Programação Orientada a Serviços

Prof. Diego Cirilo

Aula 14: Clientes JavaScript

Introdução

- Usualmente as funcionalidades de um sistema web estão no servidor
- 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
- *Front-end* - interface gráfica do usuário
- Como *desacoplar* a interface de usuário do servidor?

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

Versões do JavaScript

- Padrão ECMAScript (*European Computer Manufacturers Association*)
- Iniciais
  - ECMAScript 1 (1997): Primeira versão padronizada para compatibilidade entre navegadores.
  - ECMAScript 3 (1999): Introduziu tratamento de erros, expressões regulares e estabeleceu a base do JavaScript moderno.

Versões do JavaScript

- ECMAScript 5 (2009)
  - Adicionou modo estrito, suporte a JSON e métodos de arrays (map, filter, reduce).
- ECMAScript 6 (ES6, 2015)
  - Let/Const (variáveis de escopo de bloco).
  - *Arrow functions*.
  - Classes e módulos.
  - *Promises* para operações assíncronas.

Recursos Modernos do JavaScript

- ECMAScript 2016-2017 (ES7 & ES8)
  - ES7 (2016): Adicionou `Array.prototype.includes()` e o operador de exponenciação (`**`).
  - ES8 (2017): Introduziu async/await e métodos de objeto como `Object.entries()` .
- ECMAScript 2018-2023 (ES9 a ES13)
  - Operadores spread/rest para objetos.
  - Encadeamento opcional (`?.`) e operador de coalescência nula (`??`).
  - Top-level await para simplificar operações

Runtimes do JS

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

Declarações

- Variáveis podem ser declaradas com:
  - Automaticamente (não recomendado)
  - `var` - Escopo global com *hoisting*.
  - `let` - Variável com escopo de bloco.
  - `const` - Constantes, o valor/tipo não pode mudar.

Hoisting

- 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.

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

```
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}.`;
```

Manipulação da DOM

- *Document Object Model*
- Uma das principais funções do JS é manipular a DOM
- Criar/remover elementos, substituir conteúdo, alterar atributos, etc

Manipulação da 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 da 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)`

Manipulação da DOM

- Exemplo:

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

jQuery

- Biblioteca desenvolvida pra simplificar a manipulação da DOM, eventos, requisições, etc.
- Usa uma linguagem menos *verbosa* que o JavaScript puro (*Vanilla*)
- Já foi "obrigatória", hoje é possível fazer *quase* tudo sem ela.
- Mesmo assim, ainda é mais cômodo utiliza-la.
- *You might not need jQuery*

Eventos JS

- Os eventos reagem a ações do usuário, servidor ou temporizadas
- Permitem a execução de funções quando algo acontece
- Ex. `click`, `mouseover`, etc.

```
elemento.addEventListener('click', function() {
    alert('Elemento clicado!');
});
```

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;
```

Promises

- Algumas operações não devem bloquear a execução do código
- Esse é o princípio de operações *assíncronas*
- O JavaScript pode retornar *promessas* em uma função que pode demorar
- O código continua sua execução.

Promises

- Uma Promise pode estar em um dos três estados:
  - Pendente (*pending*): Estado inicial, ainda não resolvida ou rejeitada.
  - Resolvida (*fulfilled*): A operação foi completada com sucesso.
  - Rejeitada (*rejected*): A operação falhou.

Promises

- Podemos consumir as promessas com:
  - `.then()` - função executada se der certo
  - `.catch()` - função executada se falhar

Promises

```
promessa
    .then(resultado => {
        console.log(resultado);  // "Operação bem-sucedida!"
    })
    .catch(erro => {
        console.error(erro);  // "Falha na operação."
    });
```

Async/Await

- No ES8 surge a sintaxe de *async* e *await* para Promises
- Torna o código um pouco mais legível
- As funções assíncronas são declaradas com *async* e a chamada de funções assíncronas com *await*

```
async function minhaFuncaoAssincrona() {
    try {
        const resultado = await promessa;
        console.log(resultado);  // "Operação bem-sucedida!"
    } catch (erro) {
        console.error(erro);  // "Erro na operação."
    }
}

minhaFuncaoAssincrona();
```

Fetch API

- A Fetch API é usada para fazer requisições HTTP no navegador.
- Substitui o antigo XMLHttpRequest
- Retorna uma *Promise*

Exemplo

```
async function buscarDados() {
    try {
        const response = await fetch('https://jsonplaceholder.typicode.com/todos/1');
        if (!response.ok) {
            throw new Error('Erro: ' + response.status);
        }
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error('Erro:', error);
    }
}

buscarDados();
```

Fetch API

- Outras opções:

```
fetch('https://api.exemplo.com/usuario/1', {
    method: 'PUT',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify({ nome: 'Maria', idade: 28 })
})
    .then(response => response.json())
    .then(data => console.log('Atualizado:', data))
    .catch(error => console.error('Erro:', error));
```

Tarefa

- Desenvolva a interface e crie um cliente web para uma API aberta.
- Exemplos:
  - JSON Placeholder, PokeAPI, Tabela FIPE, etc.
- O cliente deve listar mais de um nível de informações, ex. usuários e to-dos do usuário, fabricante e modelos e veículos.

Módulos

- Permitem a melhor organização do código
- Reuso e escopo
- Implementados no ES6
- *export* e *import*

ESM

```
// Exportando (arquivo myModule.js)
export const myFunction = () => { ... };

// Importando
import { myFunction } from './myModule.js';
```

Bundling

- Processo de combinar múltiplos arquivos JS em um único arquivo
- Necessário devido à divisão de código em múltiplos módulos
- Benefícios:
  - Reduz o número de requisições HTTP
  - Melhora o desempenho do site
  - Facilita a minificação e otimização do código
- Exemplos:
  - Webpack, Rollup, Parcel

Bundling

- Módulos são combinados em um único arquivo (ou múltiplos, dependendo da configuração)
- Ferramentas de bundling resolvem dependências e geram um arquivo otimizado
- Exemplo:
  - Arquivos de entrada: main.js, utils.js, app.js
  - Saída: bundle.js

Vite.js

- Ferramenta de desenvolvimento de front-end
- Funciona como bundler e servidor de desenvolvimento
- *Lembra* o que o `django-admin` faz no back-end
- Utiliza o *Rollup* para bundling
- Documentação

Tarefa

- Converta o seu cliente JS para uma estrutura JS Vanilla do Vite.js
- Separe o .js que se comunica com a API em um *wrapper* e o .js que manipula a DOM usando módulos

Referências

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

Dúvidas?