APIs para iniciantes: HTTP, Requests e APIs com Python

Asimov Academy

ASIMOV

Conteúdo

01. Introdução ao curso - APIs para Iniciantes

Bem-vindos ao curso “APIs para iniciantes: HTTP, Requests e APIs com Python” da Asimov Academy!

Antes de mais nada, você deve estar se perguntando: “O que é uma API?”

Não se preocupe, vamos **começar do zero e aprender juntos** o que são APIs, como funcionam e como você pode utilizá-las para criar projetos incríveis.

Para quê usar APIs?

As APIs estão por toda parte. Elas permitem que **diferentes aplicativos se comuniquem** entre si. Quando você usa um aplicativo de clima para ver a previsão do tempo ou quando você se conecta ao seu site favorito usando uma conta de rede social, você está usando uma API!

Vamos dar uma olhada rápida em um dos web apps práticos que vamos construir ao longo do curso. Por exemplo, vamos criar um web app que mostra a popularidade dos nomes no Brasil usando dados do IBGE. E você consegue acessá-lo até mesmo do celular!

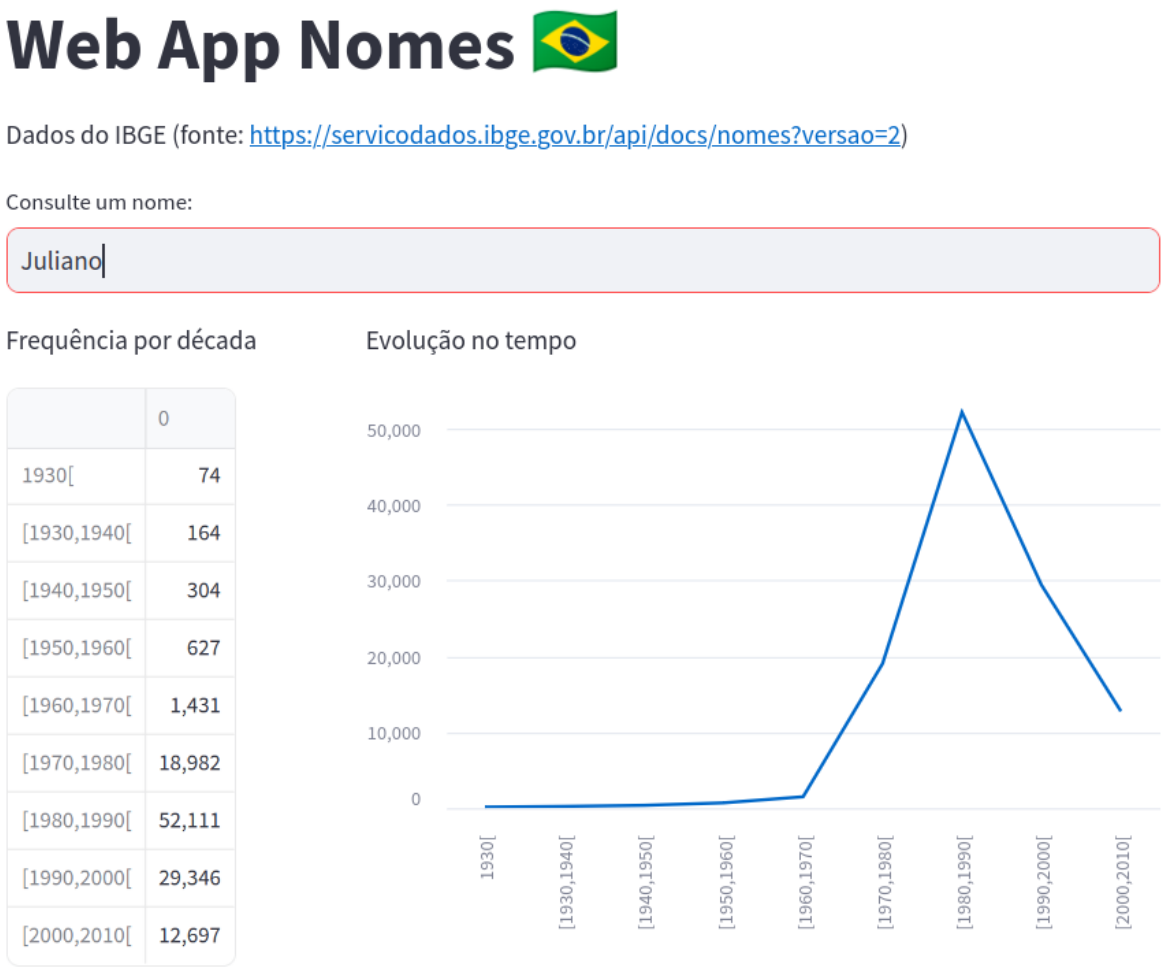

Figure 1: Web App de nomes do IBGE.

Pra quem é este curso?

Este curso foi planejado justamente olhando para o público leigo. Muito do conteúdo de APIs existente hoje em dia pela internet funciona muito bem… se você já for desenvolvedor!

Nosso objetivo com este curso é explicar APIs e todos os conceitos relacionados (HTTP, Requisições, autenticação, interfaces, REST) de uma forma simples e prática. É como eu gostaria de ter aprendido sobre APIs quando eu sabia muito pouco de programação.

O que preciso saber?

Você não precisa de nenhum conhecimento específico além do básico de Python. Portanto, tendo feito o nosso curso de Python Básico e o curso de Setup de programação Python, você já está apto a acompanhar este conteúdo.

Para os miniprojetos, usaremos outras bibliotecas, com destaque para o Streamlit na criação de interfaces. Temos também um curso de Streamlit na plataforma, além de diversos projetos, mas não se preocupe: passaremos por tudo que for necessário também aqui no curso!

O que vamos aprender neste curso

Neste curso, vamos abordar tópicos fundamentais de uma **maneira prática e fácil de entender**. Vamos falar sobre:

- O que é HTTP e por que ele é importante.
- Como fazer seu primeiro request usando Python.
- Como entender e analisar as respostas desses requests.
- O que são códigos de status HTTP e como interpretá-los.

Vamos também explorar o conceito de **interfaces e APIs**, com um foco especial nas **APIs REST**, que são as mais populares hoje em dia.

Miniprojetos

Além das aulas teóricas, você colocará a mão na massa com **miniprojetos práticos**. Ao longo das aulas, criaremos:

- Um web app de consulta de nomes, conectado à API do IBGE.
- Um web app que mostra a previsão do tempo usando a API do OpenWeather.
- Um web app de popularidade de músicas, conectado à API do Spotify.

Esses projetos foram cuidadosamente formulados para treinar os conceitos abordados em aula, como os diferentes tipos de requisições, o uso de parâmetros nas requisições, e as diferentes formas de autenticar em uma API. Não se preocupe - você aprenderá tudo aos poucos a cada aula, para então colocar em prática!

Não importa se você é novo em programação ou se já tem alguma experiência, este curso foi feito para ser **acessível e útil para todos que querem entender o que, afinal de contas, é uma API**.

02. HTTP - o protocolo da Internet

As APIs são todas construídas no topo do HTTP. Por isso, antes de nos aprofundarmos nas APIs em si, precisamos entender o que é o HTTP.

O que é HTTP?

O HTTP (*HyperText Transfer Protocol*) é uma **forma padronizada de trocar informações pela da Internet**. Quando falamos de "informação", estamos falando de tudo: texto, imagens, vídeos, buscas em bancos de dados…

O HTTP foi desenvolvido entre 1989 a 1991, pelo cientista britânico Tim Berners-Lee, que criou o primeiro "passo a passo" para colocar um servidor rodando. Claro que, naquela época, a troca de informações era muito mais simples.

Elementos básicos do HTTP

A primeira versão de HTTP 0.9 tinha os seguintes conceitos:

- **HTML** (*HyperText Markup Language*): formato padronizado de documento que permite o uso de links ("hipertexto").
- **HTTP** (*HyperText Transfer Protocol*): uma forma padronizada de enviar HTML de um computador a outro.
- **Cliente**: um programa capaz de solicitar um arquivo HTML através de HTTP e exibir seu conteúdo.
  - A solicitação do arquivo é chamada de **requisição** ou *request*.
  - Os clientes de HTTP evoluíram para os **navegadores web** (*web browser*) atuais.
- **Servidor**: um computador capaz de retornar um documento HTML para o cliente a partir de uma requisição HTTP feita por ele.
  - O envio do documento HTML para o cliente é chamado de **resposta** (*response*) do request.

É basicamente isso! O diagrama abaixo exemplifica como este fluxo ocorre conforme o diagrama abaixo, com o cliente fazendo uma requisição e o servidor retornando um documento HTML como resposta:

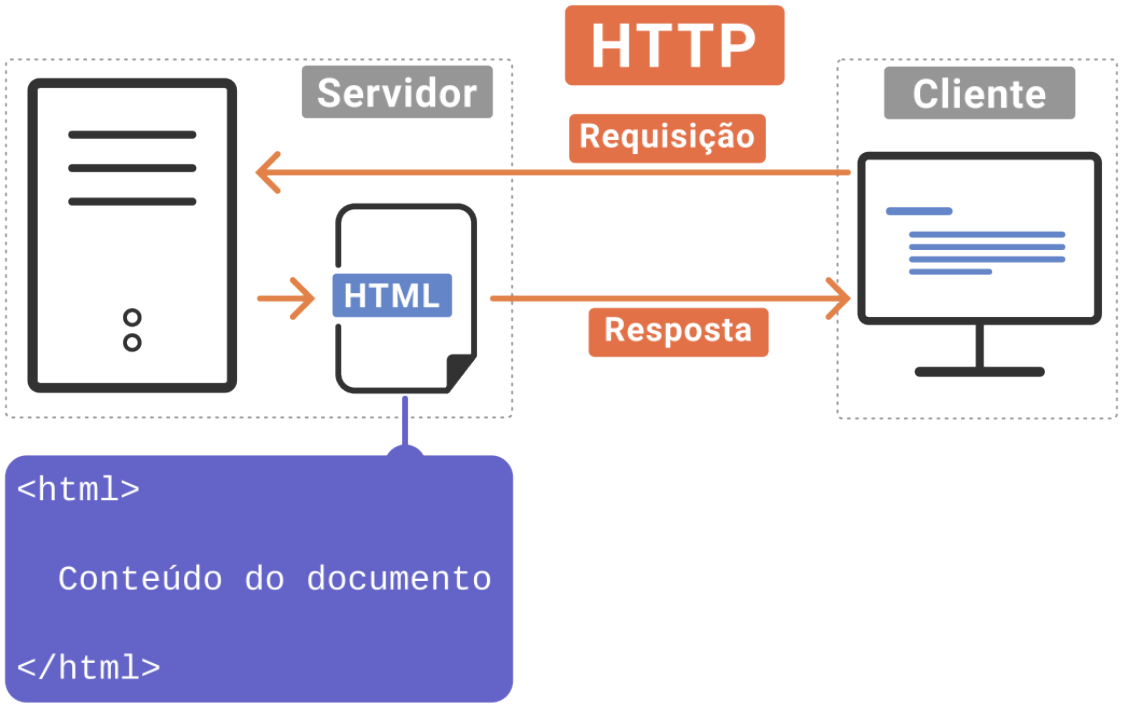

Figure 2: Estrutura básica de uma requisição HTTP

O que é o "Hipertexto"?

"Hipertexto" é simplesmente um texto contendo (hiper)links a outros (hiper)textos. Este conceito já existia antes da web, mas estava restrito a dados locais. A grande "sacada" do HTTP foi utilizar HTML (um tipo de hipertexto) como forma padrão de comunicação, pois isso permite uma navegação muito mais fluida.

Contextualizando: hoje em dia, clicar em um link da internet é algo arbitrário. Mas nos anos 90, a ideia de estruturar **documentos que permitem links para outros documentos** foi revolucionária!

Não preciso ficar digitando exatamente o endereço de cada documento que quero acessar, posso simplesmente clicar no link relacionado.

O “loop” do HTTP

A ideia é que, a partir da requisição inicial feita pelo cliente, o servidor responde com o documento X. Neste documento, há links para o documento Y e Z. O usuário então seleciona o documento Z a partir do link, e é feita uma nova requisição para o documento Z, e assim por diante.

Em última análise, esta é a forma básica com que qualquer site moderno funciona. Alguns deixam isso mais claro (ex: clicar em links na Wikipedia), mas em geral não pensamos tanto nisso enquanto navegamos na web. Isso é um indício do quão brilhante é essa forma de estruturação e transferência de arquivos: nem paramos para pensar nela de tão “natural” que parece ser!

A evolução do HTTP

Com o tempo, foi necessário evoluir o HTTP para diversas funcionalidades:

Padronização

Se cada servidor ou cliente implementar suas próprias regras, se torna difícil compartilhar arquivos globalmente, de forma unificada. Portanto, regras padronizadas foram sendo criadas com o tempo. As primeiras surgiram em 1997, com a versão 1.1 do HTTP.

Códigos de status

Se a requisição falhar, qual o motivo? Fiz algo de errado na requisição, o servidor está com bug, o arquivo não existe, ou a conexão está falhando? Os **códigos de status** foram criados justamente para esclarecer este tipo de dúvida.

Extensão

O HTTP faz muito mais coisas hoje em dia que meramente enviar um HTML. Essas funcionalidades foram criadas ao longo do tempo, conforme foi sendo necessário:

- E se eu quiser enviar outros tipos de arquivo, como imagens, vídeos ou códigos?
- E se ao invés de baixar um arquivo, eu quiser realizar uma ação, como atualizar um banco de dados ou fazer upload de algum arquivo?
- E se eu quiser indicar qual o sistema operacional ou a língua local de quem está fazendo a requisição? Todas estas funcionalidades (e muitas outras) estão implementadas dentro de uma requisição HTTP moderna.

Autenticação

E se eu só quiser compartilhar certos arquivos com certas pessoas? Foi preciso desenvolver uma camada de autenticação e acesso dentro do HTTP.

Otimização

O HTTP passou por diversos estágios de otimização, como:

- **Cacheamento**: dados são mantidos na memória do servidor para requisições idênticas, acelerando a resposta. Dados também podem ser enviados para cacheamento no computador do cliente.
- **Paralelização**: requisições podem rodar em paralelo. A requisição inicial dispara diversas outras que carregam os componentes de vídeo, imagens, banco de dados, ...
- **Compressão**: arquivos são comprimidos antes de serem enviados, para diminuir o tempo de resposta.

Segurança

Com a autenticação e o uso mais "sério" da web (dados bancários, dados pessoais), surgem também a necessidade de tornar as conexões mais seguras. Isso começou em 1994, com o SSL, e hoje evoluiu para o TLS (apesar de o nome mais comum ainda ser referenciado como "Certificado de SSL").

Não nos aprofundaremos na camada de TLS/SSL neste curso. Pense nela como algo que fica "envolta" ao redor de um Request, protegendo-o através de criptografia.

Com essa camada, o HTTP passou a se chamar HTTPS - S de "seguro" (*secure*). Atualmente, navegadores web bloqueiam chamadas HTTP que não sejam HTTPS por padrão (é preciso desabilitar para mostrar que você "aceita o risco").

03. Nosso primeiro Request

Antes de nos aprofundarmos mais nos conceitos do HTTP, vamos criar um Request em Python e entender o que acontece.

