código assíncrono

Foto por Aron Visuals | Unsplash

Antes de começar...

Interrompam e **perguntem bastante**.
Esse assunto costuma ser um pouco chato de absorver vendo pela primeira vez e pode ser que demore um tempo até todas se acostumarem com todos os tópicos.

Por isso, **nada de sair com dúvidas**, combinado?

Pra entender código **assíncrono**, vale a pena rever primeiro o código **síncrono**

```
function funcaoDemorada() {
    console.log('Iniciando funcao extensa');
    // ..
    // ..
    // ..
    console.log('Ainda sendo executada');
    // ..
    // ..
    // ..
}

console.log('Executando rotina XYZ');
funcaoDemorada();
console.log('Rotina XYZ finalizada')';
```

Vamos pensar: o que acontece ao executar esse código?

Por padrão, nosso código JavaScript é executado até sua conclusão (run to completion)

Mas nem sempre tudo que precisamos depende somente do processamento da nossa aplicação

Vocês **já trabalham** com alguns **códigos assíncronos**.
Conseguem lembrar de algum exemplo?

Vocês **já trabalham** com alguns **códigos assíncronos**.
Conseguem lembrar de algum exemplo?

```
button.addEventListener('click', function() {

});
```

Em que **momento** essa função de click é **executada**?

Código **assíncrono** é essencial quando nossa aplicação demanda algum dado que ainda não temos ou com ações que não podemos prever quando irão acontecer.

Podemos (por enquanto) resumir código assíncrono de 3 maneiras:

- Callbacks;
- Promises;
- Async/Await (Promises com tômpero).

CALLBACKS

Foto por Louis Hansel @shotsoflouis | Unsplash

Callbacks

São funções que são registradas como parâmetros que serão executadas futuramente

```
function funcaoComplexa(numero, meChameQuandoAcabar) {
    // ...
    // ...
    // ...
    // ...
    // ...
    // ...
    // ...
    const total = numero + 1;
    // ...
    return meChameQuandoAcabar(total)
}

function quandoAcabar(resultado) {
    console.log(`Finalizou! Resultado: ${resultado}`);
}

funcaoComplexa(10, quandoAcabar)
```

callback == 'chamar de volta'

Mas… Peraí...

```
button.addEventListener('click', function() {

});
```

Então a função executada no **click** é, além de **assíncrona**, um **callback**?


ISSO!

Callbacks (outros exemplos)

```
console.log('Log 1');
setTimeout(() => {
    console.log('Log 2');
}, 5000);
console.log('Log 3');
```

```
console.log('Log 1');
setTimeout(() => {
    console.log('Log 2');
}, 0);
console.log('Log 3');
```

PROMISES

Promises

São "promessas" de que uma função resultará em um valor futuro e possuem resultar em 2 cenários:

- Sucesso (resolvida);
- Falha (rejeitada).

promise == 'promessa'

Promises (resolved/rejected)

Podemos encadear execução de Promises executando o método **.then** e tratar os casos de falha usando o método **.catch**

```
// Não se apeguem à estrutura

const consultaDados = () => new Promise((resolve, reject) => {
    resolve('dados');
});


consultaDados()
.then(() => {
    // ...
});
.catch(() => {
    // ...
});
```

```
// Para criar uma Promise, basta executar
// new Promise();


// Sucesso
const promiseResolvida = new Promise((resolve, reject) => {
    // utilizar o resolve resultará resolverá a promise
    resolve('Resolveu!');
});

promiseResolvida
.then(() => {
    console.log('Primeiro then');
})
.catch((error) => {
    console.log('Caiu no catch!')
    console.log(error);
});


// Falha
const promiseRejeitada = new Promise((resolve, reject) => {
    // utilizar o reject rejeitará a promise
    reject('Ih, deu ruim!')
});

promiseRejeitada
.then(() => {
    console.log('Primeiro then');
})
.catch((error) => {
    console.log('Caiu no catch!')
    console.log(error);
});
```

Promises (resolved/rejected)

```
const promise = new Promise((resolve) => {
    resolve('resolveu')
});

console.log('Log 1');
promise.then(() => console.log('Log 2'));
console.log('Log 3');
```

Nem sempre criaremos Promises diretamente

```
promiseQualquer()
.then()
.catch();
```



```
firebase
.auth()
.signInWithEmailAndPassword(email, password)
.then()
.catch();
```

ASYNC / AWAIT

Foto por Andrea Leon | Unsplash

Async/Await

São promises por debaixo dos panos, mas com uma sintaxe mais agradável e de fácil leitura.

Utilizamos a palavra **async** na declaração da função e agora podemos usar a palavra **await** dentro da função ao executar uma Promise.

Entretanto, para lidar com erros, precisaremos agora de um **try/catch.**

```
// omiti o restante para focarmos apenas no try/catch

try {
    const personagens = await consultaPersonagens();
} catch (error) {
    console.log('Ops! Erro');
    console.log(error);
}
```

```
const consultaPersonagens = () => new Promise((resolve) => {
    const personagens = [
        {
            name: 'pikachu'
        },
        {
            name: 'charmander'
        },
        {
            name: 'bulbasaur'
        }
    ]
    resolve(personagens);
});

// declaramos a função com a palavra async
// poderia ser uma arrow function como
// const executa = async() => { ... }
async function executa() {
    /**
     * ao invés de utilizarmos
     * consultaPersonagens()
     * .then()
     *
     * podemos agora escrever nosso código usando await
     */

    const personagens = await consultaPersonagens();
    console.log(personagens);
}
```

DÚVIDAS

Foto por Camylla Battani | Unsplash

Issaê

Links úteis:

- Asynchronous JavaScript
- Testes assíncronos em JavaScript