

27/10/2025

**TP1**

**Microserviços e DevOps com Spring Boot e  
Spring Cloud**

Professor(a): Flávio da Silva Neves

## A1. DESCREVA AS PRINCIPAIS DIFERENÇAS ENTRE O SPRING BOOT E O SPRING CLOUD:

O Spring Boot tem como foco o desenvolvimento rápido e simplificado de aplicações Java independentes, ou seja, facilita a criação de um serviço de forma individual. O Spring Cloud por sua vez, tem como foco a construção de arquiteturas com vários microsserviços, onde múltiplos serviços precisam se comunicar e coordenar de maneira eficiente e resiliente

## A2. QUANDO CADA UM É UTILIZADO NA CONSTRUÇÃO DE UMA APLICAÇÃO USANDO MICROSERVIÇOS?

O Spring Boot para criar cada microsserviço individual, já o Spring Cloud devemos usar para integrar, coordenar e gerenciar os vários microsserviços Spring Boot.

### B1. Quais as diferentes arquiteturas de microsserviços?

Arquitetura	Comunicação	Descrição
<b>Tradicional REST</b>	HTTP síncrono	É o modelo mais comum e didático — cada serviço expõe APIs REST independentes, geralmente com Spring Boot
<b>Orientada a eventos</b>	Assíncrona (Kafka, RabbitMQ)	Em vez de os serviços se chamarem diretamente, eles se comunicam via eventos assíncronos — ideal para sistemas muito distribuídos.
<b>Com Gateway</b>	HTTP via API Gateway	Usa um ponto único de entrada para o sistema — o Gateway gerencia o acesso aos serviços internos
<b>Service Mesh</b>	Proxy distribuído	Uma evolução da arquitetura de microsserviços, usada em sistemas com Kubernetes e centenas de serviços
<b>Serverless</b>	Funções sob demanda	Usa funções isoladas em vez de serviços inteiros — ideal para cargas variáveis ou eventuais.
<b>Híbrida</b>	REST + Eventos	Combina APIs REST para consultas e eventos para processamento assíncrono.

### B2. Quando não devemos utilizar microsserviços?

1. Quando o sistema é pequeno ou simples;
2. Quando a equipe é pequena, 1 a 5 devs;
3. Quando não há maturidade em DevOps e infraestrutura, pois eles dependem de automação e observabilidade \*\*\*\*para funcionar bem;
4. Quando o domínio de negócio é simples e pouco volátil, ou seja, se as regras de negócio não mudam com frequência e os módulos são fortemente integrados
5. Quando há dependência forte entre módulos: Exemplo, se tudo o que o processo A faz, deve ser confirmado em B, o benefício da independência do microsserviços não existe mais;
6. Quando o custo operacional é uma preocupação, pois cada microsserviço precisa de seu ambiente, banco de dados e monitoração.

### B3. Quais os princípios de aplicações nativas da nuvem?

1. Arquitetura baseada em microsserviços
2. Containers e orquestração
3. Automação e DevOps
4. Imutabilidade e Infraestrutura como Código (IaC)
5. Escalabilidade e elasticidade

6. Resiliência e tolerância a falhas
7. Configuração e descoberta de serviços dinâmicas
8. Observabilidade e monitoramento
9. Segurança integrada (Security by Design)
10. Cultura de DevSecOps e entrega contínua

