Sistema Fiep

FIEP SESI SENAI IEL

Programação Assíncrona

Programação Assíncrona

- Capacidade de **executar tarefas de forma não sequencial**, permitindo que o programa execute outras operações enquanto aguarda a conclusão de operações demoradas;

- Útil em situações em que a execução síncrona bloquearia o fluxo de execução do programa, resultando em uma experiência ruim para o usuário.

Benefícios

Melhor desempenho: permitem que o programa execute tarefas em paralelo, aproveitando melhor os recursos do sistema e reduzindo o tempo de resposta;

Responsividade: o programa pode responder a eventos externos, como interações do usuário ou respostas de servidores, sem ficar bloqueado esperando a conclusão de uma única operação;

Benefícios

Experiência do usuário aprimorada: evitam bloqueios e travamentos, proporcionando uma experiência de usuário mais suave e responsiva;

Uso eficiente de recursos: permite que o programa faça uso eficiente dos recursos do sistema, pois pode continuar trabalhando em outras tarefas enquanto aguarda operações de entrada/saída.

Por que usar em *JavaScript*?

Por ser projetada para ser executada em navegadores *web*, **contempla muitas operações que exigem operações assíncronas, tais como: requisições de rede; manipulação de eventos de usuário; espera pela conclusão de animações** etc .

Características	Programação Assíncrona	Programação Síncrona
Fluxo de Execução	Não sequencial	Sequencial
Responsividade	Responsivo	Bloqueante
Utilização de Recursos	Melhor aproveitamento dos recursos	Uso bloqueante dos recursos
Tempo de Execução	Não bloqueante	Bloqueante
Exemplo de Sintaxe	*Callbacks*, *Promises*, *Async/Await*	Estruturas de controle (*loops*, condicionais)
Tratamento de Erros	Tratamento explícito de erros usando *try/catch* ou *.catch()*	Tratamento de erros usando *try/catch*
Exemplo de Cenário	Requisições de rede, manipulação de arquivos, operações de banco de dados	Cálculos síncronos, leitura de arquivos pequenos

Função/Método	Descrição
Callbacks	Funções passadas como argumentos que serão chamadas após uma operação assíncrona ser concluída. Podem resultar em "*callbacks hell*" e dificuldade de legibilidade e manutenção do código.
Promises	Objeto que representa a eventual conclusão ou falha de uma operação assíncrona. Fornece métodos para tratar os resultados ou erros. Introduz uma sintaxe mais clara e encadeável.
Async/Await	Palavras-chave para trabalhar com funções assíncronas. *Async* declara uma função assíncrona, enquanto *Await* pausa a execução até que uma *Promise* seja resolvida. Torna o código mais legível e se assemelha à programação síncrona.

Função/Método	Descrição
Fetch API	Interface moderna para fazer requisições HTTP assíncronas. Retorna uma *Promise* que resolve em uma resposta HTTP. Simplifica a realização de solicitações e manipulação de dados recebidos.
setTimeout	Função que agenda a execução de uma função após um certo tempo definido em milissegundos. Permite a execução de tarefas assíncronas após um intervalo de tempo.
setInterval	Função que agenda a execução repetida de uma função em intervalos de tempo definidos em milissegundos. Útil para tarefas assíncronas que precisam ser realizadas em intervalos regulares.

	Callbacks	**Promises**	**Async/Await**
Sintaxe	Funções passadas como argumento	Objeto com métodos *.then()* e *.catch()*	Prefixo *async* e palavra-chave *await*
Manipulação de erros	Necessário tratar manualmente	Uso de *.catch()* para tratar erros	Uso de *try/catch* para tratar erros
Encadeamento	Encadeamento manual através de *callbacks*	Encadeamento simplificado com *.then()*	Encadeamento linear com *await*
Legibilidade	Pode resultar em "*callback hell*"	Melhor legibilidade com encadeamento	Mais legível e sem aninhamento excessivo
Tratamento de múltiplos retornos	Difícil de lidar com múltiplos *callbacks*	Possibilita múltiplos *.then()*	Tratamento simples com *await*
Suporte nativo	Suportado desde versões mais antigas de JS	Introduzido em ES6	Introduzido em ES8