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 Parceiros 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 2

Sessão 13

Objetivo geral

Capacitar os formandos na escrita estruturada de código assíncrono utilizando async/await, promovendo a transição de fluxos baseados em encadeamento de Promises para uma sintaxe mais legível e previsível. A sessão aprofunda o consumo de dados externos através de requisições HTTP simples, introduzindo o formato JSON como padrão de serialização e desserialização de dados. Ao integrar tratamento básico de erros em operações de rede, os formandos consolidam a capacidade de construir fluxos assíncronos robustos e semanticamente claros, preparando o terreno para integrações mais avançadas em sessões futuras.

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.

Innominatum CONSULTORIA

Async / Await

O problema que queremos resolver

- Evolução das Callbacks: As callbacks tradicionais sofrem do problema de encadeamento lateral, o famoso callback hell.
- Limitações das Promises: Embora as Promises tenham resolvido o caos das callbacks, o seu crescimento através de múltiplos .then() pode recriar o mesmo problema de legibilidade e fluxo complexo.
- A Necessidade: Garantir a Integridade do fluxo de dados sem que o código se torne visualmente confuso ou difícil de depurar (debug).
- A Solução: Foi desenvolvida uma nova abordagem sintática para lidar com o assincronismo de forma mais "natural" e sequencial: o bloco async/await.

```
buscarDados()
  .then(dados => processar(dados))
  .catch(erro => console.log(erro));
```

Surge o async / await

A sua Ideia central basei-se em 4 pilares:

- **Legibilidade Superior**: Oferece uma estrutura muito mais clara e fácil de seguir do que as callbacks ou o encadeamento de .then() das promises;
- **Fundações em Promises**: Não substitui as Promises; pelo contrário, estas funções são construídas sobre elas e devolvem sempre uma Promise;
- **Aparência Síncrona**: Permite escrever código assíncrono que se lê de cima para baixo, como se fosse código síncrono convencional;
- "**Syntactic Sugar**": É considerado um "açúcar sintático", ou seja, uma sintaxe mais amigável que facilita a escrita e a leitura, sem alterar a funcionalidade lógica de base.

O que faz o async?

- A palavra chave **async**, transforma uma função normal em uma função **assíncrona**, e toda função marcada com a palavra chave **async, sempre** retorna uma **Promise**, mesmo que você não escreva uma **Promise**. Então resumindo, o **async** faz três coisas:

1. Faz a função sempre retornar uma Promise;
2. Permite usar **await** dentro da função;
3. Transforma erros em **Promise.reject**.

O que faz o await?

O **await pausa a execução da função async** até que uma Promise seja resolvida (ou rejeitada). A função que recebe a palavra reservada **await**, sai da callStack para que ela seja resolvida, e quando for resolvida ou rejeitada, é jogada para o Callback Queue no aguardo da callStack ficar 100% liberada.

```
function esperar() {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve("Terminou");
        }, 1000)
    });
};

async function executar() {
    console.log("Antes");
    const resultado = await esperar();
    console.log(resultado);
    console.log("Depois");
};

executar();
```

Comparação direta

```
buscarDados()
.then(dados => console.log(dados))
.catch(erro => console.log(erro));
```

```
async function executar() {
    try {
        const dados = await buscarDados();
        console.log(dados);
    } catch (erro) {
        console.log(erro);
    }
}
```

Código que parece síncrono

Parece execução normal,

mas ele contínua

assíncrono, através da

assinatura async await

dentro da nossa função.

```
async function fluxo() {
  const dados = await buscarDados();
  const processados = await processar(dados);
  console.log(processados);
}
```

O que realmente acontece?

A assinatura await pausa a função, na sequência ela devolve o controle ao Event Loop, e retoma o controle quando a Promise resolve, a Call Stack não é bloqueada com essa ação.

```
async function fluxo() {
  const dados = await buscarDados();
  const processados = await processar(dados);
  console.log(processados);
}
```

Consumo de Dados Assíncronos (GET com fetch)

O que é fetch?

O **fetch** é uma interface moderna do JS para manipulação de pedidos e respostas HTTP:

- **Requisições HTTP**: É a ferramenta padrão para enviar ou receber dados através da web (ex: pedir uma lista de produtos a um servidor);
- Baseado em **Promises**: o fetch trabalha nativamente com Promises, o que o torna perfeito para usar com async/await;
- **Comunicação com Servidores**: Permite que a tua aplicação "fale" com bases de dados externas e APIs de terceiros.

É uma forma moderna de fazer pedidos HTTP no navegador.

Estrutura básica

O fetch então é uma função que recebe uma url de onde ele irá capturar os dados de um servidor externo, nessa etapa ainda não possuímos os dados, estamos na etapa de captura (GET).

```
fetch("https://api.exemplo.com/dados");
```

Consumindo uma API usando o .then()

```
fetch(link)
    .then(response => response.json())
    .then(dados => console.log(dados))
    .catch(erro => console.error(erro));

// 1. Realizamos a busca dos dados através do link de uma API;
// 2. Obtemos a resposta e a transformamos em JSON;
// 3. Com a resposta obtida no resolve do primeiro then da promise;
// inserimos ela na variável dados e estamos pronto para utilizá-la.
```

