



Desenvolvimento de Serviços com Spring Boot

Etapa/Aula 04.1

Professor(a): Flávio Neves

E-mail: flavio.neves@prof.infnet.edu.br

Roteiro da Aula

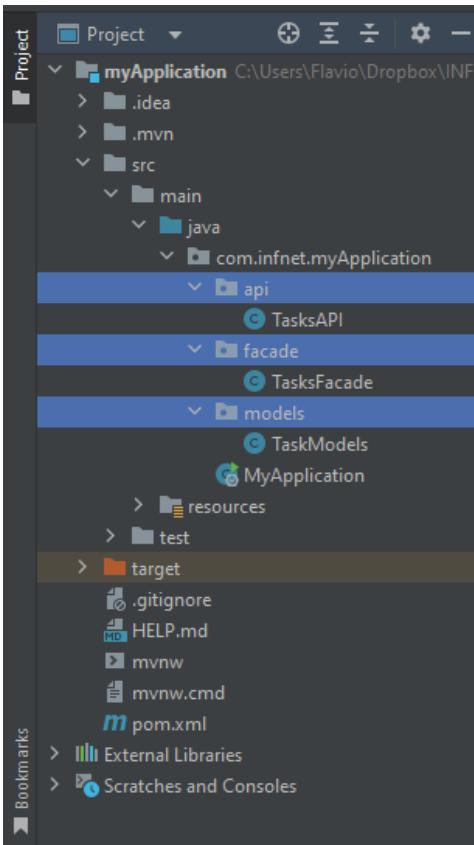
- Boas práticas desenvolvendo com Spring Boot
- Princípios do SOLID

Boas Práticas de Desenvolvimento com Spring Boot

- Usar o Spring Initializr para criar as aplicações

Boas Práticas de Desenvolvimento com Spring Boot

- Nunca usar default package



Boas Práticas de Desenvolvimento com Spring Boot

- Clean Architecture
 - Uncle Bob (Robert C. Martin) levanta quatro abordagens mais comuns para organizar a arquitetura de um projeto de software.

Boas Práticas de Desenvolvimento com Spring Boot

- **1 - Package by Layer:** definição das camadas tradicionalmente na forma horizontal, onde separamos o código com base no que ele faz em uma perspectiva técnica.
 - Por exemplo, camada de interface (web - controllers e views no Spring MVC), camada lógica de negócio (service - models, domains e services no Spring MVC) e camada de persistência (data - repository no Spring Data).

Boas Práticas de Desenvolvimento com Spring Boot

- **2 - Package by Feature:** baseada em recursos relacionados ao conceito de domínio (design orientado ao domínio).
 - Usando uma API fictícia de uma Agência de Viagens como exemplo, com dois domínios: travel e statistics, com todos os recursos (controllers, services e repository) de cada um agrupados nesse contexto.

Boas Práticas de Desenvolvimento com Spring Boot

- **3 - Ports and adapters:** o negócio/domínio deve ser independente dos detalhes técnicos da implementação, como banco de dados.
 - Cria-se o conceito de composição por “dentro” (domínio) e por “fora” (infra).
 - Dessa forma, a camada de configuração de conexão com o banco ficaria em um pacote, todo o contexto de negócio em outra (services, repository e model) e a camada web em outra.

Boas Práticas de Desenvolvimento com Spring Boot

- **4 - Package by Component:** é uma abordagem híbrida, com o objetivo de agrupar todas as responsabilidades relacionadas a um componente de em um único pacote.
 - Dessa forma, um componente seria uma composição das interfaces de service + repository e suas implementações.
 - Assim como o padrão Ports and adapters, essa abordagem mantém a interface do usuário separada desses componentes com a integração backend.

Boas Práticas de Desenvolvimento com Spring Boot

- O uso de um padrão vai depender do contexto de aplicação.
- **A dica:** por mais óbvio que seja, antes de codificar já pensar na estrutura do projeto, com base na lógica do negócio, para que o projeto já nasça sustentável.

Boas Práticas de Desenvolvimento com Spring Boot

- **Injeção de dependência via Construtor ou Setter?**
 - Os conceitos de Inversão de Controle e Injeção de Dependência, que vieram do S.O.L.I.D, são a base para as resoluções de problemas que o Spring se propõe a corrigir.
 - A Inversão de Controle (IoC - Inversion of Control) permite delegar a outro elemento o controle sobre como e quando um objeto deve ser criado e quando um método deve ser executado

Boas Práticas de Desenvolvimento com Spring Boot

- Segundo a documentação do Spring Framework, você pode utilizar um conceito misto em seu projeto, de injeção de dependência com base no construtor e nos métodos setter.
- É uma boa prática usar a injeção nos construtores para dependências ou campos obrigatórios e a injeção nos métodos setter ou métodos de configuração para dependências ou campos opcionais.

