Programação Web II

Material de Apoio da Disciplina PWEB II do IFAL/Maceió

Programação Web II

Leo Fernandes, PhD

Sumário

Capítulo 1

A Web

1.1 História da Web

A história da Web é uma jornada fascinante que começou no final da década de 1980 e se desenvolveu rapidamente, transformando a maneira como interagimos com a informação e o mundo ao nosso redor.

A World Wide Web foi inventada por Tim Berners-Lee, um cientista britânico do CERN (Organização Europeia para a Pesquisa Nuclear), em 1989. Berners-Lee desenvolveu a Web como uma solução para compartilhar documentos entre os pesquisadores do CERN. Ele propôs um sistema de gerenciamento de informações que conectaria documentos utilizando hiperlinks, permitindo que os usuários navegassem facilmente de um documento para outro .

Em 1990, Berners-Lee, juntamente com o engenheiro de sistemas Robert Cailliau, desenvolveu os primeiros componentes essenciais da Web:

- **HTML (HyperText Markup Language):** A linguagem de marcação usada para criar documentos na Web.
- **HTTP (Hypertext Transfer Protocol):** O protocolo usado para a transferência de documentos pela Web.
- **URL (Uniform Resource Locator):** O sistema de endereçamento que permite identificar e acessar documentos na Web.

No final de 1990, Tim Berners-Lee tinha o primeiro servidor Web e navegador instalado e funcionando no CERN, demonstrando suas ideias. Ele desenvolveu o código para seu servidor Web em um computador NeXT. Para evitar que fosse desligado acidentalmente, o computador trazia uma etiqueta escrita à mão em tinta vermelha: "This machine is a server. DO NOT POWER IT DOWN!!"

O endereço do servidor web era *info.cern.ch* <http://info.cern.ch/>, foi lançado em 1990 no CERN, e a primeira página web (ainda ativa em <http://info.

cern.ch/hypertext/WWW/TheProject.html>) explicou o que era a World Wide Web e como utilizá-la.

A Web começou a se popularizar em 1993, com o lançamento do navegador Mosaic, desenvolvido por Marc Andreessen e Eric Bina no National Center for Supercomputing Applications (NCSA). O Mosaic foi o primeiro navegador a suportar gráficos embutidos, além de texto, em uma interface amigável. Isso marcou o início da explosão da Web como uma ferramenta acessível para o público em geral.

Em 1994, Berners-Lee fundou o World Wide Web Consortium (W3C) [1] no MIT, com o objetivo de desenvolver padrões abertos que garantissem o crescimento contínuo da Web de maneira estável e interoperável. O W3C continua a ser a principal organização de padrões para a Web até hoje .

Desde o seu início, a Web passou por várias evoluções importantes. A transição de uma Web estática, onde as páginas eram basicamente documentos de leitura, para uma Web dinâmica e interativa, onde aplicações complexas podem ser executadas diretamente no navegador, foi um marco significativo. Tecnologias como JavaScript, CSS e AJAX contribuíram para essa evolução.

A Web também teve um impacto profundo na sociedade, revolucionando a comunicação, o comércio, a educação, o entretenimento e praticamente todos os aspectos da vida moderna. Com o surgimento da Web 2.0, a interação social online tornou-se predominante, dando origem a plataformas como redes sociais, blogs e wikis.

Hoje, a Web continua a evoluir, com tendências emergentes como a Web Semântica, a Web3 e a integração de tecnologias de Inteligência Artificial, prometendo transformar ainda mais a maneira como vivemos e trabalhamos.

Para mais informações sobre a história da web, acesse (CERN, 2024).

[1] O World Wide Web Consortium (W3C) desenvolve padrões e diretrizes para ajudar todos a construir uma web baseada nos princípios de acessibilidade, internacionalização, privacidade e segurança. - <https://www.w3.org/>

Exercício 1.1

Qual é a diferença entre Internet e Web?

1.2 Protocolo de Transferência de Hipertexto (HTTP)

1.2.1 Requisições e Respostas

O Protocolo de Transferência de Hipertexto (HTTP) é a base da comunicação na web. Em essência, o HTTP funciona no modelo de requisição-resposta, onde um cliente (geralmente um navegador) faz uma requisição a um servidor, e o servidor responde com os dados solicitados.

O cliente envia uma requisição HTTP que consiste em uma linha de requisição, cabeçalhos e um corpo opcional. O servidor responde com uma resposta HTTP, que também contém uma linha de status, cabeçalhos e um corpo opcional.

1.2.2 Estrutura das Requisições HTTP

Uma requisição HTTP possui a seguinte estrutura básica:

- **Linha de Requisição:** Contém o método HTTP, o caminho da URL, e a versão do protocolo HTTP.
- **Cabeçalhos:** Metadados que fornecem informações adicionais sobre a requisição. Exemplo: ‘Host‘, ‘User-Agent‘, ‘Accept‘.
- **Corpo (opcional):** Contém os dados enviados com a requisição, como dados de formulários em uma requisição POST.

