Desenvolvimento Web II

Introdução à Web Services REST com Java e Spring Boot

Prof. Dr. Daniel A. Furtado - FACOM/UFU

Conteúdo

- Conceito de *web service*
- Tipos de *web services*: SOAP x REST
- Métodos HTTP e conceito de Idempotência
- Introdução ao framework Spring e tecnologias associadas
- Inversão de Controle e Injeção de Dependência
- Arquitetura MVC e arquitetura em camadas em API Rest com Spring Boot

Web Services e APIs

- Definição formal: *web service* é uma entidade de software independente de linguagem, baseada em padrões, que aceita requisições HTTP especialmente formatadas de outras entidades de software em máquinas remotas, e produz respostas específicas da aplicação;
- Em outras palavras, um web service é basicamente uma funcionalidade de software criada para ser acessada por outras aplicações por meio da web;
- Para prover interoperabilidade, os *web services* utilizam padrões de comunicação universais, como o protocolo HTTP, JSON, XML etc.;
- O ViaCep (viacep.com.br), para busca de endereços a partir de CEPs, é um exemplo de web service, assim como o Alpha Vantage, utilizado para buscar cotações de ações;
- Em geral, os web services constituem APIs (*Application Programming Interface* - Interface de Programação de Aplicação), porém nem toda API é um web service (ex. API's do Windows em aplicações locais).

Tipos de Web Services

- Os web services são comumente categorizados de acordo com a tecnologia em que se baseiam, que são:
  - SOAP
  - REST

Tipos de Web Services

SOAP

- Acrônimo para **S**imple **O**bject **A**ccess **P**rotocol;
- Utiliza mensagens em XML, com sintaxe específica da aplicação, que são transmitidas no corpo das requisições e respostas HTTP;
- O formato dessas mensagens, assim como a forma de acesso ao web service, é definido pela **W**eb **S**ervices **D**escription **L**anguage (WSDL);
- Como é baseado na XML, é independente de plataforma ou linguagem;
- Mais comumente utilizado em APIs privadas, com medidas de segurança mais rígidas;
- O servidor pode manter o estado da aplicação armazenando as mensagens anteriores trocadas com um cliente.

SOAP - Exemplo

1) Exemplo de mensagem de requisição SOAP para buscar no servidor a cotação de uma ação na bolsa

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

Uma desvantagem clara do SOAP é o overhead com *metadados*, o que demanda uma maior largura de banda

2) Exemplo de uma mensagem de resposta SOAP contendo o preço da ação

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

Adaptado de *w3schools.com*

Tipos de Web Services

REST

- **Re**presentational **S**tate **T**ransfer;
- Estilo arquitetural para comunicação entre aplicações web;
- Se baseia nos métodos HTTP (GET, POST, PUT, DELETE), nos códigos de status e na identificação dos recursos pela própria URL (endpoints);
- Diferente do SOAP, não impõe restrições ao formato da mensagem, mas apenas no comportamento das entidades envolvidas:
  - Mais flexível: o desenvolvedor pode utilizar o formato que for mais apropriado, como XML, JSON, texto etc.
- **Stateless**: toda requisição deve conter todas as informações necessárias para que o servidor possa processá-la;
- RESTful é o termo comumente utilizado para designar *web services* / APIs baseados no estilo REST.

Web Services e Idempotência

- Como os web services REST se baseiam nos métodos HTTP, surge a necessidade de entender o conceito de idempotência nesse contexto;
- Um requisição HTTP é dita ser idempotente quando mantem a seguinte propriedade:
  - Executar a requisição múltiplas vezes tem o mesmo efeito no estado da aplicação no servidor que executá-la uma única vez;
  - Em outras palavras, isso significa que executar a mesma requisição pela segunda ou terceira vez não irá alterar o estado da aplicação no servidor.

