por um cliente (como um navegador ou aplicativo) a um servidor, com o objetivo de acessar, criar, alterar ou excluir recursos. Ela é composta por elementos como o método HTTP (GET, POST, PUT, DELETE), a URL do recurso,

[9] What is an API Endpoint?

os cabeçalhos (com informações como tipo de conteúdo) e, quando necessário, um corpo com os dados enviados.

- **Cabeçalho** (header): contém metadados da requisição, como o tipo de conteúdo enviado (Content-Type: application/json), o formato esperado na resposta (Accept: application/json), tokens de autenticação, e outras informações de controle.
- **Corpo** (body): contém os dados reais enviados ao servidor, geralmente em formato JSON, XML ou formulário. É usado principalmente em métodos como POST ou PUT para enviar informações como novos registros ou atualizações (ex: dados de um novo usuário).

Há diferentes tipos de **parâmetros** em APIs RESTful, onde cada tipo de parâmetro tem um papel específico e é usado de acordo com o tipo de operação que queremos realizar com a API:

- ***Path parameters***: são partes fixas da URL que servem para identificar um recurso específico. Por exemplo, em GET `/usuarios/5`, o 5 é o ID do usuário que queremos acessar. Esses parâmetros são obrigatórios e usados quando queremos manipular um recurso diretamente.
- ***Query parameters***: são usados para refinar uma consulta e aparecem após o sinal de interrogação na URL. Por exemplo, em GET `/produtos?categoria=livros&pagina=2`, os parâmetros categoria e pagina ajudam a filtrar ou paginar os resultados. Eles são opcionais e usados principalmente para busca e filtros.
- ***Header parameters***: são enviados no cabeçalho da requisição HTTP e não aparecem na URL. Eles carregam informações de controle como autenticação (Authorization) ou tipo de conteúdo (Content-Type). São úteis para configurar o comportamento da comunicação entre cliente e servidor. Ex: `Authorization: Bearer token123`
- ***Body parameters***: aparecem no corpo da requisição, geralmente em métodos como POST, PUT e PATCH. Servem para enviar dados estruturados (normalmente em JSON), como quando criamos ou atualizamos um recurso. Ex: `{ "nome": "João", "email": "joao@example.com" }`

Métodos HTTP (ou verbos HTTP) são utilizados para indicar a ação que o cliente deseja realizar em relação a um recurso no servidor.[10]

Os principais são:

Método	Descrição	Idempotente?
`GET`	Recupera dados de um recurso específico. É seguro e não altera o estado do servidor.	Sim
`POST`	Envia dados para criar um novo recurso. Pode alterar o estado do servidor e criar múltiplos recursos.	Não
`PUT`	Atualiza um recurso existente ou cria se não existir. O resultado é sempre o mesmo após múltiplas requisições.	Sim
`DELETE`	Remove um recurso específico do servidor. A mesma requisição repetida tem o mesmo efeito.	Sim

Um método HTTP é considerado **idempotente** quando a realização da mesma operação múltiplas vezes tem o mesmo efeito que realizá-la uma única vez. Em outras palavras, não importa quantas vezes você execute a mesma requisição, o resultado final no servidor será o mesmo. Isso é especial-

[10] HTTP Request Methods | GET, POST, PUT, DELETE

mente útil em situações de falhas de rede, onde uma requisição pode ser enviada várias vezes. Métodos idempotentes garantem que o estado do servidor permaneça consistente, independentemente de quantas vezes a mesma operação é realizada.

Status code ou códigos de status HTTP, são respostas numéricas de três dígitos que um servidor envia para indicar o resultado de uma requisição feita por um cliente.[11]

Eles são divididos em cinco categorias principais:

1. Os códigos **1xx** indicam que a requisição foi recebida e está em processamento;
2. Os **2xx** confirmam que a operação foi bem-sucedida, como o 200 (OK) ou 201 (Created);
3. Os **3xx** indicam redirecionamento, como o 301 (Moved Permanently) ou 302 (Found);
4. Os **4xx** sinalizam erros causados pelo cliente, como o 400 (Bad Request), 401 (Unauthorized), 403 (Forbidden) e 404 (Not Found);
5. Os **5xx** indicam falhas internas do servidor, como o 500 (Internal Server Error) ou 503 (Service Unavailable).

Figure 4: Erro 404.

Esses códigos ajudam clientes e desenvolvedores a entender o que ocorreu com a requisição, possibilitando reações apropriadas como reenvios, redirecionamentos ou exibição de mensagens de erro.

Para ver uma demonstração de todos esses conceitos em python veja a seção 4.1.9

O **versionamento de APIs**, é uma prática essencial para garantir que mudanças na API não afetem negativamente os sistemas que já a utilizam. À medida que uma API evolui, é comum que endpoints sejam modificados, parâmetros mudem ou funcionalidades antigas sejam descontinuadas. Sem versionamento, qualquer alteração pode quebrar o funcionamento de aplicações clientes, causando erros e exigindo ajustes imediatos. Para evitar isso, versionar permite manter múltiplas versões da API ativas ao mesmo tempo, oferecendo estabilidade para quem consome os serviços. Há formas de fazer esse versionamento:

- pela URI (como `/api/v1/orders`), que é a forma mais comum e de fácil identificação;
- por parâmetros de consulta (exemplo: `/api/orders?version=1`), menos usada e menos amigável a caches;
- e por cabeçalhos HTTP personalizados, como `Accept: application/vnd.myapi.v2+json`, que mantém a URL limpa e é considerada mais alinhada ao estilo REST, mas tem menor suporte por ferramentas simples.