#### B4. Liste as práticas da Twelve-Factor App.

Nº	Fator	Descrição resumida	Objetivo principal
1	<b>Codebase (Base de código)</b>	Uma única base de código versionada (ex: Git) por aplicação, mas podendo ter múltiplos deploys.	Garantir rastreabilidade e consistência.
2	<b>Dependencies (Dependências)</b>	Declare todas as dependências explicitamente (ex: via Maven, npm, pip).	Evitar dependências implícitas do ambiente.
3	<b>Config (Configuração)</b>	Armazene configurações (senhas, URLs, chaves) em variáveis de ambiente, nunca no código.	Separar código de ambiente e facilitar deploys.
4	<b>Backing Services (Serviços de apoio)</b>	Trate serviços externos (banco, cache, fila) como recursos anexáveis, acessados por URL.	Facilitar substituição e escalabilidade.
5	<b>Build, Release, Run</b>	Separe claramente as etapas de build (compilar), release (configurar) e run (executar).	Garantir reprodutibilidade e estabilidade nos deploys.
6	<b>Processes (Processos)</b>	Execute a aplicação como um ou mais processos <i>stateless</i> (sem depender de sessão local).	Facilitar escalabilidade e tolerância a falhas.
7	<b>Port Binding (Vinculação de porta)</b>	A aplicação deve expor serviços via porta (HTTP, gRPC etc.), sem depender de servidor externo.	Tornar o serviço autônomo e independente.
8	<b>Concurrency (Concorrência)</b>	Escale a aplicação clonando processos, em vez de usar threads internas.	Escalabilidade horizontal mais simples.
9	<b>Disposability (Descartabilidade)</b>	Os processos devem iniciar e encerrar rapidamente, permitindo deploy e recuperação ágeis.	Maior resiliência e tempo de resposta menor.
10	<b>Dev/Prod Parity (Paridade entre Dev e Produção)</b>	Mantenha os ambientes de desenvolvimento, teste e produção o mais similares possível.	Reduzir erros e diferenças entre ambientes.
11	<b>Logs</b>	Trate logs como fluxo de eventos, enviados para sistemas externos de agregação (ex: ELK, CloudWatch).	Facilitar análise e monitoramento centralizado.
12	<b>Admin Processes (Processos administrativos)</b>	Execute tarefas administrativas (migrar DB, scripts) como processos pontuais, separados da aplicação.	Evitar interferência com a execução normal.

## B5. Quais as principais diferenças entre o estilo arquitetura Monolítica e o estilo arquitetura de Microsserviços?

O Monolito é construído em um único bloco, mais fácil para manutenção, deploy, atualizações e é mais fácil de desenvolver inicialmente, porém é mais difícil de manter à medida que cresce, ideal para aplicações simples como uma loja de produtos locais, sistema interno de uma empresa.

O Microsserviço por sua vez é mais complexo para criar, porém à medida que cresce é mais fácil de manter, pois cada parte é independente. Imagine uma empresa que presta serviços de atendimento médico, nela temos um serviço independente para cadastro de usuários, agendamento de consultas, exames, deslocamentos externo de pacientes, etc. Dentro deste cenário, cada parte funciona de forma independente, ou seja, se o sistema de exames cair, ainda poderemos agendar consultas e fazer as demais operações pois elas são independentes umas das outras.

## C1. DESENVOLVIMENTO DE MICROSERVIÇOS:

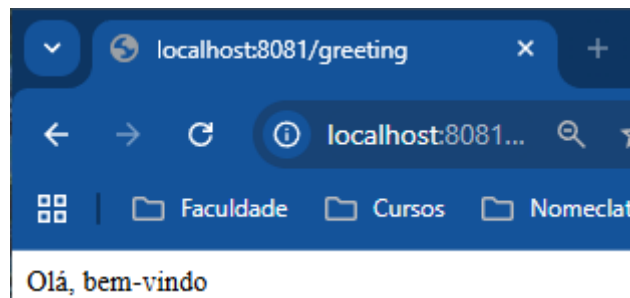
- No exemplo abaixo foram criados dois microsserviços, o service-a na porta 8081 e service-b na porta 8082.
- O service-a exporta uma mensagem padrão e o service-b obtém a mensagem padrão e acrescenta o nome do usuário na mensagem

