CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CAMPUS VII - UNIDADE TIMÓTEO
Curso: Engenharia de Computação
Disciplina: Desenvolvimento de Sistemas
Professor: Odilon Corrêa da Silva

GUIA 07 – JavaScript: jQuery e Funções assíncronas

Objetivos

- Utilizar os recursos do jQuery e seus plugins.
- Compreender e implementar funções assíncronas.

JavaScript: jQuery

Para aprimorar a interação dos usuários com páginas web, é fundamental empregar recursos avançados de CSS e HTML aliados ao JavaScript. Contudo, devido à variedade de navegadores disponíveis, o comportamento do código pode apresentar inconsistências, já que nem todos seguem exatamente os mesmos padrões. Para reduzir esses problemas e facilitar o desenvolvimento, é comum o uso do jQuery, uma das bibliotecas JavaScript mais populares do mercado (Figura 01).

Figura 01: JavaScript X jQuery

O jQuery foi criado com o objetivo de "escrever menos e fazer mais" (*write less, do more*). Essa filosofia reflete o foco da biblioteca em simplificar tarefas comuns e reduzir a quantidade de código. Por exemplo:

JavaScript

```
var altura = parseFloat(document.getElementById("altura").value);
var peso = parseFloat(document.getElementById("peso").value);

var imc = peso / (altura * altura);

document.getElementById("imc").value = imc.toFixed(2);
```

jQuery

```
var altura = parseFloat($("#altura").val());
var peso = parseFloat($("#peso").val());

var imc = peso / (altura * altura);

$("#imc").val(imc.toFixed(2));
```

Uma das principais vantagens do jQuery é sua extensibilidade, permitindo a criação e o uso de plugins. Esses plugins otimizam o desenvolvimento e oferecem soluções prontas para diversas funcionalidades, tais como:

- Validação de dados em formulários
- Criar máscaras em campos de entrada
- Organização de dados em tabela
- Criação de galerias de imagens
- Edição de elementos com um clique
- Desenvolvimento de menus interativos
- Implementação de editores de código HTML
- Upload de múltiplos arquivos
- Exibição de mensagens de confirmação

A validação de formulários é um aspecto essencial no desenvolvimento de aplicações web, pois garante que os dados enviados estejam no formato esperado. Uma maneira eficiente de implementar esse recurso é utilizando o jQuery Validation Plugin, que oferece uma abordagem prática e flexível para validar formulários. Documentação oficial: https://jqueryvalidation.org

Ainda não aplicamos máscaras nos campos do formulário. Para adicionar esse recurso de maneira simples, podemos utilizar o jQuery Mask Plugin, que permite formatar os dados inseridos pelo usuário (como CPF, telefone, data etc.). Documentação oficial: https://plugins.jquery.com/mask

O trecho de código a seguir demonstra o uso dos plugins jQuery Validation e jQuery Mask para definir regras de validação e formatação dos campos do formulário do Exemplo 01 (Figura 02).

jQuery Validation

```
$("#formulario").validate({
    rules: {
        cpf: {
            required: true
        },
        nome: {
            required: true
        },
        curso: {
            required: true
        },
        nota: {
            required: true,
            min: 0,
            max: 100
        }
    },
    messages: {
        cpf: {
            required: "Campo obrigatório"
        },
        nome: {
            required: "Campo obrigatório"
        },
        curso: {
            required: "Campo obrigatório"
        },
        nota: {
            required: "Campo obrigatório",
            min: "Valor mínimo é 0",
            max: "Valor máximo é 100"
        }
    }
});
```

jQuery Mask

```
$(document).ready(function(){
    $('#nota').mask('000');
    $('#cpf').mask('000.000.000-00');
});
```

