Programação Front-end – AULA
Java Spring

Matheus Moresco
Engenharia de Software - 5º Período
2025/01

O que é o Spring Framework?

- É um **framework Java open-source** que facilita a criação de aplicações robustas, escaláveis, modulares e testáveis.
- Inicialmente focado em **Inversão de Controle (IoC)** e **Injeção de Dependência (DI)**.
- Evoluiu para um **ecossistema completo** que suporta desenvolvimento web, acesso a banco de dados, segurança, microsserviços e mais.

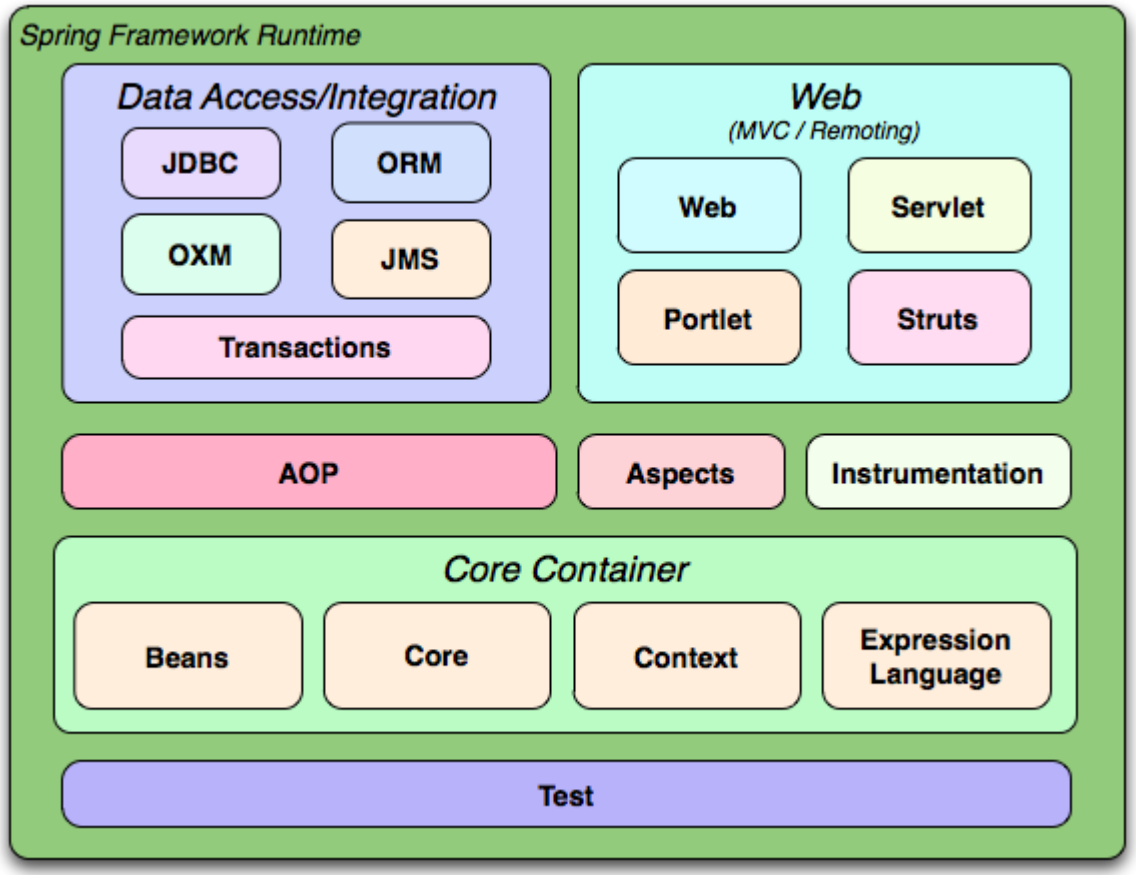
Por que usar Spring?

- Reduz o uso de código repetitivo.
- Oferece uma arquitetura padronizada e desacoplada.
- Facilita testes unitários com dependências injetadas.
- Integra-se facilmente com bancos de dados, APIs REST, mensageria, segurança e nuvem.

Ecossistema Spring

- **Spring Boot**: Criação de aplicações com configuração mínima.
- **Spring MVC**: Desenvolvimento de APIs REST e aplicações web.
- **Spring Data JPA**: Acesso simplificado a banco de dados relacional.
- **Spring Security**: Autenticação e autorização.
- **Spring Cloud**: Desenvolvimento de sistemas distribuídos/microservices.