[11] HTTP Status Codes Explained In 5 Minutes

Swagger é uma ferramenta que facilita a documentação e o teste de APIs REST, permitindo que desenvolvedores descrevam suas APIs de forma padronizada usando o formato OpenAPI. Com isso, é possível gerar automaticamente uma interface visual interativa, onde é possível ver os endpoints disponíveis, os parâmetros esperados e até testar as requisições diretamente no navegador. Isso ajuda tanto no desenvolvimento quanto na comunicação entre equipes.[12]

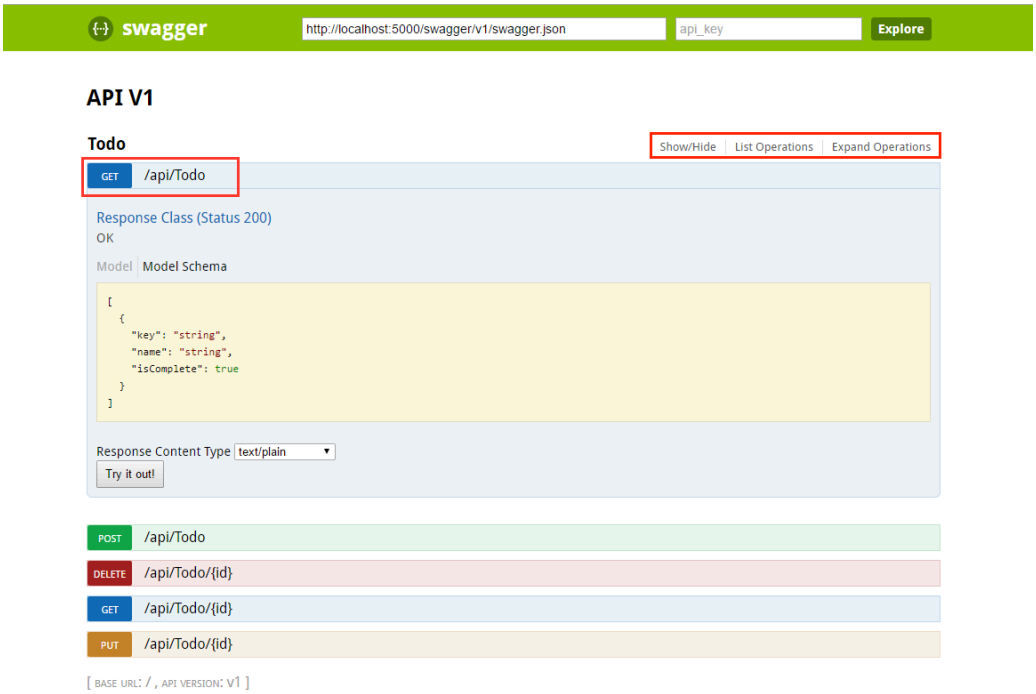

Figure 5: Interface gráfica do Swagger.

Imagine uma equipe de back-end desenvolvendo uma API e outra de front-end precisando consumi-la ao mesmo tempo. Sem uma documentação de API a comunicação entre as equipes seria confusa e cheia de erros. Com o Swagger, a equipe de front-end pode visualizar todos os endpoints, parâmetros e respostas diretamente em uma interface interativa no navegador, além de poder testar as chamadas da API sem escrever código. Isso agiliza o desenvolvimento, evita retrabalho e melhora a colaboração entre times.

3.2. 4 Níveis de Maturidade

Os 4 níveis de maturidade de uma API REST foram propostos por Leonard Richardson e representam um modelo progressivo que mostra o quanto uma API realmente adere aos princípios REST.

Nível	Característica principal
0	Transporte via HTTP
1	Recursos com URIs
2	Uso correto de verbos HTTP
3	HATEOAS

- O **nível 0** é o mais básico, onde a API usa o protocolo HTTP apenas como um meio de transporte, sem aproveitar seus recursos — as requisições geralmente são feitas para uma única URL e toda a lógica é definida no corpo da mensagem, o que se assemelha a chamadas RPC.

[12] What is Swagger? | Swagger Introduction For Beginners

- No **nível 1**, a API começa a utilizar recursos identificáveis por URIs distintas, ou seja, diferentes partes do sistema (como pedidos, clientes, produtos) são acessadas por diferentes URLs, facilitando a organização e estrutura da aplicação.
- O **nível 2** acrescenta o uso correto dos verbos HTTP, como GET, POST, PUT e DELETE, permitindo que a API se beneficie da semântica do próprio protocolo para definir as ações realizadas sobre os recursos.
- Já o **nível 3**, o mais avançado, introduz o conceito de **HATEOAS** (*Hypermedia as the Engine of Application State*), onde a própria resposta da API inclui links que guiam o cliente sobre quais ações ele pode realizar a seguir, sem depender de conhecimento prévio sobre a estrutura da aplicação. Cada nível representa um passo rumo a uma API mais padronizada, intuitiva, reutilizável e fácil de evoluir.

A maioria das APIs hoje está no nível 2: usa URLs organizadas por recursos e os verbos HTTP corretos. Ignoram o nível 3 (HATEOAS), preferindo documentar rotas (swagger) em vez de gerar links dinamicamente.

Isso ocorre pois implementar HATEOAS requer que o servidor inclua links hipermídia dinâmicos em cada resposta, o que aumenta a complexidade do desenvolvimento, documentação e manutenção da API, além de potencialmente elevar o tráfego de dados e os custos de processamento em sistemas de alta demanda. Embora seja parte da definição original de REST, não existe um padrão amplamente aceito para sua implementação, dificultando a integração com ferramentas e bibliotecas populares que não oferecem suporte nativo a hipermídia. Na prática, os clientes costumam conhecer a estrutura da API por meio de documentação, preferindo controlar a navegação por conta própria, tornando HATEOAS redundante ou confuso em muitos casos.

