

E-Book  
**Java**  
Intermediário

# Dominando o Java Intermediário

*Da Persistência de Dados  
às APIs Modernas*

Explorando Frameworks, Integrações  
e Boas Práticas para Construir  
Aplicações Profissionais

**Ana Raquel de Holanda**

Powered by ChatGPT &  
Engenharia de Prompts



## Sobre a Autora

Sou uma **estudante de Java** que começou a trilhar este caminho há cerca de **sete meses**, dedicando-me intensamente ao aprendizado dessa linguagem tão poderosa e versátil.

Faço parte da **Fuctura**, uma escola de formação de desenvolvedores, onde cada aula e cada desafio me motivam a ir além, expandindo meus horizontes técnicos e criativos.

Sou também assinante da **plataforma Dio**, onde tenho a oportunidade de aprender e dominar tecnologias de ponta, inclusive aquelas que envolvem **inteligência artificial**.

Foi nesse contexto que descobri o universo da **Engenharia de Prompts**, uma disciplina que me permitiu aliar tecnologia e criatividade na elaboração deste e-book — um projeto construído passo a passo com a ajuda do ChatGPT, explorando capítulo por capítulo tudo aquilo que um desenvolvedor Java intermediário precisa para se destacar.


💡 **É com imenso prazer e satisfação que entrego a vocês este e-book, criado cuidadosamente, que vai muito além de Java POO, abrindo portas para frameworks modernos, testes automatizados, microserviços e muito mais.**



Espero que este material inspire vocês, desenvolvedores, a explorarem novos horizontes, desbravarem frameworks e arquiteturas corporativas, e a trilharem o caminho da excelência na arte de codar. 😊


Boa leitura e bons estudos! 🚀



## Sumário



■ Sobre a Autora.....	1
■ Introdução.....	16
■ Capítulo 1. Introdução aos Frameworks ORM.....	18
1.1 O que é ORM?.....	18
1.2 Por que usar ORM?.....	18
1.3 Principais Frameworks ORM em Java.....	18
♦ JPA (Java Persistence API).....	18
♦ Hibernate ORM.....	18
♦ EclipseLink.....	19
♦ Spring Data JPA.....	19
1.4 Como o ORM funciona?.....	19
1.5 Anatomia de um ORM.....	20
1.6 Quando não usar ORM?.....	20
1.7 Conclusão.....	20
■ Capítulo 2. JPA (Java Persistence API).....	21
2.1 O que é JPA?.....	21
2.2 Principais Conceitos da JPA.....	21
♦ Entidade (Entity).....	21
♦ Contexto de Persistência.....	21
2.3 Ciclo de Vida das Entidades.....	21
♦ Diagrama Simplificado do Ciclo de Vida:.....	22
2.4 Anotações Essenciais da JPA.....	22
♦ @Column.....	22

♦ Relacionamentos.....	22
♦ @OneToMany e @ManyToOne.....	22
♦ @ManyToOne.....	22
♦ @JoinColumn.....	22
2.5 Operações Básicas com o EntityManager.....	23
2.6 Consultas com JPA.....	23
♦ JPQL (Java Persistence Query Language).....	23
♦ Named Queries.....	23
2.7 Estratégias de Geração de Chave Primária.....	23
2.8 Vantagens e Desvantagens da JPA.....	24
2.9 Conclusão.....	24
 Capítulo 3. Hibernate ORM.....	24
3.1 O que é o Hibernate?.....	24
3.2 Arquitetura do Hibernate.....	25
3.3 Configuração Básica.....	25
♦ Via XML (hibernate.cfg.xml).....	25
♦ Via Annotations e persistence.xml (usando JPA).....	25
3.4 Mapeamento de Entidades.....	26
3.5 Consultas no Hibernate.....	26
♦ HQL (Hibernate Query Language).....	26
♦ Criteria API.....	26
♦ SQL Nativo.....	27
3.6 Estratégias de Fetch.....	27
3.7 Cache.....	27
3.8 Transações.....	27

3.9 Vantagens e Desvantagens do Hibernate.....	28
3.10 Conclusão.....	28
 Capítulo 4. Spring Data JPA.....	28
4.1 O que é Spring Data JPA?.....	28
4.2 Arquitetura.....	29
4.3 Configuração.....	29
♦ Dependências no pom.xml.....	29
♦ Configuração no application.properties.....	29
4.4 Criando Repositórios.....	30
4.5 Query Methods.....	30
4.6 Queries Personalizadas.....	30
♦ JPQL.....	30
♦ SQL Nativo.....	30
4.7 Paginação e Ordenação.....	30
4.8 Auditoria.....	31
4.9 Integração com Hibernate.....	31
4.10 Vantagens e Desvantagens.....	31
4.11 Conclusão.....	32
 Capítulo 5. Boas Práticas e Padrões Avançados em JPA/Hibernate.....	32
5.1 Importância das Boas Práticas.....	32
5.2 Gerenciamento de Sessões e EntityManager.....	32
5.3 Lazy vs. Eager Loading.....	33
♦ FetchType.LAZY.....	33
♦ FetchType.EAGER.....	33
5.4 Evitando o N+1 Problem.....	33



♦ Exemplo com HQL:.....	33
5.5 Padrão DTO (Data Transfer Object).....	33
5.6 Uso Inteligente do Cache.....	33
5.7 Controle de Transações.....	34
5.8 Otimizando Consultas.....	34
♦ Projeções em Spring Data JPA:.....	34
5.9 Uso de Índices no Banco.....	34
5.10 Padrões Avançados.....	34
5.11 Conclusão.....	35
 Capítulo 6. Integração de ORM com Microserviços.....	35
6.1 Introdução.....	35
6.2 Desafios Comuns.....	35
6.3 Estratégias de Persistência em Microserviços.....	36
♦ Banco de Dados por Serviço (Database per Service).....	36
♦ Shared Database (Banco Compartilhado).....	36
6.4 Integração com JPA/Hibernate.....	36
♦ Configuração Isolada.....	36
6.5 Comunicação entre Microserviços.....	36
6.6 Consistência de Dados.....	37
♦ Transações Distribuídas (XA).....	37
♦ Exemplo de Outbox Pattern.....	37
6.7 Padrão CQRS (Command Query Responsibility Segregation).....	37
6.8 Cache Distribuído.....	37
6.9 Monitoramento e Observabilidade.....	38
6.10 Boas Práticas.....	38



6.11 Conclusão.....	38
 Capítulo 7. Testes em Ambientes Distribuídos.....	39
7.1 Introdução.....	39
7.2 Pirâmide de Testes em Microserviços.....	39
7.3 Testes Unitários com JPA.....	39
7.4 Testes de Integração com Banco de Dados.....	40
♦ Testcontainers.....	40
♦ Banco de Dados em Memória.....	40
7.5 Testes End-to-End.....	41
7.6 Estratégias para Testar Comunicação entre Microserviços.....	41
7.7 Limpeza e Isolamento de Dados.....	41
7.8 Monitoramento de Consultas.....	41
7.9 Boas Práticas.....	42
7.10 Conclusão.....	42
 Capítulo 8. Estratégias de Versionamento e Migração de Banco de Dados.....	42
8.1 Introdução.....	42
8.2 Problemas Comuns Sem Versionamento.....	42
8.3 Abordagens para Versionamento.....	43
♦ Versionamento Manual (não recomendado).....	43
♦ Versionamento Automatizado (recomendado).....	43
8.4 Ferramentas Populares.....	43
♦ Flyway.....	43
♦ Liquibase.....	43
8.5 Integração com Spring Boot.....	44


