Innominatum CONSULTORIA

Programação em JavaScript

ServiceNow - Deloitte

Rodrigo Costa

IPCA POLITÉCNICO DO CÁVADO E DO AVE
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU
IAPMEI Parcerias para o Crescimento
PORTUGAL DIGITAL

Innominatum CONSULTORIA

JavaScript

Módulo 2

IPCA POLITÉCNICO DO CÁVADO E DO AVE
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU
IAPMEI Parcerias para o Crescimento
PORTUGAL DIGITAL

Assincronismo – Parte 1

Sessão 12

Objetivo geral

Capacitar os formandos na compreensão do modelo de execução assíncrona do JavaScript, desenvolvendo uma visão clara de como a linguagem gere operações demoradas sem bloquear a execução principal. A sessão foca na transição de uma interpretação linear do código para o entendimento do funcionamento do Event Loop, permitindo prever a ordem de execução de tarefas assíncronas e compreender o papel da Call Stack, das Web APIs e da Callback Queue. Através da introdução progressiva a callbacks e Promises, os formandos aprendem a estruturar fluxos assíncronos de forma legível e robusta, aplicando também práticas adequadas de tratamento de erros em cenários síncronos e assíncronos.

Objetivos específicos

Ao final da sessão, os formandos deverão ser capazes de:

- Explicar a diferença entre execução síncrona e assíncrona;
- Descrever o funcionamento do Event Loop e o papel da Call Stack, Web APIs e Callback Queue;
- Utilizar callbacks em cenários simples e reconhecer o problema do callback hell;
- Utilizar Promises para lidar com operações assíncronas, compreendendo os seus estados e métodos principais;
- Aplicar tratamento de erros em código síncrono e assíncrono.

Síncrono vs Assíncrono

Fluxo de execução do código

Relembrando o que já sabemos:

- JavaScript executa **uma coisa de cada vez (single thread)**;
- Utiliza da callStack (Pilha de execução para coordenar a execução);
- Executa a leitura do ficheiro JS de cima para baixo.

O que significa síncrono?

A execução síncrona:

- Cada instrução espera a anterior terminar;
- A execução é bloqueante.

```
function primeiraFuncao() {
    console.log("Iniciando a primeira função...");
    for (let i = 0; i < 10000; i++) {
        // Simulando uma tarefa pesada
    }
    console.log("Primeira função concluída.");
}

// Essa função só será executada após a conclusão da primeira
function segundaFuncao() {
    console.log("Iniciando a segunda função...");
    for (let i = 0; i < 10000; i++) {
        // Simulando outra tarefa pesada
    }
    console.log("Segunda função concluída.");
}
primeiraFuncao();
segundaFuncao();
```

O Problema da execução síncrona

Se uma tarefa demora muito:

- Todo o programa espera;
- A interface bloqueia;
- Consequência má experiência para o utilizador.

O agendador do JS: setTimeOut()

- O que faz: Agenda a execução de uma função para o futuro;
- Como funciona: Recebe uma **callback** e um tempo em milissegundos;
- Natureza: Não bloqueante. O JS "dispara e esquece", seguindo a próxima linha sem esperar;
- Ele é uma funcionalidade do **navegador** (Web API), não do motor JS puro.

```
console.log("A");

// Utilizamos o setTimeout para agendar a execução de
// uma função após um determinado tempo (em milissegundos).
setTimeout(() => {
    console.log("B");
}, 1000);

console.log("C");

// Qual será a ordem de saída no console?
```

Onde ocorrem as operações assíncronas

Elas geralmente ocorrem em:

- Requisições Externas (APIs): Buscar dados de um servidor (ex: lista de produtos, clima, login);
- Interação com o Sistema (Ficheiros): Ler ou gravar arquivos;
- Controle de Tempo (Timers): setTimeout e setInterval;
- Interações do Utilizador: Cliques, digitação, scroll (o navegador "vigia" e espera o evento ocorrer);

Innominatum CONSULTORIA

O Event Loop

A Orquestra do Assincronismo: O Event Loop

- Call Stack (A Mesa): Onde o código síncrono é executado agora. (LIFO);
- Web APIs (Os Ajudantes): O navegador processa o que o JS não consegue (cronômetros, pedidos de rede, cliques)
- Callback Queue (A Fila): Onde as funções aguardam após o navegador terminar o trabalho pesado. (FIFO)
- Event Loop (O Maestro): O vigilante que move as funções da Fila para a Mesa apenas quando a mesa está limpa.

Innominatum CONSULTORIA

Callbacks: O problema do callback Hell

IPCA POLITÉCNICO DO CÁVADO E DO AVE
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU
IAPMEI Parcerias para o Crescimento
PORTUGAL DIGITAL

Callbacks: Nem tudo o que vai, volta (na hora)

- Recapitulando: Callback é uma função passada como argumento;
- Mito: "Toda callback é assíncrona“;
- Verdade: Depende de quem a executa:
  - Síncronas: Métodos de Array (forEach, map). Executam imediatamente na Call Stack.
  - Assíncronas: setTimeout, addEventListener, fetch. São delegadas à Web API e agendadas.
  - O Problema: Quando começamos a empilhar execuções dependentes, surge o Callback Hell.

