Desenvolvimento Web com JavaScript e MySQL

Capítulo 1: Métodos de Requisição HTTP

O que é HTTP?

HTTP (Hypertext Transfer Protocol) é o protocolo que permite a comunicação entre clientes (como navegadores web) e servidores.

Ele define um conjunto de métodos de requisição que indicam a ação desejada para um recurso específico.

Aqui, vamos explorar os métodos mais comuns: GET, POST, PUT, PATCH e DELETE.

Métodos de Requisição

GET

Definição: O método GET solicita a representação de um recurso específico.

As requisições usando GET devem apenas recuperar dados.

Quando usar: Quando você quer obter informações de um servidor sem alterar seu estado.

Exemplo Prático: Imagine que você queira visualizar uma lista de usuários em um site. O navegador envia uma requisição GET para o servidor, solicitando essa lista.

Código em JavaScript:

```
fetch('https://api.example.com/users')
  .then(response => response.json())
  .then(data => console.log(data));
```

POST

Definição: O método POST é usado para enviar dados ao servidor para criar um novo recurso.

Quando usar: Quando você precisa enviar dados para o servidor, como ao enviar um formulário de cadastro.

Exemplo Prático: Você preenche um formulário para se registrar em um site. Quando você clica em "Enviar", o navegador envia uma requisição POST com seus dados.

Código em JavaScript:

```
fetch('https://api.example.com/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ name: 'John Doe', age: 30 })
})
.then(response => response.json())
.then(data => console.log(data));
```

PUT

Definição: O método PUT substitui todas as representações atuais do recurso de destino pelos dados da requisição.

Quando usar: Quando você quer atualizar completamente um recurso existente.

Exemplo Prático: Se você deseja atualizar seu perfil em um site, como mudar seu nome e idade, uma requisição PUT seria apropriada.

Código em JavaScript:

```
fetch('https://api.example.com/users/1', {
  method: 'PUT',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ name: 'John Smith', age: 31 })
})
.then(response => response.json())
.then(data => console.log(data));
```

PATCH

Definição: O método PATCH é usado para aplicar modificações parciais a um recurso.

Quando usar: Quando você quer atualizar parcialmente um recurso.

Exemplo Prático: Se você só precisa alterar a idade de um usuário sem modificar outros dados, utilize o PATCH.

Código em JavaScript:

```
fetch('https://api.example.com/users/1', {
  method: 'PATCH',
  headers: {
```

```
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ age: 32 })
})
.then(response => response.json())
.then(data => console.log(data));
```

DELETE

Definição: O método DELETE remove um recurso específico do servidor.

Quando usar: Quando você quer remover um recurso existente.

Exemplo Prático: Se você deseja excluir sua conta de um site, uma requisição DELETE é enviada.

Código em JavaScript:

```
fetch('https://api.example.com/users/1', {
  method: 'DELETE'
})
.then(response => response.json())
.then(data => console.log(data));
```

Capítulo 2: Padrão Model View Control (MVC)

O que é MVC?

O padrão Model-View-Controller (MVC) é uma forma de organizar o código de uma aplicação de maneira a separar a lógica de negócios, a interface com o usuário e o controle da aplicação.

Essa separação ajuda a tornar o código mais modular, fácil de manter e escalável.

Componentes do MVC

Model

Definição: O Model representa os dados da aplicação e a lógica de negócios.

Função: Gerenciar os dados e garantir que as regras de negócios sejam aplicadas corretamente.

Exemplo Prático:

```
class User {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}
```

View

Definição: A View é responsável pela apresentação dos dados ao usuário.

Função: Exibir os dados de maneira adequada e intuitiva para o usuário.

Exemplo Prático:

```
function displayUser(user) {
  console.log(`User: ${user.name}, Age: ${user.age}`);
}
```

Controller

Definição: O Controller atua como um intermediário entre o Model e a View. Ele processa a entrada do usuário e interage com o Model para atualizar a View.

Função: Controlar o fluxo de dados entre o Model e a View, e lidar com as requisições do usuário.

Exemplo Prático:

```
class UserController {
  constructor(view) {
    this.view = view;
  }

  createUser(name, age) {
    const user = new User(name, age);
    this.view.displayUser(user);
  }
}

const view = { displayUser };
const controller = new UserController(view);
controller.createUser('John Doe', 30);
```

Diagrama MVC

Capítulo 3: Organização de Arquitetura de Sistemas

O que é Arquitetura de Sistemas?

