
# CAFÉ, DESENVOLVIMENTO & JS

# JAVASCRIPT

## JavaScript (Promises e async/await)

Paulo Henrique Amigoni

www.amigoni.com.br
paulo.amigoni@gmail.com
linkedin.com/in/pauloamigoni/
github.com/pauloamigoni/

CONHECENDO

# JAVASCRIPT

## JAVASCRIPT ASSÍNCRONO: CALLBACKS (PROMISES E ASYNC/AWAIT)

Esses tempos atrás tive um colega que estava com dificuldade de intender  Promises e Async/Await no JS, então bora tentar explicar. Vamos falar sobre uma coisa que confunde muita gente: o async/await do JavaScript. É uma espécie de magia que tem a ver com promessas (não as de fim de relacionamento, infelizmente) e é muito útil em alguns casos. Mas o que diabos é código assíncrono ou não bloqueador? Parece algo saído de um livro de ficção científica.

Antes de entender como funciona o async/await, vamos dar uma olhada no passado. Era um tempo em que os programas tinham que fazer várias chamadas assíncronas interdependentes e usar retornos de chamada para compor um resultado final. Era uma dor de cabeça. Uma festa de caça ao tesouro, mas sem a diversão e com muito mais frustração.

Mas agora, graças ao async/await, essa dor de cabeça foi reduzida drasticamente. Haverá um ovo de páscoa no final para realmente se aprofundar um pouco mais.

E, é claro, é importante saber como usar isso corretamente. Mas não se preocupe, vamos explicar tudo de uma forma que até mesmo sua tia que não entende nada de tecnologia vai entender. Então, vamos começar essa jornada de aprendizado!

## O QUE SIGNIFICA ASSÍNCRONO / NÃO BLOQUEADOR?

Cara, o que significa essa parada de assíncrono / não bloqueador? Eu sei que é alguma coisa relacionada a programação, mas preciso entender melhor. Então, o que é isso exatamente? Bom, pelo que entendi, a programação assíncrona é uma forma de executar várias tarefas ao mesmo tempo, sem ficar esperando uma tarefa terminar antes de começar outra. Isso é útil para lidar com operações que podem demorar um tempo variável ou imprevisível, como fazer uma solicitação de rede ou acessar um arquivo. Dessa forma, a gente não bloqueia o programa principal e ele continua executando outras tarefas enquanto espera a conclusão dessas operações mais demoradas. É isso mesmo? Se for, faz todo o sentido.

## O QUE SIGNIFICA ASSÍNCRONO / NÃO BLOQUEADOR?

Bem, imagine uma daquelas cafeterias hipster onde todo mundo vai para tomar um café e um sanduíche, certo? Agora, imagine que você é o único barista naquela loja movimentada e está recebendo pedidos de um monte de clientes com pressa. E é aí que entra a programação assíncrona.

Ao invés de esperar que um café ou sanduíche seja preparado antes de atender o próximo cliente, você pode receber vários pedidos ao mesmo tempo e preparar tudo ao mesmo tempo, mantendo a eficiência. Enquanto o forno e a máquina de café estão fazendo sua mágica em segundo plano, você continua a pegar pedidos, preparando tudo de forma rápida e eficiente.

CONHECENDO

# JAVASCRIPT

Este conceito não se limita a uma simples cafeteria, mas é amplamente utilizado na programação, onde processos longos e demorados são executados em segundo plano, sem bloquear o fluxo principal de execução. É como se estivéssemos trabalhando em várias coisas ao mesmo tempo, sem perder a eficiência ou a qualidade. É uma maneira de garantir que todos os clientes sejam atendidos de forma rápida e eficiente, sem que ninguém precise esperar na fila por muito tempo.

Mas se você está preso em uma programação síncrona, é como se estivesse trabalhando naquela cafeteria sozinho, tendo que esperar que cada café e sanduíche sejam feitos antes de atender ao próximo cliente. Isso pode levar a atrasos e a uma fila cada vez maior de clientes impacientes.

Portanto, é importante entender a diferença entre esses dois conceitos de programação e usar a abordagem assíncrona sempre que possível. É como ter a ajuda de um exército de baristas invisíveis para preparar os pedidos em segundo plano, permitindo que você atenda a todos os clientes de forma rápida e eficiente. E quem não quer um café e um sanduíche mais rápido?

## COMO ISSO SE RELACIONA COM O JAVASCRIPT?

Pera aí, amigão, cê tá ligado que o JavaScript é uma rosca única, né? Isso significa que ele só pode fazer uma coisa de cada vez. Mas calma, que tem solução! E aí que entra a analogia do café, saca?

No café síncrono, o fluxo de execução principal, tipo o barista, só processa uma tarefa de cada vez. Ou seja, ele fica lá parado esperando a finalização de cada tarefa antes de passar para a próxima. E isso pode ser um problemão se você tá lidando com operações demoradas, porque o barista fica lá, plantado, enquanto aguarda o término dessas operações. E aí o aplicativo pode ficar lento, travando e deixando o usuário puto.

