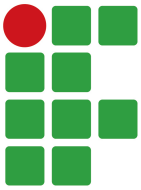

INSTITUTO FEDERAL
Espírito Santo
Campus de Alegre

Apostila — Material de apoio

JavaScript em Ação: Eventos e DOM Dinâmico

Da Teoria Fundamental à Prática de Mercado

Desenvolvimento Front-End I

Período: TADS EaD - Semana 10

Autor: Prof. Cleziel Franzoni da Costa

Instituto Federal do Espírito Santo — Campus de Alegre

2025

Sumário

1 Objetivos da Semana

Nesta semana, faremos a transição mais importante do front-end: de páginas estáticas (apenas HTML/CSS) para **aplicações interativas**. Você aprenderá a "ouvir"e "reagir"a cada ação do usuário.

Nossos objetivos são:

- **Dominar o Fluxo de Eventos:** Entender *exatamente* o que acontece quando um usuário clica, digita ou move o mouse.
- **Aplicar Padrões de Mercado:** Aprender a manipular o DOM da forma correta, performática e acessível (`classList`, Delegação de Eventos).
- **Separar Responsabilidades (SoC):** Garantir que seu HTML seja para estrutura, seu CSS para estilo e seu JavaScript *apenas* para comportamento.
- **Aplicar no Estudo de Caso:** Tornar o site da **Incubadora do Ifes** funcional, implementando a validação do formulário de pré-incubação e outras interatividades.

2 Revisão Essencial: O DOM e Seletores

2.1 O que é o DOM (Document Object Model)?

Pense no seu arquivo `index.html` como a **planta arquitetônica** de um prédio. Ele é apenas um documento estático.

Quando o navegador lê esse arquivo, ele "constrói"o prédio em sua memória. Essa "construção"viva é o **DOM**. O DOM é uma **API** (Interface de Programação de Aplicações) que nos permite interagir com a estrutura da página.

O JavaScript não edita seu arquivo `.html`. Ele edita essa estrutura em memória (o DOM), e o navegador *instantaneamente* atualiza a tela para refletir essa mudança.

2.2 Selecionando Elementos: `getElementById` vs. `querySelector`

Para manipular algo, você precisa primeiro segurá-lo.

`document.getElementById('id-unico')`:

- **O que faz:** Seleciona *um* elemento pelo seu atributo `id`.
- **Por que usar:** É o método de seleção **mais rápido** que existe. O navegador usa uma tabela de hash (como um dicionário) para encontrar IDs instantaneamente.
- **Exemplo:** `const form = document.getElementById('formPreIncubacao');`

`document.querySelector('seletor-css')`:

- **O que faz:** Seleciona o *primeiro* elemento que corresponde a *qualquer* seletor CSS válido.
- **Por que usar:** É o método **mais versátil**. Você pode selecionar por classe (`.btn`), por atributo (`[type="submit"]`), ou por hierarquia (`nav > ul > li`).
- **Exemplo:** `const botaoEnvio = document.querySelector('#formPreIncubacao .btn-submit');`

2.3 O Método Moderno: Por que usar `addEventListener`

Existem três formas de lidar com eventos. Duas delas são consideradas legadas ou más práticas.

1. **Má Prática (HTML Atrelado):**

```html
<button onclick="validarFormulario()">Enviar</button>
```

Atenção (Anti-Padrão): Viola a Separação de Preocupações (SoC). Você está misturando lógica de JavaScript (comportamento) dentro do seu HTML (estrutura).

2. **Prática Antiga (Propriedade JS):**

```javascript
const botao = document.querySelector('.btn-submit');
botao.onclick = validarFormulario;
botao.onclick = enviarAnalytics; // CUIDADO! A função validarFormulario foi SOBRESCRITA.
```

Atenção (Anti-Padrão): Você só pode ter **uma** função por evento. A segunda atribuição apaga a primeira.

3. **Boa Prática Moderna (`addEventListener`):**

```javascript
const botao = document.querySelector('.btn-submit');

// Anexa um "ouvinte" ao evento 'click'
botao.addEventListener('click', validarFormulario);

// Anexa OUTRO "ouvinte" ao MESMO evento. Ambos serão executados!
botao.addEventListener('click', enviarAnalytics);
```

Boa prática: Permite **múltiplos *listeners*** para o mesmo evento; mantém o HTML limpo (SoC); e dá mais controle (veremos *capturing* e *options* abaixo).