Arquitetura de sistemas refere-se à estrutura organizacional de um sistema de software, incluindo seus componentes e a forma como eles interagem. A escolha da arquitetura correta é crucial para garantir que o sistema seja escalável, mantenível e eficiente.

Tipos Comuns de Arquitetura

Arquitetura Monolítica

Definição: Uma arquitetura onde todos os componentes de uma aplicação são integrados em uma única aplicação.

Vantagens: Simplicidade na implementação e no desenvolvimento inicial.

Desvantagens: Dificuldade em escalar e manter à medida que a aplicação cresce.

Arquitetura de Microservices

Definição: Uma arquitetura onde a aplicação é dividida em pequenos serviços independentes que se comunicam entre si.

Vantagens: Escalabilidade, facilidade de manutenção e desenvolvimento paralelo.

Desvantagens: Complexidade na gestão e comunicação entre serviços.

Comparação Visual

Capítulo 4: APIs (Application Programming Interfaces)

O que é uma API?

Uma API (Interface de Programação de Aplicações) é um conjunto de regras e definições que permitem que diferentes sistemas de software se comuniquem.

As APIs definem a maneira como as requisições e respostas devem ser formatadas.

Tipos de APIs

REST

Definição: REST (Representational State Transfer) é um estilo de arquitetura que utiliza os métodos HTTP e é baseado em recursos.

Vantagens: Simplicidade e escalabilidade.

Desvantagens: Pode não ser adequado para operações complexas.

SOAP

Definição: SOAP (Simple Object Access Protocol) é um protocolo de mensagens baseado em XML.

Vantagens: Segurança e confiabilidade.

Desvantagens: Complexidade e maior sobrecarga.

Aplicações de APIs

- **RESTful APIs**: São amplamente utilizadas para criar serviços web que são simples de usar com HTTP.
- **GraphQL**: Uma linguagem de consulta para APIs que permite que os clientes solicitem exatamente os dados de que precisam.

Protocolo de Comunicação

- **HTTP/HTTPS**: Protocolo usado para comunicação de APIs, onde HTTPS é a versão segura do HTTP.

Capítulo 5: Metodologias Ágeis para Desenvolvimento de APIs

O que são Metodologias Ágeis?

Metodologias ágeis são práticas de desenvolvimento de software que promovem a entrega incremental e a colaboração contínua entre equipes.

Exemplos de Metodologias Ágeis

Scrum

Definição: Uma metodologia ágil que divide o trabalho em sprints (curtos períodos de tempo) e inclui reuniões diárias para discutir o progresso.

Kanban

Definição: Usa um quadro visual para gerenciar tarefas, permitindo que a equipe visualize o fluxo de trabalho e identifique gargalos.

Capítulo 6: Linguagens de Programação para APIs

Linguagens Comuns

JavaScript (Node.js)

Uso: Muito utilizado para desenvolvimento de APIs devido à sua simplicidade e ao vasto ecossistema de pacotes.

Exemplo:

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

```
const app = express();

app.get('/users', (req, res) => {
  res.json([{ name: 'John Doe', age: 30 }]);
});

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

Python (Flask/Django)

Uso: Conhecido por sua clareza e simplicidade, Python é uma escolha popular para desenvolvimento de APIs.

Exemplo com Flask:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/users', methods=['GET'])
def get_users():
    return jsonify([{'name': 'John Doe', 'age': 30}])

if __name__ == '__main__':
    app.run(debug=True)
```

Java (Spring Boot)

Uso: Amplamente usado em grandes empresas devido à sua robustez e segurança.

Exemplo:

```
@RestController
public class UserController {

    @GetMapping("/users")
    public List<User> getUsers() {
        return Arrays.asList(new User("John Doe", 30));
    }
}
```

Capítulo 7: Funcionalidades para APIs

Autenticação e Autorização

Definição: Autenticação verifica a identidade do usuário, enquanto autorização controla o acesso aos recursos.

Exemplo: Usar tokens JWT (JSON Web Tokens) para autenticação.

Validação de Dados

Definição: Garantir que os dados recebidos pela API estão corretos e seguem as regras definidas.