♦ Flyway.....	44
♦ Liquibase.....	44
8.6 Boas Práticas.....	44
8.7 Integração com Hibernate.....	44
8.8 Estratégias Avançadas.....	45
8.9 Monitoramento de Versões.....	45
8.10 Conclusão.....	45
 Capítulo 9. Padrões Avançados de Modelagem de Entidades.....	46
9.1 Introdução.....	46
9.2 Herança no Modelo Relacional.....	46
♦ Single Table (TABLE_PER_CLASS).....	46
♦ Joined.....	46
♦ Table per Class.....	47
9.3 Mapeamentos Avançados.....	47
♦ Embedded Objects.....	47
♦ ElementCollection.....	47
♦ Mapeamento Bidirecional.....	47
9.4 Otimizações de Performance.....	48
9.5 Soft Deletes.....	48
9.6 Auditoria e Histórico.....	48
9.7 Boas Práticas.....	49
9.8 Conclusão.....	49
 Capítulo 10. Gerenciamento de Conexões e Pooling.....	49
10.1 Introdução.....	49
10.2 O Problema das Conexões.....	50





10.3 O Conceito de Connection Pool.....	50
10.4 Frameworks de Pooling Populares.....	50
♦ HikariCP.....	50
♦ Apache DBCP.....	50
♦ C3P0.....	50
10.5 Configuração do HikariCP no Spring Boot.....	50
📄 application.properties.....	50
10.6 Monitoramento do Pool.....	51
10.7 Prevenção de Vazamentos.....	51
10.8 Dimensionamento e Performance.....	51
10.9 Integração com Hibernate.....	52
10.10 Boas Práticas.....	52
10.11 Conclusão.....	52
📖 Capítulo 11. Cache de Segunda Nível com Hibernate.....	52
11.1 Introdução.....	52
11.2 Conceitos Fundamentais.....	53
♦ Cache de Primeira Nível.....	53
♦ Cache de Segunda Nível.....	53
11.3 Estrutura do Cache de Segunda Nível.....	53
11.4 Ativando o Cache no Hibernate.....	53
11.5 Configuração com Ehcache.....	54
11.6 Anotando Entidades para Cache.....	54
11.7 Cache de Consultas.....	54
11.8 Boas Práticas.....	55
11.9 Monitoramento.....	55


11.10 Conclusão.....	55
 Capítulo 12. Implementação de Auditoria e Versionamento com Hibernate Envers.....	56
12.1 Introdução.....	56
12.2 Conceito de Auditoria.....	56
12.3 Como o Envers Funciona.....	56
12.4 Configuração do Envers.....	56
12.5 Anotando Entidades para Auditoria.....	57
12.6 Recuperando Histórico.....	57
12.7 Customizando a Tabela de Auditoria.....	57
12.8 Trabalhando com Auditoria Avançada.....	58
♦ Campos Excluídos da Auditoria.....	58
♦ Auditoria de Relacionamentos.....	58
12.9 Estratégias de Retenção.....	58
12.10 Boas Práticas.....	58
12.11 Conclusão.....	58
 Capítulo 13. Boas Práticas de Arquitetura e Organização de Projetos Java .....	59
13.1 Introdução.....	59
13.2 Organização de Pacotes.....	59
13.3 Camadas de Arquitetura.....	59
♦ Apresentação (Controller).....	59
♦ Negócio (Service).....	59
♦ Persistência (Repository).....	60
♦ Modelo (Domain).....	60




13.4 Boas Práticas no Spring Boot.....	60
13.5 Padrões de Projeto.....	60
13.6 Gerenciamento de Dependências.....	60
13.7 Testes.....	60
13.8 Segurança.....	61
13.9 Documentação.....	61
13.10 Monitoramento.....	61
13.11 Conclusão.....	61
 Capítulo 14. Integração Contínua e Entrega Contínua (CI/CD) em Java..	61
14.1 Introdução.....	61
14.2 Conceitos Fundamentais.....	62
♦ Integração Contínua (CI).....	62
♦ Entrega Contínua (CD).....	62
14.3 Ferramentas Populares.....	62
14.4 Estrutura de um Pipeline CI/CD.....	62
14.5 Exemplo com Jenkins.....	62
14.6 Exemplo com GitLab CI/CD.....	63
14.7 Boas Práticas.....	64
14.8 Monitoramento e Alertas.....	64
14.9 Conclusão.....	64
 Capítulo 15. Testes Automatizados e Cobertura de Código em Java.....	65
15.1 Introdução.....	65
15.2 Tipos de Testes.....	65
♦ Testes Unitários.....	65
♦ Testes de Integração.....	65

♦ Testes de Aceitação.....	65
15.3 Estrutura de Testes em Java.....	65
15.4 JUnit 5.....	66
15.5 Mockito.....	66
15.6 Testes de Integração com Spring Boot.....	66
15.7 Cobertura de Código.....	67
♦ Jacoco.....	67
15.8 Análise de Cobertura.....	67
15.9 Boas Práticas.....	67
15.10 Conclusão.....	68
 Capítulo 16. Mensageria Assíncrona com Java (Kafka, RabbitMQ).....	68
16.1 Introdução.....	68
16.2 Conceitos Fundamentais.....	68
♦ Assíncrono x Síncrono.....	68
♦ Fila x Tópico.....	69
16.3 Apache Kafka.....	69
16.3.1 O que é Kafka?.....	69
16.3.2 Conceitos-Chave.....	69
16.4 RabbitMQ.....	69
16.4.1 O que é RabbitMQ?.....	69
16.4.2 Conceitos-Chave.....	69
16.5 Configuração com Spring Boot.....	69
♦ Spring Kafka.....	69
♦ Spring AMQP (RabbitMQ).....	70
16.6 Boas Práticas.....	70

16.7 Padrões Arquiteturais.....	71
16.8 Conclusão.....	71
■ Capítulo 17. Microserviços e Comunicação Assíncrona.....	71
17.1 Introdução.....	71
17.2 Microserviços em Java.....	71
17.3 Comunicação entre Microserviços.....	72
♦ Comunicação Síncrona (REST).....	72
♦ Comunicação Assíncrona (Mensageria).....	72
17.4 Padrões de Comunicação Assíncrona.....	72
♦ Event-Driven Architecture (EDA).....	72
♦ Command-Query Responsibility Segregation (CQRS).....	72
♦ Saga Pattern.....	72
17.5 Implementando com Spring Boot e Kafka.....	72
17.6 Implementando com Spring Boot e RabbitMQ.....	73
17.7 Boas Práticas.....	73
17.8 Observabilidade.....	73
17.9 Conclusão.....	73
■ Capítulo 18. Segurança em Aplicações Java (OAuth2, JWT).....	74
18.1 Introdução.....	74
18.2 Conceitos Fundamentais.....	74
♦ Autenticação x Autorização.....	74
♦ OAuth2.....	74
♦ JWT (JSON Web Token).....	74
18.3 Implementando OAuth2 com Spring Security.....	75
18.3.1 Adicionando Dependências.....	75

