

INFORMAÇÕES

MÓDULO	AULA	INSTRUTOR
04 JavaScript Essencial e Consumo de APIs	06 - Introdução a APIs e Fetch	Pedro Henrique Amadio

CONTEÚDO

1. Nas Últimas Aulas...

Antes de mergulharmos nas APIs, vamos relembrar o que vimos até aqui:

Aula 01 - Introdução ao JavaScript

- Como o JS roda no navegador.
- Variáveis, tipos de dados, operações básicas.

Aula 02 - Condicionais e Loops

- Decisões (if/else), loops, short-circuit, filter/map/forEach.
- Importante: loops e estruturas de repetição serão úteis para iterar dados vindos de APIs.

Aula 03 - Funções e Escopo

- Funções tradicionais e arrow functions.
- Parâmetros, retornos, closures.
- Importante: Fetch usa callbacks e promises → diretamente ligado a funções!

Aula 04 - DOM e Eventos

- Seletores, criação de elementos, eventos de clique e teclado.
- Importante: Vamos exibir dados de APIs no DOM ou criar elementos de acordo com o retorno da API.

Aula 05 - Validação de Formulários

- Captura de eventos submit, manipulação do DOM, feedback dinâmico.
- Importante: FormData e manipulação do DOM serão úteis para montar telas com dados externos.

Agora juntaremos tudo: JS + Funções + DOM + Eventos + Estruturas lógicas para consumir APIs reais.

2. O que é uma API?

API significa Application Programming Interface.

É uma interface que permite que uma aplicação converse com outra.

Exemplos de APIs conhecidas:

- ViaCEP (consulta CEP).
- API do GitHub (buscar perfil de usuário).
- API de cotação (dólar, euro).
- API do OpenWeather (tempo/clima).

Para entender melhor...

Uma API é como um garçom: Você faz o pedido, ele leva para a cozinha e traz o resultado.

3. O que é REST?

REST (Representational State Transfer) é um estilo de arquitetura para criar APIs que seguem regras simples e padronizadas.

Uma API REST geralmente trabalha com:

- URLs claras (chamadas de **endpoints**)
- Verbos HTTP (GET, POST, PUT, DELETE)
- Respostas no formato **JSON**

Aqui vamos mostrar isso usando **APIs reais**, gratuitas e abertas.

1. Exemplo 1 - ViaCEP (Brasil)

API pública e gratuita para consultar CEPs.

Endpoint (GET):

```
https://viacep.com.br/ws/01001000/json/
```

Retorno:

```
{  
  "cep": "01001-000",  
  "logradouro": "Praça da Sé",  
  "bairro": "Sé",  
  "localidade": "São Paulo",  
  "uf": "SP"  
}
```

2. Exemplo 2 - GitHub API (Global)

Usada para buscar informações de usuários.

Endpoint (GET):

```
https://api.github.com/users/pedro-amadio
```

Retorno:

```
{  
  "login": "pedro-amadio",  
  "id": 59256372,  
  "node_id": "MDQ6VXNlcjU5MjU2Mzcy",  
  "avatar_url": "https://avatars.githubusercontent.com/u/59256372?v=4",  
  "gravatar_id": "",  
  "url": "https://api.github.com/users/pedro-amadio",  
  "html_url": "https://github.com/pedro-amadio",  
  "followers_url": "https://api.github.com/users/pedro-amadio/followers",  
  ...  
}
```

3. Exemplo 3 - AwesomeAPI (Cotações de moedas)

API pública e gratuita para cotações de moedas.

Endpoint (GET):

```
https://economia.awesomeapi.com.br/json/last/USD-BRL  
...Esquema...  
https://economia.awesomeapi.com.br/json/last/<moeda1>-<moeda2>
```

Retorno:

```
{  
  "USDBRL": {  
    "code": "USD",  
    "codein": "BRL",  
    "name": "Dólar Americano/Real Brasileiro",  
    "high": "5.2953",  
    "low": "5.28423",  
    "varBid": "0.0004",  
    "pctChange": "0.00756",  
    "bid": "5.2918",  
    "ask": "5.2948",  
    ...  
    ...  
  }  
}
```

O que é um Endpoint?