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html
```

1.2.3 Estrutura das Respostas HTTP

A resposta HTTP segue uma estrutura similar:

- **Linha de Status:** Inclui a versão do protocolo, um código de status, e uma frase descritiva.

- **Cabeçalhos:** Informações sobre o servidor e os dados que estão sendo enviados. Exemplo: ‘Content-Type‘, ‘Content-Length‘, ‘Set-Cookie‘.
- **Corpo (opcional):** Contém o conteúdo real da resposta, como uma página HTML.

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1234

<html>
  <body>...</body>
</html>
```

1.2.4 Cabeçalhos

Os cabeçalhos HTTP são essenciais para o controle de requisições e respostas. Eles podem ser divididos em quatro categorias principais:

- **Cabeçalhos Gerais:** Aplicam-se tanto a requisições quanto a respostas, como ‘Cache-Control‘ e ‘Date‘.
- **Cabeçalhos de Requisição:** Específicos para requisições, como ‘Accept‘, ‘User-Agent‘, e ‘Host‘.
- **Cabeçalhos de Resposta:** Específicos para respostas, como ‘Content-Type‘, ‘Server‘, e ‘Set-Cookie‘.
- **Cabeçalhos de Entidade:** Relacionam-se ao corpo da mensagem, como ‘Content-Length‘ e ‘Content-Encoding‘.

1.2.5 Métodos HTTP

Os métodos HTTP definem a ação a ser realizada na requisição. Os métodos mais comuns são:

- **GET:** Solicita a representação de um recurso específico. Não deve alterar o estado do servidor.
- **POST:** Envia dados para o servidor para criar ou modificar um recurso.
- **PUT:** Substitui o recurso no servidor pela carga útil da requisição.
- **DELETE:** Remove um recurso específico do servidor.
- **HEAD:** Solicita uma resposta idêntica a GET, mas sem o corpo de resposta.
- **PATCH:** Aplica modificações parciais a um recurso.

1.2.6 Códigos de Status

Os códigos de status HTTP informam o resultado da requisição. Eles são divididos em cinco classes:

- **1xx (Informativo):** Indica que a requisição foi recebida e está em processamento.
- **2xx (Sucesso):** Indica que a requisição foi bem-sucedida. Exemplo: 200 OK.
- **3xx (Redirecionamento):** Indica que o cliente precisa tomar uma ação adicional para completar a requisição. Exemplo: 301 Moved Permanently.
- **4xx (Erro do Cliente):** Indica que houve um erro na requisição. Exemplo: 404 Not Found.
- **5xx (Erro do Servidor):** Indica que o servidor encontrou um erro ao processar a requisição. Exemplo: 500 Internal Server Error.

1.2.7 Tipos MIME

O tipo MIME (Multipurpose Internet Mail Extensions) define o tipo de conteúdo da resposta HTTP. Os tipos MIME comuns incluem:

- **text/html:** Para documentos HTML.
- **application/json:** Para dados no formato JSON.
- **image/png:** Para imagens no formato PNG.
- **application/xml:** Para dados no formato XML.

1.2.8 Cookies

Cookies são pequenos pedaços de dados armazenados no cliente pelo servidor. Eles são usados para manter o estado entre requisições HTTP, que são inerentemente sem estado. Cookies podem armazenar informações como identificadores de sessão, preferências do usuário, e outros dados de rastreamento.

Os cookies são definidos no cabeçalho ‘Set-Cookie‘ da resposta HTTP e são enviados de volta ao servidor em requisições subsequentes.

1.2.9 Executando HTTP na Linha de Comando

Ferramentas como cURL permitem que desenvolvedores façam requisições HTTP diretamente da linha de comando. Isso é útil para testar APIs e depurar problemas de rede.

```
# Exemplo de requisição GET com cURL
curl -X GET https://api.example.com/data

# Exemplo de requisição POST com cURL
curl -X POST https://api.example.com/data -d '{"key":"value"}' -
H "Content-Type: application/json"
```

1.3 A linguagem de marcação de hyper-texto (HTML)

A **Linguagem de Marcação de Hyper-Texto (HTML)** é a espinha dorsal da Web, sendo a principal linguagem utilizada para criar e estruturar conteúdo na Internet. HTML é uma linguagem de marcação, o que significa que ela usa uma série de *tags* ou marcas para definir elementos dentro de um documento. Esses elementos podem incluir textos, imagens, links, tabelas, formulários, e muito mais.

