Guia Completo de Spring Boot: Controllers, Models, Services e Repositories

Agosto 2025

Guia detalhado sobre a arquitetura de aplicações Spring Boot com exemplos práticos e anotações JPA

Sumário

1 Introdução

Spring Boot é um framework poderoso para desenvolvimento de aplicações Java, simplificando a configuração e acelerando a criação de APIs robustas. Este guia detalha os componentes principais de uma aplicação Spring Boot Controllers, Models, Services e Repositories com explicações, exemplos práticos e um foco especial nas anotações do Java Persistence API (JPA).

1.1 Objetivo do Guia

Este documento tem como objetivo:

- Explicar o papel de Controllers, Models, Services e Repositories.
- Fornecer exemplos práticos de uma aplicação de gerenciamento de produtos.
- Detalhar as anotações do Spring e JPA, incluindo seus usos e configurações.
- Apresentar melhores práticas para estruturar aplicações Spring Boot.

2 Arquitetura do Spring Boot

Spring Boot organiza aplicações em camadas, cada uma com responsabilidades específicas:

- **Controllers**: Lidam com requisições HTTP e orquestram respostas.
- **Models**: Representam entidades de dados (geralmente mapeadas para tabelas de banco de dados).
- **Services**: Contêm a lógica de negócios.
- **Repositories**: Gerenciam a persistência de dados no banco.

3 Models (Entidades)

As entidades representam objetos do domínio que são mapeados para tabelas em um banco de dados relacional usando JPA.

3.1 Exemplo de Model

Abaixo, uma entidade `Product` com anotações JPA:

```
1  import jakarta.persistence.*;
2  import java.math.BigDecimal;
3
4  @Entity
5  @Table(name = "products")
6  public class Product {
7      @Id
8      @GeneratedValue(strategy = GenerationType.IDENTITY)
9      private Long id;
10
```

```java
    @Column(nullable = false, length = 100)
    private String name;

    @Column(nullable = false)
    private BigDecimal price;

    @Column(columnDefinition = "TEXT")
    private String description;

    // Construtores
    public Product() {}

    public Product(String name, BigDecimal price, String
       description) {
        this.name = name;
        this.price = price;
        this.description = description;
    }

    // Getters e Setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public BigDecimal getPrice() { return price; }
    public void setPrice(BigDecimal price) { this.price = price;
       }
    public String getDescription() { return description; }
    public void setDescription(String description) {
       this.description = description; }
}
```

Listing 1: Classe Product.java

3.2 Anotações Usadas

- `@Entity`: Marca a classe como uma entidade JPA.
- `@Table(name = "products")`: Especifica o nome da tabela no banco.
- `@Id`: Define a chave primária.
- `@GeneratedValue(strategy = GenerationType.IDENTITY)`: Configura a geração automática de IDs.
- `@Column`: Personaliza colunas (ex.: `nullable`, `length`, `columnDefinition`).

4 Repositories

Repositories gerenciam a interação com o banco de dados, oferecendo métodos para operações CRUD.

4.1 Exemplo de Repository

```java
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ProductRepository extends
   JpaRepository<Product, Long> {
    // Consulta personalizada
    List<Product> findByPriceGreaterThan(BigDecimal price);
}
```

Listing 2: Interface ProductRepository.java

4.2 Anotações Usadas

- `@Repository`: Marca a interface como um componente de repositório Spring.
- `JpaRepository<Product, Long>`: Extende métodos CRUD padrão para a entidade `Product` com chave primária `Long`.

4.3 Detalhes

O `JpaRepository` fornece métodos como `save()`, `findById()`, `findAll()`, e `delete()`. Métodos personalizados, como `findByPriceGreaterThan`, são criados seguindo a convenção de nomenclatura do Spring Data.

5 Services

A camada de serviço contém a lógica de negócios, coordenando operações entre controllers e repositories.