Um **endpoint** é um endereço (URL) que representa um recurso específico em uma API.

- Cada endpoint serve para uma coisa
- Cada um representa uma ação possível

Pense assim:

O endpoint é como uma “porta específica” para acessar uma informação ou funcionalidade da API.

4. O que são Métodos HTTP?

Métodos HTTP são **verbos** que indicam qual tipo de ação queremos realizar na API.

MÉTODO	AÇÃO	QUANDO USAR
GET	Buscar dados	Quando queremos somente consultar
POST	Criar um novo registro	Quando enviamos algo novo ao servidor
PUT	Substituir um registro inteiro	Quando queremos atualizar 100% de um recurso
PATCH	Atualizar parcialmente	Alterar só alguns campos
DELETE	Remover um registro	Quando queremos deletar algo

4.1. GET (Ler/Buscar dados)

É o método mais usado quando estamos começando a consumir APIs.

Ele não altera nada no servidor, só consulta.

Exemplo 1: ViaCEP

GET <https://viacep.com.br/ws/01001000/json/>

Traz o endereço do CEP.

Exemplo 2: GitHub

GET <https://api.github.com/users/pedro-amadio>

Traz dados do usuário Diego (Rocketseat).

4.2. POST (Criar novo registro)

POST é usado quando queremos enviar dados e criar algo no servidor.

Exemplo API fictício

Endpoint:

POST <https://api.meuapp.com.br/api/users>

Body enviado:

```
{  
  "name": "Pedro",  
  "job": "Developer"  
}
```

Resposta:

```
{  
  "id": "892",  
  "name": "Pedro",  
  "job": "Developer",  
  "createdAt": "2025-02-14T12:12:00.123Z"  
}
```

4.3. PUT (Atualizar TUDO)

PUT substitui todo o recurso.

Exemplo API fictício

Endpoint:

```
PUT https://api.meuapp.com.br/api/users/892
```

Body enviado:

```
{  
  "name": "Pedro",  
  "job": "Fullstack"  
}
```

Resposta:

```
{  
  "id": "892",  
  "name": "Pedro",  
  "job": "Fullstack",  
  "createdAt": "2025-02-14T12:12:00.123Z",  
  "updatedAt": "2025-02-15T12:12:00.123Z"  
}
```

4.4. PATCH (Atualizar PARTE do recurso)

PATCH atualiza somente alguns campos.

Exemplo API fictício

Endpoint:

```
PATCH https://api.meuapp.com.br/api/users/892
```

Body enviado:

```
{  
  "job": "Design"  
}
```

Resposta:

```
{  
  "id": "892",  
  "name": "Pedro",  
  "job": "Design",  
  "createdAt": "2025-02-14T12:12:00.123Z",  
  "updatedAt": "2025-02-15T12:12:00.123Z"  
}
```

Atualiza apenas o job.

4.5. DELETE (Apagar)

DELETE Usado para excluir recursos no servidor.

Exemplo API fictício

Endpoint:

```
DELETE https://api.meuapp.com.br/api/users/892
```

Body enviado:

Não envia-se BODY

Resposta:

```
{  
    ok: true,  
    mensagem: "Usuário deletado com sucesso."  
}
```

Quando usar cada método

OBJETIVO	MÉTODO CERTO	EXEMPLO
Consultar um CEP	GET	ViaCEP
Criar uma conta	POST	API Auth
Editar nome/senha	PUT ou PATCH	PUT = enviar tudo, PATCH = mudar só 1 campo
Excluir uma foto	DELETE	API de uploads
Listar posts de um blog	GET	/posts
Criar um post	POST	/posts
Editar um post	PUT/PATCH	/posts/5

5. Estrutura de um JSON

JSON significa **JavaScript Object Notation**.

É hoje o **formato padrão** usado por praticamente todas as APIs REST do mundo.

Ele é:

- leve
- simples
- legível
- compatível com qualquer linguagem

E o mais importante: **JSON é baseado na sintaxe de objetos do JavaScript**, por isso ele se encaixa perfeitamente no nosso curso.