1.3.1 Estrutura Básica de um Documento HTML

Todo documento HTML possui uma estrutura básica que deve ser seguida. A estrutura de um arquivo HTML típico é composta por:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Título da Página</title>
  </head>
  <body>
    <h1>Bem-vindo à Web</h1>
    <p>Este é um exemplo de documento HTML.</p>
  </body>
</html>
```

- **DOCTYPE:** Declara o tipo do documento e a versão do HTML que está sendo utilizada. O mais comum atualmente é o `<!DOCTYPE html>` que indica HTML5.
- **<html>:** A tag `<html>` encapsula todo o conteúdo do documento HTML.

- **<head>:** A tag <head> contém metadados sobre o documento, como o título que aparece na aba do navegador, links para folhas de estilo, scripts, e outras informações.
- **<body>:** A tag <body> contém todo o conteúdo visível para o usuário, como textos, imagens, links, e outros elementos de interface.

1.3.2 Elementos HTML

Os elementos HTML são representados por tags, que geralmente vêm em pares: uma tag de abertura e uma tag de fechamento. As tags de abertura e fechamento delimitam o início e o fim de um elemento.

- **Tags de Cabeçalho:** Usadas para criar títulos e subtítulos na página (<h1> a <h6>).
- **Tags de Parágrafo:** Usadas para definir parágrafos de texto (<p>).
- **Tags de Link:** Usadas para criar hiperlinks para outros documentos ou seções (<a href="...»).
- **Tags de Imagem:** Usadas para incluir imagens no documento (<img src="...»).
- **Tags de Lista:** Usadas para criar listas ordenadas (<ol>) ou não ordenadas (<ul>).

1.3.3 A Evolução do HTML

O HTML evoluiu significativamente desde sua criação. As primeiras versões do HTML focavam em estruturação básica de conteúdo, mas versões posteriores, como o HTML5, introduziram novos elementos semânticos, como <article>, <section>, e <nav>, além de suporte nativo para multimídia e APIs avançadas para desenvolvimento web.

O HTML5, em particular, tornou-se o padrão mais amplamente adotado, sendo projetado para ser compatível com dispositivos modernos e suportando uma gama mais ampla de funcionalidades, como vídeos embutidos, gráficos, e aplicativos interativos.

1.4 O sistema de endereçamento (URL)

A **Uniform Resource Locator (URL)**, ou Localizador Uniforme de Recursos, é um sistema de endereçamento utilizado na Web para identificar e localizar recursos disponíveis na Internet. Cada URL aponta para um recurso específico,

que pode ser uma página HTML, um documento PDF, uma imagem, ou qualquer outro tipo de conteúdo acessível na Web.

1.4.1 Estrutura de uma URL

Uma URL é composta por várias partes distintas que indicam diferentes aspectos do recurso a ser acessado. A estrutura geral de uma URL é a seguinte:

`http://www.exemplo.com:80/caminho/para/recurso?query=parametro#ancora`

Vamos decompor essa URL em suas partes componentes:

- **Protocolo:** `http://` ou `https://` especifica o protocolo de comunicação a ser utilizado. O protocolo HTTPS é uma versão segura do HTTP.
- **Nome de Domínio:** `www.exemplo.com` identifica o servidor onde o recurso está hospedado. Este nome de domínio pode ser traduzido em um endereço IP pelo DNS (Domain Name System).
- **Porta:** `:80` é a porta de comunicação do servidor utilizada pelo protocolo. A porta 80 é o padrão para HTTP, enquanto a porta 443 é padrão para HTTPS.
- **Caminho:** `/caminho/para/recurso` indica o caminho no servidor para o recurso específico. Pode referenciar diretórios ou arquivos.
- **Query String:** `?query=parametro` é uma cadeia de caracteres que fornece parâmetros adicionais para a requisição, frequentemente utilizados para busca ou filtragem de dados.
- **Fragmento:** `#ancora` refere-se a uma parte específica do recurso, como uma âncora em uma página HTML que leva a um local específico dentro do documento.

1.4.2 Funcionalidade das URLs

URLs desempenham um papel crucial na navegação na Web. Elas não apenas indicam ao navegador onde encontrar um recurso, mas também como acessar e interagir com ele. Por exemplo, uma URL pode incluir parâmetros que modificam a forma como o conteúdo é apresentado ou como uma pesquisa é realizada em um site.

1.4.3 URLs e SEO

As URLs também são um elemento importante na otimização para motores de busca (SEO). URLs claras, descritivas e bem estruturadas podem melhorar a indexação de uma página pelos motores de busca e aumentar a visibilidade do site.

1.4.4 Segurança e URLs

Com o crescimento da Web, a segurança das URLs tornou-se uma preocupação central. HTTPS, por exemplo, foi introduzido para proteger as comunicações entre o navegador e o servidor, criptografando os dados para evitar interceptações maliciosas.

1.5 Comunicação Bidirecional

1.5.1 Polling e Long Polling

Polling e Long Polling são técnicas usadas para manter uma comunicação bidirecional entre o cliente e o servidor, especialmente em aplicações que necessitam de atualizações frequentes.

- **Polling:** O cliente faz requisições periódicas ao servidor para verificar se há novas informações. Essa técnica pode ser ineficiente devido à quantidade de requisições feitas.
- **Long Polling:** O servidor mantém a conexão aberta até que haja novas informações para enviar ao cliente, reduzindo o número de requisições.

1.5.2 Server-Sent Events

Server-Sent Events (SSE) permitem que um servidor envie dados de atualização para o cliente de forma contínua, usando uma conexão HTTP única. Isso é útil para notificações em tempo real, feeds de dados e outras aplicações que necessitam de atualizações contínuas.

