Outras possibilidades de API RESTful

Introdução

Bem-vindos à nossa aula sobre "Outras possibilidades de API RESTful" na disciplina Programação II. Neste módulo, exploraremos aspectos avançados e alternativas no desenvolvimento de APIs RESTful, além das tradicionais abordagens. Iniciaremos com uma comparação entre as implementações de APIs RESTful em Node.js e Java, destacando as peculiaridades e benefícios de cada ambiente de desenvolvimento. Aprofundaremos na arquitetura de microsserviços, evidenciando como esta estrutura se alinha com as APIs RESTful para criar sistemas mais robustos e escaláveis. Abordaremos também a importância crítica dos testes e da qualidade de software, enfatizando como a prática de testar pode significativamente impactar a confiabilidade e a performance das aplicações. Por fim, investigaremos a utilização do Postman para testes de APIs, proporcionando uma visão prática e integrada do ciclo de desenvolvimento e manutenção de APIs RESTful.

Comparação da API RESTful NodeJS e Java

Na primeira parte da nossa aula, exploraremos a comparação entre as APIs RESTful desenvolvidas em NodeJS e em Java. Esse tema é essencial para compreender as nuances e especificidades de cada ambiente de desenvolvimento, fornecendo uma base sólida para o desenvolvimento de aplicações web modernas.

Iniciaremos com uma visão geral das APIs RESTful, destacando sua importância e papel no cenário atual da programação web. As APIs RESTful, representativas do estilo arquitetônico REST, são fundamentais para a

construção de serviços web que são escaláveis, performáticos e facilmente integráveis.

Ao compararmos o NodeJS e o Java no contexto de APIs RESTful, é crucial entender que ambos trazem abordagens distintas. Por exemplo, no NodeJS, podemos desenvolver uma API para gerenciar tarefas usando o framework Express, que permite a criação de rotas de forma ágil e eficiente. Em Java, utilizando o Spring Boot, podemos construir uma API similar com uma abordagem mais declarativa, onde aspectos como segurança e transações são gerenciados integradamente, demonstrando a robustez da plataforma. O NodeJS, por ser baseado em JavaScript, oferece uma experiência de desenvolvimento ágil e é frequentemente associado a um ciclo de desenvolvimento mais rápido, devido à sua natureza não bloqueante e orientada a eventos. Isso se traduz em uma performance otimizada em operações de I/O, tornando-o ideal para aplicações que requerem alta escalabilidade e tratamento eficiente de múltiplas conexões simultâneas.

Por outro lado, o Java, uma linguagem consolidada com forte presença em sistemas corporativos, é reconhecido pela sua robustez, segurança e estabilidade. O desenvolvimento de APIs RESTful em Java se beneficia de frameworks maduros e extensivamente testados, como Spring Boot, que simplificam a criação de serviços escaláveis e seguros, com recursos abrangentes de gestão de dependências, segurança e transações.

A seguir, veja os exemplos de comparação da API RESTful Node.js e Java.

Para o NodeJS usando Express:

```
// NodeJS com Express para criar uma API de gerenciamento de tarefas

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

```
const app = express();

app.use(express.json());

app.get('/tasks', (req, res) => {

    res.json([{ id: 1, title: 'NodeJS Task' }]);

});

app.post('/tasks', (req, res) => {

    // Adiciona uma tarefa à lista (simplificado)

    res.status(201).json(req.body);

});

app.listen(3000, () => console.log('API rodando na porta 3000'));
```

Para o Java usando Spring Boot:

```
// Java com Spring Boot para criar uma API de gerenciamento de tarefas

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.bind.annotation.GetMapping;
```

```
import
org.springframework.web.bind.annotation.PostMapping;

import
org.springframework.web.bind.annotation.RequestBody;

import
org.springframework.web.bind.annotation.RestController;

import java.util.ArrayList;

import java.util.List;

@SpringBootApplication

@RestController

public class TaskApplication {

    private List<Task> tasks = new ArrayList<>();

    public static void main(String[] args) {

                SpringApplication.run(TaskApplication.class,
args);