Criando o Request

Vamos usar a biblioteca `requests`, que deve ser instalada com `pip`:

```
pip install requests
```

Agora, rode o código abaixo:

```
import requests

url = "https://www.google.com"

resposta = requests.get(url)

print(resposta, end="\n\n\n\n\n")
print(resposta.text)

with open('pagina_google.html', 'w') as arquivo:
    arquivo.write(resposta.text)
```

O resultado (tanto no terminal quanto no arquivo `pagina_google.html`) é uma "mistura moderna" de HTML, CSS e Javascript. Mesmo assim, a ideia básica do HTTP está presente:

- O cliente (nosso código) faz um request de um conteúdo em HTML.
- O servidor (nesse caso, do Google) organiza o conteúdo do HTML requisitado e o retorna em uma resposta.
- A resposta e seu conteúdo são então exibidos no script (`Response 200` = requisição deu certo!)

Explorando o resultado

O Python não possui um "navegador web" embutido, portanto nosso código ficou limitado a printar o conteúdo do HTML e escrevê-lo para dentro de um arquivo.

Podemos pegar este arquivo e abrir com 2 programas diferentes:

- Um editor de texto, para ver o **conteúdo bruto** do HTML (o mesmo que foi exibido pelo `print()` em Python).
- Um navegador web, para ver **a renderização** do HTML.

Veja que um navegador web é capaz de renderizar um documento HTML local ou da Internet - na verdade, todo documento HTML é “local”, já que para renderizar o HTML ao clicarmos em um link, nosso navegador web precisa baixá-lo primeiro.

Mas o Google parece diferente

Se você abrir o arquivo `pagina_google.html` em um navegador, vai ver que a cara da página de busca do Google parece um tanto “quebrada”:

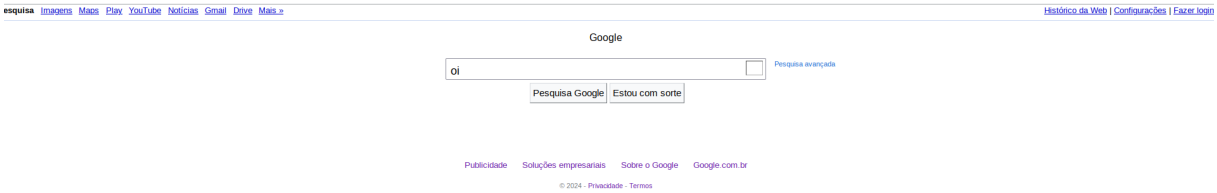

Figure 3: Arquivo `pagina_google.html` aberta em um navegador

Isso acontece porque parte da funcionalidade (imagens, estilização, funções de busca) é carregada de forma dinâmica ao acessarmos a página. Este é um conceito recente, mas muito comum na web moderna, pois agiliza o carregamento de página (lembre-se dos Requests em paralelo que surgiram nas com a evolução do HTTP).

Inspecionando o Request pelo navegador

Em qualquer navegador comum, podemos:

- Clicar com o botão direito e acessar a opção “Ver código-fonte da página” para ver o HTML “bruto”.
- Acessar as ferramentas de desenvolvedor (F12, ou clique com botão direito em “Inspecionar”)
- Acessar o menu “Inspetor” para testar a alteração de algum valor do HTML da página.
- Acessar o menu “Rede” para ver os requests acontecendo Note que o request inicial do Google dispara muitos outros! Alguns são para baixar imagens, ícones, e estilização, outros são os infames requests de coleta de dados. Inclusive é possível ver estes requests sendo bloqueados, se você usar algum tipo de programa ou extensão com maior controle de privacidade.

E quanto à URL?

URL significa **Localizador de Recurso Uniforme** (*Uniform Resource Locator*), e é uma forma de especificar qual recurso (página HTML) eu quero acessar.

Existe uma inteligência muito grande que "mapeia" uma certa URL para um certo servidor (passando desde o seu roteador até o seu provedor de internet e além), mas não iremos abordar isto neste curso.

04. Anatomia de um Request

Nesta aula, vamos entender o que de fato existe “dentro” de um request.

Métodos (Verbos) de HTTP

Na aula passada, usamos a função `requests.get()` para baixar os dados. Esta função não tem seu nome por acaso: ela realiza uma requisição de HTTP usando o método GET.

Os **métodos de HTTP** (também conhecidos por **verbos de HTTP**) representam alguma **ação a ser desempenhada** a partir do request original. Como acabamos de ver, o método GET é usado para **pegar algum dado**.

Principais métodos de HTTP

A tabela abaixo demonstra os principais métodos de HTTP, junto de sua função. Estes métodos possuem analogia direta com um CRUD de um banco de dados:

Método	Descrição	Equivalente em BD (CRUD)
GET	Pegar um dado	Ler (**R**ead)
POST	Criar um dado novo	Criar (**C**reate)
PUT	Atualizar um dado	Atualizar (**U**pdate)
PATCH	Atualizar um dado parcialmente	Atualizar (**U**pdate)
DELETE	Deletar um dado	Deletar (**D**elete)

Importante: os métodos são **convenções** que representam uma funcionalidade esperada. Nada impede que um site crie novos dados a partir de um request HTTP por método GET, por exemplo (apesar de não haver bons motivos para fugir da convenção).

Outros métodos HTTP

Existem muitos outros métodos HTTP, como:

- HEAD
- OPTIONS

- CONNECT
- TRACE
- PURGE

Contudo, eles não são usados com tanta frequência, especialmente no contexto de APIs.

Componentes de um Request

Header

O header (cabeçalho) de um Request inclui diversos **metadados que identificam o Request**, como:

- Quem está se conectando (IP, Sistema operacional, ...).
- User-Agent: qual a forma de conexão (navegador na versão X, código na linguagem Y).- Língua de resposta desejada.
- Informações de autenticação.
- Formato de arquivo de resposta desejado.

Muitos desses dados são automaticamente preenchidos para nós.

Body

O body (corpo) de um Request representa **dados que queremos transferir** para o servidor. Em geral, só precisamos incluí-lo ao usarmos métodos como POST, PUT e PATCH (o método GET geralmente não precisa enviar dados).

Também é conhecido como "payload".

Params

Os parâmetro de URL podem conter qualquer tipo de associação de chave e valor, mais ou menos como um dicionário de Python. Em muitos casos, são usados para **filtrar ou modificar a resposta** de alguma forma.

Os parâmetros de URL são inseridos dentro da URL final, e portanto sempre ficam visíveis.

Para criar parâmetros "manualmente", adicionamos um ponto de pergunta ao final da URL (?), inserimos cada chave e valor associados com igual (=), e separamos os pares de chave e valor com o "E" comercial (&).

Exemplo:

```
https://meusite.com/dashboard?acesso=admin&filtro=janeiro
```

Por que isso importa?

A flexibilidade de troca de informações por Requests pode levar a sistemas complexos. **Se a estrutura variar de site para site, fica difícil tanto de desenvolver o site quanto acessar as informações**.

Utilizar uma API nos ajuda a controlar esta complexidade, justamente por ser uma **forma padronizada de trocar informações** através de Requests!

05. Gerando e analisando Requests

Testando um Request para o HTTP Bin

O site https://httpbin.org/ funciona como um "playground". Nele, podemos testar o envio de Requests de diferentes tipos e conferir o que foi enviado.

Vamos começar com o código abaixo, que usa o método GET para testar uma URL feita especialmente para isso:

```
import requests

url = "https://httpbin.org/get"
resposta = requests.get(url=url)

print(resposta.json())
```

Note que usamos o método `resposta.json()` porque já sabemos de antemão que a resposta está em formato JSON (pois conhecemos o site). Esse método causa um erro se a resposta não puder ser convertida para um JSON, como o HTML do Google que baixamos anteriormente

Sobre o HTTP Bin

O site https://httpbin.org/ serve para testes, portanto o Request não tem nenhuma "utilidade prática". **A resposta será sempre um resumo dos dados que enviamos**, apenas para mostrar que está funcionando como esperado.

Na "vida real", a resposta será variada. A partir do nosso Request, podemos acessar dados, realizar uma operação no banco, enviar uma mensagem, ou qualquer outra atividade prática.

Testando diferentes Requests no REST Ninja

Agora vamos acessar este site: https://restninja.io/. Ele monitora e organiza Requests enviados por sua interface.

Recriando o Request GET

Vamos enviar um Request GET da mesma forma como fizemos em Python. Veja que o resultado que aparece à direita é semelhante:

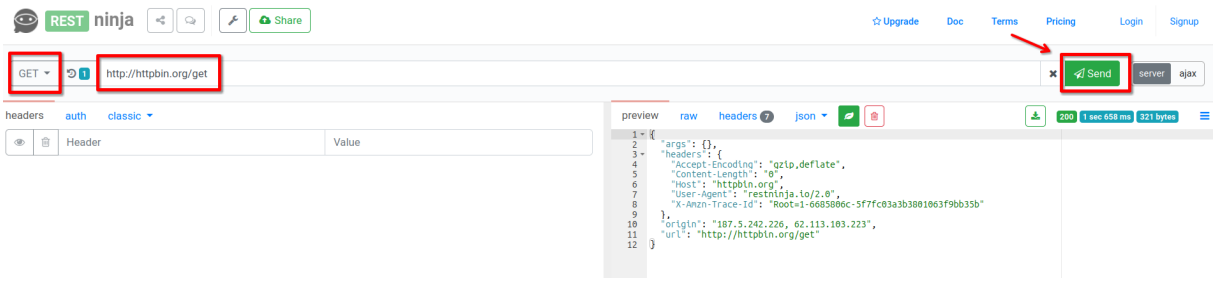

Figure 4: Interface do REST Ninja disparando um Request GET para o HTTP Bin

Sucesso! Agora vamos testar um Request que inclui dados adicionais.

Request POST com envio de dados

- Mude o tipo de Request para POST.
- Mude a URL para `/post`.
- Adicione alguns dados no body para serem passados como o "payload" do Request.

Atenção: no REST Ninja, o body precisa ser definido como JSON. A sintaxe é próxima de dicionários e listas em Python, mas com algumas diferenças:

- Os valores `True` e `False` são escritos com letras minúsculas (`true` e `false`).
- Strings são sempre definidos com aspas duplas.
- Não pode ter vírgula depois do último elemento (a chamada *trailing comma*).

Exemplo de Request:

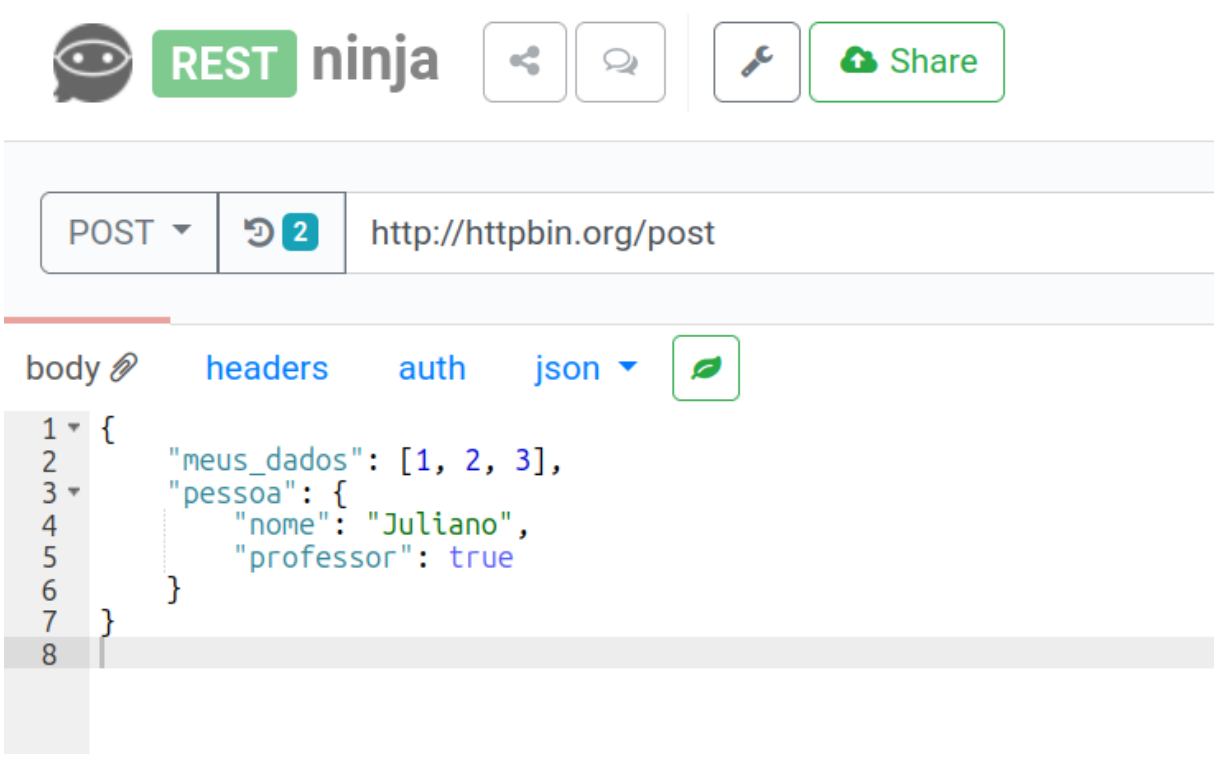

Figure 5: Exemplo de Request POST no REST Ninja

E a resposta:

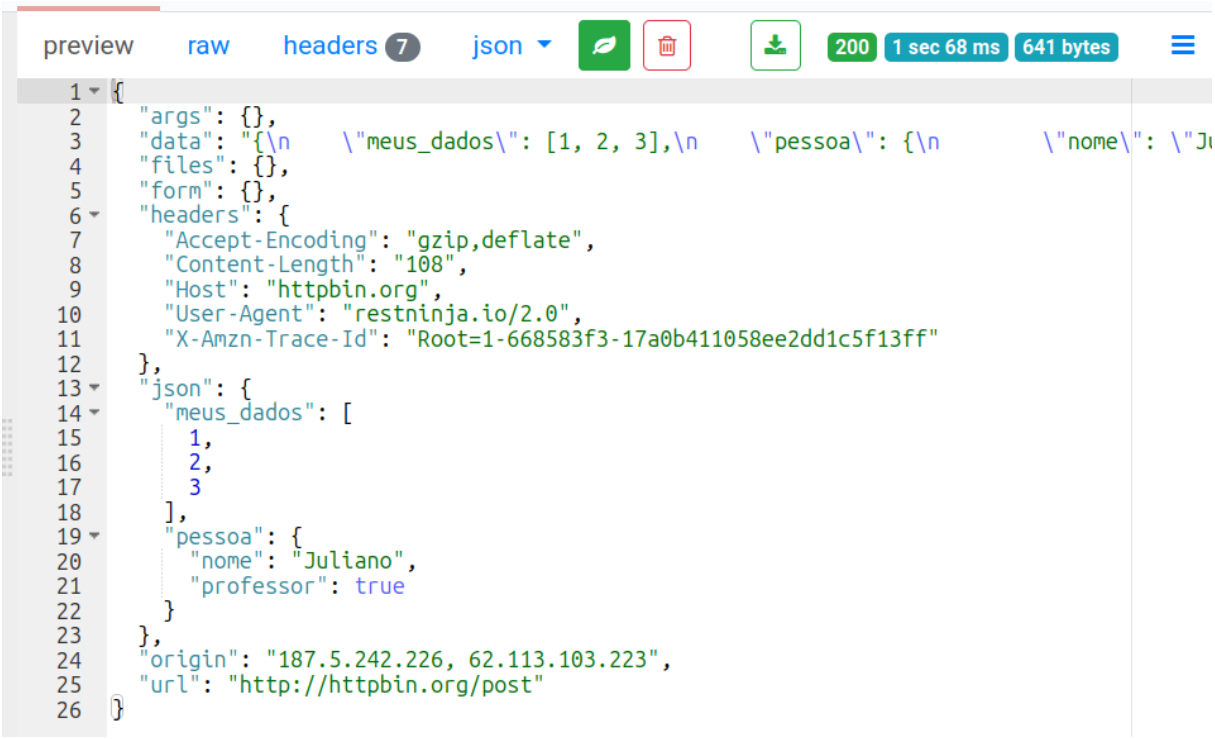

Figure 6: Resposta ao Request POST no REST Ninja

Em Python, este exemplo fica:

```
import requests

url = "https://httpbin.org/post"
data = {
    "meus_dados": [1, 2, 3],
    "pessoa": {
        "nome": "Juliano",
        "professor": True,
    }
}
resposta = requests.post(url=url, json=data)

print(resposta.json())
```