Mas relaxa, porque tem outra forma de programação: a assíncrona! No café assíncrono, o fluxo de execução principal continua processando outras tarefas enquanto espera a conclusão das operações demoradas. O loop de eventos é quem gerencia essa parada toda, monitorando a pilha de chamadas e a fila de tarefas. Quando a pilha tá vazia, ele descarta tarefas da fila e joga na pilha de chamadas para execução. E isso permite que o JavaScript lide com várias tarefas ao mesmo tempo, sem travar o fluxo principal. E isso melhora o desempenho geral e a capacidade de resposta do aplicativo.

E se você tá usando técnicas como retornos de chamada, Promises async/await, aí que a mágica acontece. Você pode criar aplicativos que lidam com eficiência com operações demoradas, enquanto o loop de eventos (o barista) continua processando outras tarefas. E aí a experiência do usuário fica top. Entendeu?

CONHECENDO

# JAVASCRIPT

## CALLBACKS, PROMISES, AND ASYNC/AWAIT

Imagine que você está em uma cafeteria tentando pedir um café. Você pede ao barista para preparar seu café e, em seguida, espera por ele. Mas, oh não! Você também quer um croissant. Então, você pede ao barista para preparar o café e o croissant. Agora, você está esperando pelo café e croissant. Mas, espere, você também precisa de um copo de água. Então, você pede ao barista para preparar o café, o croissant e o copo de água. E aí vem a surpresa, enquanto você está esperando pelo seu pedido, você percebe que não tem dinheiro suficiente para pagar. Então, você pede ao barista para cancelar tudo. O barista fica chateado, você fica chateado. Tudo é um desastre!

Promises ( tornando o código mais limpo e legível ):
Em vez de pedir cada item separadamente, imagine que você possa fazer um pedido de todos os itens ao mesmo tempo. O barista anota seu pedido e promete entregar tudo juntos, assim que estiver pronto. Agora, você pode esperar por tudo de uma vez e, se precisar cancelar, basta cancelar o pedido inteiro. Mais simples, mais limpo, mais fácil.

Async / Await ( tornando o código ainda mais fácil de ler e escrever ):
E se o barista pudesse fazer todo o pedido e entregar tudo em uma bandeja? Você só precisaria esperar a bandeja chegar e pegar o que precisar. Não há necessidade de pedir cada item separadamente ou gerenciar várias promessas. Tudo é feito de uma vez e em ordem, com um simples await na bandeja. O barista fica feliz, você fica feliz, todos ficam felizes. É o paraíso da cafeteria!

CONHECENDO

# JAVASCRIPT

1 - Fluxo baseado em retorno de chamada ( aumentando a complexidade de vários retornos de chamada ):

```
<script>
function brewCoffee(callback) {
  setTimeout(() => {
    callback(null, "Coffee");
  }, 1000);
}

function prepareSnack(callback) {
  setTimeout(() => {
    callback(null, "Snack");
  }, 500);
}

brewCoffee((error, coffee) => {
  if (error) {
    console.error("Error brewing coffee:", error);
  } else {
    console.log(coffee);
    prepareSnack((error, snack) => {
      if (error) {
        console.error("Error preparing snack:", error);
      } else {
        console.log(snack);
        // Retorno de chamada
      }
    });
  }
});
</script>
```

CONHECENDO

# JAVASCRIPT

2 - Versão aprimorada usando Promises ( e captura ):

```
<script>
function brewCoffee() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Coffee');
    }, 1000);
  });
}

function prepareSnack() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Snack');
    }, 500);
  });
}

brewCoffee()
  .then(coffee => {
    console.log(coffee);
    return prepareSnack();
  })
  .then(snack => {
    console.log(snack);
  })
  .catch(error => {
    console.error('Error:', error);
  });
</script>
```

CONHECENDO

# JAVASCRIPT


3 - Versão final usando fluxo async / await:

```
<script>
async function serveOrder() {
  try {
    const coffee = await brewCoffee();
    console.log(coffee);

    const snack = await prepareSnack();
    console.log(snack);
  } catch (error) {
    console.error('Error:', error);
  }
}

serveOrder();
</script>
```

Caramba, esses exemplos são a prova viva de como a evolução do código pode ser surpreendente e tornar nossa vida mais fácil!
É incrível como a transição do fluxo baseado em retorno de chamada para Promises e, finalmente, para a sintaxe async / await faz toda a diferença na legibilidade e gerenciamento do código.
É como se o código passasse de uma selva de chamadas aninhadas para um gramado verde e limpo.
Com o fluxo de assíncrona / await, parece que estamos programando de forma síncrona, mas aproveitando os benefícios da programação assíncrona.
É como ter o melhor de dois mundos! Agora, quem diria que o código poderia ser tão apaixonante, não é mesmo?

CONHECENDO

# JAVASCRIPT

## MAS O QUE ESTÁ ACONTECENDO AQUI?

E aí, pessoal! Quem mais está nessa busca por compreender as coisas em sua essência fundamental? Eu sei que eu estou, e aposto que você também, se chegou até aqui.