```
# Exemplo de código SSE no lado do servidor (Node.js)
const express = require('express');
const app = express();

app.get('/events', (req, res) => {
    res.setHeader('Content-Type', 'text/event-stream');
    res.setHeader('Cache-Control', 'no-cache');
    res.setHeader('Connection', 'keep-alive');

    setInterval(() => {
        res.write(`data: ${new Date().toISOString()}\n\n`);
    }, 1000);
});

app.listen(3000, () => console.log('SSE server running on port
```

1.5.3 WebSockets

Introdução ao WebSockets

WebSockets fornecem uma comunicação bidirecional em tempo real entre o cliente e o servidor sobre uma única conexão TCP. Diferente do HTTP, o WebSocket permite que tanto o cliente quanto o servidor enviem dados a qualquer momento, sem a necessidade de repetidas requisições HTTP.

Implementando WebSockets no Node.js

Para implementar WebSockets em Node.js, a biblioteca `ws` é uma escolha popular. A seguir, um exemplo básico de implementação:

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', ws => {
    ws.on('message', message => {
        console.log(`Received message => ${message}`);
    });
    ws.send('Hello! You are connected.');
});
```

Este exemplo cria um servidor WebSocket que ouve na porta 8080. Quando um cliente se conecta, o servidor pode enviar e receber mensagens em tempo real.

Capítulo 2

Usando APIs Web

2.1 Alterando Páginas Web Dinamicamente Usando a API DOM

2.1.1 O Modelo de Objeto de Documento (DOM)

O Modelo de Objeto de Documento, ou DOM (*Document Object Model*), é uma interface de programação que permite a manipulação de documentos HTML e XML como uma estrutura de árvore, onde cada nó representa uma parte do documento, como elementos, atributos ou texto. O DOM é essencial para a criação de páginas web dinâmicas, permitindo que os desenvolvedores interajam e modifiquem a estrutura e o conteúdo das páginas em tempo real.

2.1.2 Os Diferentes Tipos de Nós

No DOM, existem vários tipos de nós, cada um representando uma parte específica do documento:

- **Nós de Elemento:** Representam tags HTML, como `<div>`, `<p>`, `<a>`.
- **Nós de Atributo:** Representam os atributos de um elemento HTML, como `id`, `class`.
- **Nós de Texto:** Representam o conteúdo textual dentro de um elemento HTML.
- **Nós de Comentário:** Representam os comentários no código HTML.
- **Nós de Documento:** Representam o próprio documento inteiro, como a raiz da árvore DOM.

```
// Acessando um nó de elemento
let elementNode = document.getElementById('myElement');

// Acessando um nó de atributo
let attributeNode = elementNode.getAttributeNode('class');

// Acessando um nó de texto
let textNode = elementNode.firstChild;
```

2.1.3 Selecionando Elementos

Para modificar uma página web dinamicamente, primeiro precisamos selecionar os elementos que desejamos alterar. O DOM fornece várias maneiras de selecionar elementos:

- **getElementById**: Seleciona um único elemento pelo seu `id`.
- **getElementsByClassName**: Seleciona todos os elementos que possuem uma determinada classe.
- **getElementsByTagName**: Seleciona todos os elementos com uma determinada tag.
- **querySelector**: Seleciona o primeiro elemento que corresponde a um seletor CSS.
- **querySelectorAll**: Seleciona todos os elementos que correspondem a um seletor CSS.

```
// Exemplo de seleção de elementos
let element = document.getElementById('header');
let elements = document.querySelectorAll('.item');
```

2.1.4 Modificando Elementos

Depois de selecionar um elemento, podemos modificar suas propriedades, conteúdo, ou estilo diretamente através do DOM.

- **Modificando o Conteúdo:** Use `innerHTML` ou `textContent` para alterar o conteúdo interno de um elemento.
- **Alterando Estilos:** Modifique o estilo de um elemento através da propriedade `style`.
- **Adicionando/Removendo Classes:** Use `classList` para adicionar, remover ou alternar classes de um elemento.

```
// Exemplo de modificação de conteúdo e estilo
let element = document.getElementById('header');
element.textContent = 'Novo Título';
element.style.color = 'blue';
element.classList.add('novo-estilo');
```

2.1.5 Criando, Adicionando e Excluindo Elementos

O DOM permite que você crie novos elementos, adicione-os ao documento e remova elementos existentes.