Principais Métodos HTTP para serviços RESTful

GET

- De acordo com a especificação, GET deve ser utilizado para acesso (leitura) de recursos no servidor, mas não para alteração;
- Em caso de sucesso, retorna-se uma representação do recurso em formatos como JSON, XML, texto etc. e o código de status 200 - Ok;
- Por definição, é idempotente. Isto significa que um usuário deve poder executar a requisição múltiplas vezes sem se preocupar em produzir efeitos adversos ou alterações de dados no servidor.

POST

- Utilizado para criar novos recursos;
- Em geral, altera o estado da aplicação no servidor;
- Em caso de sucesso, deve-se retornar o código de status 201 – Created;
- Não é idempotente. Isto significa que sucessivas requisições idênticas utilizando o POST podem levar a alterações de dados no servidor (por exemplo, podem levar à criação de dois ou mais recursos contendo a mesma informação).

Principais Métodos HTTP para serviços RESTful

PUT

- Para operações de atualização/substituição por inteiro de um recurso no servidor;
- É idempotente porque execuções sucessivas de uma mesma requisição resultam no mesmo estado final do recurso. Diferente do POST, que pode gerar efeitos colaterais como duplicidade de registros, o PUT deve garantir que, após a atualização inicial, o estado da aplicação permaneça inalterado em requisições subsequentes.

DELETE

- Utilizado para excluir um recurso no servidor;
- Em caso de sucesso, deve-se retornar o código de status 200 - OK;
- É idempotente. Repetidas requisições para remoção do mesmo recurso devem ter o mesmo resultado (200 – OK), ou seja, o recurso foi apagado e continua apagado.

Principais Métodos HTTP para serviços RESTful

PATCH

- Comumente utilizado para operações de atualização parcial de um recurso (por exemplo, atualização de um dado em particular de um cliente, como número de telefone ou estado civil);
- Não é idempotente, o que significa que requisições sucessivas utilizando o PATCH podem ter efeitos diferentes (por ex., uma atualização do salário de um funcionário em 10%).

Endpoints e Rotas

- No contexto de *web services*, **endpoints** representam URLs específicas que os desenvolvedores devem utilizar para enviar solicitações e receber respostas. Exemplo:
  - `viacep.com.br/ws/38400-100/json`
- Uma **rota** normalmente inclui um método HTTP e pode conter nomes de variáveis. Rotas definem como a aplicação trata requisições para vários endpoints. Exemplos:
  - `GET minha.api.com/v1/book/{id}`
  - `DELETE minha.api.com/v1/book/{id}`

Métodos Suportados em Formulários HTML

- Vale destacar que, para submissão de formulários HTML utilizando a tag <form>, os únicos métodos suportados são GET e POST (não se deve utilizar, por exemplo, `<form method='put'>`);
- Entretanto, os demais métodos podem ser utilizados por meio de requisições HTTP utilizando o **XMLHttpRequest** ou a **API Fetch**.

Introdução ao Framework Spring

- **Spring** é um framework de código aberto amplamente utilizado no desenvolvimento de aplicações Java;
- É organizado em vários módulos e containers que permitem o desenvolvimento de vários tipos de aplicações, incluindo aplicações web;
- Por exemplo, há módulos básicos (**Core**) que implementam recursos fundamentais do framework, como suporte a **Inversão de Controle** (IoC);
- Por outro lado, o módulo **Spring Web Servlet** dá suporte ao tratamento e mapeamento de requisições HTTP em aplicações web utilizando o padrão de arquitetura MVC (model-view-controller).

Spring Boot

- Lançada em 2014, a ferramenta Spring Boot permite criar rapidamente um novo projeto utilizando o Spring
  - Vários recursos já estão pré-configurados, evitando a configuração detalhada de arquivos XML (o que é necessário quando se utiliza o Spring sem o Spring Boot);
  - Simplifica a inclusão de dependências do projeto.

