




# Kotlin

VS.



Considerações ao iniciar um projeto backend com Spring 

*Comparação entre os códigos e a estrutura de um projeto de API REST utilizando Spring com Kotlin e Java.*

Alexandre A. Santicioli

2024



# Capítulo 01: Introdução ao Kotlin

Nos últimos cinco anos, o uso de Kotlin no desenvolvimento backend tem crescido significativamente, especialmente com frameworks como Spring e Ktor. De acordo com pesquisas recentes, Kotlin tem sido adotado por muitas empresas devido à sua sintaxe concisa e recursos modernos. Em comparação, Java ainda é amplamente utilizado e possui uma base de usuários maior, mas Kotlin está ganhando espaço rapidamente.

## Uso em Backend

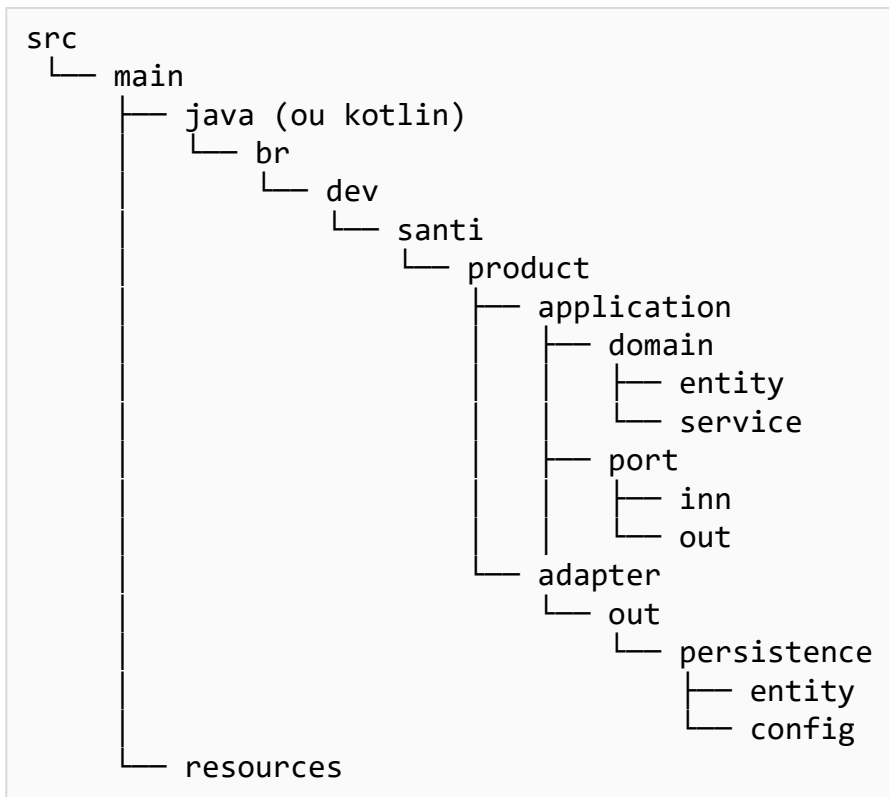
- Kotlin com Spring: Adoção crescente, especialmente em startups e empresas que buscam inovação rápida.
- Kotlin com Ktor: Popular entre desenvolvedores que preferem um framework mais leve e flexível.
- Java com Spring: Continua sendo a escolha dominante em grandes corporações e sistemas legados.





## Capítulo 02: Estrutura de Pastas

Baseado no livro “*Get Your Hands Dirty on Clean Architecture*” de Tom Hombergs, a estrutura de pastas para cada módulo do nosso e-commerce (**core**, **product**, **customer**, **order**) é organizada da seguinte forma:



### Pontos Fortes

- **Separação de Responsabilidades**  
Facilita a manutenção e evolução do código.
- **Modularidade**  
Cada módulo desenvolvido e testado de forma independente.

### Pontos Fracos

- **Complexidade Inicial**  
Pode ser intimidador para desenvolvedores juniores.
- **Sobrecarga de Configuração**  
Requer mais configuração inicial.





# Capítulo 03: Docker e Containers

## Vantagens

- **Isolamento**  
Cada módulo roda em seu próprio container, evitando conflitos.
- **Portabilidade**  
Containers podem ser executados em qualquer ambiente que suporte Docker.

## Desvantagens

- **Complexidade**  
Gerenciar múltiplos containers pode ser desafiador.
- **Recursos**  
Containers consomem recursos do sistema, o que pode ser um problema em ambientes de desenvolvimento com hardware limitado.

