



Material do Aluno

Técnico em Desenvolvimento Web e Cibersegurança

Front-end: princípios

API



**Instituto
Telles**

Parcerias:

SEDUC
Secretaria de Estado
da Educação

SECTI
Secretaria de
Estado de Ciência,
Tecnologia e Inovação

GOVERNO DE
GOIÁS
O ESTADO QUE DÁ CERTO

Sumário

Capítulo 01: API (INTERFACE DE PROGRAMAÇÃO DE APLICAÇÃO)	7
1.1 O que é uma API e qual é a sua função na programação	7
1.1.1. Tipos de APIs	8
1.1.2 Desvendando os segredos das APIs	8
1.2 Autenticação e autorização nas APIs	9
1.2.1 Autenticação: quem está fazendo a solicitação?	9
1.2.2 Autenticação: o que pode ser acessado?	9
1.3 Descobrindo um pouco mais do universo dos aplicativos através das APIs!	11
1.3.1 Ferramentas poderosas para explorar APIs	11
Capítulo 02: EXPLORANDO DOCUMENTO COM API	15
2.1 Documentação da API	15
2.1.1 Métodos	16
2.1.2 Parâmetros	16
2.1.3 Respostas	17
2.2 Como trabalhar com endpoints?	18
2.3 Códigos de status HTTP	19
2.3.1 Manipulação de erros	20
Capítulo 03: REQUISIÇÕES HTTP E MÉTODOS DE API	26
3.1 Funcionamento de APIs (Interfaces de Programação de Aplicações)	26
3.1.1 HTTP (Hypertext Transfer Protocol)	27
Capítulo 04: EXPLORANDO REQUISIÇÕES HTTP COM CURL, POSTMAN E UMA API SIMPLES	34
4.1 Requisições HTTP	35
4.1.1 Elementos de uma Requisição HTTP	35
4.2 Explorando as ferramentas como cURL e Postman	35
4.2.1 cURL	36
4.2.2 Postman	36
4.3 Funções do cURL e do Postman na análise das respostas HTTP	37
4.3.1 Como fazer uma requisição HTTP a uma API com PokéAPI	39
4.3.2 Fazendo requisições à pokémon API	39
4.3.3 Explorando a Pokémon API na prática	40

Capítulo 05: FORMATOS DE DADOS EM APIs	45
5.1 JSON (JavaScript Object Notation) e XML (eXtensible Markup Language)	45
5.2 Estrutura básica do XML e do JSON	46
5.2.1 Estrutura básica XML	46
5.2.2 Aplicando a estrutura	47
5.2.3 Estrutura básica JSON	47
5.2.4 Utilização do JSON em transferência de dados	49
5.2.5 Papel do JSON em requisições AJAX e interações web	49
5.2.6 Diferenças entre JSON e XML: estrutura e aplicações	49
5.2.7 Usos comuns do JSON	50
5.2.8 Usos comuns do XML	51
5.3 JSON estruturando dados	53
5.3.1 JSON Schema: definindo estrutura e validando dados JSON	53
5.3.2 JSON Web Tokens (JWT): compactação e autossuficiência nas trocas de informações	54
5.3.3 Ferramentas de manipulação de JSON: facilitando a interação com dados JSON	55
5.4 XML e suas variantes	56
5.4.1 Outras variantes do XML	57
Capítulo 06: FORMATOS DE DADOS EM APIs	61
6.1 Desvendando o parsing de dados JSON e XML	62
6.1.1 Parsing de JSON em JavaScript	62
6.1.2 Exemplos de parsing de JSON em JavaScript	63
6.2 Manipulando dados JSON com JavaScript	65
6.3 Manipulando dados JSON com Python	66
Capítulo 07: TIPOS DE AUTENTICAÇÃO EM APIs:	
CHAVES DE API, TOKENS, OAUTH 2.0 E AUTENTICAÇÃO DE TERCEIROS	73
7.1 Chaves de API	73
7.2 Práticas de segurança	74
7.2.1 Tokens	76
7.3 OAuth 2.0	79
7.4 Autenticação de terceiros	82
Capítulo 08: CONSIDERAÇÕES DE SEGURANÇA AO ACESSAR APIs EXTERNAS E AUTENTICAÇÃO COM CHAVES DE	87
8.1 Considerações de segurança ao acessar APIs externas	87
8.1.1 HTTPS (SSL/TLS)	87
8.1.2 Validação de entradas	89

8.1.3 Rate limiting	91
8.1.4 CORS (Cross-Origin Resource Sharing)	92
8.1.5 Monitoramento e logging	94
8.1.6 Autenticação e autorização	96
8.2 Exemplo de autenticação em uma API com chaves de API	97
Capítulo 09: UTILIZANDO A FUNÇÃO FETCH() PARA REQUISIÇÕES A APIs EM JAVASCRIPT: MANIPULAÇÃO DE RESPOSTAS E TRATAMENTO DE ERROS	104
9.1 Utilizando a função fetch() para requisições a APIs	105
9.2 Manipulação de respostas de API	106
9.3 Tratamento de erros	109
Capítulo 10: EXIBIÇÃO DE DADOS DA API EM UMA PÁGINA WEB: CRIANDO UMA APLICAÇÃO QUE CONSUME DADOS	122
10.1 Fundamentos da exibição de dados da API	123
Capítulo 11: EXPLORANDO O MUNDO DAS APIs: ESTUDOS DE CASO, INTEGRAÇÃO PRÁTICA E DESENVOLVIMENTO DE PROJETOS	134
11.1 Aplicativos dependentes de múltiplas APIs	135
11.2 Integração de APIs em um projeto	136
11.3 Desenvolvendo um projeto que utiliza APIs externas	142
Referências	151

Front-end: princípios

INTRODUÇÃO

O desenvolvimento *front-end* é super importante na criação de interfaces interativas e envolventes para as pessoas que usam a internet. Afinal, ele cuida da parte visual e funcional de um *site*, em que a gente faz tudo ficar bonito e funcionar direitinho para o usuário. No meio disso tudo, nós temos os princípios do *front-end*, que vão desde organizar o código **HTML** até a estilização do site com o **CSS** e a interatividade habilitada pelo **JavaScript**.

Quando a gente fala sobre esses princípios do *front-end*, é essencial lembrar que precisamos garantir três aspectos: **acessibilidade**, **usabilidade** e **responsividade**. Acessibilidade é garantir que todas as pessoas consigam usar o *site*, mesmo que tenham alguma limitação. A usabilidade é deixar a página *web* fácil de entender e navegar. Já a responsividade significa fazer com que o *site* se adapte bem em diferentes tipos e tamanhos de telas, como celulares, *tablets* e computadores.

No mundo do *front-end*, nós também trabalhamos bastante com algo chamado **APIs (Application Programming Interface, ou Interfaces de Programação de Aplicações)**, que são como pontes que permitem a integração de dados dinâmicos e uma comunicação eficiente entre o *front-end* e o *back-end* de uma aplicação *web*. As APIs ajudam a trazer informações em tempo real, deixando as coisas mais dinâmicas para quem está usando o *site*. Aqui, nós vamos entender como os princípios do *front-end* se entrelaçam com o uso estratégico de APIs para criar *websites* modernos e eficientes.

CAPÍTULO 01

API (INTERFACE DE PROGRAMAÇÃO DE APLICAÇÃO)

O que esperar deste capítulo:

- Compreender o que é uma API;
- Entender as práticas de autenticação e autorização nas APIs;
- Conhecer algumas ferramentas importantes para explorar APIs;

1.1 O que é uma API e qual é a sua função na programação

API é a sigla em inglês para *Application Programming Interface*, que significa Interface de Programação de Aplicações em português. Ela se refere a um conjunto de padrões que fazem parte de uma interface. Veja o esquema abaixo para entender as suas funções.



1.1.1 Tipos de APIs

✓ APIs públicas

Também conhecidas como APIs externas ou abertas, essas APIs são disponibilizadas para desenvolvedores, empresas e o público por firmas, organizações ou plataformas.

✓ APIs internas

Também chamadas de APIs privadas, elas são desenvolvidas para serem usadas dentro de uma organização.

✓ APIs de parceiros

Essa API é compartilhada com parceiros de negócios específicos.

✓ APIs compostas

Permitem executar várias solicitações de API em uma única chamada.

1.1.2 Desvendando os segredos das APIs

Vamos analisar a imagem a seguir:



- As APIs servem como intermediárias entre o usuário ou aplicativo e o servidor;
- Elas funcionam como uma ponte na comunicação, ou seja: quando você faz uma solicitação de API, ela leva essa solicitação ao servidor e, depois, retorna a resposta do servidor para você;
- Nesse processo, nós temos dois aspectos muito importantes:
 - a **autenticação**, que verifica quem está fazendo o pedido;
 - a **autorização**, que determina o que o solicitante tem permissão para acessar.

1.2 Autenticação e autorização nas APIs

1.2.1 Autenticação: quem está fazendo a solicitação?

A autenticação é o processo que responde à pergunta: "Quem é você?". Ou seja, essa é a etapa que **verifica a identidade de um usuário ou aplicativo que está tentando acessar a API**.

Qual é o objetivo?	Garantir que apenas aqueles que possuem as credentials corretas possam acessar os recursos
Como isso pode ser feito?	Por meio do uso de credenciais como nome de usuário e senha .
	Por meio da apresentação de um certificado digital
	Utilizando tecnologias mais avançadas, como o login único (SSO) .

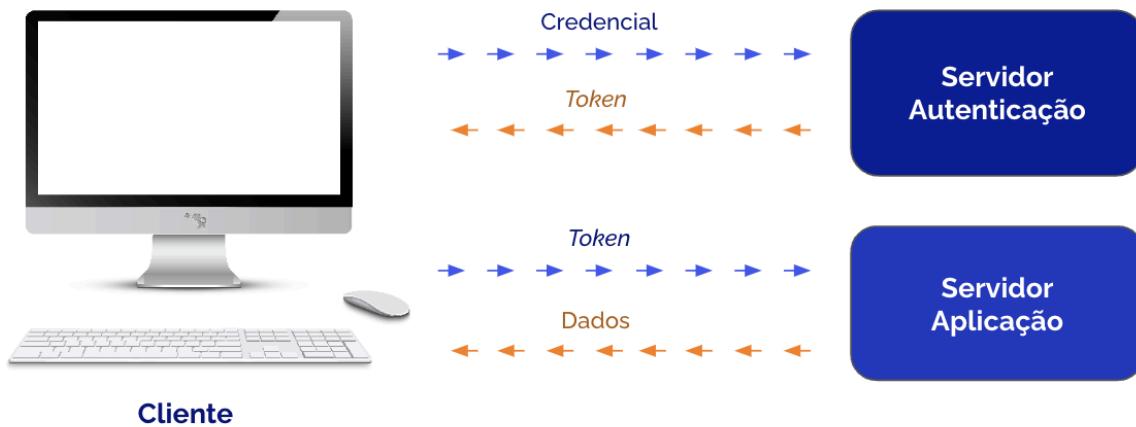
1.2.2 Autenticação: o que pode ser acessado?

Após identificar quem está tentando acessar, é hora de decidir **o que essa pessoa ou aplicativo tem permissão para fazer**. Um protocolo muito utilizado para isso é o **OAuth 2.0**, que funciona como um **crachá digital** que concede acesso aos recursos apropriados.

OAuth 2.0: o mestre dos tokens

O OAuth 2.0 é o protagonista quando se trata de autorização, fornecendo uma estrutura padrão e segura para proteger o acesso a recursos, como as adoradas APIs *web*. Ele funciona como uma dança bem coreografada, em que os aplicativos solicitam e recebem *tokens* de acesso, garantindo uma interação segura e eficiente entre sistemas.

Aplicação prática do OAuth 2.0



Imagine um aplicativo de mídia social usando OAuth 2.0: ao fazer *login*, você **valida a sua identidade (autenticação)** e, quando permite que um aplicativo de terceiros poste em seu nome, você está concedendo **permissão (autorização)** por meio de tokens OAuth. Isso mantém seu perfil seguro e te dá controle sobre quem pode fazer o quê.

Observe a seguir o passo a passo desse processo, considerando um usuário do X (antigo Twitter):

- 1** **Autenticação:** o aplicativo envia suas credenciais (chave da API e segredo da API) para a API do X (antigo Twitter).
- 2** **Token de acesso:** se as credenciais forem válidas, a API retorna um *token* de acesso.
- 3** **Solicitação de autorização:** o aplicativo direciona o usuário para uma página de *login* do X (antigo Twitter) em que o usuário insere suas credenciais.
- 4** **Concessão de autorização:** depois que o usuário faz login, o X (antigo Twitter) pede ao usuário para autorizar o aplicativo a acessar sua conta.
- 5** **Código de autorização:** se o usuário concordar, o site redireciona o usuário de volta para o aplicativo com um código de autorização.
- 6** **Postando um tweet:** o aplicativo usa o *token* de acesso para fazer solicitações à API do X em nome do usuário.

1.3 Descobrindo um pouco mais do universo dos aplicativos através das APIs!

As APIs estão presentes em muitos aplicativos populares que usamos no dia a dia. Um exemplo é o **Google Maps**, que disponibiliza uma API para que desenvolvedores possam incorporar mapas e outras funcionalidades nos seus próprios aplicativos.

Vamos conhecer outros aplicativos que utilizam APIs!



O **Instagram**, conhecido por suas fotos e histórias, integra APIs para permitir que outros aplicativos interajam com as suas funcionalidades. Por exemplo, **aplicativos de edição de fotos** podem usar a API do Instagram para compartilhar diretamente as imagens que você criou na plataforma.



O **Spotify**, um favorito entre os amantes da música, utiliza APIs para proporcionar uma experiência integrada e personalizada. Graças às APIs, uma variedade de aplicativos e dispositivos, como **alto-falantes inteligentes e sistemas de entretenimento**, podem se conectar ao Spotify. Isso permite que você desfrute de suas músicas favoritas onde quer que esteja.



O **WhatsApp**, nosso fiel companheiro de mensagens diárias, utiliza APIs para facilitar a comunicação entre usuários e serviços externos. Por exemplo, **serviços de tradução** podem se integrar ao WhatsApp através de APIs, permitindo que você traduza mensagens instantaneamente, sem sair do aplicativo.



O **Uber**, pioneiro no transporte urbano, utiliza APIs para proporcionar uma experiência de viagem suave e personalizada. A **integração com mapas, métodos de pagamento** e, até mesmo, serviços de terceiros, como **aplicativos de música**, é possível graças às APIs. Isso garante uma viagem confortável e personalizada ao seu gosto.



Aplicativos de **previsão do tempo**, como o Weather Channel, utilizam APIs para **fornecer informações precisas e atualizadas sobre as condições climáticas**. Eles se conectam a serviços meteorológicos externos por meio de APIs. Sem elas, teríamos que confiar no bom e velho método de olhar pela janela para saber se vai chover!

Portanto, da próxima vez que você estiver curtindo o seu aplicativo favorito ou jogando aquele *game* incrível, lembre-se que, por trás dessas experiências, há APIs trabalhando silenciosamente, conectando tudo para tornar a sua jornada digital mais emocionante.

1.3.1 Ferramentas poderosas para explorar APIs

No mundo da programação, trabalhar com APIs é uma tarefa comum e essencial. Para facilitar esse trabalho, os desenvolvedores contam com uma variedade de ferramentas poderosas, cada uma delas com as suas próprias características e vantagens.

Na tabela a seguir, você terá uma visão geral de três ferramentas populares usadas para trabalhar com APIs: **Postman**, **cURL** e **Insomnia**. Vamos explorar o que cada uma dessas ferramentas oferece e por que você pode querer usá-las em seus projetos.

	Postman	cURL	Insomnia
Descrição	Uma plataforma completa que permite aos desenvolvedores testar, desenvolver e documentar APIs de maneira eficiente.	Uma ferramenta de linha de comando usada para fazer requisições HTTP.	Um cliente RESTful que oferece recursos avançados para testar APIs.
Função	Facilita a criação de solicitações HTTP, a realização de testes automatizados e a visualização de respostas de API em um formato fácil de ler.	Ótimo para a automação e <i>scripts</i> , pois permite enviar requisições diretamente do terminal.	Interface amigável, suporte a testes de GraphQL e possui a capacidade de gerenciar variáveis de ambiente.
Destaques	Interface intuitiva, suporte a coleções para organizar requisições e capacidade de trabalhar com diferentes ambientes.	É simples e flexível, suportando uma ampla gama de protocolos e é capaz de lidar com <i>cookies</i> , autenticação e muito mais.	Possui um <i>design</i> bonito e intuitivo, além de oferecer recursos avançados para desenvolvedores, como a capacidade de gerar código para mais de trinta linguagens de programação.



Nesta jornada, nós mergulhamos no universo das Interfaces de Programação de Aplicações (APIs), que são os pilares da comunicação entre diferentes sistemas de software. Conhecemos os conceitos fundamentais de autenticação e autorização, compreendendo que a autenticação é o processo de confirmação da identidade de um usuário ou aplicativo, enquanto a autorização determina quais recursos um usuário ou aplicativo autenticado pode acessar.

Por fim, nos familiarizamos com as funções de APIs utilizados em diversos aplicativos presentes no nosso dia a dia e exploramos ferramentas poderosas para trabalhar com APIs, como Postman, cURL e Insomnia.



ATIVIDADE DE FIXAÇÃO

1. O que é uma API e por que ela é importante no desenvolvimento de *software*?
2. Explique a diferença entre autenticação e autorização.
3. Descreva o que é o protocolo OAuth 2.0.
4. Como o protocolo OAuth 2.0 concede acesso a recursos específicos?
5. Como você usaria o Postman para enviar uma solicitação **GET** para uma API?
6. Escreva um comando cURL para enviar uma solicitação **POST** para uma API.
7. Descreva um cenário em que você usaria o Insomnia em vez do Postman ou cURL.
8. Imagine que você está desenvolvendo um aplicativo que precisa postar *tweets* em nome de um usuário. Descreva o processo de autenticação e autorização que você seguiria para acessar a API do X (antigo Twitter).
9. Descreva um cenário em que você precisaria testar uma API. Quais ferramentas você usaria? Por quê?
10. Quais são algumas práticas recomendadas para garantir a segurança ao trabalhar com APIs?



DESAFIO PRÁTICO

API (Interface de Programação de Aplicação)

Desafio: Desenvolvendo uma lista de tarefas simples que se integra com a API do Google Tasks.



Descrição

Neste desafio, você será desafiado a aplicar os seus conhecimentos sobre APIs, autenticação e autorização para desenvolver uma lista de tarefas simples. O aplicativo deve permitir que os usuários criem, visualizem e excluam tarefas usando a API do Google Tasks.



Objetivos

- Compreender o funcionamento das APIs e como elas permitem a comunicação entre diferentes sistemas de *software*;
- Aprender sobre os conceitos de autenticação e autorização e como eles são aplicados ao acessar APIs;
- Ganhar experiência prática no uso de ferramentas de API, como Postman, cURL e Insomnia;
- Desenvolver uma lista de tarefas simples que se integra com a API do Google Tasks.



Orientações

- Comece revisando o que são APIs e como elas funcionam;
- Estude os conceitos de autenticação e autorização. Entenda como eles se aplicam ao acesso às APIs;
- Use ferramentas como Postman, cURL e Insomnia para explorar a API do Google Tasks. Familiarize-se com a criação de solicitações e a visualização de respostas^[10] ^[11] ;
- Crie um relatório explicativo sobre a utilização dessas ferramentas na construção e implantação da API junto ao Google Tasks. Lembre-se de implementar a autenticação e a autorização corretamente garantindo que o aplicativo possa acessar a API do Google Tasks de maneira segura e eficaz.



CAPÍTULO 02

EXPLORANDO DOCUMENTO COM API

O que esperar deste capítulo:

- Compreender sobre a documentação da API;
- Trabalhar com *endpoints*;
- Interpretar corretamente os códigos de *status* HTTP.

2.1 Documentação da API

No vasto mundo da tecnologia, a inovação e a criatividade se juntam para criar soluções incríveis que transformam o nosso dia a dia. Neste mundo, a **documentação da API serve como um cardápio digital**, revelando os segredos e as maravilhas que uma aplicação pode oferecer.

Neste capítulo, nós vamos conhecer elementos essenciais que são documentados dentro de uma API para fornecer informações detalhadas sobre como interagir com ela.



Endpoints

- Cada *endpoint* representa uma URL específico que a API disponibiliza para que você faça as suas solicitações;
- Os *endpoints* são pontos de acesso específicos dentro de uma API, permitindo aos desenvolvedores interagir com os recursos disponíveis.



A documentação da API lista e descreve cada *endpoint* disponível, fornecendo informações que detalham seu propósito, a URL correspondente, como acessá-los e quais recursos estão disponíveis em cada um.

2.1.1 Métodos

- Os métodos são as diferentes maneiras que temos de interagir com o cardápio digital. Ou seja, são os **tipos de solicitações que você pode fazer a um endpoint**;
- Os métodos HTTP são usados para indicar a ação que o cliente deseja realizar em um recurso específico. Eles são:
 - **GET**: utilizado quando você deseja **buscar** informações;
 - **POST**: utilizado quando você deseja **enviar** informações para o servidor;
 - **PUT**: utilizado quando você deseja **atualizar** informações;
 - **DELETE**: utilizado quando você deseja **deletar** informações.



A documentação da API informa quais métodos são suportados em cada *endpoint* e as operações que podem ser realizadas com cada um deles.

2.1.2 Parâmetros

- Os parâmetros são informações adicionais que podem ser incluídas em uma solicitação a fim de personalizar a operação desejada;
- Eles ajustam a sua interação com a API de acordo com suas preferências digitais.



A documentação da API lista os parâmetros disponíveis para cada *endpoint*, juntamente com a sua descrição e o seu uso correto.

Alguns parâmetros possíveis são:

Nome do parâmetro	Tipo	Descrição
<code>id</code>	Inteiro	O identificador único do recurso.
<code>nome</code>	<i>String</i>	O nome do recurso.
<code>dataCriacao</code>	Data	A data em que o recurso foi criado.
<code>ativo</code>	Booleano	Um sinalizador indicando se o recurso está ativo.

2.1.3 Respostas

- As respostas fornecem uma visão antecipada do que você pode esperar quando a sua solicitação for atendida;
- Isso pode incluir dados solicitados, códigos de *status* HTTP e informações adicionais;
- Geralmente, a documentação da API fornece exemplos de respostas para que você saiba o que esperar.



Geralmente, a documentação da API fornece exemplos de respostas para que você saiba o que esperar. Ela pode descrever os tipos de respostas que podem ser recebidos, juntamente com os códigos de *status* possíveis e os seus significados.

De maneira geral, a documentação da API atua como um guia completo que ajuda os desenvolvedores a entenderem como interagir com a API, incluindo quais **endpoints** estão disponíveis, como usar os **métodos** corretos, quais **parâmetros** podem ser utilizados e quais **respostas** esperar em retorno.



Vamos revisar tudo que aprendemos até aqui?

Termo	Descrição	Exemplo
Endpoint	Uma URL específica que a API disponibiliza para que você faça suas solicitações.	https://api.exemplo.com/usuarios
Método	O tipo de solicitação que você pode fazer a um endpoint.	GET, POST, PUT, DELETE.
Parâmetros	Informações adicionais que você pode enviar com sua solicitação.	?nome=João
Resposta	O que você pode esperar receber de volta da API.	{ "nome": "João", "idade": 25 }

2.2 Como trabalhar com endpoints?

Suponha que temos uma API para um serviço de filmes e um dos seus *endpoints* é:
<https://api.servicodofilme.com/filmes>



GET: se você quiser obter uma lista de todos os filmes, você pode fazer uma solicitação **GET** para <https://api.servicodofilme.com/filmes>. Isso retornará uma lista de filmes disponíveis.

```
GET /filmes HTTP/1.1
Host: api.servicodofilme.com
```



POST: se você quiser adicionar um novo filme, você pode fazer uma solicitação **POST** para <https://api.servicodofilme.com/filmes>, com os detalhes do filme no corpo da solicitação.

```
POST /filmes HTTP/1.1
Host: api.servicodofilme.com
Content-Type: application/json
{
  "titulo": "Novo Filme",
  "diretor": "Diretor",
  "ano": 2024
}
```



GET com parâmetros: Se você quiser obter detalhes de um filme específico, você pode adicionar o ID do filme ao *endpoint*, como <https://api.servicodofilme.com/filmes/123>.

```
GET /filmes/123 HTTP/1.1
```

```
Host: api.servicodofilme.com
```

2.3 Códigos de status HTTP

Em nossa fascinante jornada pelo vasto universo das APIs, os códigos de *status* HTTP assumem o papel de semáforos digitais, nos orientando pelo caminho das solicitações e respostas. Esses códigos indicam o estado preciso da interação entre nossos comandos e a API.



Aqui, vamos te ajudar a ser um verdadeiro detetive digital e desvendar o significado dos códigos mais comuns.

Código de status	Nome do código	Ele significa
200	OK	A solicitação foi bem-sucedida. Tudo correu conforme o planejado e você está pronto para a próxima etapa da jornada.
201	<i>Created</i>	Além do sucesso, um novo recurso foi criado como resultado da sua solicitação.
400	<i>Bad Request</i>	A API não conseguiu entender o que você pediu. Podem ter faltado parâmetros essenciais ou houve algum mal-entendido na requisição.
401	<i>Unauthorized</i>	Para prosseguir, a API exige que você se autentique. Sem as credenciais certas, a entrada está vedada.
403	<i>Forbidden</i>	Você se autenticou, mas, por algum motivo, não tem permissão para acessar o recurso solicitado.
404	<i>Not Found</i>	O recurso que você está procurando simplesmente não está lá. Pode ser que você tenha digitado o <i>endpoint</i> errado ou que o recurso tenha sido removido.
500	<i>Internal Server Error</i>	Algo deu errado no lado da API. Não é culpa sua; algo falhou internamente no servidor.

2.3.1 Manipulação de erros

Ao navegar pelo labirinto das solicitações e das respostas das APIs, podemos encontrar obstáculos. Quando um contratempo ocorre durante uma solicitação, a API age como um guia experiente e não nos deixa no escuro. Em vez disso, ela responde com um **código de status de erro e uma mensagem explicativa no corpo da resposta**.

Como já vimos, ao tentar acessar um recurso ausente, a API pode retornar um **status 404** com a mensagem “**Recurso não encontrado**”. Por isso, é importante sempre prestar atenção a esses códigos e mensagens para entender precisamente o que ocorreu com a sua solicitação.

Exemplo prático

A partir de agora, vamos mostrar como você pode observar os **status de erros**, **erros específicos** e **analisar as respostas de erros da API**. Fique "ligado"!

Verificando o código de *status* da resposta

```
import requests
resposta = requests.get('https://api.exemplo.com/recurso')
# Verifique o código de status
if resposta.status_code != 200:
    print(f'Erro ao acessar o recurso: {resposta.status_code}')
else:
    print('Recurso acessado com sucesso!')
```

Neste exemplo, estamos fazendo uma solicitação **GET** para um recurso na API. Se o código de *status* da resposta não for 200 (que indica sucesso), imprimimos uma mensagem de erro.

Lidando com erros específicos

```
import requests

try:
    resposta = requests.get('https://api.exemplo.com/recurso')
    resposta.raise_for_status()
except requests.exceptions.HTTPError as err:
    print(f'Erro HTTP: {err}')
except requests.exceptions.ConnectionError as err:
    print(f'Erro de conexão: {err}')
except requests.exceptions.Timeout as err:
    print(f'Erro de tempo limite: {err}')
except requests.exceptions.RequestException as err:
    print(f'Erro: {err}')
```

Neste exemplo, estamos usando um bloco `try/except` para lidar com diferentes tipos de erros que podem ocorrer durante a solicitação. Isso nos permite fornecer mensagens de erro mais úteis e específicas.

Analisando a resposta de erro da API

Muitas APIs incluirão informações adicionais sobre o erro no corpo da resposta. Você pode precisar analisar essa resposta para obter mais detalhes.

```
import requests
import json

resposta = requests.get('https://api.exemplo.com/recurso')

if resposta.status_code != 200:
    erro = json.loads(resposta.text)
    print(f'Erro ao acessar o recurso: {erro["mensagem"]}')
```

Neste exemplo, se a solicitação falhar, estamos analisando o corpo da resposta como JSON e imprimindo a mensagem de erro.