Ecossistema Spring



Principais Características

Característica	Descrição
Inversão de Controle (IoC)	O contêiner Spring gerencia a criação e ciclo de vida dos objetos.
Injeção de Dependência (DI)	Facilita o desacoplamento entre classes.
Spring Boot	Facilita a criação de aplicações Spring com configuração mínima.
Programação Orientada a Aspectos (AOP)	Separação de preocupações como logging, segurança, transações.
Integração com diversos bancos de dados e frameworks	JPA, Hibernate, MongoDB, etc.
Modularidade	Pode-se usar apenas os módulos necessários (ex: Spring Web sem AOP).

Decorators no Java Spring

- **Anotações (annotations)** funcionam como decoradores de classes, métodos, atributos e parâmetros para adicionar comportamentos automáticos sem a necessidade de escrever código repetitivo.
- Essas anotações são um dos pilares do Spring e permitem que o framework gerencie **injeção de dependências**, **rotas de APIs**, **validações**, **persistência**, **transações** e muito mais.

@Component, @Service, @Repository

Usadas para indicar que uma classe é um bean gerenciado pelo Spring, ou seja, que pode ser injetada automaticamente.

- @Component → genérica, usada em qualquer tipo de classe.
- @Service → usada em classes de lógica de negócio.
- @Repository → usada em classes de acesso a dados (DAO), com tratamento automático de exceções.

```
@Service
public class ProdutoService {
    // lógica de negócio
}
```

@Autowired

- Ela evita que você tenha que instanciar objetos manualmente com new, promovendo o princípio de inversão de controle (IoC), onde a responsabilidade de criar e gerenciar objetos passa a ser do Spring Container.

```
@Service
public class ProdutoService {

    @Autowired
    private ProdutoRepository produtoRepository;
}
```

@RestController e @RequestMapping

- @RestController → diz que a classe expõe endpoints REST.
- @RequestMapping → define o prefixo da URL.
- Também temos @GetMapping, @PostMapping, etc., como atalhos.

```
@RestController
@RequestMapping("/produtos")
public class ProdutoController {

    @GetMapping
    public List<Produto> listar() {
        return produtoService.listarTodos();
    }
}
```

@RequestBody, @PathVariable, @RequestParam

Decoram os parâmetros dos métodos de controller:

- @**RequestBody** → mapeia o corpo da requisição para um objeto Java.
- @**PathVariable** → extrai dados da URL.
- @**RequestParam** → extrai dados da query string.

```
@PostMapping
public Produto criar(@RequestBody ProdutoDTO dto) { ... }

@GetMapping("/{id}")
public Produto buscarPorId(@PathVariable Long id) { ... }
```

@Valid

- Usada para ativar validações nos objetos (geralmente DTOs), com base em anotações como @NotBlank, @Min, @Size, etc.

```
@PostMapping
public ResponseEntity<Produto> salvar(@RequestBody @Valid ProdutoRequestDTO dto) {
    // Validações aplicadas automaticamente
}
```

@Entity, @Id, @GeneratedValue

- Decoradores do JPA (Java Persistence API), usados para mapear classes para tabelas no banco de dados.

```java
@Entity
public class Produto {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
}
```

@Transactional

- Indica que um método (ou classe) deve ser executado dentro de uma transação de banco de dados.

```
@Transactional
public void realizarCompra() {
    // Tudo aqui será executado de forma atômica
}
```

Conclusão

- As **anotações (decorators)** no Spring permitem:
- Declarar **comportamentos complexos com poucas linhas** de código.
- Tornar o projeto mais **limpo, legível e padronizado**.
- Permitir que o Spring **injete dependências**, configure rotas e controle a aplicação com base nesses metadados.

Benefícios do Spring Boot

- Agilidade no desenvolvimento.
- Convenções inteligentes.
- Hot reload com DevTools.
- Configuração via application.properties.