Request GET com parâmetros

Os parâmetros de um Request ficam na URL. Vamos simular um Request que busca dados de 2023, usando os seguintes parâmetros:

- `dataInicio = 2023-01-01`

- `dataFim = 2023-12-31`

A URL fica:

- `https://httpbin.org/get?dataInicio=2023-01-01&dataFim=2023-12-31`

No REST Ninja:

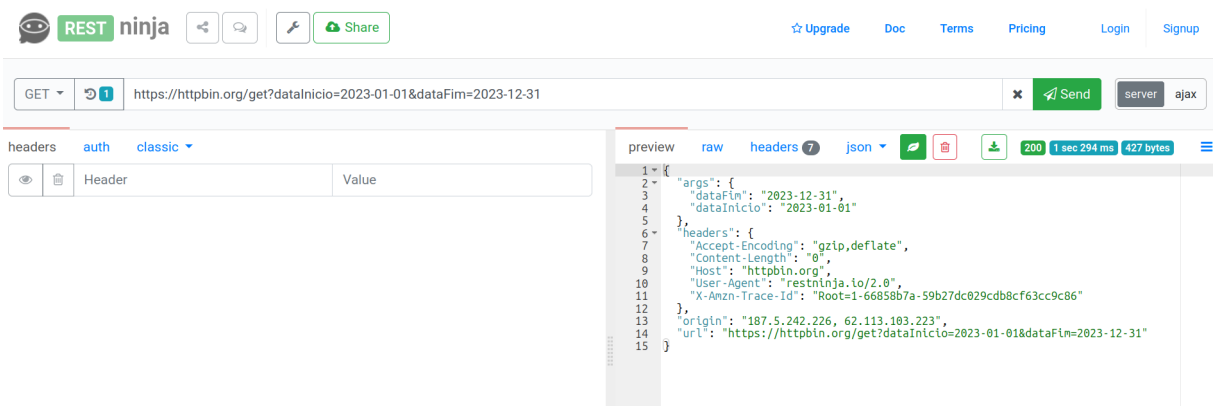

Figure 7: Request GET com parâmetros no REST Ninja

Em Python, passamos os parâmetros como um dicionário (a biblioteca `requests` se encarrega de inseri-los na URL):

```
import requests

url = "https://httpbin.org/get"
params = {
    "dataInicio": "2023-01-01",
    "dataFim": "2023-12-31",
}
resposta = requests.get(url=url, params=params)

print(resposta.json())
```

Request POST com dados + parâmetros

Agora juntando as duas coisas!

- Dados no Body
- Parâmetros na URL

No REST Ninja:

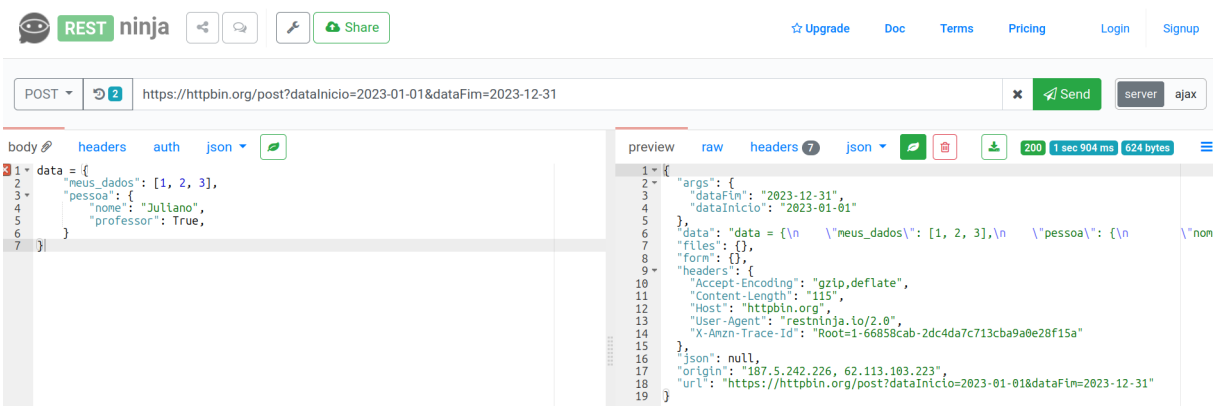

Figure 8: Request POST com dados e parâmetros no REST Ninja

Em Python:

```
import requests

url = "https://httpbin.org/post"
data = {
    "meus_dados": [1, 2, 3],
    "pessoa": {
        "nome": "Juliano",
        "professor": True,
    }
}
params = {
    "dataInicio": "2023-01-01",
    "dataFim": "2023-12-31",
}
resposta = requests.post(url=url, json=data, params=params)

print(resposta.json())
```

Mais informações sobre os Requests

Qual a diferença entre os dados do Body e os parâmetros de URLs?

Quando escrevemos código Python, tanto os dados do Body quanto os parâmetros de URL são parecidos, já que são definidos como dicionários. Mas há uma distinção entre eles:

Dados do Body

- Representam dados enviados para o servidor, sem os quais não é possível completar o Request (ex: dados que devem ser atualizados em um banco)
- Não ficam evidentes na URL.

Parâmetros de URL

- Representam informações que o servidor precisa para retornar os dados corretos (em geral, seletores ou filtros).
- Podem ser obrigatórios ou opcionais (ex: se não passar nenhum valor retorna todos os dados, se filtrar por data retorna para alguns dias)
- Ficam evidentes na URL.

Tempo de execução e código de resposta

Logo abaixo do botão “Send”, há outras informações úteis sobre o Request:

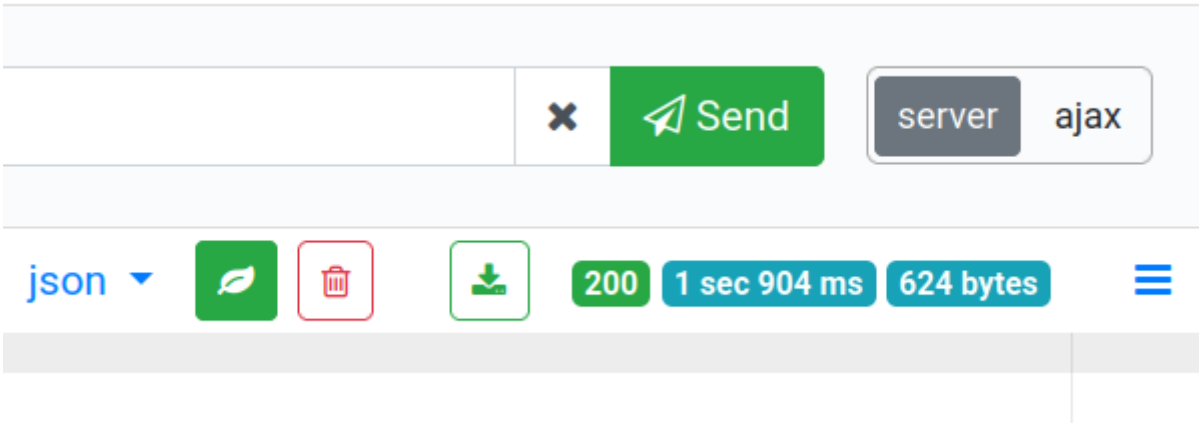

Figure 9: Informações adicionais do Request

Aqui, podemos ver:

- O tempo de execução do Request (neste caso: 1,9 segundos)
- A quantidade de informação transmitida (neste caso: 624 bytes)
- O **código de status da resposta** (neste caso: código 200)

O código de status 200 representa um Request feito com sucesso. A seguir, vamos ver os outros códigos existentes e quais erros representam!

06. Códigos de status HTTP

Os códigos de status HTTP indicam o resultado do Request (se deu certo ou não, e por qual motivo).

Eles são divididos em cinco categorias, cada uma iniciando por um número (100, 200, 300, ...).

Aqui estão os principais códigos de erro HTTP e o que eles representam:

Status 1xx: Informativo

- **100 Continue**: O servidor recebeu o cabeçalho da requisição e o cliente deve continuar com o corpo da requisição.
- **101 Switching Protocols**: O servidor está mudando os protocolos conforme solicitado pelo cliente.

Status 2xx: Sucesso

- **200 OK**: A requisição foi bem-sucedida.
- **201 Created**: A requisição foi bem-sucedida e um novo recurso foi criado.
- **204 No Content**: A requisição foi bem-sucedida, mas não há conteúdo para retornar.

Status 3xx: Redirecionamento

- **301 Moved Permanently**: O recurso solicitado foi movido permanentemente para uma nova URL.
- **302 Found**: O recurso solicitado foi encontrado, mas em uma URL diferente temporariamente.
- **304 Not Modified**: O recurso não foi modificado desde a última requisição.

Status 4xx: Erro do Cliente

- **400 Bad Request**: A requisição é inválida ou malformada.
- **401 Unauthorized**: Autenticação é necessária e falhou ou não foi fornecida.
- **403 Forbidden**: O servidor entendeu a requisição, mas se recusa a autorizá-la.
- **404 Not Found**: O recurso solicitado não foi encontrado.
- **405 Method Not Allowed**: O método HTTP usado não é permitido para o recurso.
- **429 Too Many Requests**: O cliente enviou muitas requisições em um curto período de tempo.

Status 5xx: Erro do Servidor

- **500 Internal Server Error**: O servidor encontrou uma condição inesperada que impediu a execução da requisição.
- **501 Not Implemented**: O servidor não suporta a funcionalidade necessária para atender à requisição.
- **502 Bad Gateway**: O servidor, ao atuar como gateway ou proxy, recebeu uma resposta inválida do servidor upstream.
- **503 Service Unavailable**: O servidor não está disponível para processar a requisição no momento (por exemplo, devido a manutenção).
- **504 Gateway Timeout**: O servidor, ao atuar como gateway ou proxy, não recebeu uma resposta a tempo do servidor upstream.

Testando o código de status do Request

Em Python, podemos usar `requests.raise_for_status()` para gerar uma mensagem de erro, caso o código não seja `200`.

Isso é importante porque uma resposta inválida nunca vai gerar um erro por si só!

Código que não gera erro

```
import requests

url = "https://httpbin.org/get"
resposta = requests.get(url=url)

print(resposta.status_code)  # 200
resposta.raise_for_status()  # Não causa erro aqui!
```

Código que gera erro de método errado (405)

No código abaixo, acesso a rota `/get`, que só aceita Requests com método GET, mas utilizo o método POST!

```
import requests

url = "https://httpbin.org/get"
resposta = requests.post(url=url)

print(resposta.status_code)  # 405
resposta.raise_for_status()  # HTTPError: 405 Client Error: METHOD NOT ALLOWED for url:
↪  https://httpbin.org/get
```

Código que gera erro de URL não encontrada (404)

No código abaixo, acessamos uma URL que não existe!

```
import requests

url = "https://httpbin.org/esta-url/nao-existe"
resposta = requests.get(url=url)

print(resposta.status_code)  # 404
resposta.raise_for_status()  # HTTPError: 404 Client Error: NOT FOUND for url:
↪  https://httpbin.org/esta-url/nao-existe
```

Estrutura para testar erro no seu código

Utilize um block `try`/`except` para que o erro não pause o seu código, gerando uma mensagem informativa no lugar:

```
import requests

url = "https://httpbin.org/get"
resposta = requests.get(url=url)

try:
    resposta.raise_for_status()
except requests.HTTPError as e:
    print(f'Impossível fazer requisição!\nErro: {e}')
else:
    print('Resultado:')
    print(resposta.json())
```

07. O que é uma interface

Agora que já nos aprofundamos em requisições HTTP, estamos prontos para entendermos o que é de fato uma API.

Significado de API

O nome API significa **Application Programming Interface**, ou Interface de Programação de Aplicações.

É um nome complexo para uma ideia simples: uma forma padronizada pra troca de informações. E no centro dessa ideia, existe o conceito de **Interface**.

O que é, de fato, uma interface?

Ao longo do dia, é comum ouvirmos falar em interfaces:

- A interface do meu computador travou!
- A interface do meu celular foi atualizada.
- Essa nova interface ficou mais bonita!

Com essa experiência, temos uma ideia mais ou menos intuitiva do que é uma interface, mas com pouca clareza.

Etimologia

- Interface = “entre formas” / “entre caras” / “entre corpos”.
- É **algo que está entre duas coisas**, como se fosse um elo, um ponto de conexão ou de comunicação.
- Em termos práticos, é aquilo que conecta um **Usuário** a um **Sistema** (e aqui não estamos falando de programação).
- Sua principal função é **esconder a complexidade do Sistema, de modo que a interação do Usuário com o Sistema se torne fácil e óbvia**.