Arquivo: Trecho do código JS do Exemplo 01

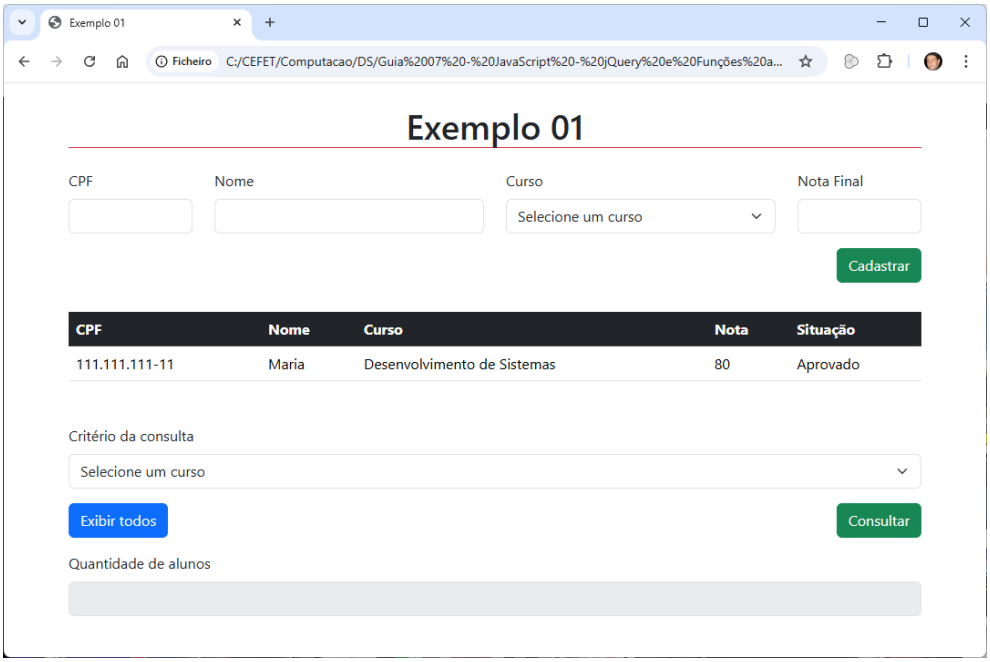

Figura 02: Visualizar do Exemplo 01

JavaScript: Funções assíncronas

Em JavaScript, a programação assíncrona é um importante recurso da linguagem. Trata-se de um mecanismo que permite a execução de operações em segundo plano, sem bloquear a thread principal. Esse recurso é especialmente importante no navegador, onde a thread principal é responsável por atualizar a interface do usuário e responder às ações do usuário.

A palavra-chave `async` transforma uma função comum em uma função assíncrona, que retorna automaticamente uma `Promise`. Dentro dessa função, podemos usar `await` para aguardar a resolução de uma `Promise` antes de continuar a execução do código. Isso torna o fluxo mais legível e semelhante ao de código síncrono.

- `async`: declara uma função assíncrona. **Importante**: o uso de `async` por si só não torna o código assíncrono.
- `await`: pausa a execução da função assíncrona até que a `Promise` seja resolvida (ou rejeitada).

Antes de trabalhar diretamente com `async`/`await`, é importante conhecer duas funções assíncronas que fazem parte da Web API do navegador (ou do ambiente Node.js):

- `setTimeout()`: Executa uma função apenas uma vez após um tempo especificado (em milissegundos).
  - Por exemplo:

```
setTimeout(() => {
    console.log("Executado após 2 segundos");
}, 2000);
```

- `setInterval()`: Executa uma função repetidamente em intervalos de tempo fixos.
  - Por exemplo

```
setInterval(() => {
    console.log("Executado a cada 2 segundos");
}, 2000);
```

Ambas as funções retornam um **ID** numérico, que pode ser usado para cancelar sua execução:

```
// Cancelar setTimeout
var timeoutId = setTimeout(() => console.log("Não será executado"), 3000);
clearTimeout(timeoutId);

// Cancelar setInterval após 5 segundos
var intervalId = setInterval(() => console.log("Será interrompido"), 1000);
setTimeout(() => clearInterval(intervalId), 5000);
```

Tanto `setTimeout` quanto `setInterval` são exemplos de operações assíncronas que interagem com a fila de tarefas (*task queue*) e o *event loop* do JavaScript. Onde:

I. A função é agendada para execução após o tempo especificado.
II. O JavaScript continua executando o restante do código normalmente.
III. Quando o tempo se esgota, a função agendada é colocada na fila de tarefas.
IV. O *event loop* executa essa função quando a pilha de execução estiver livre.

Por exemplo:

- Código JS

```
console.log("Início");

setTimeout(() => {
  console.log("Executado depois de 2 segundos");
}, 2000);

console.log("Fim");
```

- Saída esperada

```
Início
Fim
Executado depois de 2 segundos
```

Essa abordagem assíncrona é essencial para criar aplicações web eficientes e dinâmicas. Agora vamos explorar como trabalhar com `Promises`, `fetch`, e como aplicar `async`/`await` de forma prática.