Ambiente de Desenvolvimento

IntelliJ IDEA

- Ambiente de desenvolvimento integrado (IDE) multiplataforma para linguagens baseadas na Java Virtual Machine (JVM) como Java e Kotlin;
- Também suporta outras linguagens por meio de plugins;
- Pode ser instalado gratuitamente. Recursos fundamentais são gratuitos, porém funcionalidades avançadas podem exigir a licença paga Ultimate (disponibilizada gratuitamente para docentes e alunos da UFU).
- https://www.jetbrains.com/idea/download

Outras Tecnologias Envolvidas

Apache Tomcat

- Servidor Web de código aberto desenvolvido pela Apache Software Foundation que implementa uma série de especificações para desenvolvimento de aplicações web com Java.

Ferramenta de Automação de Compilação (build automation)

- Ferramenta para automatizar o processo de compilação (build) de software, gerenciando as dependências do projeto tais como bibliotecas e frameworks utilizados;
- Geralmente fazem o download automático das dependência do projeto e as mantêm em cache localmente;
- Alguns exemplos são o Gradle e o Maven.

Algumas Tecnologias Envolvidas

Gradle

- Ferramenta de automação de compilação de código aberto que utiliza scripts declarativos em Groovy ou Kotlin;
- Pode ser utilizada no desenvolvimento com Java e outras linguagens, mas sua popularidade é maior no desenvolvimento para Android;
- Suas vantagens incluem flexibilidade e desempenho.

Maven

- Ferramenta de automação de compilação desenvolvida pela Apache;
- Bastante utilizada no desenvolvimento com Java;
- Suas vantagens incluem maturidade, ampla documentação e ecossistema de plugins;
- Utiliza um arquivo de configuração em XML (pom.xml).

Exemplo de arquivo pom.xml utilizado no Maven

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https:/
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.1.0</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>demo</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>17</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

Tutorial Introdutório – Java com Spring Boot e Maven

https://furtado.prof.ufu.br/site/teaching/PPI/Trabalho14-Intro-Spring.pdf

Conceito de Acoplamento e SRP

- Acoplamento é o grau de interdependência entre os módulos ou classes de um sistema.
- Um dos objetivos da arquitetura de software é alcançar um baixo acoplamento para que eventuais alterações em um módulo não desencadeie alterações nos demais.
- Sistemas com alto acoplamento tendem a ser mais difíceis de testar e manter.
- Um baixo acoplamento normalmente é buscado em conjunto com o Princípio da Responsabilidade Única (SRP), que desestimula a criação de classes com múltiplos propósitos (que "fazem de tudo").
- A instanciação manual de dependências dentro das classes (new Classe) tende a gerar maior acoplamento e ser menos eficiente. O princípio de Inversão de Controle (IoC) e a Injeção de Dependência focam em contornar esse problema.

Inversão de Controle (IoC – Inversion of Control)

- A Inversão de Controle (IoC) é um princípio de design de software que visa transferir o controle do fluxo de execução e criação de objetos do código do desenvolvedor para um framework ou container externo.
- Diferente da programação tradicional, onde o código do programador chama o código de bibliotecas, com a IoC é o framework quem chama o código do programador (Princípio de Hollywood: não nos chame, nós chamaremos você).
- No contexto do Spring, o framework assume a responsabilidade de gerenciar o ciclo de vida completo dos objetos da aplicação.

O Container Spring de Inversão de Controle

- No contexto da aplicação Spring, a Inversão de Controle é viabilizada por meio da interface ApplicationContext, que tem papel fundamental no framework.
- Uma classe do framework que implementa essa interface é instanciada assim que a aplicação é iniciada (o tipo específico varia conforme a aplicação).
- Esse objeto faz o papel de container da aplicação e tem atribuições importantes como carregar as configurações, identificar os componentes (por meio de anotações como @RestController, @Repository, @Service) e instanciar objetos.
- Os objetos gerenciados automaticamente pelo container são denominados de Beans.