Interfaces no dia a dia

Temos diversos exemplos de “interfaces” no nosso dia a dia. Por exemplo:

Carro

- **Sistema**: motor do carro
- **Usuário**: motorista
- **Interface**: pedais

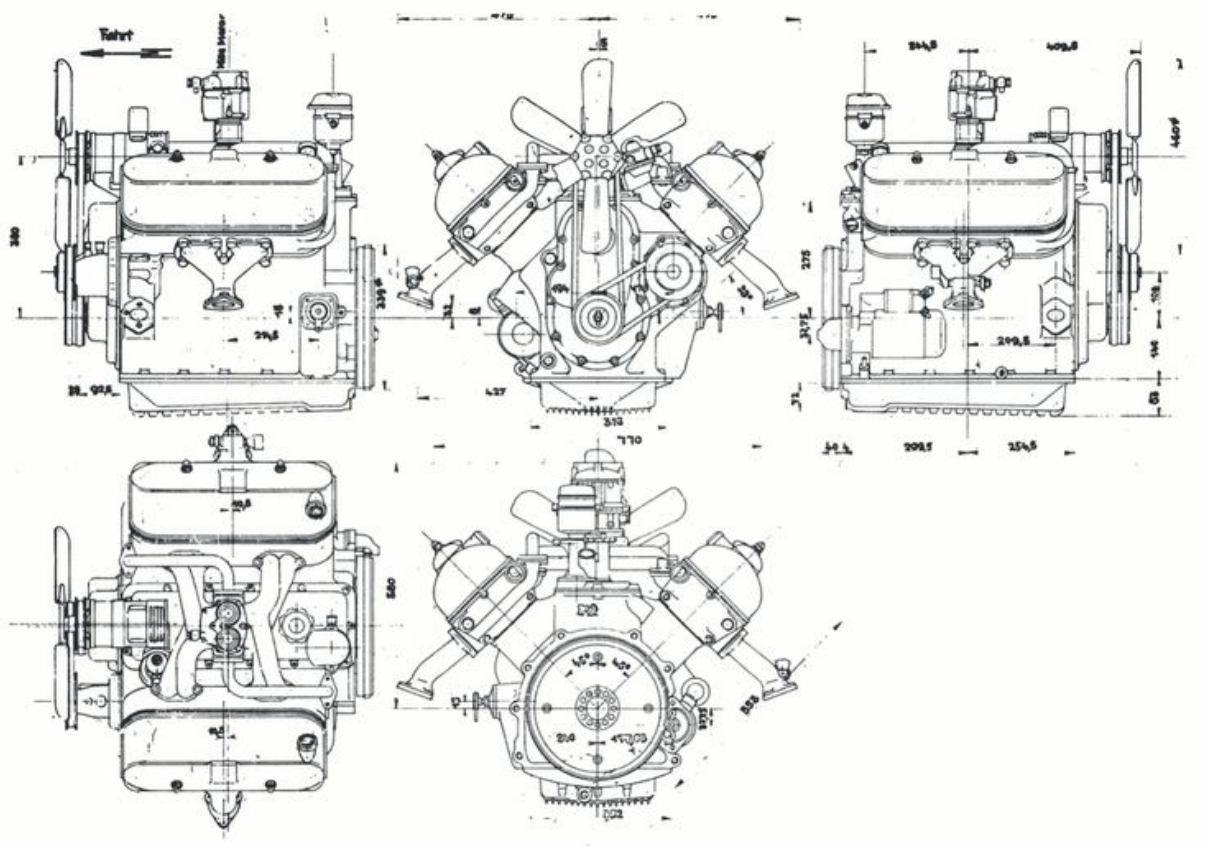

Figure 10: Motor de carro

Figure 11: Pedais: interface entre motor e motorista

Luz elétrica

- **Sistema**: instalação elétrica da casa
- **Usuário**: morador
- **Interface**: interruptor

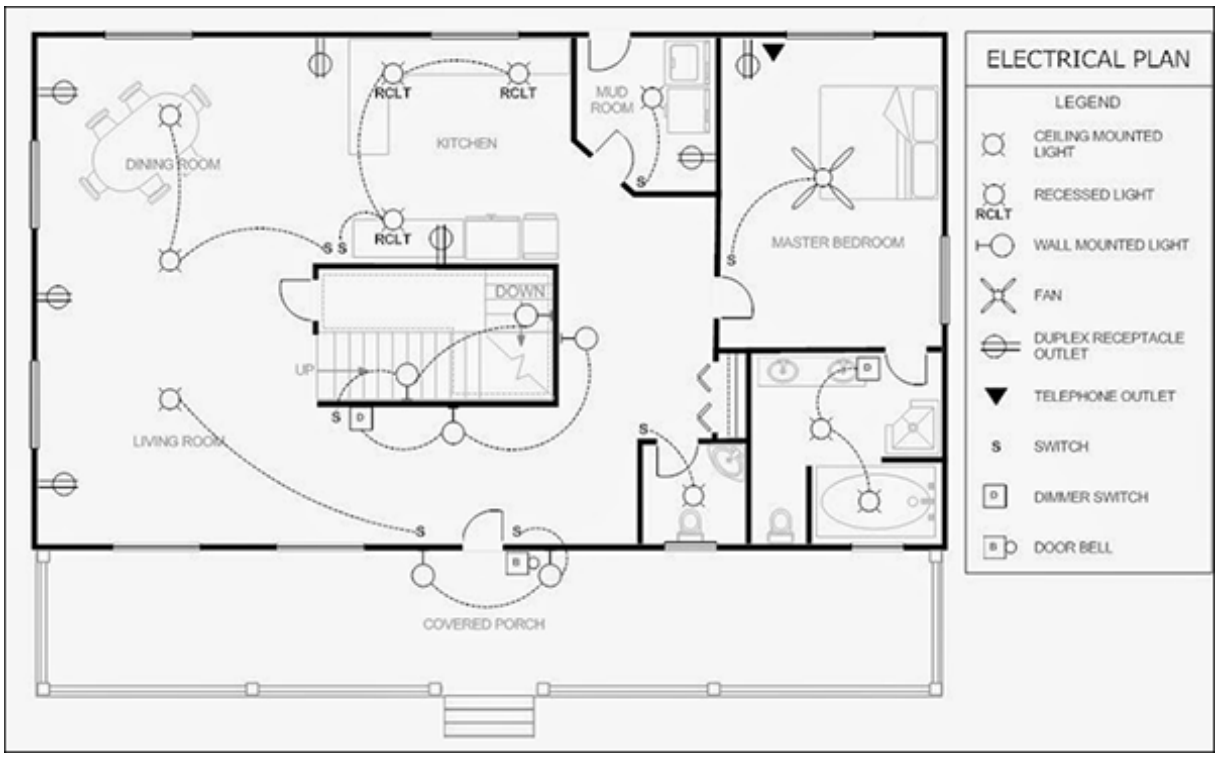

Figure 12: Instalação elétrica

Figure 13: Interruptor: interface entre instalação elétrica e morador

Elevador

- **Sistema**: elevador (motor, sistema de pesos, sistema elétrico …)
- **Usuário**: pessoa querendo ir para outro andar
- **Interface**: botões dentro do elevador

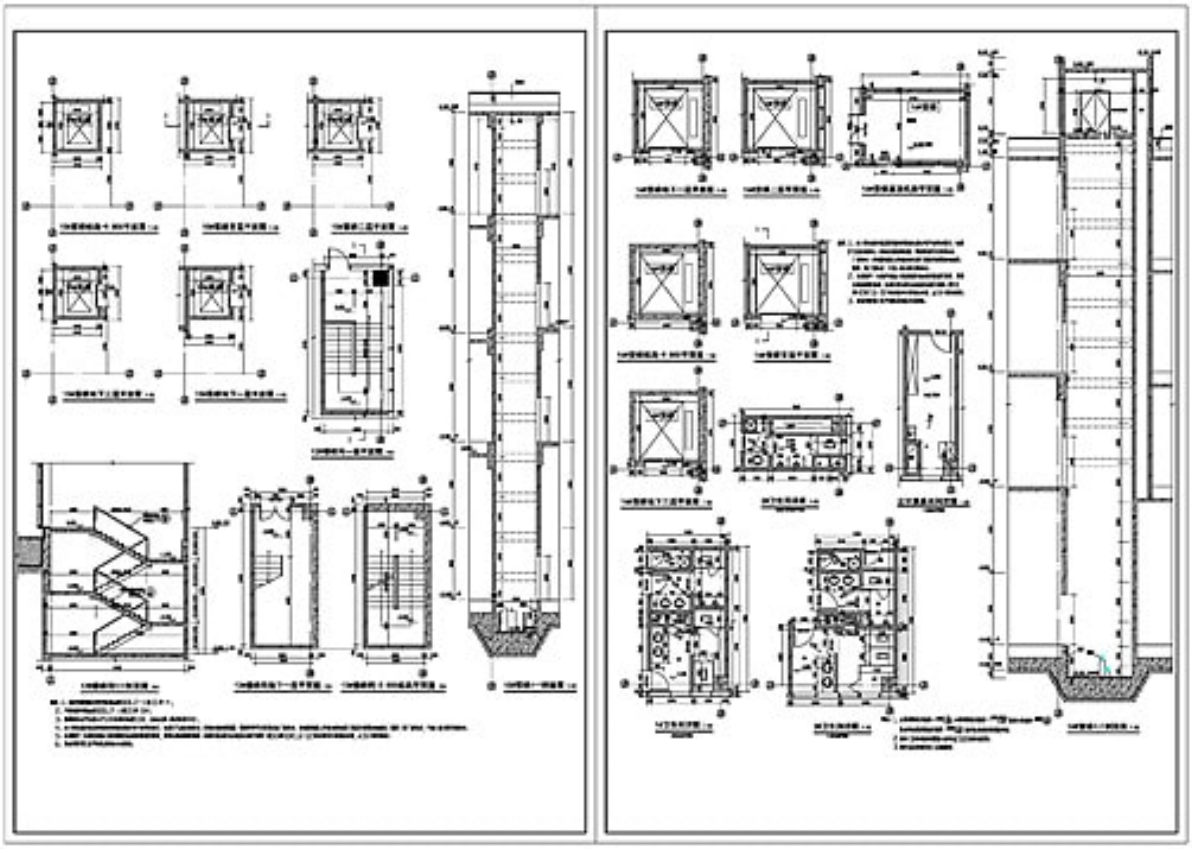

Figure 14: Planta de um elevador

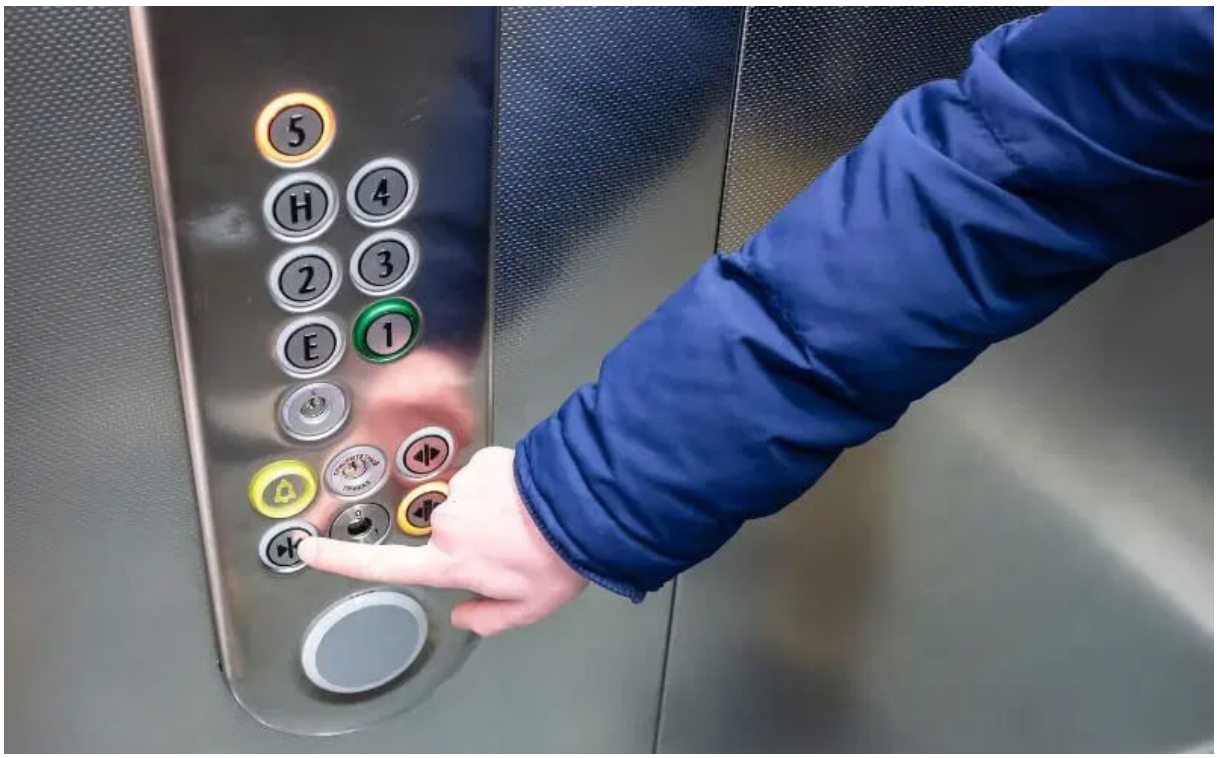

Figure 15: Botões: interface entre elevador e usuário

Para quê uma interface?

Se as interfaces são tão presentes no dia a dia a ponto de nem percebermos que elas estão lá, é sinal de que são úteis e surgiram de alguma necessidade real.

Com as interfaces, o usuário não precisa conhecer a complexidade por trás do sistema!

- Posso até entender um pouco sobre o sistema, mas para utilizá-lo não é necessário saber nada!
- Pessoas especializadas podem tomar conta do sistema, enquanto eu fico na posição de "consumidor" dela.
- Padronização é importante: qualquer carro deve ter os mesmos pedais!
- O sistema pode até entrar em manutenção, ou ser trocado totalmente, e o usuário nem perceber! Se o carro continuar rodando, posso nem perceber que o motor foi trocado.

08. O que é uma API

Agora que entendemos as **interfaces**, fica simples intender o que é uma API.

Uma API é uma interface padronizada que conecta um sistema (site, aplicativo) ao meu código.

API só existe na Internet?

Não! Quando falamos de APIs, estamos geralmente falando de APIs REST na Internet (que é algo que veremos na próxima aula). Mas existem outras APIs, mesmo fora do ambiente web.

Vamos ver conhecer algumas APIs abaixo (que não tem a ver com Web ou REST):

APIs dos sistemas operacionais

Os próprios sistemas operacionais possuem APIs. Elas são usadas **tanto internamente** pelos desenvolvedores do sistema operacional, **quanto por desenvolvedores de programas** para cada sistema operacional. Exemplos:

- API do Windows (Win32)
- API do kernel Linux

API do Python

O próprio Python possui sua "API" (apesar de a chamarmos mais comumente de "biblioteca padrão"): https://docs.python.org/3/library/index.html.

Estas são as coleções de funções que os desenvolvedores querem fornecer para a comunidade (pedais do carro), enquanto escondem outras funções e detalhes mais técnicos de como a linguagem funciona (motor do carro).

API de uma biblioteca Python

A biblioteca Matplotlib (usada para criar gráficos em Python) diz que possui duas "APIs":

- Interface `pyplot`, que lida com figuras e gráficos de forma simplificada.
- Interface orientada a objetos, que trata cada pedaço de uma figura como um objeto que pode ser configurado individualmente.

Você pode ver mais aqui: https://matplotlib.org/stable/users/explain/figure/api_interfaces.html#api-interfaces

API Vulkan

API de gráficos 3D: https://www.vulkan.org. Qualquer desenvolvedor de animação ou jogos consegue usá-la, de qualquer sistema operacional!

09. O surgimento da API REST

Agora que conhecemos APIs de uma forma geral, vamos entender o que é uma Web API, e o que faz ela ser REST ou não.

O que é REST?

REST (*REpresentational State Transfer*) é um termo definido pelo cientista da computação Roy Fielding em 2000. Durante o doutorado, Roy de debruçou sobre uma pergunta: como um sistema distribuído de informação (como a Internet) deve se comportar para que seja escalável e interoperável?

O termo REST representa um "guia de estilo" de como construir um sistema de troca de informações (diferente de HTTP, que é um protocolo estrito). Assim, uma API REST (também conhecida como APIs "RESTful") são APIs que se adequam a este estilo.