- **Criando Elementos:** Use `document.createElement()` para criar um novo elemento.
- **Adicionando Elementos:** Use `appendChild()` ou `insertBefore()` para adicionar o novo elemento à árvore DOM.
- **Excluindo Elementos:** Use `removeChild()` ou `element.remove()` para remover um elemento.

```
// Exemplo de criação e adição de um novo elemento
let newElement = document.createElement('p');
newElement.textContent = 'Este é um novo parágrafo.';
document.body.appendChild(newElement);

// Exemplo de remoção de um elemento
let elementToRemove = document.getElementById('oldElement');
elementToRemove.remove();
```

2.1.6 Exemplo Prático: Criação Dinâmica de uma Tabela

Vamos criar dinamicamente uma tabela HTML utilizando JavaScript e a API DOM. Este exemplo prático ajudará a solidificar os conceitos discutidos.

```
// Criando a tabela e os elementos associados
let table = document.createElement('table');
let headerRow = document.createElement('tr');
let headers = ['Nome', 'Idade', 'Profissão'];

headers.forEach(headerText => {
    let header = document.createElement('th');
    header.textContent = headerText;
    headerRow.appendChild(header);
});
```

```
table.appendChild(headerRow);

// Adicionando linhas de dados à tabela
let data = [
    ['Alice', 30, 'Engenheira de Software'],
    ['Bob', 25, 'Designer'],
    ['Carol', 28, 'Gerente de Projetos']
];

data.forEach(rowData => {
    let row = document.createElement('tr');
    rowData.forEach(cellData => {
        let cell = document.createElement('td');
        cell.textContent = cellData;
        row.appendChild(cell);
    });
    table.appendChild(row);
});

// Adicionando a tabela ao corpo do documento
document.body.appendChild(table);
```

2.2 Carregando Dados Sincronamente via Ajax e a API Fetch

2.2.1 Comunicação Sincrona versus Assíncrona

A comunicação síncrona envolve a execução de tarefas de maneira sequencial, onde uma tarefa deve ser concluída antes que a próxima comece. Em contrapartida, a comunicação assíncrona permite que as tarefas sejam iniciadas sem esperar pela conclusão das anteriores, o que é crucial para melhorar a performance e a experiência do usuário em aplicações web.

2.2.2 Carregando Dados via Ajax

Ajax (Asynchronous JavaScript and XML) é uma técnica amplamente usada para carregar dados de forma assíncrona sem recarregar a página inteira. Ajax permite que as aplicações web se tornem mais interativas e responsivas.

```
// Exemplo básico de uma requisição Ajax usando XMLHttpRequest
let xhr = new XMLHttpRequest();
xhr.open('GET', 'https://api.example.com/data', true);
```

```
xhr.onload = function() {
    if (xhr.status === 200) {
        console.log(xhr.responseText);
    } else {
        console.error('Erro na requisição.');
    }
};

xhr.send();
```

2.2.3 Carregando Dados via a API Fetch

Visão Geral das APIs Web

A API Fetch é uma API moderna e baseada em Promises para fazer requisições HTTP assíncronas. Diferente do XMLHttpRequest, Fetch proporciona uma maneira mais simples e poderosa de buscar recursos através da rede.

```
// Exemplo básico de uma requisição com a Fetch API
fetch('https://api.example.com/data')
    .then(response => {
        if (!response.ok) {
            throw new Error('Erro na requisição');
        }
        return response.json();
    })
    .then(data => console.log(data))
    .catch(error => console.error('Erro:', error));
```

Suporte dos Navegadores para APIs Web

Embora a API Fetch seja amplamente suportada pelos navegadores modernos, é importante considerar o suporte para navegadores legados ao desenvolver aplicações web. O uso de polyfills pode ser uma solução para garantir compatibilidade.

Capítulo 3

JavaScript no Lado do Servidor com Node.js

3.1 Introdução ao Node.js

Arquitetura do Node.js

- Arquitetura Orientada a Eventos no Node.js
- I/O Não Bloqueante

Um Primeiro Programa

- Construindo um Servidor Web Simples

Gerenciamento de Pacotes com npm

- Instalando e Gerenciando Pacotes
- Criando e Publicando seu Próprio Pacote

3.2 Utilizando Módulos Nativos do Node.js

Trabalhando com o Sistema de Arquivos

- Lendo Arquivos
- Escrevendo Arquivos
- Deletando Arquivos

Trabalhando com Buffers e Streams

Rede

- Criando Servidores TCP e HTTP
- Trabalhando com Sockets

3.3 Construindo Servidores Web

Preparando o Ambiente de Desenvolvimento

Fornecendo Arquivos Estáticos

Usando o Framework Express.js

- Configurando o Express.js
- Roteamento no Express.js
- Middleware no Express.js

Processando Dados de Formulários

Usando Motores de Template com Express.js

- Pug, Mustache, e EJS

3.3.1 Extra: Formatos Web

- Formatos de dados: CSV, XML e JSON
- Formatos de imagem: JPG, GIF, PNG, SVG, WebP
- Comprando formatos de imagem
- Programas para processar imagens
- Formatos de audio e vídeo

Capítulo 4

Implementando Serviços Web

4.1 Introdução aos Serviços Web

Visão Geral dos Serviços Web

4.2 Serviços Web RESTful

Princípios do REST

Implementando uma API REST

- Usando Express.js para Criar Endpoints RESTful
- Manipulando Operações CRUD

Consumindo uma API REST

- Consumindo APIs RESTful com JavaScript
- Tratamento de Erros e Melhores Práticas