Injeção de Dependência (DI – Dependency Injection)

- A Injeção de Dependência (DI) é um padrão de projeto utilizado para implementar a Inversão de Controle na prática.
- Em vez de um objeto buscar ou criar suas próprias dependências, ele apenas declara o que precisa e o container fornece as instâncias prontas (por exemplo, via construtor).
- Essa técnica permite, por exemplo, que uma implementação real possa ser trocada mais facilmente por uma simulada, para fins de testes.

Estratégias de Injeção de Dependência

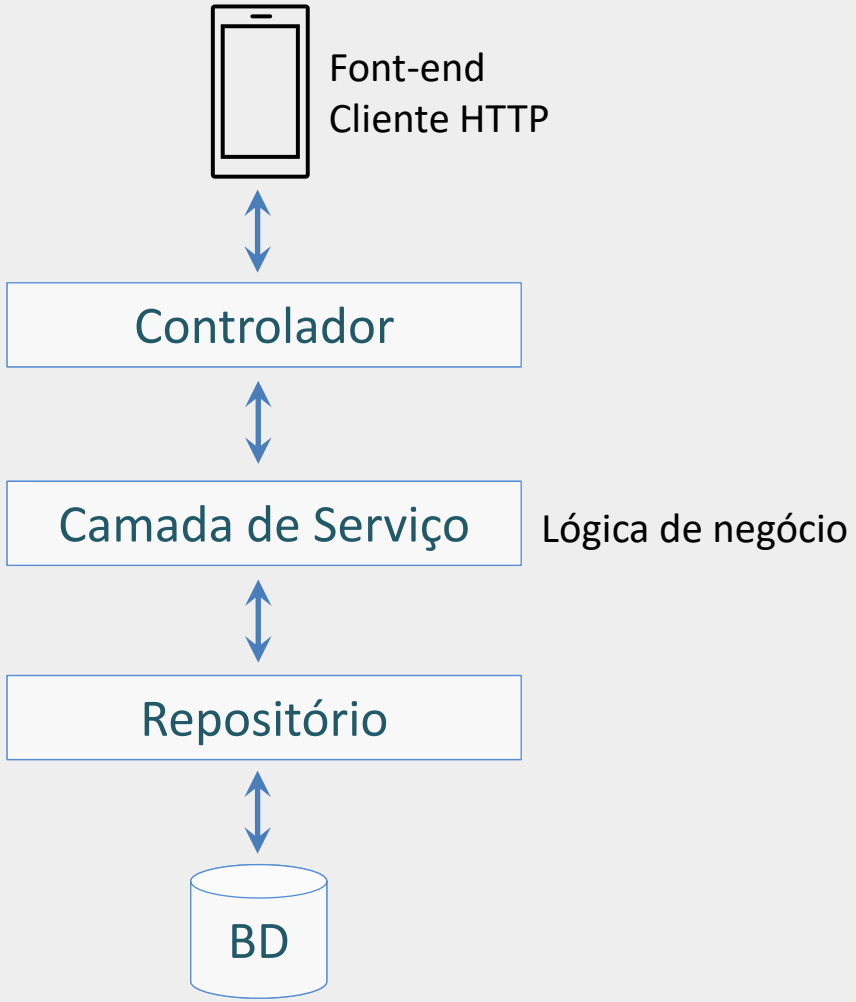
- Pelo Construtor: É a estratégia recomendada, pois garante que o objeto seja instanciado em um estado válido, com todas as dependências obrigatórias preenchidas e imutáveis.
- Por método Setter: Permite a injeção de dependências opcionais ou que podem ser alteradas após a criação do objeto. É menos comum.
- Por Campo (@Autowired): É desencorajada atualmente, pois viola o encapsulamento e dificulta a realização de testes unitários fora do container Spring.

