Programação de Sistemas para Internet

Prof. Diego Cirilo

Aula 13: Introdução ao JavaScript

Introdução

- Até agora as funcionalidades do sistema web estão no servidor (*back-end*);
- O servidor recebe as requisições, processa/acessa dados e *monta* o HTML;
- As páginas HTML são enviadas *prontas* para o cliente (navegador);
- Depois de enviado ao cliente, o servidor não tem mais controle sobre a página;
- Os sistemas atuais não estão mais limitados a isso.

JavaScript

- Linguagem *interpretada*, com tipagem dinâmica e multi-paradigma;
- Desenvolvida nos anos 90 para **dinamizar** páginas web;
- Permite alterar o conteúdo da página no lado do cliente;
- É executada por uma *engine* no navegador;
- Em meados dos anos 2000 surgiram os *runtimes* nativos, como o Node.js;
- Hoje em dia é uma linguagem de uso geral (*full-stack*).

Runtimes do JS

- Nativas:
  - Ambiente de execução de JavaScript *server-side*;
  - Oferece APIs para acessar o sistema de arquivos, redes e outras funcionalidades do servidor;
  - Exemplos: Node.js, Deno, Bun;
- Browser Engines:
  - V8 (Chrome, Edge), SpiderMonkey (Firefox), JavaScriptCore (Safari);
  - Executam JavaScript diretamente nos navegadores, oferecendo suporte para aplicações web interativas.

JavaScript em PSI

- Nessa disciplina utilizaremos o JS no *browser*;
- Os objetivos são:
  - Melhorar a interface com a manipulação do DOM;
  - Criar páginas mais dinâmicas;
  - Trocar informações com servidor sem recarregar a página (AJAX).

Executando o JS

- No *browser*:
  - É possível usar o *console* do navegador;
  - É possível embutir o JS em arquivos HTML (ou templates Django);
    - Tag `<script></script>`;
- É possível também executar nativamente usando um *runtime* como Node.js, Deno, etc;
  - Não é nosso objetivo agora.

Embutindo o JS no HTML

- Podemos escrever o JS diretamente no arquivo HTML, dentro da tag `<script>`;
- Podemos separar o código JS em arquivos estáticos `.js`;
- No HTML é carregado com:
  - `<script src="meuscript.js"></script>`;
- No Django usamos a *tag* `static`:
  - `<script src="{% static 'meuscript.js' %}"></script>`;

Embutindo o JS no HTML

- A tag `<script>` pausa o carregamento do HTML para baixar e executar o JS;
- A ordem importa!
- Normalmente os *scripts* JS são carregados no final do arquivo, antes do `</body>`:
  - Não atrapalha o carregamento do HTML;
  - "Garante" que DOM já foi toda carregada antes.

Embutindo o JS no HTML

- É possível importar scripts no `<head>`, como é feito com o CSS;
- Para garantir que só sejam executados com o DOM carregado usamos o atributo `defer`:
  - `<script defer src="meuscript.js"></script>`
- A vantagem em relação a colocar a tag no final do `<body>` é que o *script* é baixado junto com a página;
- A desvantagem é que os blocos `<script>` *inline* não terão acesso às funções importadas dessa forma, pois serão carregadas antes;
- A ordem de carregamento importa!

Embutindo o JS no HTML

- Em um bloco `<script>` *inline*, podemos garantir que o código só será executado após o carregamento completo do conteúdo com:

```
<script>
    window.onload = function () {
        // codigo executado apenas após o
        // carregamento completo
        // da pagina
    }
</script>
```

Sintaxe JS

- A sintaxe é parecida com C/C++/Java;
- Usa `;` para indicar o fim de uma diretiva;
- Usa `{}` para abrir e fechar diretivas, funções, etc;
- O *whitespace* não importa, ao contrário do Python;
- Comentários com `//` (linha) ou `/* etc */` (bloco).

Sintaxe JS

- Comumente usamos:
  - camelCase para nome de funções e variáveis;
  - Um espaço antes do `{}` ;
  - *Style Guide*

Declarações

- Variáveis podem ser declaradas com:
  - Automaticamente (não recomendado);
  - `var` - Escopo global com *hoisting* (*legado*);
  - `let` - Variável *normal* com escopo de bloco;
  - `const` - Escopo de bloco, o valor não pode ser atribuído novamente;
- No caso do `const`, se o valor for um objeto/array, o conteúdo do objeto pode ser modificado.