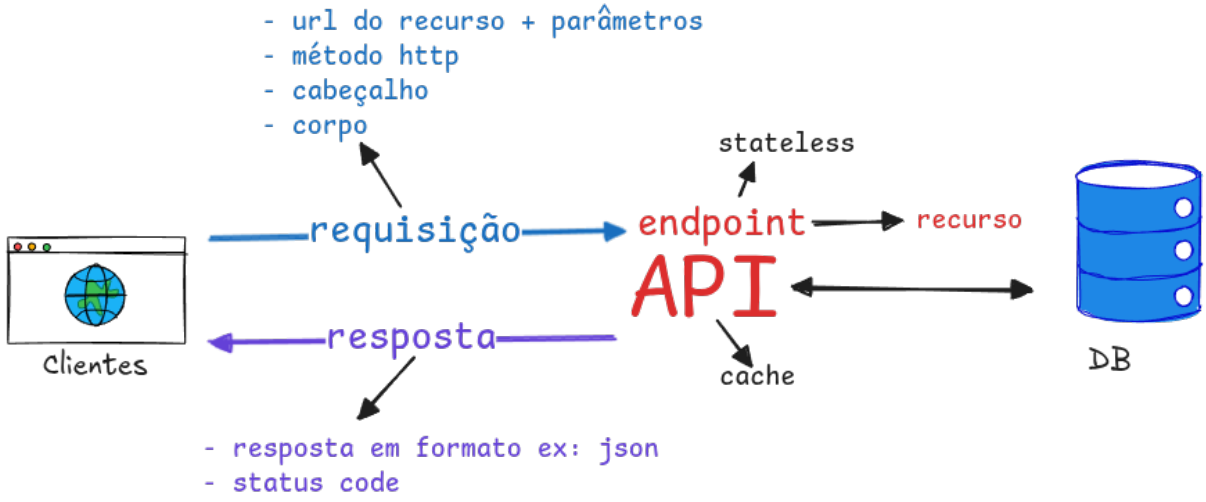

Figure 6: Funcionamento de API RESTful.

Através dessa imagem pode-se dizer que uma API REST "descansa" (*rest* em inglês significa descansar) até ser acionada por um cliente que solicita dados. Em uma arquitetura REST, a API permanece em um estado de espera, pronta para receber requisições HTTP. Quando um cliente faz uma solicitação, a API processa essa requisição e retorna a resposta apropriada. Esse modelo de operação é eficiente, pois a API não consome recursos desnecessariamente enquanto não está em uso, ativando-se apenas quando necessário.

4. Python

Python é uma linguagem de programação amplamente utilizada no desenvolvimento de APIs e microsserviços devido à sua simplicidade, legibilidade e vasto ecossistema de bibliotecas.

Abaixo a explicação dos conceitos é feita em maior parte em códigos. É importante que você faça os exercícios em código para internalizar os conceitos (seja no Google Colab, Jupyter Notebook ou de outra forma).

Referências.[13] |[14]

4.1. Básico

```
# Comentários em uma única linha começam com uma cerquilha (também conhecido por
sustenido).

""" Strings de várias linhas podem ser escritas
    usando três ", e são comumente usadas
    como comentários.
"""

# Matemática é como você espera que seja
1 + 1   # 2
8 - 1   # 7
10 * 2  # 20
3 ** 2 # 9 (exponenciação)
5 // 2 # 2 (divisão inteira)
5 % 2 # 1 (resto)
```

4.1.1. Operadores Relacionais

```
2 > 3 # False
2 < 3 # True
2 >= 2 # True
4 <= 3 # False
3 == 3 # True
4 != 4 # False
```

4.1.2. Operadores Lógicos

```
not True # False
True and False # False
True or False # True
```

4.1.3. Variáveis, `print` e `input`

```
alguma_variavel = "Python"

# Python tem uma função print
print("Eu sou o", alguma_variavel, ", Prazer em conhecer!")
# Eu sou o Python , Prazer em conhecer!

# Forma simples para capturar dados de entrada via console
input_string_var = input("Digite alguma coisa: ") # Retorna o que foi digitado em
uma string
```

[13] Aprenda X em Y Minutos

[14] Fundamentos do Pensamento Algorítmo - Natanale Antonioli

4.1.4. Strings

```
# Strings são criadas com " ou '
"Isto é uma string."
'Isto também é uma string.'

# Strings também podem ser somadas!
"Olá " + "mundo!"  # => "Olá mundo!"
# Strings podem ser somadas sem usar o '+'
"Olá " "mundo!"    # => "Olá mundo!"

# Uma string pode ser manipulada como se fosse uma lista de caracteres
"Isso é uma string"[0]  # => 'I'

# .format pode ser usado para formatar strings, dessa forma:
"{} podem ser {}".format("Strings", "interpoladas")  # => "Strings podem ser
interpoladas"

# Você pode repetir os argumentos para digitar menos.
"Seja ágil {0}, seja rápido {0}, salte sobre o {1} {0}".format("Jack", "castiçal")
# => "Seja ágil Jack, seja rápido Jack, salte sobre o castiçal Jack."

# Você pode usar palavras-chave se quiser contar.
"{nome} quer comer {comida}".format(nome="Beto", comida="lasanha")  # => "Beto quer
comer lasanha"
```

4.1.5. Controle de fluxo

```
some_var = 5

if some_var > 10:
    print(some_var," é absolutamente maior que 10.")
elif some_var < 10:    # Esta cláusula elif é opcional.
    print(some_var," é menor que 10.")
else:                  # Isto também é opcional.
    print(some_var," é, de fato, 10.")
# 5  é menor que 10.
```

