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));
    }
}
```