Exemplo: Utilizar bibliotecas como Joi em Node.js para validação de dados.

```
const Joi = require('joi');

const schema = Joi.object({
  name: Joi.string().min(3).required(),
  age: Joi.number().integer().min(0)
});

const result = schema.validate({ name: 'John', age: 30 });
if (result.error) {
  console.error(result.error.details);
}
```

Capítulo 8: Técnicas de Depuração

Console Logs

Definição: Imprimir mensagens no console para inspecionar o comportamento do código.

Exemplo:

```
console.log('This is a debug message');
```

Debuggers

Definição: Ferramentas que permitem inspecionar o código em tempo de execução.

Exemplo: Chrome DevTools.

Capítulo 9: Documentação do Sistema

Ferramentas de Documentação

Swagger

Definição: Ferramenta para documentar APIs RESTful de forma interativa.

Exemplo:

```
openapi: 3.0.0
info:
  title: API de Exemplo
  version: 1.0.0
paths:
  /users:
    get:
      summary: Retorna uma lista de usuários
      responses:
        '200':
          description: Lista de usuários
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
                  properties:
                    name:
                      type: string
                    age:
                      type: integer
```

JSDoc

Definição: Usado para documentar código JavaScript.

Exemplo:

```
/**
 * Função para somar dois números
 * @param {number} a - O primeiro número
 * @param {number} b - O segundo número
 * @returns {number} A soma dos dois números
 */
function sum(a, b) {
  return a + b;
}
```

Capítulo 10: Técnicas de Programação e Controle

Modularização

Definição: Dividir o código em módulos menores e reutilizáveis.

Exemplo:

```
// user.js
class User {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}

module.exports = User;

// main.js
const User = require('./user');
const user = new User('John Doe', 30);
console.log(user);
```

Controle de Versão

Definição: Uso de ferramentas como Git para gerenciar mudanças no código.

Exemplo:

```
git init
git add .
git commit -m "Initial commit"
```

Capítulo 11: Frameworks

Frameworks Populares

Express.js

Definição: Framework para Node.js que facilita a criação de servidores web e APIs.

Exemplo:

```
const express = require('express');
const app = express();

app.get('/users', (req, res) => {
  res.json([{ name: 'John Doe', age: 30 }]);
});

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

Koa.js

Definição: Outro framework para Node.js, mais leve e flexível.

Exemplo:

```
const Koa = require('koa');
const app = new Koa();

app.use(async ctx => {
  ctx.body = [{ name: 'John Doe', age: 30 }];
});

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

Capítulo 12: Status de Respostas

Códigos de Status Comuns

- **200 OK**: Requisição bem-sucedida.
- **201 Created**: Recurso criado com sucesso.
- **400 Bad Request**: Requisição inválida.
- **401 Unauthorized**: Autenticação necessária.
- **404 Not Found**: Recurso não encontrado.
- **500 Internal Server Error**: Erro no servidor.

Capítulo 13: Tratamento de Exceções

Try/Catch

Definição: Blocos de código para capturar e tratar erros.

Exemplo:

```
try {
  // Código que pode lançar um erro
  throw new Error('Something went wrong');
} catch (error) {
  console.error(error.message);
}
```

Capítulo 14: Técnicas de Formato de Comunicação

REST e GraphQL

REST

Definição: Estilo de arquitetura para criar APIs, usando métodos HTTP.

Exemplo:

```
app.get('/users', (req, res) => {
  res.json([{ name: 'John Doe', age: 30 }]);
});
```

GraphQL

Definição: Linguagem de consulta para APIs que permite que os clientes solicitem exatamente os dados de que precisam.

Exemplo:

```
const { graphql, buildSchema } = require('graphql');

const schema = buildSchema(`
  type Query {
    user(id: Int!): User
  }

  type User {
    id: Int
    name: String
    age: Int
  }
`);

const root = {
  user: ({ id }) => {
    return { id, name: 'John Doe', age: 30 };
  }
};

graphql(schema, '{ user(id: 1) { name, age } }', root).then(response
=> {
  console.log(response);
});
```

Capítulo 15: Formatos e Requisição

XML e JSON

XML

Definição: Linguagem de marcação para estruturar dados.

Exemplo:

```
xml

<user>
  <name>John Doe</name>
  <age>30</age>
</user>
```

JSON

Definição: Formato de dados leve e fácil de ler/escrever.