18.3.2 Configurando o Resource Server.....	75
18.4 Gerando Tokens JWT.....	75
18.5 Validando Tokens JWT.....	76
18.6 Boas Práticas.....	76
18.7 Integração com Microserviços.....	76
18.8 Ferramentas Recomendadas.....	76
18.9 Conclusão.....	76
 Capítulo 19. APIs REST e HATEOAS em Java.....	77
19.1 Introdução.....	77
19.2 Fundamentos de REST.....	77
19.3 O que é HATEOAS?.....	77
19.4 Implementando APIs REST com Spring Boot.....	78
19.4.1 Dependências.....	78
19.4.2 Exemplo de Endpoint Básico.....	78
19.5 Adicionando HATEOAS com Spring HATEOAS.....	78
19.5.1 Dependência.....	78
19.5.2 Usando RepresentationModel.....	79
19.5.3 Construindo o Modelo com Links.....	79
19.6 Boas Práticas.....	79
19.7 Conclusão.....	80
 Capítulo 20. Padrões de Versionamento de API.....	80
20.1 Introdução.....	80
20.2 Por que versionar?.....	80
20.3 Estratégias de Versionamento.....	81
♦ Versionamento na URL.....	81

♦ Versionamento via Header.....	81
♦ Versionamento por Media Type.....	81
20.4 Implementando Versionamento em Spring Boot.....	81
20.4.1 Versionamento na URL.....	81
20.4.2 Versionamento via Header.....	82
20.5 Boas Práticas.....	82
20.6 Conclusão.....	82
 Capítulo 21. Testes Automatizados em Java (JUnit, Mockito, Testcontainers).....	83
21.1 Introdução.....	83
21.2 Tipos de Testes.....	83
21.3 Frameworks Essenciais.....	83
♦ JUnit.....	83
♦ Mockito.....	83
♦ Testcontainers.....	84
21.4 Testes Unitários com JUnit.....	84
21.4.1 Dependência (JUnit 5).....	84
21.4.2 Exemplo de Teste Unitário.....	84
21.5 Testes com Mockito.....	84
21.5.1 Dependência.....	84
21.5.2 Exemplo de Uso.....	85
21.6 Testes de Integração com Testcontainers.....	85
21.6.1 Dependência.....	85
21.6.2 Exemplo com PostgreSQL.....	85
21.7 Boas Práticas.....	86

21.8 Conclusão.....	86
 Capítulo 22. Estudos de Casos.....	87
22.1 Introdução.....	87
22.2 A Sinfonia da Arquitetura Moderna.....	87
♦ Frameworks ORM (JPA e Hibernate).....	87
♦ Spring Boot como Plataforma Principal.....	87
♦ APIs REST e HATEOAS.....	88
♦ Versionamento de API.....	88
♦ Testes Automatizados.....	88
22.3 Integração Contínua e Entrega Contínua (CI/CD).....	88
22.4 Estudos de Casos Reais.....	89
🔍 Startup Ágil (5-10 devs).....	89
🔍 Empresa de Médio Porte (50-100 devs).....	89
🔍 Corporação Multinacional (200+ devs).....	89
22.5 Conclusão.....	89
 Conclusão do E-Book Java Nível Intermediário.....	90
 Bibliografia.....	90



## Introdução

Neste e-book **Java Nível Intermediário**, embarcaremos em uma jornada metódica e profunda pelo vasto universo da linguagem Java — partindo das bases sólidas até as orquestras complexas que movem as empresas de tecnologia modernas.

Aqui, cada capítulo é como uma nota que compõe a sinfonia da programação, trazendo desde frameworks poderosos até boas práticas de integração e testes automatizados. Você encontrará não apenas códigos, mas também explicações meticulosas, 💡 dicas valiosas e análises práticas para que você possa **entender, aplicar e evoluir** como desenvolvedor.

Ao longo dos **21 capítulos**, você irá:

- ✓ **Capítulo 1 — Frameworks ORM:** Explorar os conceitos essenciais de mapeamento objeto-relacional com Hibernate e JPA.
- ✓ **Capítulo 2 — JPA (Java Persistence API):** Dominar a especificação oficial de persistência em Java.
- ✓ **Capítulo 3 — Hibernate:** Mergulhar no framework mais utilizado para ORM e suas melhores práticas.
- ✓ **Capítulo 4 — Spring Data JPA:** Aprender como simplificar o acesso a dados com este poderoso módulo.
- ✓ **Capítulo 5 — Versionamento de APIs:** Entender como gerenciar a evolução das suas APIs sem quebrar contratos.
- ✓ **Capítulo 6 — API RESTful com Spring Boot:** Criar APIs limpas, escaláveis e seguras.
- ✓ **Capítulo 7 — Validações:** Garantir a integridade e a robustez dos dados trafegados em sua aplicação.
- ✓ **Capítulo 8 — Documentação de APIs:** Produzir documentação elegante e útil com Swagger/OpenAPI.
- ✓ **Capítulo 9 — Boas Práticas em Código:** Desenvolver códigos legíveis, manuteníveis e de alta qualidade.
- ✓ **Capítulo 10 — Controle de Transações:** Dominar a consistência e a confiabilidade das operações no banco de dados.
- ✓ **Capítulo 11 — Integração com Banco de Dados:** Conectar-se e interagir de forma eficaz com bancos relacionais e não relacionais.
- ✓ **Capítulo 12 — Padrões de Projeto:** Aplicar soluções elegantes e reconhecidas para problemas recorrentes.
- ✓ **Capítulo 13 — Segurança de Aplicações Java:** Proteger suas aplicações contra ameaças comuns.
- ✓ **Capítulo 14 — Autenticação e Autorização:** Implementar segurança robusta usando Spring Security.
- ✓ **Capítulo 15 — Microserviços:** Arquitetar sistemas distribuídos escaláveis com Java.
- ✓ **Capítulo 16 — Mensageria Assíncrona:** Integrar sistemas desacoplados com Kafka e RabbitMQ.
- ✓ **Capítulo 17 — Logging e Monitoramento:** Acompanhar e manter a saúde das aplicações em produção.
- ✓ **Capítulo 18 — Configuração e Gerenciamento de Dependências:** Usar

Maven/Gradle de forma eficiente.

✓ **Capítulo 19 — Testes Automatizados:** Implementar testes unitários e de integração para qualidade contínua.

✓ **Capítulo 20 — Integração Contínua e Entrega Contínua (CI/CD):** Automatizar builds, testes e deploys.