Estratégias de paginação

Quando uma API tem muitos dados para retornar, em vez de nos inundar com todos os dados disponíveis de uma só vez, ela divide os dados em "páginas". Cada página é como uma ilha de dados, oferecendo um subconjunto gerenciável dos dados totais.

"Se liga" no exemplo

Imagine que estamos navegando pela API de um serviço de *streaming* de filmes que armazena 1000 filmes. Em vez de nos enviar todos os filmes de uma vez, a API pode optar por nos enviar 50 filmes por página. Isso significa que precisaríamos fazer 20 solicitações para visitar todas as "ilhas" e coletar todos os 1000 filmes.



A documentação da API é como um mapa de navegação, explicando como podemos navegar de uma página para outra. Normalmente, nós adicionamos parâmetros às nossas solicitações para indicar qual página queremos explorar.

"Se liga" no exemplo

Por exemplo, se quisermos solicitar a segunda página de filmes, faríamos uma solicitação **GET** para o *endpoint* /filmes, adicionando um parâmetro de consulta **page** com o valor 2. Isso ficaria assim:

```
GET /filmes?page=2 HTTP/1.1  
Host: api.servicodestreaming.com
```

Neste exemplo, estamos dizendo à API que queremos a segunda página de filmes. A API então nos retornará os próximos 50 filmes em sua lista.



Agora, nós vamos conhecer algumas das estratégias de paginação

Aspectos	Descrição
Tipos de paginação.	Existem diferentes tipos de paginação que uma API pode implementar, como uma paginação baseada em números de página , em cursor ou em token .
Parâmetros de paginação.	Algumas APIs também permitem que você especifique o número de itens por página através de um parâmetro de limit ou size .
Links de paginação.	Algumas APIs incluem <i>links</i> de paginação na resposta que apontam diretamente para a próxima , anterior , primeira e última página.
Meta-informações de paginação.	Algumas APIs fornecem meta-informações sobre a paginação na resposta, como o número total de itens , o número total de páginas , o número da página atual etc.
Considerações de desempenho.	A paginação pode ter um impacto significativo no desempenho da API. Retornar muitos itens de uma vez pode sobrecarregar o servidor ou o cliente.
Ordenação.	Muitas vezes, a paginação é usada em conjunto com a ordenação. A ordenação determina a ordem dos itens na resposta .



RESUMO

Neste capítulo, nós discutimos o conceito da documentação da API, comparando-a a um cardápio digital que detalha os recursos disponíveis. Também abordamos os elementos essenciais, como os *endpoints*, os métodos, os parâmetros e as respostas, que oferecem uma visão prévia do que se pode esperar ao fazer uma solicitação.

Além disso, exploramos o uso dos códigos de *status* HTTP pelas APIs para indicar o sucesso ou falha de uma requisição, assim como a forma como uma API reage diante de erros. Destacamos, também, a importância da paginação quando uma API tem muitos dados para retornar e que, nesse caso, os dados geralmente são divididos em páginas e a documentação da API deve explicar como navegar de uma página para outra.



ATIVIDADE DE FIXAÇÃO

1. Encontre a documentação para a API do X (antigo Twitter) e liste cinco *endpoints* que ela oferece.
2. Para cada *endpoint* listado no exercício anterior, identifique os métodos HTTP que ele suporta e quaisquer parâmetros que ele aceite.
3. O que os códigos de *status* HTTP **200, 201, 400, 401, 403 e 500** indicam?
4. Suponha que você fez uma solicitação **GET** para um *endpoint* de uma API e recebeu um *status* 404. O que isso significa? Como você lidaria com isso?
5. Suponha que você está trabalhando com uma API que retorna cem itens por página. Se você quiser obter os itens de 150 a 200, qual página você solicitaria?
6. Escreva um código em Python (ou em qualquer outra linguagem de programação de sua escolha) para fazer uma solicitação **GET** para um *endpoint* de uma API.

- 7.** Suponha que você fez uma solicitação para uma API e recebeu a seguinte resposta: **{"nome": "João", "idade": 30}**. O que isso significa?
- 8.** Pesquise sobre autenticação de API. Quais são algumas maneiras comuns de autenticar solicitações de API?
- 9.** Encontre uma API pública interessante na Internet. Liste alguns dos *endpoints* que ela oferece e o que eles fazem.
- 10.** Escolha uma API pública, escreva um pequeno programa que faça uma solicitação para essa API e imprima a resposta.

CAPÍTULO 03

REQUISIÇÕES HTTP E MÉTODOS DE API

O que esperar deste capítulo:

- Adquirir uma visão geral sobre o protocolo HTTP;
- Conhecer métodos HTTP comuns, como **GET, POST, PUT e DELETE**.

Hoje em dia, estamos rodeados por tecnologia, desde os nossos *smartphones* até os dispositivos inteligentes nas nossas casas. Porém, você já parou para pensar em **como esses diferentes aparelhos se comunicam entre si?** É aí que entram as APIs, ou as **Interfaces de Programação de Aplicativos**. Mesmo que o termo possa parecer complicado à primeira vista, entender o que são APIs e como elas funcionam pode abrir as portas para um mundo de possibilidades na criação e interação com tecnologia.

Neste capítulo, vamos explorar como as APIs funcionam, quais são seus componentes principais e como elas permitem que diferentes programas interajam entre si de maneira eficiente e segura. Vamos lá?

3.1 Funcionamento de APIs (Interfaces de Programação de Aplicações)

A API é como um garçom em um restaurante. Imagine que você está em um restaurante e quer pedir uma refeição.

Você não vai diretamente à cozinha e faz o pedido ao chef, certo?

Em vez disso, você dá o seu pedido ao garçom, que, em seguida, leva o seu pedido para a cozinha e traz a refeição de volta para você.



No mundo da programação, as APIs funcionam de maneira semelhante. Quando um programa precisa de dados ou funcionalidades de outro programa, ele não acessa esses dados ou funcionalidades diretamente. Em vez disso, ele **faz um pedido à API do outro programa, que retorna os dados ou realiza a funcionalidade solicitada**.



Você deve estar pensando “**Como esse processo acontece?**”. Para isso, as APIs usam o **HTTP**, que permite que os computadores troquem informações na internet.

3.1.1 HTTP (*Hypertext Transfer Protocol*)

O HTTP é o protocolo padrão para comunicação na *web*, sendo amplamente utilizado para **transmitir dados** entre clientes (como navegadores ou aplicativos) e servidores. Ou seja, ele é o como o idioma que as APIs usam para se entenderem.

Assim, as APIs funcionam como as **regras** que dizem como essa comunicação acontece, incluindo onde procurar (*endpoints*), que tipo de ação fazer (métodos) e como os dados são organizados (formatos de dados).



Agora que sabemos a importância que o HTTP tem para as APIs, é importante entender como o HTTP funciona e transmite informações.

Principais componentes de um sistema HTTP

- **Cliente (agente-usuário):** é qualquer ferramenta que age em nome do usuário. O cliente pode ser:
 - um navegador da *web*, como Chrome e o Mozilla Firefox;
 - um aplicativo móvel;
 - um dispositivo qualquer que envie requisições HTTP, incluindo robôs de busca que exploram a *web* para indexação.
- **Servidor:** é o computador que hospeda os recursos ou serviços solicitados pelos clientes. Ele recebe as requisições deles, processando-as e enviando de volta as respostas apropriadas;
- **Protocolo HTTP:** é o protocolo de comunicação utilizado para trocar informações entre o cliente e o servidor. Define as regras e formatos para as requisições e respostas HTTP;
- **Recursos:** são os dados ou serviços disponibilizados pelo servidor que podem ser acessados pelos clientes. Cada recurso é identificado por um URI (*Uniform Resource Identifier*).

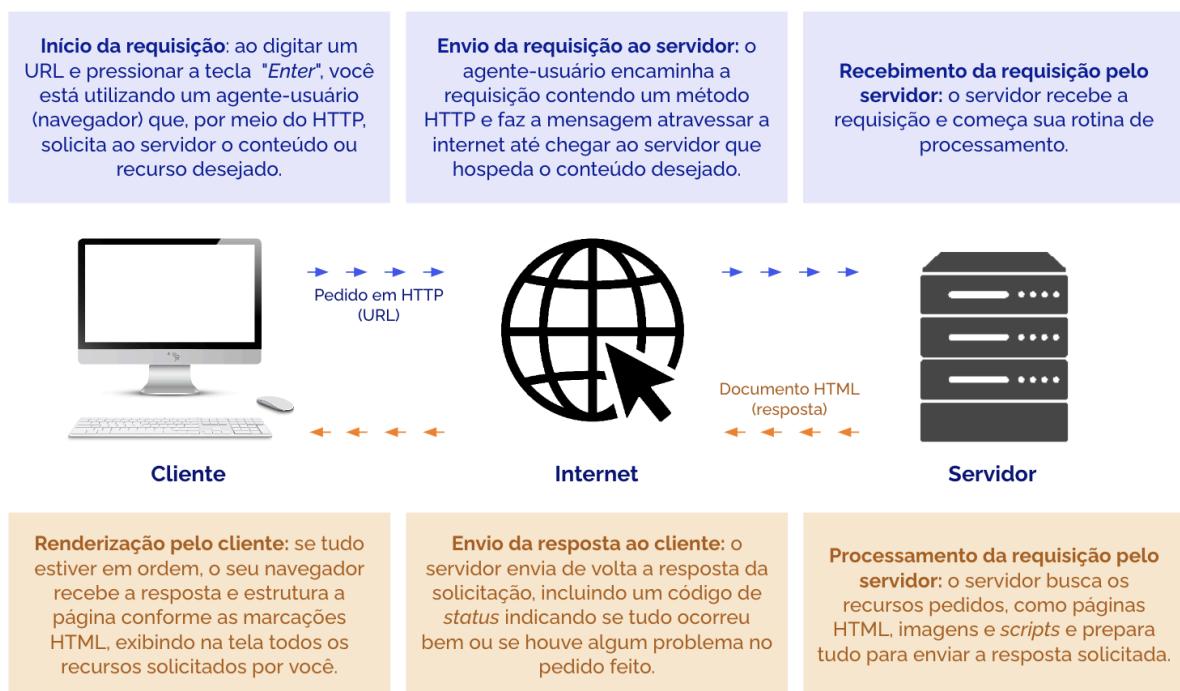
Como o HTTP funciona?

O HTTP é um protocolo **cliente-servidor**, o que significa que os pedidos são iniciados pelo cliente ou agente-usuário, que, geralmente, é um navegador.

Quando um usuário *on-line* deseja carregar ou interagir com uma página na internet, o **navegador web do dispositivo envia uma solicitação HTTP ao servidor de origem** que hospeda os arquivos do *site*. Estas solicitações são, basicamente, linhas de texto enviadas pela internet.

Dessa forma, uma conexão é estabelecida entre o navegador e o servidor e, depois disso, o servidor do *site* processa a solicitação desejada e envia uma resposta de volta ao navegador do usuário. Isso faz com que as páginas dos *sites* na internet sejam acessíveis aos visitantes.

Observe a seguir um exemplo desse funcionamento:



Esta interação é vital para o funcionamento dinâmico da *web*, pois permite que diferentes entidades se comuniquem de forma eficiente, seja um usuário explorando a internet ou um robô indexando informações para um mecanismo de pesquisa.



Você percebeu que nos pedidos que enviamos ao servidor existem **métodos**? Eles são muito importantes nessa comunicação, por isso, agora vamos conhecer os métodos HTTP mais comuns. Preparado(a)?

Métodos HTTP comuns

Os métodos HTTP são diferentes comandos ou solicitações que fazemos com frequência quando interagimos com os *sites* na internet. Eles são os mensageiros que levam os nossos pedidos aos servidores e trazem de volta as respostas desejadas.

Esses métodos são a linguagem que nosso navegador usa para se comunicar com os servidores, permitindo a interação fluida e dinâmica que experimentamos ao navegar na internet.

Cada método tem seu propósito específico, contribuindo para a riqueza da comunicação *on-line*. Os utilizados com mais frequência são:

GET

Conceito: é uma espécie de **buscador de informações** por meio de perguntas diretas. Assim, é como se você pedisse ao servidor "me mostre esta página" e ele retornasse a resposta correspondente.

Exemplo: ao navegar em um *site* de notícias e clicar em uma manchete, o seu navegador envia uma requisição **GET** para o servidor, solicitando os detalhes da notícia.

```
GET /noticias/123 HTTP/1.1
```

```
Host: exemplo.com
```

POST

Conceito: agora, se precisa **enviar informações** para o servidor, você vai utilizar o método **POST**.

Exemplo: ao preencher um formulário de inscrição *on-line* e clicar em "Enviar", o seu navegador usa o **POST** para enviar os dados do formulário ao servidor.

```
POST /inscricao HTTP/1.1
```

```
Host: exemplo.com
```

```
Conteúdo do Corpo: nome=Joao&email=joao@email.com
```

PUT

Conceito: é como editar uma entrada existente, **atualizando informações**. Se você já enviou informações para o servidor, mas precisa fazer uma atualização ou correção, o **PUT** é o seu aliado.

Exemplo: ao editar seu perfil em uma rede social e salvar as alterações, o seu navegador usa o método **PUT** para enviar as informações atualizadas ao servidor.

```
PUT /perfil/usuario123 HTTP/1.1  
Host: rede-social.com  
Conteúdo do Corpo: {"nome": "Joao Silva", "cidade": "Sao Paulo"}
```

DELETE

Conceito: e se você quiser se livrar de alguma informação? É aí que o método **DELETE** entra em cena. É como dizer ao servidor, "por favor, **remova** esta entrada ou recurso".

Exemplo: ao clicar em "Excluir" para deletar uma publicação no seu *blog*, o navegador envia uma requisição **DELETE** para solicitar a remoção do conteúdo.

```
DELETE /postagem/456 HTTP/1.1  
Host: blog.com
```

Segurança em comunicações HTTP

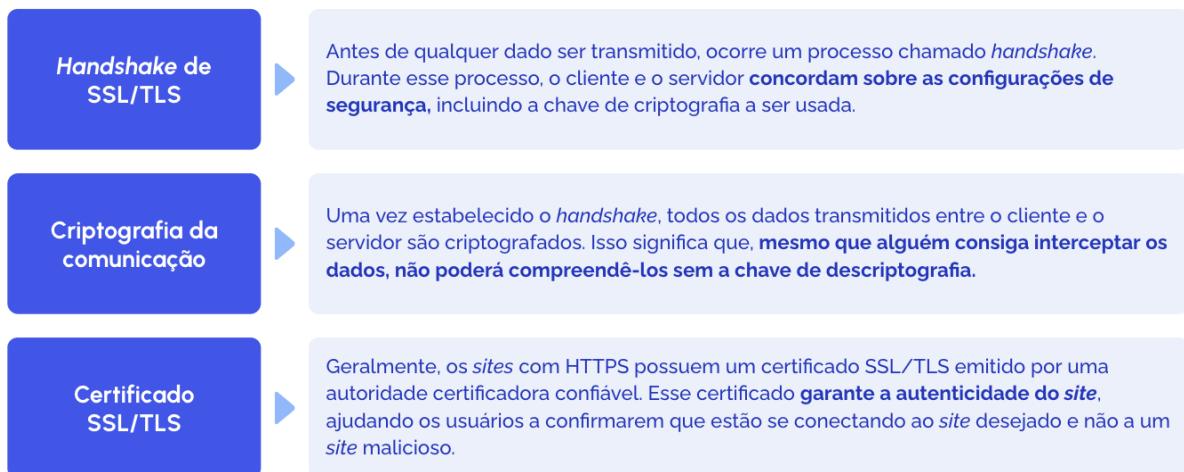
Apesar do protocolo HTTP (*Hypertext Transfer Protocol*) ser fundamental para a comunicação na *web*, ele possui uma lacuna importante: a **falta de segurança na transmissão de dados**. Em uma comunicação HTTP padrão, os dados são transmitidos em texto simples, o que significa que informações sensíveis podem ser interceptadas e compreendidas por terceiros mal-intencionados.

HTTPS (HTTP Secure)

Para resolver esse problema de segurança do HTTP, foi introduzido o HTTPS, ou *HTTP Secure*. Ele é uma **extensão do HTTP, mas com uma camada adicional de segurança**. Ele utiliza o protocolo SSL/TLS para criptografar os dados transmitidos entre o cliente (navegador) e o servidor, garantindo uma comunicação segura.

Como funciona o HTTPS?

Quando você acessa um site com HTTPS, ocorre o seguinte:



Vantagens do HTTPS

- **Privacidade:** protege a privacidade das informações transmitidas;
- **Integridade:** garante que os dados não foram alterados durante a transmissão;
- **Autenticidade:** confirma a identidade do *site*.

Adotar HTTPS é crucial para garantir a segurança e privacidade dos usuários na *web*, tornando a experiência de navegação mais confiável e protegida contra ameaças cibernéticas. 



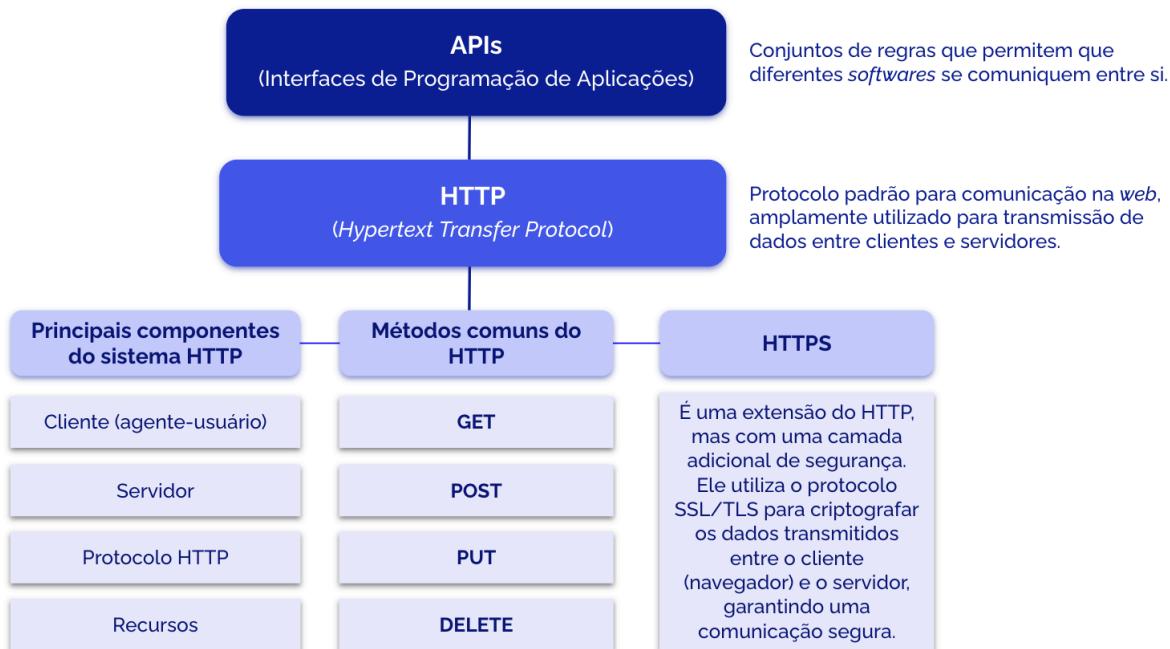
Neste capítulo, nós aprendemos que as APIs servem como intermediárias na comunicação entre diferentes programas de computador, agindo como os garçons em um restaurante, facilitando o processo de pedir e receber informações.

Paralelamente, vimos que o HTTP, ou Protocolo de Transferência de Hipertexto, é a infraestrutura que possibilita essa comunicação na *web*, permitindo que os clientes enviem requisições aos servidores, que, por sua vez, processam essas solicitações e enviam respostas de volta aos clientes. Aprendemos que esse processo ocorre em um modelo cliente-servidor, no qual o agente-usuário, como um navegador da *web*, é responsável por iniciar as requisições. Além disso, estudamos que o HTTP define

diferentes métodos, como **GET**, **POST**, **PUT** e **DELETE**, que são usados para realizar diversas ações, como buscar, enviar, atualizar ou excluir informações.

Por fim, falamos sobre a segurança das comunicações HTTP, com o HTTPS (HTTP Secure) sendo apresentado como uma versão mais segura, que criptografa os dados transmitidos, garantindo uma troca de informações mais segura e protegida contra ameaças externas.

Veja o resumo no esquema a seguir:



ATIVIDADE DE FIXAÇÃO

1. Defina a sigla API e algumas comparações que podem ser relacionadas.
2. Descreva o processo de uma requisição HTTP pelo cliente até a renderização do conteúdo pelo navegador.
3. Qual é o papel do agente-usuário no contexto do protocolo HTTP? Dê exemplos de agentes-usuários.
4. Explique o motivo do protocolo HTTP não ser considerado seguro. Como o HTTPS soluciona essa questão?

5. Quais são os métodos HTTP comuns? Dê exemplos práticos de situações em que cada um seria utilizado.
6. Como o **GET** e o **POST** diferem em termos de funcionalidade no contexto de requisições HTTP?
7. Por que a segurança em comunicações *web* é crucial? Explique como o HTTPS contribui para a segurança.
8. Quais são os componentes principais de sistemas baseados em HTTP? Forneça exemplos de situações práticas de uso.
9. Como você descreveria a relação entre o HTTP e a API no contexto da comunicação *web*?
10. Em que situações um desenvolvedor pode optar por utilizar o método HTTP **DELETE** em uma aplicação *web*?

CAPÍTULO 04

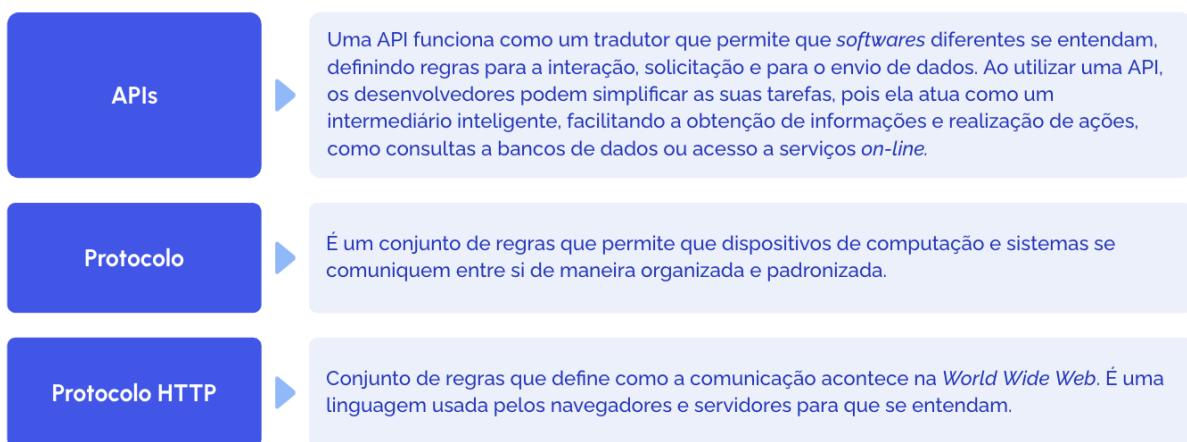
EXPLORANDO REQUISIÇÕES HTTP COM cURL, POSTMAN E UMA API SIMPLES

O que esperar deste capítulo:

- Compreender como utilizar as ferramentas cURL e Postman para enviar requisições HTTP;
- Fazer requisições HTTP a uma API simples.

Neste capítulo, nós vamos desbravar as complexidades da comunicação entre clientes e servidores por meio do protocolo HTTP. Nesse processo, ferramentas como **cURL** e **Postman** serão nossas grandes aliadas. Ao longo dessa jornada, não só vamos compreender mais sobre o protocolo HTTP, como também vamos aprender a interagir com as APIs de maneira descomplicada e eficiente. O seu entendimento sobre como enviar requisições, analisar respostas e interagir com APIs será levado a novos patamares. Então, embarque conosco nessa aventura HTTP!

Mas, antes, vamos relembrar alguns conceitos importantes. Veja o esquema abaixo:



Além disso, precisamos relembrar como ocorre o **funcionamento do HTTP**:

- se um usuário *on-line* deseja interagir com uma página na internet, o navegador do dispositivo envia uma solicitação ou requisição HTTP ao servidor de origem que hospeda os arquivos do *site*;

- basicamente, estas solicitações são linhas de texto enviadas pela internet e, assim, uma conexão é estabelecida entre o navegador e o servidor;
- depois disso, o servidor do *site* processa a solicitação desejada e envia uma resposta de volta ao navegador do usuário. Isso faz com que as páginas dos *sites* na internet sejam acessíveis aos visitantes.

Tendo tudo isso em mente, agora nós podemos explorar os elementos de uma requisição HTTP e as ferramentas cURL e Postman essenciais para desenvolvedores que trabalham com APIs.

4.1 Requisições HTTP

As requisições HTTP são mensagens enviadas pelo cliente (como um navegador) para um servidor *web*. Essas mensagens solicitam a obtenção de recursos específicos, como páginas da *web*, imagens, arquivos etc. Para entender bem essas requisições, precisamos conhecer seus elementos.

4.1.1 Elementos de uma Requisição HTTP

Métodos: são ações que você pede para a web realizar. Por exemplo, ao preencher um formulário *on-line*, você está usando o método **POST** para enviar os dados.

URL: URLs são como endereços únicos na *web*. Cada página, imagem ou recurso tem um endereço para encontrá-lo. Assim, "www.exemplo.com/pagina" seria um URL que leva você a uma página específica deste *site*.

Cabeçalhos: são como etiquetas adicionadas à requisição, fornecendo informações extras. Podem incluir detalhes sobre o tipo de conteúdo que o cliente solicitou ou o tipo de resposta desejada. Um cabeçalho "**Content-Type: application/json**" informa que o conteúdo é um formato JSON.

Corpo: nem toda requisição tem um corpo, mas, quando tem, é o local em que dados adicionais podem ser enviados para o servidor. Um exemplo é o preenchimento de um formulário *on-line*. Em uma requisição POST para criar um usuário, o corpo pode ter `{"nome": "João", "idade": 25}`.

4.2 Explorando as ferramentas como cURL e Postman

Tanto a ferramenta cURL quanto a Postman são amplamente utilizadas para interagir com APIs e analisar respostas HTTP. Porém, é importante ter em mente que elas têm abordagens diferentes em termos de proposta e funcionalidade. A seguir, veja as características de cada uma.

4.2.1 cURL

- cURL é a abreviação de *Client for URLs* e é uma ferramenta de linha de comando que permite a transferência de dados com suporte para diversos protocolos, incluindo HTTP;
- com ela, é possível fazer solicitações HTTP para interagir com APIs;
- dessa forma, os desenvolvedores podem usar comandos específicos para enviar solicitações, receber respostas e analisar os resultados diretamente no terminal;
- É especialmente útil para tarefas diretas e integração em *scripts*, proporcionando uma abordagem eficaz, embora mais técnica, para lidar com requisições web.

4.2.2 Postman

- Postman é uma ferramenta com interface gráfica que simplifica o processo de interação com APIs;
- com sua interface gráfica intuitiva, ele transformou a exploração e o teste de APIs em uma jornada mais acessível;
- essa ferramenta permite que os desenvolvedores criem, enviem e analisem requisições HTTP de forma visual, simplificando significativamente o processo, especialmente para aqueles que estão começando na área de desenvolvimento.

Ferramentas	Vantagens	Desvantagens
cURL	<ul style="list-style-type: none">- Simplicidade rápida: ideal para tarefas diretas e rápidas.- Integração eficiente: pode ser facilmente incorporado em scripts e automações.	<ul style="list-style-type: none">- Interface de linha de comando: pode parecer intimidadora para iniciantes.- Menos intuitivo visualmente: menos adequado para exploração visual.
Postman	<ul style="list-style-type: none">- Interface gráfica amigável: ótima para iniciantes e visualmente intuitiva.- Funcionalidades avançadas: oferece recursos avançados para testes e automação.	<ul style="list-style-type: none">- Excesso para tarefas simples: pode parecer um pouco robusto para tarefas básicas.- Curva de aprendizado: exige algum tempo para se familiarizar com os recursos avançados.