3 O Ciclo de Vida do Evento (O Aprofundamento)

Este é o conceito mais importante da semana. Quando você clica em um link dentro de um `<div>`, quem é "clicado"primeiro?

O evento acontece em **Três Fases**:

1. **Fase de Captura (Capturing):** O evento "desce"da raiz até o alvo.
   `Window` → `document` → `<html>` → `<body>` → `<div>` → `<a>`
2. **Fase de Alvo (Target):** O evento chega no elemento exato que você interagiu.
   Chegou no `<a>`
3. **Fase de Propagação (Bubbling):** O evento "borbulha"de volta para a raiz.
   `<a>` → `<div>` → `<body>` → `<html>` → `document` → `Window`

Por padrão, todos os `addEventListener` "escutam"na Fase de Bubbling (Propagação).

3.1 O Objeto `event` (O Mensageiro)

Sua função de *callback* sempre recebe um argumento, que chamamos por convenção de `e` ou `event`. Ele é um objeto com todos os dados sobre o que aconteceu.

Os mais importantes são `e.target` vs. `e.currentTarget`:

- `e.target`: **Quem originou o evento?** É o elemento exato onde o usuário clicou.
- `e.currentTarget`: **Quem está ouvindo o evento?** É o elemento onde você anexou o `addEventListener`.

Exemplo crucial (linha a linha):

```html
<button id="meuBotao">
  Clique <strong>aqui</strong>
</button>
```

Listing 1: Exemplo de HTML com elemento aninhado

```javascript
// Seleciona o botão
const botao = document.getElementById('meuBotao');

// Adiciona o ouvinte AO BOTÃO
botao.addEventListener('click', (e) => {
  // 'e.currentTarget' éo elemento onde o listener foi anexado.
  // Será SEMPRE o <button id="meuBotao">
  console.log('e.currentTarget:', e.currentTarget);

  // 'e.target' éo elemento exato que o usuário clicou.
```

```
  // Se o usuário clicar no texto "aqui", e.target será o <strong>.
  // Se o usuário clicar no espaço vazio do botão, e.target será o <button>.
  console.log('e.target:', e.target);
});
```

Listing 2: Diferença entre target e currentTarget

Isso é a base para a **Delegação de Eventos** (ver 6.2).

3.2 `e.preventDefault()`: Interceptando Ações Padrão

- **O que faz:** Impede o comportamento padrão do navegador para aquele evento.
- **Quando usar:**
  - `submit` de um `<form>`: Para impedir o recarregamento da página e fazer a validação com JS.
  - `click` em um `<a>`: Para impedir que o navegador siga o link (ex: em Single Page Applications).
  - `keydown` em um `<input type="text">`: Para impedir que o usuário digite certos caracteres.

Exemplo (linha a linha):

```
// Pega o formulário da Incubadora
const form = document.getElementById('formPreIncubacao');

// Ouve o evento de 'submit' (envio)
form.addEventListener('submit', (e) => {
  // A PRIMEIRA COISA a se fazer.
  // Impede que o formulário recarregue a página.
  e.preventDefault();

  // Agora o navegador parou, e podemos rodar nossa lógica de validação
  console.log('Formulário interceptado! O JavaScript assume daqui.');
  validarCampos(e);
});
```

Listing 3: Interceptando o envio de um formulário

3.3 `e.stopPropagation()`: Parando a "Bolha"

- **O que faz:** Impede que o evento continue sua fase de *Bubbling* (subindo) para os elementos pais.

- **Quando usar:** Quando um elemento interno tem uma ação que não deve disparar a ação de um elemento externo.
- **Exemplo Clássico:** Um Pop-up (Modal) com um fundo escuro (Overlay).