4.1.6. Repetição, range

```

# Laços for iteram sobre listas

for animal in ["cachorro", "gato", "rato"]:
    # Você pode usar format() para interpolar strings formatadas
    print("{} é um mamífero".format(animal))

# range(número) retorna um iterável de números de zero até o número escolhido

for i in range(4):
    print(i)

# range(menor, maior)" gera um iterável de números começando pelo menor até o maior

for i in range(4, 8):
    print(i)

# range(menor, maior, passo)" retorna um iterável de números começando pelo menor
número até o maior númeno, pulando de passo em passo. Se o passo não for indicado,
```

```
o valor padrão é um.

for i in range(4, 8, 2):
    print(i)

# Laços while executam até que a condição não seja mais válida (False)

x = 0
while x < 4:
    print(x)
    x += 1  # Maneira mais curta para for x = x + 1

# O break interrompe completamente um laço (for ou while) antes que ele termine
naturalmente.

for numero in range(1, 11):
    if numero == 5:
        break
    print(numero)

# O continue pula a iteração atual e vai para a próxima do laço, sem executar o que
vem depois dele naquela rodada.

for numero in range(1, 6):
    if numero == 3:
        continue
    print(numero)
```

4.1.7. Funções

```
# Use "def" para criar novas funções.
def add(x, y):
    print("x é {} e y é {}".format(x, y))
    return x + y  # Retorne valores com a cláusula return

# Chamando funções com parâmetros
add(5, 6)

'''
x é 5 e y é 6
11
'''
```

4.1.8. Módulos

```
# Você pode importar módulos
import math
print(math.sqrt(16))  # => 4.0

# Você pode importar apenas funções específicas de um módulo
from math import ceil, floor
print(ceil(3.7))   # => 4.0
print(floor(3.7))  # => 3.0

# Você pode importar todas as funções de um módulo para o namespace atual
# Atenção: isso não é recomendado
from math import *
```

```
# Você pode encurtar o nome dos módulos
import math as m
math.sqrt(16) == m.sqrt(16)  # => True

# Módulos python são apenas arquivos python comuns. Você pode escrever os seus, e
importá-los. O nome do módulo é o mesmo nome do arquivo.

# Importando funções do arquivo operacoes.py

# Opção 1: importa tudo do módulo
import operacoes

print(operacoes.somar(5, 3))      # => 8
print(operacoes.subtrair(10, 4))  # => 6

# Opção 2: importa funções específicas
from operacoes import somar

print(somar(7, 2))  # => 9
# subtrair não está disponível aqui, a menos que também seja importada

# Opção 3: importa com apelido
import operacoes as op

print(op.somar(1, 1))  # => 2
```

4.1.9. `Request`

Requisição à API ViaCEP que deve retornar os dados de um cep, como logradouro, cidade, bairro e estado:

`> request.py`

```
import requests

# CEP de exemplo
cep = '02211050'
url = f'https://viacep.com.br/ws/{cep}/json/' # url base

response = requests.get(url) # envia a requisição, salvando resposta em variável

if response.status_code == 200: # verifica se a status code foi 200 (OK).
    dados = response.json() # transforma o JSON da resposta em um dicionário
Python.
    if 'erro' in dados: # A chave 'erro' no JSON indica que o CEP não foi
encontrado.
        print("CEP não encontrado.")
    else:
        print(f"Logradouro: {dados['logradouro']}")
        print(f"Bairro: {dados['bairro']}")
        print(f"Cidade: {dados['localidade']}")
        print(f"Estado: {dados['uf']}")
else:
    print(f"Erro ao consultar o CEP. Código HTTP: {response.status_code}")
```

Execute com `python request.py`

4.2. Estruturas de dados

Em Python, as estruturas de dados são maneiras de organizar e armazenar dados de forma eficiente.

Tipo	Descrição	Mutável?
`List` 4.2.1	Coleção ordenada de elementos que permite duplicatas e é mutável.	Sim
`Tuple` 4.2.2	Coleção ordenada de elementos imutável que permite duplicatas.	Não
`Sets` 4.2.3	Coleção não ordenada de elementos únicos que não permite duplicatas e é mutável.	Sim
`Dict` 4.2.4	Coleção de pares chave-valor, onde as chaves são únicas e os valores podem ser duplicados, e é mutável.	Sim

4.2.1. Listas

Uma lista é uma coleção ordenada de elementos que pode conter itens de diferentes tipos.

- Sintaxe: Criada usando colchetes, por exemplo: `minha_lista = [1, 2, 3, "quatro"]`.
- Mutabilidade: As listas são mutáveis, ou seja, você pode alterar, adicionar ou remover elementos após a criação.
- Duplicatas: Permite elementos duplicados.
- Acesso: Os elementos podem ser acessados por índice.

```
li = []
# Você pode iniciar uma lista com valores
outra_li = [4, 5, 6]

# Adicione conteúdo ao fim da lista com append
li.append(1)    # li agora é [1]
li.append(2)    # li agora é [1, 2]
li.append(4)    # li agora é [1, 2, 4]
li.append(3)    # li agora é [1, 2, 4, 3]
# Remova do final da lista com pop
li.pop()        # => 3 e agora li é [1, 2, 4]
# Vamos colocá-lo lá novamente!
li.append(3)    # li agora é [1, 2, 4, 3] novamente.

# Acesse uma lista da mesma forma que você faz com um array
li[0]   # => 1
# Acessando o último elemento
li[-1]  # => 3

# Acessar além dos limites gera um IndexError
li[4]  # Gera o IndexError

# Você pode acessar vários elementos com a sintaxe de limites
# Inclusivo para o primeiro termo, exclusivo para o segundo
li[1:3]   # => [2, 4]
# Omitindo o final
li[2:]    # => [4, 3]
```

```
# Omitindo o início
li[:3]    # => [1, 2, 4]
# Selecione cada segunda entrada
li[::2]   # => [1, 4]

del li[2]  # li agora é [1, 2, 3]

# Você pode somar listas
# Observação: valores em li e other_li não são modificados.
li + other_li  # => [1, 2, 3, 4, 5, 6]

# Concatene listas com "extend()"
li.extend(other_li)  # Agora li é [1, 2, 3, 4, 5, 6]

# Verifique se algo existe na lista com "in"
1 in li  # => True

# Examine  tamanho com "len()"
len(li)  # => 6
```

4.2.2. Tuplas

Uma tupla é semelhante a uma lista, mas é imutável.

