

Javascript

Programa de atração de formação de talentos
Bradesco / Visionaire

Aula 11 – Programação assíncrona
Parte 2 – Mais promises e async / await

Mark Joselli
mark.Joselli@pucpr.pr



Calendário

- ~~Aula 11 – Async 2~~
- Aula 12 – TDD – Trabalho 2
- Aula 13 – Expressões regulares
- Aula 14 – Acompanhamento trabalho 2

Promises

- Os Promises também possuem um conjunto de métodos estáticos, que permitem trabalhar com vários Promises ao mesmo tempo
- Isso poupa a necessidade de criação de variáveis externas na maioria dos casos



Promises: Métodos estáticos

- **all**: Recebe um array de promises e retorna um promise, que será resolvido quando todos os promises desse array estiverem resolvidos. O valor de entrada desse novo promise será um array, com os resultados dos promises anteriores, na mesma ordem que foram declarados. É rejeitado assim que o primeiro Promise do array for rejeitado.
- **allSettled**: Similar ao all, mas executa até que todos os promises tenham sido rejeitados ou resolvidos, retornando um array com os resultados.
- **any**: Recebe um array de promises e retorna um único promise que será resolvido quando o primeiro promise do array resolver.
- **race**: Recebe um array de promises e retorna um único promise que será resolvido quando o primeiro promise do array ou for rejeitado.
- **resolve**: Cria uma promessa resolvida com o valor passado
- **reject**: Cria uma promise rejeitada com o valor passado

Vamos agora resolver $a + b * c$ com Promises

Para isso precisamos:

1. Utilizar a função `emitirValor(valor)`
2. Utilizar o `Promise.all` para capturar todos os valores em paralelo
3. Realizar o cálculo

Vamos agora resolver $a + b * c$ com Promises

```
function emitirValor(valor, timeout = 1000) {  
    return new Promise(resolve => {  
        setTimeout(() => {  
            resolve(valor);  
        }, timeout);  
    });  
}
```

```
Promise.all([  
    emitirValor(5, 500),  
    emitirValor(10, 200),  
    emitirValor(2, 800)  
]).then(([a,b,c]) => console.log(a + b * c));
```

Fetch

- O javascript por padrão possui o comando **fetch**. Ele faz exatamente o que nossa função request faz.
- Como segundo parâmetro, ele aceita um objeto com as seguintes chaves (todas opcionais):
 - method: Verbo http usado na consulta (por padrão get)
 - headers: Um array de strings com cabeçalhos HTTP para serem incluídos na requisição
 - Consulte: <https://developer.mozilla.org/en-US/docs/Web/API/fetch> para ver todas as opções

async / await

- No ECMA 6, as palavras chave `async` e `await` foram criadas para simplificar drasticamente a sintaxe dos Promises
- Toda função `async` retornará automaticamente um Promise, que resolve com o que ela devolver no `return`, e rejeita suas exceções
- Dentro de uma função assíncrona podemos utilizar a palavra chave `await` no lugar do `then` para aguardar o valor da função
- Com `await` erros são capturados em blocos `try catch` comuns.



async / await

Isso permite escrevermos código assíncrono de maneira muito similar ao código síncrono. Vamos fazer isso?

```
emitirValor(50)
  .then(valor => {
    const num = parseFloat(valor);
    if (isNaN(num)) {
      throw `Não numérico: ${valor}`;
    }
    return valor * 50;
  })
  .then(produto => emitirValor(produto, 2000))
  .then(console.log)
  .catch(error => console.log(`ERRO: ${error}`));
```

async / await

Vamos iniciar colocando todo ele em uma função assíncrona

```
async function exemplo() {  
    emitirValor(50)  
        .then(valor => {  
            const num = parseFloat(valor);  
            if (isNaN(num)) {  
                throw `Não numérico: ${valor}`;  
            }  
            return valor * 50;  
        })  
        .then(produto => emitirValor(produto, 2000))  
        .then(console.log)  
        .catch(error => console.log(`ERRO: ${error}`));  
}
```