    }

    @GetMapping(“/tasks”)

    public List<Task> getTasks() {

        return tasks;

    }
```

```
    @PostMapping("/tasks")
    public Task addTask(@RequestBody Task task) {
        tasks.add(task);
        return task;
    }
}
class Task {
    public int id;
    public String title;
}
```

É importante salientar que a escolha entre Node.js e Java para o desenvolvimento de APIs RESTful não deve ser baseada apenas em preferências pessoais, mas sim em uma análise criteriosa das necessidades do projeto, considerando fatores como: a escala da aplicação, os requisitos de performance, a familiaridade da equipe com a linguagem e o ecossistema, e as particularidades do ambiente de execução.

Arquitetura de Microsserviços

Continuando, nesta segunda parte, aprofundaremos no tema "Arquitetura de Microsserviços", um elemento crucial ao integrar APIs RESTful em sistemas complexos. A arquitetura de microsserviços, como já vimos em aulas anteriores, é projetada para compreender uma coleção de serviços menores, cada um com responsabilidades específicas. Um exemplo prático seria um sistema de e-commerce onde cada serviço gerencia um aspecto

distinto, como pedidos, clientes e inventário, trabalhando de forma independente, mas integrada, para proporcionar uma experiência de usuário coesa e eficiente. Essa modularidade permite uma gestão eficiente e a evolução independente de cada serviço.

A razão pela qual estamos revisitando a arquitetura de microsserviços no contexto das APIs RESTful é sua afinidade com o modelo de requisição e resposta. Em uma arquitetura de microsserviços, é comum ter várias APIs que colaboram para construir um ambiente integrado para hospedar seu produto. Cada API, seja para recuperar ou enviar dados, opera de forma autônoma, assegurando alta disponibilidade e responsabilidade clara dos serviços.

No desenvolvimento de produtos, a arquitetura de microsserviços facilita a implementação de novas funcionalidades e a manutenção de serviços existentes, sem afetar o sistema na totalidade. Isso significa que se uma funcionalidade específica necessita de evolução, ela pode ser desenvolvida e integrada sem interromper ou comprometer o restante do sistema.

Explorando a dinâmica de requisição e resposta, destacamos o papel do API Gateway, uma espécie de intermediário que gerencia o acesso aos diversos microsserviços. Ele não apenas direciona as requisições aos serviços apropriados, mas também adiciona uma camada de segurança, evitando que os clientes acessem diretamente os serviços internos. Esse gateway também pode modificar dados para garantir que apenas informações seguras e autorizadas transitem entre o cliente e os serviços.

Avançando na arquitetura, cada microsserviço opera de forma independente, lidando com suas próprias responsabilidades, desde interações com o banco de dados até a execução de lógicas de negócio específicas. Essa independência evita que alterações em um serviço afetem outros, promovendo estabilidade e flexibilidade. Como no exemplo

de microsserviços em um sistema de e-commerce (conceitualmente, pois o código completo seria extenso):

- Serviço de Pedidos: Gerencia a criação, atualização e consulta de pedidos.