Um JSON é composto por:

Objetos

```
{  
    "nome": "Pedro",  
    "idade": 24  
}
```

Arrays

```
{  
    "frutas": ["maçã", "banana", "uva"]  
}
```

Valores válidos no JSON:

- string
- number
- boolean
- null
- object

- array

Exemplo completo:

```
{
  "usuario": {
    "id": 12,
    "nome": "Pedro",
    "ativo": true
  },
  "permissoes": ["admin", "editor"],
  "notas": [10, 7.5, 9],
  "ultimoAcesso": null
}
```

JSON NÃO é um objeto JavaScript!

- JSON é TEXTO
- Objeto JavaScript é CÓDIGO

Exemplos:

```
// JSON (texto)
"{ \"nome\": \"Pedro\", \"idade\": 24 }"
```

```
// Objeto JavaScript (estrutura)
{ nome: "Pedro", idade: 24 }
```

- Convertendo JSON para Objeto JavaScript

As APIs SEMPRE devolvem respostas como **texto JSON**, e precisamos converter para objeto.

Usamos:

```
JSON.parse()
```

Exemplo:

```
const json = '{ "nome": "Pedro", "idade": 24 }';
const obj = JSON.parse(json);

console.log(obj.nome); // Pedro
console.log(typeof obj); // object
```

- Convertendo Objeto JavaScript para JSON

Para enviar dados para uma API (POST, PUT, PATCH)

Usamos:

```
JSON.stringify()
```

Exemplo:

```
const usuario = { nome: "Pedro", idade: 24 }; //Objeto
const json = JSON.stringify(usuario); //JSON (String)

console.log(json);
// {"nome":"Pedro","idade":24}

console.log(typeof json); // string
```

- `JSON.stringify` com “replacer” e “space”

Formatando JSON com identação

Usamos:

```
const pessoa = { nome: "João", idade: 30, ativo: true };
```

```
console.log(JSON.stringify(pessoa, null, 2));
```

Resultado:

```
{
  "nome": "João",
  "idade": 30,
  "ativo": true
}
```

Isso é ótimo para debug.

Como o JSON aparece em APIs reais

Exemplo - API do GitHub:

```
{
  "login": "pedro-amadio",
  "id": 59256372,
  "node_id": "MDQ6VXNlcjU5MjU2Mzcy",
  "avatar_url": "https://avatars.githubusercontent.com/u/59256372?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/pedro-amadio",
  "html_url": "https://github.com/pedro-amadio",
  "followers_url": "https://api.github.com/users/pedro-amadio/followers",
  ...
}
```

Para acessar no JS:

```
dados.login  
dados.followers  
dados.avatar_url
```

JSON dentro de Arrays

APIs normalmente retornam **listas**.

```
[
  {
    "name": "Brazil",
    "population": 212600000
  },
  {
    "name": "Argentina",
    "population": 45000000
  }
]
```

Manipulação no JS:

```
dados[0].name  
dados[1].population
```

Ou percorrendo:

```
dados.forEach(pais => console.log(pais.name));
```

6. Introdução ao fetch()

fetch() é a função nativa do JavaScript usada para fazer requisições HTTP, ou seja, conversar com APIs, buscar dados externos e enviar informações para servidores.

Ele substituiu tecnologias antigas como:

- XMLHttpRequest
- jQuery.ajax()

Hoje, praticamente 100% das aplicações modernas utilizam fetch().

O que é o fetch()?

É uma função global:

```
fetch(url)
```

Ela recebe no mínimo um parâmetro: a URL do endpoint.

Ela retorna uma Promise, o que significa que o JavaScript não espera a resposta imediatamente.

Isso é importante porque:

- Requisições demoram
- A página não trava
- O JS continua rodando no fundo

Estrutura completa da requisição

Tudo começa com:

```
fetch("https://api.github.com/users/pedro-amadio")
```

Mas fetch NÃO retorna imediatamente os dados.

O retorno inicial é um objeto Response:

```
Response {  
  status: 200,  
  ok: true,  
  json: function() {...},  
  ...  
}
```

Por isso precisamos de DUAS etapas:

ETAPA 1 → Receber a resposta do servidor

```
const resposta = await fetch(url);
```

ETAPA 2 → Ler o corpo da resposta (JSON)

```
const dados = await resposta.json();
```

Como funciona o Response.json()?