Conceitos importantes no REST

Para que uma API seja considerada REST, ela precisa implementar alguns pontos específicos. Como vocês verão, muitos destes pontos já foram abordados ao longo do curso, ainda que não tenhamos chamado à atenção para eles.

Vejamos alguns dos requisitos para uma API REST:

Cliente e Servidor

Em um sistema REST, há sempre um **cliente** (que pede algo com uma **requisição**) e um **servidor** (que entrega algo como uma **resposta**).

Ausência de estado (stateless)

REST não possui estado: olhando apenas para o Request, um servidor não tem como dizer se um cliente está se conectando pela primeira vez ou se já fez centenas de requisições.

Uma consequência disso é que **toda informação necessária deve ser enviada no Request**. Se for necessário autenticar (e veremos exemplos mais para frente), os dados de autenticação são enviados com cada Request.

Isso pode parecer contraprodutivo, mas *simplifica muito* quando falamos de servidores grandes, recebendo diversos Requests simultâneos. O servidor pode "reagir" muito rapidamente a novos

Requests porque sabe que sua resposta não depende de nada que veio antes. **Gerenciar estado é complicado**!

Recursos identificados de forma padrão (URI)

Uma API REST retorna recursos identificados por alguma nomenclatura padronizada. Chamamos esta nomenclatura de URI, ou *uniform resource identifier* (identificador de recurso uniforme).

Na prática, é algo bem parecido com uma URL. Mas existe uma diferença teórica: URLs apontam sempre para uma **localização** (página web), enquanto URI é um **recurso genérico**. Poderia representar um documento, uma imagem, um usuário, uma entrada em um banco de dados, …

Em geral, os recursos retornados pela API vêm no formato JSON ou XML. E as formas de interagir com as URIs são os métodos de HTTP que já conhecemos: `GET`, `UPDATE`, …

Cacheamento

Por fim, uma API REST deve permitir cacheamento de resultados. Isso nada mais significa que, se um resultado for acessado com muita frequência, o servidor pode mantê-lo em memória sem precisar carregá-lo toda vez de um banco de dados.

Por que APIs REST importam?

Se sei que uma API é REST, já conheço a forma padronizada de como interagir com ela. Afinal, ela também é uma **interface**!

Isso significa que o servidor pode ser completamente trocado ou reconfigurado, e desde que aqueles "pontos de contato" da API permaneçam retornando os mesmos recursos, o usuário pode nem perceber a diferença. Equivale a trocar o motor do carro, mas manter o mesmo pedal.

Além disso, uma mesma API pode servir dados para diferentes ambientes: o site da empresa, para frontend de clientes de diferentes níveis, acessos via código… É basicamente o que a API da OpenAI faz para acessar o ChatGPT.

Outros modelos de APIs

Por mais famosa que seja, nem toda API na web se baseia em REST. A seguir, listamos alguns outros modelos de APIs:

- **SOAP (*Simple Object Access Protocol*):** protocolo mais rigoroso e padronizado que REST, baseado apenas em XML.
- **GraphQL:** usado para buscas em bancos de dados complexos.
- **gRPC e WebSockets**: dois formatos usados para comunicação bidirecional de baixa latência (*streaming* e *updates* em tempo real).

10. Acessando nossa primeira API

Hora de botar a mão na massa!

Vamos explorar algumas APIs do IBGE, para trabalharmos com dados brasileiros. Você encontra todas as APIs do IBGE aqui: https://servicodados.ibge.gov.br/api/docs/

API de Nomes

O IBGE disponibiliza a **API de Nomes**, com a qual podemos receber a frequência de nomes por década de nascimento: https://servicodados.ibge.gov.br/api/docs/nomes?versao=2. Por ser uma API relativamente simples, é um ótimo exemplo para teste.

Logo no primeiro exemplo, vemos uma URL (tecnicamente, um URI) com alguns exemplos de uso:

Figure 16: Endpoint “Frequência por nome” da API de nomes do IBGE

A seguir, vamos acessar os dados dessa URL.

Componentes da URL

Pela documentação, o método HTTP a ser utilizado é o GET.

A URL é composta por:

- Uma **URL base**, que é a mesma para todas as URLs dessa API:

`https://servicodados.ibge.gov.br/api/v2/censos/`

- Uma **terminação ou *endpoint*** da URL, que indica qual o recurso requisitado - neste exemplo, `nomes/{nome}`

Note que neste caso, **o próprio endpoint recebe um parâmetro**. Ele indica o nome a ser buscado no banco de dados do IBGE, e é obrigatório (segundo a própria documentação).

- **Atenção**: não confundir com parâmetros da URL (*query parameters*)! O parâmetro do endpoint representa o caminho completo até o recurso, enquanto os *query parameters* representam formas de filtrar o conteúdo (e vão ao final da URL, após o `?` como já vimos anteriormente).

Fazendo o Request

Vamos fazer agora nosso primeiro Request para a API!

```
from pprint import pprint
import requests

url = "https://servicodados.ibge.gov.br/api/v2/censos/nomes/juliano"
resposta = requests.get(url=url)

try:
    resposta.raise_for_status()
except requests.HTTPError as e:
    print(f"Erro no request: {e}")
    resultado = None
else:
    resultado = resposta.json()

pprint(resultado)
```

E o resultado (usamos `pprint.pprint` no lugar do `print` para que a saída apareça formatada, facilitando a leitura):

```
[{'localidade': 'BR',
  'nome': 'JULIANO',
  'res': [{'frequencia': 74, 'periodo': '1930['},
          {'frequencia': 164, 'periodo': '[1930,1940['},
          {'frequencia': 304, 'periodo': '[1940,1950['},
```

```
          {'frequencia': 627, 'periodo': '[1950,1960['},
          {'frequencia': 1431, 'periodo': '[1960,1970['},
          {'frequencia': 18982, 'periodo': '[1970,1980['},
          {'frequencia': 52111, 'periodo': '[1980,1990['},
          {'frequencia': 29346, 'periodo': '[1990,2000['},
          {'frequencia': 12697, 'periodo': '[2000,2010['}],
  'sexo': None}]
```

Veja que a resposta surge em listas e dicionários - como esperado!

Lidando com erros no Request

Qualquer erro no Request cai no bloco try/except, exibindo a mensagem. Assim, sabemos se o Request deu certo ou não.

Faça as seguintes alterações e veja as mensagens de erro:

Mudar método de `requests.get` por `requests.post`

- Erro gerado: `Erro no request: 405 Client Error: Method Not Allowed for url: https://servicodados.ibge.gov.br/api/v2/censos/nomes/juliano`

Mudar endpoint de `nomes/juliano` por `xxx/juliano`

- Erro gerado: `Erro no request: 503 Server Error: Service Unavailable for url: https://servicodados.ibge.gov.br/api/v2/censos/xxx/juliano`
- Tecnicamente, um erro `404` seria mais correto aqui.

Note que estes erros são diferentes de não retornar dados: o Request para

`https://servicodados.ibge.gov.br/api/v2/censos/nomes/xxx`

Funciona, mas retorna uma lista vazia porque não há dados para este nome!

11. Schemas de resposta e parâmetros de URL

Voltando para a documentação na página do IBGE, vemos que existem algumas seções adicionais. Vamos entendê-las agora:

Schema de resposta

Esta seção exemplifica a estrutura de dados a ser retornada. Isso nos ajuda a entender todos os campos da resposta.

Imagine sempre que a resposta virá no formato de listas e dicionários de Python, de forma "aninhada" (listas e dicionários dentro de outras listas e dicionários).

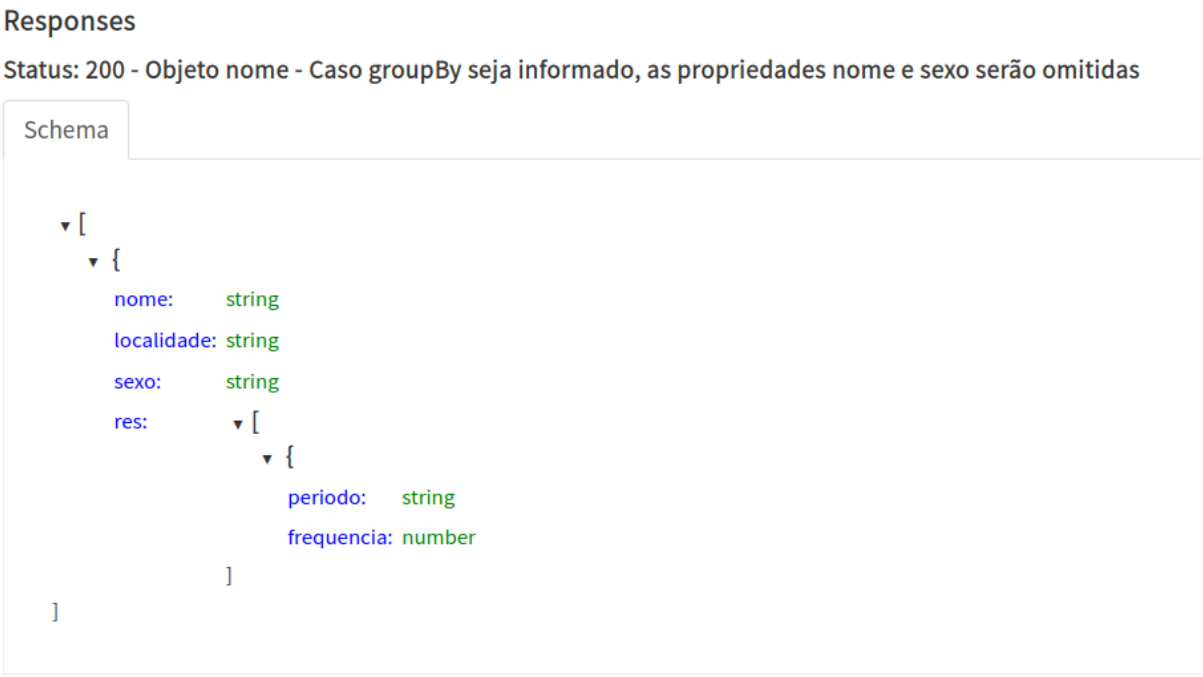

Figure 17: Schema de resposta para o endpoint `/nomes` da API de nomes do IBGE.

Parâmetros de URL

Aqui ficam listados os parâmetros de URL que podemos adicionar para modificar nosso Request:

Query parameters

Name	Description
sexo	▾ string *Por padrão, a consulta pelo nome é unissex. Caso deseje filtrar pelo sexo, informe o parâmetro sexo, cujos valores podem ser **M**, para o sexo masculino, ou **F**, para o feminino* https://servicodados.ibge.gov.br/api/v2/censos/nomes/ariel?sexo=F *Frequência referente ao nome **Ariel**, sexo Feminino*
groupBy	▾ string *Caso deseje obter a frequência por algum nível geográfico, informe o parâmetro groupBy - Válido apenas quando informado um único nome. Nesta versão, apenas o valor **UF** é válido, no qual se obtém a frequência do nome informado por Unidade da Federação* https://servicodados.ibge.gov.br/api/v2/censos/nomes/joao?groupBy=UF *Frequência referente ao nome **João**, agrupados por Unidade da Federação (UF)*
localidade	▾ string *Caso deseje obter a frequência referente a uma dada localidade, informe o parâmetro localidade. Por padrão, assume o valor **BR**, mas pode ser o identificador de um município ou de uma Unidade da Federação* https://servicodados.ibge.gov.br/api/v2/censos/nomes/joao?localidade=33 *Frequência referente ao nome **João** no Rio de Janeiro (33)*

Figure 18: Parâmetros de URL para o endpoint `/nomes` da API de nomes do IBGE.

Com estes parâmetros, podemos controlar como o Request é feito! Veja o exemplo:

```
from pprint import pprint
import requests

url = "https://servicodados.ibge.gov.br/api/v2/censos/nomes/juliano"
params = {
    'sexo': 'M',
    'localidade': 33,
}

resposta = requests.get(url=url, params=params)

try:
    resposta.raise_for_status()
except requests.HTTPError as e:
    print(f"Erro no request: {e}")
    resultado = None
else:
    resultado = resposta.json()

pprint(resultado)
```

Aqui, filtramos o resultado de nomes por pessoas de sexo masculino e que nasceram no Estado do Rio de Janeiro:

```
[{'localidade': '33',
  'nome': 'JULIANO',
  'res': [{'frequencia': 17, 'periodo': '[1930,1940['},
          {'frequencia': 28, 'periodo': '[1940,1950['},
          {'frequencia': 55, 'periodo': '[1950,1960['},
          {'frequencia': 110, 'periodo': '[1960,1970['},
          {'frequencia': 755, 'periodo': '[1970,1980['},
          {'frequencia': 1804, 'periodo': '[1980,1990['},
          {'frequencia': 1059, 'periodo': '[1990,2000['},
```

```
          {'frequencia': 546, 'periodo': '[2000,2010['}],
  'sexo': 'M'}]
```

Confirmando os parâmetros

No código, podemos adicionar a seguinte linha para ver qual a URL final que de fato foi utilizada:

```
print(resposta.request.url)

# output
# https://servicodados.ibge.gov.br/api/v2/censos/nomes/juliano?sexo=M&localidade=33
```

Ou seja, os argumentos do dicionário `params` de fato se tornaram parâmetros da URL final. Poderíamos ter escrito a URL com parâmetros "na mão", mas a biblioteca `requests` serve justamente para nos poupar deste trabalho.

Efeito dos parâmetros no retorno

Já o parâmetro `groupBy` nos permite agrupar a resposta por Estado. Neste caso, o schema de resposta será diferente (a API não permite obter valores por Estado e para cada década simultaneamente):

```
from pprint import pprint
import requests

url = "https://servicodados.ibge.gov.br/api/v2/censos/nomes/juliano"
params = {
    'sexo': 'M',
    'groupBy': 'UF',
}

resposta = requests.get(url=url, params=params)

try:
    resposta.raise_for_status()
except requests.HTTPError as e:
    print(f"Erro no request: {e}")
    resultado = None
else:
    resultado = resposta.json()

pprint(resultado)
print(resposta.request.url)
```

E o resultado:

```
[{'localidade': '43',
  'res': [{'frequencia': 19548, 'populacao': 10693929, 'proporcao': 182.8}]},
 {'localidade': '42',
  'res': [{'frequencia': 11180, 'populacao': 6248436, 'proporcao': 178.92}]},
```

```
 {'localidade': '41',
  'res': [{'frequencia': 14213, 'populacao': 10444526, 'proporcao': 136.08}]},
 {'localidade': '50',

[...]
```

12. Combinando Requests de APIs diferentes

Na última aula, foi retornado um número de ID para cada localidade, mas não sabemos que Estado representam.

Para descobrir isso, precisamos cruzar com uma chamada para outra API do IBGE: a **API de Localidades** https://servicodados.ibge.gov.br/api/docs/localidades.