Exemplo (linha a linha):

```html
<div id="overlay" class="overlay-container">
  <div id="modal" class="modal-conteudo">
    <p>Clique fora de mim para fechar.</p>
  </div>
</div>
```

Listing 4: Estrutura de um Modal/Overlay

```javascript
// O ouvinte para fechar está no PAI (overlay)
const overlay = document.getElementById('overlay');
overlay.addEventListener('click', () => {
  console.log('Clique no OVERLAY. Fechando modal...');
  overlay.style.display = 'none'; // Esconde o modal
});

// O ouvinte para PARAR o clique está no FILHO (modal)
const modal = document.getElementById('modal');
modal.addEventListener('click', (e) => {
  console.log('Clique no MODAL. Parando a propagação!');
  // Impede que o clique "borbulhe" para o overlay
  e.stopPropagation();
});
```

Listing 5: Usando stopPropagation para evitar fechamento indesejado

4 Catálogo de Eventos Essenciais

- **Eventos de Mouse:**
  - `click`: Clique (pressionar e soltar).
  - `mouseover`: Ponteiro *entra* no elemento (dispara repetidamente se entrar em elementos filhos).
  - `mouseout`: Ponteiro *sai* do elemento (dispara ao sair de filhos).
  - `mouseenter`: Ponteiro *entra* no elemento (dispara **só uma vez**, não dispara para filhos). **(Preferido sobre `mouseover`)**
  - `mouseleave`: Ponteiro *sai* do elemento (dispara **só uma vez**, não dispara para filhos). **(Preferido sobre `mouseout`)**

- **Eventos de Teclado:**
  - `keydown`: Tecla é **pressionada**. Dispara repetidamente se a tecla for mantida pressionada.
  - `keyup`: Tecla é **solta**.
  - `input`: **O melhor evento para formulários.** Dispara *imediatamente* quando o valor de `<input>` ou `<textarea>` muda (seja digitando, colando com o mouse, cortando, etc.).
- **Eventos de Formulário:**
  - `submit`: Disparado no elemento `<form>` ao ser enviado.
  - `focus`: Elemento (ex: `<input>`) recebe foco.
  - `blur`: Elemento perde o foco (ótimo para validar um campo assim que o usuário sai dele).
  - `change`: Valor de um elemento muda *e* ele perde o foco (para inputs de texto) ou muda imediatamente (para `<select>`, `<input type="checkbox»`).

5 Manipulação Dinâmica do DOM

5.1 A Prática Correta: Gerenciando Classes com `classList`

Atenção (Anti-Padrão): Manipular o `style` diretamente no JS:

```
elemento.style.color = 'red';
elemento.style.fontSize = '1.2rem';
```

Por que é ruim?

1. **Mistura de Responsabilidades (SoC):** Seu JS agora está fazendo o trabalho do CSS.
2. **Manutenção Impossível:** E se o "estado de erro"mudar e precisar de um fundo também? Você teria que editar o JS, não o CSS.
3. **Especificidade:** Estilos *inline* (via `style`) têm altíssima precedência e "vencem"suas folhas de estilo, causando bugs.

Boa prática: O PADRÃO MODERNO (`classList`): O JS gerencia o **estado** (a classe), e o CSS gerencia a **aparência** desse estado.

```
/* No seu arquivo CSS (ex: form.css) */
.campo-formulario {
```

```css
  border: 1px solid #ccc;
  transition: all 0.3s ease;
}

.campo-formulario.erro {
  border-color: red;
  background-color: #fff8f8;
}

.nav-principal.menu-aberto {
  display: block; /* Exemplo de menu */
}
```

Listing 6: CSS: Definindo os estados

```javascript
/* No seu arquivo JS (ex: main.js) */

// Simplesmente adicione ou remova a classe. O CSS faz o resto.
const emailInput = document.getElementById('email');

// 1. Adicionar uma classe
emailInput.classList.add('erro');

// 2. Remover uma classe
emailInput.classList.remove('erro');

// 3. Alternar (toggle): Se tem, remove. Se não tem, adiciona.
// Perfeito para menus, modo escuro, etc.
const botaoMenu = document.getElementById('btn-menu');
const nav = document.querySelector('.nav-principal');

botaoMenu.addEventListener('click', () => {
  nav.classList.toggle('menu-aberto');
});

// 4. Verificar se contém
if (emailInput.classList.contains('erro')) {
  console.log('Este campo está com erro.');
}
```

Listing 7: JS: Gerenciando as classes (estados)

5.2 Alterando Conteúdo: `textContent` vs. `innerHTML`

`elemento.textContent = 'Novo Título'`:

- **O que faz:** Insere **texto puro**. É seguro e rápido.