O método:

```
resposta.json()
```

converte o texto JSON enviado pelo servidor em um objeto JavaScript.

Se você tentar exibir resposta.json sem await:

- Você recebe uma Promise [ERRADO]

- Precisa do await para transformar o texto em dados utilizáveis [CORRETO]

Exemplo completo usando then()

A forma clássica:

```
fetch("https://api.github.com/users/pedro-amadio")
  .then(res => res.json())resposta
  .then(data => console.log(data));
```

Fluxo:

1. faz a requisição
2. res é a resposta (status, headers...)
3. res.json() converte o corpo
4. data contém os dados utilizáveis

Exemplo moderno usando async/await (recomendado)

```
async function carregar() {
  const resposta = await fetch("https://api.github.com/users/pedro-
amadio");
  const dados = await resposta.json();
  console.log(dados);
}

carregar();                                com arrow
                                                function ->      const carregar = async=>{
                                                fetch("https://api.github.com/users/p
amadio");
                                                const dados = await resposta.json();
                                                console.log(dados);
```

Por que async/await é melhor?

- Código mais limpo
- Leitura mais natural
- Fácil de tratar erros com try/catch
- Usado em frameworks modernos (React, Next.js, Vue, Angular)

fetch() com parâmetros dinâmicos

Exemplo com nome de usuário digitado:

```
const user = document.getElementById("campo").value;

const resposta = await fetch(`https://api.github.com/users/${user}`);
const dados = await resposta.json();
```

7. Configurando requisições com método, headers e body

fetch() aceita um segundo parâmetro com configurações:

```
fetch("https://api.meuapp.com.br/api/users", {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({
    name: "Pedro",
    job: "Developer"
  })
});
```

Essa estrutura permite:

- POST
- PUT
- PATCH
- DELETE

E enviar dados no formato esperado pela API.

API token ou API Key

Às vezes, para acessar uma api precisamos ter um TOKEN ou KEY que são chaves de acesso. Por mais que uma API seja pública, nem todas deixam o acesso 100% livre para evitar abusos ou uso mal-intencionado.

- Obtendo acesso à uma API via token:

Para obter acesso talvez seja necessário se cadastrar ou solicitar acesso ao suporte do provedor da API.

Uma vez que tiver o seu TOKEN, basta configurar o cabeçalho da sua API e passar o TOKEN como Bearer. Vemos a seguir:

```
fetch("https://api.meuapp.com.br/api/users", {
  method: "GET",
  headers: {
    "Content-Type": "application/json",
    "Authorization": "Bearer <SEU TOKEN AQUI>"
  }
});
```

Para esse exemplo, o método e o “Body” não fazem diferença, o que importa é alterar o “<SEU TOKEN AQUI>” pelo seu TOKEN obtido anteriormente.

- Obtendo acesso à uma API via API-KEY:

Algumas APIs são ainda mais rigorosas e, além de solicitar uma autorização (Seja ela via Bearer-TOKEN, user-password, etc) ainda solicitam uma API-KEY que é informada no HEADER da requisição também. As vezes aparece como “X-API-KEY” ou “API-KEY” dependendo de como foi desenvolvida. **Por isso é sempre importante ler a documentação da API antes de começar a usa-la.**

```
fetch("https://api.meuapp.com.br/api/users", {
  method: "GET",
  headers: {
    "Content-Type": "application/json",
    "Authorization": "Bearer <SEU TOKEN AQUI>", //Opcional, as vezes
    "X-API-KEY": "API-KEY-AQUI"
  }
});
```

Para esse exemplo, o que importa é alterar o “API-KEY-AQUI” pela sua CHAVE(KEY) obtida anteriormente.

8. Verificando status da requisição

O fetch **não gera erro automático** quando o servidor retorna falha (como 404). Por isso, você **SEMPRE** deve verificar:

```
if (!resposta.ok) {
  console.log("Falha:", resposta.status);
}
```

CÓDIGO	SIGNIFICA
200	OK (sucesso)
201	Criado com sucesso
400	Erro de validação
401	Não autorizado
403	Proibido
404	Não encontrado
500	Erro no servidor

Tratamento de erros com try/catch

```
try {
  const resposta = await fetch(url);

  if (!resposta.ok) {
    throw new Error("Erro HTTP: " + resposta.status);
  }

  const dados = await resposta.json();
  console.log(dados);

} catch (erro) {
  console.log("Erro ao tentar acessar API:", erro);
}
```

O catch captura:

- falha de internet
- endpoint fora do ar
- CORS bloqueado
- erros de código

9. Fetch e CORS

Muitas APIs bloqueiam requisições diretas do navegador por motivos de segurança.