A API de Localidades do IBGE

Na API, encontramos o *endpoint* que lista todos os estados: https://servicodados.ibge.gov.br/api/docs/localidades#api-UFs-estadosGet. Vamos usar também o parâmetro `view=nivelado` para retornar uma lista de dicionários de um nível apenas:

```
from pprint import pprint
import requests

url = "https://servicodados.ibge.gov.br/api/v1/localidades/estados"
params = {
    'view': 'nivelado',
}

resposta = requests.get(url=url, params=params)

try:
    resposta.raise_for_status()
except requests.HTTPError as e:
    print(f"Erro no request: {e}")
    resultado = None
else:
    resultado = resposta.json()

pprint(resultado)
```

Recebemos uma lista de dicionários, com cada dicionário contendo informações de um Estado:

```
[{'UF-id': 11,
  'UF-nome': 'Rondônia',
  'UF-sigla': 'RO',
  'regiao-id': 1,
  'regiao-nome': 'Norte',
  'regiao-sigla': 'N'},

[...]
```

Cruzando os dados entre chamadas

Agora temos tudo que precisamos! Vamos fazer o seguinte:

- Mover o código do Request para dentro de uma função, de forma que possa ser reaproveitado.
- Chamar a API de Localidades e criar um dicionário de `ID: Nome do Estado` para cada Estado.
- Chamar a API de Nomes, passando o parâmetro `groupBy=UF` para agrupar a presença de um nome em cada Estado.
- Exibir de forma útil no console.

Código final:

```
from pprint import pprint
import requests


def fazer_request(url, params=None):
    resposta = requests.get(url=url, params=params)
    try:
        resposta.raise_for_status()
    except requests.HTTPError as e:
        print(f"Erro no request: {e}")
        resultado = None
    else:
        resultado = resposta.json()
    return resultado


def pegar_id_estados():
    url = "https://servicodados.ibge.gov.br/api/v1/localidades/estados"
    dados_estados = fazer_request(url=url, params={'view': 'nivelado'})
    dict_estados = {}
    for dados in dados_estados:
        id_estado = dados['UF-id']
        nome_estado = dados['UF-nome']
        dict_estados[id_estado] = nome_estado
    return dict_estados


def pegar_frequencia_nome_por_estado(nome):
    url = f"https://servicodados.ibge.gov.br/api/v2/censos/nomes/{nome}"
    frequencias_nome_por_estado = fazer_request(url=url, params={'groupBy': 'UF'})
    dict_frequencia = {}
    for dados in frequencias_nome_por_estado:
        id_estado = int(dados['localidade'])
        frequencia = dados['res'][0]['proporcao']
        dict_frequencia[id_estado] = frequencia
    return dict_frequencia


def main(nome):
    dict_estados = pegar_id_estados()
    dict_frequencia = pegar_frequencia_nome_por_estado(nome=nome)
    print(f'--- Frequência do nome "{nome}" no Estados (por 100.000 habitantes)')
    for id_estado, nome_estado in dict_estados.items():
```

```
        frequencia_estado = dict_frequencia[id_estado]
        print(f'-> {nome_estado}: {frequencia_estado}')


if __name__ == '__main__':
    main(nome='juliano')
```

13. Miniprojeto - Web App de popularidade de nomes do IBGE

Vamos construir um WebApp para exibir os dados obtidos na última aula!

Para isso, vamos usar as bibliotecas **Streamlit** (para gerar o webapp) e **Pandas** (para organizar os dados em tabela e plotá-los em um gráfico).

Resultado

```
import pandas as pd
import requests
import streamlit as st


def fazer_request(url, params=None):
    resposta = requests.get(url=url, params=params)
    try:
        resposta.raise_for_status()
    except requests.HTTPError as e:
        print(f"Erro no request: {e}")
        resultado = None
    else:
        resultado = resposta.json()
    return resultado


def pegar_nome_por_decada(nome):
    url = f"https://servicodados.ibge.gov.br/api/v2/censos/nomes/{nome}"
    nome_por_decada = fazer_request(url=url)
    if not nome_por_decada:  # Sem resultados para a chamada da API!
        return {}
    dict_decadas = {}
    for dados in nome_por_decada[0]['res']:
        decada = dados['periodo']
        quantidade = dados['frequencia']
        dict_decadas[decada] = quantidade
    return dict_decadas


def main():
    # Cabeçalho do Web App
    st.title('Web App Nomes')
    st.write('Dados do IBGE (fonte:
↪   https://servicodados.ibge.gov.br/api/docs/nomes?versao=2)')
    nome = st.text_input('Consulte um nome:')
    if not nome:
        st.stop()

    # Acessa dados do IBGE
    dict_decadas = pegar_nome_por_decada(nome=nome)
    if not dict_decadas:  # Sem dados do IBGE para este nome
```

```
        st.warning(f'Nenhum dado encontrado para o nome "{nome}"!')
        st.stop()

    # Exibe dados do IBGE
    df = pd.DataFrame.from_dict(dict_decadas, orient='index')
    col1, col2 = st.columns([0.3, 0.7])
    with col1:  # coluna esquerda
        st.write('Frequência por década')
        st.dataframe(df)
    with col2:  # coluna direita
        st.write('Evolução no tempo')
        st.line_chart(df)


if __name__ == '__main__':
    main()
```

14. APIs privadas e autenticação básica

Até aqui, utilizamos **APIs públicas**, isto é, que não requerem nenhum tipo de autenticação. Mas no cotidiano de um programador, é muito mais comum nos depararmos com **APIs fechadas ou privadas**, que requerem **autenticação** para serem consumidas.

O motivo para isso são diversos: os dados retornados podem ser privados, ou o sistema pode querer que o usuário se identifique (mais fácil de bloquear acesso em caso de abuso no uso). Além disso, algumas APIs são vendidas como serviços, uma vez que há um custo associado a disponibilizar servidores e dados.

Autenticação e autorização

APIs privadas não são necessariamente pagas, mas requerem que você se identifique ao utilizar o serviço. Essa identificação é feita através da **autenticação**. Uma vez autenticado, o cliente (usuário) informa ao servidor quem ele é em cada Request feito.

Com base na autenticação, o servidor consegue definir a **autorização** (ou bloqueio) do usuário a certo recurso. Isso permite que certos recursos fiquem bloqueados apenas a certos usuários que tenham a autorização para acessá-los.

Autenticação básica (usuário e senha)

A autenticação básica é o modo mais "simples" (e também menos seguro) de autenticação. O nome de usuário e senha vão no header no formato de string `base64`, e são enviados em todos os requests.

Para garantir a segurança, só deve ser usado com HTTPS. Mesmo assim, não é o mais recomendado atualmente!

Exemplo de autenticação básica

Nos exemplos abaixo, usaremos novamente o https://httpbin.org/ para testar os nossos requests de autenticação. Ele possui o endpoint `/basic-auth`, capaz de simular uma autenticação correta/incorreta de acordo com a URL criada:

```
import base64
from pprint import pprint

import requests

url = "https://httpbin.org/basic-auth/meu-usuario/senha-secreta"
```

```
usuario = "meu-usuario"
senha = "senha-secreta"

auth_string = f'{usuario}:{senha}'.encode()  # virou bytes UTF-8
auth_string = base64.b64encode(auth_string)  # virou bytes b64
auth_string = auth_string.decode()  # virou string b64

print('String de autenticação final:')
print(auth_string)

headers = {
    'Authorization': f'Basic {auth_string}'
}

resposta = requests.get(url, headers=headers)

try:
    resposta.raise_for_status()
except requests.HTTPError as e:
    print(f"Erro no request: {e}")
    resultado = None
else:
    resultado = resposta.json()

pprint(resultado)
```

Note que o `requests` tem uma classe facilitadora `HTTPBasicAuth`, que simplifica a parte "chata" de ajustar os strings:

```
import base64
from pprint import pprint

import requests
from requests.auth import HTTPBasicAuth

url = "https://httpbin.org/basic-auth/meu-usuario/senha-secreta"

usuario = "meu-usuario"
senha = "senha-secreta"

auth = HTTPBasicAuth(username=usuario, password=senha)

resposta = requests.get(url, auth=auth)

try:
    resposta.raise_for_status()
except requests.HTTPError as e:
    print(f"Erro no request: {e}")
    resultado = None
else:
    resultado = resposta.json()

pprint(resultado)
```

15. Autenticação Bearer com chave de API

Autenticação Bearer

Na autenticação Bearer (portador), o usuário gera algum tipo de “chave” ou “token” a partir de sua senha. A partir daí, esse token é usado para identificá-lo. Isso é mais simples e seguro porque, em caso de vazamento, é mais simples revogar um token que a senha do usuário!

Como funciona no Request

Do ponto de vista do Request, o ajuste é muito simples: basta modificar o header para incluir a seguinte porção:

```
token = "xxx"
headers = {
    'Authorization': f'Bearer {token}'
}
```

É isso! O token é enviado no Request, e o servidor se encarrega de verificar a que usuário pertence.

Formas de fazer autenticação Bearer

Existem **3 métodos principais** de se fazer uma autenticação Bearer:

- Chave de API
- Token de acesso
- JSON web token (JWT)

Do ponto de vista da programação, todas funcionam da mesma forma: um string passado dentro do header. O que muda entre elas é a forma com que o token é obtido.

Chave de API

Uma chave de API é um string qualquer que identifica o usuário. Apesar de aparentemente aleatória, essa string fica vinculada ao cadastro do usuário no servidor. Assim, o servidor entende que um Request feito com aquela string veio daquele usuário.

Geralmente, é criada pela interface web de um site. Ou seja, preciso primeiro criar o cadastro no site e gerar a chave de forma “manual” para depois incluí-la nos meus códigos.

Nem sempre é enviada pelo header do Request: consulte sempre a documentação do serviço para entender como usá-lo!

Criando uma chave de API no OpenWeather

O OpenWeather é um serviço que disponibiliza previsões do tempo para qualquer região do mundo: https://openweathermap.org/. A empresa fornece um nível gratuito para consultas via código com chave de API, bastando criar uma conta.

O processo é simples: crie a conta, valide seu endereço de email e entre na seção "API keys":

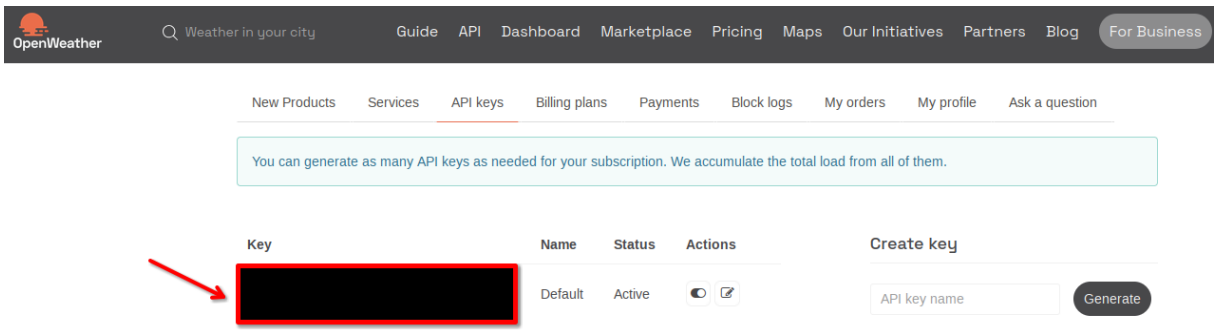

Figure 19: Seção de chaves de API do OpenWeather (chave foi ocultada na imagem)

Com esta chave em mãos, você consegue fazer uma chamada para a API. Note que, segundo a documentação do OpenWeather (https://openweathermap.org/current), é necessário passar a chave de API como um parâmetro.

Importante: pode levar algumas horas até sua chave de API ser liberada. Portanto, se você usar o código abaixo e receber um erro 403 de acesso não autorizado, **espere algumas horas e tente novamente**:

```
from pprint import pprint

import requests

url = "http://api.openweathermap.org/data/2.5/weather"
params = {
    'q': 'Porto Alegre',
    'appid': 'SUA_CHAVE_DE_API_VAI_AQUI'
}

resposta = requests.get(url, params=params)

try:
    resposta.raise_for_status()
except requests.HTTPError as e:
    print(f"Erro no request: {e}")
    print(resposta.json())
    resultado = None
```

```
else:
    resultado = resposta.json()

pprint(resultado)
```

Mantendo sua chave segura

No exemplo acima, escrevemos a chave diretamente no código. **Isso não é uma boa alternativa**, já que se qualquer pessoa ler nosso código, terá acesso à nossa conta!

Uma boa forma de manter nossa chave segura é passando ela como uma **variável de ambiente**. E uma forma simples de gerenciar estas variáveis em Python é com a biblioteca `python-dotenv`:

```
pip install python-dotenv
```

Após instalá-la, crie um arquivo chamado `.env` ("ponto-env"), o nome padrão para um arquivo de variáveis de ambiente:

```
CHAVE_API_OPENWEATHER="SUA_CHAVE_DE_API_VAI_AQUI"
```

Agora, adaptamos o código para ler a chave a partir deste arquivo, sem que fique escrita diretamente no código:

```
import os
from pprint import pprint

import dotenv
import requests

# Carrega e lê variáveis de ambiente
dotenv.load_dotenv(dotenv.find_dotenv())
app_id = os.environ['CHAVE_API_OPENWEATHER']

url = "http://api.openweathermap.org/data/2.5/weather"
params = {
    'q': 'Porto Alegre',
    'appid': app_id,
}

resposta = requests.get(url, params=params)

try:
    resposta.raise_for_status()
except requests.HTTPError as e:
    print(f"Erro no request: {e}")
    print(resposta.json())
    resultado = None
else:
    resultado = resposta.json()

pprint(resultado)
```

O dicionário `os.environ` mantém as variáveis de ambiente. Assim, se o carregamento do arquivo der certo, a variável `app_id` terá o valor da sua chave!

Agora, você está seguro para compartilhar o código com seus colegas, passá-lo para o GitHub, etc - mantendo sempre o cuidado para que o arquivo `.env` não vá junto.

16. Miniprojeto - Web App de tempo com OpenWeather

Vamos agora criar um webapp que exibe o clima atual em algum local qualquer. Para isso, usarmos a API da OpenWeather juntamente do **Streamlit** que usamos anteriormente.

Resultado

```
import os

import dotenv
import requests
import streamlit as st

dotenv.load_dotenv(dotenv.find_dotenv())

dict_clima = {
    'céu limpo': 'Céu limpo',
    'algumas nuvens': 'Céu com algumas nuvens',
    'nublado': 'Nublado',
    'névoa': 'Névoa',
}


def fazer_request(url, params=None):
    resposta = requests.get(url=url, params=params)
    try:
        resposta.raise_for_status()
    except requests.HTTPError as e:
        print(f"Erro no request: {e}")
        resultado = None
    else:
        resultado = resposta.json()
    return resultado


def pegar_tempo_para_local(local):
    app_id = os.environ['CHAVE_API_OPENWEATHER']
    url = f"https://api.openweathermap.org/data/2.5/weather"
    params = {
        'q': local,
        'appid': app_id,
        'units': 'metric',
        'lang': 'pt_br',
    }
    dados_tempo = fazer_request(url=url, params=params)
    return dados_tempo


def main():
    # Cabeçalho do Web App
    st.title('Web App Tempo')
```

```
    st.write('Dados do OpenWeather (fonte: https://openweathermap.org/current)')
    local = st.text_input('Busque uma cidade:')
    if not local:
        st.stop()

    # Acessa dados do OpenWeather
    dados_tempo = pegar_tempo_para_local(local=local)
    if not dados_tempo:  # Sem dados para este local
        st.warning(f'Localidade "{local}" não foi encontrada no banco de dados da
↪  OpenWeather!')
        st.stop()

    # extrai dados retornados para variáveis
    clima_atual = dados_tempo['weather'][0]['description']
    clima_atual = dict_clima.get(clima_atual, clima_atual)
    temperatura = dados_tempo['main']['temp']
    sensacao_termica = dados_tempo['main']['feels_like']
    umidade = dados_tempo['main']['humidity']
    cobertura_nuvens = dados_tempo['clouds']['all']

    # Exibe no Web App
    st.metric(label='Tempo atual', value=clima_atual)
    col1, col2 = st.columns(2)
    with col1:
        st.metric(label='Temperatura', value=f'{temperatura} °C')
        st.metric(label='Sensação térmica', value=f'{sensacao_termica} °C')
    with col2:
        st.metric(label='Umidade do ar', value=f'{umidade}%')
        st.metric(label='Cobertura de nuvens', value=f'{cobertura_nuvens}%')


if __name__ == '__main__':
    main()
```

17. A documentação de uma API

Ao longo do curso, mencionamos a documentação de APIs conforme fomos precisando delas. Mas é importante dedicarmos um momento para entender a estrutura de uma documentação de API.