- **Segurança:** Se a variável `nome` for `«strong>Gabriel</strong>"`, a página mostrará literalmente o texto `<strong>Gabriel</strong>`. O navegador *não* interpreta o HTML.
- **Quando usar: SEMPRE**, a menos que você *precise* inserir HTML.

`elemento.innerHTML = '<strong>Novo Título</strong>'`:

- **O que faz:** Insere **HTML**. O navegador "lê"a string e a transforma em elementos do DOM.
- **PERIGO (XSS):** É a principal porta de entrada para ataques de **Cross-Site Scripting (XSS)**. Se um usuário malicioso insere `<img src=x onerror="alert('Hackeado!')">` em um comentário, e você usa `innerHTML` para exibi-lo, esse script *vai rodar* no navegador de outros usuários.
- **Quando usar:** Apenas quando *você*, o desenvolvedor, controla 100% da string HTML.

6 Padrões e Boas Práticas de Mercado

6.1 O `this` em Eventos: Arrow Functions vs. Funções Tradicionais

- **Função Tradicional (`function() {...}`)**: O `this` é dinâmico. Dentro de um *listener*, `this` se torna o elemento que está ouvindo (`e.currentTarget`).

```
botao.addEventListener('click', function(e) {
  // 'this' aqui éo PRÓPRIO 'botao'
  this.classList.toggle('ativo');
  console.log(this === e.currentTarget); // true
});
```

Listing 8: Função tradicional: 'this' é o elemento

- **Arrow Function (`() => {...}`)**: O `this` é léxico. Ele "herda"o `this` do escopo onde foi criado (geralmente `window`, ou o `this` de uma Classe).

```
botao.addEventListener('click', (e) => {
  // 'this' aqui é'window' (ou o escopo externo)
  // this.classList.toggle('ativo'); // ISSO DARIA ERRO!

  // Para acessar o botão, usamos 'e.currentTarget'
  e.currentTarget.classList.toggle('ativo');
});
```

Listing 9: Arrow function: 'this' é o escopo externo

Boa prática: Use **Arrow Functions** por padrão. Elas são mais previsíveis e evitam confusões com o `this` dinâmico. Se precisar do elemento, use a variável original (`botao`) ou `e.currentTarget`.

6.2 Padrão de Performance: Delegação de Eventos (Event Delegation)

Atenção (Anti-Padrão): O Problema: No site da Incubadora, imagine uma lista com 100 *startups* (`<li>`). Você precisa que cada `<li>` seja clicável. Você *não deve* fazer um *loop* e adicionar 100 `addEventListener`. Isso consome muita memória.
Boa prática: A Solução (Delegação): Adicione **UM** *listener* no elemento PAI (`<ul>`). Graças ao *Bubbling* (Seção 3), o clique no `<li>` (filho) vai "borbulhar"até a `<ul>` (pai). Lá, nós usamos `e.target` para descobrir *quem* foi clicado.

Exemplo (linha a linha):

```html
<ul id="lista-startups" class="lista-interativa">
  <li data-id="123">Startup A</li>
  <li data-id="124">Startup B</li>
  <li data-id="125">Startup C</li>
  </ul>
```

Listing 10: HTML: Uma lista de itens

```javascript
// 1. Seleciona o PAI
const lista = document.getElementById('lista-startups');

// 2. Adiciona UM ÚNICO listener no PAI
lista.addEventListener('click', (e) => {
  // 'e.target' équem foi clicado (o <li>)
  // 'e.currentTarget' équem está ouvindo (a <ul>)

  // 3. Verificamos se o alvo (e.target) équem nos interessa.
  // Usamos .matches() para ver se o alvo éum 'li'
  if (e.target.matches('li')) {

    // 4. A mágica acontece!
    console.log('Você clicou na startup com ID:', e.target.dataset.id);

    // Remove a classe 'selecionado' de todos os outros
    lista.querySelectorAll('li').forEach(li => li.classList.remove('selecionado'));

    // Adiciona a classe 'selecionado' APENAS no alvo
    e.target.classList.add('selecionado');
  }
```

```
});
```

Listing 11: JS: Delegação de eventos

Vantagens: Performance (1 *listener* vs 100) e funciona para elementos adicionados *depois* (se você adicionar uma nova `<li>` com JS, ela já será "clicável").