Boas Práticas de Desenvolvimento com Spring Boot

● **Aplique conceito de classe de Configuration para setups:**

- Além das configurações via XML, o Spring Boot oferece mais uma forma de configurar a aplicação: por meio de annotation `@Configuration` em uma classe Java.
- Além da Configuration, temos outra um pouco mais completa, que é a `@SpringBootApplication`:
 - Essa annotation habilita: a função de auto-configuração, o scan de componentes e definição de configurações extras.
 - Ou seja, usar o `@SpringBootApplication` é o mesmo que declarar as annotations `@Configuration`, `@ComponentScan` e `@EnableAutoConfiguration` na sua classe.

Boas Práticas de Desenvolvimento com Spring Boot

- **Use sempre @Service na camada de negócio:**
 - Em aplicações no modelo MVC (Model-View-Controller) ou orientado à serviços (SOA), devemos separar muito bem as camadas da aplicação.
 - No desenho tradicional do MVC, temos:
 - **Model:** um POJO (Plain Old Java Object), um Domain com informações de uma entidade – que é uma classe contendo um ou mais construtores, atributos e métodos que encapsulam seu comportamento.
 - **Controller:** é a camada responsável tanto por receber requisições como por enviar a resposta ao usuário.
 - **View:** é a camada de aplicação, onde temos a implementação das páginas web que o usuário acessa e interage.

Boas Práticas de Desenvolvimento com Spring Boot

- **Convenção de nomes no Spring - Spring Bean Naming Conventions:**
 - Usar a convenção padrão do Java para nomes de campos de instância ao nomear os beans: começar com uma letra minúscula e o restante continua em Camel Case.
 - Por exemplo, travelService, statisticService, walletRepository, etc.

O **@Bean** serve para **exportar** uma classe para o Spring, para que ele consiga carregar essa classe e fazer injeção de dependência dela em outras classes.

Boas Práticas de Desenvolvimento com Spring Boot

- **Use H2 para testes unitários:**

- O Spring possui integrações com uma vastidão de tipos de base de dados no projeto Spring Data.
- Um desses bancos de dados disponíveis é o H2.
- O H2 é um SGBDR(sistema de gerenciamento de banco de dados relacional) escrito em Java e open-source, além de possuir uma interface incorporada para executar consultas SQL via browser.

Boas Práticas de Desenvolvimento com Spring Boot

- Crie uma estratégia de versionamento na sua API:
 - Uma boa prática para a manutenibilidade de uma API é fazer o versionamento dela.
 - O versionamento é importante não só para caracterizar a sua evolução, mas também é útil para os clientes, que faz integrações com a sua API em soluções próprias, conhecer o que estão consumindo e caso exista uma nova versão, que ela seja compatível com sua solução.

<https://travels-java-api.herokuapp.com/v1/travels>

Boas Práticas de Desenvolvimento com Spring Boot

- **Faça tratamento de exceção (Spring Exception Handling):**

- Nas versões mais antigas do Spring Framework (< 3.2), existiam duas formas de tratar exceções: usando o HandlerExceptionResolver, para tratar de forma global as exceções (como um dispatcher) ou com @ExceptionHandler em cada um método dentro de cada Controller.
- Posteriormente, nas versões mais atuais (> 3.2), foi criado o ControllerAdvice, que funciona como um interceptador de exceções geradas nos métodos das controllers (via annotation RequestMapping).

Boas Práticas de Desenvolvimento com Spring Boot

- Já no Spring 6, temos uma outra estrutura para tratamento de exceção: o *ResponseStatusException*.
 - Funciona como qualquer uma exceção não-checada (*Unchecked Exceptions*): basta cercar o código do método com um *try-catch* e lançá-la como exceção.
- Quando usar cada uma das abordagens? O *ResponseStatusException* é uma boa alternativa em um contexto de múltiplos cenários de uma mesma exceção sendo tratados de forma diferente.
 - Em um contexto em que as exceções são genéricas e globais, o *ControllerAdvice* é uma forma mais prática de obter maior controle do mecanismo de tratamento de erros na aplicação.