O Padrão de Arquitetura MVC (Model-View-Controller)

- Originalmente concebido na década de 70, o MVC organiza a aplicação em três componentes lógicos: Model (modelo/dados), View (visão/apresentação) e Controller (controlador/interação).
- Em APIs REST modernas, a View tradicional (HTML) é frequentemente substituída pela representação dos dados em JSON (ou XML), para consumo por clientes diversos (web, mobile, etc.)
- O Controller atua como um intermediário que recebe as requisições e delega o processamento para o Model.

Arquitetura em Camadas

- O modelo utilizado pelo Spring segue o padrão de arquitetura em camadas;
- No lugar do componente Model do padrão MVC, temos a camada de serviço (service), onde as regras de negócio são implementadas, e a camada de repositório (repository), responsável pela comunicação com o BD, abstraindo e encapsulando o acesso aos dados.
- Essa separação lógica (Controller ➔ Service ➔ Repository ➔ Database) protege as regras de negócio de eventuais mudanças na interface de acesso da API ou de alterações no SGBD.



Camada do Controlador

- A anotação @RestController permite indicar que a classe fará o papel de controlador na API Rest. Ela será especializada em lidar com as requisições HTTP e retornará os dados diretamente no corpo da resposta HTTP.
- A classe deve se limitar à interpretação dos métodos HTTP utilizados nas requisições, e mapeá-los em chamadas de serviço adequadas.
- É responsabilidade do controlador retornar o código de status adequado como 201 Created ou 404 Not Found.
- É possível ter várias classes marcadas com @RestController atuando como controlador.

Camada de Serviço

- Todo código relacionado à lógica da API deve ser incorporado na camada de serviço, deixando o controlador estritamente com seu papel de tratar requisições HTTP (thin controller).
- A camada de serviço, portanto, representa o núcleo da aplicação, sendo responsável pelas regras de negócio, validações lógicas e cálculos em geral.
- A camada de serviço pode se comunicar com múltiplos repositórios de dados sem ter que se preocupar com a parte técnica da persistência, como a escrita de código SQL e o gerenciamento de conexões com o banco de dados.
- A anotação @Service deve ser inserida na classe para que o framework a reconheça como um componente de serviço. É possível ter múltiplas classes exercendo esse papel.

Camada de Repositório

- A camada de repositório atua como uma interface entre a lógica de negócio (camada de serviço) e o mecanismo de persistência de dados, escondendo a complexidade de acesso aos dados.
- Na camada de repositório os dados são tratados como uma coleção de objetos em memória (criados sob demanda), e não como tabelas e tuplas.
- Nesse contexto surgem as ferramentas de Mapeamento Objeto-Relacional (ORM), permitindo que os dados do modelo relacional sejam representados de maneira automatizada como objetos da linguagem.
- O Spring Boot utiliza o Spring Data JPA com o Hibernate como implementação de ORM padrão.

Classes de Repositório com o Spring Data JPA

- Para criar um componente de repositório, basta definir uma interface que deriva da interface JpaRepository do Spring e marca-la opcionalmente com a anotação @Repository.
  - Ex: public interface ProdutoRepository extends JpaRepository<Produto, Long> ...
- Dessa forma, o Spring criará dinamicamente uma classe de repositório implementando as definições da interface. Vários métodos comuns de acesso a dados estarão disponíveis: findAll, findById, save, delete, etc.
- Se necessário, assinaturas de métodos especializados podem ser acrescentados na interface. Há duas formas principais:
  - Utilizando query methods (ex. findByCpf)
  - Utilizando a anotação @Query

Query Methods

- O Spring possui um mecanismo que interpreta o nome do método na interface e gera a consulta SQL automaticamente.
- É adequado para realizar consultas simples, sem a necessidade de escrever o código SQL diretamente.
- Por exemplo, um método com o nome findByMarcaAndQdeEstoque(String marca, Integer qdeEstoque) será traduzido para um SELECT ... WHERE marca = ? AND qdeEstoque = ?

