



**Fundamentos em Desenvolvimento Front End**

**Capítulo 5. Programação Assíncrona com JavaScript**

**Prof. Raphael Gomide**



## **Aula 5.1. Introdução**

## □ Introdução:

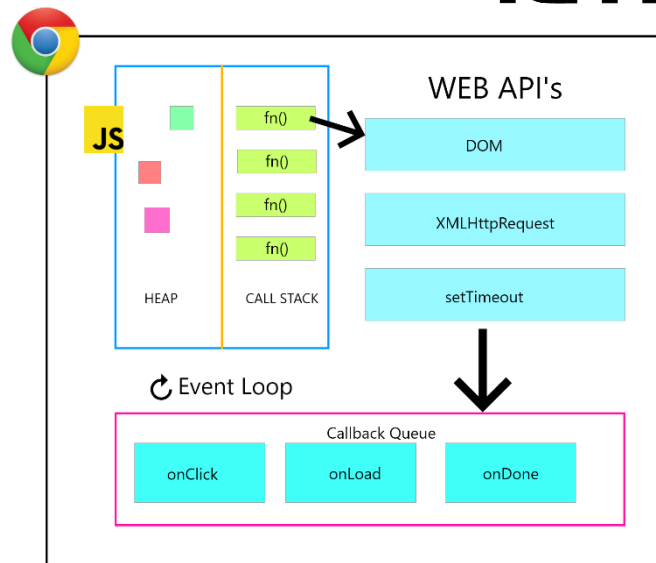
- Operações lentas.
- Event loop.

- Em JavaScript existem operações que podem ser lentas, como por exemplo:
  - Requisição de dados à APIs.
  - Processamento intenso de dados.
  - Comunicação com bancos de dados (Node.js).
- É extremamente importante que o JavaScript **não espere o término de instruções lentas.**
- A principal técnica para garantir a afirmação acima é a utilização do **event loop**.

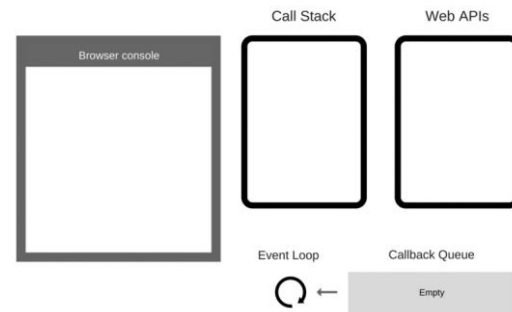
# JavaScript – Event loop

- Funções a serem executadas ficam em uma pilha lógica de invocações (call stack).
- Quando a função utiliza Web APIs, ela precisa passar pelo event loop, pois está sujeita à lentidões.
- O event loop executa uma função por vez e faz a orquestração que permite execução assíncrona.
- Em geral, funções que usam WEB APIs possuem **callbacks** (funções passadas por parâmetro).
- Mais informações [aqui](#).

```
1 console.log('Hi');
2 setTimeout(function cb1() {
3   console.log('cb1');
4 }, 5000);
5 console.log('Bye');
```



1 / 16



- ☑ O JavaScript é **síncrono** por padrão.
- ☑ Entretanto, possui o **event loop** para ter comportamento **assíncrono**.
- ☑ Funções que utilizam Web APIs utilizam o **event loop**.
- ☑ Em geral, após a execução no event loop, a função executa uma nova função de completude, denominada **callback**.
- ☑ **Callbacks** são funções passadas por parâmetros em funções que irão utilizar a Web API.

- ❑ As funções `setTimeout` e `setInterval`.



## **Aula 5.2. As funções setTimeout e setInterval**



- ☐ setTimeout.

- ☐ setInterval.

- **`setTimeout`:**

- Utilizada para postergar a execução de uma função.
- Tempo de atraso configurável em milissegundos.

- **`setInterval`:**

- Semelhante ao **`setTimeout`**, mas repete a execução a cada x milissegundos.
- Pode ser cancelada com **`clearInterval`**.
- Para isso, devemos guardar a referência em uma variável.

- Acompanhe o professor.

- ✓ A função **setTimeout** é utilizada para postergar execuções.
- ✓ A função **setInterval** posterga e repete as execuções a cada x milissegundos.
- ✓ Em geral, é importante parar execuções de **setInterval**. Isso é feito com **clearInterval**.

- ❑ Requisições HTTP com JavaScript.



## **Aula 5.3. Requisições HTTP com JavaScript**

- ☐ O comando fetch.
- ☐ Promises.
- ☐ Async/await.

- Utilizado para requisições HTTP.
- Trabalha internamente com promises.
- O primeiro retorno de fetch são dados binários.
- Em geral, convertemos esses dados para JSON, que retorna outra promise.
- Acompanhe o professor.