- Serviço de Clientes: Administra o cadastro e as informações dos clientes.
- Serviço de Inventário: Controla o estoque de produtos.

Por fim, abordamos a integração com o banco de dados, a qual é essencialmente gerida pelos microsserviços, sem interação direta com o API Gateway ou o cliente final. Isso enfatiza a importância da segmentação clara na arquitetura de microsserviços, onde cada camada tem seu papel específico, garantindo que a estrutura seja mantida de forma coesa e segura.

Em síntese, a arquitetura de microsserviços e APIs RESTful é fundamental para a construção de sistemas escaláveis e robustos, facilitando a gestão, evolução e manutenção dos serviços. Posteriormente, aprofundaremos em testes e qualidade de software, explorando como estes aspectos impactam no desenvolvimento de APIs eficazes.

Testes e qualidade de software

No contexto do desenvolvimento de APIs, os testes e qualidade de software representam um pilar fundamental. Surpreendentemente, muitas APIs são lançadas sem os devidos testes, resultando em problemas de funcionalidade e segurança.

Testar uma API envolve uma série de procedimentos para verificar a correta operação das funcionalidades implementadas. Um bom exemplo é o uso de testes unitários no endpoint de criação de usuários de uma API RESTful. Cada parte do processo, desde a validação de entrada até a persistência

dos dados, é examinada individualmente para garantir que tudo funcione conforme esperado, minimizando falhas e melhorando a qualidade do software.

Os testes são vitais para detectar bugs, relatar erros e contribuir para a solução de problemas. Eles enfatizam que nenhuma aplicação está livre de falhas, independentemente dos recursos investidos.

É importante entender que a presença de testes não elimina os erros, mas prepara a equipe para gerenciá-los eficientemente, reduzindo o impacto nos usuários finais. O desenvolvimento de software deve visar não apenas a prevenção de erros, mas também a implementação de estratégias adequadas para o seu gerenciamento.

Testes devem ser realizados em todas as fases do desenvolvimento para identificar e corrigir problemas precocemente, evitando grandes complicações na produção. Existem vários tipos de testes, incluindo unitários, funcionais, de segurança, de carga, de regressão e de integração, cada um com sua importância e papel no ciclo de desenvolvimento.

Os testes unitários testam a menor unidade de código, garantindo o funcionamento isolado de cada parte. Testes funcionais avaliam se a aplicação como um todo atende aos requisitos. Testes de segurança identificam vulnerabilidades, testes de carga examinam o comportamento sob alta demanda, testes de regressão verificam se novas funcionalidades não comprometem as existentes, e testes de integração avaliam a coesão entre diferentes partes do sistema.

A pirâmide de testes ilustra o equilíbrio entre o custo e a cobertura dos testes. Testes unitários são mais baratos e abrangentes em termos de possíveis falhas, enquanto testes end-to-end no topo da pirâmide são mais caros e complexos, mas simulam a experiência do usuário final.

Aqui está um exemplo de teste unitário para um endpoint de criação de usuários em uma API RESTful usando Node.js:

```
// Teste unitário em Node.js para um endpoint de criação de usuários

const request = require('supertest');

const app = require('../app'); // Suponha que app é a instância do Express

describe('POST /users', () => {

    it('should create a new user', async () => {

        const userData = { name: 'John Doe', email: 'john@example.com' };

        const response = await request(app).post('/users').send(userData);

        expect(response.statusCode).toBe(201);

        expect(response.body.name).toEqual(userData.name);

    });

});
```

Nesse exemplo, usamos `supertest` para simular uma requisição POST ao endpoint `/users`. O teste verifica se o status da resposta é 201 (criado) e se o nome do usuário na resposta corresponde ao enviado.

Concluindo, os testes são um investimento necessário na garantia da qualidade e confiabilidade de uma aplicação. A seguir, exploraremos como

os testes unitários são implementados no contexto do desenvolvimento com Node.js, destacando sua contribuição para a qualidade do software.

Utilização do Postman para testes de APIs

Na quarta e última parte de nossa aula, vamos nos aprofundar no uso do Postman para testar APIs. O Postman é uma ferramenta que facilita a execução e o teste de requisições HTTP, permitindo verificar se as respostas obtidas estão alinhadas com as necessidades e expectativas do desenvolvimento. Por exemplo, podemos usar o Postman para testar uma API de listagem de produtos, onde simulamos diferentes cenários, como respostas de sucesso e erro, verificando se a API está tratando corretamente os casos de uso, retornando os dados esperados ou mensagens de erro adequadas.

A utilização do Postman nos testes de API é fundamental para garantir que as funcionalidades implementadas estão corretas. Diferente dos testes unitários, que podem se basear em dados simulados, o Postman permite testar as requisições em um ambiente mais próximo ao real, validando não apenas se a API está funcionando, mas se os dados retornados são os esperados.

Neste contexto, o Postman valida a funcionalidade prática da API, testando as operações de CRUD (Create, Read, Update, Delete) através de requisições GET, POST, PUT, entre outras. Esse processo ajuda a verificar a integridade e a funcionalidade da API, além de permitir a automação desses testes.

Ao trabalhar com o Postman, é crucial entender a configuração do ambiente de testes, utilizando variáveis globais para simular diferentes cenários de uso. Essas variáveis podem incluir tokens de segurança, URLs de serviços e outros dados necessários para a execução dos testes, garantindo que o ambiente de teste esteja adequadamente preparado para cada caso.

É importante ter cautela ao configurar esses ambientes, especialmente para evitar o vazamento de informações sensíveis, como URLs de produção e dados de autenticação. O Postman oferece recursos como environments e workspaces para organizar e proteger as informações utilizadas nos testes.

Durante os testes, a documentação gerada pelo Postman se torna um recurso valioso, facilitando o entendimento das requisições e das respostas obtidas. A ferramenta também suporta a geração de snippets de código para diferentes linguagens e frameworks, auxiliando na integração e na replicação dos testes em outros ambientes de desenvolvimento.

A execução dos testes no Postman segue uma sequência definida, onde cada request é processada com base em scripts de pré-requisição e de teste, permitindo a validação dos resultados e a limpeza do ambiente após os testes. Esse procedimento assegura que os testes possam ser repetidos sem inconsistências ou erros de duplicidade nos dados.

Encerrando esta parte da aula, destacamos a importância de aplicar testes práticos no Postman, acompanhando o fluxo de execução e analisando os resultados obtidos.

Na próxima aula, exploraremos as boas práticas e os detalhes técnicos do teste unitário e de como integrá-lo ao nosso aprendizado sobre Postman, fornecendo um entendimento completo do ciclo de testes em desenvolvimento de software. Todo o material, incluindo coleções de testes e exemplos práticos, será disponibilizado no GitHub da disciplina para que vocês possam praticar e solidificar o conhecimento adquirido nesta série de aulas.

GitHub da Disciplina

Você pode acessar o repositório da disciplina no GitHub a partir do seguinte link: https://github.com/FaculdadeDescomplica/ProgramacaoII. Esse repositório tem como principal objetivo guardar os códigos das aulas práticas da disciplina para aprimorar suas habilidades em vários tópicos, incluindo a criação e consumo de APIs com controle de autenticação utilizando Node.js e utilizando boas práticas de programação e mercado.

Conteúdo Bônus

Para aprofundar o entendimento sobre APIs RESTful, sugiro o vídeo "O que é uma API RESTful na Prática? Maturidade de Richardson" apresentado por Michelli Brito em seu canal no YouTube. Esse conteúdo é uma excelente referência para entender os níveis de maturidade de Richardson, um modelo que ajuda a classificar as APIs RESTful de acordo com suas características e conformidade com os princípios REST.

Referências Bibliográficas

Bibliografia Básica:

ELMASRI, R.; NAVATHE, S. B. **Sistemas de banco de dados.** 7.ed. Pearson: 2018.

MEDEIROS, L. F. de. **Banco de dados:** princípios e prática. Intersaberes: 2013.

VICCI, C. (Org.). **Banco de dados.** Pearson: 2014.

Bibliografia Complementar:

CARDOSO, L. da C. **Design de aplicativos.** Intersaberes: 2022.

JOÃO, B. do N. **Usabilidade e interface homem-máquina.** Pearson: 2017.

LEAL, G. C. L. **Linguagem, programação e banco de dados:** guia prático de aprendizagem. Intersaberes: 2015.

PUGA, S.; FRANÇA, E.; GOYA, M. **Banco de dados:** implementação em SQL, PL/SQL e Oracle 11g. Pearson: 2013.

SETZER, V. W.; SILVA, F. S. C. **Bancos de dados.** Blucher: 2005.

Ir para exercício