Drops de conteúdo: Array; JSON; Funções Assíncronas; Funções com Parâmetro e com retorno; sessionStorage.

Disclaimer / Aviso legal

Antes de ler este material importante saber que:

1. São assuntos que, se ainda não viram, serão tratados na aula de Algoritmos
2. Como o Projeto de PI engloba todas as disciplinas, vamos abordar destes assuntos só o suficiente para que seja possível:
    a. **Identificar** e **Replicar**;
    b. O intuito deste material não é ensinar a **Implementar** "do zero" em algum projeto.

Para seguir o passo a passo neste material:

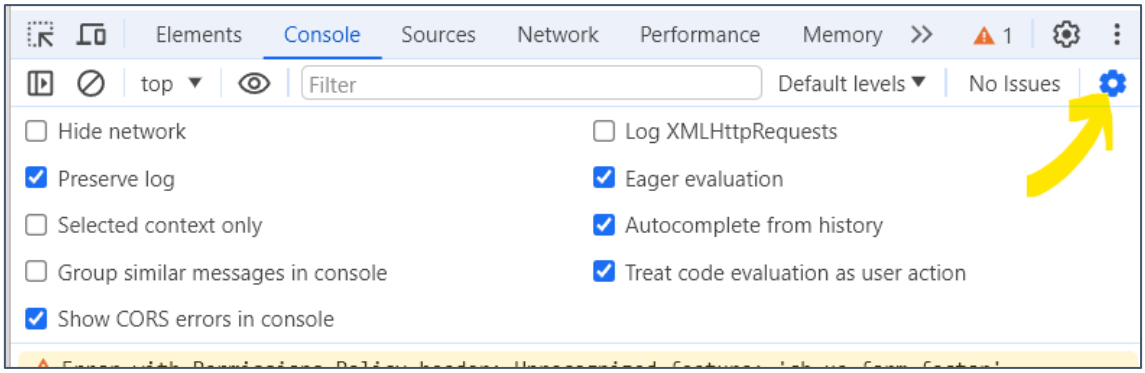
- Utilize o console do navegador Google Chrome, em *DevTools*. Você pode:
    - Abrir o navegador e pressionar `CTRL+Shift+I`
    - Clicar com o botão direito do mouse em sua janela e escolhendo a opção "Inspecionar".
- Certifique-se de que o log está sendo preservado:

- Certifique-se de que o console não está agrupando mensagens similares.

Estas configurações estão disponíveis em:



Array

Definição: Array ou Vetores são estruturas utilizadas para armazenar uma lista de dados.

Como identificar: utilizamos [ ]

Como declaramos: em uma variável, em vez de armazenar um valor único, podemos armazenar vários valores separados por vírgula, dentro de colchetes [ ].

Implementação de Referência: lista de compras.

```
var listaDeCompras = ["pão", "leite", "farinha", "ovos"];
```

Como acessar o valor:

```
nomeDaVariavel[posicaoIniciandoNoZero]
```

Ex.: Se quisermos trazer o 'pão':

```
listaDeCompras[0] // este código retorna 'pão'
```

Ex.: Se quisermos trazer o 'leite':

```
listaDeCompras[1] // este código retorna 'leite'
```

Ex.: Se quisermos saber quantos elementos há neste vetor:

```
listaDeCompras.length // este código retorna 4
```

Ex.: Se quisermos trazer o último elemento:

```
listaDeCompras[listaDeCompras.length - 1] // retorna 'ovos'
```

JSON

Definição: JSON ou Java Script Object Notation são estruturas utilizadas para armazenar mais do que um valor em uma mesma variável, representando por exemplo objetos do mundo real, mundo não-virtual.

Como identificar: utilizamos { }

Como declaramos: em uma variável, em vez de armazenar um valor único, podemos armazenar vários valores, dentro de chaves { } e cada um com seu próprio "identificador", em uma estrutura comumente chamada de "chave-valor".

Implementação de Referência: objeto livro.