Uma `Promise` (promessa) representa o resultado futuro de uma operação assíncrona que pode ser resolvida (com sucesso) ou rejeitada (em caso de erro). Ela possui dois métodos principais:

- `resolve(valor)`: finaliza a `Promise` com sucesso e entrega um valor.
- `reject(erro)`: finaliza a `Promise` com erro, permitindo o tratamento posterior.

Por exemplo:

```
function esperar(tempo) {
    return new Promise((resolve, reject) => {
        if (tempo < 0) {
            reject("Tempo inválido! O valor deve ser positivo.");
        } else {
            setTimeout(() => {
                resolve("Finalizado após " + tempo +"ms");
            }, tempo);
        }
    });
}

async function testarEspera() {
    try {
        console.log("Início da espera...");

        // Teste com tempo válido
        var resultado = await esperar(2000);
        console.log("Sucesso:", resultado);

        // Teste com tempo inválido (gera erro)
        var erroResultado = await esperar(-1000);
        console.log("Isso não será executado:", erroResultado);

    } catch (erro) {
        console.error("Erro capturado:", erro);
    }
}
```

Saída esperada:

```
Início da espera...
Sucesso: Finalizado após 2000ms
Erro capturado: Tempo inválido! O valor deve ser positivo.
```

O `fetch` é uma função nativa do JavaScript usada para fazer requisições HTTP (como GET, POST, PUT etc.). Ela permite buscar dados de APIs ou servidores e retorna uma `Promise`, o que possibilita o uso de `.then()` ou `async`/`await` para lidar com a resposta. É recomendável também usar `.catch()` para tratar possíveis erros durante a requisição.

Por exemplo:

```
function buscarEndereco(cep) {
    fetch("https://viacep.com.br/ws/" + cep + "/json/")
        .then(endereco => {
            console.log('Endereço recebido:', endereco);
        })
        .catch(erro => {
            console.error('Erro na requisição:', erro);
        });
}
```

ou

```
async function buscarEndereco(cep) {
    try {
        var resposta = await fetch("https://viacep.com.br/ws/" + cep + "/json/");
        var endereco = await resposta.json();
        console.log('Endereço recebido:', endereco);
    } catch (erro) {
        console.error('Erro na requisição:', erro);
    }
}


buscarEndereco('01001000');
```

Exemplo 02

O botão **Executar Síncrono** bloqueia a interface até o fim da contagem, enquanto o botão **Executar Assíncrono** permite que a contagem ocorra sem travar a tela, mantendo a interface responsiva e atualizada em tempo real.

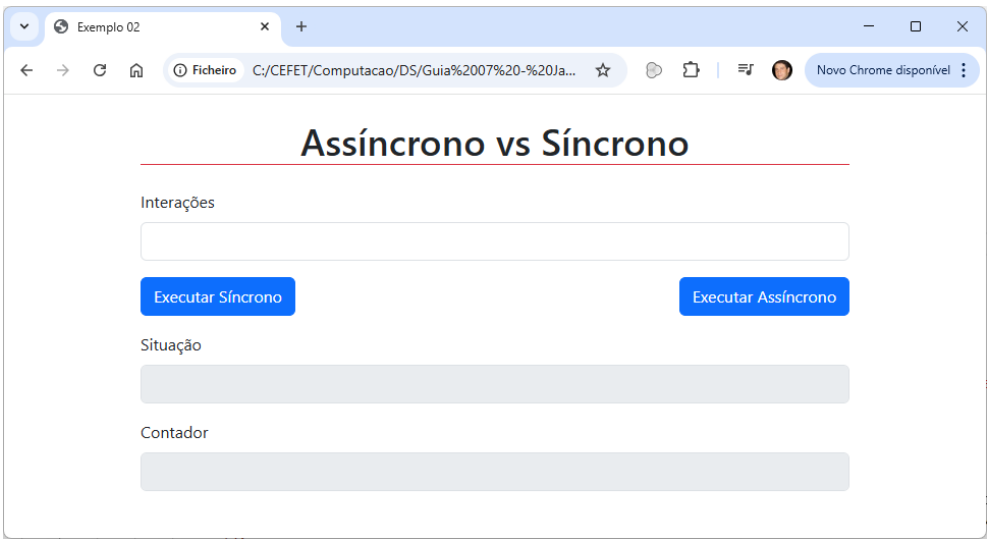

Figura 02: Visualizar do Exemplo 02

Na função `executarAssincrono()`, a instrução `await new Promise(resolve => setTimeout(resolve, 0));` divide a execução do *loop* em pequenos intervalos e libera o *event loop* do JavaScript. Isso permite que o navegador atualize a interface. Sem essa pausa assíncrona, a interface fica travada durante a execução do loop.