4.3 GraphQL

As Desvantagens do REST

Introdução ao GraphQL

Implementando GraphQL com Node.js

- Configurando o GraphQL no Node.js

- Escrevendo Queries e Mutations
- GraphQL vs REST

Capítulo 5

Armazenando Dados em Bancos de Dados

5.1 Bancos de Dados Relacionais

Entendendo Bancos de Dados Relacionais

A Linguagem SQL

- Consultas SQL Básicas (SELECT, INSERT, UPDATE, DELETE)
- Joins, Subconsultas e Transações

Usando MySQL com Node.js

- Configurando MySQL no Node.js
- Realizando Operações CRUD com MySQL

Mapeamento Objeto-Relacional (ORM)

- Usando Sequelize com Node.js

5.2 Bancos de Dados Não-Relacionais

Bancos de Dados Relacionais vs Não-Relacionais

Visão Geral dos Bancos de Dados Não-Relacionais

- Bancos de Dados de Chave-Valor

- Bancos de Dados Orientados a Documentos
- Bancos de Dados de Grafos
- Bancos de Dados Colunares

Usando MongoDB com Node.js

- Configurando MongoDB no Node.js
- Operações CRUD com MongoDB
- Framework de Agregação no MongoDB

Comparando MySQL e MongoDB

Capítulo 6

Arquiteturas Web

6.1 Arquiteturas em Camadas

Estrutura Básica das Arquiteturas em Camadas

Arquitetura Cliente-Servidor (Arquitetura de Dois Níveis)

Arquitetura de Múltiplos Níveis

- Arquitetura N-Camadas
- Vantagens e Desvantagens
- Other Web APIs:

6.2 Arquiteturas Monolíticas e Distribuídas

Arquitetura Monolítica

Arquitetura Orientada a Serviços (SOA)

Arquitetura de Microsserviços

- Comparando SOA e Microsserviços
- Implementando Microsserviços com Node.js

Arquitetura Baseada em Componentes

Arquitetura de Microfrontends

Arquitetura de Mensagens

- Arquitetura Orientada a Eventos
- Filas de Mensagens e Brokers (RabbitMQ, Kafka)

Arquitetura de Serviços Web

- SOAP vs REST
- Implementando Serviços Web com Node.js

6.3 Arquiteturas MV*

Model-View-Controller (MVC)

- Implementando MVC com Express.js

Model-View-Presenter (MVP)

Model-View-ViewModel (MVVM)

Capítulo 7

Padrões de Projeto

7.1 Introdução aos Padrões de Design

O que são Padrões de Design?

- Importância dos Padrões de Design em Desenvolvimento de Software
- Categorias de Padrões de Design: Criacionais, Estruturais e Comportamentais

7.2 Padrões Criacionais

7.2.1 Singleton

7.2.2 Factory Method

7.2.3 Abstract Factory

7.2.4 Builder

7.2.5 Prototype

- Implementando Padrões Criacionais com JavaScript/Node.js

7.3 Padrões Estruturais

7.3.1 Adapter

7.3.2 Decorator

7.3.3 Facade

7.3.4 Proxy

7.3.5 Composite

7.3.6 Flyweight

7.3.7 Bridge

- Implementando Padrões Estruturais com JavaScript/Node.js

7.4 Padrões Comportamentais

7.4.1 Observer

7.4.2 Strategy

7.4.3 Command

7.4.4 Chain of Responsibility

7.4.5 Template Method

7.4.6 State

7.4.7 Iterator

7.4.8 Mediator

- Implementando Padrões Comportamentais com JavaScript/Node.js

Capítulo 8

Testando Aplicações Web

8.1 Introdução aos Testes

A Importância dos Testes no Desenvolvimento Web

8.2 Testes Unitários

Escrevendo Testes Unitários em JavaScript

- Usando Mocha e Chai
- Desenvolvimento Orientado por Testes (TDD)

8.3 Testes de Integração

Testando APIs RESTful

- Usando Supertest com Mocha

8.4 Testes de Ponta a Ponta (E2E)

Automatizando Testes de Ponta a Ponta

- Usando Cypress para Testes E2E

Capítulo 9

Implantação e Hospedagem de Aplicações Web

9.1 Preparando para Implantação

Processo de Build e Release

Variáveis de Ambiente e Configurações

9.2 Implantando Aplicações

Implantando em Provedores de Nuvem

- AWS, Azure e Google Cloud Platform

Usando Docker para Implantação

- Containerização com Docker
- Dockerizando Aplicações Node.js

Usando Pipelines CI/CD

- Automatizando a Implantação com Jenkins, GitHub Actions e GitLab CI

Capítulo 10

Segurança em Aplicações Web

10.1 Compreendendo a Segurança Web

Ameaças Comuns à Segurança

- Injeção de SQL
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)

10.2 Implementando Segurança no Node.js

Usando Helmet para Cabeçalhos de Segurança

Protegendo Aplicações Express

- Validação e Sanitização de Entrada
- Limitação de Taxa e Proteção contra DDoS

Autenticação e Autorização

- Implementando OAuth e JWT

Capítulo 11

Otimizando a Performance de Aplicações Web

11.1 Identificando Gargalos de Performance

Ferramentas de Monitoramento e Profiling

- Usando PM2 e New Relic

11.2 Técnicas de Otimização de Performance

Estratégias de Cache

- Usando Redis para Cache

Balanceamento de Carga

Otimizando Consultas ao Banco de Dados

- Indexação e Otimização de Consultas

Otimização de Código

- Vazamentos de Memória e Coleta de Lixo no Node.js
- Otimização de Programação Assíncrona e do Event Loop