### Como testar:

1. Inicie o service-a e acesse no navegador:

<http://localhost:8081/greeting>

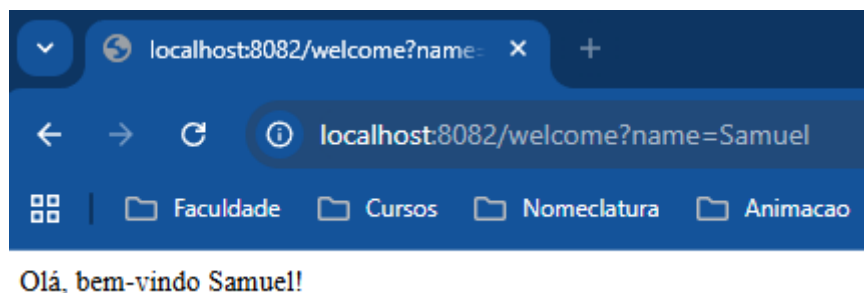
- **Resultado:**



2. Inicie o service-b e acesse no navegador:

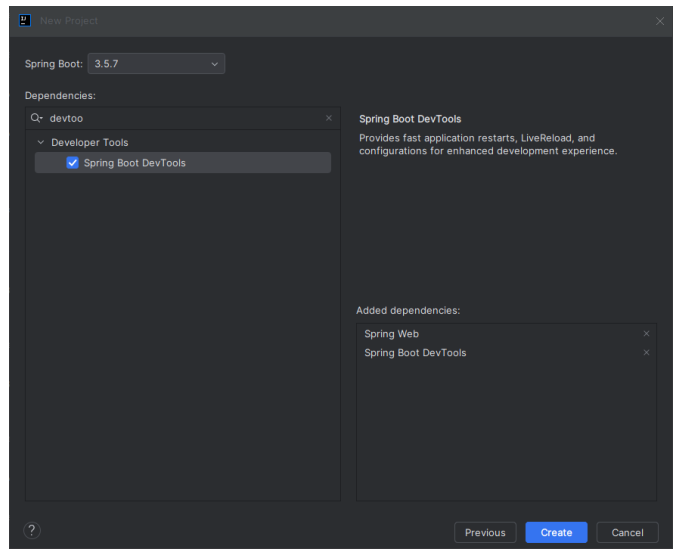
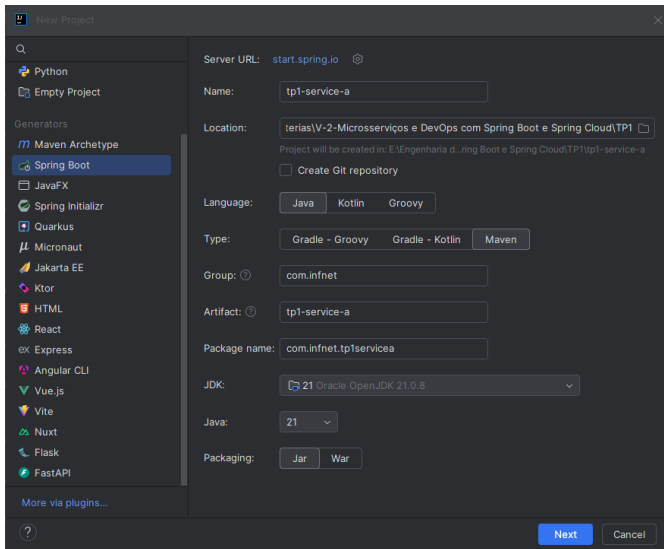
<http://localhost:8082/welcome?name=Samuel>

### Resultado:

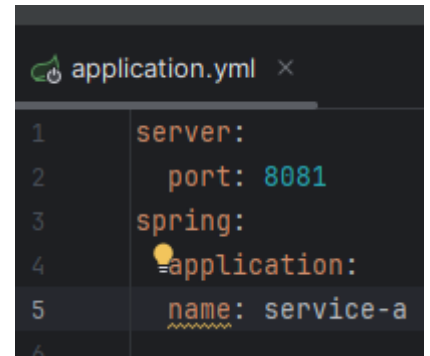
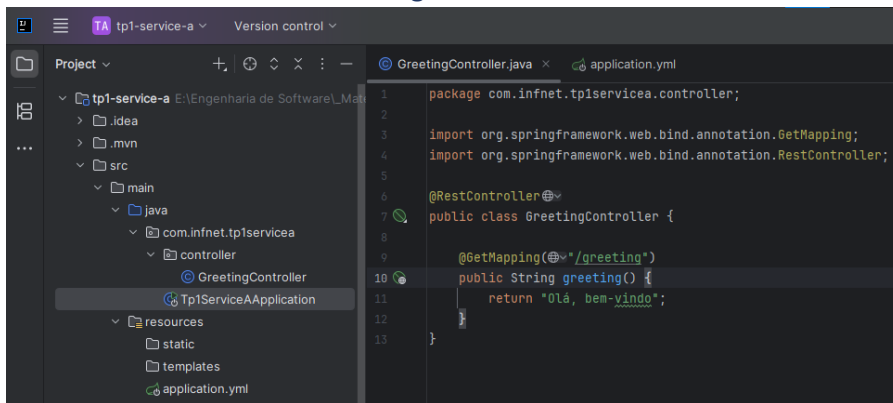