✓ **Capítulo 21 — Estudos de Casos:** Ver como tudo isso se conecta na prática nas empresas de tecnologia.

💡 **Dica:** Leia cada capítulo como quem escuta um instrumento distinto em uma orquestra; juntos, eles formam a harmonia que o mercado espera de um desenvolvedor Java de nível intermediário.

Prepare-se para mergulhar em uma jornada de aprendizado técnico e poético, onde cada linha de código é uma partitura a ser explorada. Boa leitura e bom estudo! 🚀

---

## Capítulo 1. Introdução aos Frameworks ORM

### 1.1 O que é ORM?

**Object-Relational Mapping (ORM)** é uma técnica que conecta o paradigma de objetos (Java) com o modelo relacional (banco de dados). Essa técnica mapeia **objetos Java em tabelas** de bancos de dados relacionais, traduzindo automaticamente operações de leitura, escrita e atualização.

💡 **Dica:** Pense no ORM como uma ponte: você trabalha com objetos e o framework cuida da tradução para SQL.

---

### 1.2 Por que usar ORM?

1. **Produtividade:** Reduz a quantidade de código SQL repetitivo.
  2. **Manutenção:** Facilita a manutenção, pois você trabalha em alto nível, focando na lógica de negócios.
  3. **Portabilidade:** Abstrai detalhes do banco de dados, facilitando a troca de SGBD (MySQL, PostgreSQL, Oracle).
  4. **Cache e Performance:** Alguns frameworks implementam caches que reduzem o número de queries.
  5. **Segurança:** Minimiza riscos de SQL Injection ao gerar queries de forma parametrizada.
-

## 1.3 Principais Frameworks ORM em Java

### ♦ JPA (Java Persistence API)

- **O que é?:** É uma especificação que define um conjunto de APIs para persistência de dados em Java.
- **Como funciona?:** Define interfaces e anotações, mas precisa de uma implementação (Hibernate, EclipseLink).
- **Diferencial:** Padroniza a persistência, permitindo trocar implementações sem alterar o código da aplicação.

### ♦ Hibernate ORM

- **O que é?:** Framework de persistência de dados mais popular no ecossistema Java.
- **Como funciona?:** Implementa a JPA e adiciona recursos avançados (cache, herança, interceptadores).
- **Diferencial:** Recursos além da especificação JPA, como mapeamentos avançados, otimizações de performance e suporte a múltiplos bancos de dados.

### ♦ EclipseLink

- **O que é?:** Outra implementação da JPA.
- **Como funciona?:** Mantido pela Eclipse Foundation; bom suporte para integração corporativa.
- **Diferencial:** Integração com outros frameworks como MOXy (marshalling XML/JSON).

### ♦ Spring Data JPA

- **O que é?:** Extensão do Spring Framework que facilita a implementação de repositórios JPA.
  - **Como funciona?:** Gera implementações automáticas de repositórios a partir de interfaces, reduzindo ainda mais o código.
  - **Diferencial:** Integração perfeita com Spring Boot, suporte a queries personalizadas, paginação e ordenação.
- 

## 1.4 Como o ORM funciona?

Imagine a classe Java abaixo:

```
@Entity
@Table(name = "clientes")
public class Cliente {
```

```

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

private String nome;
}

```

\Explicar uma regra dentro do código:

- **@Entity:** Declara que a classe é persistente.
- **@Table(name = "clientes"):** Mapeia a tabela de banco de dados.
- **@Id:** Indica a chave primária.
- **@GeneratedValue:** Configura a estratégia de geração da chave.

O ORM cuida de: ☒ Criar tabelas no banco (se configurado).

☒ Gerar INSERT, SELECT, UPDATE e DELETE.

☒ Sincronizar alterações no objeto com o banco.

---

## 1.5 Anatomia de um ORM

Um ORM geralmente possui:

- **Entity Manager ou Session:** Responsável por gerenciar o ciclo de vida das entidades (inserir, atualizar, excluir).
  - **Mapeamento:** Define como classes e atributos são mapeados para tabelas e colunas.
  - **Consultas:** Fornece APIs para construir queries (JPQL, Criteria API).
  - **Transações:** Garante a atomicidade das operações no banco de dados.
  - **Cache:** Pode armazenar entidades e resultados de queries para melhorar performance.
- 

## 1.6 Quando não usar ORM?

Embora seja poderoso, o ORM não é solução para todos os casos:

☒ Quando o sistema exige queries complexas ou de alto desempenho onde o SQL nativo é mais eficiente.

☒ Quando a aplicação não utiliza objetos (por exemplo, processamento em lote ou ETL puro).

☒ Quando a curva de aprendizado é um problema imediato para a equipe.

 **Dica:** Se necessário, é possível usar SQL nativo no Hibernate e no JPA via

`@NamedNativeQuery` ou `EntityManager.createNativeQuery()`.

---

## 1.7 Conclusão

Os frameworks ORM em Java transformam a forma como você interage com bancos de dados, focando na produtividade e na organização do código. Eles são essenciais para aplicações corporativas modernas, sendo a JPA o ponto de partida para escolher uma implementação como Hibernate ou EclipseLink. O Spring Data JPA oferece ainda mais produtividade para quem já usa Spring Boot.


No próximo capítulo, vamos mergulhar fundo na **JPA**, explorando como criar entidades, gerenciar o ciclo de vida e construir relacionamentos entre tabelas de forma prática e elegante.

---

## Capítulo 2. JPA (Java Persistence API)

### 2.1 O que é JPA?

A **Java Persistence API (JPA)** é a especificação oficial da plataforma Java para persistência de dados em aplicações corporativas. Ela padroniza o mapeamento objeto-relacional (ORM), definindo interfaces e anotações que permitem transformar objetos Java em tabelas de banco de dados relacionais.

 **Dica:** O JPA é apenas a **especificação**; ele não implementa a persistência por si só. Para isso, usamos frameworks como **Hibernate**, **EclipseLink** ou **OpenJPA**, que implementam a JPA.

---

### 2.2 Principais Conceitos da JPA

#### ◆ Entidade (Entity)

Uma **entidade** é uma classe Java que representa uma tabela no banco de dados.

```
@Entity
@Table(name = "clientes")
public class Cliente {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
}
```


\Explicar uma regra dentro do código:

- **@Entity:** Marca a classe como persistente.
  - **@Table(name = "clientes"):** Mapeia a tabela no banco.
  - **@Id:** Identifica a chave primária.
  - **@GeneratedValue:** Configura a estratégia de geração da chave primária (IDENTITY, AUTO, SEQUENCE, TABLE).
-

### ♦ Contexto de Persistência

O **contexto de persistência** é o ambiente gerenciado onde as entidades são rastreadas e sincronizadas com o banco de dados. Ele:  Garante consistência de dados.

 Permite o gerenciamento do ciclo de vida da entidade (persistir, atualizar, remover).