Boas Práticas de Desenvolvimento com Spring Boot

- **Crie a sua documentação de forma automatizada:**

- Ter uma boa documentação é essencial para que a sua API seja fácil de ser utilizada em todos os contextos possíveis, seja na integração com aplicações de terceiros, no trabalho da própria equipe de desenvolvimento ou nas versões sandbox disponibilizadas para o público em geral.
- E imaginem poder criar essa documentação de forma automática, fazendo pouquíssimas configurações?

Boas Práticas de Desenvolvimento com Spring Boot

- Crie a sua documentação de forma automatizada:

- Existem vários frameworks open-source que implementam a OpenAPI, que é uma especificação que define e padroniza vários recursos que uma API deve ter (modelo de dados, URIs, content-types, métodos HTTP aceitos e códigos de respostas).
- E é essa especificação que viabilizou a criação dessas soluções automatizadas, como o Swagger.
- O **Swagger** é ferramenta que cria, com base nas rotas das classes Controller, o esqueleto da API e provê uma interface bem intuitiva para acessar essas rotas (Swagger UI), permitindo qualquer pessoa fazer requests na API sem precisar logar no software original ou usar outras interfaces de apoio como o Postman.

Boas Práticas de Desenvolvimento com Spring Boot

- Crie a sua documentação de forma automatizada:

travel-controller Travel Controller

GET /api-travels/v1/travels Route to find all travels of the API in a period of time

POST /api-travels/v1/travels Route to create travels

GET /api-travels/v1/travels/{id} Route to find a trip by your id in the API

PUT /api-travels/v1/travels/{id} Route to update a trip

DELETE /api-travels/v1/travels/{id} Route to delete a trip in the API

GET /api-travels/v1/travels/byOrderNumber/{orderNumber} Route to find a trip by the orderNumber in the API

SOLID

SOLID - Definições

- SOLID é um acrônimo criado por Michael Feathers, após observar que cinco princípios da orientação a objetos e design de código Criados por Robert C. Martin (a.k.a. Uncle Bob) e abordados no artigo The Principles of OOD poderiam se encaixar nesta palavra.



SOLID - Definições

- S.O.L.I.D: Os 5 princípios da POO:
- **S - Single Responsibility Principle** (Princípio da responsabilidade única)
- **O - Open-Closed Principle** (Princípio Aberto-Fechado)
- **L - Liskov Substitution Principle** (Princípio da substituição de Liskov)
- **I - Interface Segregation Principle** (Princípio da Segregação da Interface)
- **D - Dependency Inversion Principle** (Princípio da inversão da dependência)