Exemplo:

```
json

{
  "name": "John Doe",
  "age": 30
}
```

Capítulo 16: Iniciando uma API com Node.js e MySQL

Introdução

Neste capítulo, vamos aprender a criar uma API básica utilizando Node.js e MySQL. Vamos abordar desde a configuração do ambiente até a criação e consulta de dados no banco de dados.

Pré-requisitos

Antes de começarmos, certifique-se de ter o seguinte instalado:

1. **Node.js**: Plataforma JavaScript que permite a execução de código no servidor.
2. **MySQL**: Sistema de gerenciamento de banco de dados relacional.
3. **MySQL Workbench**: Ferramenta gráfica para gerenciar bancos de dados MySQL.
4. **Postman**: Ferramenta para testar APIs.

Configurando o Ambiente

Instalando Node.js

Baixe e instale o Node.js a partir do site oficial.

Configurando o Projeto

1. Crie uma pasta para o seu projeto e navegue até ela no terminal:

```
mkdir my-api
cd my-api
```

2. Inicialize um novo projeto Node.js:

```
npm init -y
```

3. Instale as dependências necessárias:

```
npm install express mysql2 body-parser
```

- **express**: Framework web para Node.js.
- **mysql2**: Conector MySQL para Node.js.
- **body-parser**: Middleware para processar corpos de requisição.

Estrutura do Projeto

A estrutura básica do nosso projeto será a seguinte:

```
my-api/
├── node_modules/
├── package.json
├── package-lock.json
├── server.js
├── config/
│   └── database.js
└── routes/
    └── users.js
```

Configurando o Banco de Dados

Criando o Banco de Dados

Abra o MySQL Workbench e crie um novo banco de dados:

```
CREATE DATABASE myapi;
USE myapi;

CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL
);
```

Configurando a Conexão com o Banco de Dados

Crie o arquivo `config/database.js`:

javascript

```
const mysql = require('mysql2');

const pool = mysql.createPool({
  host: 'localhost',
```

```
  user: 'root',
  password: 'password',
  database: 'myapi'
});

module.exports = pool.promise();
```

Criando a API

Configurando o Servidor

Crie o arquivo `server.js`:

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
const port = 3000;

app.use(bodyParser.json());

// Rota para usuários
const userRoutes = require('./routes/users');
app.use('/users', userRoutes);

app.listen(port, () => {
  console.log(`Server running on port ${port}`);
});
```

Criando as Rotas

Crie o arquivo `routes/users.js`:

```
const express = require('express');
const router = express.Router();
const pool = require('../config/database');

// Rota para obter todos os usuários
router.get('/', async (req, res) => {
  try {
    const [rows, fields] = await pool.query('SELECT * FROM users');
    res.json(rows);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// Rota para adicionar um novo usuário
router.post('/', async (req, res) => {
  const { name, email } = req.body;
  try {
    const [result] = await pool.query('INSERT INTO users (name, email)
VALUES (?, ?)', [name, email]);
    res.status(201).json({ id: result.insertId, name, email });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

```
// Rota para atualizar um usuário existente
router.put('/:id', async (req, res) => {
  const { id } = req.params;
  const { name, email } = req.body;
  try {
    await pool.query('UPDATE users SET name = ?, email = ? WHERE id =
?', [name, email, id]);
    res.json({ message: 'User updated successfully' });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// Rota para deletar um usuário
router.delete('/:id', async (req, res) => {
  const { id } = req.params;
  try {
    await pool.query('DELETE FROM users WHERE id = ?', [id]);
    res.json({ message: 'User deleted successfully' });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

module.exports = router;
```

Testando a API

Com o servidor configurado e as rotas criadas, agora podemos iniciar o servidor e testar a API.

1. Inicie o servidor:

```
node server.js
```

2. Use o Postman para testar as seguintes requisições:

- **GET /users**: Retorna todos os usuários.
- **POST /users**: Adiciona um novo usuário. Corpo da requisição:

json

```
{
  "name": "Jane Doe",
  "email": "jane@example.com"
}
```

- **PUT /users/**

  : Atualiza um usuário existente. Corpo da requisição:

json

```
{
  "name": "Jane Smith",
  "email": "jane.smith@example.com"
```