 **Dica:** O `EntityManager` é a porta de entrada para gerenciar o contexto de persistência.

---


## 2.3 Ciclo de Vida das Entidades

O ciclo de vida de uma entidade na JPA é composto por 4 estados principais:

Estado	Descrição
<b>New</b>	A entidade é nova, não está no banco e não é gerenciada.
<b>Managed</b>	A entidade está sendo gerenciada pelo <code>EntityManager</code> .
<b>Detached</b>	A entidade está desanexada do contexto, não é mais rastreada.
<b>Removed</b>	A entidade está marcada para remoção no banco.

### ♦ Diagrama Simplificado do Ciclo de Vida:

```
[NEW] -> persist() -> [MANAGED] -> remove() -> [REMOVED]
[MANAGED] -> detach() -> [DETACHED] -> merge() -> [MANAGED]
```

 **Dica:** Use o método `merge()` para reanexar entidades detached ao contexto.

---

## 2.4 Anotações Essenciais da JPA

### ♦ @Column

Configura detalhes da coluna: nome, tipo, tamanho e restrições.

```
@Column(name = "nome_cliente", length = 100, nullable = false)
private String nome;
```

### ♦ Relacionamentos

#### ♦ @OneToMany e @ManyToOne

```
@OneToMany(mappedBy = "cliente")
private List<Pedido> pedidos;
```

\Explicar uma regra dentro do código:

- `mappedBy`: Define o lado inverso do relacionamento.

### ♦ @ManyToOne

```
@ManyToOne
@JoinColumn(name = "cliente_id")
private Cliente cliente;
```

### ♦ @JoinColumn

Define a coluna de chave estrangeira.

---

## 2.5 Operações Básicas com o EntityManager

```
EntityManager em = entityManagerFactory.createEntityManager();
em.getTransaction().begin();
```

```
Cliente cliente = new Cliente();
cliente.setNome("Ana");
em.persist(cliente);
```

```
em.getTransaction().commit();
em.close();
```

\Explicar uma regra dentro do código:

- `em.getTransaction().begin();`: Inicia uma transação.
- `em.persist(cliente);`: Marca a entidade como **Managed**.
- `em.getTransaction().commit();`: Confirma a transação no banco.
- `em.close();`: Libera os recursos.

 **Dica:** Use sempre transações para garantir a consistência.

---

## 2.6 Consultas com JPA

### ♦ JPQL (Java Persistence Query Language)


```
List<Cliente> clientes = em.createQuery("SELECT c FROM Cliente c WHERE
c.nome = :nome", Cliente.class)
    .setParameter("nome", "Ana")
    .getResultList();
```

\Explicar uma regra dentro do código:

- `"SELECT c FROM Cliente c"`: Sintaxe orientada a objetos, não ao banco de dados.
- `:nome`: Parâmetro nomeado.

### ♦ Named Queries

```
@NamedQuery(name = "Cliente.findByNome",
    query = "SELECT c FROM Cliente c WHERE c.nome = :nome")
```

 **Dica:** Permite reaproveitar consultas de forma padronizada.

---

## 2.7 Estratégias de Geração de Chave Primária


- **AUTO:** Delega ao provedor JPA a escolha da estratégia.
- **IDENTITY:** Usa auto-incremento do banco.
- **SEQUENCE:** Usa uma sequência específica (Oracle, PostgreSQL).
- **TABLE:** Usa uma tabela auxiliar.

```
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator =  
"cliente_seq")  
@SequenceGenerator(name = "cliente_seq", sequenceName = "seq_cliente",  
allocationSize = 1)  
private Long id;
```

---

## 2.8 Vantagens e Desvantagens da JPA

Vantagens	Desvantagens
Abstrai detalhes SQL repetitivo	Performance pode ser inferior a JDBC puro
Mapeamento automático de objetos	Curva de aprendizado inicial
Integração com frameworks modernos (Spring)	Pouco controle sobre SQL gerado em casos complexos

 **Dica:** Use `@NamedNativeQuery` se precisar executar SQL nativo em casos especiais.

---

## 2.9 Conclusão

Neste capítulo, exploramos a essência da JPA: desde o conceito de entidade até as operações básicas com o `EntityManager`, ciclo de vida das entidades, consultas JPQL e estratégias de chave primária. A JPA é a base para frameworks mais avançados como **Hibernate** e **Spring Data JPA**, que aprofundaremos nos próximos capítulos.

No próximo capítulo, mergulharemos no **Hibernate**, explorando configurações avançadas, mapeamentos complexos e estratégias de performance.


---



## Capítulo 3. Hibernate ORM

### 3.1 O que é o Hibernate?

O **Hibernate ORM** é o framework de persistência mais utilizado no universo Java. Ele implementa a especificação JPA, mas também oferece recursos próprios que vão além da API padrão, como cache de segundo nível, suporte a herança, otimizações de desempenho, interceptadores e muito mais.

 **Dica:** O Hibernate é a base de muitas implementações JPA, inclusive do **Spring Data JPA**.

---

### 3.2 Arquitetura do Hibernate

O Hibernate possui uma arquitetura robusta que facilita a integração com diversos bancos de dados. Seus principais componentes são:

- **SessionFactory:** Fabrica de sessões; responsável por criar objetos de sessão (`Session`) para interagir com o banco de dados.
  - **Session:** Gerencia o ciclo de vida das entidades, operações CRUD e transações.
  - **Transaction:** Gerencia as transações com commit ou rollback.
  - **Configuration:** Responsável pela configuração do Hibernate (via XML ou API fluente).
  - **Query:** Suporte para HQL (Hibernate Query Language) e Criteria API.
- 


### 3.3 Configuração Básica

Existem duas maneiras principais de configurar o Hibernate:

#### ♦ Via XML (`hibernate.cfg.xml`)

```
<hibernate-configuration>
  <session-factory>
    <property
name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</
property>
    <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/exemplo</
property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">senha</property>
    <property
name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property
>
    <property name="hibernate.hbm2ddl.auto">update</property>
    <mapping class="com.exemplo.Cliente"/>
  </session-factory>
```

</hibernate-configuration>

 **Dica:** `hibernate.hbm2ddl.auto` pode ter valores como `update`, `create`, `create-drop`, ou `validate` — cada um define como o Hibernate gerencia as tabelas.

---

#### ♦ Via Annotations e `persistence.xml` (usando JPA)

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
version="2.2">
  <persistence-unit name="exemploPU">
    <class>com.exemplo.Cliente</class>
    <properties>
      <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/exemplo"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.password" value="senha"/>
      <property name="javax.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver"/>
      <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL8Dialect"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
    </properties>
  </persistence-unit>
</persistence>
```

---

## 3.4 Mapeamento de Entidades

Embora o Hibernate suporte XML, o uso de anotações é amplamente recomendado por ser mais legível e menos verboso.