# Promises

- Promises são construções cuja execução **retorna algo no futuro**, ou seja, é uma **promessa de execução**.
- A execução pode ser **resolvida (ok)**, ou **rejeitada (erro)**.
- A promise resolvida é interceptada com **then**.
- A promise rejeitada é interceptada com **catch**.
- Resolve parcialmente o problema do ***callback hell***.



```
1 function hell(win) {
2   // for listener purpose
3   return function() {
4     loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
5       loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
6         loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
7           loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
8             loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {
9               loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
10                loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
11                  loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
12                    loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
13                      async.eachSeries(SCRIPTS, function(src, callback) {
14                        loadScript(win, BASE_URL+src, callback);
15                      });
16                    });
17                  });
18                });
19              });
20            });
21          });
22        });
23      });
24    });
25  });
26 }
```



# Async/await

- Açúcar sintático (syntax sugar) sobre promises.
- Melhoram a **legibilidade** do código.
- Dá a impressão de código síncrono.
- Deve-se decorar a função com **async**.
- Toda instrução relacionada à promise deve ser precedida de **await**.
- Acompanhe o professor.

## #CALLBACK

```
GetUser(function(err, user){
  GetProfile(user, function(err, profile){
    GetAccount(profile, function(err, acc){
      GetReport(acc, function(err, report){
        SendStatistics(report, function(e){
          ...
        });
      });
    });
  });
});
```

## #PROMISE

```
GetUser()
  .then(GetProfile)
  .then(GetAccount)
  .then(GetReport)
  .then(SendStatistics)
  .then(function (success) {
    console.log(success)
  })
  .catch(function (e) {
    console.error(e)
  })
```

## #ASYNC/AWAIT

```
async function SendAsync() {
  let user = await GetUser(1)
  let profile = await GetProfile(user);
  let account = await GetAccount(profile);
  let report = await GetReport(account);

  let send = SendStatistic(report);

  console.log(send)
}
```



- ☑ O comando **fetch** é utilizado para requisições HTTP.
- ☑ Trabalha internamente com **promises**.
- ☑ O mais comum é que os dados sejam retornados no formato JSON.
- ☑ Promises.
  - Muito utilizadas no JavaScript assíncrono.
  - Resolvem parcialmente o problema do callback hell.
- ☑ Async/await:
  - Açúcar sintático de **promises**.
  - Melhora a legibilidade do código.

# Próxima aula

☐ Desafios finais.



## **Aula 5.4. Desafios finais**

- ☐ Descrição do desafio 2.
- ☐ Descrição do desafio 3.

- Construir código JavaScript para simular a ocorrência de incidentes e requisições, com as seguintes regras:
  1. Requisições são solicitações comuns.
  2. Incidentes indicam problema grave.
  3. Se o percentual de incidentes superar 30% do total, um alerta deve ser emitido de alguma forma. Esses 30% devem ser parametrizáveis.
  4. As requisições e incidentes serão simuladas a cada meio segundo, incrementando valores entre 1 e 5 aleatoriamente.

- Construir código JavaScript para simular a ocorrência de incidentes e requisições, com as seguintes regras:
  5. Uma barra horizontal deve medir a proporção entre incidentes e requisições.
  6. Devem haver botões para iniciar e parar a simulação. Os botões devem melhorar a experiência do usuário conforme vídeo.
  7. Deve haver um indicativo de quantas requisições e quantos incidentes existem.
  8. A simulação deve ser iniciada por um botão. O usuário pode preencher os valores manualmente também.

- Construir código JavaScript para simular a ocorrência de incidentes e requisições, com as seguintes regras:
  9. Utilize `setInterval` para a simulação. Lembre-se de guardar a simulação em uma variável para que possa pará-la com `clearInterval`.
  10. Os valores devem ser simulados utilizando `Math.max` e `Math.random`.
  11. O HTML e o CSS serão fornecidos pelo professor.
  12. O HTML e CSS utilizam o [Materialize](#) para melhorar o visual.



## Desafio 2: execução

### Desafio 01

Requisições:

100

Incidentes

1

Percentual de alerta

30

SIMULAR

PARAR SIMULAÇÃO

### Proporção

Total de requisições: 100 (99%)

Total de incidentes: 1 (1%)

## Desafio 3

- Construir código JavaScript para listar países e marcá-los como favoritos:
  1. Busque os dados em <https://restcountries.eu/rest/v2/all> com fetch.
  2. Transforme os dados de forma que somente os seguintes itens sejam disponíveis:
    - Nome.
    - Bandeira.
    - População.
    - Id (numericCode).
  3. Mostre a quantidade de países e o total de população na lista da esquerda e da direita.

- Construir código JavaScript para listar países e marcá-los como favorito:
  4. Ao adicionar aos favoritos, o país deve ser movido da esquerda para a direita.
  5. Ao remover dos favoritos, o país deve ser reinserido na lista da esquerda.
  6. As listas de países devem ser **sempre** exibidas em **ordem alfabética**.
  7. Um **desafio extra**: formatar os números com JavaScript puro (pesquise por **Intl**).



















## Desafio 3

- Desafio proposto em execução:

## Desafio 02

Países (248)

População total: 7.321.402.072

		Åland Islands 28.875
		Albania 2.886.026
		Algeria 40.400.000
		American Samoa 57.100
		Angola 25.868.000
		Anguilla 13.452
		Antarctica 1.000
		Antigua and Barbuda 86.295
		Argentina 43.590.400

Países (2)

População total: 27.735.159

		Afghanistan 27.657.145
		Andorra 78.014

- Acompanhe o professor.

☑ Descrição do desafio 2.

☑ Descrição do desafio 3.