Em um livro, temos vários atributos, por exemplo o autor, o título, o ano... cada um destes atributos é uma "chave" e cada chave tem um "valor".

```
var meuLivro = {
      'autor': 'J.K. Rowling',
      'titulo': 'Harry Potter e a Pedra Filosofal',
      'ano': 1998
};
```

Como acessar o valor:

```
nomeDaVariavel.chave
```

Ex.: Se quisermos trazer o 'ano':

```
meuLivro.ano // este código retorna 1998
```

Ex.: Se quisermos trazer o 'autor':

```
meuLivro.autor // este código retorna 'J.K. Rowling'
```

Extra: Podemos ter uma lista de livros, usando array/vetores com JSON. Fica assim:

```
var meusLivrosFavoritos = [
      {
            'autor': 'Liev Tolstói',
            'titulo': 'Guerra e Paz',
            'ano': 1869
      },
      {
            'autor': 'Jane Austen',
            'titulo': 'Orgulho e Preconceito',
            'ano': 1813
      }
];
```

Ex.: Se quisermos acessar o primeiro livro da lista:

```
meusLivrosFavoritos[0] // este código retorna o objeto completo
```

Ex.: Se quisermos acessar o título do primeiro livro da lista:

```
meusLivrosFavoritos[0].titulo // retorna 'Guerra e Paz'
```

setTimeout / setInterval

Introdução: Na vida "real" podemos fazer tudo de maneira "assíncrona", ou seja, podemos iniciar mais de uma tarefa e executá-las simultaneamente, sem que uma interfira na outra. Exemplo: quando fazemos macarrão, podemos colocar a água para ferver e não precisamos aguardar ferver para fazer o molho; podemos fazer o molho enquanto a água ferve e só então voltar à panela para colocar o macarrão. Quando programamos, também podemos configurar para que as funções aconteçam em simultâneo, sem que elas "travem" umas às outras.

Quando invocamos uma função, ela é executada quase que instantaneamente. Podemos fazer com que uma função seja invocada assim que outra coisa acontecer, como vocês faz com o macarrão, por exemplo (você espera estar cozido e em seu prato para então comê-lo)... ou até mesmo determinar um intervalo de tempo para a execução da função!

Como identificar: usamos as palavras `setTimeout` se quisermos que uma função seja invocada uma vez daqui um determinado tempo e `setInterval` se quisermos que uma função seja invocada regularmente, com um intervalo de tempo, e de maneira "infinita", até que algo a interrompa.

Como declaramos:

```
setTimeout(funcaoDesejada, tempoEmMilissegundos)

setInterval(funcaoDesejada, intervaloDeTempoEmMilissegundos)
```

Implementação de Referência:

setTimeout

A função abaixo, ao ser executa em meu console, será executada quase que instantaneamente.

```
console.log('oi, agora mesmo')
```

Se eu quiser que a função seja executada daqui 5 segundos, posso fazer desta maneira:

```
setTimeout(() => console.log('oi, mas daqui a pouco'), 5000)
```

Importante: `() =>` é uma das maneiras usadas para declarar uma função sem dar nome e se chama arrow function. *Curiosidade: há 7 maneiras de declarar funções em JavaScript.*

setInterval

Caso eu queira que uma função seja executada de maneira intervalada e até que eu a interrompa, faço assim:

```
var dizerOla = setInterval(() => console.log('OLÁ!'), 2000);
var fazerPsiu = setInterval(() => console.log('psiu'), 5000);
clearInterval(dizerOla) // este código interrompe as repetições
clearInterval(fazerPsiu) // este código interrompe as repetições
```

Funções com parâmetros e com retorno

Introdução: Quando declaramos funções, pode ser interessante que reutilizemos a mesma estrutura caso tenhamos algumas variações para estas.

Implementação de Referência:

```
function somarDoisMaisDois() {
      return 2+2
}

function somarDoisMaisTres() {
      return 2+3
}
```

Fazer somas assim é insustentável, precisaríamos de milhares de funções fazendo praticamente a mesma coisa, que é somar valores. Faremos então com que a função fique mais reutilizável, usando **parâmetros** entre parênteses ao lado do nome da função e usando estas variáveis no corpo da função.

```
function somar(a, b) {
      return a + b
}
```

Assim, podemos somar 2 e 2, 2 e 3, 10 e 1000, 50 e 20... Execute alguns exemplos:

```
somar(2,2) // este código retorna 4

somar(2,3) // este código retorna 5

somar(10, 50) // este código retorna 60
```