```
@Entity
@Table(name = "clientes")
public class Cliente {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "nome", nullable = false, length = 100)
    private String nome;
}
```

\Explicar uma regra dentro do código:

- `@Column(name = "nome", nullable = false, length = 100)`: **Controla as restrições da coluna.**
- 

## 3.5 Consultas no Hibernate

O Hibernate oferece três formas principais de consultar dados:

#### ♦ HQL (Hibernate Query Language)

```
String hql = "FROM Cliente c WHERE c.nome = :nome";
```

```
List<Cliente> clientes = session.createQuery(hql, Cliente.class)
    .setParameter("nome", "Ana")
    .getResultList();
```

💡 **Dica:** HQL é orientado a objetos, usando o nome das classes em vez das tabelas.

---

#### ♦ Criteria API

```
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Cliente> cq = cb.createQuery(Cliente.class);
Root<Cliente> root = cq.from(Cliente.class);
cq.select(root).where(cb.equal(root.get("nome"), "Ana"));
List<Cliente> clientes = session.createQuery(cq).getResultList();
```

---

#### ♦ SQL Nativo

```
String sql = "SELECT * FROM clientes WHERE nome = :nome";
List<Cliente> clientes = session.createNativeQuery(sql, Cliente.class)
    .setParameter("nome", "Ana")
    .getResultList();
```

---

### 3.6 Estratégias de Fetch

O Hibernate suporta **fetching strategies** para controlar como associações são carregadas:

- **Lazy (preguiçoso):** Carrega sob demanda (padrão).
- **Eager (ávido):** Carrega imediatamente.

```
@OneToMany(mappedBy = "cliente", fetch = FetchType.LAZY)
private List<Pedido> pedidos;
```

💡 **Dica:** Use **Lazy** para coleções grandes e carregue explicitamente com `Hibernate.initialize()` quando necessário.

---

### 3.7 Cache

O Hibernate suporta cache de primeiro nível (por sessão) e cache de segundo nível (compartilhado).

- **Primeiro nível:** Automaticamente habilitado, gerencia entidades dentro da mesma `Session`.
  - **Segundo nível:** Configurável, melhora a performance com frameworks como Ehcache ou Infinispan.
- 

### 3.8 Transações

```
Session session = sessionFactory.openSession();
```

```
Transaction tx = session.beginTransaction();

Cliente cliente = new Cliente();
cliente.setNome("Ana");
session.persist(cliente);

tx.commit();
session.close();
```

💡 **Dica:** Sempre envolva operações de escrita em transações para evitar inconsistências.

---

## 3.9 Vantagens e Desvantagens do Hibernate

Vantagens	Desvantagens
Suporte robusto a JPA	Pode gerar queries ineficientes em cenários complexos
Extensão além da especificação JPA	Configuração detalhada pode ser extensa
Cache integrado (segundo nível)	Consumo de memória maior
Suporte a herança e polimorfismo	Curva de aprendizado mais íngreme

---

## 3.10 Conclusão


Neste capítulo, exploramos o Hibernate ORM, desde sua configuração até o uso avançado de HQL, Criteria API e cache. O Hibernate é a espinha dorsal de muitas aplicações corporativas, especialmente quando combinado com o Spring Data JPA. No próximo capítulo, aprofundaremos o **Spring Data JPA**, abordando repositórios automáticos, queries personalizadas e integração com Spring Boot para uma abordagem moderna e poderosa.

---

## Capítulo 4. Spring Data JPA

### 4.1 O que é Spring Data JPA?

**Spring Data JPA** é um projeto da **Spring** que integra a JPA (Java Persistence API) com o ecossistema Spring, fornecendo uma camada de abstração poderosa e simplificada para persistência de dados. Ele elimina grande parte do código repetitivo necessário para implementar repositórios, oferecendo métodos prontos para CRUD, paginação e consultas dinâmicas.

 **Dica:** O Spring Data JPA é construído sobre o Hibernate (ou outro provedor JPA) e herda todas as funcionalidades robustas do ORM.

---

### 4.2 Arquitetura

A arquitetura do Spring Data JPA é composta por:


- **Repository:** Interface base para CRUD, paginação e queries personalizadas.
  - **JpaRepository:** Interface que estende `PagingAndSortingRepository` e adiciona suporte completo a JPA.
  - **Query Methods:** Métodos baseados na convenção de nomes para gerar queries automaticamente.
  - **Custom Queries:** Queries personalizadas usando JPQL ou SQL nativo.
  - **Auditoria:** Controle automático de auditoria (quem criou/modificou, data/hora).
- 

### 4.3 Configuração

Com Spring Boot, a configuração é extremamente simplificada:

#### ◆ Dependências no `pom.xml`

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
```

 **Dica:** Substitua o `mysql-connector-java` pelo driver adequado ao seu banco.

---

#### ◆ Configuração no `application.properties`

```
spring.datasource.url=jdbc:mysql://localhost:3306/exemplo
spring.datasource.username=root
spring.datasource.password=senha
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

\Explicar uma regra dentro do código:

- `spring.jpa.hibernate.ddl-auto=update`: Atualiza automaticamente o schema do banco.
  - `spring.jpa.show-sql=true`: Exibe as queries geradas no console.
- 

## 4.4 Criando Repositórios

O Spring Data JPA permite a criação de repositórios com apenas uma interface:

```
public interface ClienteRepository extends JpaRepository<Cliente, Long> {
}
```



**Dica:** A interface `JpaRepository` oferece métodos como `save()`, `findById()`, `findAll()`, `deleteById()` e muito mais, prontos para uso.

---

## 4.5 Query Methods

O Spring Data JPA suporta a criação automática de queries baseadas em nomes de métodos:

```
List<Cliente> findByName(String nome);
List<Cliente> findByNameContaining(String fragmento);
List<Cliente> findByIdBetween(Long inicio, Long fim);
```

\Explicar uma regra dentro do código:

- `findByName`: **Gera** `WHERE nome = ?`
- `findByNameContaining`: **Gera** `WHERE nome LIKE %??%`
- `findByIdBetween`: **Gera** `WHERE id BETWEEN ? AND ?`



**Dica:** O Spring Data JPA interpreta o nome do método e converte para uma query JPQL.

---

## 4.6 Queries Personalizadas

Quando as query methods não são suficientes, é possível usar:

### ♦ JPQL

```
@Query("SELECT c FROM Cliente c WHERE c.nome = :nome")
List<Cliente> buscarPorNome(@Param("nome") String nome);
```

## ♦ SQL Nativo

```
@Query(value = "SELECT * FROM clientes WHERE nome = :nome",
nativeQuery = true)
List<Cliente> buscarPorNomeNativo(@Param("nome") String nome);
```

---


## 4.7 Paginação e Ordenação

O Spring Data JPA oferece suporte nativo à paginação e ordenação:

```
Page<Cliente> findByNome(String nome, Pageable pageable);
```

Exemplo de uso:

```
Pageable pageable = PageRequest.of(0, 10,
Sort.by("nome").ascending());
Page<Cliente> pagina = clienteRepository.findByNome("Ana", pageable);
```

 **Dica:** `PageRequest.of(page, size, sort)` permite paginação eficiente.

---

## 4.8 Auditoria

Com a anotação `@EnableJpaAuditing` na classe de configuração principal, é possível rastrear automaticamente quem criou/modificou as entidades.