APÊNDICE A

O Ambiente de Desenvolvimento

Este apêndice descreve as etapas necessárias para configurar o ambiente de desenvolvimento utilizado ao longo deste livro. O ambiente será configurado no sistema operacional Linux Ubuntu, utilizando o editor de código Visual Studio Code (VS Code), a linguagem Node.js, o framework Express, e os bancos de dados MySQL e MongoDB.

A.1 Instalando o Ubuntu

- Certifique-se de que você tenha a versão mais recente do Ubuntu instalada em sua máquina. Você pode baixar a versão mais recente do Ubuntu a partir do site oficial: <https://ubuntu.com/download>.
- Siga as instruções para instalar o Ubuntu, configurando o particionamento do disco e outras opções conforme necessário.
- Após a instalação, execute o comando abaixo para garantir que todos os pacotes estejam atualizados:

```
sudo apt update && sudo apt upgrade -y
```

A.2 Instalando o Visual Studio Code (VS Code)

- Baixe e instale o Visual Studio Code (VS Code), um editor de código leve, mas poderoso, ideal para o desenvolvimento em Node.js.

- Execute o seguinte comando para instalar o VS Code via linha de comando:

```
sudo snap install --classic code
```

- Após a instalação, abra o VS Code e configure as extensões recomendadas para desenvolvimento em Node.js:
  - **Node.js Extension Pack**
  - **Prettier - Code Formatter**
  - **ESLint**

A.3 Instalando o Node.js e npm

- O Node.js é uma plataforma JavaScript que permite executar código JavaScript no lado do servidor.
- Para instalar o Node.js e o npm (Node Package Manager) em Ubuntu, use o seguinte comando:

```
sudo apt install nodejs npm
```

- Verifique a instalação executando:

```
node -v
npm -v
```

  Esses comandos devem retornar as versões instaladas do Node.js e do npm.

A.4 Instalando o Framework Express

- O Express é um framework web para Node.js, amplamente utilizado para construir aplicações web robustas e escaláveis.
- Para instalar o Express globalmente, execute:

```
npm install -g express-generator
```

- Para criar uma nova aplicação Express, use:

```
express nome-do-projeto
```

- Navegue até o diretório do projeto e instale as dependências:

```
cd nome-do-projeto
npm install
```

A.5 Instalando o MySQL

- O MySQL é um sistema de gerenciamento de banco de dados relacional que será utilizado para armazenar dados estruturados.
- Para instalar o MySQL, execute:

```
sudo apt install mysql-server
```

- Após a instalação, inicie o serviço MySQL e configure a segurança inicial:

```
sudo systemctl start mysql
sudo mysql_secure_installation
```

- Conecte-se ao MySQL utilizando:

```
sudo mysql -u root -p
```

A.6 Instalando o MongoDB

- O MongoDB é um banco de dados NoSQL orientado a documentos que será utilizado para armazenar dados não estruturados.
- Para instalar o MongoDB, execute os seguintes comandos:

```
sudo apt install -y mongodb
```

- Inicie o serviço MongoDB:

```
sudo systemctl start mongodb
```

- Verifique se o MongoDB está funcionando corretamente:

```
sudo systemctl status mongodb
```

- Para interagir com o MongoDB, utilize o cliente de linha de comando:

```
mongo
```

A.7 Configurando o Ambiente de Desenvolvimento

- **Configuração do Ambiente Node.js no VS Code:**
  - Abra o VS Code e carregue o projeto Node.js.
  - Configure o arquivo `launch.json` no VS Code para depuração de aplicações Node.js.
- **Conectando ao MySQL e MongoDB:**
  - Para se conectar ao MySQL em uma aplicação Node.js, utilize o pacote `mysql`:

```
npm install mysql
```

  - Para se conectar ao MongoDB, utilize o pacote `mongoose`:

```
npm install mongoose
```

A.8 Testando a Configuração

- Para garantir que tudo está configurado corretamente, crie um pequeno projeto Node.js que se conecta ao MySQL e ao MongoDB, e execute-o usando o VS Code.
- Verifique se o projeto pode iniciar o servidor Express e realizar operações básicas de CRUD (Create, Read, Update, Delete) nos bancos de dados MySQL e MongoDB.
- Abaixo está um exemplo básico de uma aplicação Node.js que se conecta ao MySQL:

```
const mysql = require('mysql');
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'password',
  database: 'test'
});

connection.connect((err) => {
  if (err) throw err;
  console.log('Connected to MySQL!');
});

connection.query('SELECT * FROM users', (err, results)
  if (err) throw err;
  console.log(results);
});
```

- E um exemplo para conexão com o MongoDB:

```
const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/test', {use

const db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection
db.once('open', () => {
  console.log('Connected to MongoDB!');
});
```

APÊNDICE B

Tutorial Básico de Linhas de Comando no Linux

Este apêndice oferece um tutorial básico sobre como utilizar a linha de comando no shell do Linux. Ele abrange desde comandos simples para manipulação de arquivos e diretórios até comandos para acessar a rede e usar ferramentas como o cURL para interagir com APIs.