Exemplo 03

A função `preencherEndereco()` é executada automaticamente quando o campo de CEP perde o foco. Isso acontece por meio do evento `onblur`, que é acionado sempre que um elemento deixa de estar em foco.

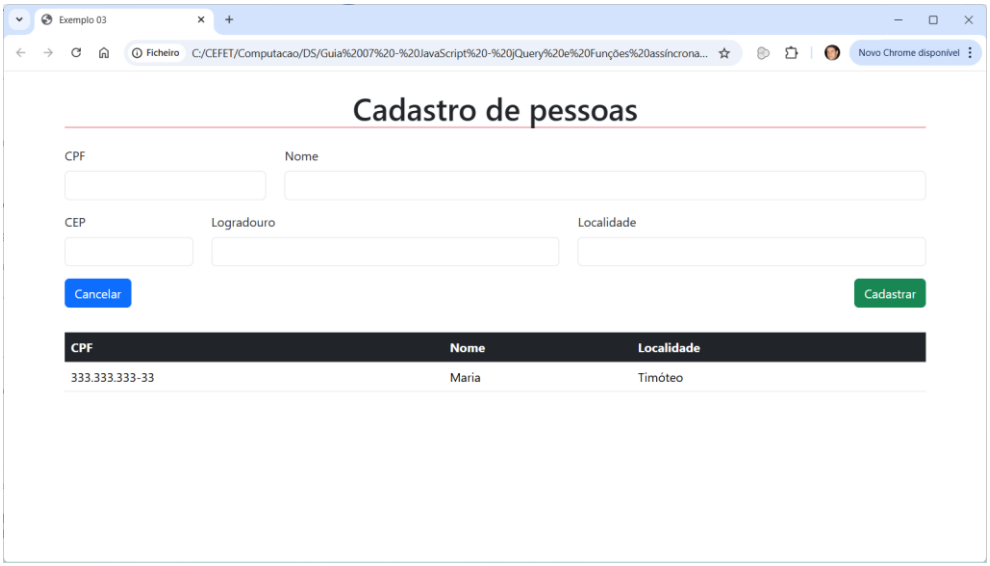

Figura 03: Visualizar do Exemplo 03

```
async function preencherEndereco() {
    var cep = $("#cep").val();
    try {
        if(cep.length >= 9){
            var resposta = await fetch("https://viacep.com.br/ws/" + cep + "/json/");
            var endereco = await resposta.json();
            $("#logradouro").val(endereco.logradouro)
            $("#localidade").val(endereco.localidade)
        }
    } catch (erro) {
        console.error('Erro na requisição:', erro);
    }
}
```

Exercícios

1. Com base na solução desenvolvida para o **Exercício 02** do **Guia 06**, realize as modificações necessárias no sistema de controle de estacionamento para implementar um mecanismo de segurança por inatividade. Para isso, você deve implementar a funcionalidade que:
    a. Monitore a inatividade do usuário na tela de registro de vaga.
    b. Caso o usuário não registre nenhuma vaga por um período contínuo de 10 segundos:
        i. Exiba uma mensagem de aviso informando sobre o encerramento da sessão por inatividade.
        ii. Desconecte automaticamente o usuário.
        iii. Redirecione o usuário para a página de login.

    Dica: crie uma função `iniciarTemporizador()` que reinicia o temporizador a cada registro de vaga e uma função `encerrarSessao()` para executar as ações de finalização.