```
@Entity
@EntityListeners(AuditingEntityListener.class)
public class Cliente {
    @CreatedDate
    private LocalDateTime criadoEm;


    @LastModifiedDate
    private LocalDateTime atualizadoEm;
}
```

---

## 4.9 Integração com Hibernate

Por trás dos panos, o Spring Data JPA usa o Hibernate como provedor padrão. Isso significa:

- ✓ Suporte a cache de primeiro e segundo nível.
- ✓ Estratégias de fetching e otimização de consultas.
- ✓ Mapeamentos avançados (herança, polimorfismo).

 **Dica:** Configure caches adicionais ou personalize o `JpaVendorAdapter` para ajustes finos.

---

## 4.10 Vantagens e Desvantagens

Vantagens	Desvantagens
Redução drástica de boilerplate code	Pode mascarar queries complexas
Integração completa com Spring Boot	Performance pode depender do tuning do ORM
Paginação, sorting e auditoria integrados	Curva de aprendizado em queries personalizadas

---

## 4.11 Conclusão

O Spring Data JPA é a forma mais prática e moderna de integrar persistência de dados com o ecossistema Spring. Ele reduz a complexidade e acelera o desenvolvimento de aplicações corporativas. No próximo capítulo, exploraremos as **Boas Práticas e Padrões Avançados**, incluindo transações, otimizações e padrões de design para aplicações robustas.


---



## Capítulo 5. Boas Práticas e Padrões Avançados em JPA/Hibernate

### 5.1 Importância das Boas Práticas

Ao trabalhar com JPA e Hibernate, é essencial adotar boas práticas para garantir **manutenibilidade, desempenho e escalabilidade** da aplicação. Ignorar essas recomendações pode gerar problemas como consultas ineficientes, vazamento de memória, deadlocks e inconsistências de dados.

 **Dica:** O uso correto de transações e caching é vital para aplicações corporativas de grande porte.

---

### 5.2 Gerenciamento de Sessões e EntityManager

Evite o uso de sessões ou EntityManagers “globais”. Sempre preferir o **escopo transacional** (geralmente por meio da anotação `@Transactional`).



```
@Transactional
public void salvarCliente(Cliente cliente) {
    clienteRepository.save(cliente);
}
```

\Explicar uma regra dentro do código:

- **@Transactional:** Garante que a operação seja executada em uma única transação, com commit ou rollback automáticos.
- 

## 5.3 Lazy vs. Eager Loading

### ◆ FetchType.LAZY

Utilize **LAZY** sempre que possível para evitar consultas desnecessárias e sobrecarga de memória.

```
@OneToMany(mappedBy = "cliente", fetch = FetchType.LAZY)
private List<Pedido> pedidos;
```

### ◆ FetchType.EAGER

Use apenas quando a associação for **essencial** para cada uso da entidade.

---

## 5.4 Evitando o N+1 Problem

O problema **N+1** ocorre quando o ORM faz uma consulta principal e depois várias queries para carregar as associações (um para cada item da lista).

💡 **Dica:** Use **JOIN FETCH** em HQL ou **EntityGraph** no Spring Data JPA.

### ◆ Exemplo com HQL:

```
String hql = "SELECT c FROM Cliente c JOIN FETCH c.pedidos";
List<Cliente> clientes = entityManager.createQuery(hql,
    Cliente.class).getResultList();
```

---

## 5.5 Padrão DTO (Data Transfer Object)

Evite expor entidades diretamente em APIs. Use DTOs para mapear apenas os dados necessários.

```
public class ClienteDTO {
    private Long id;
    private String nome;
    // getters e setters
}
```

💡 **Dica:** Use bibliotecas como **MapStruct** ou **ModelMapper** para conversão automática.

---

## 5.6 Uso Inteligente do Cache

Ative o cache de segundo nível apenas para entidades que raramente mudam ou são acessadas com frequência.

```
<property name="hibernate.cache.use_second_level_cache" value="true"/>
<property name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.ehcache.EhCacheRegionFactory"/>
```

---

## 5.7 Controle de Transações

Em ambientes Spring, delegue o controle de transações ao container com `@Transactional` e evite gerenciar transações manualmente.

```
@Transactional(readOnly = true)
public List<Cliente> listarTodos() {
    return clienteRepository.findAll();
}
```



**Dica:** Use `readOnly = true` para otimizar a performance em consultas de leitura.

---

## 5.8 Otimizando Consultas

- Prefira **JPQL/HQL** para queries complexas e portáteis.
- Evite `SELECT *` (use apenas campos necessários).
- Use **projeções** com JPQL ou Spring Data JPA para carregar apenas os campos necessários.

### ♦ Projeções em Spring Data JPA:

```
interface ClienteResumo {
    Long getId();
    String getNome();
}

List<ClienteResumo> findByNome(String nome);
```

---

## 5.9 Uso de Índices no Banco

Garanta que colunas frequentemente consultadas sejam indexadas no banco de dados. Isso reduz drasticamente o tempo de execução de queries.



**Dica:** Combine o uso de índices com o monitoramento de queries usando logs do Hibernate (`spring.jpa.show-sql=true`).

---

## 5.10 Padrões Avançados

- **Padrão Repository:** Organiza a persistência de forma clara e testável.

- **Padrão Service Layer:** Isola regras de negócio das operações de persistência.
  - **Specification Pattern:** Permite construir queries dinâmicas e reutilizáveis.
- 

## 5.11 Conclusão

Neste capítulo, aprendemos como usar **boas práticas e padrões avançados** para tornar sua aplicação JPA/Hibernate mais robusta, escalável e manutenível. Estas estratégias são essenciais para aplicações corporativas modernas, onde o desempenho e a clareza de código são fundamentais.


No próximo capítulo, exploraremos a **Integração de ORM com Microserviços**, abordando como conectar JPA/Hibernate a arquiteturas distribuídas usando Spring Cloud e outras tecnologias modernas.

---

# Capítulo 6. Integração de ORM com Microserviços

## 6.1 Introdução

Em arquiteturas modernas de microserviços, a integração entre persistência de dados e serviços distribuídos é um desafio crítico. Este capítulo explora como combinar JPA/Hibernate com microserviços, focando em **desempenho, consistência de dados e escalabilidade**.

 **Dica:** A persistência deve ser isolada por serviço para evitar acoplamentos e gargalos.

---

## 6.2 Desafios Comuns

- **Isolamento de dados:** Cada microserviço deve ter seu próprio banco de dados ou esquema.
  - **Transações distribuídas:** Evitar transações entre múltiplos bancos (XA Transactions).
  - **Sincronização de dados:** Replicação e consistência eventual em sistemas distribuídos.
  - **Escalabilidade:** ORM pode introduzir gargalos se mal configurado.
-

## 6.3 Estratégias de Persistência em Microserviços

### ♦ Banco de Dados por Serviço (Database per Service)

Cada microserviço é responsável pela sua própria persistência.

#### ✓ Vantagens:

- Isolamento de dados e maior independência.
- Melhor escalabilidade.

#### ✗ Desvantagens:

- Dados duplicados entre serviços.
- Necessidade de estratégias de sincronização (event sourcing, CDC).

💡 **Dica:** Use `schema-per-service` para bancos compartilhados, como Oracle ou PostgreSQL.

---

### ♦ Shared Database (Banco Compartilhado)

Evitar ao máximo, pois quebra o isolamento e aumenta o acoplamento. Só use em cenários legados onde refatorar não seja viável.

---

## 6.4 Integração com JPA/Hibernate

Mesmo em microserviços, é comum usar JPA/Hibernate para manipular o banco local de cada serviço.

### ♦ Configuração Isolada

Cada microserviço deve ter seu próprio `DataSource`, `EntityManagerFactory` e `TransactionManager`.

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages = "com.exemplo.pedido.repository")
public class PedidoJpaConfig {
    @Bean
    public LocalContainerEntityManagerFactoryBean
    entityManagerFactory() {
        // configurações específicas para o serviço
    }
}
```