A escolha entre eles depende das preferências individuais e da natureza específica das tarefas. **Mas, como essas ferramentas funcionam na prática?**

4.3 Funções do cURL e do Postman na análise das respostas HTTP:

Tanto o cURL quanto o Postman são ferramentas essenciais para entender e interpretar as interações com APIs. Elas atuam das seguintes maneiras:

1. Indicando resultados

Tanto o Postman quanto o cURL exibem códigos de *status* HTTP como indicadores de resultado das requisições. Um exemplo de código é o **200 OK**, que indica que a requisição foi bem-sucedida.

Exemplo

Vamos usar o cURL para fazer uma solicitação GET para uma API de exemplo e observar o código de status retornado:

```
curl -I https://jsonplaceholder.typicode.com/posts/1
```

Neste exemplo, **-I** solicita apenas os cabeçalhos da resposta. Assim, o resultado incluirá algo como:

```
HTTP/1.1 200 OK
```

Indicando que a requisição foi bem-sucedida (código "200 OK").

2. Fornecendo detalhes informativos

Essas ferramentas permitem visualizar os cabeçalhos das respostas HTTP, que fornecem informações detalhadas sobre a requisição e a resposta. Isso inclui detalhes como o tipo de conteúdo, o tamanho da resposta, *cookies* etc.

Exemplo

Após enviar uma solicitação GET para a API no Postman, você pode clicar na guia "Headers" para visualizar os cabeçalhos da resposta, como mostrado na imagem a seguir:

```
Content-Type: application/json
```

O Postman exibindo o cabeçalho "Content-Type: application/json", indica que o conteúdo da resposta está estruturado em JSON.

3. Facilitando a leitura do corpo da resposta

Tanto o cURL quanto o Postman permitem a visualização do corpo das respostas HTTP, que contêm os dados retornados pela API. Isso inclui informações cruciais para entender o resultado da requisição, como dados estruturados em formatos como JSON, XML ou HTML.

Exemplo

O cURL, por exemplo, pode retratar um simples JSON como resposta:

```
{"mensagem": "Sucesso!"}
```



No caso do Postman, ele exibe claramente o corpo da resposta na interface do usuário, facilitando a visualização e análise dos dados.

4. Formatando e validando dados

Tanto o cURL quanto o Postman oferecem recursos que auxiliam na formatação e na validação de dados para garantir precisão e usabilidade durante o desenvolvimento de APIs.

- **cURL:** embora o cURL seja uma ferramenta de linha de comando mais básica em comparação com o Postman, ele ainda oferece algumas opções para facilitar a formatação e a validação de dados;
 - **Formatação de dados:** o cURL permite o envio de dados em diferentes formatos, como JSON, XML ou formulários URL-encoded, utilizando as opções adequadas no momento da criação da solicitação;
 - **Validação de dados:** embora o cURL não tenha recursos integrados para validação de dados, os desenvolvedores podem usar ferramentas externas ou scripts para validar os dados recebidos nas respostas das solicitações.
- **Postman:** essa ferramenta oferece recursos mais avançados para formatação e validação de dados, tornando o processo mais conveniente e eficiente.
 - **Formatação automática:** o Postman pode formatar automaticamente os dados de solicitação e resposta em formatos como JSON, XML, *form-data* etc., facilitando o envio e o recebimento de dados estruturados;

- **Validação de esquema:** o Postman permite que os desenvolvedores definam esquemas de validação JSON ou XML para as respostas recebidas. Ele pode validar automaticamente as respostas recebidas em relação a esses esquemas, garantindo que os dados estejam corretos e estruturados conforme o esperado;
- **Testes automatizados:** além disso, o Postman possui recursos de teste integrados que permitem que os desenvolvedores criem testes automatizados para verificar a precisão e a integridade dos dados recebidos nas respostas das solicitações.

4.3.1 Como fazer uma requisição HTTP a uma API com PokéApi

Senta, que lá vem história!

A pokemon API é uma interface de programação de aplicação que permite acessar dados sobre os pokémons, mas você sabe o que são pokémons?

Os pokémons são criaturas fictícias criadas em 1996 pelo designer japonês Satoshi Tajiri, que se inspirou em sua infância, quando gostava de colecionar insetos. O nome Pokémon vem da abreviação de "Pocket Monsters" (Monstros de Bolso). O primeiro jogo da série foi lançado para Game Boy em 1996 no Japão, e se tornou um sucesso, gerando várias adaptações para TV, cinema, jogos de cartas e outros produtos.

O pokemon API foi criada em 2014 por Paul Hallett, um desenvolvedor britânico que é fã de Pokémon desde criança. Ele decidiu criar a API como um projeto pessoal, usando dados extraídos dos jogos originais. A API é gratuita e aberta, e permite que qualquer pessoa possa usar os dados para criar seus próprios projetos, como jogos, aplicativos, sites, etc. Essa API é atualizada constantemente, e conta com mais de 800 pokémons e mais de 40 recursos diferentes, como tipos, habilidades, evoluções, etc.



4.3.2 Fazendo requisições à pokémon API

A seguir, veja alguns exemplos de requisições que podem ser feitas tanto no cURL, quanto no Postman para obter informações de um Pokémon.

cURL

```
curl -X GET "https://pokeapi.co/api/v2/pokemon/1/"
```

Este comando cURL solicita informações sobre o Pokémon com a ID 1. A resposta incluirá dados como nome, tipo e habilidades.

Postman

No Postman, ao enviarmos as requisições a seguir, nós vamos obter uma lista de Pokémon do tipo 1. Cada item na lista contém detalhes sobre um Pokémon específico.

- **Método:** GET;
- **URL:** <https://pokeapi.co/api/v2/type/1/> ;
- **Cabeçalhos:** (opcional);
- **Corpo:** (vazio).

4.3.3 Explorando a Pokémon API na prática

Agora, nós vamos mergulhar em alguns exemplos práticos de como fazer requisições HTTP para a Pokémon API. Vamos utilizar ferramentas como cURL e Postman para explorar essa fonte de conhecimento Pokémon. Antes de começarmos, certifique-se de ter o cURL ou o Postman instalado. Ambos são excelentes ferramentas, então escolha aquela com a qual você se sentir mais confortável.

Instalando o cURL e o Postman - Windows

cURL	Postman
<ul style="list-style-type: none">• Baixe o instalador do cURL para Windows no <i>site</i> oficial: https://curl.se/windows/;• Execute o instalador baixado;• Siga as instruções do assistente de instalação;• Após a instalação, o cURL estará pronto para uso no seu <i>prompt</i> de comando.	<ul style="list-style-type: none">• Baixe o instalador do Postman para Windows no <i>site</i> oficial: https://www.postman.com/downloads/;• Execute o instalador baixado.• Siga as instruções do assistente de instalação;• Após a instalação, você pode iniciar o Postman a partir do menu Iniciar.

Agora, vamos aos exemplos práticos! Para executar as requisições de forma correta, siga o passo a passo a seguir:



1

Conhecendo a API: primeiro, acesse o *site* oficial da PokéAPI (<https://pokeapi.co/>) para entender quais *endpoints* estão disponíveis. Examine a documentação para ter uma visão geral dos recursos oferecidos.



Endpoints são pontos de extremidade em uma API que representam os diferentes recursos ou funcionalidades disponíveis para interação. Cada *endpoint* é uma URL única que é acessada através de métodos HTTP.



2

Obtendo informações de um pokémon: vamos começar de uma forma simples. Use o cURL ou o Postman para fazer uma requisição **GET** ao *endpoint* de um Pokémons específico. Por exemplo, para Charizard:

```
curl https://pokeapi.co/api/v2/pokemon/charizard
```

Ou, no Postman, crie uma nova requisição **GET** para o mesmo URL.



3

Explorando detalhes: analise a resposta para entender os detalhes do Pokémons. Identifique características como habilidades, tipos, estatísticas, entre outros. Isso te dará uma ideia de como os dados são estruturados.



4

Personalizando requisições: experimente fazer requisições personalizadas. Por exemplo, obtenha informações sobre os tipos de Pokémons, as habilidades disponíveis ou, até mesmo, os locais em que eles podem ser encontrados.



5

Criando um projeto simples: agora, vamos dar um passo adiante. Use sua linguagem de programação favorita (como Python, JavaScript etc.) para criar um projeto simples que faça requisições à API e exiba as informações de um Pokémons em um formato amigável.

Explore outros recursos da API, como os *endpoints* de tipos, habilidades e evoluções para enriquecer ainda mais seu projeto. Lembre-se, a prática leva à maestria! Ao criar um projeto interativo, você ganhará uma compreensão mais profunda de como trabalhar com APIs. Divirta-se explorando o mundo Pokémons digital!

DESAFIO PRÁTICO

Explorando requisições HTTP com cURL, Postman e uma API Simples

Desafio: Desenvolver uma pesquisa das possíveis aplicações do PokeAPI.



Descrição

Imagine que você foi contratado para desenvolver uma Pokédex *on-line* utilizando a PokeAPI. A ideia é criar uma aplicação *web* em que os usuários podem visualizar detalhes sobre diferentes Pokémons, como os seus tipos, as suas habilidades, as suas características e imagens. Para isso, você terá que entender os processos de como uma Pokédex funciona.



Objetivos

- Entender a Integração com a PokeAPI;
- Compreender como é relacionada a exibição de lista de Pokémons;
- Criar e exibir os detalhes do Pokémon;
- Exibir as informações.

Orientações

- Use uma linguagem de programação *web* de sua escolha e crie uma pequena *one page*;
- Considere o *design* responsivo para tornar a aplicação amigável em dispositivos móveis;
- Faça bom uso de boas práticas de codificação, como modularização e tratamento de erros;
- Considere toda a pesquisa realizada sobre o PokeAPI.



Neste capítulo, relembramos que o HTTP é a base da comunicação entre *sites* na internet e que ele estabelece as regras e os formatos padrão para a troca de informações, permitindo que diferentes sistemas se entendam. Além disso, nós aprendemos que essa comunicação é feita por meio das **mensagens HTTP** que, além dos **métodos** e das **URLs**, contêm **cabeçalhos** de requisição e resposta, fornecendo informações adicionais sobre a requisição ou resposta. Paralelamente a isso, os **corpos** de requisição e resposta contêm os dados reais transmitidos entre os clientes e os servidores, como o conteúdo das mensagens trocadas.

Entendemos também que, para facilitar o envio e recebimento de requisições HTTP, existem ferramentas especializadas, como o **cURL** e o **Postman**. O cURL é um programa de **linha de comando** amplamente utilizado para interagir com URLs e APIs, enquanto o Postman oferece uma **interface gráfica** mais amigável, ideal para usuários menos familiarizados com a linha de comando. Ambas as ferramentas permitem enviar requisições HTTP e analisar as respostas recebidas, simplificando o processo de desenvolvimento e teste de APIs.

Por fim, vimos um exemplo prático da utilização desses conceitos explorando a **Pokémon API**, uma interface que fornece acesso a informações sobre Pokémons. Essa API permite aos desenvolvedores obter dados sobre diferentes aspectos dos Pokémons, como os seus atributos, as suas habilidades e os seus tipos. Usando ferramentas como cURL ou Postman, entendemos que é possível enviar requisições HTTP para a API e receber os dados desejados, que podem, então, ser utilizados em projetos de *software*, jogos ou análises relacionadas ao universo Pokémons.



1. Explique a importância do protocolo HTTP na comunicação *web* e como ele facilita a troca de informações entre clientes e servidores.
2. Quais são os métodos HTTP mais comuns e como eles diferem em suas funcionalidades?
3. Como as URLs são utilizadas na *web*? Por que elas são consideradas endereços únicos?

- 4.** Qual é o papel dos cabeçalhos de requisição HTTP? Como eles podem influenciar a comunicação entre cliente e servidor?
- 5.** Em que situações uma requisição HTTP pode incluir um corpo e qual é a sua função?
- 6.** Como os códigos de *status* em uma resposta HTTP fornecem informações sobre o resultado da requisição?
- 7.** Por que as APIs são essenciais na comunicação entre sistemas diferentes?
- 8.** Quem criou a Pokémon API? Quais são os principais recursos oferecidos por ela?
- 9.** Dê um exemplo prático de uma requisição cURL para obter informações sobre um Pokémon.
- 10.** Qual é a importância de configurar o ambiente antes de interagir com uma API?

CAPÍTULO 05

FORMATOS DE DADOS EM APIs

O que esperar deste capítulo:

- Compreender as linguagens JSON (JavaScript *Object Notation*) e XML (*eXtensible Markup Language*);
- Entender como se estruturam os dados em JSON e XML.

5.1 JSON (JavaScript Object Notation) e XML (*eXtensible Markup Language*)

Nesse capítulo, nós vamos conhecer o funcionamento do JSON e do XML, dois formatos de dados que tem algumas características em comum, como:

- ambos são formatos muito utilizados para representar e transmitir informações de forma estruturada e legível por uma máquina;
- são amplamente empregados em diversas aplicações de desenvolvimento de *software*, ainda que tenham características distintas;
- além disso, eles são independentes de linguagem. Ou seja, podem ser gerados e interpretados por diversas linguagens de programação, promovendo a **interoperabilidade**.



A interoperabilidade entre sistemas quer dizer que, apesar de possuirem diferentes linguagens, eles podem trabalhar juntos.

Por exemplo, um sistema pode ser desenvolvido em Java para gerar documentos XML, enquanto outro sistema, que pode ser desenvolvido em Python, pode interpretar esses documentos XML para processamento adicional. Desde que as duas partes entendam a estrutura do XML (o que é garantido pelas regras de marcação definidas no XML), elas podem trocar informações de forma eficaz, independentemente das linguagens em que foram desenvolvidas.

Juntos, nós vamos conhecer as características e as estruturas básicas de cada um desses formatos de dados. Porém, antes, temos que entender melhor o conceito de JSON e XML.

XML	JSON
<ul style="list-style-type: none"> • É uma linguagem de marcação extensível que permite definir conjuntos de regras para a codificação de documentos em um formato legível tanto por humanos quanto por máquinas; • É utilizado em uma variedade de aplicações, incluindo a troca de dados entre sistemas heterogêneos, o armazenamento de configurações e metadados e a representação de documentos estruturados; • Embora tenha sido amplamente adotado, o seu uso tem diminuído em favor de formatos mais simples e leves, como JSON. 	<ul style="list-style-type: none"> • Foi criado em meados dos anos 2000, como uma alternativa mais simples e legível ao formato XML; • É um formato de dados leve e fácil de ler, sendo originalmente derivado da sintaxe de objetos JavaScript; • É amplamente utilizado na comunicação entre sistemas <i>web</i>, especialmente em serviços de API; • É usado para transmitir dados estruturados entre um servidor e um cliente, seja em aplicativos <i>web</i>, aplicativos móveis, entre outros.

5.2 Estrutura básica do XML e do JSON

5.2.1 Estrutura básica XML

A estrutura básica do XML (*eXtensible Markup Language*) é composta por **tags**, que são elementos fundamentais para estruturar e representar os dados. As *tags* são marcadores de início e fim que definem a estrutura **hierárquica** do documento XML.

```
<root>
  <elemento1>
    <subelemento1>Valor1</subelemento1>
    <subelemento2>Valor2</subelemento2>
  </elemento1>
  <elemento2>
    <subelemento3>Valor3</subelemento3>
  </elemento2>
</root>
```

No exemplo:

<root> é o elemento raiz, que contém todos os outros elementos.

<elemento1> e **<elemento2>** são elementos filhos de **<root>**.

<subelemento1>, **<subelemento2>**, e **<subelemento3>** são elementos filhos de **<elemento1>** e **<elemento2>**.

Os valores são inseridos entre as *tags* de abertura e fechamento dos elementos, como "**Valor1**", "**Valor2**", e "**Valor3**".

5.2.2 Aplicando a estrutura

Em um documento XML simples que representa informações sobre um **livro**, você pode ter *tags* como **<livro>**, **<titulo>**, **<autor>** etc. Elas delimitam os diferentes elementos do documento e indicam como eles estão relacionados uns aos outros.

```
<livro>
    <titulo>Dom Quixote</titulo>
    <autor>Miguel de Cervantes</autor>
    <ano>1605</ano>
</livro>
```

Neste exemplo, **<livro>**, **<titulo>**, **<autor>** e **<ano>** são todas as *tags* que compõem o documento XML. Elas são cruciais para a interpretação e o processamento dos dados em um documento XML.

5.2.3 Estrutura básica JSON

O JSON não é apenas um formato de dados, mas uma linguagem que transcende barreiras, impulsionando a comunicação entre sistemas e possibilitando interações dinâmicas na *web*. Compreender o JSON é desbravar as fronteiras da programação moderna e, por isso, é importante conhecer a sua estrutura.

No JSON, os dados são organizados em **pares de chave e valor** que formam um **objeto**.

Chave: identifica o conteúdo e é uma **string** delimitada por **aspas**.

Valor: representa o conteúdo e pode ser de diversos tipos: **string, array, object, number, boolean ou null**.

Objeto: são delimitados por **chaves {}** e podem conter **pares de chave e valor aninhados**.

```
{
    "chave1": "valor1",
    "chave2": "valor2",
    "chave3": {
        "subchave1": "subvalor1",
        "subchave2": "subvalor2"
    },
    "chave4": ["item1", "item2", "item3"],
    "chave5": 123,
    "chave6": true,
    "chave7": null
}
```

Você sabe o que as palavras **strings**, **número**, **objeto**, **booleano**, **array** e **null** representam? Para entender melhor, observe o esquema a seguir:

Esclarecendo alguns termos...

String	É uma sequência de caracteres , sejam letras, frases, números ou símbolos, que representam um texto e são delimitadas por aspas duplas.
Número	Usados para representar quantidades , como a idade de alguém, o preço de um produto, entre outros. Podem ser número inteiros ou número de ponto flutuante (decimais).
Objeto	É uma coleção de pares chave-valor , em que a chave é sempre uma string e o valor pode ser qualquer tipo de dado JSON.
Booleano	É um tipo de dado que pode ter apenas dois valores: verdadeiro (true) ou falso (false) . Eles não são envolvidos por aspas e são usados para representar estados , como se algo está ligado ou desligado, se algo é verdadeiro ou falso etc.
Array	Representa uma coleção ou lista ordenada de valores, delimitada por colchetes [] . Os valores podem ser de qualquer tipo JSON, incluindo strings , números , objetos , booleanos ou até mesmo outro array . Além disso, os elementos do array são separados por vírgulas.
["valor1", 42, true, null]	
Null	É escrito sem aspas e representa um valor nulo , indicando a ausência de valor ou a inexistência de um válido para ser representado.

O JSON estrutura os dados de maneira hierárquica, permitindo representar informações de forma organizada e semântica. Sendo assim, se temos o **livro "Dom Quixote" de Miguel de Cervantes, escrito em 1605**, os dados podem ser representados da seguinte forma:

```
{
  "livro": {
    "titulo": "Dom Quixote",
    "autor": "Miguel de Cervantes",
    "ano": 1605
  }
}
```

No código apresentado anteriormente, é possível observar que:

- o objeto JSON principal começa com **{ }** e contém um par de chave e valor;
- a chave "**livro**" tem como valor um objeto JSON aninhado, indicado pelas chaves

internas {};

- Dentro do objeto "livro", há pares de chave e valor representando diferentes informações sobre o livro, como:
 - "titulo": uma *string* com o título do livro;
 - "autor": uma *string* com o nome do autor do livro;
 - "ano": um número inteiro representando o ano de publicação do livro.

5.2.4 Utilização do JSON em transferência de dados

- **API (Application Programming Interface)**: facilita a transferência estruturada de informações entre aplicações;
- **Formato leve**: comparado ao XML, o JSON é uma alternativa mais leve e eficiente.

5.2.5 Papel do JSON em requisições AJAX e interações web

- **Requisições AJAX**: amplamente utilizado em interações dinâmicas em sites;
- **Integração com bancos de dados**: facilita operações como consulta, inclusão e exclusão de registros.

5.2.6 Diferenças entre JSON e XML: estrutura e aplicações

Agora, nós vamos explorar as principais diferenças entre eles. Observe o esquema a seguir:

Características	JSON	XML
Sintaxe.	Baseado na sintaxe literal do JavaScript.	Uma linguagem de marcação generalizada simples (SGML).
Estrutura de dados.	Representa dados como objetos.	Usa uma estrutura de tags para representar itens de dados.
Namespaces.	Não suporta.	Suporta.
Arrays.	Suporta.	Não suporta.
Segurança.	Considerado menos seguro.	Mais seguro.
Legibilidade.	Facilmente legível.	Menos legível.
Tamanho do arquivo.	Menor tamanho de arquivo.	Maior tamanho de arquivo.
Acesso aos dados.	Acessa dados através de objetos JSON.	Precisa de dados para ser analisado.
Orientação.	Orientado a dados.	Orientado a documentos.

5.2.7 Usos comuns do JSON

- **Comunicação cliente-servidor:** o JSON é amplamente utilizado para a troca de dados entre clientes e servidores em aplicações web e móveis;
- **Persistência de dados:** muitas vezes, o JSON é usado para persistir dados em arquivos de configuração ou bancos de dados NoSQL, devido à sua natureza estruturada e legível;
- **Configuração de aplicações:** arquivos de configuração de aplicações, especialmente em ambientes web, são frequentemente representados em JSON;
- **Serialização de objetos:** em linguagens orientadas a objetos, a serialização de objetos para JSON e a subsequente desserialização são práticas comuns.
- **Troca de mensagens em sistemas distribuídos:** o JSON é uma escolha popular para representar mensagens em sistemas distribuídos devido à sua simplicidade e interoperabilidade.

Exemplo de uso de JSON: intercâmbio de dados em uma aplicação web

```
{  
  "produto": {  
    "id": 123,  
    "nome": "Smartphone XYZ",  
    "preco": 599.99,  
    "estoque": 50,  
    "categorias": ["eletrônicos", "tecnologia"]  
  },  
  "cliente": {  
    "id": 456,  
    "nome": "Hanzo Sasaki",  
    "endereco": "Rua Principal, 123",  
    "cidade": "Cidade A"  
  },  
  "pedido": {  
    "id": 789,  
    "produtos": [  
      {"produto_id": 123, "quantidade": 2},  
      {"produto_id": 456, "quantidade": 1}  
    ],  
    "total": 1299.98  
  }  
}
```

Imagine uma aplicação *web* de comércio eletrônico que precisa trocar informações sobre produtos entre o servidor e o cliente.

Nesse caso, o formato JSON pode ser amplamente utilizado. Veja, ao lado, um exemplo simplificado de como os dados podem ser representados em JSON

Neste exemplo, os dados sobre um **produto**, **cliente** e **pedido** são estruturados usando a notação JSON, tornando fácil a transmissão e interpretação dessas informações entre o servidor e o cliente em uma aplicação web.

5.2.8 Usos comuns do XML

- **Configuração de Aplicações:** o XML é frequentemente utilizado para armazenar configurações de aplicações devido à sua estrutura hierárquica e capacidade de representar dados complexos.
- **Documentos Estruturados:** o XML é amplamente empregado na criação de documentos estruturados, como artigos científicos, manuais técnicos e documentos legais.
- **Troca de Dados entre Sistemas Heterogêneos:** assim como o JSON, o XML é usado para trocar dados entre sistemas heterogêneos, proporcionando uma forma padrão de representação.
- **Intercâmbio de Dados em Web Services:** muitos Web Services utilizam XML para a troca de mensagens, especialmente aqueles que seguem os padrões SOAP (Simple Object Access Protocol).
- **Persistência de Dados:** XML é utilizado em bancos de dados NoSQL e em sistemas que exigem uma representação clara e hierárquica dos dados.
- **Representação de Documentos Markup:** XML é a escolha padrão para representar documentos markup, onde a estrutura hierárquica é essencial para a compreensão do conteúdo.

Exemplo de uso de XML: representação de dados em um documento de configuração

```
<configuracoes>
  <conexao>
    <host>localhost</host>
    <porta>3306</porta>
    <usuario>admin</usuario>
    <senha>secreta123</senha>
  </conexao>
  <parametros>
    <limite_registros>100</limite_registros>
    <idioma>pt-BR</idioma>
  </parametros>
</configuracoes>
```

Agora, considere um cenário em que um sistema precisa armazenar configurações em um formato estruturado.

O XML, devido à sua capacidade de representar hierarquias e estruturas complexas, pode ser a escolha adequada. Veja, ao lado, um exemplo simplificado:

Neste exemplo, as configurações do sistema são expressas em XML, proporcionando uma estrutura organizada e legível para representar detalhes como as informações de conexão e os parâmetros de funcionamento. Esses dados podem ser facilmente interpretados por um sistema que lê e processa configurações em XML.

DESAFIO PRÁTICO

Organizador de tarefas - JSON versus XML

Desafio: Desenvolvendo um organizador de tarefas.



Descrição

Imagine que você está criando um aplicativo de organização de tarefas. A sua missão é implementar um sistema eficiente para armazenar e intercambiar dados sobre as tarefas dos usuários. Neste desafio, você vai explorar as diferenças entre JSON e XML na representação dessas informações.



Objetivos

- Utilizar tanto JSON quanto XML para representar informações sobre tarefas;
- Implementar a capacidade de adicionar, editar e excluir tarefas na aplicação;
- Comparar e contrastar as características de JSON e XML na persistência de dados de tarefas.



Orientações

- Faça uma pesquisa que demostre como é a aplicação do desafio acima em JSON e em XML;
- Pesquise as funcionalidades que permitem aos usuários adicionar, editar e excluir tarefas na aplicação;
- Pesquise como utilizar arquivos separados para persistir dados em JSON e XML;
- Analise a facilidade de leitura e manutenção dos dados em ambos os formatos;
- Crie um relatório de dados referente às pesquisas realizadas. Porém, ele deve ser elaborado conforme o seu entendimento dos assuntos pautados.



5.3 JSON estruturando dados

5.3.1 JSON Schema: definindo estrutura e validando dados JSON

O JSON *Schema* é uma especificação poderosa que oferece a capacidade de definir a estrutura, os tipos de dados e as regras de validação para dados JSON. Imagine-o como um **conjunto de diretrizes detalhadas que garantem a consistência e a validade dos dados em larga escala**.

Ao utilizar o JSON Schema, os desenvolvedores podem criar uma base sólida para garantir que os dados JSON sigam um **padrão específico**, promovendo interoperabilidade e consistência nas aplicações.