- Sintaxe: Criada usando parênteses, por exemplo: `minha_tupla = (1, 2, 3, "quatro")`.
- Mutabilidade: As tuplas são imutáveis, ou seja, não podem ser alteradas após a criação.
- Duplicatas: Permite elementos duplicados.
- Acesso: Os elementos podem ser acessados por índice.

```
# Tuplas são como l istas, mas imutáveis.
tup = (1, 2, 3)
tup[0]      # => 1
tup[0] = 3  # Gera um TypeError

# Observe que uma tupla de tamanho um precisa ter uma vírgula depois do
# último elemento mas tuplas de outros tamanhos, mesmo vazias, não precisa,.
type((1))   # => <class 'int'>
type((1,))  # => <class 'tuple'>
type(())    # => <class 'tuple'>

# Você pode realizar com tuplas a maior parte das operações que faz com listas
len(tup)         # => 3
tup + (4, 5, 6)  # => (1, 2, 3, 4, 5, 6)
tup[:2]          # => (1, 2)
2 in tup         # => True

# Você pode desmembrar tuplas (ou listas) em variáveis.
a, b, c = (1, 2, 3)  # a é 1, b é 2 e c é 3
# Por padrão, tuplas são criadas se você não coloca parêntesis.
d, e, f = 4, 5, 6
```

4.2.3. Sets

Um conjunto é uma coleção não ordenada de elementos únicos.

- Sintaxe: Criado usando chaves, por exemplo: `meu_conjunto = {1, 2, 3, 4}`.
- Mutabilidade: Os conjuntos são mutáveis, mas não podem conter elementos duplicados.
- Duplicatas: Não permite elementos duplicados.

- Acesso: Não suporta acesso por índice, pois não é ordenado.

```
# Armazenamento em sets... bem, são conjuntos
empty_set = set()
# Inicializa um set com alguns valores. Sim, ele parece um dicionário. Desculpe.
some_set = {1, 1, 2, 2, 3, 4}  # some_set agora é {1, 2, 3, 4}

# Inclua mais um item no set
filled_set.add(5)  # filled_set agora é {1, 2, 3, 4, 5}

# Faça interseção de conjuntos com &
other_set = {3, 4, 5, 6}
filled_set & other_set  # => {3, 4, 5}

# Faça união de conjuntos com |
filled_set | other_set  # => {1, 2, 3, 4, 5, 6}

# Faça a diferença entre conjuntos com -
{1, 2, 3, 4} - {2, 3, 5}  # => {1, 4}

# Verifique a existência em um conjunto com in
2 in filled_set   # => True
10 in filled_set  # => False
```

4.2.4. Dicionários

Um dicionário é uma coleção de pares chave-valor.

- Sintaxe: Criado usando chaves, por exemplo: `meu_dict = {"nome": "João", "idade": 30}`.
- Mutabilidade: Os dicionários são mutáveis, permitindo a adição, remoção e alteração de pares chave-valor.
- Duplicatas: As chaves devem ser únicas, mas os valores podem ser duplicados.
- Acesso: Os valores são acessados por suas chaves, não por índice.

```
# Dicionários armazenam mapeamentos
empty_dict = {}
# Aqui está um dicionário preenchido na definição da referência
filled_dict = {"um": 1, "dois": 2, "três": 3}

# Observe que chaves para dicionários devem ser tipos imutáveis. Isto é para
# assegurar que a chave pode ser convertida para uma valor hash constante para
# buscas rápidas.
# Tipos imutáveis incluem inteiros, flotas, strings e tuplas.
invalid_dict = {[1,2,3]: "123"}  # => Gera um TypeError: unhashable type: 'list'
valid_dict = {(1,2,3):[1,2,3]}   # Já os valores, podem ser de qualquer tipo.

# Acesse valores com []
filled_dict["um"]  # => 1

# Acesse todas as chaves como um iterável com "keys()". É necessário encapsular
# a chamada com um list() para transformá-las em uma lista. Falaremos sobre isso
# mais adiante. Observe que a ordem de uma chave de dicionário não é garantida.
# Por isso, os resultados aqui apresentados podem não ser exatamente como os
# aqui apresentados.
list(filled_dict.keys())  # => ["três", "dois", "um"]
```

```python
# Acesse todos os valores de um iterável com "values()". Novamente, é
# necessário encapsular ele com list() para não termos um iterável, e sim os
# valores. Observe que, como foi dito acima, a ordem dos elementos não é
# garantida.
list(filled_dict.values())  # => [3, 2, 1]

# Verifique a existência de chaves em um dicionário com "in"
"um" in filled_dict  # => True
1 in filled_dict     # => False

# Acessar uma chave inexistente gera um KeyError
filled_dict["quatro"]  # KeyError

# Use o método "get()" para evitar um KeyError
filled_dict.get("um")      # => 1
filled_dict.get("quatro")     # => None
# O método get permite um parâmetro padrão para quando não existir a chave
filled_dict.get("um", 4)   # => 1
filled_dict.get("quatro", 4)  # => 4

# "setdefault()" insere em dicionário apenas se a dada chave não existir
filled_dict.setdefault("cinco", 5)  # filled_dict["cinco"] tem valor 5
filled_dict.setdefault("cinco", 6)  # filled_dict["cinco"] continua 5

# Inserindo em um dicionário
filled_dict.update({"quatro":4})  # => {"um": 1, "dois": 2, "três": 3, "quatro": 4}
#filled_dict["quatro"] = 4        #outra forma de inserir em um dicionário

# Remova chaves de um dicionário com del
del filled_dict["um"]  # Remove a chave "um" de filled_dicts
```