Mas vamos ao que interessa, porque para realmente entender essas promessas, precisamos quebrar a mágica toda. E a melhor maneira de fazer isso é construindo nossa própria versão de uma promessa, ou melhor, um padrão AsyncTask. E aí sim, vamos poder expor e pegar métodos, permitindo que sejamos acorrentados. Então, bora lá, sem medo de ser feliz!

```
class AsyncTask {
  constructor(executor) {
    this.callbacks = [];
    this.catchCallbacks = [];
    this.state = "pending";
    this.result = null;

    const resolve = (value) => {
      if (this.state === "pending") {
        this.state = "fulfilled";
        this.result = value;
        this.callbacks.forEach((callback) => callback(value));
      }
    };

    const reject = (reason) => {
      if (this.state === "pending") {
        this.state = "rejected";
        this.result = reason;
        this.catchCallbacks.forEach((callback) => callback(reason));
      }
    };

    try {
      executor(resolve, reject);
    } catch (error) {
      reject(error);
    }
  }
```

CONHECENDO

# JAVASCRIPT


```
  then(onFulfilled) {
    return new AsyncTask((resolve, reject) => {
      const wrappedOnFulfilled = (value) => {
        try {
          resolve(onFulfilled(value));
        } catch (error) {
          reject(error);
        }
      };

      if (this.state === "fulfilled") {
        wrappedOnFulfilled(this.result);
      } else if (this.state === "pending") {
        this.callbacks.push(wrappedOnFulfilled);
      }
    });
  }

  catch(onRejected) {
    if (this.state === "rejected") {
      onRejected(this.result);
    } else if (this.state === "pending") {
      this.catchCallbacks.push(onRejected);
    }
    return this;
  }
}
```

Esta é essencialmente a coragem de um padrão de promessa. É uma abstração sobre retornos de chamada. Obviamente, há uma tonelada de ferramentas e otimizações adicionais em torno dele, já que agora é nativo do idioma, mas é isso mesmo.

CONHECENDO

# JAVASCRIPT


É assim que o usamos com o exemplo anterior de cafeteria:

```
function brewCoffee(order) {
  return new AsyncTask((resolve, reject) => {
    console.log("Brewing coffee:", order);

    setTimeout(() => {
      console.log("Coffee ready:", order);
      resolve(order);
    }, Math.random() * 2000);
  });
}

function prepareSnack(order) {
  return new AsyncTask((resolve, reject) => {
    console.log("Preparing snack:", order);

    setTimeout(() => {
      console.log("Snack ready:", order);
      resolve(order);
    }, Math.random() * 3000);
  });
}

function serveItems(items) {
  console.log("Serving:", items.join(" and "));
  return "Served: " + items.join(" and ");
}
```

...

CONHECENDO

# JAVASCRIPT

...

```
function logServedStatus(status) {
  console.log(status);
}

const coffeeOrder = brewCoffee("Cappuccino");
const snackOrder = prepareSnack("Croissant");

coffeeOrder
  .then((coffee) ⇒
    snackOrder
      .then((snack) ⇒ [coffee, snack])
      .then(serveItems)
      .then(logServedStatus)
  )
  .catch((error) ⇒ {
    console.log("Error:", error);
  });

console.log("Taking the next customer...");
```

A parte que não podemos realmente simular aqui é o uso de palavras-chave async / await.
É importante entender que é apenas o açúcar sintático que permite fingir que o código é executado de forma síncrona.
Sob o capô, ainda está usando promessas.

CONHECENDO

# JAVASCRIPT

## CONCLUSÃO

Manja só, programação assíncrona é daquelas coisas que não só fazem parte do JavaScript moderno, mas também de outras linguagens de programação populares como Go, Java e C#. Ao usar técnicas assíncronas, tipo callbacks, Promises e a sintaxe async/await mais recente no JavaScript, os programadores conseguem criar aplicativos mais espertos, rápidos e que mandam bem em operações que comem um tempo absurdo.

Saca só, compreender essa programação assíncrona no JavaScript ajuda pacas a entender uns conceitos parecidos em outras linguagens, já que, geralmente, elas se ligam aos mesmos princípios e objetivos. Tipo, o Go saca as goroutines e canais para conseguir concorrência, o Java usa threads e a classe CompletableFuture, e o C# usa um padrão bem esperto, o async/await, com objetos Task.

O que é que acontece, é que essa evolução toda, sacou, dos fluxos baseados em callbacks para Promises e async/await deixou a legibilidade e manutenção do código JavaScript muito mais fácil, mano.
Isso é uma mão na roda, já que os programadores conseguem lidar de boa com operações complexas e dar ao usuário uma experiência muito melhor. E a analogia da cafeteria só prova isso, porque mostra como a programação assíncrona permite o processamento eficiente de tarefas de forma concorrente, o que leva a um aplicativo de melhor desempenho.

Tá ligado, o que é importante mesmo, é que, à medida que as linguagens de programação evoluem e ficam maduras, é essencial que os programadores se atualizem com as últimas técnicas e melhores práticas em programação assíncrona. É assim que eles vão conseguir dominar esses conceitos em JavaScript e entender como aplicar eles em outras linguagens, o que leva a aplicativos de alta qualidade e rápidos, que agradam o usuário moderno em todas as plataformas e tecnologias.