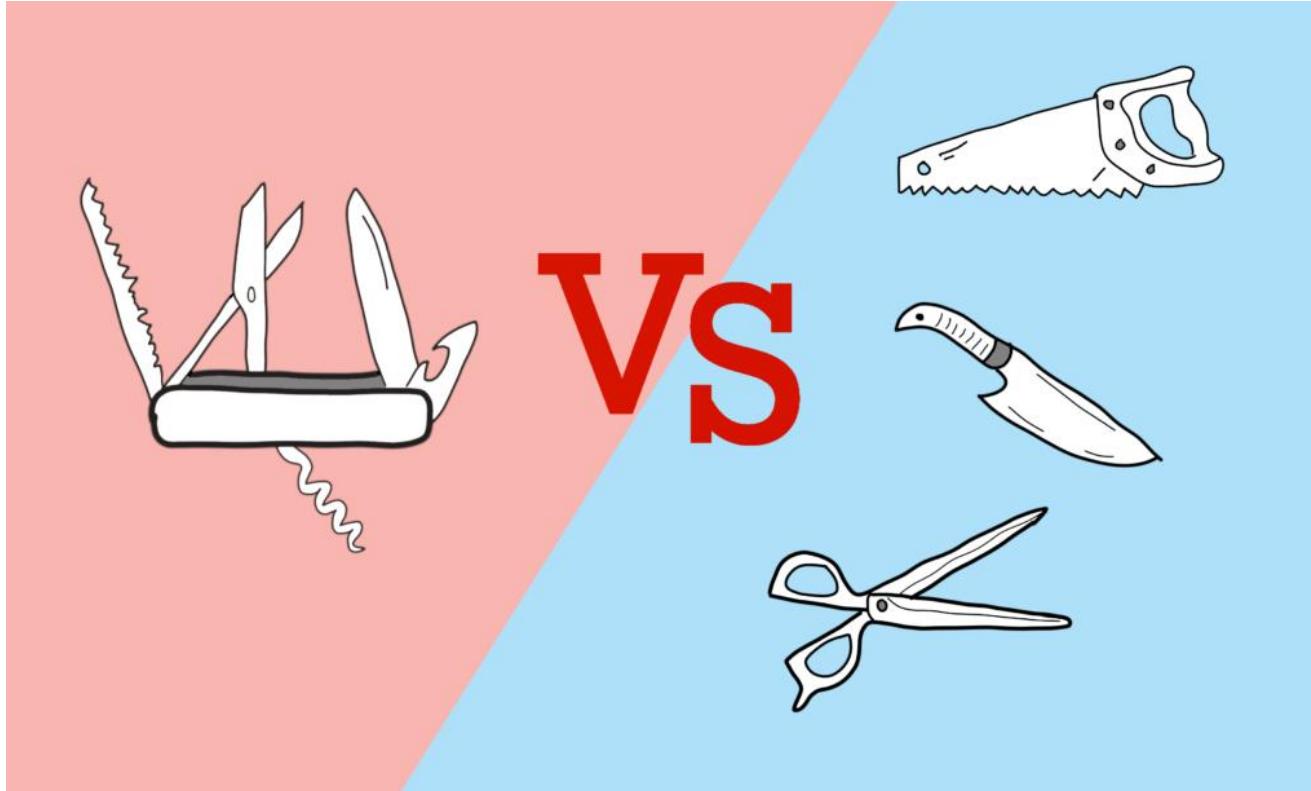
SOLID - Definições

- Todos esses princípios facilitam o desenvolvimento de um software ajudando os desenvolvedores a melhorarem seu código, separando responsabilidades, tornando o código mais limpo e consequentemente mais legível.
- Também fica mais fácil dar manutenção no projeto, reaproveitar trechos de códigos entre muitos outros benefícios.

SRP - Single Responsibility Principle

- De acordo com o Princípio da Responsabilidade Única: Uma classe deve ter um, e somente um motivo para existir.
- Esse princípio declara que uma classe deve ser especializada em um único assunto e possuir apenas uma responsabilidade dentro do software, ou seja, a classe deve ter uma única tarefa ou ação para executar.

SRP - Single Responsibility Principle



SRP - Single Responsibility Principle

- O princípio da responsabilidade única não se limita somente a classes, ele também pode ser aplicado em:
 - Métodos e
 - Funções.
 - Ou seja, tudo que é responsável por executar uma ação, deve ser responsável por apenas aquilo que se propõe a fazer.

Open-Closed Principle

- De acordo com o Princípio Aberto-Fechado: Objetos ou entidades devem estar abertos para extensão, mas fechados para modificação, ou seja, quando novos comportamentos e recursos precisam ser adicionados no software, devemos estender e não alterar o código fonte original.
- Isso porque alterar uma classe pai pode ser perigoso, já que outras classes dentro da aplicação podem estar utilizando-a.
- Ao realizar uma alteração nela, impactará todas as outras que estão utilizando ela.