Anotação @Query com SQL Nativo

- Para situações onde os métodos por nome não são suficientes, pode-se utilizar a linguagem de consulta JPQL ou até mesmo o SQL nativo, como no exemplo abaixo.

```
@Query(value = """
   SELECT * FROM produto
   WHERE preco <= :precoLimite
   ORDER BY preco ASC
""", nativeQuery=true)
List<Produto> buscarPorPrecoLimiteComSQL(@Param("precoLimite") Double precoLimite);
```

Entidades - Anotação @Entity

- No contexto da aplicação Spring, uma Entidade é uma classe Java que possui correspondência direta a uma tabela do banco de dados.
- Ao utilizar a anotação @Entity em uma classe, estamos instruindo o framework a trata-la como uma unidade de persistência gerenciada.
- A classe marcada com @Entity deve possuir um construtor padrão (sem argumentos) e não pode ser declarada como final.

Gerenciamento de Transações

- Para executar uma sequência de operações que caracterizam uma transação de banco de dados, basta marcar o respectivo método com a anotação @Transactional.
- Dessa forma, as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade) serão garantidas automaticamente pelo framework.
- Se uma exceção for lançada durante a execução desse método, o framework realizará o rollback automaticamente, revertendo as alterações parciais.

Gerenciamento de Transações

```java
@Transactional  1 usage
public Cliente registrarClienteEEndereco(ClienteEnderecoDTO dto) {
    // Instanciação da entidade Cliente (principal)
    Cliente cliente = new Cliente();
    cliente.setNome(dto.nome());
    cliente.setCpf(dto.cpf());

    // Instanciação da entidade EnderecoCliente (dependente)
    EnderecoCliente endereco = new EnderecoCliente();
    endereco.setEndereco(dto.endereco());

    // Criação do vínculo
    cliente.adicionarEndereco(endereco);

    // Gravação no banco
    // O Hibernate identificará que o cliente possui um endereço e fará o INSERT
    // na tabela 'cliente' seguido pelo INSERT na tabela 'endereco_cliente'.
    return repository.save(cliente);
}
```

Programação Orientada a Aspectos (AOP)

- Ao utilizar a notação @Transactional no exemplo anterior, estamos utilizando, na prática, o conceito de Programação Orientada a Aspectos.
- Não foi necessário escrever explicitamente os comandos para iniciar a transação (begin transaction), aplicar o commit no final das operações ou executar um rollback para desfazer em caso de falha.
- Ao utilizar @Transaction no método RegistraClienteEEndereco, o módulo Spring AOP do framework intercepta a chamada desse método, envolve a execução com a lógica de gerenciamento de transação e então retoma o fluxo normal de execução.

Programação Orientada a Aspectos (AOP)

- Suponha que seja necessário implementar, na API de exemplo do trabalho anterior, um mecanismo de registro de logs em todos os métodos das classes de serviço (ClienteService e ProdutoService).
- O objetivo é registrar, para cada método chamado, os valores dos parâmetros, o nome do método e seu tempo de execução.
- Em uma abordagem tradicional precisamos adicionar várias linhas de código em cada método de cada classe de serviço para registrar os valores dos parâmetros e o tempo de execução.
- Esse tipo de solução gera dois problemas principais:
  - Espalhamento de código: mesma lógica repetida em vários lugares
  - Emaranhamento de código: regras de negócio misturadas com lógica de infraestrutura.

Programação Orientada a Aspectos (AOP)

- Na arquitetura de sistemas, requisitos como auditoria, segurança e controle de transações são chamados de interesses transversais (*crosscutting concerns*).
- Podem estar presentes nas várias camadas do sistema, mas não fazem parte das regras de negócio propriamente ditas.
- A Programação Orientada a Aspectos propõe que todo o código relacionado a esses interesses transversais sejam movidos para módulos isolados denominados Aspectos.
- Em vez de modificar o código original, cria-se regras para ditar quando e onde esse comportamento adicional deve ser injetado.

