# Consumo de APIs

### Introdução

Bem-vindos à nossa aula sobre o "Consumo de APIs", um componente vital na disciplina de Programação II. Neste módulo, mergulharemos no universo das APIs, explorando desde os fundamentos teóricos até a aplicação prática. Decifraremos o funcionamento das APIs através de conceitos-chave como eventos, callbacks e promessas, e entenderemos como esses elementos se integram no desenvolvimento de software. Além disso, examinaremos os módulos de consumo de API no Node.js, como Axios e Fetch, e a utilização de APIs falsas para simulação e teste. Esse curso é projetado para fornecer uma compreensão abrangente do consumo de APIs, essencial para qualquer desenvolvedor que deseja aprimorar suas habilidades em programação e desenvolvimento de aplicativos modernos.

### Conceito de eventos, callback e promessas

Neste módulo, focaremos no universo do consumo de APIs, explorando como esse processo funciona. Para compreender plenamente o mecanismo de consumo de APIs, é crucial dominar certos conceitos fundamentais no ambiente de programação, especialmente em Node.js, embora esses conceitos também se apliquem a outras linguagens.

Iniciaremos nossa jornada de aprendizado com o entendimento dos eventos, callbacks e promessas, fundamentais para a programação assíncrona em Node.js. Esses elementos são essenciais para o consumo eficiente de APIs e serão detalhados através de exemplos práticos.

1. Eventos

Eventos são ações ou ocorrências que o sistema ou aplicação pode perceber e responder. No Node.js, a arquitetura orientada a eventos utiliza emissores (emitters) e ouvintes (listeners):

```
// Evento

process.on('start', () => {

    console.log('Processo iniciado');

});
```

Neste código, `process.on` registra um ouvinte para o evento `start`, que é disparado com `process.emit('start')`, executando a função que imprime 'Processo iniciado'.

2. Callbacks

Callbacks são funções passadas como argumentos para outras funções, que são chamadas após a operação ser concluída. São usados para operações assíncronas, como leitura de arquivos ou requisições a APIs:

```
// Callback

const fetchData = (callback) => {

    setTimeout(() => {

        callback('Dados carregados');

    }, 1000);

};
```

No exemplo, `fetchData` usa `setTimeout` para simular uma operação assíncrona, chamando o callback após 1 segundo com a mensagem 'Dados carregados'.

3. Promessas

Promessas (Promises) representam a eventual conclusão (ou falha) de uma operação assíncrona, proporcionando uma maneira mais limpa de tratar operações assíncronas consecutivas:

```
// Promessa

const fetchDataPromise = () => {

    return new Promise((resolve) => {

        setTimeout(() => {

            resolve('Dados carregados via Promessa');

        }, 1000);

    });

};
```

Aqui, `fetchDataPromise` cria uma promessa que resolve após 1 segundo, demonstrando uma operação assíncrona.

4. Execução e integração

Ao integrar eventos, callbacks e promessas, temos um modelo completo de operações assíncronas em Node.js:

```
// Disparando o evento
```

```
process.emit('start');

// Usando o callback

fetchData(data => console.log(data));

// Usando a promessa

fetchDataPromise().then(data => console.log(data));
```

Esse fluxo ilustra como os eventos são emitidos, seguidos pela execução de callbacks e a resolução de promessas, evidenciando a sequência e a integração desses conceitos na programação assíncrona.

Prosseguiremos com a aplicação desses conceitos no consumo de APIs, especialmente focando em APIs falsas, para enriquecer nossa compreensão prática e teórica no desenvolvimento de software.

**Módulos de Consumo de API**

Vamos explorar alguns módulos de consumo de API, fundamentais para entendermos como efetuar requisições e obter respostas de APIs, sejam elas parte de nosso próprio código ou APIs públicas externas. No ecossistema do Node.js, temos diversos módulos dedicados a esse fim, e embora alguns já sejam conhecidos por nós, é essencial reconhecer a variedade disponível para atender às necessidades específicas de cada projeto.

Apresentarei três módulos principais utilizados no Node para consumo de APIs: Request, Axios e o método Fetch nativo. Apesar de o Request ter caído em desuso devido à sua depreciação, ainda serve como referência histórica, embora não seja recomendado para novos projetos. O Axios e o Fetch, por outro lado, são mais modernos e amplamente adotados em nossas práticas.

O Axios se destaca por sua integração eficiente com o Express e outras ferramentas de front-end, oferecendo uma série de funcionalidades como interceptação de requisições e respostas, transformação de dados e suporte a operações assíncronas baseadas em promessas. Já o Fetch, por ser nativo do navegador, facilita a implementação e execução de requisições HTTP, tornando a aplicação mais ágil e leve.