6.3 Acessibilidade (a11y): Garantindo Interações via Teclado

Se algo é interativo, *deve* funcionar com teclado.

1. **Use Elementos Semânticos:** `<button>` e `<a>` já vêm com acessibilidade de graça (focam com 'Tab', ativam com 'Enter'/'Espaço').
2. **Se usar `<div>` como botão:**
   - Adicione `role="button"` (para o leitor de tela saber que é um botão).
   - Adicione `tabindex="0"` (para permitir que seja focado com 'Tab').
   - Adicione um *listener* de `keydown` para `e.key === 'Enter'` ou `e.key === ' '`.
3. **Feedback Dinâmico (ARIA):** Para mensagens de erro (como no nosso formulário), use `aria-live="polite"`. Isso faz o leitor de tela *anunciar* a mensagem de erro quando ela aparecer.

7 Exemplos Práticos Comentados (Estudo de Caso da Incubadora)

7.1 Exemplo 1: Trocando Temas (Modo Escuro/Claro) com `localStorage`

Vamos adicionar um botão de modo escuro ao site da Incubadora e *lembrar* a escolha do usuário.

```
<button id="toggle-tema">Alternar Tema</button>
```

Listing 12: HTML: Botão de toggle

```
/* Por padrão (light mode) */
body {
  background-color: white;
  color: black;
}

```

```css
/* Quando o body tiver a classe 'dark-mode' */
body.dark-mode {
  background-color: #121212;
  color: white;
}
```

Listing 13: CSS: Estados de tema

```javascript
// Seleciona o botão de toggle
const toggleBtn = document.getElementById('toggle-tema');

// Define a chave que usaremos no "banco de dados" do navegador
const CHAVE_STORAGE = 'tema-preferido-incubadora';

// 1. FUNÇÃO PARA APLICAR O TEMA
function aplicarTema(tema) {
  // Adiciona ou remove a classe do <body>
  // O segundo argumento do toggle força um estado:
  // true = adiciona, false = remove
  document.body.classList.toggle('dark-mode', tema === 'dark');
  // Salva a escolha no localStorage
  localStorage.setItem(CHAVE_STORAGE, tema);
}

// 2. EVENTO DE CLIQUE NO BOTÃO
toggleBtn.addEventListener('click', () => {
  // Verifica qual éo tema ATUAL
  let temaAtual = document.body.classList.contains('dark-mode') ? 'dark' : 'light';

  // Define o NOVO tema (o oposto)
  let novoTema = temaAtual === 'dark' ? 'light' : 'dark';

  // Aplica o novo tema
  aplicarTema(novoTema);
});

// 3. AO CARREGAR A PÁGINA
document.addEventListener('DOMContentLoaded', () => {
  // 'DOMContentLoaded' éo evento que dispara quando o HTML foi
  // totalmente carregado e processado (antes de imagens, etc).

  // Busca a preferência salva do usuário
  const temaSalvo = localStorage.getItem(CHAVE_STORAGE);

  // Se houver uma preferência salva (ex: 'dark' ou 'light')
  if (temaSalvo) {
```

```
    aplicarTema(temaSalvo); // Aplica o tema salvo
  }
  // Se for a primeira visita, não faz nada (usa o padrão do CSS)
});
```

Listing 14: JS: Lógica de toggle e localStorage (comentado)

7.2 Exemplo 2: Validação (Melhorada) do Formulário da Incubadora

```
<form id="formPreIncubacao">
  <div>
    <label for="nome">Nome Completo:</label>
    <input type="text" id="nome" required>
  </div>
  <div>
    <label for="email">E-mail:</label>
    <input type="email" id="email" required>
  </div>

  <p id="feedback-form" class="feedback" aria-live="polite"></p>

  <button type="submit">Enviar Inscrição</button>
</form>
```

Listing 15: HTML: Formulário com acessibilidade (aria-live)

```
.feedback {
  padding: 1em;
  border-radius: 5px;
  margin-top: 1em;
  font-weight: bold;
  display: none; /* Começa oculto */
}
.feedback.sucesso {
  display: block;
  background-color: #d4edda;
  color: #155724;
}
.feedback.erro {
  display: block;
  background-color: #f8d7da;
  color: #721c24;
}
```