Exemplo

Suponha que você esteja desenvolvendo um **aplicativo de gerenciamento de tarefas** e deseja definir a estrutura para representar uma **tarefa**. Você pode usar o JSON Schema para criar um esquema que garanta que cada tarefa tenha um **título (string)**, uma **descrição (string opcional)**, uma **data de criação (data e hora)** e um **status (booleano)**.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Esquema da Tarefa",
  "type": "object",
  "properties": {
    "titulo": {
      "type": "string",
      "description": "O título da tarefa"
    },
    "descricao": {
      "type": "string",
      "description": "A descrição da tarefa",
      "default": ""
    },
    "data_criacao": {
      "type": "string",
      "format": "date-time",
      "description": "A data e hora de criação da tarefa"
    },
    "status": {
      "type": "boolean",
      "description": "O status da tarefa (concluída ou não)"
    }
  },
  "required": ["titulo", "data_criacao", "status"]
}
```

Explicando o exemplo:

- "**titulo**" é definido como uma *string* e é obrigatório;
- "**descricao**" é definido como uma *string* opcional, com um valor padrão vazio ("");
- "**data_criacao**" é definido como uma *string* no formato de data e hora e é obrigatório;
- "**status**" é definido como um booleano e é obrigatório.

5.3.2 JSON Web Tokens (JWT): compactação e autossuficiência nas trocas de informações

Os *JSON Web Tokens* (ou *JWTs*) representam uma forma eficiente e autossuficiente de trocar informações entre duas partes. Em um formato compacto, as informações são codificadas como um objeto JSON e assinadas digitalmente usando a assinatura *JSON Web (JWS)*. Essa abordagem confere **segurança e integridade às informações**, tornando os *JWTs* uma escolha popular para a **autenticação e autorização** em sistemas distribuídos.

Exemplo

Ao implementar um sistema de autenticação em um aplicativo web, você pode utilizar JWTs para criar **tokens de acesso que contenham informações sobre o usuário logado**. Por exemplo, um JWT pode conter o ID do usuário, o seu nome de usuário e um tempo de expiração.

Ao lado, nós temos um exemplo de como criar e verificar um JWT em uma aplicação web usando a biblioteca **jsonwebtoken** em Node.js.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Esquema da Tarefa",
  "type": "object",
  "properties": {
    "titulo": {
      "type": "string",
      "description": "O título da tarefa"
    },
    "descricao": {
      "type": "string",
      "description": "A descrição da tarefa",
      "default": ""
    },
    "data_criacao": {
      "type": "string",
      "format": "date-time",
      "description": "A data e hora de criação da tarefa"
    },
    "status": {
      "type": "boolean",
      "description": "O status da tarefa (concluída ou não)"
    }
  },
  "required": ["titulo", "data_criacao", "status"]
}
```

A seguir, veja o que foi feito no código que vimos:

- importamos a biblioteca '**jsonwebtoken**';
- definimos uma chave secreta para assinar e verificar o *token*. Esta chave deve ser mantida em segredo;
- criamos um objeto '**usuario**' com informações do usuário, incluindo um ID, o nome de usuário e o tempo de expiração do *token*;
- usamos '**jwt.sign()**' para criar um JWT, passando o objeto '**usuario**', a chave secreta e as opções adicionais, como o tempo de expiração;
- exibimos o JWT criado no console;
- usamos **jwt.verify()** para validar e decodificar tokens JWT. Ao passar o *token*, a chave secreta e uma função de *callback*, podemos lidar com o resultado. Se o *token* for válido, suas informações são decodificadas e disponibilizadas para acesso. Este processo é essencial em sistemas de autenticação e autorização.

5.3.3 Ferramentas de manipulação de JSON: facilitando a interação com dados JSON

Explorar e manipular dados JSON pode ser otimizado ao aproveitar as diversas ferramentas disponíveis. Uma delas é o **jq**, um poderoso recurso de linha de comando

que oferece flexibilidade e praticidade na manipulação de JSON. Além disso, muitas linguagens de programação, como JavaScript e Python, fornecem bibliotecas integradas para lidar com JSON de forma eficiente.

Essas ferramentas simplificam tarefas como a **análise**, a **modificação** e a **geração** de dados JSON, proporcionando uma experiência mais eficaz para os desenvolvedores e para os engenheiros de dados.

5.4 XML e suas variantes

Definindo a estrutura em XML

XML Schema

O XML Schema define a estrutura de um documento XML, usando XML. Além de especificar a hierarquia e a ordem dos elementos, ele suporta a definição de tipos de dados para elementos e atributos. Isso adiciona uma camada adicional de robustez à validação de dados, tornando-o uma escolha mais avançada para situações que exigem uma maior precisão na descrição de tipos de dados XML.

DTD (*Document Type Definition*)

O DTD, assim como o XML Schema, define a estrutura de um documento XML. No entanto, diferencia-se deste último por não suportar tipos de dados. Sua principal característica é a sua capacidade de especificar a estrutura hierárquica do documento, a ordem dos elementos e os tipos de atributos permitidos. Apesar de sua simplicidade, DTD continua a ser uma escolha válida para alguns cenários devido à sua abordagem direta.

Navegação e consulta em documentos XML

XPath

É uma linguagem específica para a navegação em documentos XML. Com a XPath, é possível selecionar elementos e atributos com base nas suas localizações no documento, permitindo uma navegação eficiente e precisa.

XQuery

É uma linguagem de consulta que utiliza a XPath como base. Porém, enquanto XPath concentra-se na navegação, ela estende as suas capacidades ao permitir consultas mais complexas e elaboradas sobre estruturas XML. Juntas, essas linguagens oferecem poderosas ferramentas para interagir e extrair informações de documentos XML.

Inclusão e referência precisa

XInclude

É uma tecnologia que permite a inclusão de partes de outros documentos XML em um documento principal. Isso é útil para modularizar documentos XML e reutilizar partes específicas.

XPointer

É utilizado para referenciar partes específicas de documentos XML, oferecendo uma maneira precisa de apontar para localizações específicas em um documento. Essa capacidade é especialmente valiosa ao lidar com documentos XML extensos, permitindo referências diretas e eficientes.

5.4.1 Outras variantes do XML

XSLT (*Extensible Stylesheet Language Transformations*)

XSLT é uma linguagem baseada em XML usada para transformar documentos XML em diferentes formatos, como HTML ou outros documentos XML. Com ela, os desenvolvedores podem criar estilos ou regras que definem como os elementos XML devem ser apresentados ou transformados, adicionando uma camada de flexibilidade à exibição e à manipulação de dados.

Namespaces XML

Namespaces XML são fundamentais para fornecer elementos e atributos com nomes exclusivos em um documento XML. Eles são cruciais para evitar conflitos de nomes e garantir que diferentes partes de um documento possam ser diferenciadas corretamente. Ao definir namespaces, os desenvolvedores podem organizar e estruturar documentos XML de forma mais eficiente, promovendo a modularidade e a clareza.



Neste capítulo, nós entendemos que o JSON e XML são formatos de dados utilizados em APIs para estruturar informações. O JSON, uma alternativa mais simples ao XML, é integrado ao JavaScript, organizando dados em pares de nome e valor ou em listas de valores, sendo amplamente utilizado em transferências de dados entre sistemas via APIs e em requisições AJAX. Também vimos que o XML é flexível e hierárquico, sendo empregado em documentos estruturados e validações complexas. Estudamos que, ao comparar as suas semelhanças e diferenças, percebe-se que a escolha entre JSON e XML depende das necessidades do projeto.

Além disso, exploramos o JSON Schema, que define a estrutura e as regras de validação para dados JSON, os JSON Web Tokens (JWT), que representam informações de forma segura, e ferramentas como jq, que manipulam os dados JSON.

No caso do XML, exploramos o DTD e o XML Schema, que são mecanismos para definir a estrutura de um documento XML. Além disso, estudamos linguagens como XPath e XQuery, que facilitam a navegação e a consulta desses documentos. Vimos que o XSLT é outra ferramenta valiosa, permitindo a transformação de documentos XML em diferentes formatos, e que os Namespaces XML garantem a identificação única de elementos e atributos. Também abordamos o XInclude, usado para incluir partes de documentos XML em um principal, e o XPointer, para referenciar partes específicas do XML.

Os conceitos que trabalhamos aqui são muito importantes para você ampliar a sua compreensão sobre a aplicabilidade das tecnologias de dados JSON e XML.



XML

É uma linguagem de marcação extensível utilizada para estruturar dados e tem uma estrutura hierarquizada, baseada em *tags*. Além disso, a sua flexibilidade a torna aplicável em diversas situações, especialmente na representação de documentos estruturados.

DTD (Document Type Definition) e XML Schema

- Duas formas de definir estrutura em XML, diferindo na capacidade de suportar tipos de dados.

XPath e XQuery

- Linguagens para navegação e consulta em documentos XML.

XInclude e XPointer

- Tecnologias para inclusão e referência precisa em documentos XML.

XSLT (Extensible Stylesheet Language Transformativas)

- Linguagem baseada em XML para transformar documentos XML em diferentes formatos.

Namespaces XML

- Usados para fornecer elementos e atributos com nomes únicos.



ATIVIDADE DE FIXAÇÃO

1. Quais são os elementos essenciais em um objeto JSON?
2. Em que contexto a notação JSON é normalmente utilizada para a transferência de dados entre sistemas?
3. Como os elementos em um documento XML são geralmente organizados?
4. Liste as principais diferenças entre JSON e XML.
5. Qual é o principal propósito do JSON Schema?
6. Qual é a principal diferença entre DTD e XML Schema?
7. Qual é a função do XPath em relação ao XML?

- 8.** O que JWT representa e como as suas informações são codificadas?
- 9.** Qual é o propósito principal da linguagem XSLT?
- 10.** Por que os Namespaces XML são usados?

CAPÍTULO 06

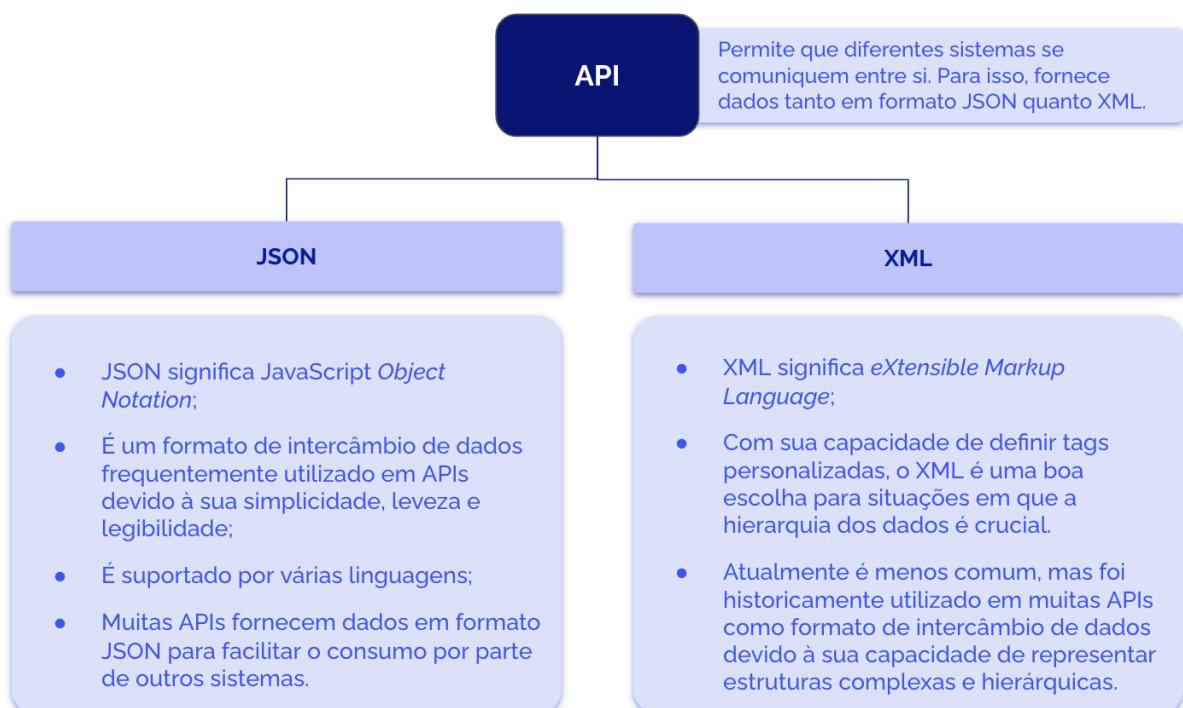
FORMATOS DE DADOS EM APIs

O que esperar deste capítulo:

- Saber como fazer *parsing* de dados JSON e XML em linguagens de programação;
- Compreender como acontece a manipulação de dados JSON de uma API.

Introdução

Antes de começar a nossa jornada no universo do *parsing*, vamos relembrar alguns conceitos que serão importantes para compreender melhor este tema.



6.1 Desvendando o parsing de dados JSON e XML

O que é parsing?



Parsing nada mais é do que o processo de **interpretar e converter dados de um formato para outro**. Aqui, vamos aprender como os dados em formatos **JSON e XML** são manipulados através de linguagem de programação.

6.1.1 Parsing de JSON em JavaScript

`JSON.parse()`

A linguagem JavaScript oferece uma ferramenta poderosa para lidar com dados JSON por meio do **objeto JSON**. Em particular, o método **JSON.parse()** permite analisar *strings* JSON e transformá-las em objetos JavaScript.

Vamos explorar esse universo fascinante juntos?

```
let obj = JSON.parse('{"name":"John", "age":30, "city":"New York"}');
console.log(obj.name); // Saída: John
```

No exemplo acima, a *string* JSON é convertida em um objeto JavaScript, permitindo-nos acessar as suas propriedades, assim como faríamos com qualquer outro objeto.

É crucial destacar que **JSON.parse()** lançará um **erro** se a *string* fornecida não estiver em um formato JSON válido. Portanto, é como um guardião atento, certificando-se de que apenas dados JSON legítimos são aceitos.

`Função reviver`

JSON.parse() pode ir além e aceitar um segundo argumento: uma **função reviver**. Essa função é convocada para cada item durante a análise e oferece a oportunidade de transformar valores antes de serem incorporados ao objeto resultante. Assim, é uma função utilizada para a manipulação durante o *parsing*.

```
let obj = JSON.parse('{"p": 5}', (key, value) =>
  typeof value === 'number'
    ? value * 2 // retorna o valor * 2 para números
    : value // retorna tudo sem alteração
);

console.log(obj.p); // Saída: 10
```

Neste exemplo, a nossa função **reviver** duplica todos os valores numéricos no objeto JSON, adicionando uma camada extra de personalização ao processo de *parsing*.

6.1.2 Exemplos de parsing de JSON em JavaScript

Objeto simples

Vamos começar com um exemplo básico ao converter uma **string JSON que representa um objeto simples**:

String JSON: comece com uma *string* JSON que representa um objeto. Por exemplo:

1

```
let jsonString = '{"nome": "João", "idade": 30}';
```

JSON.parse(): utilize o método **JSON.parse()** para converter a *string* JSON em um objeto JavaScript. Este método analisa a *string* JSON e retorna um objeto JavaScript correspondente. Veja o exemplo:

2

```
let objeto = JSON.parse(jsonString);
```

Acesso aos atributos: agora, você pode acessar os atributos do objeto JavaScript diretamente. No exemplo, você pode acessar os atributos "**nome**" e "**idade**".

3

```
console.log(objeto.nome); // Saída: João
console.log(objeto.idade); // Saída: 30
```

Array de objetos

Nas situações em que lidamos com uma coleção de objetos, o **JSON.parse()** também é poderoso:

String JSON que representa um array de objetos: começamos com uma *string* JSON que representa um *array* de objetos. Por exemplo:

1

```
let jsonString = '[{"nome": "João", "idade": 30}, {"nome": "Maria", "idade": 25}]';
```

2

JSON.parse(): usamos o método **JSON.parse()** para converter a *string* JSON em um *array* de objetos JavaScript. Este método analisa a *string* JSON e retorna um *array* de objetos correspondente. Veja o exemplo a seguir:

```
let arrayDeObjetos = JSON.parse(jsonString);
```

3

Acesso aos elementos do array: agora, podemos acessar os elementos do *array* e os atributos dos objetos dentro dele. Por exemplo:

```
console.log(arrayDeObjetos[0]); // Saída: { nome: 'João', idade: 30 }
console.log(arrayDeObjetos[1]); // Saída: { nome: 'Maria', idade: 25 }
```

4

Acesso aos atributos dos objetos dentro do array: podemos acessar os atributos dos objetos dentro do *array* da mesma forma que acessamos os atributos de um objeto simples. Por exemplo:

```
console.log(arrayDeObjetos[0].nome); // Saída: João
console.log(arrayDeObjetos[1].idade); // Saída: 25
```

Manipulação durante o parsing

Além disso, o método **JSON.parse()** permite a manipulação de dados durante o *parsing*.

Veja a seguir:

O método **JSON.parse()** aceita um segundo parâmetro opcional chamado "**reviver**", que é uma função usada para transformar os valores *parseados* antes de serem retornados. Isso permite a manipulação dos dados durante o processo de *parsing*.

Suponha que temos o seguinte JSON que representa datas como *strings*:

```
let jsonString = '{"data": "2024-02-27"}';
```

Por padrão, quando você analisa este JSON usando '**JSON.parse()**', a data é interpretada como uma *string*. No entanto, você pode querer que ela seja convertida em um objeto '**Date**'. Para fazer isso, você pode usar uma função **reviver**.

Veja como você pode fazer isso:

```
// Definindo a função reviver
function reviverChaves(chave, valor) {
  // Se o valor parecer uma data (no formato YYYY-MM-DD), converta-o em um objeto Date
  if (/^\d{4}-\d{2}-\d{2}$/.test(valor)) {
    return new Date(valor);
  }
  // Caso contrário, retorne o valor sem modificação
  return valor;
}

// Usando JSON.parse() com a função reviver
let objeto = JSON.parse(jsonObjectString, reviverChaves);

console.log(objeto.data); // Saída: Fri Feb 27 2024 00:00:00 GMT+0000 (Coordinated Universal Time)
console.log(objeto.data instanceof Date); // Saída: true
```

Neste exemplo, a função **reviver 'reviverChaves'** é definida para verificar se o valor *parseado* parece ser uma data no formato **YYYY-MM-DD**. Se for o caso, ele converte esse valor em um objeto '**Date**'. Caso contrário, ele retorna o valor sem modificação. Quando passamos essa função reviver como segundo argumento para '**JSON.parse()**', a data dentro do JSON é convertida em um objeto '**Date**' durante o *parsing*.

Esses exemplos demonstram a versatilidade do **JSON.parse()** e oferecem *insights* valiosos para o trabalho prático com dados JSON em JavaScript.

6.2 Manipulando dados JSON com JavaScript

Ao trabalhar com dados JSON em JavaScript, é essencial saber como manipular e interagir com essas estruturas corretamente. Vamos explorar algumas operações comuns:

Filtrando dados JSON

Para filtrar dados em um objeto JSON, podemos utilizar métodos como '**filter**'. Veja o exemplo a seguir:

```
let jsonData = [
  { "name": "John", "age": 25, "city": "New York" },
  { "name": "Alice", "age": 30, "city": "San Francisco" },
  { "name": "Bob", "age": 22, "city": "Los Angeles" }
];
let filteredData = jsonData.filter(item => item.age > 25);
console.log(filteredData);
// Saída: [{ "name": "Alice", "age": 30, "city": "San Francisco" }]
```

Ordenando dados JSON

A ordenação de dados pode ser feita usando o **método sort** e, aqui, vamos ver um exemplo simples de como ordenar um *array* de objetos JSON em JavaScript usando o método **sort()**:

```
// Dados JSON não ordenados
let dados = [
  { "nome": "Maria", "idade": 25 },
  { "nome": "João", "idade": 30 },
  { "nome": "Ana", "idade": 20 }
];

// Ordenando os dados pelo nome
dados.sort((a, b) => {
  // Comparando os nomes para determinar a ordem
  if (a.nome < b.nome) {
    return -1; // a vem antes de b
  } else if (a.nome > b.nome) {
    return 1; // a vem depois de b
  } else {
    return 0; // os nomes são iguais
  }
});

// Exibindo os dados ordenados
console.log(dados);
```

Neste exemplo, temos um *array* de objetos JSON chamado '**dados**', em que cada objeto tem uma chave "**nome**". Utilizamos o método '**sort()**' para ordenar os dados com base nos valores da chave "**nome**". A função de comparação passada para o método '**sort()**' compara os valores da chave "**nome**" de cada objeto e retorna -1 se o primeiro objeto deve vir antes do segundo, 1 se o primeiro deve vir depois do segundo, e 0 se os valores forem iguais.

Modificando dados JSON

Para modificar dados em um objeto JSON, basta acessar e atribuir novos valores. Por exemplo, vamos aumentar a idade de John em dois anos:

```
let sortedData = jsonData.sort((a, b) => a.name.localeCompare(b.name));

console.log(sortedData);

// Saída: [{ "name": "Alice", "age": 30, "city": "San Francisco" }, { "name": "Bob", "age": 22, "city": "Los Angeles" }, { "name": "John", "age": 25, "city": "New York" }]
```

6.3 Manipulando dados JSON com Python

A **biblioteca json em Python** oferece uma maneira simples e eficaz de manipular dados JSON. A seguir, vamos explorar como realizar algumas operações comuns:

Carregando dados JSON

Para carregar dados de uma *string* JSON, usamos **json.loads**:

```
import json

jsonString = '{"name": "John", "age": 25, "city": "New York"}'

jsonData = json.loads(jsonString)

print(jsonData)

# Saída: {'name': 'John', 'age': 25, 'city': 'New York'}
```

Modificando dados JSON

É simples modificar dados em Python. Vamos adicionar uma nova chave "**gender**" ao nosso objeto JSON:

```
jsonData['gender'] = 'Male'

print(jsonData)
# Saída: {'name': 'John', 'age': 25, 'city': 'New York', 'gender': 'Male'}
```

Convertendo para JSON

Para converter um objeto Python de volta para uma *string* JSON, usamos **json.dumps**:

```
jsonStringModified = json.dumps(jsonData)

print(jsonStringModified)
# Saída: '{"name": "John", "age": 25, "city": "New York", "gender": "Male"}'
```

A manipulação de dados JSON em Python é uma habilidade valiosa e a biblioteca json torna esse processo acessível e eficiente.

Comparação entre operações de manipulação de dados JSON em JavaScript e Python

Operações	JavaScript	Python
Carregar dados JSON	<code>JSON.parse(jsonString).</code>	<code>json.loads(jsonString).</code>
Modificar dados	<code>jsonData[key] = value.</code>	<code>jsonData[key] = value.</code>
Adicionar nova chave	<code>jsonData.newKey = value.</code>	<code>jsonData['newKey'] = value.</code>
Filtrar dados	<code>jsonData.filter(item => condition)..</code>	<code>filter(lambda item: condition, jsonData).</code>
Ordenar dados	<code>jsonData.sort((a, b) => compare).</code>	<code>sorted(jsonData, key=compare).</code>
Converter para JSON	<code>JSON.stringify(jsonData).</code>	<code>json.dumps(jsonData).</code>

Parsing de XML em linguagens de programação

O *parsing* de dados XML é uma tarefa comum na programação, permitindo que desenvolvedores interajam com informações estruturadas em documentos XML. A seguir, vamos explorar como diferentes linguagens lidam com esse processo essencial.

JavaScript

Em JavaScript, o *parsing* de XML geralmente é realizado com a ajuda da interface '**DOMParser**'. Aqui está um exemplo simples:

```
let xmlString = '<book><title>Introduction to XML</title><author>John Doe</author></book>';
let parser = new DOMParser();
let xmlDoc = parser.parseFromString(xmlString, 'text/xml');

console.log(xmlDoc.getElementsByTagName('title')[0].childNodes[0].nodeValue); // Saída: Introduction to XML
```

Neste exemplo, criamos uma *string* XML e a convertemos em um documento XML manipulável usando '**DOMParser**'.

Python

Em Python, a biblioteca `xml.etree.ElementTree` é frequentemente utilizada para *parsing* de XML. Aqui está um exemplo:

```
import xml.etree.ElementTree as ET

xmlString = '<book><title>Introduction to XML</title><author>John Doe</author></book>'
root = ET.fromstring(xmlString)

print(root.find('title').text) # Saída: Introduction to XML
```

No exemplo em Python, usamos `ET.fromstring` para obter um elemento raiz a partir da *string* XML.

Java

Em Java, a biblioteca `javax.xml.parsers` é comumente usada para *parsing* de XML. Aqui está um exemplo:

```
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.xml.sax.InputSource;

String xmlString = "<book><title>Introduction to XML</title><author>John Doe</author></book>";
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
InputSource is = new InputSource(new StringReader(xmlString));
Document xmlDoc = builder.parse(is);

Element titleElement = (Element) xmlDoc.getElementsByTagName("title").item(0);
System.out.println(titleElement.getTextContent()); // Saída: Introduction to XML
```

Neste exemplo em Java, criamos um **Document** a partir da *string* XML usando **DocumentBuilder**.

Desenvolvendo um catálogo de livros



Descrição

Imagine que você está construindo um aplicativo para um catálogo de livros *on-line* e que a sua tarefa é decidir entre JSON e XML para representar as informações dos livros na API. Este desafio prático permitirá explorar as características e aplicabilidades dos dois formatos.



Objetivos

- Utilizar JSON e XML para representar informações detalhadas sobre livros;
 - Implementar funcionalidades para buscar livros por categoria, autor ou título;
 - Comparar e contrastar as vantagens e desvantagens de JSON e XML na contextura de uma API de catálogo de livros.
- 

Orientações

- Estruture dados sobre livros em ambos os formatos, destacando as diferenças na sintaxe e estrutura;
- Desenvolva funcionalidades de busca que permitam aos usuários encontrarem livros por categoria, autor ou título;
- Crie *endpoints* distintos para fornecer dados em JSON e XML. Explore as características específicas de cada formato na construção desses *endpoints*;
- Avalie o desempenho da API ao utilizar JSON e XML para fornecer informações sobre os livros. Considere fatores como tamanho da resposta, facilidade de leitura para desenvolvedores e tempo de resposta para isso.



RESUMO

Neste capítulo, aprendemos que, ao lidar com dados no formato JSON em JavaScript, utilizamos o método **JSON.parse()** para converter *strings* JSON em objetos JavaScript. Além disso, vimos que é possível empregar uma função **reviver** durante o *parsing* para alterar valores conforme necessário.

Já em relação à manipulação de dados JSON em JavaScript, aprendemos que podemos filtrar dados específicos usando métodos como **filter**, organizar dados com o método **sort**, ou acessar e modificar valores diretamente em objetos JSON. Estudamos que, em Python, usamos **json.loads()** para carregar dados JSON em objetos Python, e **json.dumps()** para converter objetos Python de volta para *strings* JSON. Podemos também adicionar novas chaves e valores a objetos JSON em Python.

Fizemos, ainda, uma comparação entre as operações de manipulação de dados JSON em JavaScript e Python, destacando as diferenças entre as duas linguagens.

Por fim, abordamos o *parsing* de XML em diversas linguagens de programação, como JavaScript, Python e Java.



ATIVIDADE DE FIXAÇÃO

1. O que é *parsing* de JSON? Explique o conceito e a importância do *parsing* de JSON em aplicações *web*.
2. Como o método **JSON.parse()** funciona em JavaScript? Forneça um exemplo de código que demonstra o seu uso.
3. O que é uma função **reviver** no contexto do método **JSON.parse()**? Dê um exemplo de como uma função **reviver** pode ser usada durante o *parsing* de JSON.
4. Como você faria o *parsing* de um *array* de objetos JSON em JavaScript? Escreva um exemplo de código.

5. Como você manipularia dados durante o *parsing* de JSON em JavaScript? Dê um exemplo de código que demonstra isso.
6. O que é parsing de XML? Explique como diferentes linguagens de programação, como JavaScript, Python e Java, o realizam.
7. Como você usaria a interface DOMParser em JavaScript para fazer o *parsing* de XML? Forneça um exemplo de código.
8. Como você usaria a biblioteca **xml.etree.ElementTree** em Python para fazer o *parsing* de XML? Forneça um exemplo de código.
9. Como você usaria a biblioteca **javax.xml.parsers** em Java para fazer o *parsing* de XML? Forneça um exemplo de código.
10. Quais são as diferenças e semelhanças entre JSON e XML em termos de *parsing*? Discuta as vantagens e desvantagens de cada um.

CAPÍTULO 07

TIPOS DE AUTENTICAÇÃO EM APIs: CHAVES DE API, TOKENS, OAUTH 2.0 E AUTENTICAÇÃO DE TERCEIROS

O que esperar deste capítulo:

- Identificar as chaves de API e os *tokens* e utilizá-los em partes específicas do desenvolvimento do app;
- Utilizar o OAuth 2.0 para trabalhar autenticações, inclusive de terceiros.

As APIs (Interfaces de Programação de Aplicações) desempenham um papel fundamental na conectividade e na integração entre sistemas e serviços na web. Para garantir a **segurança** e o **controle de acesso**, é crucial implementar **mecanismos de autenticação** eficientes.

Por isso, neste capítulo, vamos explorar os principais métodos de autenticação em APIs, incluindo **chaves de API, tokens, OAuth 2.0 e autenticação de terceiros**. Além disso, vamos trazer exemplos práticos para que os conceitos sejam melhor entendidos.