# Imagens da Aplicação

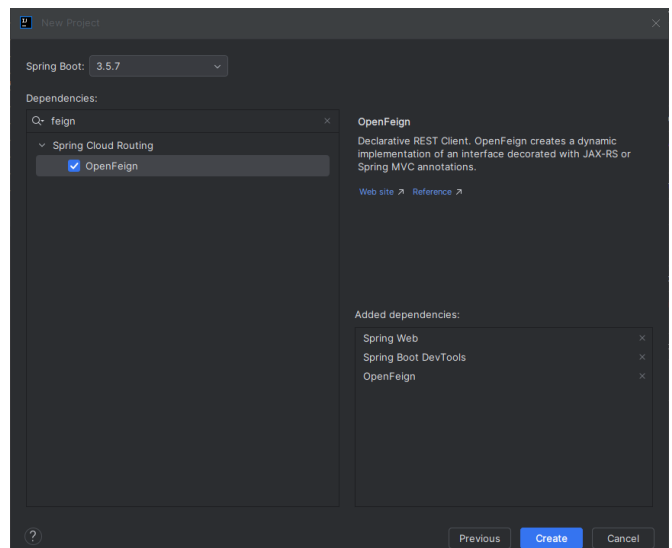
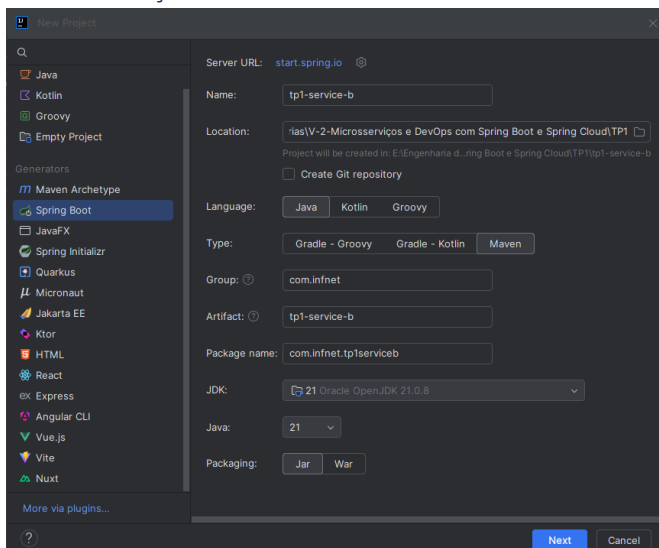
## 1. Criação Service a