Reescrevendo com async/await

```javascript
async function obterDados() {
    try {
        const response = await fetch(link);
        dados = await  response.json();
        console.log(dados);
    } catch (erro) {
        console.log("Erro: ", erro);
    }
}
```

```javascript
// 1. Criamos uma função com a assinatura async
// 2. Inserimos o código que esperamos que funciona dentro
// do bloco try
// 3. Realizamos a busca dos dados utilizando a assinatura await
// que indica que esse bloco precisa ser tirado da call stack
// 4. Quando a busca dos dados é finalizada,
// salvamos os dados na const dados
// 5. Exibimos os dados capturados no console;
// 6. Criamos um bloco de tratamento de erro;
// 7. Ainda não realizamos um tratamento,
// Contudo o console captura os erros que possam vir da consulta da api.
```

Uma importante observação aqui, é que temos dois blocos não bloqueantes (Promises) indicados pelo await, que é na captura desse dado pela API e depois da leitura desse dado na resposta em JSON.

O que é um pedido GET?

O **get** é um método HTTP para **obter dados**, ele não altera informação no servidor, a sua função é apenas ler os dados obtidos.

1. O **fetch** envia o pedido;
2. O navegador delega para **Web API**;
3. **Event Loop** aguarda a resposta;
4. **Promise** resolve;
5. **Await** retoma a execução.

JSON (Serialização e Parsing

O que é JSON?

- **Significado:** JSON é o acrónimo para JavaScript Object Notation;
- **Troca de Dados**: É um formato de texto leve utilizado universalmente para trocar informações entre diferentes sistemas;
- **Padrão de Comunicação**: Atua como a "língua comum" na comunicação entre o front-end (o navegador) e o back-end (o servidor).

JSON é texto, não é um objeto JS

Exemplo JSON

```
const json = {
  "nome": "Ana",
  "idade": 25,
  "ativo": true
}
```

```
const objeto = {
    nome: "Ana",
    idade: 25,
    ativo: true
}
```

O JSON é uma **string estruturada** e tem uma sintaxe muito parecida com a de um objeto JS.

Objetos x JSON

Diferenças importantes:

Objeto JS	JSON
Pode ter funções	Não pode ter funções
Pode usar aspas simples	Apenas aspas duplas
É executável	É apenas texto

JSON é um formato de transporte.

JSON.stringify()

Seu objetivo é converter **objeto** em **JSON**.

```
const objeto = {
    nome: "Ana",
    idade: 25,
    ativo: true
}

const json = JSON.stringify(objeto);
console.log(json);
```

```
{"nome":"Ana","idade":25,"ativo":true}
```

JSON.parse()

Seu objetivo é converter **JSON** em **objeto**.

```
const json = '{"nome":"Ana","idade":25}';

const objeto = JSON.parse(json);

console.log(objeto);
```

```
{ nome: 'Ana', idade: 25 }
```

Fluxo completo com JSON

1. Servidor envia **JSON** (texto);
2. **fetch** recebe resposta;
3. **.json()** faz parsing;
4. Agora temos objeto JS manipulável.

```
const response = await fetch(url);
const dados = await response.json();
```

Innominatum CONSULTORIA

Tratamento de Erros em Requisições

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

A realidade das requisições

Quando fazemos um **fetch**, pode acontecer:

- A internet falhar;
- O servidor não responder;
- O endereço estar errado;
- O servidor devolver erro;

Em resumo, as requisições podem falhar.

O problema sem tratamento

Se houver um erro:

- Promise é rejeitada;
- Código quebra;
- Nenhuma mensagem amigável é retornada ao utilizador.

```
async function obterDados() {
    const response = await fetch("url-invalida");
    const dados = await response.json();
    console.log(dados);
}
obterDados();
```

```
node:internal/deps/undici/undici:16416
      Error.captureStackTrace(err);
            ^

TypeError: Failed to parse URL from url-invalida
    at node:internal/deps/undici/undici:16416:13
    at async obterDados (C:\Users\rfcos\Proton Dr
```

Usando o try/catch

O try/catch caputra:

- Os erros de rede;
- As promises rejeitadas;
- As Exceções dentro do bloco.

```
async function obterDados() {
    try {
        const response = await fetch("url");
        const dados = await response.json();
        console.log(dados);
    } catch(erro) {
        console.error("Erro na requisição: ", erro);
    }
}
obterDados();
```

```
Erro na requisição:  TypeError: Failed to parse URL from url
```

Verificando resposta HTTP

O **fetch** só rejeita em erro de rede, os erros HTTP(404 e 500) falaremos em outros módulos, precisam de tratamentos manuais.

```
async function obterDados() {
    try {
        const response = await fetch("url");
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }
        const dados = await response.json();
        console.log(dados);
    } catch(erro) {
        console.error("Erro na requisição: ", erro);
    }
}
obterDados();
```

Síntese

- **Async / await**: Como abstração das promises, permitindo escrita de códigos mais lineares;
- **Consumo de dados com fecth**: Entendendo que **fetch** retorna uma **promise** e a sua conversão da resposta em **JSON** também é uma operação assíncrona;
- **JSON**: Aplicação correta do **JSON.string**() e **JSON.parse()**;
- **Tratamento de erros**: Aplicação de try/catch/finally em conjunto com o **async/await** para capturar erros de rede e falhas na execução, reforçando a importância da resiliência da aplicação.

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 Parceiros para o Crescimento
PORTUGAL DIGITAL