Programação Orientada a Aspectos (AOP)

- O Spring implementa a AOP por meio do padrão de projeto Proxy.
- Quando o ProdutoService é injetado no ProdutoController, o framework não entrega uma instância direta da classe ProdutoService.

```
public class ProdutoController {
    private final ProdutoService service;  8 usages

    public ProdutoController(ProdutoService service) { this.service = service; }
```

- Ao invés, ele entrega uma instância de uma classe derivada de ProdutoService, gerada automaticamente pelo framework, que faz o papel de proxy.
- Essa classe derivada contem definições próprias (override) dos métodos de ProdutoService. Portanto, o framework tem uma forma de interceptar as chamadas dos métodos reais, acrescentar o código de um aspecto e, apenas depois, permitir que o método real seja executado.
- Dessa forma, o método original pode permanecer livre do código transversal, focando apenas em sua funcionalidade principal.

Programação Orientada a Aspectos (AOP) - Conceitos

Aspect

- Representa a modularização de um interesse transversal como controle de transação ou registro de logs.
- No Spring AOP, um aspecto pode ser implementado utilizando classes convencionais em conjunto com notações em XML ou por meio da anotação @Aspect

Join Point

- Representa um ponto de interceptação no código, como um método em particular ou um atributo, onde o código do aspecto será injetado para permitir a execução do interesse transversal. No caso do Spring AOP, o join point se limita a métodos.

Pointcut

- Expressão utilizada para selecionar um conjunto de join points.

Advice

- É o bloco de código do aspecto que será injetado e executado no join point. Ao definir um advice, define-se também o momento exato em que esse código será executado: antes, depois ou 'ao redor' do método original a ser interceptado.

```java
@Aspect // Marca a classe como um 'aspecto' no contexto da AOP para que a execução possa ser feita de maneira transversal.
@Component // Indica que a classe deve ser tratada como um componente gerenciado pelo container de Inversão de Controle (IoC) do Spring.
public class AuditoriaAspect {
    private static final Logger logger = LoggerFactory.getLogger(AuditoriaAspect.class);  3 usages

    // A anotação @Around define este metodo como um 'advice', tornando possível monitorar por completo
    // a execução do metodo original (ex. service.save())
    // A expressão entre aspas define o 'Pointcut', que seleciona todos os locais (join points) onde o 'advice' será executado.
    // O parametro jointPoint é um objeto que dá acesso ao contexto do metodo sendo monitorado.
    @Around("execution(* com.example.demo.service.*.*(..))")  no usages
    public Object auditarExecucao(ProceedingJoinPoint joinPoint) throws Throwable {
        long tempoInicio = System.currentTimeMillis();
        String nomeMetodo = joinPoint.getSignature().toShortString();
        Object[] argumentos = joinPoint.getArgs();

        logger.info("--- LOG automático com AOP: Iniciando a execução de: {} com os argumentos: {}", nomeMetodo, Arrays.toString(argumentos));
        Object resultado;
        try {
            resultado = joinPoint.proceed(); // Inicia a execucao do metodo monitorado
        } catch (Throwable e) {
            logger.error("--- LOG automático com AOP: Falha na execução de: {}. Erro: {}", nomeMetodo, e.getMessage());
            throw e; // Repassa a exceção para não quebrar o fluxo original da aplicação
        }
        long tempoTotal = System.currentTimeMillis() - tempoInicio;
        logger.info("--- LOG automático com AOP: Finalizando a execução de: {}. Tempo de processamento: {} ms", nomeMetodo, tempoTotal);
        return resultado;
    }
}
```

Referências

- https://developer.mozilla.org/en-US/docs/Web/HTTP
- https://docs.spring.io/spring-boot/documentation.html
- https://docs.spring.io/spring-framework/reference/index.html
- https://docs.spring.io/spring-framework/reference/web/webmvc.html
- https://docs.spring.io/spring-framework/reference/core/aop.html
- https://spring.io/guides/tutorials/rest