7.1 Chaves de API

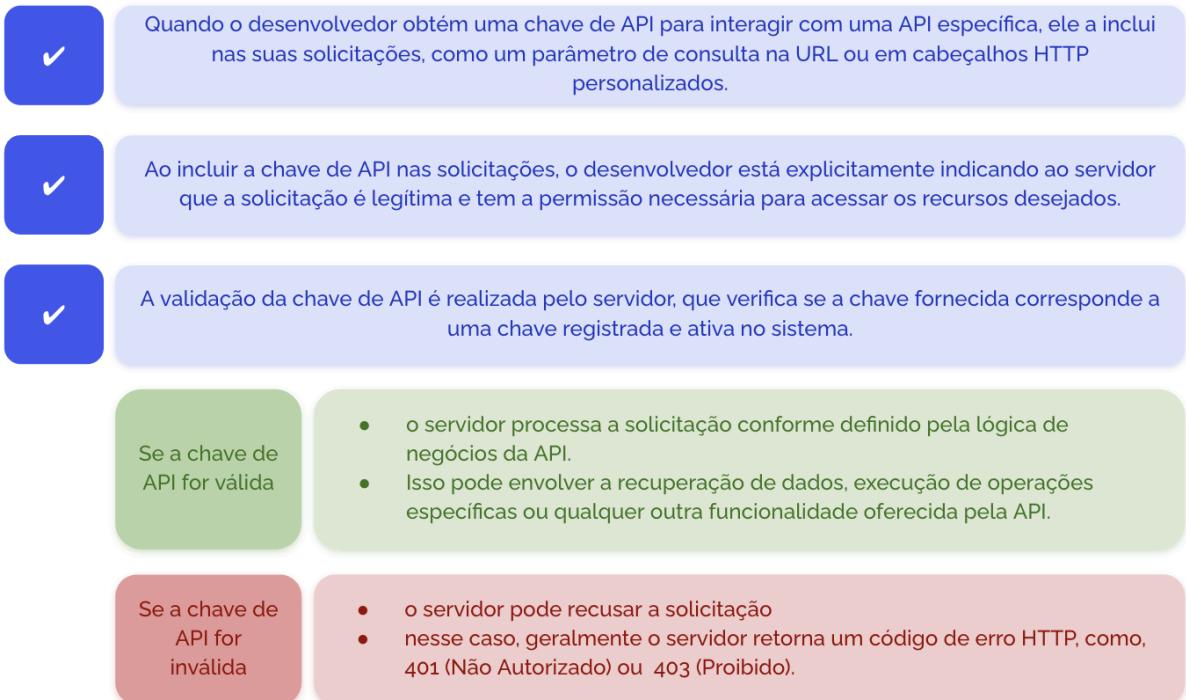
O que são chaves de API?

- Chaves de API ou API *keys* são *strings* de caracteres que funcionam como **credenciais para autenticar** uma aplicação ou usuário ao acessar uma API;
- Frequentemente, elas são usadas como uma forma simples de **controle de acesso**, permitindo que os desenvolvedores identifiquem e autorizem solicitações provenientes de uma aplicação específica.

Como funcionam as chaves de API?

As chaves de API desempenham um papel essencial na autenticação de usuários e em aplicações ao acessar APIs. Essa chave é como uma credencial única que o servidor da API utiliza para autenticar e autorizar a requisição de um desenvolvedor.

Mas, como isso acontece?



7.2 Práticas de segurança

A validação da chave de API é uma camada crítica de segurança, garantindo que apenas usuários autorizados possam acessar os recursos protegidos pela API. Para isso, os desenvolvedores devem tomar algumas medidas, como:

- manter suas chaves de API em segredo;
- Fazer um armazenamento seguro e rotação periódica de chaves, a fim de diminuir potenciais riscos, como o vazamento acidental da chave ou o uso indevido por terceiros não autorizados;
- além disso, muitas APIs também oferecem funcionalidades avançadas, como limites de taxa e controle de acesso baseado em chave, para aprimorar ainda mais a segurança e o gerenciamento do uso da API.

Exemplo prático

Imagine que você desenvolveu uma aplicação web simples que consome uma API de previsão do tempo. Antes de fazer as solicitações à API, você precisa **obter uma chave de API do provedor do serviço meteorológico**. O código JavaScript a seguir ilustra como a chave de API pode ser usada para autenticar as solicitações:

```

const apiKey = 'SuaChaveDeAPI';
const apiUrl = 'https://api.servico-meteorologico.com/previsao';

// Exemplo de solicitação utilizando a chave de API
fetch(`${apiUrl}?key=${apiKey}`)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Erro:', error));

```

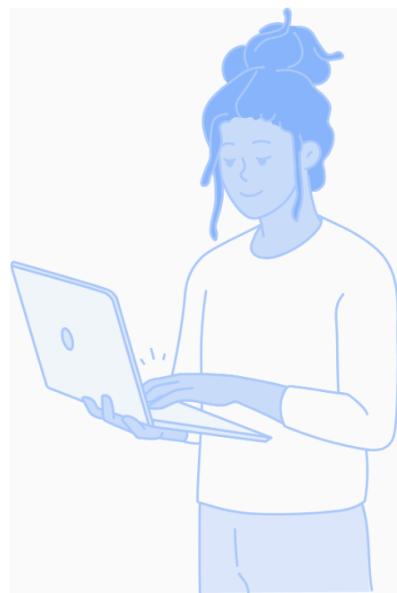
Fonte: Autor, 2024.

Explicando o código:

Variável	Descrição
apiKey	Constante que armazena sua chave de API pessoal.
apiUrl	URL da API que você está acessando para obter dados de previsão do tempo.
fetch	Função usada para fazer uma solicitação à API, incluindo a chave da API na URL da solicitação.
.then()	Método que converte a resposta em JSON.
.then()	Método que imprime os dados JSON no console.
.catch()	Captura e registra qualquer erro que ocorra durante o processo.

Chaves API no dia a dia

Aplicativos de pagamento, como carteiras digitais ou sistemas de pagamento online, usam chaves de API para se conectar aos sistemas de processamento de pagamentos e autorizar transações seguras entre usuários e comerciantes.



7.2.1 Tokens

O que são tokens?

- Os *tokens* são *strings* de caracteres que representam a autorização concedida para um usuário ou aplicação específica;
- Eles são amplamente utilizados em **autenticação e autorização**, proporcionando uma maneira mais **segura** e **flexível** de controlar o acesso a recursos protegidos.

Como os tokens funcionam?

- ✓ Após a autenticação das credenciais do usuário com nome e senha, por exemplo, o servidor gera um token exclusivo para esse usuário
- ✓ Sempre que o usuário for fazer uma solicitação ao servidor para acessar um recurso protegido, o cliente inclui esse token na solicitação.
- ✓ O servidor verifica a validade do token antes de fornecer acesso aos recursos solicitados. Essas validações podem incluir: garantir que o token seja válido; que continua dentro do prazo de validade; e que ele tem permissão para acessar os recursos solicitados
- ✓ Se o token for válido e autorizado, o servidor permite o acesso aos recursos solicitados pelo usuário. Isso pode incluir exibir informações protegidas, enviar dados ou realizar qualquer outra ação permitida para esse usuário.



Periodicamente, o token pode precisar ser **renovado** para garantir a segurança. Isso pode acontecer automaticamente pelo servidor ou exigir que o usuário faça login novamente para obter um novo token.

Exemplo prático

Vamos considerar um cenário em que um desenvolvedor criou uma aplicação de compartilhamento de fotos que utiliza uma API para armazenar e recuperar imagens. Após o *login* bem-sucedido, o servidor retorna um *token* JWT (JSON Web Token). No exemplo a seguir, o desenvolvedor o utiliza para autenticar solicitações de *upload* e *download* de imagens:

```
const userToken = 'SeuTokenJWT';

const apiEndpoint = 'https://api.servico-de-imagens.com';

// Exemplo de solicitação utilizando o token JWT

// Upload de imagem

fetch(`${apiEndpoint}/upload`, {
  method: 'POST',
  headers: {
    'Authorization': `Bearer ${userToken}`,
    'Content-Type': 'multipart/form-data',
  },
  body: formData // Objeto FormData contendo a imagem
})
```

```
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Erro:', error));
```

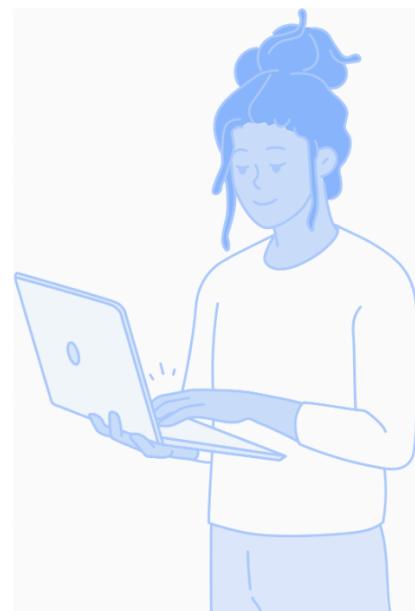
// Download de imagem

```
fetch(`${apiEndpoint}/download?id=123`, {
  headers: {
    'Authorization': `Bearer ${userToken}`,
  },
})
```

```
.then(response => response.blob())
.then(imageBlob => displayImage(imageBlob))
.catch(error => console.error('Erro:', error));
```

Tokens no dia a dia

Em jogos online, os jogadores frequentemente recebem tokens de autenticação depois de fazerem login. Esses tokens são usados para confirmar a identidade do jogador durante as sessões de jogo e permitir acesso a recursos exclusivos do jogo.



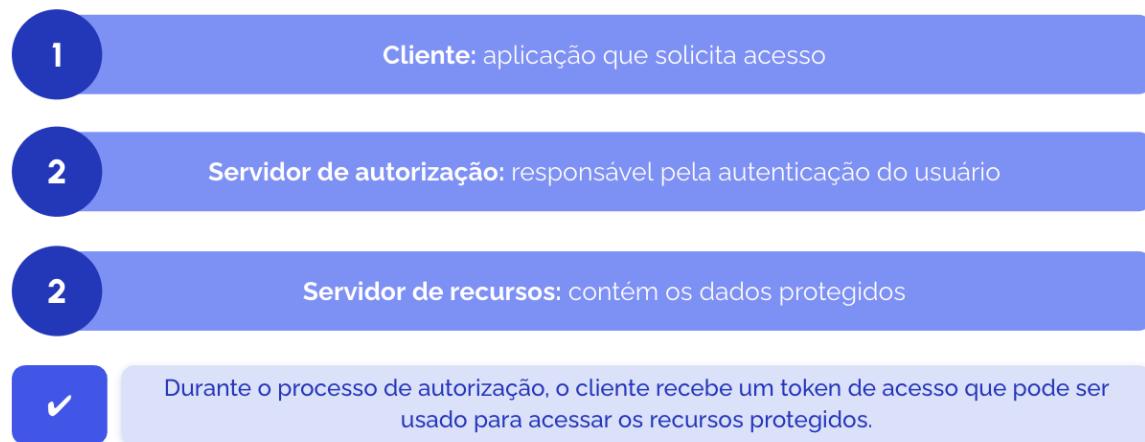
7.3 OAuth 2.0

O que é OAuth 2.0?

- OAuth 2.0 é um protocolo de autorização amplamente adotado na web para conceder acesso limitado a recursos, sem a necessidade de compartilhar credenciais diretas;
- Com ele, os usuários podem conceder permissões específicas para que aplicativos acessem os seus dados em outros serviços, como em redes sociais ou serviços de armazenamento na nuvem, de forma segura e controlada, sem a necessidade de fornecer as suas senhas.

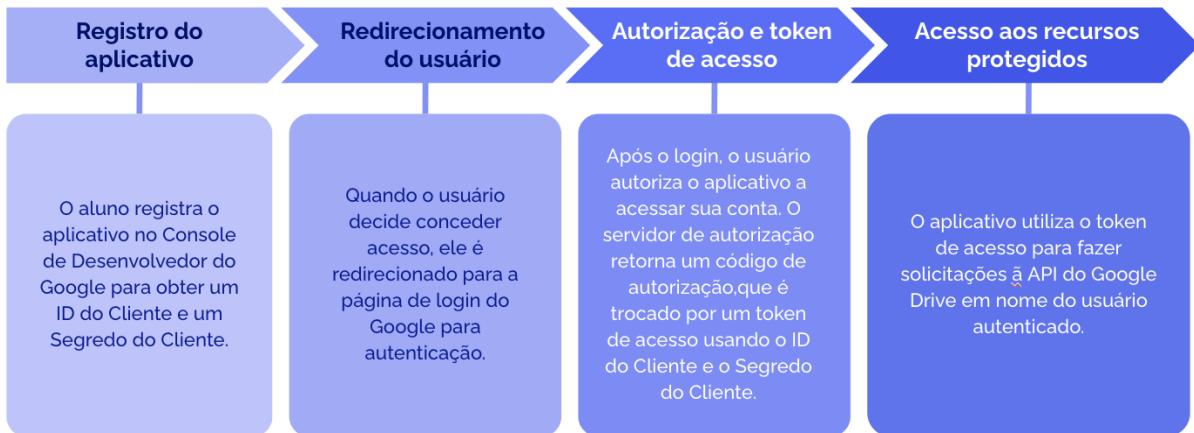
Como funciona o OAuth 2.0?

O OAuth 2.0 envolve três partes principais:



Exemplo prático

Imagine que você esteja desenvolvendo um aplicativo que precisa acessar a conta do Google Drive de um usuário para realizar operações de leitura e gravação de arquivos. Nesse cenário, o passo a passo do OAuth 2.0 seria o seguinte:



```
// Exemplo de fluxo OAuth 2.0 com JavaScript (usando uma biblioteca como OAuth.js)

const oauth = OAuth({
  client: {
    id: 'SeuIDdoCliente',
    secret: 'SeuSegredoDoCliente',
  },
  auth: {
    tokenHost: 'https://accounts.google.com',
    authorizePath: '/o/oauth2/auth',
    tokenPath: '/o/oauth2/token',
  },
});

// Redirecionamento do usuário para a página de login do Google

const authorizationUri = oauth.authorizationCode.authorizeURL({
```

```
    redirect_uri: 'https://seu-aplicativo.com/callback',  
    scope: 'https://www.googleapis.com/auth/drive',  
  );  
  
  window.location.href = authorizationUri;
```

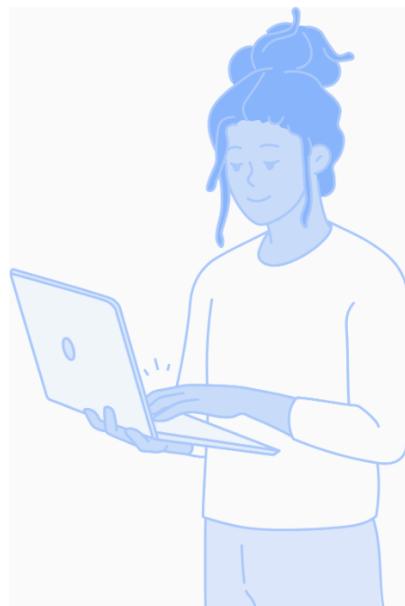


E aí, conseguiu acompanhar a leitura desse código?

Você deve analisar os códigos como um detetive digital para descobrir o que cada linha está fazendo.

OAuth 2.0. no dia a dia

Você decide se inscrever em um novo serviço de streaming de música usando sua conta do Google. Quando você clica em "Entrar com o Google", o serviço de streaming redireciona você para a página de login do Google. Após fazer login com suas credenciais do Google, o Google solicita sua permissão para **compartilhar informações com o serviço de streaming**. Uma vez concedida a permissão, o serviço de streaming recebe um **token OAuth do Google**, permitindo acesso limitado aos seus dados, como nome e endereço de e-mail, para personalizar sua experiência no serviço de streaming.

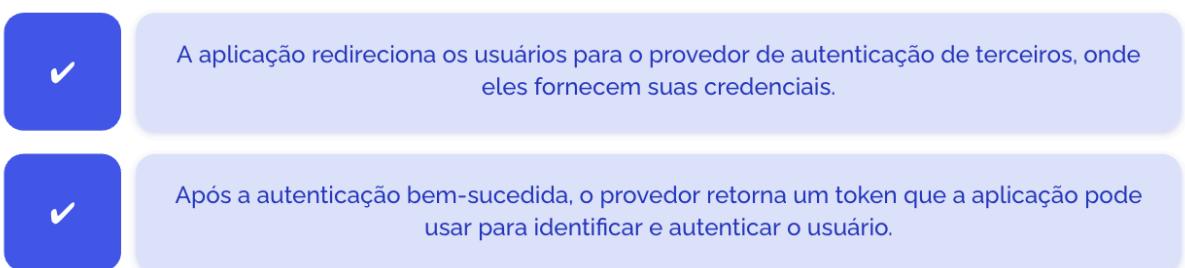


7.4 Autenticação de terceiros

O que é autenticação de terceiros?

- Ela envolve a delegação da autenticação de usuários para um provedor externo. Ou seja, é um processo em que um serviço confia na autenticação realizada por outro serviço.;
- Em outras palavras, um aplicativo ou *site* permite que os usuários façam *login* utilizando as credenciais de uma outra plataforma, como Google, Facebook ou GitHub.

Como funciona a autenticação de terceiros?



Exemplo prático

Suponha que um desenvolvedor esteja criando um *site* e queira permitir que os usuários façam *login* usando as suas contas do Google. O exemplo a seguir mostra como isso pode ser implementado com JavaScript usando a API de autenticação do Google:

```
// Carregar a API de Autenticação do Google
gapi.load('auth2', function() {
    gapi.auth2.init({
        client_id: 'SeuIDdoCliente',
    });
});

// Função de login com Google
```

```

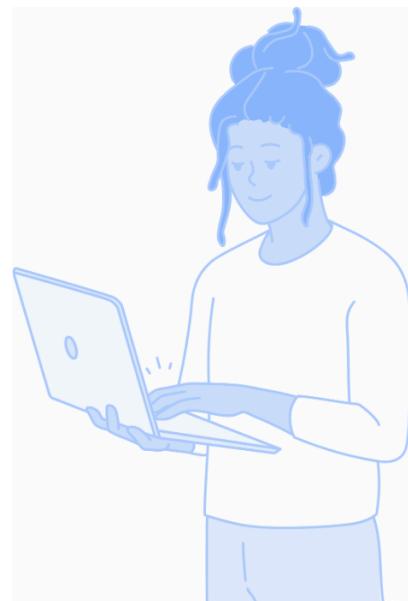
function loginWithGoogle() {
  const auth2 = gapi.auth2.getAuthInstance();
  auth2.signIn().then(function(googleUser) {
    const idToken = googleUser.getAuthResponse().id_token;
    // Enviar o token para o servidor para autenticação do usuário
    authenticateUser(idToken);
  });
}

// Função para autenticar o usuário no servidor
function authenticateUser(idToken) {
  // Implementar a lógica de autenticação no servidor
  // Verificar a validade do token e criar uma sessão de usuário
  // Retornar um token de acesso para o cliente, se necessário
}

```

Autenticação de terceiros no dia a dia

Você visita um site de comércio eletrônico e decide fazer login usando sua conta do Facebook em vez de criar uma nova conta. Ao clicar em "Login com o Facebook", o site de comércio eletrônico redireciona você para a página de login do Facebook. Depois de fazer login com suas credenciais do Facebook, o **Facebook envia uma confirmação de autenticação de volta para o site de comércio eletrônico, permitindo que você accesse sua conta no site sem precisar criar um novo nome de usuário e senha.**

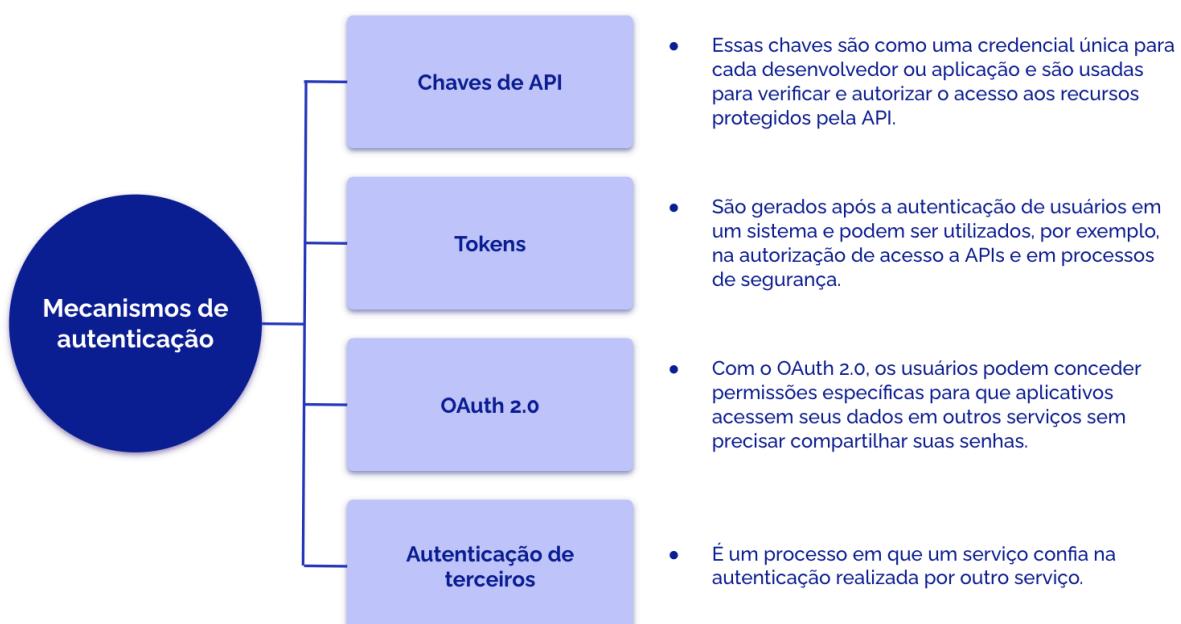




Neste capítulo, nós aprendemos que, no ecossistema das APIs, a autenticação desempenha um papel vital na segurança e na integridade dos sistemas. Dois métodos fundamentais são as chaves de API e os *tokens*. As chaves de API, *strings* de caracteres que autenticam aplicações, oferecem um controle direto de acesso, enquanto os *tokens*, que representam uma autorização, proporcionam flexibilidade e segurança ao autenticar usuários.

Além disso, estudamos que o OAuth 2.0 e a autenticação por terceiros, possibilita autorizações e autenticações sem que os usuários compartilhem as suas credenciais diretamente na aplicação. Com exemplos práticos utilizando HTML, CSS e JavaScript, os desenvolvedores iniciantes podem compreender e implementar esses métodos, capacitando-se a construir aplicações *web* seguras e integradas.

Para ter um resumo desses mecanismos de autenticação, observe o mapa a seguir:





ATIVIDADE DE FIXAÇÃO

- 1.** O que são chaves de API?
 - a. *Strings* de caracteres usadas em códigos HTML.
 - b. Credenciais usadas para autenticação em APIs.
 - c. Identificadores de sessão em JavaScript.
 - d. Nomes de variáveis em CSS.

- 2.** Qual é a principal função dos *tokens* em autenticação de APIs?
 - a. Controlar o acesso a recursos protegidos.
 - b. Definir estilos em páginas *web*.
 - c. Armazenar *cookies* no navegador.
 - d. Gerenciar o tráfego de rede.

- 3.** O que OAuth 2.0 permite fazer?
 - a. Acesso direto a recursos sem autenticação.
 - b. Autenticação de terceiros de forma insegura.
 - c. Conceder acesso limitado a recursos sem compartilhar credenciais diretas.
 - d. Troca de senhas entre servidores.

- 4.** Como as chaves de API são geralmente enviadas nas solicitações HTTP?
 - a. No corpo da mensagem.
 - b. Como parâmetros de consulta na URL.
 - c. Em *cookies*.
 - d. Em cabeçalhos personalizados.

- 5.** Qual é o principal benefício da autenticação de terceiros?
 - a. Maior controle sobre a segurança.
 - b. Simplificação do processo de autenticação.
 - c. Necessidade de compartilhar credenciais diretas.
 - d. Restrição de acesso a APIs.

6. Explique como as chaves de API são usadas para autenticar uma solicitação em uma API. Dê um exemplo prático.
7. Compare os *tokens* e as chaves de API em termos de funcionalidade e flexibilidade. Quando você escolheria usar um no lugar do outro?
8. Descreva o fluxo básico do OAuth 2.0 e como ele permite a autenticação de terceiros. Dê um exemplo de aplicação prática.
9. Como os JWTs (JSON Web Token) são diferentes de outros tipos de *tokens* em termos de estrutura e utilização? Dê um exemplo de como um JWT pode ser utilizado em autenticação.
10. Quais são os possíveis riscos associados à autenticação de terceiros? Explique medidas que podem ser tomadas para mitigar esses riscos.

CAPÍTULO 08

CONSIDERAÇÕES DE SEGURANÇA AO ACESSAR APIs EXTERNAS E AUTENTICAÇÃO COM CHAVES DE API

O que esperar deste capítulo:

- Implementar as principais considerações de segurança ao acessar APIs externas;
- Gerenciar e proteger as chaves de API de forma segura nas suas aplicações.

Em um cenário digital cada vez mais interconectado, a integração de APIs externas desempenha um papel fundamental no enriquecimento das funcionalidades das aplicações *web*. Contudo, esse benefício vem acompanhado da responsabilidade de assegurar a **integridade** e a **confidencialidade** dos dados manipulados.

À medida que a complexidade das aplicações *web* aumenta, a conscientização sobre a importância da **segurança na interação com APIs externas** se torna ainda mais importante. Estar atento a isso significa estar preparado para enfrentar os desafios do desenvolvimento *web* moderno, criando aplicações que não apenas atendem às expectativas funcionais, mas também estabelecem uma base sólida de segurança digital.

8.1 Considerações de segurança ao acessar APIs externas

Já sabemos que, ao integrar APIs externas em aplicações *web*, é crucial considerar a segurança para proteger os dados sensíveis e garantir um ambiente confiável, não é? Então, agora, vamos abordar as principais considerações de segurança ao acessar APIs externas, com foco especial na autenticação utilizando **chaves de API**. Vamos lá?

8.1.1 HTTPS (SSL/TLS)

Ao realizar requisições para APIs externas, é crucial garantir a segurança por meio do protocolo HTTPS (SSL/TLS). Isso pode ser implementado facilmente do lado do

cliente em JavaScript. Por exemplo, ao usar a biblioteca **fetch** para fazer uma requisição, a seguinte configuração pode ser aplicada:

```
fetch('https://api.externa.com/dados', {  
  
  method: 'GET',  
  
  headers: {  
  
    'Content-Type': 'application/json',  
  
    // Adicione outras headers, se necessário  
  
  },  
  
  // Adicione a opção para garantir que a requisição seja segura  
  
  mode: 'cors',  
  
})  
  
.then(response => response.json())  
  
.then(data => console.log(data))  
  
.catch(error => console.error('Erro na requisição:', error));
```

Explicando os códigos

URL da API externa

A primeira linha do código define a URL da API externa que você deseja acessar. No exemplo, a URL é `https://api.externa.com/dados`.

Configuração da requisição

- **Método HTTP:** a requisição é feita usando o método HTTP **GET** (indicado pelo parâmetro `method: 'GET'`);
- **Headers:** o objeto `headers` contém informações sobre o tipo de conteúdo que está sendo enviado. No exemplo, estamos definindo o cabeçalho `'Content-Type'` como `'application/json'`;
- **Modo CORS:** o parâmetro `mode: 'cors'` permite que a requisição seja feita de forma segura, seguindo a política de mesma origem, assim como aponta o seu nome (CORS é uma sigla para *Cross-Origin Resource Sharing*). Isso é importante para evitar problemas de segurança ao fazer requisições de domínios diferentes.

Tratamento da resposta

- O método `fetch` retorna uma promessa (`promise`) que representa a resposta da requisição;
- O método `.then(response => response.json())` converte a resposta em formato JSON;
- O método `.then(data => console.log(data))` exibe os dados obtidos no console;
- O método `.catch(error => console.error('Erro na requisição:', error))` trata erros caso a requisição falhe.

Segurança e certificados SSL/TLS

- O uso de HTTPS garante que a comunicação entre o cliente (navegador) e o servidor da API seja **criptografada e segura**;
- Certificados SSL/TLS são essenciais para verificar a **autenticidade do servidor** e garantir que a conexão não seja interceptada por terceiros mal-intencionados.