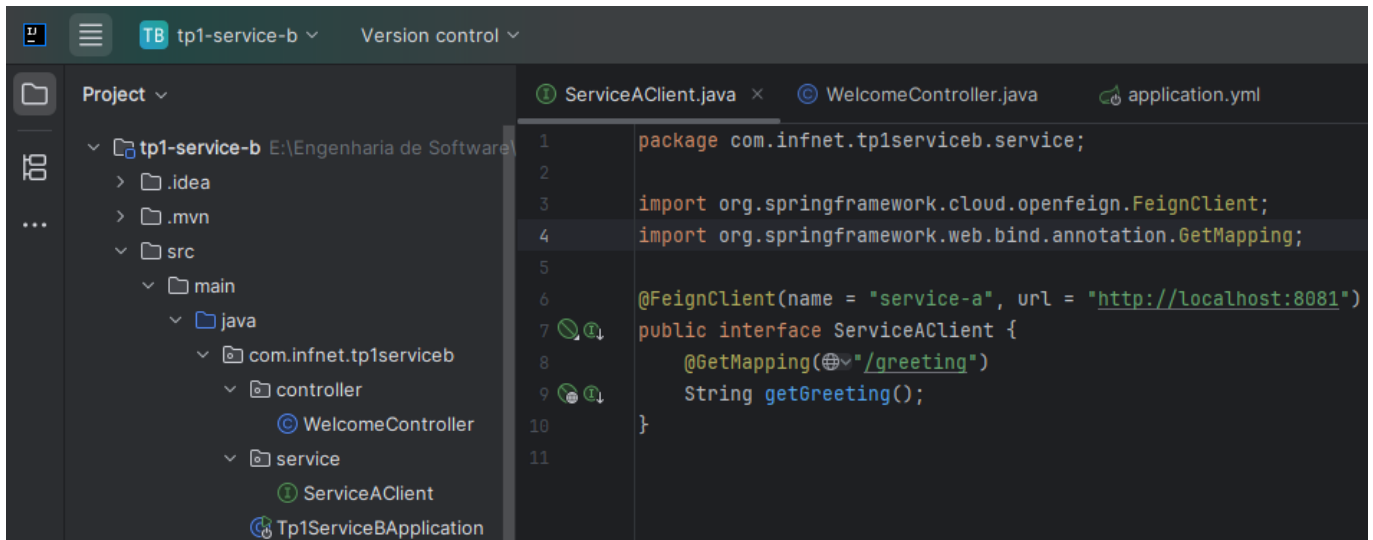
## 2. Controller com a mensagem default



## 3. Criação Service b



#### 4. Service-a Acesso



The screenshot shows the IntelliJ IDEA interface. On the left, the 'Project' view displays the directory structure of 'tp1-service-b', including 'src/main/java/com/infnet/tp1serviceb/controller' and 'service'. The main editor shows 'ServiceAClient.java' with the following code:

```
1 package com.infnet.tp1serviceb.service;
2
3 import org.springframework.cloud.openfeign.FeignClient;
4 import org.springframework.web.bind.annotation.GetMapping;
5
6 @FeignClient(name = "service-a", url = "http://localhost:8081")
7 public interface ServiceAClient {
8     @GetMapping("/greeting")
9     String getGreeting();
10 }
11
```

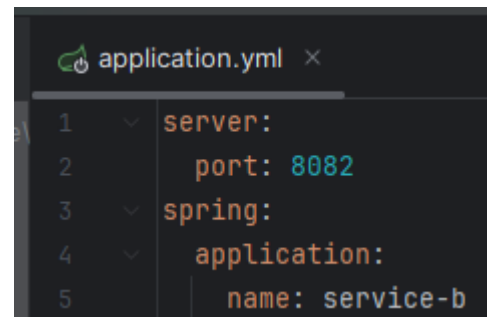
#### 5. Welcome Controller

Faz a união da mensagem recebida do service-a e cria o end point final a ser exibido utilizando nome informado na url via RequestParam.



The screenshot shows the 'WelcomeController.java' file in the IntelliJ IDEA editor. The code is as follows:

```
1 package com.infnet.tp1serviceb.controller;
2
3 import ...
4
5 @RestController
6 public class WelcomeController {
7     private final ServiceAClient serviceAClient;
8
9     public WelcomeController(ServiceAClient serviceAClient) {
10         this.serviceAClient = serviceAClient;
11     }
12
13     @GetMapping("/welcome")
14     public String welcome(@RequestParam String name) {
15         String baseMessage = serviceAClient.getGreeting();
16         return baseMessage + " " + name + "!";
17     }
18 }
19
```



The screenshot shows the 'application.yml' file in the IntelliJ IDEA editor. The configuration is as follows:

```
1 server:
2     port: 8082
3 spring:
4     application:
5         name: service-b

```