A API do Spotify

A API do Spotify contém recursos para quem quer acessar dados de músicas e artistas. Também é possível desenvolver aplicações a partir dela (por exemplo, controlar o playback de músicas).

Comece explorando a API a partir daqui: https://developer.spotify.com/documentation/web-api

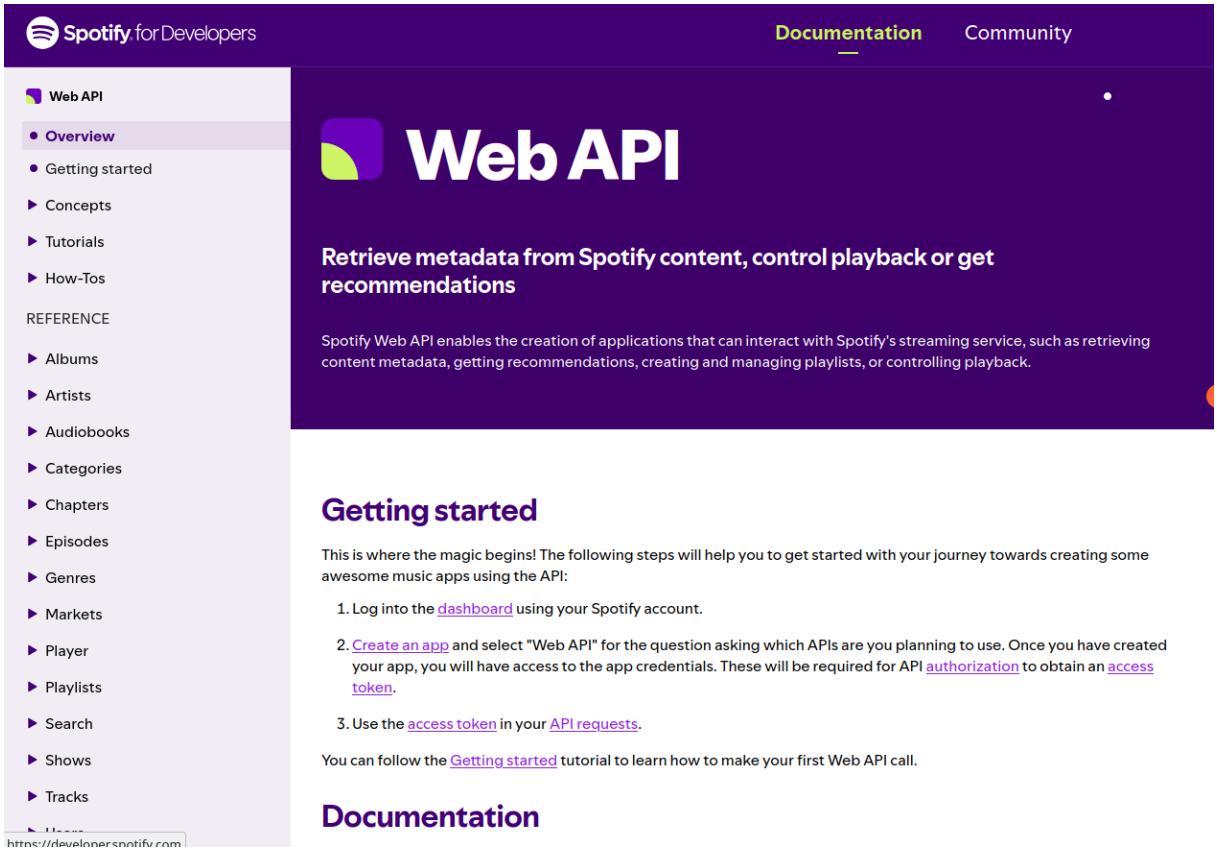

Figure 20: Página inicial da API do Spotify.

As páginas da documentação

As páginas da documentação do Spotify (por exemplo: https://developer.spotify.com/documentation/web-api/reference/get-track) seguem mais ou menos o “padrão ouro” de documentações de APIs:

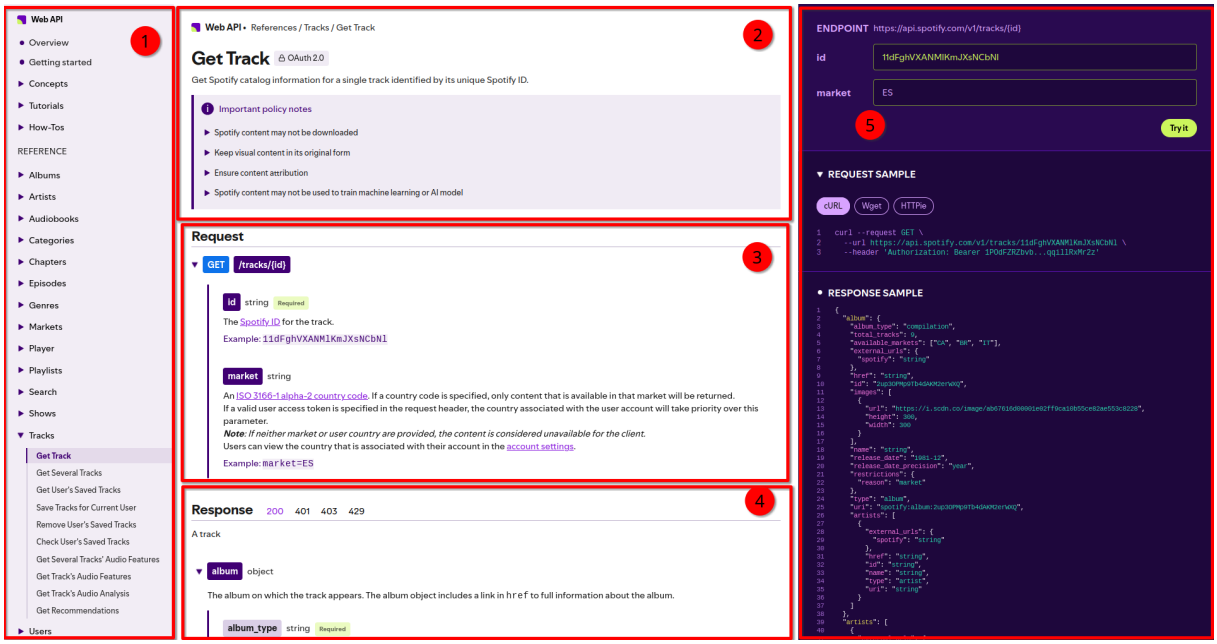

Figure 21: Página do endpoint `Get Track` da API do Spotify.

Os números da imagem representam os seguintes elementos:

1) **Lista de endpoints:** use esta lista para navegar entre endpoints diferentes
2) **Título do endpoint:** explica para que este endpoint serve e outros detalhes
3) **Request:** explica o método, parâmetros e outros detalhes para fazer o Request para este endpoint
4) **Response:** exibe a estrutura dos dados na resposta, bem como as diferentes mensagens de erro e suas causas possíveis.
5) **Painel de exemplos:** mostra na prática um exemplo de Request sendo feito para este endpoint, junto da estrutura da resposta.

Teste com cURL

No painel de exemplos à direita, há um espaço com códigos escritos em `cURL`, `wget` e `HTTPie`. Estes são programas de linha de comando que efetivamente fazem o mesmo que estamos fazendo neste curso com Python!

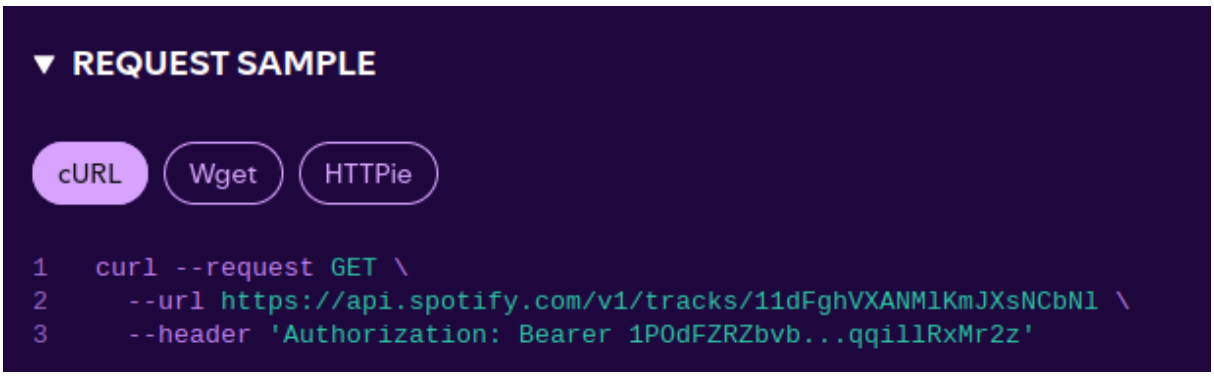

Figure 22: Exemplo de Request em cURL.

Em alguns casos, pode ser mais fácil fazer testes rápidos a uma API usando ferramentas como esta. Mas neste caso específico do Spotify, há um problema: qualquer Request requer a presença de um **token de acesso**.

18. Autenticação Bearer com tokens de acesso

Token de acesso

Um token de acesso funciona da mesma forma que a chave de API, com a diferença (mais ou menos conceitual) de que ele é gerado através de uma chamada à API.

Assim, **são feitas pelo menos duas chamadas à API**: uma para obter o token de acesso, e outra (ou outras) para acessar os dados de interesse.

Obtendo um token de acesso na API do Spotify

Para obtermos um token de acesso no Spotify, precisaremos seguir o Fluxo de Credenciais de Cliente (https://developer.spotify.com/documentation/web-api/tutorials/client-credentials-flow). Esta é uma forma própria do Spotify de organizar o fluxo de obtenção do token.

Com este fluxo, criaremos um "App" dentro do Spotify, que contém um `client_id` e `client_secret`. Estes valores equivalem a usuário e senha, mas ficam vinculados a este "App" e não à nossa conta.

Acompanhe o passo a passo a seguir:

Passo 1: Crie conta no Spotify

Caso não tenha ainda, crie uma conta (gratuita) no Spotify.

Passo 2: Crie um app no Spotify

Acesse https://developer.spotify.com/dashboard e clique em "Criar App".

Em seguida, preencha as informações do seu App. Como é apenas um teste, o nome e descrição não importam muito.

- Para a URI de redirecionamento, pode utilizar `http://localhost:3000`.
- Marque a a caixinha de Web API e os termos de uso ao final!

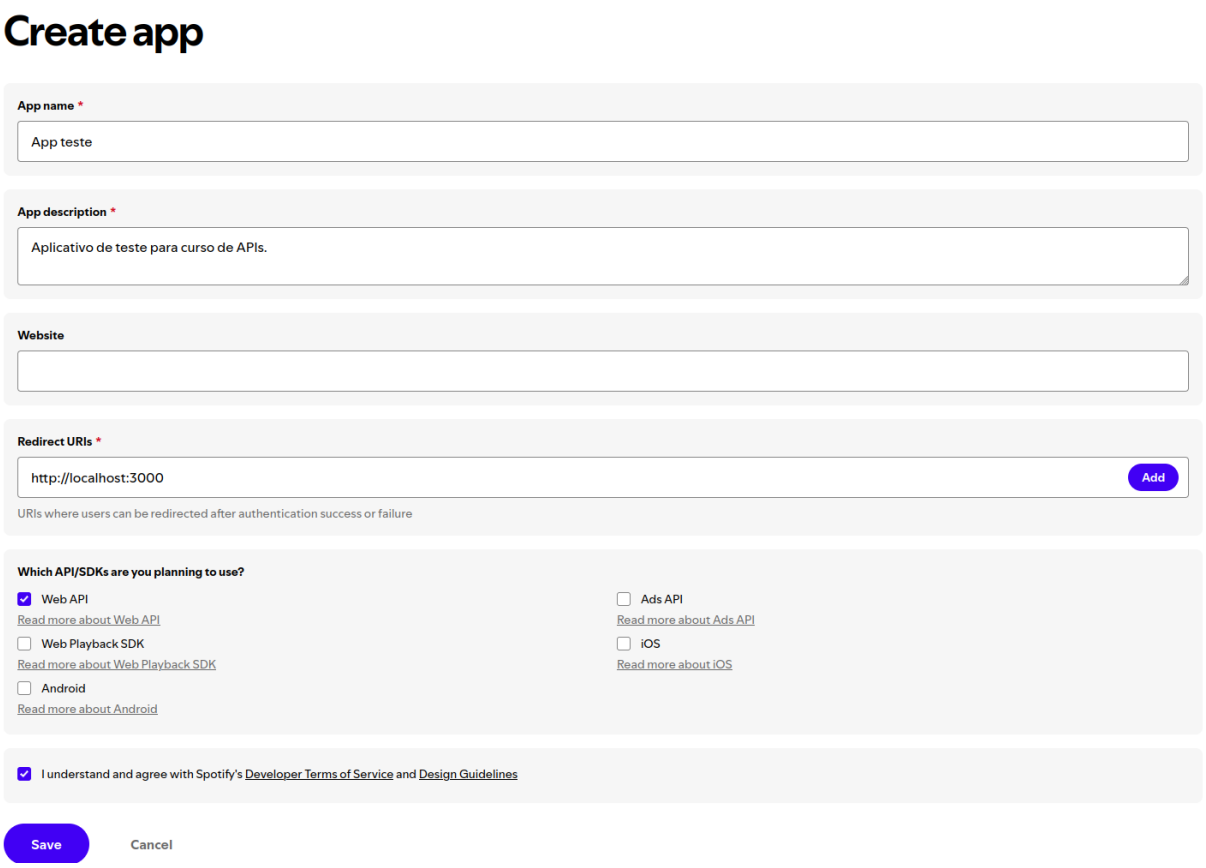

Figure 23: Exemplo de preenchimento para criação de App no Spotify

Passo 3: anote o `client_id` e `client_secret`

Com o App criado, vá para as configurações do App.