As funções acima contém o resultado de soma ao lado da palavra **return**. Utilizamos para "devolver para quem pediu". Se uma função invocou esta função, o resultado será "devolvido" para a função que invocou. Usamos esta estrutura com frequência quando queremos que uma função "chame" a outra. Como por exemplo:

```
function informarResultadoDeSoma(nome, a, b) {
      console.log (
            `Olá, ${nome}, seu resultado de soma é ${somar(a,b)}`
      )
};
```

Ao invocar a função desta maneira:

```
informarResultadoDeSoma("Maria", 1,2)
```

O resultado será:

```
Olá, Maria, seu resultado de soma é 3
```

sessionStorage

Introdução: Quando estamos navegando nas páginas da internet, é comum que precisemos usar na próxima página uma informação que tínhamos na página anterior. Há várias maneiras de fazer isso e uma delas é usando algumas variáveis que o navegador nos permite manipular. **Como identificar em nosso projeto web-data-viz**: Execute o projeto Aquatech, efetue o login. Repare: a informação de nome de usuário, que não foi inserida no momento do login, aparece em tela! Abra as ferramentas de desenvolvedor no navegador, vá até a aba Application e veja os valores salvos em `sessionStorage`. As informações nesta aba seguem a mesma estrutura de "chave-valor", como vimos em JSON.

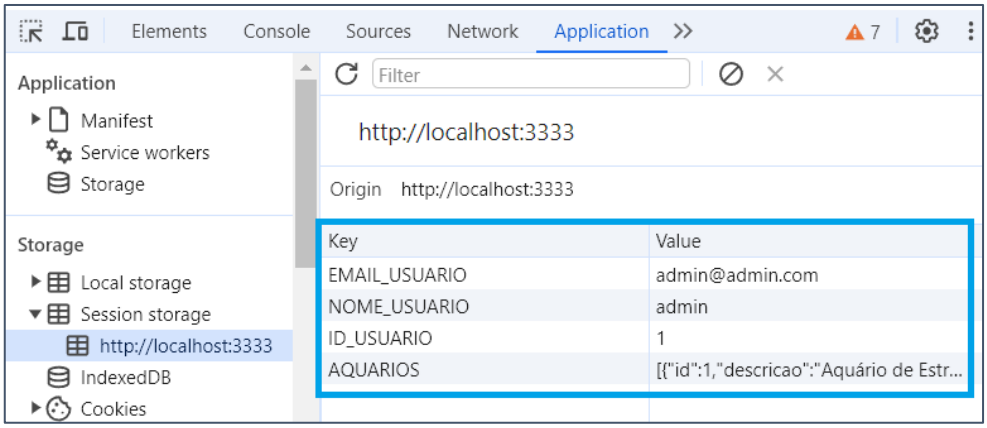


Figura 1 - Screenshot de janela de Ferramentas de Desenvolvedor no navegador Google Chrome

Acesse o código do web-data-viz e veja que em login.html armazena-se os valores (recebidos em login) nas variáveis do `sessionStorage`:

```
login.html
public > login.html > script > entrar > then() callback > then() callback
87   <script>
90       function entrar() {
118          }).then(function (resposta) {
124                  resposta.json().then(json => {
125                      console.log(json);
126                      console.log(JSON.stringify(json));
127                      sessionStorage.EMAIL_USUARIO = json.email;
128                      sessionStorage.NOME_USUARIO = json.nome;
129                      sessionStorage.ID_USUARIO = json.id;
130                      sessionStorage.AQUARIOS = JSON.stringify(json.aquarios)
```

Veja também que, em dashboard.html, fazemos a manipulação de HTML usando uma variável presente em `sessionStorage`:

```
login.html   dashboard.html
public > dashboard > dashboard.html > script > exibirAquariosDoUsuario > aquarios.forEach() callback
81
82   <script>
83       b_usuario.innerHTML = sessionStorage.NOME_USUARIO;
84
85       let proximaAtualizacao;
```

Definição: `sessionStorage` é um cookie que existe apenas na sessão da aba (note que, se trocarmos de aba, não teremos o dado anterior). É como uma variável global é armazenada no seu navegador. **Por que usar no projeto**? Use `sessionStorage` para armazenar as informações do usuário que foram trazidas assim que efetuar o login para que não precise ficar buscando no banco (por exemplo, permissão de usuário, ou a música favorita deste usuário que acabou de *logar*).