5.1 Exemplo de Service

```java
import org.springframework.stereotype.Service;
import org.springframework.beans.factory.annotation.Autowired;
import java.math.BigDecimal;
import java.util.List;
import java.util.Optional;

@Service
public class ProductService {
    private final ProductRepository productRepository;

    @Autowired
    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    public Product createProduct(Product product) {
```

```
17          if (product.getPrice().compareTo(BigDecimal.ZERO) <= 0) {
18              throw new IllegalArgumentException("O preço deve ser
                    maior que zero");
19          }
20          return productRepository.save(product);
21      }
22
23      public Optional<Product> getProductById(Long id) {
24          return productRepository.findById(id);
25      }
26
27      public List<Product> getAllProducts() {
28          return productRepository.findAll();
29      }
30
31      public List<Product>
            getProductsByPriceGreaterThan(BigDecimal price) {
32          return productRepository.findByPriceGreaterThan(price);
33      }
34
35      public Product updateProduct(Long id, Product
            updatedProduct) {
36          Optional<Product> existingProduct =
                productRepository.findById(id);
37          if (existingProduct.isEmpty()) {
38              throw new IllegalArgumentException("Produto não
                    encontrado");
39          }
40          Product product = existingProduct.get();
41          product.setName(updatedProduct.getName());
42          product.setPrice(updatedProduct.getPrice());
43          product.setDescription(updatedProduct.getDescription());
44          return productRepository.save(product);
45      }
46
47      public void deleteProduct(Long id) {
48          if (!productRepository.existsById(id)) {
49              throw new IllegalArgumentException("Produto não
                    encontrado");
50          }
51          productRepository.deleteById(id);
52      }
53  }
```

Listing 3: Classe ProductService.java

5.2 Anotações Usadas

- `@Service`: Marca a classe como um componente de serviço.
- `@Autowired`: Injeta dependências automaticamente.

5.3 Detalhes

O serviço valida a lógica de negócios (ex.: preço maior que zero) e delega operações de persistência ao repositório.

6 Controllers

Controllers lidam com requisições HTTP, mapeando endpoints para métodos de serviço.

6.1 Exemplo de Controller

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import java.math.BigDecimal;
import java.util.List;

@RestController
@RequestMapping("/api/products")
public class ProductController {
    private final ProductService productService;

    @Autowired
    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @PostMapping
    public ResponseEntity<Product> createProduct(@RequestBody
       Product product) {
        Product savedProduct =
           productService.createProduct(product);
        return ResponseEntity.status(201).body(savedProduct);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable
       Long id) {
        return productService.getProductById(id)
            .map(ResponseEntity::ok)
            .orElseGet(() -> ResponseEntity.notFound().build());
    }

    @GetMapping
    public ResponseEntity<List<Product>> getAllProducts() {
        return
           ResponseEntity.ok(productService.getAllProducts());
    }