4.3. POO

POO ou programação orientada a objetos, é um paradigma de programação que organiza o código imitando o mundo real. Em vez de escrever funções isoladas e manipular variáveis soltas, você modela classes (moldes) que descrevem objetos (instâncias desses moldes) com atributos (dados) e métodos (comportamentos).[15] |[16]

As vantagens principais são:

- **Encapsulamento**: protege dados internos através de modificadores como public, protected e private;
- **Herança**: permite que classes compartilhem lógica comum;
- **Polimorfismo**: possibilita que classes tenham métodos com o mesmo nome, mas comportamentos diferentes dependendo do contexto
- **Abstração**: permite esconder detalhes complexos e mostrar apenas o que é essencial, facilitando o uso e a manutenção do código, permitindo controlar o acesso aos atributos, escondendo detalhes internos e protegendo os dados.

```python
class ContaBancaria:
    def __init__(self, titular, saldo=0):
        self.titular = titular
```

[15] Programação Orientada a Objetos | Explicação Simples

[16] Como Funcionam Classes e Programação Orientada a Objetos em Python - Aprenda em 10 Minutos!

```
        self.saldo = saldo

    def depositar(self, valor):
        if valor > 0:
            self.saldo += valor
            print(f"Depósito de R${valor} feito por {self.titular}.")
        else:
            print("Valor de depósito inválido.")

    def sacar(self, valor):
        if valor <= self.saldo:
            self.saldo -= valor
            print(f"Saque de R${valor} realizado por {self.titular}.")
        else:
            print(f"Saldo insuficiente para {self.titular}.")

    def exibir_saldo(self):
        print(f"{self.titular} tem R${self.saldo} na conta.")


# Criando objetos (contas)
conta1 = ContaBancaria("Alice", 1000)
conta2 = ContaBancaria("Bob", 500)

# Operações
conta1.depositar(200)
conta1.sacar(150)
conta1.exibir_saldo()

conta2.sacar(700)
conta2.exibir_saldo()
```

5. Flask

Flask é um microframework para desenvolvimento web em Python, criado por Armin Ronacher em 2010. Ele é leve e flexível, permitindo que os desenvolvedores construam aplicações web de forma rápida, sem impor uma estrutura rígida. É indicado para criar APIs, aplicações web simples e protótipos, sendo especialmente útil em projetos que não requerem a complexidade de frameworks maiores, como Django. Sua extensibilidade permite adicionar funcionalidades conforme necessário, tornando-o uma escolha popular para desenvolvedores que buscam simplicidade e personalização.

Framework é uma estrutura de software que fornece um conjunto de ferramentas, bibliotecas e convenções para facilitar o desenvolvimento de aplicações. Ele estabelece uma base imposta sobre a qual os desenvolvedores podem construir, tendo funcionalidades pré-definidas. Exemplo: Django.

Micro-Framework é uma versão mais leve e simplificada de um framework tradicional. Ele oferece funcionalidades básicas e essenciais, permitindo que escolham de bibliotecas adicionais conforme necessário. Isso faz com que que os desenvolvedores criem aplicações de forma rápida e com menos complexidade, sendo ideais para projetos menores ou para desenvolvedores que desejam mais controle sobre a estrutura da aplicação, sem as imposições de um framework completo.

5.1. Python Virtual Environments

Python Virtual Environments ou Ambientes Virtuais Python são ferramentas que permitem criar ambientes isolados para projetos em Python. Isso significa que cada projeto pode ter suas próprias versões de pacotes e dependências, sem interferir em outros projetos ou no Python instalado globalmente no sistema.[17]

Esses ambientes são úteis para evitar conflitos entre bibliotecas, garantir que um projeto funcione com versões específicas e manter o sistema organizado. Sem ambientes virtuais, haveria conflito de versões de bibliotecas, bagunça no sistema, e dificuldades para colaborar ou fazer deploy. Com eles, cada projeto funciona em uma "bolha" independente, tornando o desenvolvimento mais estável e profissional.

Os ambientes virtuais funcionam dentro da pasta onde foram criados, e para usá-los, normalmente você precisa estar dentro da pasta do projeto. Uma vez ativado, o terminal entra naquele ambiente, e você pode instalar ou rodar tudo isoladamente.

PIP ou *Pip Installs Packages* é o gerenciador de pacotes oficial do Python, usado para instalar, atualizar e remover bibliotecas e módulos disponíveis no repositório PyPI (Python Package Index) com um simples comando como por exemplo `pip install nome-do-pacote`.

```
# Criar ambiente virtual
python -m venv nome_do_ambiente #OBS: será criada pasta dentro do projeto com oo
nome do ambiente que escolher

# ativar o ambiente OBS: SEMPRE QUE REABRIR O PROJETO PRECISARÁ ATIVAR ESSE
AMBIENTE

# em linux
source nome_do_ambiente/bin/activate
#ou
```

[17] Python Virtual Environments - Full Tutorial for Beginners

```
.\nome_do_ambiente\bin\Activate

# ativar o ambiente em windows (CMD)
.\nome_do_ambiente\Scripts\activate

# Com o ambiente ativado, use:
pip install nome_do_pacote

# desativa o ambiente
deactivate

# para excluir o ambiente apenas delete a pasta do ambiente virtual
rm -rf nome_do_ambiente # linux
rmdir /s /q nome_do_ambiente # windows

# Para instalar o flask dentro do ambiente python ativo:
pip install flask
```

5.2. `requirements.txt`

O **`requirements.txt`** é um arquivo de texto que lista todas as dependências (bibliotecas) necessárias para um projeto Python funcionar corretamente, geralmente com as versões específicas usadas. Ele é essencial para compartilhar o ambiente do projeto com outras pessoas ou para fazer deploy em servidores, garantindo que todos usem os mesmos pacotes. O comando `pip freeze > requirements.txt` cria esse arquivo.