Listing 16: CSS: Classes de feedback (sucesso/erro)

```
// Aguarda o DOM estar pronto para rodar o script
document.addEventListener('DOMContentLoaded', () => {

  // --- 1. SELEÇÃO DOS ELEMENTOS ---
  // Seleciona o formulário principal
  const form = document.getElementById('formPreIncubacao');

  // Seleciona os campos que vamos validar
  const inputNome = document.getElementById('nome');
  const inputEmail = document.getElementById('email');

  // Seleciona o parágrafo onde daremos o feedback
  const msgFeedback = document.getElementById('feedback-form');

  // --- 2. OUVINTE DE EVENTO 'SUBMIT' ---
  // Ouve o evento de 'submit' (envio) do formulário
  form.addEventListener('submit', (e) => {

    // Impede o comportamento padrão (recarregar a página)
    e.preventDefault();

    // --- 3. LÓGICA DE VALIDAÇÃO ---
    // Pega os valores ATUAIS dos campos
    // .trim() remove espaços em branco do início e do fim
    const nomeVal = inputNome.value.trim();
    const emailVal = inputEmail.value.trim();

    // Reseta o feedback anterior
    resetarFeedback();

    // Variável de controle
    let isValid = true;

    // Valida o nome
    if (nomeVal.length < 3) {
      // Se for inválido, mostra o erro e marca como inválido
      mostrarErro('Por favor, insira seu nome completo.');
      isValid = false;
    }

    // Valida o email (simples)
    if (!emailVal.includes('@') || !emailVal.includes('.')) {
      mostrarErro('Por favor, insira um e-mail válido.');
      isValid = false;
    }
```

```
46
47      // --- 4. FEEDBACK FINAL ---
48      // Se passou em todas as validações
49      if (isValid) {
50        mostrarSucesso(`Obrigado, ${nomeVal}! Sua pré-inscrição foi recebida.`);
51        // Limpa o formulário
52        form.reset();
53      }
54    });
55
56    // --- 5. FUNÇÕES AUXILIARES ---
57
58    // Função para mostrar mensagem de erro
59    function mostrarErro(mensagem) {
60      // Adiciona o texto da mensagem
61      msgFeedback.textContent = mensagem;
62      // Adiciona a classe CSS de 'erro'
63      msgFeedback.classList.add('erro');
64    }
65
66    // Função para mostrar mensagem de sucesso
67    function mostrarSucesso(mensagem) {
68      msgFeedback.textContent = mensagem;
69      // Adiciona a classe CSS de 'sucesso'
70      msgFeedback.classList.add('sucesso');
71    }
72
73    // Função para limpar as mensagens e classes
74    function resetarFeedback() {
75      // Limpa o texto
76      msgFeedback.textContent = '';
77      // Remove as classes de estado
78      msgFeedback.classList.remove('sucesso', 'erro');
79    }
80  });
```

Listing 17: JS: Lógica de validação completa (comentada)

8 Conclusão da Semana

Você agora entende a mecânica da interatividade. Você não está mais apenas "construindo páginas", está "arquitetando experiências". O domínio de `addEventListener`, `classList`, `preventDefault` e **Delegação de Eventos** é o que define um desenvolvedor front-end profissional. O site da Incubadora do Ifes – Campus de Alegre agora pode se tornar uma

plataforma dinâmica e responsiva às necessidades do usuário.

9 Referências para Aprofundamento

- **MDN Web Docs - `addEventListener`:** A documentação oficial e mais completa.
  https://developer.mozilla.org/pt-BR/docs/Web/API/EventTarget/addEventListener
- **MDN Web Docs - O Objeto `Event`:** Veja todas as propriedades (como `target`, `key`, `clientX`).
  https://developer.mozilla.org/pt-BR/docs/Web/API/Event
- **MDN Web Docs - `classList`:**
  https://developer.mozilla.org/pt-BR/docs/Web/API/Element/classList
- **JavaScript.info - Delegação de Eventos (Event Delegation):** Uma das melhores explicações visuais sobre o tema.
  https://javascript.info/event-delegation
- **web.dev (Google) - Acessibilidade (a11y):**
  https://web.dev/learn/accessibility/