    @GetMapping("/price-greater-than/{price}")
```

```
36      public ResponseEntity<List<Product>>
           getProductsByPrice(@PathVariable BigDecimal price) {
37          return
               ResponseEntity.ok(productService.getProductsByPriceGreaterThan(pr
38      }
39
40      @PutMapping("/{id}")
41      public ResponseEntity<Product> updateProduct(@PathVariable
           Long id, @RequestBody Product product) {
42          return
               ResponseEntity.ok(productService.updateProduct(id,
               product));
43      }
44
45      @DeleteMapping("/{id}")
46      public ResponseEntity<Void> deleteProduct(@PathVariable Long
           id) {
47          productService.deleteProduct(id);
48          return ResponseEntity.noContent().build();
49      }
50  }
```

Listing 4: Classe ProductController.java

6.2 Anotações Usadas

- `@RestController`: Marca a classe como um controlador REST.
- `@RequestMapping("/api/products")`: Define o prefixo base para os endpoints.
- `@PostMapping`, `@GetMapping`, `@PutMapping`, `@DeleteMapping`: Mapeiam métodos HTTP.
- `@RequestBody`: Vincula o corpo da requisição a um objeto.
- `@PathVariable`: Extrai variáveis da URL.

6.3 Detalhes

O controlador usa `ResponseEntity` para retornar respostas HTTP com códigos de status apropriados (ex.: 201 para criação, 404 para não encontrado).

7 Anotações JPA

O Java Persistence API (JPA) fornece anotações para mapear objetos para bancos de dados relacionais. Abaixo, uma lista completa das anotações disponíveis no JPA (baseado na especificação Jakarta EE):

7.1 Anotações de Mapeamento de Entidade

- `@Entity`: Marca uma classe como entidade persistente.

- `@Table(name = "table_name", schema = "schema_name")`: Especifica a tabela associada.
- `@Id`: Define a chave primária.
- `@GeneratedValue(strategy = GenerationType.AUTO | IDENTITY | SEQUENCE | TABLE)`: Configura a geração de valores para a chave primária.
- `@Column(name = "column_name", nullable = false, length = 255, unique = true)`: Personaliza uma coluna.
- `@Transient`: Marca um campo como não persistente.
- `@Enumerated(EnumType.STRING | ORDINAL)`: Mapeia um enum para uma coluna.
- `@Temporal(TemporalType.DATE | TIME | TIMESTAMP)`: Define o tipo de dado para campos de data/hora.
- `@Lob`: Marca um campo como um grande objeto (ex.: BLOB ou CLOB).

7.2 Anotações de Relacionamento

- `@OneToOne`: Define um relacionamento 1:1.
- `@OneToMany`: Define um relacionamento 1:N.
- `@ManyToOne`: Define um relacionamento N:1.
- `@ManyToMany`: Define um relacionamento N:N.
- `@JoinColumn(name = "foreign_key")`: Especifica a coluna de chave estrangeira.
- `@JoinTable`: Define uma tabela de junção para relacionamentos `@ManyToMany`.
- `@MappedBy`: Marca o lado não proprietário de um relacionamento bidirecional.
- `@Cascade`: Define operações em cascata (ex.: `CascadeType.ALL`).
- `@Fetch(FetchType.LAZY | EAGER)`: Configura a estratégia de carregamento.

7.3 Anotações de Consultas

- `@NamedQuery(name = "queryName", query = "JPQL query")`: Define uma consulta JPQL nomeada.
- `@NamedQueries`: Agrupa várias `@NamedQuery`.
- `@NamedNativeQuery`: Define uma consulta SQL nativa.
- `@QueryHint`: Configura dicas para otimização de consultas.

7.4 Anotações de Configuração Avançada

- `@Access(AccessType.FIELD | PROPERTY)`: Define o acesso via campos ou propriedades.
- `@AttributeOverride`: Sobrescreve mapeamentos de colunas em classes herdadas.

- `@Embeddable`: Marca uma classe como incorporável.
- `@Embedded`: Usa uma classe incorporável em uma entidade.
- `@Inheritance(strategy = InheritanceType.SINGLE_TABLE | TABLE_PER_CLASS | JOINED)`: Define a estratégia de herança.
- `@DiscriminatorColumn`: Especifica a coluna discriminadora para herança.
- `@DiscriminatorValue`: Define o valor discriminador.
- `@SequenceGenerator`: Configura um gerador de sequência.
- `@TableGenerator`: Configura um gerador de tabela para IDs.

7.5 Exemplo com Relacionamento

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

    @Column(nullable = false)
    private String name;

    @OneToMany(mappedBy = "category", cascade = CascadeType.ALL,
       fetch = FetchType.LAZY)
    private List<Product> products;

    // Construtores, Getters e Setters
}
```

Listing 5: Entidade com Relacionamento OneToMany

8 Exemplo Prático Completo

Abaixo, uma aplicação completa de gerenciamento de produtos com dependências configuradas.

8.1 Configuração do Projeto (pom.xml)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>product-management</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.2.0</version>
```

```
    </parent>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
    </dependencies>
</project>
```

Listing 6: pom.xml

8.2 Configuração do Banco (application.properties)

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
```

Listing 7: application.properties

8.3 Classe Principal

```
import org.springframework.boot.SpringApplication;
import
   org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Listing 8: Application.java

8.4 Execução

1. Configure o projeto com Maven.
2. Execute a aplicação com `mvn spring-boot:run`.

3. Teste os endpoints usando ferramentas como Postman:
   - `POST /api/products`: Criar produto.
   - `GET /api/products`:
   - `GET /api/products/{id}`: Obter produto por ID.
   - `PUT /api/products/{id}`: Atualizar produto.
   - `DELETE /api/products/{id}`: Excluir produto.

9 Melhores Práticas

- **Separação de Responsabilidades**: Mantenha cada camada focada em sua função.
- **Validação**: Use `@Valid` com Bean Validation nos controllers.
- **Exceções**: Trate erros com `@ControllerAdvice`.
- **Documentação**: Use Swagger/OpenAPI para documentar a API.
- **Testes**: Escreva testes unitários e de integração para todas as camadas.

10 Conclusão

A arquitetura em camadas do Spring Boot, combinada com anotações JPA, permite criar aplicações robustas e escaláveis. Este guia cobriu Controllers, Models, Services, Repositories e todas as anotações JPA, fornecendo exemplos práticos para uma aplicação de gerenciamento de produtos.

11 Referências

- https://spring.io/projects/spring-boot
- https://jakarta.ee/specifications/persistence/
- https://docs.spring.io/spring-data/jpa/docs/current/reference/html/