## Uso do Docker Compose

- **Desenvolvimento**  
Facilita a orquestração de múltiplos containers em um ambiente de desenvolvimento.
- **Produção**  
Kubernetes é recomendado para orquestração em produção devido à sua escalabilidade e robustez.





# Capítulo 04: Gradle

Gradle é uma ferramenta de automação de build que permite definir tarefas de build de forma declarativa.

## Configuração do Projeto

Projeto Kotlin:

```
plugins {  
    kotlin("jvm") version "1.5.21"  
    id("org.springframework.boot") version "2.5.3"  
    id("io.spring.dependency-management") version "1.0.11.RELEASE"  
}  
  
dependencies {  
    implementation("org.springframework.boot:spring-boot-starter-data-jpa")  
    implementation("org.springframework.boot:spring-boot-starter-web")  
    testImplementation("org.springframework.boot:spring-boot-starter-test")  
}
```

Projeto Java:

```
plugins {  
    id 'org.springframework.boot' version '2.5.3'  
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'  
    id 'java'  
}  
  
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
    testImplementation 'org.springframework.boot:spring-boot-starter-test'  
}
```

Diferenças:

- DSL Kotlin  
A configuração em Kotlin é mais concisa e oferece melhor suporte a IDEs.
- Sintaxe  
Kotlin usa uma sintaxe mais moderna e menos verbosa.





# Capítulo 05: Domínio do DDD

No DDD (*Domain-Driven Design*), o domínio representa a lógica de negócios central da aplicação.

Em uma aplicação real, distribuir os domínios da aplicação em módulos Gradle ajuda a manter a separação de responsabilidades e facilita a manutenção.

Exemplo do Módulo `:product`

## Entidades de Domínio

Projeto Kotlin:

```
package br.dev.santi.product.application.domain.entity

data class Product(val id: Long, val name: String, val price: Double)
```

Projeto Java:

```
package br.dev.santi.product.application.domain.entity;

public class Product {
    private Long id;
    private String name;
    private Double price;

    // Getters and Setters
}
```





## Serviços de Domínio

Projeto Kotlin:

```
package br.dev.santi.product.application.domain.service

class ProductService {
    fun calculateDiscount(product: Product, discount: Double): Double {
        return product.price - discount
    }
}
```

Projeto Java:

```
package br.dev.santi.product.application.domain.service;

public class ProductService {
    public Double calculateDiscount(Product product, Double discount) {
        return product.getPrice() - discount;
    }
}
```

## Portas de Entrada e Saída:

Projeto Kotlin:

```
package br.dev.santi.product.application.port.inn

interface GetProductQuery {
    fun getProductById(id: Long): Product
}
```

Projeto Java:

```
package br.dev.santi.product.application.port.inn;

public interface GetProductQuery {
    Product getProductById(Long id);
}
```





# Capítulo 06: Adaptadores na Arquitetura Hexagonal

Na arquitetura hexagonal, adaptadores são componentes que permitem a comunicação entre o núcleo da aplicação e o mundo externo. Eles são divididos em adaptadores de entrada (*driving adapters*) e adaptadores de saída (*driven adapters*).

Utiliza-se adaptadores para isolar a lógica de negócios das preocupações externas, como interfaces de usuário e bancos de dados. Isso permite que a lógica de negócios evolua independentemente das mudanças nas tecnologias externas.

Projeto Kotlin:

```
// Adaptador de Entrada (Web)
package br.dev.santi.product.adapter.inn.web

@RestController
@RequestMapping("/products")
class ProductController(private val getProductQuery: GetProductQuery) {
    @GetMapping("/{id}")
    fun getProduct(@PathVariable id: Long): ResponseEntity<ProductResponse> {
        val product = getProductQuery.getProductById(id)
        return ResponseEntity.ok(ProductResponse(product.id, product.name,
product.price))
    }
}

// Data Class para RequestBody
data class ProductRequest(val name: String, val price: Double)

// Data Class para Response
data class ProductResponse(val id: Long, val name: String, val price: Double)
```







## Projeto Java:

```
// Adaptador de Entrada (Web)
package br.dev.santi.product.adapter.inn.web;

@RestController
@RequestMapping("/products")
public class ProductController {
    private final GetProductQuery getProductQuery;

    public ProductController(GetProductQuery getProductQuery) {
        this.getProductQuery = getProductQuery;
    }

    @GetMapping("/{id}")
    public ResponseEntity<ProductResponse> getProduct(@PathVariable Long id) {
        Product product = getProductQuery.getProductById(id);
        return ResponseEntity.ok(new ProductResponse(product.getId(),
        product.getName(), product.getPrice()));
    }
}

// Classe para RequestBody
public class ProductRequest {
    private String name;
    private Double price;

    // Getters and Setters
}

// Classe para Response
public class ProductResponse {
    private Long id;
    private String name;
    private Double price;

    public ProductResponse(Long id, String name, Double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    // Getters
}
```

## Diferenças:

- **Sintaxe:** Kotlin é mais conciso e utiliza data classes para representar dados de forma mais simples.
- **Injeção de Dependência:** A injeção de dependência em Kotlin é feita diretamente no construtor, enquanto em Java é necessário declarar explicitamente.