8.1.2 Validação de entradas

A validação de entradas pode ser realizada no lado do cliente e do servidor em JavaScript. No exemplo, ao enviar dados para uma API, é possível **validar os campos do formulário** antes de realizar a requisição. Aqui está um exemplo simples usando JavaScript puro:

```
const inputUsuario = document.getElementById('usuario');
const inputSenha = document.getElementById('senha');

// Validar campos antes da requisição

if (inputUsuario.value.trim() !== '' && inputSenha.value.trim() !== '') {
    // Realizar a requisição apenas se os campos forem válidos
    fetch('https://api.externa.com/login', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({
            usuario: inputUsuario.value,
            senha: inputSenha.value,
        }),
    })

    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error('Erro na requisição:', error));
}
```

```
    } else {
        console.error('Campos inválidos. Preencha todos os campos corretamente.');
    }
}
```

Explicando os códigos

Campos de entrada

O código começa definindo duas variáveis: `inputUsuario` e `inputSenha`. Essas variáveis são obtidas usando o método `document.getElementById` para buscar os elementos HTML com os IDs correspondentes ('`usuario`' e '`senha`').

Validação dos campos

Antes de fazer a requisição à API externa, o código verifica se os campos de entrada não estão vazios. Isso é feito com a seguinte condição:

```
if (inputUsuario.value.trim() !== '' && inputSenha.value.trim() !== '') {
    // Realizar a requisição apenas se os campos forem válidos
    // ...
} else {
    console.error('Campos inválidos. Preencha todos os campos corretamente.');
}
```

- `inputUsuario.value.trim() !== ''` verifica se o valor do campo de usuário (após remover espaços em branco no início e no final) não é uma `string` vazia;
- `inputSenha.value.trim() !== ''` faz o mesmo para o campo de senha;
- Se os dois campos não estiverem vazios, a requisição à API será realizada. Caso contrário, uma mensagem de erro é exibida no console.

Requisição à API externa

- O código utiliza o método `fetch` para fazer uma requisição HTTP `POST` para a URL `https://api.externa.com/login`;
- O cabeçalho da requisição é configurado com '`Content-Type: application/json`';
- O corpo da requisição é um objeto JSON criado com base nos valores dos campos de entrada (`usuario` e `senha`);
- A resposta da requisição é convertida para JSON usando `.then(response => response.json())`;
- Os dados obtidos são exibidos no console com `.then(data => console.log(data))`;
- Em caso de erro na requisição, a mensagem de erro é exibida no console com `.catch(error => console.error('Erro na requisição:', error))`.

8.1.3 Rate limiting

Ao controlar a frequência das requisições, o respeito aos **limites de taxa** pode ser implementado no lado do cliente.



O **rate limiting** é uma estratégia para restringir o tráfego de rede recebido por uma aplicação, limitando a quantidade de requisições realizadas em um certo período de tempo. Geralmente, isso é feito através do rastreamento de endereços IP que realizam as requisições, junto ao período de tempo delas. Quando a quantidade de requisições excede o limite permitido dentro do intervalo de tempo definido, novas requisições não são completadas por um período adicional.

A seguir, veja um exemplo simples em JavaScript usando a função **setTimeout** para simular um *rate limiting*:

```
const fazerRequisicaoComRateLimit = () => {

    // Realizar a requisição apenas após 1 segundo

    setTimeout(() => {

        fetch('https://api.externa.com/dados')

        .then(response => response.json())

        .then(data => console.log(data))

        .catch(error => console.error('Erro na requisição:', error));

    }, 1000); // 1 segundo de intervalo

};

// Exemplo de uso da função com rate limiting

fazerRequisicaoComRateLimit();
```

Explicando os códigos



Em resumo, esse código simula um *rate limiting* ao atrasar a execução da requisição em um segundo. Isso pode ser útil para evitar sobrecarga de servidores ou proteger contra atividades maliciosas, como ataques de *bots*.

8.1.4 CORS (Cross-Origin Resource Sharing)

Permitir o *Cross-Origin Resource Sharing* (CORS) de forma adequada é crucial para a segurança ao acessar APIs externas. Alguns dos motivos para isso são:

- **prevenção de ataques de origem cruzada (XSS):** o CORS ajuda a evitar ataques de origem cruzada, em que um *site* malicioso tenta acessar recursos em outro domínio. Sem ele, um *site* mal-intencionado poderia fazer solicitações HTTP em nome do usuário, comprometendo a segurança;
- **controle de acesso baseado em origem:** o CORS permite que o servidor especifique quais origens (domínios) têm permissão para acessar os seus recursos. Isso é importante para limitar o acesso apenas a domínios confiáveis e evitar solicitações não autorizadas;

- **proteção de cookies e autenticação:** com CORS ativado, os navegadores modernos não enviam automaticamente *cookies* e credenciais (como *tokens* de autenticação) em solicitações de origens diferentes. Isso protege contra vazamento de informações confidenciais;
- **evitar o vazamento de dados sensíveis:** sem CORS, um *site* malicioso poderia fazer solicitações AJAX para APIs externas em nome do usuário, expondo dados sensíveis. O CORS impede que isso aconteça, garantindo que apenas domínios permitidos accessem os recursos;
- **segurança de terceiros:** ao usar APIs de terceiros, como serviços de pagamento ou redes sociais, o CORS garante que apenas os domínios autorizados possam interagir com essas APIs. Isso protege os usuários e os dados da aplicação.

Para controlar as políticas de CORS no lado do servidor, o Node.js e *Express* pode ser útil. O código a seguir habilita o CORS para todas as origens:

```
const express = require('express');
const cors = require('cors');
const app = express();

// Habilitar CORS para todas as origens
app.use(cors());

// Restante do código do servidor...
```

Explicando os códigos

Importação de módulos	O código começa importando os módulos necessários do Node.js. Express é um <i>framework web</i> para Node.js, e CORS é um <i>middleware</i> que lida com as políticas de <i>Cross-Origin Resource Sharing</i> .
Criação de uma aplicação express	A variável app é criada usando a função <code>express()</code> , que inicializa uma instância da aplicação Express.
Habilitando o CORS para todas as origens	A linha <code>app.use(cors())</code> ativa o CORS para todas as origens. Isso significa que o servidor permitirá solicitações de qualquer domínio, independentemente da origem.
Restante do código do servidor	O comentário "Restante do código do servidor..." indica que o restante do código específico da aplicação deve ser inserido aqui. Isso pode incluir rotas, manipuladores de solicitação, lógica de negócios e outras configurações.

8.1.5 Monitoramento e logging

O *logging*, ou registro de atividades, é fundamental para a segurança ao acessar APIs externas. A seguir, veja alguns motivos pelos quais o *logging* é importante:

- **deteção de atividades suspeitas:** os *logs* registram todas as atividades relacionadas à API, incluindo solicitações, respostas e erros. Isso permite detectar comportamentos anômalos ou atividades suspeitas, como tentativas de invasão ou acessos não autorizados;
- **investigação de incidentes:** quando ocorre um problema de segurança, os *logs* são essenciais para investigar o incidente. Eles fornecem informações detalhadas sobre o que aconteceu, quais endpoints foram acessados e quais dados foram transferidos;
- **conformidade com políticas de Segurança:** os *logs* ajudam a garantir que a API esteja em conformidade com as políticas de segurança estabelecidas. Eles permitem rastrear se as medidas de segurança estão sendo aplicadas corretamente;
- **monitoramento de integridade da API:** analisar regularmente os *logs* ajuda a manter a integridade da API. Qualquer atividade incomum ou tentativa de exploração pode ser identificada e tratada rapidamente;
- **resposta a incidentes em tempo real:** com *logs* detalhados, é possível responder a incidentes de segurança em tempo real. Isso é crucial para minimizar danos e proteger os dados dos usuários.

Implementar monitoramento e *logging* pode ser feito no lado do servidor, registrando informações relevantes. Um exemplo usando Node.js e Express para *logging* pode ser:

```
const express = require('express');
const app = express();
// Middleware para logging
app.use((req, res, next) => {
    console.log(`Requisição: ${req.method} ${req.url}`);
    next();
});

// Restante do código do servidor...
```

Explicando os códigos

Importação do módulo express

- A primeira linha `const express = require('express');` importa o módulo *Express* no seu aplicativo Node.js;
- O módulo *Express* é uma estrutura de aplicativo da *web* que simplifica a criação de servidores HTTP e o gerenciamento de rotas.

Criação de uma instância do aplicativo express

- A segunda linha `const app = express();` cria uma instância do aplicativo *Express*;
- Essa instância representa o seu servidor *web* e permite que você defina rotas, *middlewares* e outras configurações.

Middleware para logging

- A seção comentada `// Middleware para logging` indica que o próximo trecho de código é um middleware.
- *Middlewares* são funções que processam as requisições HTTP antes que elas alcancem as rotas finais;
- O middleware definido aqui será executado para todas as requisições recebidas pelo aplicativo;
- O middleware registra no console a informação sobre a requisição, incluindo o método HTTP (`req.method`) e a URL (`req.url`);
- Após o registro, o middleware chama `next()` para passar o controle para o próximo middleware ou rota.

Restante do código do servidor

- O comentário `// Restante do código do servidor...` indica que há mais código a seguir.
- Essa parte não está explicitamente mostrada no snippet, mas é onde você pode definir suas rotas, configurar endpoints e lidar com as requisições específicas do seu aplicativo.

Para um monitoramento mais avançado, ferramentas como o Morgan (<https://www.npmjs.com/package/morgan>) podem ser incorporadas para registrar detalhes adicionais das requisições.

8.1.6 Autenticação e autorização

Ao implementar a autenticação com chaves de API no lado do cliente, é possível enviar a chave como parte dos cabeçalhos da requisição. Veja, a seguir, um exemplo usando JavaScript e **fetch**:

```
const apiKey = 'SUA_CHAVE_API';

fetch('https://api.externa.com/dados', {
    method: 'GET',
    headers: {
        'Content-Type': 'application/json',
        'Authorization': `Bearer ${apiKey}` // Adicione a chave de
API nos cabeçalhos
    },
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Erro na requisição:', error));
```

Explicando os códigos

Definição da chave de API



- A primeira linha `const apiKey = 'SUA_CHAVE_API';` define uma variável chamada `apiKey` com o valor da sua chave de API;
- Essa chave será usada para autenticar a requisição à API externa.

Requisição HTTP com fetch

- A função **fetch** é utilizada para fazer uma requisição HTTP para a URL '<https://api.externa.com/dados>';
- O método da requisição é definido como '**GET**';
- Os cabeçalhos da requisição são configurados com duas informações importantes:
 - '**Content-Type**: **'application/json'**': indica que o conteúdo da requisição é do tipo JSON;
 - '**Authorization**: **Bearer \$lapiKey!**': adiciona a chave de API como um *token* de autorização no cabeçalho.
- O resultado da requisição é tratado com os métodos **.then()** e **.catch()**.
 - **.then(response => response.json())**: converte a resposta da requisição para um objeto JSON;
 - **.then(data => console.log(data))**: exibe os dados obtidos no console;
 - **.catch(error => console.error('Erro na requisição:', error))**: trata os erros caso a requisição não seja bem-sucedida.

Certifique-se de substituir 'SUA_CHAVE_API' pela sua chave de API real.
O token de autorização é adicionado ao cabeçalho como Bearer [chave]

8.2 Exemplo de autenticação em uma API com chaves de API

A autenticação é um aspecto crítico ao acessar APIs externas e uma maneira comum de autenticar requisições é por meio de chaves de API. Cada chave é única para um desenvolvedor ou aplicação, sendo utilizada para identificar e autorizar o acesso à API.

Obtendo uma chave de API



Geralmente, para obter uma chave de API, é necessário criar uma conta na plataforma que oferece a aplicação. Após o registro, o desenvolvedor recebe uma chave exclusiva que deve ser incluída nas requisições para autenticação.



Momento mão na massa: consumindo uma API com chave

Vamos observar um exemplo prático em que poderemos consumir uma API que requer autenticação por chave.

Neste caso, utilizaremos a [\[OpenWeatherMap API\]](https://openweathermap.org/api) (<https://openweathermap.org/api>), que fornece dados meteorológicos.

Neste exemplo, você pode criar uma conta na OpenWeatherMap, obter uma chave de API e substituir '**SUA_CHAVE_API**' pela chave fornecida. Essa aplicação permite ao usuário consultar o clima de uma cidade específica. Note que é crucial manter a chave de API **confidencial**, evitando compartilhá-la publicamente.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Consulta Meteorológica</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            text-align: center;
            padding: 20px;
        }
    </style>
</head>
<body>

    <h1>Consulta Meteorológica</h1>
    <label for="cidade">Digite o nome da cidade:</label>
    <input type="text" id="cidade" placeholder="Ex: São Paulo">
    <button onclick="consultarClima()">Consultar Clima</button>
```

```
<div id="resultado"></div>

<script>
// Função para consultar o clima usando a OpenWeatherMap API
async function consultarClima() {

    const cidadeInput = document.getElementById('cidade');

    const cidade = cidadeInput.value.trim();

    if (cidade !== '') {

        try {

            // Substitua 'SUA_CHAVE_API' pela chave de API fornecida pela
            OpenWeatherMap

            const apiKey = 'SUA_CHAVE_API';

            const apiUrl =
`https://api.openweathermap.org/data/2.5/weather?q=${cidade}&appid=${apiKey}`;

            const resposta = await fetch(apiUrl);

            const dadosClima = await resposta.json();

            // Exibir os dados do clima na página

            exibirResultado(dadosClima);

        } catch (erro) {

            console.error('Erro ao consultar o clima:', erro);

        }

    }

}

// Função para exibir os dados do clima na página

function exibirResultado(dadosClima) {

    const resultadoDiv = document.getElementById('resultado');
    // Verificar se a requisição foi bem-sucedida
```

```

if (dadosClima.cod === '404') {

    resultadoDiv.innerHTML = '<p>Cidade não encontrada. Verifique o nome informado.</p>';

} else {
    // Criar uma string HTML com os dados do clima
    const htmlString = `

        <p><strong>Cidade:</strong> ${dadosClima.name}, ${dadosClima.sys.country}</p>
        <p><strong>Temperatura:</strong> ${dadosClima.main.temp} °C</p>
        <p><strong>Condição:</strong> ${dadosClima.weather[0].description}</p>

    `;
    // Inserir a string HTML na div

    resultadoDiv.innerHTML = htmlString;
}

}

</script>

</body>
</html>

```

Explicando os códigos

Aqui, temos um formulário simples para inserir o nome da cidade e um botão para consultar o clima.

JavaScript para consulta de clima

- Neste código, o JavaScript é responsável por consultar a API da OpenWeatherMap e exibir os resultados na página web;
- A função **consultarClima()** é chamada quando o botão "Consultar Clima" é clicado. Ela extrai o nome da cidade digitado pelo usuário, constrói a URL da API com a chave de autenticação e faz uma requisição assíncrona usando fetch;
- Os dados retornados pela API são então manipulados na função **exibirResultado()**, que verifica se a cidade foi encontrada e exibe as informações meteorológicas relevantes na página.

Manipulação dos dados da API

Os dados da API são manipulados na linguagem JavaScript. O nome da cidade, a temperatura e a condição meteorológica são extraídos dos dados retornados pela API e inseridos no HTML da página para exibição ao usuário.

Tratamento de erros

O código inclui tratamento de erros para lidar com situações em que a cidade não é encontrada ou ocorre algum erro na consulta à API. Os erros são exibidos no console do navegador para fins de depuração.

Segurança da chave da API

A chave da API é sensível e deve ser protegida. No código fornecido, a chave é *hardcoded* diretamente no JavaScript. Normalmente, isso não é recomendado em aplicações reais. Em vez disso, a chave deveria ser armazenada de forma segura no servidor e não diretamente no código JavaScript enviado ao navegador.



Neste capítulo, aprendemos que a integração de APIs externas oferece inúmeras possibilidades para aprimorar aplicações *web*, mas é essencial abordar considerações de segurança, especialmente quando se trata de autenticação. Discutimos as principais considerações de segurança ao acessar APIs externas, enfatizando a importância de HTTPS, validação de entradas, *rate limiting*, CORS, monitoramento e *logging*, além de autenticação e autorização.

Também exploramos a autenticação com chaves de API por meio de um exemplo prático usando a OpenWeatherMap API. Vimos como podemos aplicar esses conceitos em projetos pessoais, garantindo que nossas aplicações sejam seguras, confiáveis e em conformidade com as práticas recomendadas de segurança na *web*. O conhecimento adquirido nesta área contribui significativamente para o desenvolvimento responsável e seguro de aplicações *web* modernas.



ATIVIDADE DE FIXAÇÃO

- 1.** O que é fundamental para garantir a confidencialidade dos dados ao fazer requisições para APIs externas?

 - a. Utilizar o protocolo HTTP.
 - b. Garantir que as requisições sejam realizadas apenas durante o horário comercial.
 - c. Utilizar conexões seguras através do protocolo HTTPS (SSL/TLS).
 - d. Criptografar apenas os dados sensíveis.

- 2.** Qual é a finalidade da validação de entradas ao enviar requisições para uma API?

 - a. Prevenir ataques de injeção, como SQL *injection* e XSS.
 - b. Aumentar o número de requisições permitidas pela API.
 - c. Melhorar a velocidade de resposta da API.
 - d. Garantir que todas as requisições sejam aceitas sem restrições.

- 3.** O que é *rate limiting* em uma API?

 - a. A velocidade máxima que um veículo pode atingir.
 - b. A quantidade máxima de dados que pode ser transmitida por segundo.
 - c. Uma técnica para limitar o número de requisições em um determinado período de tempo.
 - d. O tempo necessário para uma API responder a uma requisição.

- 4.** Qual é o propósito do CORS (*Cross-Origin Resource Sharing*) em uma API?

 - a. Limitar o acesso de usuários autenticados.
 - b. Permitir que apenas uma origem específica acesse os recursos da API.
 - c. Evitar a necessidade de autenticação em uma API.
 - d. Aumentar a taxa de resposta da API.

5. Por que a autenticação e autorização são consideradas práticas essenciais ao acessar APIs externas?

 - a. Para aumentar a complexidade do código.
 - b. Para tornar as requisições mais lentas.
 - c. Para garantir que apenas usuários autenticados e autorizados possam acessar recursos específicos.
 - d. Para diminuir a segurança da aplicação.
6. Explique como a implementação do HTTPS (SSL/TLS) contribui para a segurança ao acessar APIs externas.
7. Dê um exemplo prático de como realizar a validação de entradas em JavaScript antes de enviar uma requisição para uma API.
8. Descreva a importância do *rate limiting* em uma API e como isso contribui para a segurança e a eficiência do sistema.
9. Como você configuraria as políticas de CORS em um servidor *Express* para permitir o acesso de todas as origens a recursos da API?
10. Explique por que a autenticação com chaves de API é considerada uma prática segura ao acessar APIs externas. Dê um exemplo prático usando JavaScript e **fetch**.

CAPÍTULO 09

UTILIZANDO A FUNÇÃO FETCH() PARA REQUISIÇÕES A APIs EM JAVASCRIPT: MANIPULAÇÃO DE RESPOSTAS E TRATAMENTO DE ERROS

O que esperar deste capítulo:

- Manipular e processar diferentes tipos de respostas de API, como JSON, texto e *blobs*;
- Implementar estratégias eficazes de tratamento de erros ao lidar com requisições assíncronas usando a função **fetch()** em JavaScript.

Introdução

A integração de APIs em aplicações *web* tornou-se uma prática comum para enriquecer a experiência do usuário e acessar dados dinâmicos. Nesse contexto, a função **fetch()** entra em ação, já que é uma ferramenta poderosa, sendo frequentemente utilizada para fazer solicitações a APIs, pois realiza requisições HTTP de forma assíncrona, permitindo a interação eficiente com APIs. Neste capítulo, vamos explorar detalhadamente como usar a função **fetch()**, manipular as respostas obtidas e implementar estratégias de tratamento de erros. Além disso, vamos analisar exemplos práticos para que você possa praticar os seus conhecimentos em HTML, CSS e JavaScript.



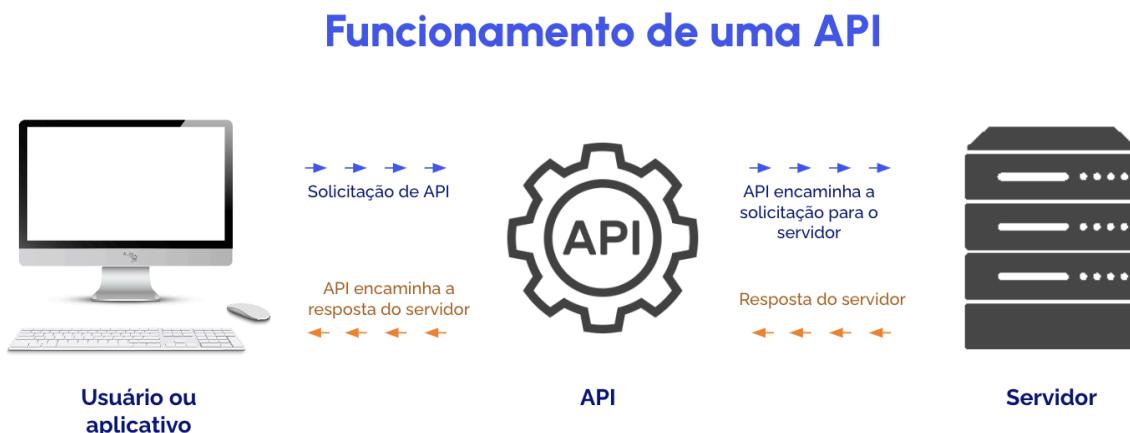
Disponível em:

<https://www.shutterstock.com/pt/image-vector/cloud-network-test-application-isometric-illustration-2046522920>.

Acesso em: 07 mar. 2024.

9.1 Utilizando a função `fetch()` para requisições a APIs

Antes de começar a estudar este assunto, vamos relembrar como uma API funciona? As APIs (*Application Programming Interface*) servem como intermediárias, funcionando como uma ponte na comunicação entre o usuário ou aplicativo e o servidor. Dessa forma, quando você faz uma solicitação de API, ela leva essa solicitação ao servidor e, depois, retorna a resposta do servidor para você. Veja como esse processo acontece no esquema a seguir:



A **função `fetch()`** é uma API moderna e nativa dos navegadores que **simplifica a realização de requisições HTTP assíncronas** em JavaScript. Ela retorna uma promessa (**Promise**) que, ao ser resolvida, entrega um objeto **Response**, que representa a resposta à requisição.

Essa função tem uma estrutura específica, que você deve conhecer para poder utilizá-las em seus futuros projetos. A sintaxe básica da função **`fetch()`** é a seguinte:

```
fetch(url)
  .then(response => {
    // Manipular a resposta
  })
  .catch(error => {
```

```
// Tratar erros  
});
```

Como é possível analisar neste código, a função **fetch()** aceita um argumento obrigatório, que é a **URL** da API para a qual desejamos fazer a requisição. Depois, a promessa retorna um objeto **Response**, que pode ser manipulado para obter os dados necessários. O **método .catch()** é utilizado para capturar e tratar quaisquer erros que ocorram durante a execução da promessa, seja na solicitação **fetch()** propriamente dita ou nas operações realizadas nos métodos **.then()** anteriores. Juntos, esses componentes formam uma cadeia de promessas que permite lidar com operações assíncronas de uma maneira mais estruturada e legível, facilitando o tratamento de respostas bem-sucedidas e a captura de erros.

9.2 Manipulação de respostas de API

Muitas vezes, as APIs respondem com dados no formato JSON. Isso acontece porque este é um formato leve de troca de dados, fácil de ler e simples de interpretar. Para manipular esses dados, é comum usar o **método json()** do objeto **Response**. Aqui está um exemplo prático de como isso é feito:

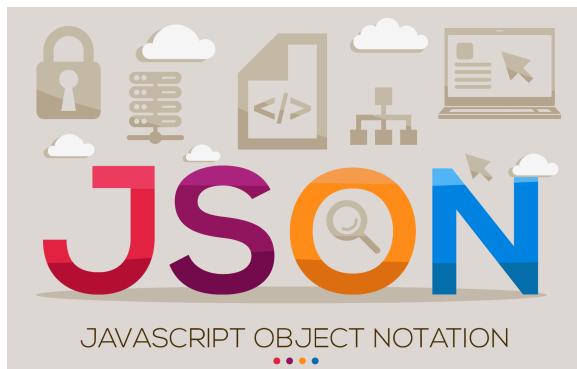
```
fetch('https://jsonplaceholder.typicode.com/todos/1')  
  
.then(response => response.json())  
  
.then(data => console.log(data))  
  
.catch(error => console.error('Erro na requisição:', error));
```

Vamos analisar este código passo a passo:

- **fetch('https://jsonplaceholder.typicode.com/todos/1')**: esta linha inicia uma solicitação para a URL especificada utilizando a função **fetch()**;

- **.then(response => response.json()):** após a promessa ser resolvida, o primeiro método **.then()** é chamado com o objeto **Response**, porém, essa resposta ainda não é o corpo do conteúdo. Para extrair o conteúdo em JSON, é necessário utilizar o método **.json()** do objeto **Response**, que é o que acontece na segunda linha do código;
- **.then(data => console.log(data)):** o segundo método **.then()** recebe o objeto JavaScript obtido na etapa anterior. Neste ponto, **data** contém os dados da resposta já convertidos de JSON para um objeto JavaScript. Estes dados são, então, passados para **console.log()**, permitindo visualizar o conteúdo no console do navegador ou do ambiente de execução;
- **.catch(error => console.error('Erro na requisição:', error)):** o método **.catch()** é usado para capturar e tratar qualquer erro que possa ocorrer durante a solicitação ou no processo de manipulação da resposta. Isso inclui erros de rede, se a promessa retornada por **fetch()** for rejeitada, ou erros que possam surgir ao tentar converter a resposta em JSON. No nosso exemplo, se um erro acontecer, a mensagem 'Erro na requisição:' seguida pelo detalhe do erro é exibida no console.

Esse código é um exemplo clássico de como fazer solicitações HTTP assíncronas e tratar respostas JSON em aplicações *web* modernas, fazendo com que o código fique mais limpo e fácil de entender.



Disponível em:

<https://www.shutterstock.com/pt/image-vector/json-mean-javascript-object-notation-computer-1871534536>. Acesso em:
07 mar. 2024.

Se a resposta da API não for JSON, pode ser necessário usar **outros métodos de manipulação**, como **text()** ou **blob()**. Para que você entenda melhor, vamos analisar um trecho de código que busca obter o conteúdo de uma resposta como texto:

```
fetch('https://www.example.com/text')

.then(response => response.text())

.then(data => console.log(data))

.catch(error => console.error('Erro na requisição:', error));
```

Este código demonstra uma solicitação de rede para buscar um recurso de texto. Em seguida, essa resposta é processada como texto, usando o método **text()** do objeto **Response** para ler o corpo da resposta e convertê-lo em uma *string* de texto. Como a leitura do corpo da resposta é uma operação assíncrona, o método **text()** retorna outra promessa, que é resolvida com o texto completo da resposta.

Utilizar solicitações de recurso de texto com a função **fetch()** e processá-las como texto é necessário em várias situações no desenvolvimento *web*, como, por exemplo, para recuperar códigos fonte ou dados em formato textual puro.

Já o **método blob()** é usado em objetos **Response** para extrair o corpo da resposta como um objeto *blob* (*Binary Large Object*), que representa dados imutáveis de um arquivo de dados brutos. Esse método é particularmente útil quando você está lidando com dados binários ou arquivos, como imagens, vídeos, PDFs etc., ou qualquer outro tipo que não seja texto simples. Esses dados podem ser tratados ou manipulados de várias maneiras, como salvá-los em disco, exibi-los em uma página da *web* (por exemplo, exibindo uma imagem), ou enviá-los para outro servidor.

Veja, a seguir, um exemplo de como é possível usar o método **blob()** em uma cadeia de promessas após uma solicitação **fetch()** para extrair e manipular dados binários:

```
```javascript
fetch('https://example.com/arquivo.pdf')
```

```

.then(response => response.blob())

.then(blobData => {
 // Aqui, 'blobData' contém o objeto Blob com os dados do arquivo PDF

 // Você pode então fazer o que precisar com esses dados binários, como
 salvá-los em disco ou exibi-los na página

 console.log('Dados binários do arquivo:', blobData);

})

.catch(error => {
 console.error('Erro na solicitação:', error);
});

...