Declarações

- *Hoisting* (içamento):
  - Joga as declarações automaticamente para o topo do script;
  - Permite usar variáveis/funções que ainda serão declaradas;
  - Funciona com `var` e declaração de funções;
  - Pode ser uma fonte de *bugs* se não for tratado com cuidado.
- O uso de `var` não é mais recomendado, mas ainda existe em exemplos de código antigos;

Exemplos

```
casa = "IFRN";
casa = 8; //funciona
var casa; //declarou depois de usar, funciona e não perde o conteúdo (hoisting)

// recomendado atualmente
let rua = "Principal";
rua = "Rua de Cima";
rua = 77; //funciona
let rua; //erro! não pode declarar novamente

const bairro = "Centro";
bairro = "Mirassol"; //não funciona!
```

Tipo de dados

- O JavaScript tem tipagem *dinâmica* e *fraca*;
- `var` e `let` podem receber tipos de dados diferentes;
- Tipos primitivos:
  - String, Number, Bigint, Boolean, Undefined, Null, Symbol;
- O resto é objeto (*Object*).

Objetos JS

- JavaScript Object Notation (JSON):

```
const bejeto = {
    nome: "Ana",
    idade: 20,
    profissao: "Desenvolvedora",
    saudacao: function() {
        return `Olá, meu nome é ${this.nome}.`;
    }
};
```

Strings

- Podem ser delimitadas com:
  - `...`
  - "..."
  - '...'
- O `...` é chamado de *string literal* e permite interpolação, múltiplas linhas, etc.

```
const cor = "azul";
const informacao = `O display é ${cor}.`;
```

Condicionais

- `if`:

```
if (media > 60) {
  alert( 'Aprovado!' );
} else if (media > 40) {
  alert( 'Recuperação!' );
} else {
  alert( 'Reprovado!' );
}
```

- Operador ternário:

```
let resultado = (idade > 18) ? "acesso permitido" : "acesso negado";
```

Condicionais

- `switch`

```
switch (a) {
  case 1:
    alert( 'Primeiro!' );
    break;
  case 2:
    alert( 'Segundo!' ); // sem o break;
  case 3:
    alert( 'Terceiro!' );
    break;
  default:
    alert( "Desclassificado!" );
}
```

Operadores de comparação

- `==` igual, mas aceita tipos de dados diferentes;
- `===` igual até no tipo;
- `>, >=, <, <=, !=, !==`;

```
const a = 5;
const b = "5";
if (a == b){
  // é verdade!
}
if (a === b){
  // falso!
}
```

Laços

- `for`

```
for (let i = 0; i < 3; i++) {
  alert(i);
}
```

- `while`

```
while (i < 3) { // shows 0, then 1, then 2
  alert( i );
  i++;
}
```

- `do..while`

```
do {
  alert( i );
  i++;
} while (i < 3);
```

Funções no JS

- Funções padrão:

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

- Funções anônimas:

```
const saudacao = function(nome) {
    return `Olá, ${nome}!`;
};
```

- *Arrow functions*

```
const multiplicar = (a, b) => {
    const resultado = a * b;
    return resultado;
};
```

Arrow functions

- Retornam o valor por padrão

```
hello = () => "Hello World!";
```

- Se houver apenas um parâmetro

```
hello = val => "Hello " + val;
```

Funções *Callback*

- Funções que são passadas como parâmetro para outra função;
- Permite que a função original tenha controle sobre o momento de chamar a segunda função, mesmo que não saiba o nome dela;
- Muito usado no JS;
- Podemos passar uma função anônima *inline* como parâmetro;
- Ou definir a função antes, e passar apenas seu nome.