Para ilustrar, abordaremos o Axios detalhadamente, explorando sua documentação e funcionalidades. O Axios é um cliente HTTP isomórfico, ou seja, pode ser executado tanto no servidor (Node.js) quanto no cliente (navegador), utilizando XMLHttpRequest no lado do cliente. Suas principais vantagens incluem a conversão automática de dados para JSON e a proteção contra vulnerabilidades de segurança como XSRF.

Por exemplo, com Axios podemos fazer uma requisição GET simples:

```
const axios = require('axios');

axios.get('https://jsonplaceholder.typicode.com/posts/1')

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

    .catch(error => console.log(error));
```

Este código faz uma chamada à API JSON Placeholder e imprime a resposta no console.

Ao final desta aula, você terá uma compreensão sobre os módulos de consumo de API no Node.js, especialmente o Axios, preparando o terreno para nossas próximas discussões sobre a instalação e uso prático dessas ferramentas no desenvolvimento de projetos, incluindo a integração com APIs falsas para simular cenários reais de consumo.

**Conceito de APIs falsas para consumo**

Nesta parte da nossa aula sobre consumo de APIs, vamos nos aprofundar no conceito de APIs falsas, ou fake APIs, e entender como elas se encaixam no desenvolvimento de software. Pode parecer estranho inicialmente, mas as APIs falsas são ferramentas essenciais no processo de prototipagem, teste e validação de integrações em desenvolvimento de software.

As APIs falsas, também conhecidas como mock APIs, simulam o comportamento de APIs reais, permitindo aos desenvolvedores testar e ajustar suas aplicações antes de integrar com APIs de produção. Elas são particularmente úteis para validar a arquitetura de um sistema, realizar testes de integração, e permitir o desenvolvimento paralelo de diferentes partes de um projeto.

Vamos exemplificar com o JSON Server para criar uma API falsa rapidamente:

```
npx json-server --watch db.json
```

'db.json' é um arquivo que contém dados fictícios que representam os recursos da API falsa, permitindo testar o consumo desses dados em sua aplicação.

Um dos principais benefícios das APIs falsas é isolar componentes, o que possibilita testar partes específicas de um sistema sem a necessidade de ter toda a infraestrutura real disponível. Isso não apenas facilita a detecção de erros e a validação de funcionalidades específicas, mas também reduz a dependência de sistemas externos, minimizando os riscos associados a instabilidades ou mudanças inesperadas em APIs de terceiros.

Além disso, as APIs falsas permitem a simulação de diversos cenários, incluindo respostas de erro, variações de tempo de resposta e comportamentos atípicos, ajudando na preparação da aplicação para lidar com diferentes situações no ambiente de produção. A capacidade de

simular esses cenários contribui significativamente para a robustez e confiabilidade da aplicação desenvolvida.

Outro ponto importante é a contribuição das APIs falsas para a documentação e o design de interface. Através delas, é possível criar e aprimorar a documentação do projeto, facilitando o entendimento e a manutenção do sistema. A documentação não apenas serve como um guia para o desenvolvimento, mas também como um recurso valioso para novos integrantes da equipe ou para futuras referências.

Do ponto de vista econômico, o uso de APIs falsas representa uma economia de recursos significativa. Elas permitem testar e validar integrações sem o custo associado ao uso de serviços pagos ou à necessidade de ter uma infraestrutura completa de APIs reais disponível para testes.

Para exemplificar, vamos citar duas APIs falsas populares: o JSON Placeholder e o Fake Store API. O JSON Placeholder oferece uma ampla variedade de dados para testes e prototipagem, enquanto o Fake Store API é ideal para simular cenários de e-commerce, com dados de produtos, carrinhos de compra e informações de usuários.

Na prática, utilizaremos essas APIs falsas para demonstrar como integrá-las em um projeto, explorando suas funcionalidades e avaliando como elas podem ser utilizadas para testar e aprimorar as aplicações. A seguir, veremos como integrar uma API falsa em nosso projeto para ver como ela funciona e como pode ser consumida eficientemente.

**Consumo de uma API Falsa**

Na última parte desta aula sobre consumo de APIs, vamos focar na implementação prática de uma API falsa em um projeto front-end, usando a JSON Placeholder como exemplo. Essa API é frequentemente utilizada para

testar e simular a interação com um back-end, fornecendo dados falsos para desenvolvimento e testes.