```

De forma resumida, o método **blob()** é usado para extrair e manipular dados binários de uma resposta HTTP, permitindo que você trabalhe com arquivos e outros tipos de dados binários nas suas aplicações web.

### 9.3 Tratamento de erros



`.catch(error =>`

**Você já viu essa estrutura em algum lugar antes?**

Assim como você já deve ter notado, a estrutura `.catch(error =>` está presente em todos os códigos que analisamos neste capítulo e isso tem uma explicação. O tratamento de erros é uma parte crucial ao lidar com requisições assíncronas e a função

**catch()** nos permite capturar erros que podem ocorrer durante a requisição. Vamos conhecer agora dois erros comuns durante o processo de requisição de APIs e aprender como tratá-los.

### Erros de rede

São falhas que ocorrem quando um aplicativo tenta se comunicar com um servidor ou serviço externo através da Internet, mas encontra problemas que impedem a comunicação ou o recebimento de dados conforme esperado.

```
fetch('https://api.externa.com/dados')

.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.error('Erro de rede:', error));
```

Neste exemplo, caso exista um erro na rede, como uma falha na conexão, ele será capturado e tratado no bloco **catch()**.

### Erros na resposta da API

Situações em que uma solicitação feita a uma API não é concluída com sucesso, resultando em uma resposta que indica um estado de erro. Esses erros podem ocorrer por diversas razões e, geralmente, são expressos por códigos de status HTTP específicos e mensagens associadas. O mais conhecido deles é o **404 Not Found**, que indica que o recurso solicitado não foi encontrado.

```
fetch('https://api.externa.com/dados')
 .then(response => {
 if (!response.ok) {
 throw new Error(`Erro na resposta da API: ${response.status}`);
 }
 return response.json();
 })
 .then(data => console.log(data)) |
 .catch(error => console.error('Erro na requisição:', error));
```

Neste código, a verificação `if (!response.ok)` verifica se a resposta da API indica um erro (`status` diferente de 200-299). Se for o caso, um erro é lançado e capturado no bloco **catch()**.

O tratamento de erros com a função **catch()** em JavaScript é uma técnica fundamental para gerenciar as exceções e os erros que podem ocorrer durante a execução de operações assíncronas, especialmente quando trabalhamos com **Promises**. A capacidade de tratar erros de forma eficaz é crucial para desenvolver

aplicações confiáveis, que conseguem lidar com falhas e comportamentos inesperados sem quebrar ou interromper a experiência do usuário.

### Hora de praticar!



Agora que você já conhece a função **fetch()** e aprendeu a como lidar com os erros usando a função **catch()** durante a requisição a APIs, vamos colocar a mão na massa e praticar tudo o que aprendemos ao longo deste capítulo. A seguir, estão dois exercícios para você treinar a utilização da função **fetch()** em situações do mundo real. Essa é uma ótima chance de aprimorar as suas habilidades, deixando você ainda mais preparado para atuar na área do desenvolvimento web!

Para realizar os exercícios, acesse um ambiente de desenvolvimento de sua preferência. Aqui estão duas sugestões:

**CODEPEN**



<https://codepen.io/>

**Visual Studio**



<https://code.visualstudio.com/>



## Exercício 1: consumindo uma API de piadas

Crie uma aplicação simples que consome uma API de piadas e exibe uma piada na tela. Para isto, siga o exemplo do código a seguir, que utiliza a **JokeAPI** (<https://v2.jokeapi.dev/>).

```
<!DOCTYPE html>

<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width,
initial-scale=1.0">
 <title>API de Piadas</title>
 <style>
 body {
 font-family: Arial, sans-serif;
 text-align: center;
 padding: 20px;
 }
 </style>
</head>
<body>

 <h1>Piada do Dia</h1>
 <button onclick="obterPiada()">Obter Piada</button>
 <div id="piada"></div>
 <script>

 function obterPiada() {
```

```
fetch('https://v2.jokeapi.dev/joke/Any')
 .then(response => {
 if (!response.ok) {
 throw new Error(`Erro na resposta da API:
${response.status}`);
 }

 return response.json();
 })

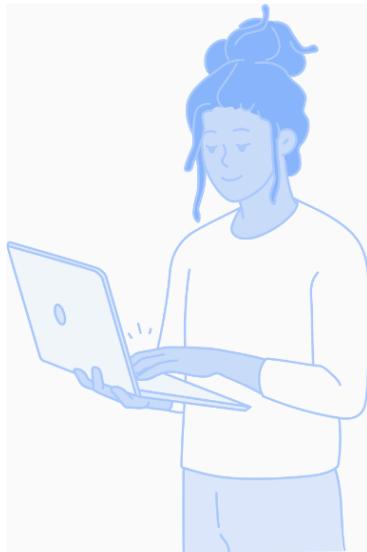
 .then(data => {
 const piadaDiv = document.getElementById('piada');
 piadaDiv.textContent = data.joke || `${data.setup}
${data.delivery}`;

 })
 .catch(error => console.error('Erro na requisição:',
error));
}

</script>
</body>
</html>
```

## E aí, consegui completar a tarefa?

Se necessário, fale com seu(a) professor(a) para esclarecer as suas dúvidas. Estamos apenas no início da nossa jornada de aprendizado.



## Exercício 2: postando dados para uma API

Agora, crie uma aplicação que permite aos usuários postar comentários e visualizar os já existentes.  
Siga o exemplo do código a seguir, que utiliza a **JSONPlaceholder** (<https://jsonplaceholder.typicode.com/>), uma API de teste.

```
<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,
initial-scale=1.0">

<title>Comentários</title>

<style>

body {

 font-family: Arial, sans-serif;

 text-align: center;

 padding: 20px;

}

</style>
```

```
</head>

<body>

 <h1>Comentários</h1>

 <textarea id="novoComentario" placeholder="Digite seu comentário"></textarea>

 <button onclick="adicionarComentario()">Adicionar Comentário</button>

 <div id="comentarios"></div>

<script>

function adicionarComentario() {

 const novoComentarioInput = document.getElementById('novoComentario');

 const novoComentario = novoComentarioInput.value.trim();

 if (novoComentario !== '') {

 // Postar o novo comentário

 fetch('https://jsonplaceholder.typicode.com/comments', {

 method: 'POST',

 headers: {

 'Content-Type': 'application/json',

```

```
 } ,

 body: JSON.stringify({

 postId: 1, // O ID do post ao qual o comentário está
associado

 name: 'Usuário', // Nome do usuário (pode ser
personalizado)

 email: 'usuario@example.com', // E-mail do usuário
(pode ser personalizado)

 body: novoComentario,

 }) ,

 })

.then(response => {

 if (!response.ok) {

 throw new Error(`Erro na resposta da API:
${response.status}`);

 }

 return response.json();

})

.then(data => {

 // Limpar o campo de input

 novoComentarioInput.value = '';

 // Atualizar os comentários exibidos

 obterComentarios();
})
```

```
 }

 .catch(error => console.error('Erro na requisição:', error));

 }

}

function obterComentarios() {

 // Obter os comentários existentes

fetch('https://jsonplaceholder.typicode.com/comments?postId=1')

 .then(response => {

 if (!response.ok) {

 throw new Error(`Erro na resposta da API: ${response.status}`);
 }

 return response.json();
 })

 .then(data => {

 const comentariosDiv = document.getElementById('comentarios');

 comentariosDiv.innerHTML = '';

 // Exibir os comentários na página
 });
}
```

```
 data.forEach(comentario => {

 const comentarioP = document.createElement('p');

 comentarioP.textContent = comentario.body;

 comentariosDiv.appendChild(comentarioP);

 });

 }

 .catch(error => console.error('Erro na requisição:', error));

}

// Chamar a função para obter comentários quando a página carregar

obterComentarios();

</script>

</body>

</html>
```

### E aí, conseguiu completar a tarefa?

Se precisar de dicas, *feedbacks* ou quiser compartilhar as suas experiências, fale com seu(a) professor(a), troque ideias com os seus colegas e continue aprofundando seus estudos sobre o tema.



Neste capítulo, vimos que função **fetch()** é uma ferramenta poderosa para realizar requisições a APIs em JavaScript. Exploramos a sintaxe básica da função **fetch()**, a manipulação de respostas de API utilizando métodos como **json()** e as estratégias de tratamento de erros para garantir robustez nas aplicações. Para finalizar, trouxemos dois exercícios para que você colocasse em prática seus conhecimentos em HTML, CSS e JavaScript, promovendo a consolidação das habilidades necessárias para interagir com APIs de forma eficiente e segura. Ao praticar esses conceitos, você estará mais preparado para enfrentar desafios do mundo real no desenvolvimento web.

### Capítulo 9

Utilizando a função **fetch()** para requisições a APIs em JavaScript

**fetch():**  
retorna uma promessa (**Promise**) que, ao ser resolvida, entrega um objeto **Response**, que representa a resposta à requisição.

**Métodos de manipulação de respostas de API:**

- **json()**
- **text()**
- **blob()**

**catch():**  
permite capturar erros que podem ocorrer durante a requisição.



### ATIVIDADE DE FIXAÇÃO

1. Qual é o propósito principal da função **fetch()** em JavaScript?
  - a. Manipular elementos HTML.
  - b. Fazer requisições assíncronas para APIs.
  - c. Criar estilos de página com CSS.
  - d. Realizar operações matemáticas complexas.
2. O que a função **json()** faz ao ser aplicada a um objeto **Response** na manipulação de respostas de API?

- a. Converte a resposta para um formato de áudio.
  - b. Converte a resposta para um objeto JavaScript.
  - c. Converte a resposta para um formato de imagem.
  - d. Converte a resposta para um formato de texto simples.
- 3.** Qual é a finalidade do bloco **catch()** ao usar a função **fetch()** para tratar erros em requisições assíncronas?
- a. Manipular a resposta da API.
  - b. Exibir os dados obtidos na página.
  - c. Tratar erros que podem ocorrer durante a requisição.
  - d. Converter a resposta para JSON.
- 4.** Como podemos verificar se a requisição foi bem-sucedida usando a função **fetch()** no tratamento de erros na resposta de uma API?
- a. Verificando se o *status* da resposta é um código 404.
  - b. Utilizando o método **text()** na resposta.
  - c. Checando se o método **ok** do objeto **Response** é verdadeiro.
  - d. Capturando qualquer exceção com o bloco **try**.
- 5.** Qual é a função do método **text()** ao manipular respostas de API com a função **fetch()**?
- a. Converte a resposta para JSON.
  - b. Converte a resposta para um formato de áudio.
  - c. Converte a resposta para um objeto JavaScript.
  - d. Converte a resposta para um formato de texto simples.
- 6.** Explique como você utilizaria a função **fetch()** para obter dados de uma API que fornece informações sobre o clima. Forneça um exemplo prático em HTML, CSS e JavaScript.
- 7.** Ao manipular respostas de API que podem conter erros de rede, como você implementaria um tratamento adequado usando a função **catch()**? Dê um exemplo prático.

8. Descreva a importância da conversão de respostas para JSON ao trabalhar com APIs em JavaScript. Forneça um exemplo prático usando a função **fetch()**.
9. Em um cenário em que é necessário enviar dados para uma API, explique como você faria isso utilizando a função **fetch()** com o método **POST**. Apresente um exemplo prático envolvendo a criação de comentários.
10. Suponha que você esteja desenvolvendo uma aplicação que consome uma API de imagens. Como você manipularia as respostas da API para exibir as imagens em sua página? Dê um exemplo prático envolvendo HTML, CSS e JavaScript.

# CAPÍTULO 10

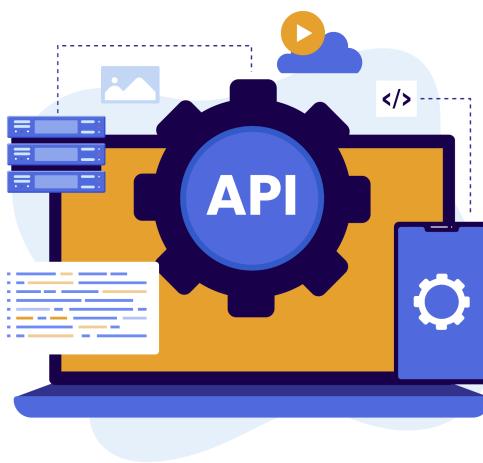
## EXIBIÇÃO DE DADOS DA API EM UMA PÁGINA WEB: CRIANDO UMA APLICAÇÃO QUE CONSUME DADOS

### O que esperar deste capítulo:

- Implementar a renderização dinâmica de dados da API em uma página web utilizando HTML, CSS e JavaScript;
- Desenvolver habilidades de manipulação do DOM para exibir informações de forma organizada e esteticamente agradável na interface do usuário.

### Introdução

A exibição de dados de uma API (*Application Programming Interface*) em uma página web é uma habilidade essencial para desenvolvedores web, pois proporciona a criação de aplicações dinâmicas e interativas. A integração das APIs permite que os desenvolvedores acessem uma variedade de dados, desde informações climáticas e notícias até dados de usuários e imagens. Neste capítulo, vamos explorar como criar uma aplicação simples que consome dados de uma API e os exibe de maneira elegante em uma página web. Para isso, utilizaremos os nossos conhecimentos sobre HTML, CSS e JavaScript, que são as linguagens base da programação.



Disponível em:

<https://www.shutterstock.com/pt/image-vector/application-programming-interface-illustration-concept-websites-2174940029>. Acesso em: 26 fev. 2024.

Além da capacidade de tornar as aplicações mais dinâmicas e interativas, a exibição de dados de APIs em páginas *web* também abre portas para a criação de experiências personalizadas e altamente relevantes para os usuários. Essa prática permite que desenvolvedores ofereçam conteúdo em tempo real, atualizando automaticamente as informações conforme elas são modificadas na fonte de dados. Essa abordagem não apenas enriquece a experiência do usuário, mas também contribui para a construção de aplicações mais sofisticadas e alinhadas com as expectativas dos consumidores digitais.

No contexto do desenvolvimento *web* moderno, no qual a colaboração entre diferentes serviços e plataformas é comum, a habilidade de consumir e exibir dados de APIs é um diferencial valioso. Isso possibilita a criação de aplicações que transcendem os limites de uma única fonte de dados, proporcionando aos usuários acesso a uma gama diversificada de informações. Dito isto, estar capacitado(a) com essas habilidades, não apenas irá fazer com que você possa contribuir para a promoção da inovação na *web*, mas também fará com que você se torne parte de uma nova geração de profissionais preparada para enfrentar os desafios complexos e empolgantes do ambiente digital em constante evolução.

## 10.1 Fundamentos da exibição de dados da API

Antes de mergulharmos nos exemplos práticos, é muito importante que você compreenda os fundamentos da exibição de dados de uma API. Esse processo envolve a realização de uma **requisição à API**, a **manipulação dos dados recebidos** e, por fim, a **renderização desses dados na página web**. Vamos conhecer cada um desses passos de maneira mais detalhada no quadro a seguir.

1

**Requisição à API:** o cliente envia uma solicitação à API, especificando o que está sendo solicitado e, muitas vezes, incluindo parâmetros relevantes, como filtros, consultas ou informações de autenticação. Em JavaScript, é utilizada a função **fetch()** para fazer uma requisição à API. Essa função retorna uma Promessa (**Promise**) que, quando resolvida, fornece um objeto **Response** contendo os dados da API.

2

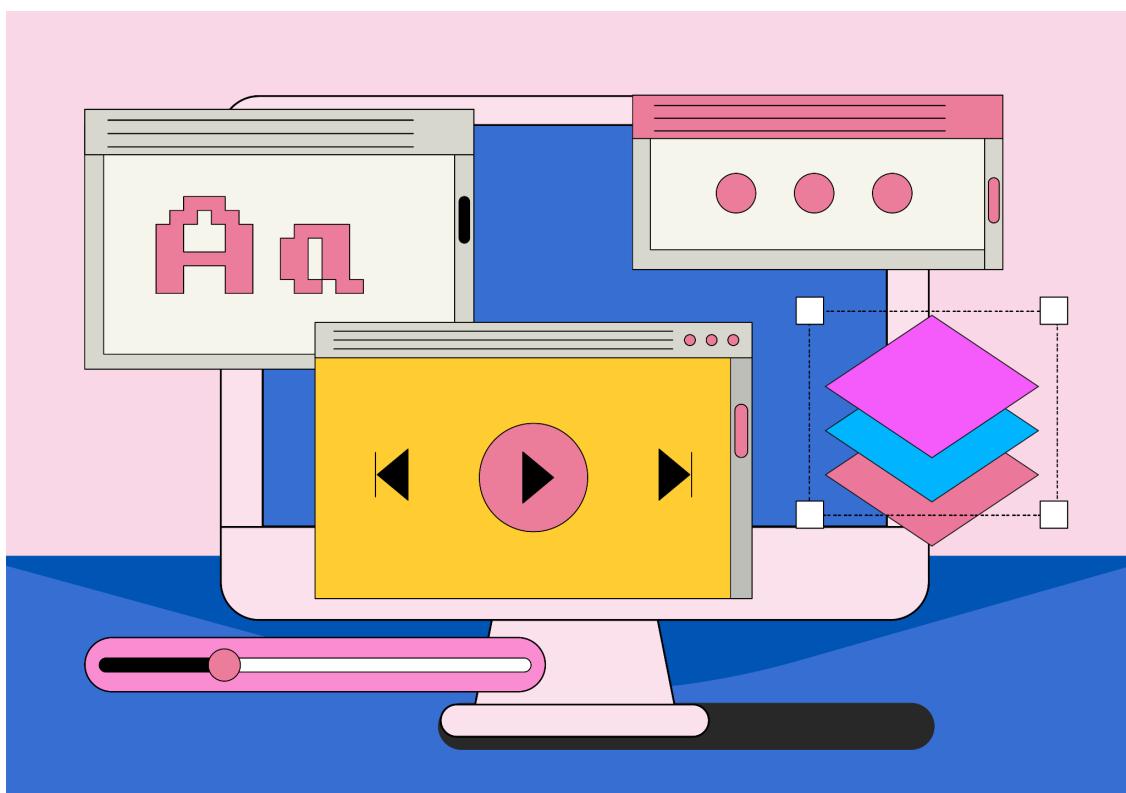
**Manipulação dos dados:** após obter a resposta da API, é comum manipular os dados para extrair as informações relevantes. Isso pode envolver a conversão da resposta para JSON, a verificação de erros e a extração de dados específicos. O cliente, em seguida, interpreta esses dados e realiza qualquer manipulação adicional necessária.

3

**Renderização na página web:** por fim, os dados manipulados são exibidos na página *web*. Isso geralmente é feito **manipulando o DOM** (Modelo de Objeto do Documento) com JavaScript e aplicando estilos com CSS para criar uma apresentação visual atraente.

Essas etapas são fundamentais na interação entre clientes (navegadores web, aplicativos móveis etc.) e servidores que disponibilizam dados através de uma API. O desenvolvedor da aplicação do lado do cliente precisa entender a estrutura da API, como fazer solicitações corretas e como interpretar os dados recebidos para proporcionar uma experiência de usuário eficaz.

Para que você entenda melhor como isso funciona, vamos analisar um exemplo prático de uso da API do YouTube. Ao utilizar a API dessa plataforma, os desenvolvedores poderiam criar um aplicativo que permite aos usuários pesquisar vídeos diretamente do YouTube, visualizar informações detalhadas sobre vídeos específicos, incorporar vídeos em páginas da web ou, até mesmo, criar *playlists* personalizadas.



Disponível em:

<https://www.shutterstock.com/pt/image-vector/computer-screen-programs-applications-conceptual-flat-1950569122>.

Acesso em: 28 fev. 2024.

Agora, vamos propor um desafio para que você coloque a mão na massa e pratique as suas habilidades para consolidar os seus conhecimentos sobre este assunto.



## Hora do desafio!

### Consumindo uma API de filmes

Vamos criar uma aplicação simples que consome a API pública **The Movie Database (TMDb)** (<https://www.themoviedb.org/documentation/api>) para exibir informações sobre filmes. Essa API oferece uma ampla variedade de dados relacionados a filmes, incluindo detalhes sobre títulos, elenco, avaliações e muito mais.

Para cumprir o desafio, abra um ambiente de desenvolvimento de sua preferência, como o **CodePen** (<https://codepen.io/>) ou o **Visual Studio Code** (<https://code.visualstudio.com/>), e reproduza o passo a passo que veremos a seguir.

### Passo 1 - Estrutura HTML Básica

```
<!DOCTYPE html>

<html lang="en">

<head>

 <meta charset="UTF-8">

 <meta name="viewport" content="width=device-width, initial-scale=1.0">

 <title>API de Filmes</title>

 <link rel="stylesheet" href="styles.css">

</head>

<body>
```

```
<div id="app">

 <!-- Conteúdo será adicionado dinamicamente aqui -->

</div>

<script src="script.js"></script>

</body>

</html></pre>
```

Como você pode observar, essa estrutura HTML básica contém uma `<div>` com o ID "app", que será usada para exibir dinamicamente os dados da API. Os arquivos de estilo ('styles.css') e script ('script.js') são vinculados à página.

## Passo 2 - Estilo CSS (styles.css)

```
body {

 font-family: Arial, sans-serif;

 background-color: #f0f0f0;

 margin: 0;

 padding: 0;

 display: flex;

 align-items: center;

 justify-content: center;

 height: 100vh;
```

```
}

#app {
 background-color: #fff;
 border-radius: 8px;
 box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
 padding: 20px;
 width: 400px;
}
```

Aqui, o estilo CSS fornece uma aparência mínima à aplicação, garantindo um *layout* agradável e legível.

### Passo 3 - Script JavaScript (script.js)

```
document.addEventListener('DOMContentLoaded', () => {

 const apiKey = 'SUA_CHAVE_API'; // Substitua com sua chave da TMDb API

 const apiUrl = 'https://api.themoviedb.org/3/movie/popular';

 // Função para fazer a requisição à API

 const fetchData = async () => {

 try {

 const response = await fetch(` ${apiUrl}?api_key=${apiKey}`);
 if (!response.ok) {
```

```
 throw new Error(`Erro na resposta da API: ${response.status}`);

 }

 const data = await response.json();

 displayMovies(data.results);

} catch (error) {

 console.error('Erro na requisição:', error);

}

};

// Função para exibir os filmes na página

const displayMovies = (movies) => {

 const appContainer = document.getElementById('app');

 movies.forEach((movie) => {

 const movieCard = document.createElement('div');

 movieCard.classList.add('movie-card');

 const title = document.createElement('h2');

 title.textContent = movie.title;

 const releaseDate = document.createElement('p');

 releaseDate.textContent = `Lançamento: ${movie.release_date}`;

 const overview = document.createElement('p');

 overview.textContent = movie.overview;

 movieCard.appendChild(title);
```

```
 movieCard.appendChild(releaseDate);

 movieCard.appendChild(overview);

 appContainer.appendChild(movieCard);

}) ;

};

// Chamar a função para buscar e exibir os filmes ao carregar a página

fetchData();

}) ;
```

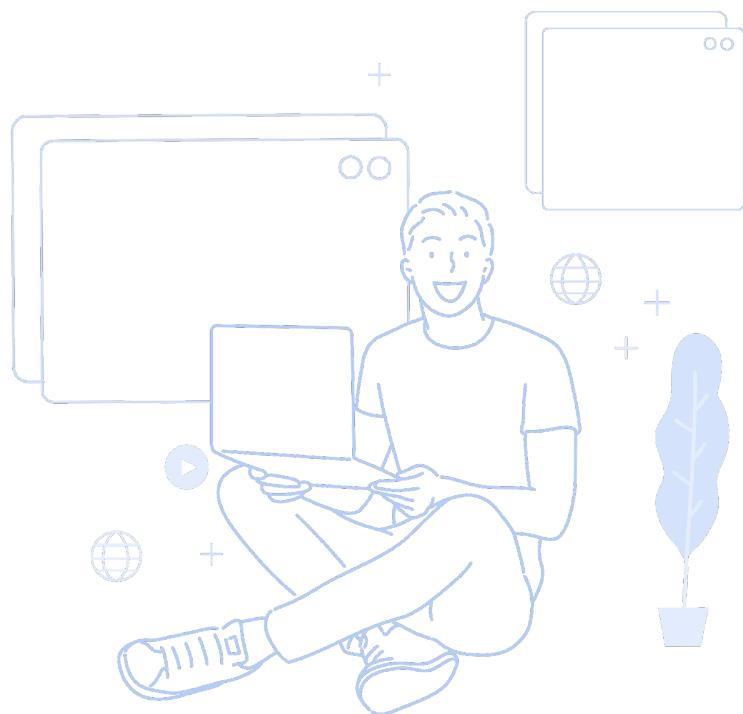
O *script* JavaScript utiliza a função **fetchData()** para fazer uma requisição à API da TMDb e a função **displayMovies()** para renderizar os filmes na página. A chamada à função **fetchData()** é feita quando o conteúdo da página é carregado (**DOMContentLoaded**). E aí, como é que a sua aplicação está ficando até aqui?

#### Passo 4 - Chave da API

Antes de executar o exemplo, substitua '**SUA\_CHAVE\_API**' pela sua chave de API da TMDb. Você pode obtê-la neste *link*: (<https://www.themoviedb.org/documentation/api>).

#### Passo 5 - Resultado final

Após substituir a chave da API, abra o arquivo HTML em um navegador. Você verá uma lista de filmes populares exibidos na página, consumidos da API da TMDb. Este exemplo prático demonstra como criar uma aplicação que consome dados de uma API e os exibe de forma interativa na página web.



*Level up!*

Parabéns pelo seu esforço até aqui! Se você quiser alcançar um nível ainda mais além, acesse o vídeo a seguir para saber mais sobre **fetch** API.

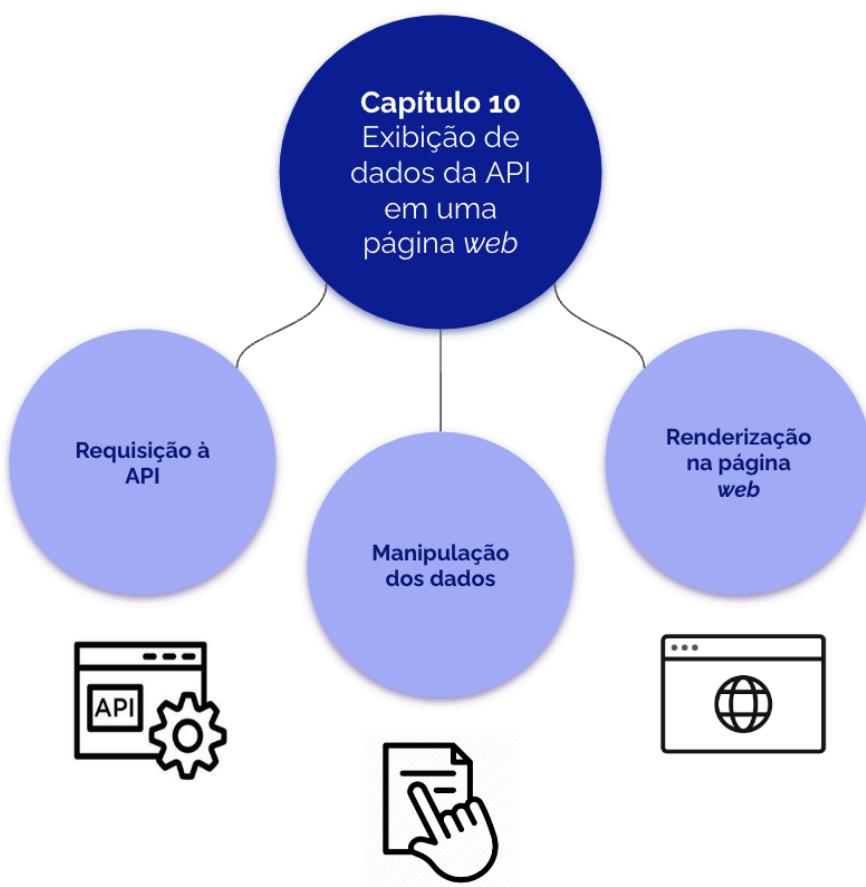


Fonte: Vídeo Aprenda Fetch API de Javascript com Projeto, do canal do YouTube. Matheus Battisti - Hora de Codar.