CallBack Hell

Acontecem quando precisamos de várias operações em sequência e dependentes uma das outras, o que causam um aninhamento de callbacks que logo, se tornará uma dor de cabeça em nossa vida, repare que o código começa a crescer para direita e não para baixo.

```
login(usuario, () => {
    obterDados(() => {
        validarDados( () => {
            validarPermissaoDeAcesso(() => {
                processarDados(() => {
                    mostrarResultado();
                });
            });
        });
    });
});
```

Innominatum CONSULTORIA

Promises

O que é uma Promise?

- Definição: Um objeto especial que "guarda o lugar" de um valor que ainda não conhecemos.
- O Contrato: Ela garante que vai nos avisar quando a operação terminar, seja com sucesso ou com erro.
- Estado: Enquanto a operação ocorre, a Promise fica em estado de "espera" (pending).

Estados de uma Promise

Uma Promise pode estar 3 estados:

- Pending: Pending (Pendente): É o estado inicial. A operação assíncrona ainda está a processar;
- Fulfilled (Resolved): A operação terminou com sucesso. O método resolve() foi chamado e agora o .then() pode ser executado;
- Rejected: A operação falhou. O método reject() foi chamado e o fluxo será desviado para o .catch();

Sintaxe básica da Promise

A Promise é uma classe, que necessita ser instanciada e por isso usamos a palavra chave new para criar essa nova instância da classe

```
// Declaração de uma promise
const promessa = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("Concluído");
    }, 1000)
})

// Execução de uma promise
promessa.then(resultado => {
    console.log(resultado);
});
```

Promise x CallBack

```
login(usuario)
    .then(obterDados)
    .then(validarDados)
    .then(validarPermissaoDeAcesso)
    .then(processarDados)
    .then(mostrarResultado)
    .catch(erro => {
        console.error("Ocorreu um erro:", erro);
    });
```

```
login(usuario, () => {
    obterDados(() => {
        validarDados( () => {
            validarPermissaoDeAcesso(() => {
                processarDados(() => {
                    mostrarResultado();
                });
            });
        });
    });
});
```

Tratamento de erros (try, catch, finally)

Erros acontecem sempre

Em aplicações reais:

- Dados podem falhar;
- APIs podem cair;
- Utilizadores podem introduzir dados inválidos;

O objetivo do try…catch não é evitar todos os erros, mas é saber lidar com eles, impedindo que a aplicação derrube a aplicação inteira.

O que acontece sem tratamento

```
console.log('Início da execução');
erroInexistente(); // Isso vai gerar um ReferenceError
console.log("Fim da execução");
```

```
ReferenceError: erroInexistente is not defined
    at Object.<anonymous> (C:\Users\rfcos\Proton
ascript\exemplos\exemplos-sessao-08\resolucao-dos
    at Module._compile (node:internal/modules/cjs
    at Object..js (node:internal/modules/cjs/loade
    at Module.load (node:internal/modules/cjs/loa
    at Module._load (node:internal/modules/cjs/lo
    at TracingChannel.traceSync (node:diagnostics
    at wrapModuleLoad (node:internal/modules/cjs/
    at Module.executeUserEntryPoint [as runMain]
    at node:internal/main/run_main_module:33:47
```

Estrutura – try e catch

Quando introduzimos um código dentro do bloco try catch, o erro será capturado sem que isso cause uma parada total do programa.

```
try {
    // Código que pode falhar
} catch (erro) {
    // tratamento do erro
}
```

Erro tratado com o try e catch

```
try {
    console.log('Início da execução');
    erroInexistente(); // Isso vai gerar um ReferenceError
} catch (erro) {
    console.error('Ocorreu um erro:', erro.message);
}

console.log('Continuação da execução após o erro tratado');
console.log("Fim da execução");
```

```
Início da execução
Ocorreu um erro: erroInexistente is not defined
Continuação da execução após o erro tratado
Fim da execução
```

Erros em Promises

```
login(usuario)
    .then(obterDados)
    .then(validarDados)
    .then(validarPermissaoDeAcesso)
    .then(processarDados)
    .then(mostrarResultado)
    .catch(erro => {
        console.error("Ocorreu um erro:", erro);
    });
```

O .catch(erro => {}) é equivalente ao catch do try e catch.

Síntese

- Execução síncrona vs Assíncrona: Mesmo com o JS sendo single thread;
- Event Loop: Entendimento do papel da call Stack, event loop e web APIs;
- Callbacks: Callback Hell;
- Promises: Introdução às promises;
- Tratamento de erros: Aplicação de try/catch/finally.

Conclusão

Innominatum CONSULTORIA

Programação em JavaScript

ServiceNow - Deloitte

Rodrigo Costa

IPCA POLITÉCNICO DO CÁVADO E DO AVE
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU
IAPMEI Parcerias para o Crescimento
PORTUGAL DIGITAL