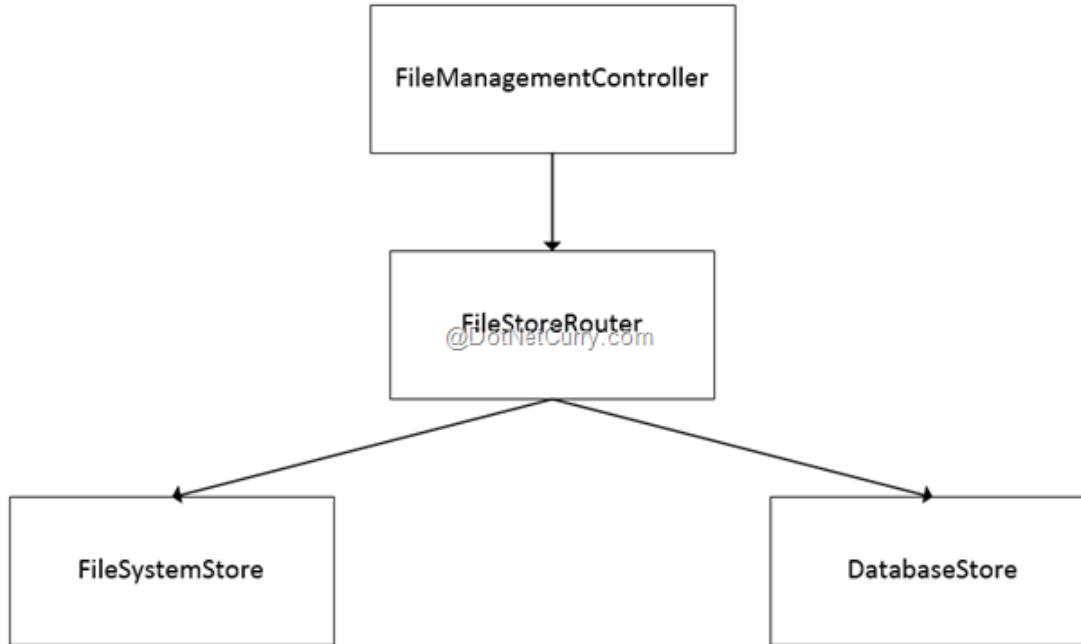
Open-Closed Principle

- Talvez você esteja pensando: “e se a minha classe precisar executar uma outra tarefa?” Você pode simplesmente criar uma nova tarefa dentro da classe.
- A ideia aqui não é não mexer na classe em hipótese alguma, e sim, caso necessário, adicionar uma nova função àquela classe e não alterar o que já existe nela.

Liskov Substitution Principle

- Suponhamos que você tenha uma classe Pessoa. Essa classe contém atributos como nome, CPF, RG... Mas, e se você criar uma outra classe chamada Aluno, quais os atributos que a classe Aluno pode conter?
- Se você pensou em nome, CPF, RG, você está certíssimo, porém, não é interessante criar esses mesmos atributos para a classe aluno, o ideal seria que Aluno herdasse de Pessoa.
- Esse é o princípio que traz a ideia de herança. Temos uma classe pai, que geralmente possui atributos genéricos e temos uma classe filha, que herda os atributos da classe pai e pode ter outros atributos específicos para si mesma.

Liskov Substitution Principle



Interface Segregation Principle

- De acordo com o Princípio da Segregação da Interface: Uma classe não deve ser forçada a implementar interfaces e métodos que não irão utilizar.
- Esse princípio basicamente diz que é melhor criar interfaces mais específicas ao invés de termos uma única interface genérica.

Interface Segregation Principle

- O princípio da segregação de interfaces afirma que uma entidade nunca deve ser forçada a implementar uma interface que contenha elementos que ela nunca utilizará.
- Por exemplo, um Pinguin nunca deve ser forçado a implementar uma interface Bird se essa interface Bird incluir funcionalidades relacionadas ao vôo, já que os pinguins (alerta de spoiler) não podem voar.
- No entanto, podemos demonstrá-lo usando composição.

Dependency Inversion Principle

- De acordo com o Princípio da Inversão de Dependência - Dependa de abstrações e não de implementações.
- De acordo com Uncle Bob, esse princípio pode ser definido da seguinte forma:
 - 1. Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender da abstração.
 - 2. Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.

Dependency Inversion Principle

- “Classes não devem ser forçadas a depender de métodos que não usam.”
- Quando você aplica o princípio de herança, fazendo uma classe herdar da outra, sua classe filha é obrigada a implementar os métodos da classe pai e como você já deve estar imaginando, isso vai contra os princípios do SOLID, pois não é nada interessante que uma classe implementa métodos que não é útil para ela.
- Com o princípio de segregação de interface, é possível implementar somente o que importa para as nossas classes.

Conclusão

- A sistemática dos princípios SOLID tornam o software mais robusto, escalável e flexível, deixando-o tolerante a mudanças, facilitando a implementação de novos requisitos para a evolução e manutenção do sistema.

Conclusão

- O SOLID fornece diversos benefícios na hora de criarmos uma aplicação, dentre eles:
- 1. Segurança: As classes e aplicação se tornam mais seguras graças a divisão de responsabilidades. Com o SOLID, não corremos o risco de ter classes “quebradas” por que alguém alterou algum método de outra classe.
- 2. Manutenção: Com responsabilidades divididas, fica muito mais fácil dar manutenção na aplicação, principalmente se a pessoa desenvolvedora que for dar manutenção não for a mesmo que desenvolveu a aplicação. Como cada parte do código fonte está exatamente aonde deveria estar, fica mais fácil entender.
- 3. Reutilizável: Com POO não é necessária ficar reescrevendo o mesmo código para executar uma determinada tarefa, basta estender de outra classe que já tenha a função que você precisa.



Desenvolvimento de Serviços com Spring Boot

Etapa/Aula 04.1

Professor(a): Flávio Neves

E-mail: flavio.neves@prof.infnet.edu.br