Exemplo

```
function minhaFuncao(par1, par2, outraFuncao) {
    let soma = par1 + par2;
    outraFuncao(soma); //callback
}

let resultado = minhaFuncao(3, 5, function(valor){
    //estou dentro do callback
    console.log(valor);
    return valor;
});

// como arrow function
resultado = minhaFuncao(3, 5, (valor) => {
    console.log(valor);
    return valor;
});

// definindo a função antes
function funcaoNova(resultado){
    console.log(resultado);
}

//passando a função definida
minhaFuncao(3, 5, funcaoNova);
```

Arrays

- Funcionam como listas do Python;
- Coleção indexada de objetos;
- Ex.

```
const meuArray = [];
meuArray = ["coisa1", {coisa2: "coisa 2"}, 5]; // não pode!
meuArray.push("coisa1");
meuArray.push({coisa2: "coisa 2"});
meuArray.push(5);
meuArray[6] = "coisa 6";
console.log(meuArray[0]);
```

Arrays

- Percorrendo Arrays:

```
meuArray.forEach( function (item) {
    console.log(item);
});
```

Manipulação do DOM

- *Document Object Model*;
- Uma das principais funções do JS é manipular o DOM;
- Criar/remover elementos, substituir conteúdo, alterar atributos, etc;
- Isso permite interfaces de usuário dinâmicas.

Manipulação do DOM

- Selecionar elementos:
  - `document.getElementById('id')`
  - `document.querySelector('.classe')`
  - `document.querySelectorAll('tag')`
- Modificar Conteúdo:
  - `element.textContent = 'Novo texto'`
  - `element.innerHTML = '<p>Novo HTML</p>'`

Manipulação do DOM

- Alterar Estilos
  - `element.style.color = 'red'`
  - `element.classList.add('nova-classe')`
  - `element.classList.remove('classe-existente')`
- Criar e Inserir Elementos
  - `document.createElement('div')`
  - `parentElement.appendChild(novoElemento)`
- Substituir conteúdo
  - `element.replaceChildren(novoElemento)`

Manipulação do DOM

- Exemplo:

```
const paragrafo = document.createElement('p');
paragrafo.textContent = 'Este é um novo parágrafo.';
document.body.appendChild(paragrafo);
```

Eventos DOM

- Os eventos reagem a ações do usuário, servidor ou temporizadas;
- Permitem a execução de funções quando algo acontece;
- Ex. `click`, `mouseover`, etc;
- Usamos o `elemento.addEventListener('nome_do_evento', funcao_callback)`.

Eventos DOM

- Funções *callback* são executadas quando o *EventListener* detecta o evento;
- Ex.:

```
const elemento = document.getElementById("meuBotao");
elemento.addEventListener('click', function() {
  minhaDiv = document.getElementById("minhaDiv");
  minhaDiv.style.backgroundColor = "red";
  minhaDiv.innerHTML = "<p class="alert">Clicaram no botão!</p>";
});
```

Exemplo

- `teste.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">
    <title>Teste</title>
    <script defer src="meuScript.js"></script>
  </head>
  <body>
    <div class="conteudo outra-classe">
      <h1>Teste</h1>
      <button id="meuBotao">Clique aqui!</button>
      <p>Meu conteúdo relevante.</p>
    </div>
  </body>
</html>
```

Exemplo

- `meuScript.js`

```
const botao = document.getElementById("meuBotao");
const conteudo = document.querySelector(".conteudo p");
const header1 = document.querySelector(".conteudo h1")
let contador = 0;
botao.addEventListener("click", () => {
  contador++;
  if (contador < 10) {
    conteudo.innerHTML = `<p>O botão foi clicado ${contador} vezes!</p>`;
  } else if (contador < 16){
    conteudo.innerHTML = `<p>O botão foi clicado ${contador} vezes! Por favor, pare!</p>`;
    header1.innerText = "TESTE";
    botao.innerText = "Não clique aqui!";
    botao.style.position = "absolute";
    botao.style.top = `${contador**2}px`;
  } else {
    let aviso = document.createElement('h1');
    aviso.innerText = "PARE!!!";
    aviso.style.fontSize = "20em";
    aviso.style.color = "yellow";
    document.body.style.backgroundColor = "red"
    document.body.replaceChildren(aviso);
  }
});
```

Bibliotecas JS

- No contexto da disciplina podem ser importadas na tag `<script src="biblioteca.js"></script>`;
- A ordem importa!
- Fornecem funcionalidades prontas;
- Exemplos: jQuery, React, Bootstrap, PDF.js, Babylon...

Referências

- https://javascript.info/
- https://developer.mozilla.org/pt-BR/docs/Web/JavaScript
- https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript

Dúvidas?