async / await

Agora, podemos substituir o **then** por **await**. Compare:

```
emitirValor(50)
  .then(valor => {
    const num = parseFloat(valor);
    if (isNaN(num)) {
      throw `Não numérico: ${valor}`;
    }
  })
```



```
const valor = await emitirValor(50);
const num = parseFloat(valor);
if (isNaN(num)) {
  throw `Não numérico: ${valor}`;
}
```

async / await

Agora, podemos substituir o `then` por `await`. E o `catch` por um `try catch`

```
async function exemplo() {  
  try {  
    const valor = await emitirValor(50);  
    const num = parseFloat(valor);  
    if (isNaN(num)) {  
      throw `Não numérico: ${valor}`;  
    }  
    const produto = await emitirValor(valor * 50, 2000);  
    console.log(produto);  
  } catch (error) {  
    console.log(`ERRO: ${error}`);  
  }  
}
```

async / await

Note que os retornos ficam mais naturais.

```
async function exemplo() {  
  try {  
    const valor = await emitirValor(50);  
    const num = parseFloat(valor);  
    if (isNaN(num)) {  
      throw `Não numérico: ${valor}`;  
    }  
    const produto = await emitirValor(valor * 50, 2000);  
    console.log(produto);  
  } catch (error) {  
    console.log(`ERRO: ${error}`);  
  }  
}
```

async / await

E fica mais fácil distinguir que trechos acionam partes assíncronas.

```
async function exemplo() {  
  try {  
    const valor = await emitirValor(50);  
    const num = parseFloat(valor);  
    if (isNaN(num)) {  
      throw `Não numérico: ${valor}`;  
    }  
    const produto = await emitirValor(valor * 50, 2000);  
    console.log(produto);  
  } catch (error) {  
    console.log(`ERRO: ${error}`);  
  }  
}
```


async / await

E o tratamento de erro ficou bastante natural

```
async function exemplo() {  
  try {  
    const valor = await emitirValor(50);  
    const num = parseFloat(valor);  
    if (isNaN(num)) {  
      throw `Não numérico: ${valor}`;  
    }  
    const produto = await emitirValor(valor * 50, 2000);  
    console.log(produto);  
  } catch (error) {  
    console.log(`ERRO: ${error}`);  
  }  
}
```

Dicas:

- O `await` só pode ser chamado dentro de funções síncronas.
- Mas lembre-se que fora delas, você poderá utilizar o Promise retornado por ela da maneira tradicional. Por exemplo:

```
exemplo()  
  .then()  
  .catch(e => console.error(e));
```

Dicas:

Lembre-se que o código do interior da função assíncrona retornará um Promise. Portanto, não faz sentido realizar o `await` de uma função assíncrona no `return` (ou mesmo na última linha de código):

```
async function calcularImposto(valor) {  
    const num = parseFloat(valor);  
    if (isNaN(num)) throw `Não numérico: ${valor}`;  
  
    const imposto = await obterImposto(valor);  
    return await enviar(valor * imposto);  
}
```

É melhor retornar esse Promise, como no exemplo.

Dicas:

Além disso, para chamar funções assíncronas em métodos como *Promise.all*, basta chama-las diretamente:

```
const [imp1, imp2, imp3] = Promise.all([
  calcularImposto(100),
  calcularImposto(200),
  calcularImposto(300)
]);
```

Atividades

1. Refaça o exercício 5 da aula passada e os exercícios anteriores utilizando `async` e `await`.
2. Usando `async` e `await` crie uma página que calcula o fibonacci de um número mostrando o cálculo passo a passo.
3. Usando programação assíncrona, crie uma webpage que realize um timer ou cronometro.
4. Usando programação assíncrona, crie uma webpage que realize um jogo de adivinhe o número, onde o usuário entra com um número, e ele verifique se o número está certo ou não após 3 segundos. Depois disso, após 5 segundos ele deixa o usuário tentar novamente.

Dúvidas?

- Materiais:
 - Livro Eloquent Javascript:
 - 3ª edição (inglês):
<https://eloquentjavascript.net/>
 - 2ª edição (português):
<https://github.com/braziljs/eloquente-javascript>
 - The Modern Javascript Tutorial:
<https://javascript.info/>