Quando o arquivo requirements.txt já existe e você executa o comando `pip freeze > requirements.txt`, o conteúdo anterior do arquivo é sobrescrito. Isso significa que a lista de bibliotecas e suas versões que estava no arquivo será substituída pela nova lista gerada pelo comando pip freeze, refletindo o estado atual das bibliotecas instaladas no ambiente Python.

Ao mover o projeto para um novo ambiente, com o ambiente virtual ativado, use o seguinte comando para instalar as dependências listadas no requirements.txt: `pip install -r requirements.txt`

5.3. API básica

Criação de endpoint (conforme teoria na seção 3). Abaixo há um exemplo de uma API básica usando o framework Flask em Python:

```
> app.py

from flask import Flask, jsonify # Importa o Flask e a função jsonify, que converte
dicionários Python em JSON (formato usado em APIs).
app = Flask(__name__) # O argumento __name__ indica ao Flask onde está o ponto de
entrada da aplicação.

@app.route('/users', methods=['GET']) # Define uma endpoint da aplicação: Quando o
navegador ou cliente acessar o caminho /users usando o método HTTP GET, a função
getUsers será chamada.

def getUsers(): # Função que será executada quando a rota /users for acessada com
GET
    dados = { 'mensagem': 'OLA MUNDO!'} # Cria um dicionário dados com uma
mensagem.
    return jsonify(dados) # Usa jsonify para transformar esse dicionário em uma
resposta JSON, que será enviada para o cliente.
```

```
if __name__ == '__main__': # Verifica se o arquivo está sendo executado diretamente
(e não importado)
    app.run(debug=True) #  inicia o servidor Flask em modo debug: Isso permite ver
erros mais detalhados no navegador. O servidor reinicia automaticamente se o código
for alterado.
```

Ao executar `python app.py` esse código entra em execução. Acesse a url informada em execução: `Running on http://127.0.0.1:5000` ou algo parecido e adicione `/users` no final, ficando `http://127.0.0.1:5000/users`. Ao fazer isso a mensagem `OLA MUNDO!` é exibida em json.

O endereço `http://127.0.0.1:5000/` retorna "Not Found" porque sua aplicação Flask só define a rota `/users`, e não há nenhuma rota associada à raiz (`/`). Para que algo apareça ali, seria necessário adicionar uma rota específica para `/` no código.

Código sem comentários, incluindo a rota raiz (`/`), retornando uma mensagem genérica:

`> app.py`

```
from flask import Flask, jsonify
app = Flask(__name__)

@app.route('/users', methods=['GET'])

def getUsers():
    dados = { 'mensagem': 'OLA MUNDO!'}
    return jsonify(dados)

@app.route('/', methods=['GET'])
def getIndex():
    dados = {'msg': 'Bem vindo ao Index'}
    return jsonify(dados)

if __name__ == '__main__':
    app.run(debug=True)
```

Ferramentas como o **Postman** e similares (Insomnia, Ply e etc) são utilizada para testar e desenvolver APIs, permitindo que desenvolvedores enviem requisições HTTP (incluindo POST com dados no body) e analisem as respostas para verificar o funcionamento correto da API.

Utilizando a extensão Ply no VsCode (identificação: `ply-ct.vscode-ply`) podemos testar essa API criada. Depois de instalar a extensão, vá em **New Request**, salve o arquivo `.ply` em algum lugar, logo será aberta nova aba conforme abaixo, inclua a **url**, escolha o **método** GET e teste os endpoints criados `/` e `/users`.

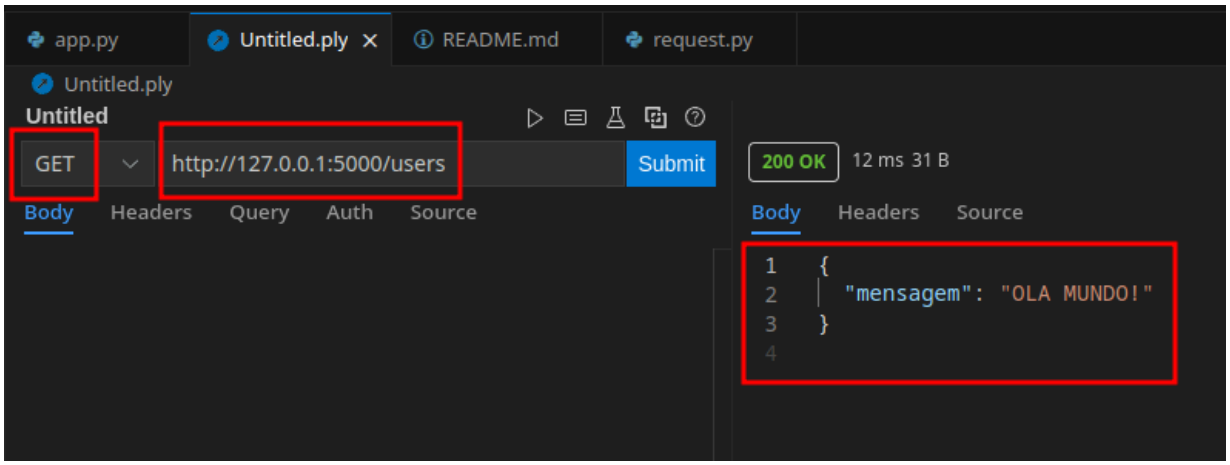

Figure 7: Teste de endpoints da API

Através dessa demonstração vemos que a API permanece em um estado de espera, pronta para receber requisições HTTP. Quando um cliente faz uma solicitação, a API processa essa requisição e retorna a resposta apropriada. Esse modelo de operação é eficiente, pois a API não consome recursos desnecessariamente enquanto não está em uso, ativando-se apenas quando necessário.