Vamos detalhar o processo de consumo de uma API falsa. Ao usar a função Fetch para obter dados da JSON Placeholder, o código básico seria:

```
fetch('https://jsonplaceholder.typicode.com/posts')

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

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

    .catch(error => console.error('Erro ao buscar dados:', error));
```

Aqui, `fetch` é uma função nativa do JavaScript para realizar requisições HTTP. Nesse exemplo, ela é usada para fazer uma requisição GET à URL fornecida. O método `.then()` é encadeado para tratar a resposta, convertendo-a de JSON para um objeto JavaScript, seguido de outro `.then()` que loga o resultado no console. O `.catch()` é usado para tratar erros que possam ocorrer durante a requisição.

Para a implementação no ambiente de front-end, após atualizar os pacotes necessários no back-end e executar a aplicação, adicionaremos Axios como uma dependência no front-end. O Axios simplifica as requisições HTTP e oferece mais funcionalidades comparado ao Fetch.

No projeto front-end, criaremos uma pasta `fake` dentro do diretório `pages` para organizar os componentes relacionados à API falsa. Dentro desta pasta, um componente `Page.tsx` fará as requisições GET, similarmente ao que foi feito com Fetch, mas utilizando o Axios. Este componente pode ser estruturado da seguinte forma:

```
import React, { useEffect, useState } from 'react';
```

```
import axios from 'axios';

function FakeAPIPage() {

    const [data, setData] = useState(null);

    useEffect(() => {

        axios.get('https://jsonplaceholder.typicode.com/posts')

            .then(response => setData(response.data))

              .catch(error => console.error('Erro ao buscar
dados:', error));

    }, []);

    return (

        <div>

            <h1>Dados da API Falsa</h1>

            <pre>{JSON.stringify(data, null, 2)}</pre>

        </div>

    );

}

export default FakeAPIPage;
```

No exemplo acima, utilizamos React junto com Axios para fazer uma requisição GET assim que o componente é montado, usando o hook

`useEffect`. Os dados recebidos são armazenados no estado do componente com `useState` e, posteriormente, renderizados na interface.

Adicionando esse componente ao nosso aplicativo React, facilitamos a interação dos usuários com os dados provenientes da API falsa, o que simula a interação com um back-end real. Isso é importante para o desenvolvimento de front-end, pois permite testar a interface e a lógica do usuário antes da integração com o back-end real.

Concluindo, esta parte da aula comenta sobre como pode ser feito o consumo de dados de uma API falsa, preparando o front-end para a futura integração com um back-end real. Essa orientação reforça a compreensão sobre o funcionamento das APIs e a importância de testar e validar integrações em ambientes de desenvolvimento, estabelecendo uma base sólida para o desenvolvimento de aplicações web.

Com isso, finalizamos nossa discussão sobre o consumo de APIs, preparando o caminho para futuras explorações práticas e teóricas no desenvolvimento de software. Nas próximas aulas, aprofundaremos nosso conhecimento e habilidades em programação.

**GitHub da Disciplina**

Você pode acessar o repositório da disciplina no GitHub a partir do seguinte link: https://github.com/FaculdadeDescomplica/ProgramacaoII. Esse repositório tem como principal objetivo guardar os códigos das aulas práticas da disciplina para aprimorar suas habilidades em vários tópicos, incluindo a criação e consumo de APIs com controle de autenticação utilizando Node.js e utilizando boas práticas de programação e mercado.

**Conteúdo Bônus**

Para aqueles interessados em aprofundar seus conhecimentos sobre APIs falsas e como implementá-las, recomendo assistir ao vídeo "API fake com JSON SERVER" do canal Pablo Codes no YouTube. Esse conteúdo prático aborda a criação de uma API falsa utilizando o JSON Server, uma ferramenta simples e eficaz para simular APIs reais em seus projetos de desenvolvimento.

**Referências Bibliográficas**


*Bibliografia Básica:*

ELMASRI, R.; NAVATHE, S. B. **Sistemas de banco de dados.** 7.ed. Pearson: 2018.

MEDEIROS, L. F. de. **Banco de dados:** princípios e prática. Intersaberes: 2013.

VICCI, C. (Org.). **Banco de dados**. Pearson: 2014.

*Bibliografia Complementar:*

CARDOSO, L. da C. **Design de aplicativos**. Intersaberes: 2022.

JOÃO, B. do N. **Usabilidade e interface homem-máquina.** Pearson: 2017.

LEAL, G. C. L. **Linguagem, programação e banco de dados:** guia prático de aprendizagem. Intersaberes: 2015.

PUGA, S.; FRANÇA, E.; GOYA, M. **Banco de dados:** implementação em SQL, PL/SQL e Oracle 11g. Pearson: 2013.

SETZER, V. W.; SILVA, F. S. C. **Bancos de dados.** Blucher: 2005.



**Ir para exercício**