B.1 Comandos Básicos de Arquivos e Diretórios

B.1.1 Criando uma Pasta

Para criar uma nova pasta (diretório), utilize o comando `mkdir`:

```
mkdir nome_da_pasta
```

Este comando criará um diretório chamado `nome_da_pasta` no local atual.

B.1.2 Navegando até uma Pasta

Para navegar até um diretório específico, use o comando `cd` (change directory):

```
cd nome_da_pasta
```

Para voltar ao diretório anterior, você pode usar:

```
cd ..
```

B.1.3 Criando um Arquivo HTML

Para criar um novo arquivo HTML, você pode utilizar o comando `touch`:

```
touch index.html
```

Este comando criará um arquivo vazio chamado `index.html` no diretório atual.

B.1.4 Listando Arquivos e Diretórios

Para listar todos os arquivos e diretórios no diretório atual, use o comando `ls`:

```
ls
```

Você pode adicionar opções ao comando `ls` para listar arquivos com mais detalhes:

```
ls -l      # Lista em formato longo, incluindo permissões e taman
ls -a      # Lista todos os arquivos, incluindo os ocultos (começ
ls -lh     # Lista com tamanhos de arquivo em um formato legível
```

B.1.5 Procurando Arquivos

Para procurar arquivos no sistema, use o comando `find`:

```
find /caminho/para/diretorio -name "nome_do_arquivo"
```

Este comando pesquisa recursivamente a partir do diretório especificado por arquivos que correspondam ao nome fornecido.

B.1.6 Verificando o Histórico de Comandos

Para ver o histórico dos comandos que você executou, utilize o comando `history`:

```
history
```

Este comando exibe uma lista de todos os comandos que você executou na sessão atual do terminal.

B.2 Comandos de Rede e Acesso a Sites Web

B.2.1 Verificando a Configuração de Rede

Para verificar a configuração de rede da sua máquina, use o comando `ifconfig` (ou `ip` nas versões mais recentes do Ubuntu):

```
ifconfig        # Exibe as interfaces de rede e suas configura
ip addr show    # Comando alternativo mais moderno para mostra
```

B.2.2 Acessando Sites Web

Para acessar e recuperar dados de um site a partir da linha de comando, você pode usar o comando `wget`:

```
wget https://www.exemplo.com
```

Este comando baixa o conteúdo da URL especificada para o diretório atual.

B.3 Usando o cURL para Acessar APIs

B.3.1 Introdução ao cURL

O cURL é uma ferramenta de linha de comando que permite realizar requisições a URLs, sendo particularmente útil para acessar APIs RESTful.

B.3.2 Realizando uma Requisição GET

Para realizar uma requisição GET simples a uma API, use o seguinte comando:

```
curl https://api.exemplo.com/dados
```

Este comando envia uma requisição GET para a URL especificada e exibe a resposta no terminal.

B.3.3 Enviando Dados com uma Requisição POST

Para enviar dados em uma requisição POST, você pode usar o cURL da seguinte forma:

```
curl -X POST https://api.exemplo.com/usuario \
     -H "Content-Type: application/json" \
     -d '{"nome": "João", "idade": 30}'
```

Neste exemplo, o comando envia um objeto JSON contendo os dados do usuário para a API.

B.3.4 Autenticação com cURL

Se a API requer autenticação, você pode adicionar um cabeçalho de autorização à sua requisição:

```
curl -H "Authorization: Bearer seu_token_aqui" \
     https://api.exemplo.com/dados_protegidos
```

Este comando envia uma requisição GET autenticada com um token de acesso.

B.4 Outros Comandos Úteis

- **pwd:** Mostra o diretório atual (o caminho completo até onde você está).

  ```
  pwd
  ```

- **rm:** Remove (exclui) arquivos ou diretórios.

  ```
  rm nome_do_arquivo
  rm -r nome_do_diretorio  # Remove um diretório e todo o s
  ```

- **cp:** Copia arquivos ou diretórios.

  ```
  cp arquivo_origem arquivo_destino
  cp -r diretorio_origem diretorio_destino  # Copia diretór
  ```

- **mv:** Move ou renomeia arquivos ou diretórios.

  ```
  mv arquivo_origem arquivo_destino
  mv nome_antigo nome_novo  # Renomeia um arquivo ou diretó
  ```

- **man:** Exibe o manual de um comando, útil para aprender sobre suas opções e uso.

  ```
  man comando
  ```

Referências Bibliográficas

CERN. *A short history of the Web*. 2024. <https://home.cern/science/computing/birth-web/short-history-web>. [Accessed: 13-Aug-2024]. Disponível em: <https://home.cern/science/computing/birth-web/short-history-web>.