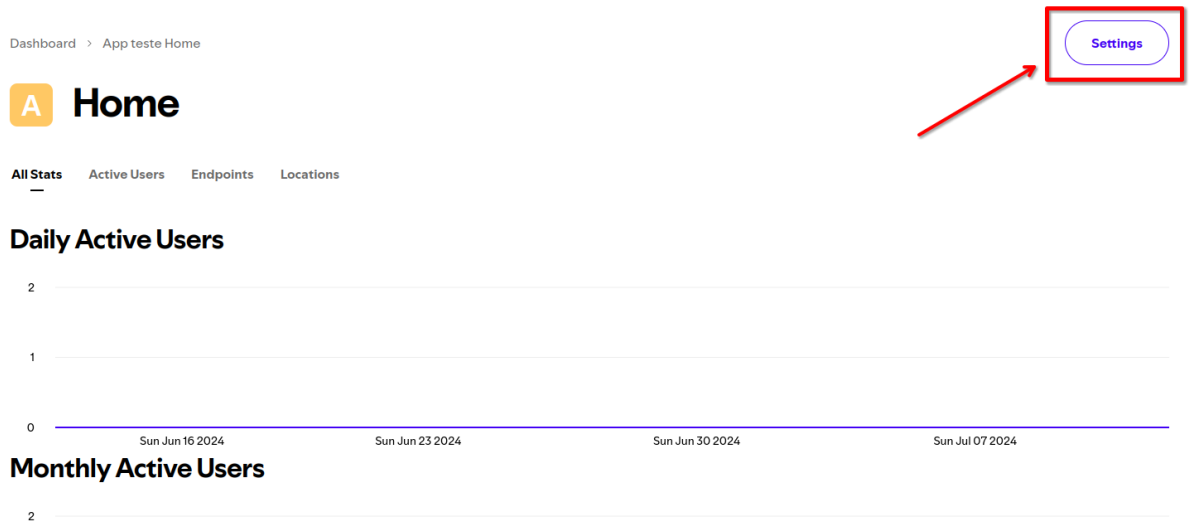

Figure 24: Configurações do seu App recém-criado.

Lá você verá o seu `client_id`, bem como um botão para ver seu `client_secret`. Há a opção de gerar um novo `client_secret` se ele for comprometido.

Guarde estes valores - de preferência, dentro do arquivo `.env`. Eles funcionarão como o usuário e senha!

Autenticação Bearer com token de acesso no Spotify

Agora podemos construir o código responsável por autenticar seu App e gerar um token:

```
import os

import dotenv
import requests
from requests.auth import HTTPBasicAuth

dotenv.load_dotenv(dotenv.find_dotenv())

url = "https://accounts.spotify.com/api/token"

body = {
    'grant_type': 'client_credentials',
}

usuario = os.environ['SPOTIFY_CLIENT_ID']
senha = os.environ['SPOTIFY_CLIENT_SECRET']
auth = HTTPBasicAuth(username=usuario, password=senha)
```

```
resposta = requests.post(url=url, data=body, auth=auth)

try:
    resposta.raise_for_status()
except requests.HTTPError as e:
    print(f"Erro no request: {e}")
    conteudo = None
else:
    print('Token obtido com sucesso!')
    conteudo = resposta.json()

if conteudo:
    print(f'Conteúdo da resposta: {conteudo}')
```

Note o seguinte:

- É preciso usar o método POST e enviar parâmetros adicionais no body do Request. Isto está explícito na documentação: https://developer.spotify.com/documentation/web-api/tutorials/client-credentials-flow
- O token expira em 3600 segundos (uma hora). Nos exemplos da aula, iremos sempre recriar o token fazendo uma nova chamada, mas o ideal seria manter o token em um local simples (ex: arquivo de texto) e só renová-lo quando necessário. Um token expirado causa um erro 401 "Bad or expired token", portanto é relativamente simples descobrir quando é a hora de renová-lo.

Utilizando o token para acessar dados

Agora, podemos usar o token em um novo request. Vamos acessar o artista de ID `246dkjvS1zLTtiykXe5h60` e listar seus atributos, conforme este endpoint: https://developer.spotify.com/documentation/web-api/reference/get-an-artist

```
import os
import sys

import dotenv
import requests
from requests.auth import HTTPBasicAuth

dotenv.load_dotenv(dotenv.find_dotenv())

# Request de autenticação
url = "https://accounts.spotify.com/api/token"

body = {
    'grant_type': 'client_credentials',
}

usuario = os.environ['SPOTIFY_CLIENT_ID']
senha = os.environ['SPOTIFY_CLIENT_SECRET']
```

```
auth = HTTPBasicAuth(username=usuario, password=senha)

resposta = requests.post(url=url, data=body, auth=auth)

try:
    resposta.raise_for_status()
except requests.HTTPError as e:
    print(f"Erro no request: {e}")
    print(resposta.json())
    token = None
else:
    token = resposta.json()['access_token']
    print('Token obtido com sucesso!')

if not token:
    sys.exit()


# Request de busca de dados
id_artista = '246dkjvS1zLTtiykXe5h60'
url = f'https://api.spotify.com/v1/artists/{id_artista}'
headers = {
    'Authorization': f'Bearer {token}'
}

resposta = requests.get(url=url, headers=headers)
print(resposta.json())
```

E o retorno:

```
{'external_urls': {'spotify': 'https://open.spotify.com/artist/246dkjvS1zLTtiykXe5h60'},
 'followers': {'href': None, 'total': 44457707},
 'genres': ['dfw rap', 'melodic rap', 'pop', 'rap'],
 'href': 'https://api.spotify.com/v1/artists/246dkjvS1zLTtiykXe5h60',
 'id': '246dkjvS1zLTtiykXe5h60',
 'images': [{'height': 640,
             'url': 'https://i.scdn.co/image/ab6761610000e5ebe17c0aa1714a03d62b5ce4e0',
             'width': 640},
            {'height': 320,
             'url': 'https://i.scdn.co/image/ab67616100005174e17c0aa1714a03d62b5ce4e0',
             'width': 320},
            {'height': 160,
             'url': 'https://i.scdn.co/image/ab6761610000f178e17c0aa1714a03d62b5ce4e0',
             'width': 160}],
 'name': 'Post Malone',
 'popularity': 90,
 'type': 'artist',
 'uri': 'spotify:artist:246dkjvS1zLTtiykXe5h60'}
```

O ID do artista (Post Malone) foi descoberto a partir de uma busca no Google - é o final da URL do Spotify. Mas poderíamos fazer esta mesma busca pela API. Vamos ver isso em seguida, mas antes vamos abordar alguns outros tópicos.

19. JSON Web Tokens (JWTs) e bibliotecas de APIs

Antes de finalizar o curso, vamos abordar dois outros tópicos importantes no contexto de APIs: **JWTs** e **bibliotecas**.

JSON Web Tokens (JWT)

Na prática, JSON Web Tokens (JWTs) funcionam de forma similar a um token de acesso que vimos na aula anterior. A diferença é que o string "aleatório" do token não é um conteúdo qualquer: ele contém informação JSON codificada em um string.

Para testar e entender isso melhor, podemos utilizar o site https://jwt.io/:

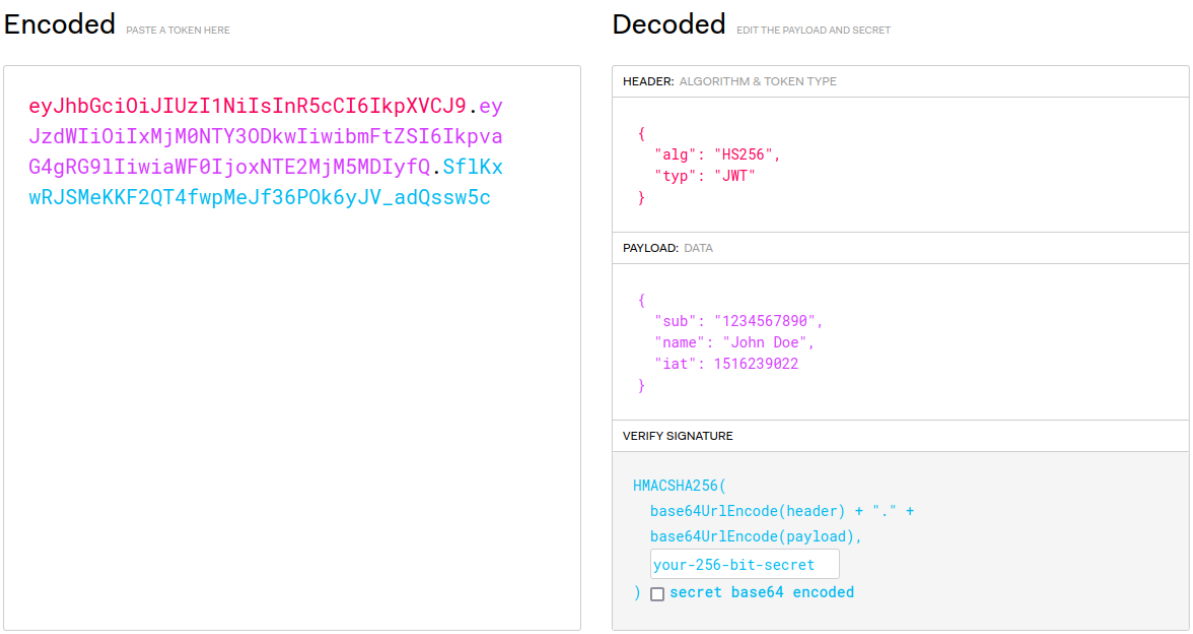

Figure 25: Exemplo de JWT.

O conteúdo do string enviado (à esquerda) pode ser transformando em um JSON com 3 partes (à direita): header, payload, e assinatura. Isso significa que o **próprio token pode conter informação adicional!**

Mas qual a diferença?

Ao enviarmos um token de acesso "comum", o servidor precisa bater o token em um banco de dados interno, para avaliar a qual usuário ele pertence. Em seguida, o servidor terá de entender qual tipo

de acesso aquele usuário possui: é administrador? Há quanto tempo criou conta? Houve alguma alteração?… Para só então decidir se deve ou não liberar o acesso a um recurso.

Esse passo a passo acontece **para cada request**, **com cada usuário**.

Como o **JWT consegue carregar informação adicional junto de si**, a carga no servidor é mais baixa. O servidor pode apenas validar o token usando a assinatura e aceitar qualquer informação contida nele (usuário é administrador, usuário tem acesso, usuário é membro …). Isso faz com que ele seja uma tecnologia **mais escalável**. É claro que a escala onde isso começa a ser relevante é muito grande (Instagram, YouTube, …), mas é considerado o "padrão-ouro" de segurança para autenticação atualmente.

Onde JWTs são usados?

JWTs são muito usados em um fluxo chamado OAuth2, onde um site autoriza o uso de recursos de outro. É o famoso "Logar com Google" / "Logar com Facebook": quando damos acesso a um aplicativo ou site através desse recurso, ocorre uma comunicação entre o servidor do Google/Facebook e o servidor do aplicativo para trocar informações (nome, email, foto do perfil). Tudo isso sem precisar que o usuário envie senha.

Bibliotecas de acesso a APIs

Muitos códigos e funções que criamos ao longo deste curso foram construídos em cima da biblioteca `requests`. A ideia era justamente facilitar ou simplificar o uso de Requests, já que muita estrutura segue o mesmo padrão e poucos detalhes mudam entre um Request e outro.

Se levarmos este conceito um pouco além, chegaremos a alguma **biblioteca de API**. Elas são **formas simplificadas de acessar APIs**, sem que precisemos ficar repetindo estes passos intermediários de construção de Requests "na mão".

No fundo, todas as bibliotecas acabam fazendo os mesmos Requests que fizemos até aqui, mas "escondem" isso de quem as utiliza. O resultado é um fluxo um pouco mais simplificado: em geral, criamos um cliente, passamos algum tipo de credencial a ele, e chamamos alguma função/método que retorne os valores de nosso interesse.

Abaixo, há exemplos de duas bibliotecas. Você consegue identificar os passos que fizemos de autenticação e acesso aos dados? Como eles foram "escondidos"?

`spotipy`: biblioteca de Python para acesso à API do Spotify

https://spotipy.readthedocs.io/en/2.24.0/

`openai`: biblioteca de Python para acesso à API da OpenAI (ChatGPT)

https://platform.openai.com/docs/quickstart?context=python

20. Miniprojeto - Web App com dados do Spotify

Para finalizar o curso, vamos criar mais um webapp que exibe o as principais músicas de um artista do Spotify. Para isso, usarmos a API do Spotify para:

1) Fazer a busca pelo nome do artista;
2) Usar o ID do artista para encontrar as suas músicas mais tocadas;
3) Exibir esta informação em um Web App usando o **Streamlit**.

Resultado

```
import os

import dotenv
import requests
import streamlit as st
from requests.auth import HTTPBasicAuth

dotenv.load_dotenv(dotenv.find_dotenv())


def autenticar():
    url = "https://accounts.spotify.com/api/token"
    body = {
        'grant_type': 'client_credentials',
    }
    usuario = os.environ['SPOTIFY_CLIENT_ID']
    senha = os.environ['SPOTIFY_CLIENT_SECRET']
    auth = HTTPBasicAuth(username=usuario, password=senha)

    resposta = requests.post(url=url, data=body, auth=auth)

    try:
        resposta.raise_for_status()
    except requests.HTTPError as e:
        print(f"Erro no request: {e}")
        token = None
    else:
        token = resposta.json()['access_token']
        print('Token obtido com sucesso!')
    return token


def busca_artista(nome_artista, headers):
    url = "https://api.spotify.com/v1/search"
    params = {
        'q': nome_artista,
        'type': 'artist',
    }
```

```
    resposta = requests.get(url=url, headers=headers, params=params)
    try:
        primeiro_resultado = resposta.json()['artists']['items'][0]
    except IndexError:
        primeiro_resultado = None
    return primeiro_resultado


def busca_top_musicas(id_artista, headers):
    url = f"https://api.spotify.com/v1/artists/{id_artista}/top-tracks"
    resposta = requests.get(url=url, headers=headers)
    musicas = resposta.json()['tracks']
    return musicas


def main():
    # Cabeçalho do Web App
    st.title('Web App Spotify')
    st.write('Dados da API do Spotify (fonte:
↪   https://developer.spotify.com/documentation/web-api)')
    nome_artista = st.text_input('Busque um artista:')
    if not nome_artista:
        st.stop()

    # Autentica no Spotify
    token = autenticar()
    if not token:
        st.stop()
    headers = {
        'Authorization': f'Bearer {token}'
    }

    # Busca pelo artista
    artista = busca_artista(nome_artista=nome_artista, headers=headers)
    if not artista:  # Artista não encontrado
        st.warning(f'Sem dados para o artista {nome_artista}!')
        st.stop()

    # Extrai dados do artista
    id_artista = artista['id']
    nome_artista = artista['name']  # Atualiza para nome "oficial"
    popularidade_artista = artista['popularity']

    # Busca pelas top músicas do artista
    musicas = busca_top_musicas(id_artista=id_artista, headers=headers)

    # Exibe dados no Web App
    st.subheader(f'Artista: {nome_artista} (pop: {popularidade_artista})')
    st.write('Melhores músicas:')
    for musica in musicas:
        nome_musica = musica['name']
        popularidade_musica = musica['popularity']
        link_musica = musica['external_urls']['spotify']
        link_em_markdown = f'[{nome_musica}]({link_musica})'
```

```
        st.markdown(f'{link_em_markdown}: (pop: {popularidade_musica})')


if __name__ == '__main__':
    main()
```