# Capítulo 07: Persistência

A camada de persistência é responsável por armazenar e recuperar dados do banco de dados.

Java Persistence API (JPA) é uma especificação para gerenciamento de dados relacionais em Java.

Flyway é uma ferramenta de migração de banco de dados que permite versionar e aplicar mudanças no esquema do banco de dados.

## Configuração do Flyway

Projeto Kotlin:

```
package br.dev.santi.product.adapter.out.persistence.config

import org.springframework.context.annotation.Configuration
import org.springframework.data.jpa.repository.config.EnableJpaRepositories

@Configuration
@EnableJpaRepositories(basePackages =
    ["br.dev.santi.product.adapter.out.persistence"])
class PersistenceConfig
```

Projeto Java:

```
package br.dev.santi.product.adapter.out.persistence.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@Configuration
@EnableJpaRepositories(basePackages =
    "br.dev.santi.product.adapter.out.persistence")
public class PersistenceConfig {
}
```





# Mapeadores

## Projeto Kotlin:

```
package br.dev.santi.product.adapter.out.persistence.mapper

class ProductMapper {
    fun toEntity(product: Product): ProductEntity {
        return ProductEntity(product.id, product.name, product.price)
    }

    fun toDomain(productEntity: ProductEntity): Product {
        return Product(productEntity.id, productEntity.name, productEntity.price)
    }
}
```

## Projeto Java:

```
package br.dev.santi.product.adapter.out.persistence.mapper;

public class ProductMapper {
    public ProductEntity toEntity(Product product) {
        return new ProductEntity(product.getId(), product.getName(),
product.getPrice());
    }

    public Product toDomain(ProductEntity productEntity) {
        return new Product(productEntity.getId(), productEntity.getName(),
productEntity.getPrice());
    }
}
```





# Capítulo 08: Conceitos de RESTful

RESTful é um estilo de arquitetura para sistemas distribuídos, baseado em recursos e operações HTTP. Ele é amplamente utilizado para criar APIs web.

Spring oferece suporte robusto para a criação de APIs RESTful através de anotações como `@RestController`, `@RequestMapping`, `@GetMapping`, `@PostMapping`, entre outras.

## Implementação dos Adaptadores de Entrada Web - API REST

Projeto Kotlin:

```
package br.dev.santi.product.adapter.inn.web

@RestController
@RequestMapping("/products")
class ProductController(private val getProductQuery: GetProductQuery) {
    @GetMapping("/{id}")
    fun getProduct(@PathVariable id: Long): ResponseEntity<ProductResponse> {
        val product = getProductQuery.getProductById(id)
        return ResponseEntity.ok(ProductResponse(product.id, product.name,
        product.price))
    }
}

// RequestBody (data class)
data class ProductRequest(val name: String, val price: Double)

// Response (data class)
data class ProductResponse(val id: Long, val name: String, val price: Double)
```





## Projeto Java:

```
package br.dev.santi.product.adapter.inn.web;

@RestController
@RequestMapping("/products")
public class ProductController {
    private final GetProductQuery getProductQuery;

    public ProductController(GetProductQuery getProductQuery) {
        this.getProductQuery = getProductQuery;
    }

    @GetMapping("/{id}")
    public ResponseEntity<ProductResponse> getProduct(@PathVariable Long id) {
        Product product = getProductQuery.getProductById(id);
        return ResponseEntity.ok(new ProductResponse(product.getId(),
product.getName(), product.getPrice()));
    }
}

// RequestBody
public class ProductRequest {
    private String name;
    private Double price;

    // Getters and Setters
}

// Response
public class ProductResponse {
    private Long id;
    private String name;
    private Double price;

    public ProductResponse(Long id, String name, Double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    // Getters
}
```





## Capítulo 09: Conclusão

Kotlin oferece uma sintaxe mais moderna e concisa, promovendo a imutabilidade e a segurança do código. A integração com Spring é robusta, e a configuração via `build.gradle.kts` e Docker é direta. Para desenvolvedores juniores, Kotlin pode parecer mais amigável e menos verboso que Java, facilitando a manutenção e a escalabilidade de projetos backend.