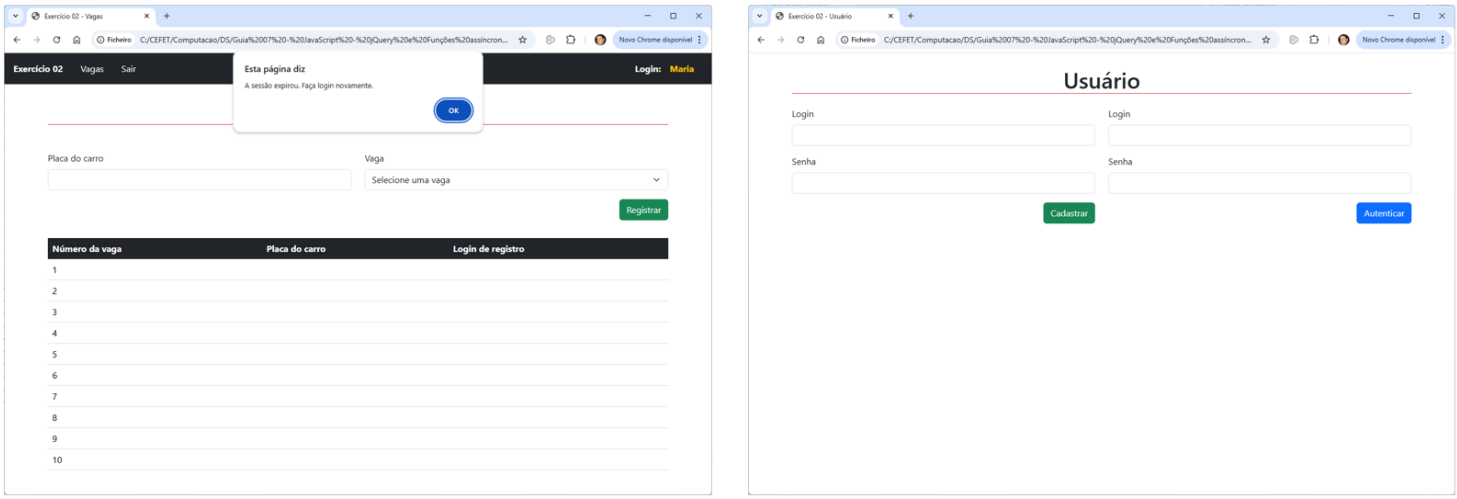

Figura 04: Resultado das modificações do exercicio

2. Desenvolva uma aplicação web composta por uma única página, utilizando HTML, CSS e JavaScript, que permita ao usuário selecionar um estado da região Sudeste e visualizar todas as suas cidades (municípios). A obtenção dos dados deve ser feita por meio da API do IBGE, que fornece informações sobre estados, municípios, população, área e outras características geográficas e demográficas.

    Documentação da API IBGE:
    https://servicodados.ibge.gov.br/api/docs/localidades

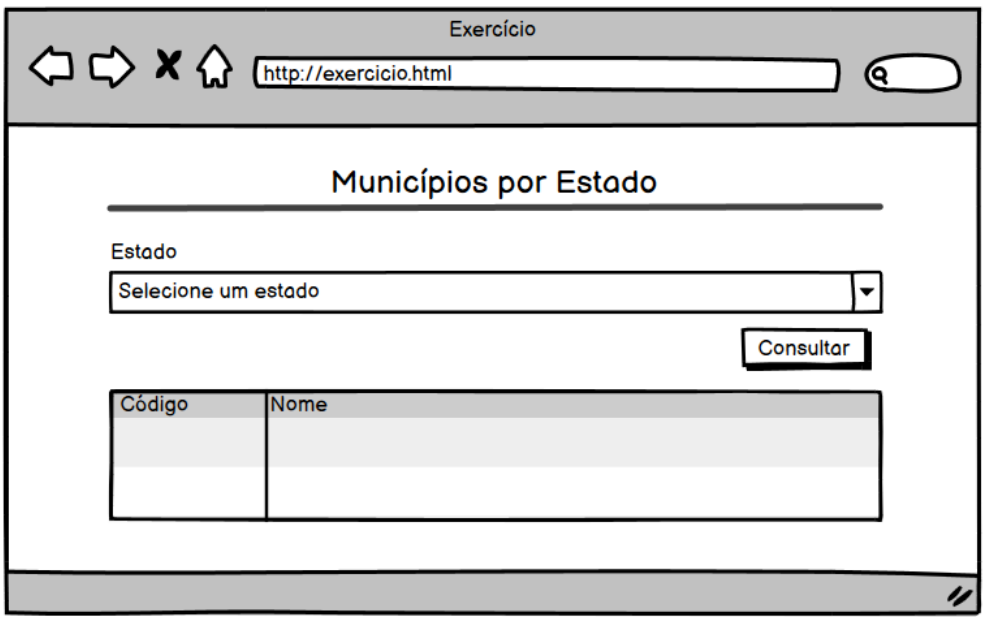

Figura 05: Protótipo do exercício 02