Se ao usar fetch aparecer:

CORS error [*] -> **usar todos**

- Não é erro no JavaScript
- É bloqueio do servidor
- Usa-se proxy, backend ou API configurada corretamente

10. Exemplo completo + DOM

```

document.getElementById("buscar").addEventListener("click", async () => {
  const nome = document.getElementById("user").value;
  const resultado = document.getElementById("resultado");

  try {
    resultado.textContent = "Carregando...";

    const resposta = await fetch(`https://api.github.com/users/${nome}`);

    if (!resposta.ok) {
      resultado.textContent = "Usuário não encontrado!";
      return;
    }

    const dados = await resposta.json();

    resultado.innerHTML = `
      
      <h2>${dados.name}</h2>
      <p>Seguidores: ${dados.followers}</p>
    `;
  } catch {
    resultado.textContent = "Erro ao conectar com a API.";
  }
});
```

11. Atividade Prática em Sala

- 1) Buscador de Usuários do GitHub
 - Campo de texto + botão
 - Buscar API do GitHub
 - Exibir nome, seguidores, avatar, bio
 - Tratar erros
 - Usar `async/await`

JS:

```
document.getElementById("buscar").addEventListener("click", async () => {
  const nome = document.getElementById("user").value;
  const resultado = document.getElementById("resultado");

  try {
    resultado.textContent = "Carregando...";

    const resposta = await fetch(`https://api.github.com/users/${nome}`);

    if (!resposta.ok) {
      resultado.textContent = "Usuário não encontrado!";
      return;
    }

    const dados = await resposta.json();

    resultado.innerHTML =
      `
       <h2>${dados.name}</h2>
       <p>Seguidores: ${dados.followers}</p>
      `;
  } catch {
    resultado.textContent = "Erro ao conectar com a API.";
  }
});
```

2) Criar elementos com dados da API

- Usar a API <https://api.adviceslip.com/> para obter conselhos
- Usar o endpoint <https://api.adviceslip.com/advice/search/<pesquisa>>
- Botão “Buscar conselhos de vida”
- Div com o resultado
- Iterar sobre arrays
- Usar o createElement("p") para criar um elemento

HTML:

```
<button id="btn">Buscar conselhos de vida</button>
<div id="resultado"></div>
```

JS:

```

document.getElementById("btn").addEventListener("click", async () => {
  const area = document.getElementById("resultado");
  area.textContent = "Carregando...";

  try {
    const resposta = await
fetch("https://api.adviceslip.com/advice/search/life");
    const dados = await resposta.json();

    // Limpa antes de renderizar
    area.textContent = "";

    // Se a API não encontrar nada
    if (!dados.slips) {
      const p = document.createElement("p");
      p.textContent = "Nenhum conselho encontrado.";
      p.style.color = "red";
      area.appendChild(p);
      return;
    }

    // Título
    const titulo = document.createElement("h3");
    titulo.textContent = `Foram encontrados ${dados.slips.length}
conselhos:`;
    area.appendChild(titulo);

    // Lista de conselhos
    const lista = document.createElement("ul");

    dados.slips.forEach(item => {
      const li = document.createElement("li");
      li.textContent = item.advice;
      li.style.marginBottom = "6px";
      lista.appendChild(li);
    });

    area.appendChild(lista);
  } catch (erro) {
    area.textContent = "Erro ao buscar conselhos.";
  }
});

```

12. Para casa

Projeto: Consulta de CEP usando ViaCEP

1. Criar input e botão.
2. Validar CEP (8 dígitos).
3. Consumir:

<https://viacep.com.br/ws/01001000/json/>

- 4. Exibir logradouro, bairro, cidade e UF.
- 5. Tratar erros com try/catch.
- 6. Usar feedback visual (cores) conforme aula 05.