Organização de Projetos no Java Spring Boot

- Ao criar um projeto Spring Boot, a estrutura recomendada é baseada na **separação de responsabilidades**. Cada camada da aplicação tem uma função específica.
- A arquitetura mais comum é:

```
src/
└── main/
    └── java/
        └── com.seuprojeto/
            ├── controller/
            ├── service/
            ├── model/
            ├── repository/
            └── config/ (opcional)
```

Model (ou Entity)

- Responsável por representar os dados da aplicação, geralmente mapeados para uma tabela no banco de dados.
- Mapeada com **@Entity**
- Usa anotações JPA como
  - **@Id**,
  - **@GeneratedValue**,
  - **@Column**,
  - **@ManyToOne**, etc.
- Contém apenas atributos, **getters**, **setters** (e às vezes toString, equals, hashCode).

Model (ou Entity)

```java
@Entity
public class Produto {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nome;
    private double preco;

    @ManyToOne
    private Categoria categoria;

    // Getters e Setters
}
```

Repository

Interface responsável pelo acesso aos dados (CRUD) no banco.

- Estende **JpaRepository**, **CrudRepository** ou outro repositório Spring.
- Spring cria automaticamente a implementação em tempo de execução.
- Suporta métodos derivados por nome, como **findByNome**, **findByPrecoGreaterThan**, etc.

```
public interface ProdutoRepository extends JpaRepository<Produto, Long> {
    List<Produto> findByNomeContainingIgnoreCase(String nome);
}
```

Service

- Contém a lógica de negócio da aplicação.
- Fica entre o controller e o repository.
- Pode validar regras, chamar múltiplos repositórios, tratar exceções, etc.
- Anotado com **@Service**.

```java
@Service
public class ProdutoService {

    @Autowired
    private ProdutoRepository repository;

    public Produto salvar(Produto produto) {
        // validações ou lógica de negócio aqui
        return repository.save(produto);
    }

    public List<Produto> buscarTodos() {
        return repository.findAll();
    }
}
```

Controller

- Responsável por receber requisições HTTP, coordenar chamadas à camada de serviço e retornar respostas.
- Anotado com **@RestController**.
- Define rotas com **@RequestMapping**, **@GetMapping**, **@PostMapping**, etc.
- Pode usar **@PathVariable**, **@RequestParam**, **@RequestBody**.

```java
@RestController
@RequestMapping("/produtos")
public class ProdutoController {

    @Autowired
    private ProdutoService service;

    @GetMapping
    public List<Produto> listar() {
        return service.buscarTodos();
    }

    @PostMapping
    public ResponseEntity<Produto> salvar(@RequestBody @Valid Produto produto) {
        Produto salvo = service.salvar(produto);
        return ResponseEntity.status(HttpStatus.CREATED).body(salvo);
    }
}
```

DTOs (opcional, mas recomendado)

- Data Transfer Object: classe auxiliar usada para separar o que chega/retorna da API do que está no banco de dados.
- Evita expor diretamente a entidade.
- Permite controlar melhor os dados enviados e recebidos.
- Ideal para projetos mais complexos.

```
public class ProdutoResponseDTO {

    private Long id;
    private String nome;
    private double preco;

    public ProdutoResponseDTO(Produto produto) {
        this.id = produto.getId();
        this.nome = produto.getNome();
        this.preco = produto.getPreco();
    }

    // Getters
}
```

Camada de Configuração (opcional)

Pode conter configurações globais como:

- Mapeamento de CORS
- Beans customizados
- Segurança (SecurityConfig)
- Internacionalização, etc.

Exemplo de fluxo completo

Requisição: POST /produtos

1. O **ProdutoController** recebe um JSON no corpo da requisição e transforma em um objeto **Produto**.
2. Chama **ProdutoService.salvar**(produto).
3. A lógica é executada (validação, regras), com o **ProdutoDTO**.
4. O **ProdutoRepository** salva o objeto no banco.
5. O serviço retorna o objeto salvo.
6. O *controller* devolve um 201 Created com os dados do novo produto.

Boas práticas e seus benefícios no uso do Spring Boot

- **Separação de camadas**
  → Permite um código limpo, organizado e de fácil manutenção.
- **Uso da Service Layer**
  → Centraliza as regras de negócio em uma única camada, facilitando a lógica da aplicação.
- **Repository abstrato com Spring Data**
  → Reduz a quantidade de código repetitivo para operações com banco de dados.
- **Facilidade para testes**
  → Permite testar camadas separadamente, como Service e Controller, com mais eficiência.
- **Modularização do sistema**
  → Torna possível reutilizar Services, DTOs, Repositories e outras partes em diferentes APIs ou módulos.