5.4. Aprofundando

Vamos iniciar um projeto do zero, abaixo os comandos que deve executar:

```
python -m venv env

source env/bin/activate # se em linux
.\env\bin\Activate # ou em Windows

pip freeze > requirements.txt
pip install flask
pip install flask-restful
pip install flask_sqlalchemy
# se em windows substitua `touch` por `type nul > nome arquivo`
touch .gitignore # inclua env dentro
touch api.py
touch create_db.py
```

Detalhando as dependências instaladas:

O Flask-RESTful é uma extensão do Flask que simplifica a criação de APIs RESTful em Python, permitindo definir recursos como classes, facilitar a serialização de dados e gerenciar erros de forma eficiente.

ORM (*Object-Relational Mapping*) é uma técnica que permite interagir com bancos de dados relacionais usando objetos da linguagem de programação, evitando a necessidade de escrever SQL diretamente para Python. Ele facilita a manipulação de bancos de dados, permitindo que desenvolvedores definam modelos de dados como classes e realizem operações de consulta e persistência de forma intuitiva e eficiente.

O SQLAlchemy é uma biblioteca Python que implementa esse conceito, permitindo que desenvolvedores definam tabelas como classes e colunas como atributos, facilitando a manipulação de dados com comandos em Python. Com ele, é possível criar, consultar, atualizar e deletar registros de forma mais intuitiva e integrada ao código, mantendo a estrutura e regras do banco relacional.

O Flask-SQLAlchemy é uma extensão do Flask que simplifica a integração do SQLAlchemy.

Abaixo, o código define uma API RESTful usando Flask, Flask-RESTful e SQLAlchemy para gerenciar usuários armazenados em um banco de dados SQLite[18]. Ele permite criar, listar, buscar, atualizar e excluir usuários por meio das rotas `/api/users/` e `/api/users/<id>`. A estrutura inclui validação de entrada com `reqparse`, formatação das respostas com `marshal_with` e mapeamento do modelo `UserModel` para a tabela do banco.

> api.py

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_restful import Resource, Api, reqparse, fields, marshal_with, abort

# Criação da aplicação Flask
app = Flask(__name__)

# Configuração do banco SQLite
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///database.db'
db = SQLAlchemy(app)

# Criação da API RESTful
api = Api(app)

# Modelo de usuário para o banco de dados
class UserModel(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    email = db.Column(db.String(80), unique=True, nullable=False)

    def __repr__(delf):
        return f"User(name = {self.username}, email = {self.email})"

# Validação dos dados recebidos nas requisições
user_args = reqparse.RequestParser()
user_args.add_argument('username', type=str, required=True, help="Username não pode
estar em branco")
user_args.add_argument('email', type=str, required=True, help="Email não pode estar
em branco")

# Especificação de como os dados do usuário devem ser serializados na resposta
userFields = {
    'id': fields.Integer,
    'username': fields.String,
    'email': fields.String
}

class Users(Resource):
    # Retorna a lista de todos os usuários cadastrados
    @marshal_with(userFields)
    def get(self):
```

[18]SQLite é um sistema de banco de dados relacional leve que armazena os dados em um único arquivo local, sem precisar de um servidor dedicado. Ele interpreta comandos SQL e gerencia estruturas como tabelas e índices diretamente nesse arquivo, sendo ideal para aplicações pequenas, testes, protótipos e apps móveis.

```
        return UserModel.query.all()

    # Cria um novo usuário com os dados fornecidos e retorna todos os usuários
    @marshal_with(userFields)
    def post(self):
        args = user_args.parse_args()
        user = UserModel(username=args["username"], email=args["email"])
        db.session.add(user)
        db.session.commit()
        return UserModel.query.all(), 201

class User(Resource):
    # Retorna os dados de um único usuário com base no ID
    @marshal_with(userFields)
    def get(self, id):
        user = UserModel.query.filter_by(id=id).first()
        if not user:
            abort(404, "Usuário não encontrado")
        return user

    # Atualiza os dados (username e email) de um usuário existente
    @marshal_with(userFields)
    def patch(self, id):
        args = user_args.parse_args()
        user = UserModel.query.filter_by(id=id).first()
        if not user:
            abort(404, "Usuário não encontrado")
        user.username = args["username"]
        user.email = args["email"]
        db.session.commit()
        return UserModel.query.all(), 204

    # Exclui um usuário com base no ID fornecido
    @marshal_with(userFields)
    def delete(self, id):
        user = UserModel.query.filter_by(id=id).first()
        if not user:
            abort(404, "Usuário não encontrado")
        db.session.delete(user)
        db.session.commit()
        return user


# Registro dos recursos na API com suas rotas
api.add_resource(Users, '/api/users/')
api.add_resource(User, '/api/users/<int:id>')

# Execução da aplicação Flask em modo debug
if __name__ == '__main__':
    app.run(debug=True)
```

> create_db.py

```
from api import app, db
```

```
with app.app_context():
    db.create_all()
```

O código acima (`create_db.py`) é usado para inicializar o banco de dados com as tabelas definidas nos modelos. Ao executar o comando `python create_db.py`, será criado o banco de dados no arquivo `instance/database.db`

Em seguida, ao executar `python api.py`, o Flask inicia o servidor web local disponibilizando os endpoints da API para requisições HTTP. A partir daí você deve usar a extensão Ply para testar sua API:

1. Crie usuários
2. Liste todos os usuários cadastrados
3. Retorne os dados de um único usuário
4. Atualize os dados de um usuário
5. Exclua um usuário