Disponível em: <https://www.youtube.com/watch?v=qIGYM4S8x5o>. Acesso em: 10 mar. 2024.



Neste capítulo, vimos que a exibição de dados de uma API em uma página web é uma habilidade essencial para desenvolvedores *web* e aprendemos os fundamentos desse processo, desde a realização de requisições à API até a manipulação e exibição dos dados. A proposta de um desafio prático de uma aplicação de filmes pôde proporcionar uma experiência *hands-on*, permitindo que você colocasse em prática a integração de uma API utilizando HTML, CSS e JavaScript. Após seguir o passo a passo do desafio e explorar outros conjuntos de dados e APIs, você está ainda mais bem preparado para criar aplicações dinâmicas e interativas que enriquecerão a experiência do usuário. Além disso, a prática constante desses conceitos fortalecerá a sua compreensão sobre como lidar com dados externos em projetos *web* do mundo real.





## ATIVIDADE DE FIXAÇÃO

1. Qual é o principal benefício de exibir dados de uma API em uma página *web*?

  - a. Aumentar a complexidade do código.
  - b. Tornar a aplicação menos interativa.
  - c. Criar experiências dinâmicas e interativas.
  - d. Reduzir a quantidade de dados disponíveis.
  
2. Qual das alternativas apresenta a função em JavaScript utilizada para fazer requisições a APIs?

  - a. request()
  - b. ajax()
  - c. fetch()
  - d. loadAPI()
  
3. O que é necessário para realizar a exibição de dados de uma API em uma página *web*?

  - a. Apenas HTML.
  - b. Somente CSS.
  - c. HTML, CSS e JavaScript.
  - d. JavaScript e Python.
  
4. Em qual parte do processo de exibição de dados de uma API a função **fetch()** é comumente utilizada?

  - a. Na manipulação de dados.
  - b. Na renderização na página *web*.
  - c. Na requisição à API.
  - d. Na estilização com CSS.
  
5. Qual é a finalidade da chave da API ao criar uma aplicação que consome dados de uma API?

  - a. Aumentar a velocidade da requisição.
  - b. Personalizar o estilo da página.
  - c. Autenticar e autorizar o acesso à API.
  - d. Adicionar efeitos visuais à página.

6. Descreva os passos fundamentais para criar uma aplicação que consome dados de uma API e exibe esses dados em uma página *web*. Inclua exemplos práticos utilizando JavaScript.
7. Ao consumir dados de uma API, por que a manipulação de respostas, como a conversão para JSON, é uma etapa importante? Dê um exemplo prático de como isso pode ser realizado
8. Explique a importância da integração de APIs na criação de experiências *web* mais ricas. Forneça um exemplo prático de uma aplicação que utiliza dados de uma API para melhorar a experiência do usuário.
9. Ao criar uma aplicação que consome dados de uma API, como você lidaria com erros durante a requisição? Forneça estratégias de tratamento de erros e dê um exemplo prático.
10. Suponha que você precisa criar uma aplicação que exibe uma lista de livros provenientes de uma API de biblioteca virtual. Descreva os passos que você seguiria, desde a requisição à API até a exibição dos dados na página *web*. Se possível, forneça o código prático.

# CAPÍTULO 11

## EXPLORANDO O MUNDO DAS APIs: ESTUDOS DE CASO, INTEGRAÇÃO PRÁTICA E DESENVOLVIMENTO DE PROJETOS

### O que esperar deste capítulo:

- Identificar e analisar os diferentes tipos de APIs disponíveis para atender a requisitos específicos de um projeto;
- Integrar múltiplas APIs em um projeto prático, aplicando os conceitos aprendidos para desenvolver soluções funcionais e inovadoras.

### Introdução

As APIs (*Application Programming Interfaces* ou, em português, Interfaces de Programação de Aplicações) têm se tornado peças fundamentais no cenário da tecnologia, proporcionando uma forma eficiente e padronizada para que **diferentes sistemas se comuniquem e compartilhem informações**. O papel das APIs no desenvolvimento de *software* é tão crucial que muitos aplicativos modernos dependem não apenas de uma, mas de várias APIs para fornecer funcionalidades robustas e dados em tempo real. Neste capítulo, nós vamos conhecer melhor este assunto, explorando exemplos de aplicativos que dependem de múltiplas APIs, analisando uma demonstração prática de integração de APIs em um projeto e, por fim, desenvolvendo um projeto prático que utiliza APIs externas.



Disponível em:

<https://www.shutterstock.com/pt/image-photo/businessman-touched-api-acronym-word-icon-520064494>. Acesso em:  
28 fev. 2024.

## 11.1 Aplicativos dependentes de múltiplas APIs

Para aprofundar ainda mais nossos conhecimentos de API, vamos analisar dois exemplos que utilizam os serviços de API para funcionarem. Dessa forma, você poderá entender melhor como esse conceito se aplica na prática.

### Aplicativo de previsão do tempo e tráfego

Esses aplicativos podem depender de **APIs de diferentes provedores para obter dados precisos** de previsão do tempo, informações de tráfego em tempo real e outras métricas relacionadas. A integração de várias APIs permite que o aplicativo ofereça aos usuários uma experiência abrangente, reunindo dados de diversas fontes para proporcionar informações precisas e personalizadas tendo como base a localização do usuário.



### Plataforma de mídia social

Frequentemente, grandes plataformas de mídia social integram várias APIs para oferecer uma experiência completa aos usuários. Isso pode incluir integrações com APIs de autenticação, de compartilhamento de conteúdo, de análise de dados para avaliar o engajamento do usuário e de terceiros para incorporar conteúdo externo diretamente nas postagens.

A combinação de várias APIs permite que a plataforma permaneça dinâmica e inovadora, incorporando novos recursos e mantendo-se conectada a diversas fontes de informação.



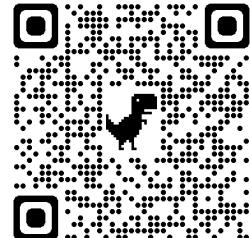
Com certeza você já deve ter usado algum app para saber mais sobre o clima na sua cidade ou tem um perfil ativo em alguma rede social, não é mesmo? Você já tinha parado para pensar como eles conseguem reunir tantos recursos e informações em um só lugar? Agora você sabe: é com a ajuda de APIs, que fornecem uma estrutura de

comunicação que permite que desenvolvedores de *software* integrem as suas aplicações com outros sistemas de maneira eficiente. Isso permite a interoperabilidade entre diferentes serviços e aplicativos, facilitando a troca de dados e funcionalidades.



Saiba mais

Acesse o vídeo a seguir para saber como integrar uma API no desenvolvimento de *software* e aprofunde ainda mais os seus conhecimentos sobre este assunto!



Fonte: <https://www.youtube.com/watch?v=Bi5HsQz-87A>. Acesso em: 28 fev. 2024.

## 11.2 Integração de APIs em um projeto

Agora que você já está por dentro do mundo das APIs, vamos acompanhar um passo a passo de **como integrar duas APIs distintas em um projeto**. Será uma ótima oportunidade para você, que está começando a estudar programação e desenvolvimento de *software*, pois este é o momento de colocar a mão na massa e seguir todos os passos, praticando tudo que aprendemos até aqui!

A seguir, você irá acompanhar as etapas da criação de uma aplicação que exibe informações sobre filmes e músicas, combinando dados provenientes das APIs do The Movie Database (TMDb), uma base de dados sobre filmes e séries, e do Last.fm, um site de serviço musical.



## Level up!

Abra um ambiente de desenvolvimento de sua preferência, como o **CodePen** (<https://codepen.io/>) ou o **Visual Studio Code** (<https://code.visualstudio.com/>) e siga o passo a passo que veremos a seguir.

### Passo 1 - Estrutura básica

```
<!DOCTYPE html>
<html lang="en">
 <head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width,
initial-scale=1.0">
 <title>Projeto de Integração de APIs</title>
 <link rel="stylesheet" href="styles.css">
 </head>
 <body>
 <div id="app">
 <!-- Conteúdo será adicionado dinamicamente aqui -->
 </div>
 <script src="script.js"></script>
 </body>
</html>
```

Essa estrutura HTML básica contém uma **<div>** com a ID **"app"**, que será usada para exibir dinamicamente os dados da API. Os arquivos de estilo (**styles.css**) e script (**script.js**) são vinculados à página.



**E aí, conseguiu acompanhar a leitura desse código?**

A sua jornada no universo da programação está apenas começando e, nesse caminho, você deve analisar os códigos como um detetive digital para descobrir o que cada linha está fazendo.

## Passo 2 - Estilo CSS (styles.css)

```
body {
 font-family: Arial, sans-serif;
 background-color: #f0f0f0;
 margin: 0;
 padding: 0;
 display: flex;
 align-items: center;
 justify-content: center;
 height: 100vh;
}

#app {
 background-color: #fff;
 border-radius: 8px;
 box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
 padding: 20px;
 width: 400px;
}

.movie-card,
.music-card {
 margin-bottom: 20px;
 padding: 10px;
 border: 1px solid #ccc;
 border-radius: 8px;
}
```

Neste código, o estilo CSS fornece uma aparência mínima à aplicação, garantindo um *layout* agradável e legível para o *app*. Nesse momento, também foram adicionadas duas classes para estilizar os cartões de filmes (**movie-card**) e músicas (**music-card**).

### Passo 3 - Script JavaScript (script.js)

```
document.addEventListener('DOMContentLoaded', () => {

 const tmdbApiKey = 'SUA_CHAVE_API_TMDB'; // Substitua com sua chave da
 TMDB API

 const lastFmApiKey = 'SUA_CHAVE_API_LASTFM'; // Substitua com sua chave
 da Last.fm API

 const tmdbApiUrl = 'https://api.themoviedb.org/3/movie/popular';
 const lastFmApiUrl =
https://ws.audioscrobbler.com/2.0/?method=chart.gettoptracks&api_key=;

 const fetchData = async (url, apiKey) => {

 try {

 const response = await fetch(` ${url}?api_key=${apiKey}`);

 if (!response.ok) {

 throw new Error(`Erro na resposta da API: ${response.status}`);
 }

 return await response.json();
 } catch (error)

 {
 console.error('Erro na requisição:', error);
 }
 };

 const displayData = (movies, music) => {
 const appContainer = document.getElementById('app');
```

```

movies.forEach((movie) => {
 const movieCard = document.createElement('div');
 movieCard.classList.add('movie-card');
 movieCard.textContent = `Filme: ${movie.title}`;
 appContainer.appendChild(movieCard);
});

music.forEach((track) => {

 const musicCard = document.createElement('div');
 musicCard.classList.add('music-card');
 musicCard.textContent = `Música: ${track.name} - Artista: ${track.artist.name}`;
 appContainer.appendChild(musicCard);
});

const init = async () => {
 const movies = await fetchData(tmdbApiUrl, tmdbApiKey);
 const music = await fetchData(`${lastFmApiUrl}${lastFmApiKey}`, '');

 displayData(movies.results, music.tracks.track);
};

init();
}) ;

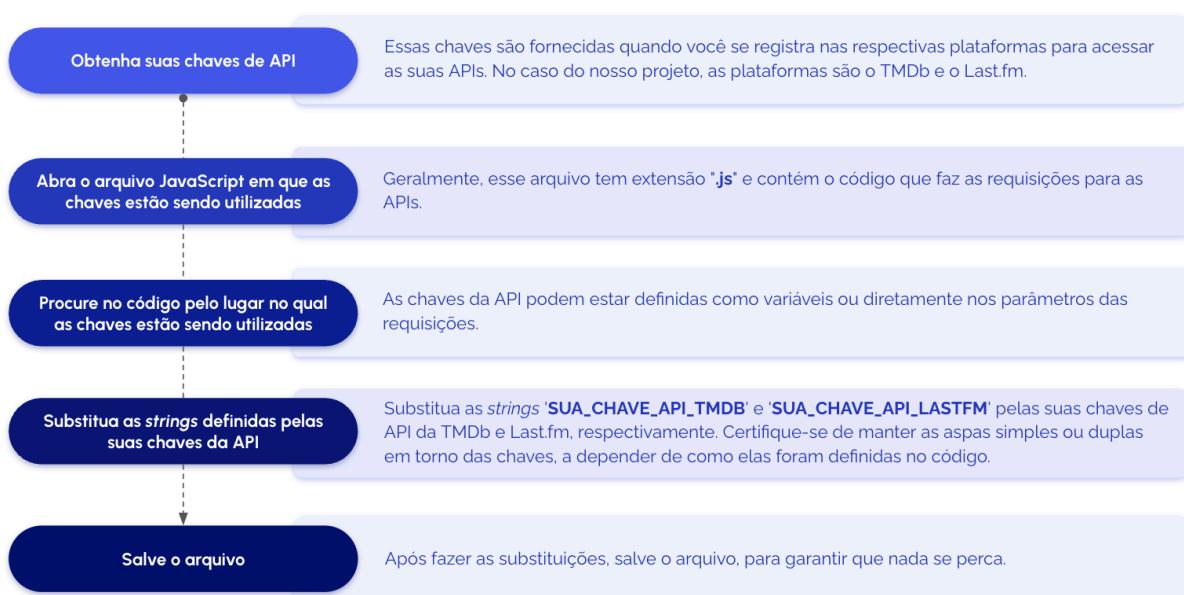
```

Nesta etapa, o *script* JavaScript tem duas funções importantes: **fetchData()** e **displayData()**. A primeira função, **fetchData()**, é como se fosse a mensageira que vai até as APIs do TMDb e do Last.fm buscar as informações. Já a segunda função, **displayData()**, é a responsável por mostrar essas informações na página.

Quando o conteúdo da página é totalmente carregado, a função **fetchData()** é chamada. Isso garante que essa “função mensageira” vá buscar os dados somente quando todos os elementos da página estiverem prontos para recebê-los. Depois que a mensageira retorna com os dados das APIs, a função **displayData()** entra em ação, organizando e exibindo essas informações na página para que o usuário possa vê-las. Interessante, não é mesmo? Agora, vamos para o próximo passo: a substituição das chaves API.

## Passo 4 - Substituição das chaves da API

Antes de executar o código que você estruturou ao seguir os passos anteriores, substitua '**SUA\_CHAVE\_API\_TMDB**' e '**SUA\_CHAVE\_API\_LASTFM**' pelas suas chaves de API da TMDb e Last.fm, respectivamente. Veja um esquema de como isso deve ser feito:



Ao seguir todas essas etapas, as suas chaves de API estarão atualizadas no código, permitindo que as requisições sejam feitas corretamente.

## Passo 5 - Resultado final

Após substituir as chaves da API, abra o arquivo HTML em um navegador. Você verá uma lista de filmes populares provenientes da API do TMDb e as principais músicas da Last.fm, todos exibidos na mesma página. Este exemplo prático ilustra bem o processo de desenvolvimento da integração de duas APIs distintas em um projeto e, com isso, você pode colocar a mão na massa e ter uma experiência real de como os aplicativos do mundo real podem depender de múltiplas fontes de dados e utilizar os serviços proporcionados por uma API.



Disponível em:

<https://www.shutterstock.com/pt/image-vector/web-development-programmer-engineering-coding-website-203744651>

3. Acesso em: 29 fev. 2024.

## 11.3 Desenvolvendo um projeto que utiliza APIs externas

Para continuar praticando os seus conhecimentos, vamos desenvolver um projeto mais completo que utiliza uma API externa. Neste projeto, criaremos uma aplicação de notícias, na qual os usuários podem obter as últimas notícias de diferentes fontes. Para isso, utilizaremos a API NewsAPI, um banco de dados que reúne notícias do mundo todo, para obter os dados.



**Level up!**

Abra um ambiente de desenvolvimento de sua preferência, como o **CodePen** (<https://codepen.io/>) ou o **Visual Studio Code** (<https://code.visualstudio.com/>) e siga o passo a passo que veremos a seguir.

### Passo 1 - Estrutura básica do projeto

Aqui, note que a estrutura HTML inclui um contêiner adicional para exibir as notícias:

```
<!DOCTYPE html>
<html lang="en">
<head>

<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
<title>Aplicação de Notícias</title>
<link rel="stylesheet" href="styles.css">

</head>

<body>

<div id="app">

 <h1>Últimas Notícias</h1>
 <div id="news-container"></div>

</div>

<script src="script.js"></script>

</body>

</html>
```

## Passo 2 - Estilo CSS (styles.css)

```
body {
 font-family: Arial, sans-serif;
 background-color: #f0f0f0;
 margin: 0;
 padding: 0;
 display: flex;
 flex-direction: column;
 align-items: center;
 justify-content: center;
 height: 100vh;
}

#app {
 background-color: #fff;
 border-radius: 8px;
 box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
 padding: 20px;
```

```
width: 600px;
text-align: center;

}

#news-container {
 margin-top: 20px;
 display: grid;
 grid-template-columns: repeat(auto-fill, minmax(250px, 1fr));
 gap: 20px;

}
```

Nesse momento, o CSS foi atualizado para estilizar adequadamente os elementos da aplicação de notícias.



### Code challenge

Experimente fazer mais modificações em seu projeto, editando os estilos no código CSS da maneira que você quiser. O céu é o limite!

### Passo 3 - Script JavaScript (script.js)

```
document.addEventListener('DOMContentLoaded', () => {
 const apiKey = 'SUA_CHAVE_API_NEWSAPI'; // Substitua com sua chave da NewsAPI
 const apiUrl = 'https://newsapi.org/v2/top-headlines';
 const newsSources = 'bbc-news,cnn,techcrunch';
 const fetchData = async () => {

 try {
 const response = await fetch(` ${apiUrl}?sources=${newsSources}&apiKey=${apiKey}`);
 if (!response.ok) {
 throw new Error(`Erro na resposta da API: ${response.status}`);
 }
 }
 };
});
```

```

 }

 return await response.json();

} catch (error) {

 console.error('Erro na requisição:', error);

}

};

const displayNews = (articles) => {
 const newsContainer = document.getElementById('news-container');
 articles.forEach((article) => {
 const newsCard = document.createElement('div');
 newsCard.classList.add('news-card');
 const title = document.createElement('h3');
 title.textContent = article.title;
 const description = document.createElement('p');
 description.textContent = article.description;
 const link = document.createElement('a');
 link.href = article.url;
 link.target = '_blank';
 link.textContent = 'Leia mais';

 newsCard.appendChild(title);
 newsCard.appendChild(description);
 newsCard.appendChild(link);
 newsContainer.appendChild(newsCard);
 }) ;
};

const init = async () => {
 const newsData = await fetchData();
 displayNews(newsData.articles);

};

init();

```

```
});
```

Analise o código novamente e perceba que o *script* JavaScript foi modificado para se adaptar à nova API (NewsAPI) e exibir notícias. Ele utiliza a mesma abordagem de requisição à API e de renderização dinâmica na página.

#### Passo 4 - Chave da API

Antes de executar o exemplo, substitua '**SUA\_CHAVE\_API\_NEWSAPI**' pela sua chave de API da NewsAPI. A seguir, veja como isso deve ser feito:

1

Acesse o *site* da NewsAPI em <https://newsapi.org/>.

2

Clique no botão "**Get API Key**" ou "**Sign Up**" para criar uma conta. Geralmente, essas opções estão localizadas no canto superior direito da página inicial.

3

Se você já tiver uma conta, faça *login*. Caso contrário, siga as instruções para criar uma nova conta, fornecendo as informações solicitadas.

4

Após o *login*, você será redirecionado para o painel da sua conta. Nele, você encontrará sua chave de API da NewsAPI. Geralmente, ela é exibida no painel ou em uma seção dedicada às suas configurações de API.

5

Copie sua chave de API, que geralmente tem um formato alfanumérico longo, como "abcd1234efgh5678ijkl9omnop".

6

Agora, abra o exemplo de código em que está escrito '**SUA\_CHAVE\_API\_NEWSAPI**' e substitua essa *string* pela chave de API que você acabou de copiar.

7

Salve o arquivo após fazer a substituição para que o processo seja realizado.

 Pronto! Após todos esses passos, você terá substituído com sucesso a *string* '**SUA\_CHAVE\_API\_NEWSAPI**' pela sua própria chave de API da NewsAPI, permitindo que o exemplo de código seja executado corretamente.

## Passo 5 - Etapa final

Após substituir a chave da API, abra o arquivo HTML em um navegador. Você deverá ver as últimas notícias de fontes como BBC News, CNN e TechCrunch exibidas na página!



E aí, deu tudo certo no seu *app* de notícias?  
Após seguir todas as etapas desse projeto, agora você já sabe como desenvolver um aplicativo completo que utiliza uma API externa para fornecer informações em tempo real!



### Bônus pela leitura!

Parabéns! Esta é sua recompensa pela leitura. Você ganhou a sugestão de um vídeo que indica cinco APIs públicas para você usar nos seus projetos de programação.

Aproveite!



Fonte: Vídeo 5 APIs públicas para projetos de programação que você precisa conhecer, do canal do YouTube Programador Aventureiro. Disponível em: <https://www.youtube.com/watch?v=H3KP7odJqQY>. Acesso em: 10 mar. 2024.



Neste capítulo, estudamos sobre a relevância das APIs no cenário do desenvolvimento *web*, focando na análise e no desenvolvimento de dois projetos de integração de API com base em HTML, CSS e JavaScript, que são as linguagens base que você já deve conhecer.

Inicialmente, aprendemos que os aplicativos podem depender de múltiplas APIs, como, por exemplo, as plataformas de previsão do tempo e tráfego e as redes sociais. Esses dois exemplos deixam claro como a integração de diversas APIs permite a criação de experiências completas e personalizadas para os usuários, entregando informações em tempo real para os usuários.

Em seguida, analisamos o desenvolvimento de um projeto que integra duas APIs distintas para exibir informações sobre filmes e músicas em uma única aplicação. A partir disso, pudemos compreender como é feito o processo de requisição, manipulação de dados e renderização na página, ajudando a consolidar as suas habilidades de integração de APIs.

Por fim, acompanhamos o desenvolvimento de um projeto mais completo, sendo uma aplicação de notícias que utiliza a API externa para fornecer notícias de diferentes fontes. Isso permitiu que você praticasse os seus conhecimentos sobre a integração de APIs de forma mais aprofundada, exercitando as suas habilidades no desenvolvimento de aplicações *web*. Ao compreender como integrar e utilizar APIs de maneira eficaz, você estará preparado(a) para desafios mais complexos no desenvolvimento *web*, sendo capaz de criar aplicações dinâmicas e inovadoras, enriquecendo a experiência digital dos usuários.



## ATIVIDADE DE FIXAÇÃO

- 1.** Qual é a principal função de uma API em um aplicativo de música?
  - a. Gerenciar a interface gráfica do aplicativo.
  - b. Fornecer recursos de monetização.
  - c. Permitir a comunicação e a integração com outros sistemas.
  - d. Controlar a reprodução de músicas.
  
- 2.** Qual é o principal objetivo da autenticação usando o protocolo OAuth 2.0 em uma API?
  - a. Gerenciar a interface do usuário.
  - b. Controlar o acesso a recursos protegidos.
  - c. Fornecer informações sobre artistas.
  - d. Realizar análise de áudio.
  
- 3.** Em um aplicativo de previsão do tempo, qual tipo de API seria mais apropriado para obter dados meteorológicos em tempo real?
  - a. API de streaming de música.
  - b. API de metadados musicais.
  - c. API de previsão do tempo.
  - d. API de busca e recomendação.
  
- 4.** Qual é a diferença entre uma API pública e uma API privada?
  - a. APIs públicas são gratuitas, enquanto APIs privadas exigem pagamento.
  - b. APIs públicas são acessíveis por qualquer pessoa, enquanto APIs privadas têm restrições de acesso.
  - c. APIs públicas são usadas apenas para fins de teste, enquanto APIs privadas são usadas em produção.
  - d. APIs públicas são mais seguras do que APIs privadas.

5. Quais são os benefícios de se utilizar várias APIs no desenvolvimento de aplicativos de mídias sociais?

  - a. Melhorar a interface do usuário.
  - b. Incorporar anúncios.
  - c. Integrar funcionalidades como autenticação, compartilhamento e análise de dados.
  - d. Reduzir a segurança.
6. Descreva um cenário hipotético de aplicação que poderia se beneficiar da integração de múltiplas APIs, fornecendo detalhes sobre as funcionalidades e as APIs envolvidas.
7. Explique o papel da função **fetch()** em um código que está sendo desenvolvido para um projeto de integração de APIs. Além disso, forneça um trecho de código onde esta função é utilizada.
8. Discorra sobre os potenciais desafios relacionados à segurança e à autenticação das requisições que podem surgir durante o desenvolvimento de um projeto que utiliza APIs externas. Em seguida, apresente algumas estratégias para lidar com esses desafios.
9. Considerando o projeto do aplicativo de notícias que desenvolvemos, como a manipulação de respostas da API e a renderização dinâmica são realizadas em JavaScript? Forneça um trecho de código para ilustrar.
10. Suponha que você está encarregado de desenvolver um aplicativo que depende de três APIs distintas. Descreva os benefícios dessa abordagem e como você lidaria com eventuais problemas de sincronização entre as diferentes fontes de dados.

## Referências

**SANTOS**, Elinei. Introdução à Programação Numérica em Python - Uma Abordagem Através da Solução de Problemas Físicos e Matemáticos. Rio de Janeiro: Editora Ciência Moderna, 2021. ISBN: 9786558420095.

**FORBELLONE**, André Luiz Villar; EBERSPÄCHER, Henri Frederico. *Lógica de Programação: A Construção de Algoritmos e Estruturas de Dados*. 4. ed. São Paulo: Pearson Prentice Hall, 2019.

**PUGA**, Sandra; RISSETTI, Gerson. Lógica de Programação e Estrutura de Dados: Com Aplicações em Java. Rio de Janeiro: LTC, 2015. ISBN: 978-85-216-2151-8.

**CRUZ**, Felipe. Python: Escreva Seus Primeiros Programas. 1<sup>a</sup> ed. São Paulo: Casa do Código, 2012. ISBN: 978-85-8260-572-1.

**LUTZ**, Mark; ASCHER, David. *Aprendendo Python: Do Iniciante ao Profissional*. São Paulo: Novatec, 2014.

**JACOBSON**, Daniel; BRAIL, Greg; WOODS, Dan. APIs: A Strategy Guide: Creating Channels with Application Programming Interfaces. Capa comum – Ilustrado. 27 de dezembro de 2011. Edição em inglês. São Paulo: O'Reilly Media, Inc., 2011. ISBN: 9781449308926.

**PokeAPI**. "Documentação da PokeAPI (v2)." Disponível em: PokeAPI Documentation .

**MANZANO**, José Augusto N. G.; OLIVEIRA, Jayr Figueiredo de. *Algoritmos: Lógica para Desenvolvimento de Programação de Computadores*. 28. ed. São Paulo: Érica, 2018.

**SILVEIRA**, Paulo; ALMEIDA, Adriano. Lógica de Programação: Crie Seus Primeiros Programas Usando Javascript e HTML. 1<sup>a</sup> ed. São Paulo: Casa do Código, 2012. ISBN: 978-85-8260-572-1.

**RAMALHO**, Luciano. *Python Fluente: Programação Clara, Concisa e Eficaz*. São Paulo: Novatec, 2016.

**BORGES**, Luiz Eduardo. Python Para Desenvolvedores: Aborda Python 3.3. 1<sup>a</sup> ed. São Paulo: Novatec Editora, 2014. ISBN: 978-8575224052.

**ZIVIANI**, Nivio. *Projeto de Algoritmos com Implementações em Pascal e C*. 4. ed. São Paulo: Cengage Learning, 2015.

**FLANAGAN**, David. JavaScript: O Guia Definitivo. 6<sup>a</sup> ed. Porto Alegre: Bookman, 2013. ISBN: 978-85-65837-19-4.