💡 **Dica:** Use `basePackages` para isolar os repositórios de cada microserviço.

---

## 6.5 Comunicação entre Microserviços

JPA é local; comunicação entre microserviços ocorre via:

- **REST APIs** (Spring Web)

- **Mensageria** (RabbitMQ, Kafka)
- **Eventos** (event sourcing, CDC)

💡 **Dica:** Nunca compartilhe a mesma `Entity` entre serviços diferentes; use DTOs e transformação de dados.

---

## 6.6 Consistência de Dados

### ♦ Transações Distribuídas (XA)

Evite transações distribuídas sempre que possível. São complexas e impactam o desempenho. Prefira:

- **Consistência Eventual:** Confiança de que os dados ficarão consistentes com o tempo.
- **Outbox Pattern:** Grava eventos em uma tabela local e depois publica em fila.

### ♦ Exemplo de Outbox Pattern

- 1 Transação salva a entidade e o evento na tabela outbox.
  - 2 Um worker lê a outbox e publica o evento no Kafka/RabbitMQ.
  - 3 Outro serviço consome o evento e sincroniza os dados.
- 

## 6.7 Padrão CQRS (Command Query Responsibility Segregation)

Dividir a escrita (Command) e a leitura (Query) em serviços separados.

- **Command Service:** Usa JPA para gravação.
- **Query Service:** Pode usar bancos otimizados para leitura ou Elasticsearch.

💡 **Dica:** Facilita a escalabilidade horizontal e melhora a performance em consultas complexas.

---

## 6.8 Cache Distribuído

Evite cache de segundo nível do Hibernate em microserviços, pois não é distribuído. Prefira caches como:

- **Redis**
- **Hazelcast**
- **Apache Ignite**

Configure o cache para armazenar dados compartilhados e reduzir consultas desnecessárias.

---

## 6.9 Monitoramento e Observabilidade

- **Logs SQL:** Ative logs no Hibernate (`show_sql`) para analisar queries.
  - **Tracing Distribuído:** Use Spring Cloud Sleuth ou OpenTelemetry para rastrear transações entre serviços.
  - **Métricas:** Integre com Prometheus ou Micrometer para monitorar o uso de JPA e conexões.
- 

## 6.10 Boas Práticas

- ✓ Cada microserviço deve ser dono do seu próprio modelo de dados.
  - ✓ Evite usar DTOs como entidades JPA.
  - ✓ Use pagination para evitar `OutOfMemory` em consultas.
  - ✓ Prefira JPQL ou Criteria API em consultas complexas.
  - ✓ Externalize a configuração do banco para evitar acoplamentos.
- 

## 6.11 Conclusão

Integrar JPA/Hibernate com microserviços exige disciplina arquitetural e atenção às melhores práticas. Neste capítulo, abordamos o isolamento de dados, a comunicação entre serviços, padrões como Outbox e CQRS, e dicas avançadas para manter consistência e desempenho.

No próximo capítulo, vamos explorar **Testes em Ambientes Distribuídos**, abordando como testar serviços com JPA/Hibernate em cenários de microserviços e bancos de dados isolados.

---

# Capítulo 7. Testes em Ambientes Distribuídos

## 7.1 Introdução

Testar microserviços que utilizam JPA/Hibernate apresenta desafios significativos, como isolamento de banco de dados, consistência transacional e comunicação entre

serviços. Neste capítulo, vamos explorar **estratégias práticas e ferramentas recomendadas** para garantir qualidade, confiabilidade e desempenho em ambientes distribuídos.

💡 **Dica:** Combine testes unitários, de integração e end-to-end para obter uma cobertura robusta.

---

## 7.2 Pirâmide de Testes em Microserviços

- ♦ **Testes Unitários (base):** Validam a lógica de negócios isoladamente, sem dependências externas.
  - ♦ **Testes de Integração (meio):** Validam a interação com o banco de dados, JPA/Hibernate e a camada de persistência.
  - ♦ **Testes End-to-End (topo):** Validam o fluxo completo de dados entre microserviços e dependências reais ou simuladas.
- 

## 7.3 Testes Unitários com JPA

Evite depender do banco real nos testes unitários. Use **mocks** (Mockito) para isolar o comportamento:

```
@ExtendWith(MockitoExtension.class)
public class ClienteServiceTest {

    @Mock
    private ClienteRepository clienteRepository;

    @InjectMocks
    private ClienteService clienteService;

    @Test
    void deveSalvarClienteComSucesso() {
        Cliente cliente = new Cliente();
        cliente.setNome("Ana");

        when(clienteRepository.save(any(Cliente.class))).thenReturn(cliente);

        Cliente salvo = clienteService.salvar(cliente);

        assertEquals("Ana", salvo.getNome());
    }
}
```

💡 **Dica:** Use `@Mock` para injetar repositórios e evitar a dependência do banco de dados.

---

## 7.4 Testes de Integração com Banco de Dados

### ♦ Testcontainers

Para simular o ambiente de produção, utilize **Testcontainers** para criar instâncias reais de banco de dados em Docker:

```
@Testcontainers
@SpringBootTest
public class ClienteRepositoryIT {

    @Container
    static MySQLContainer<?> mysql = new
MySQLContainer<>("mysql:8.0");

    @DynamicPropertySource
    static void configureProperties(DynamicPropertyRegistry registry)
    {
        registry.add("spring.datasource.url", mysql::getJdbcUrl);
        registry.add("spring.datasource.username",
mysql::getUsername);
        registry.add("spring.datasource.password",
mysql::getPassword);
    }

    @Autowired
    private ClienteRepository clienteRepository;

    @Test
    void deveSalvarCliente() {
        Cliente cliente = new Cliente();
        cliente.setNome("Ana");
        Cliente salvo = clienteRepository.save(cliente);

        assertNotNull(salvo.getId());
    }
}
```

\Explicar uma regra dentro do código:

- **@Container:** Cria um container Docker gerenciado pelo JUnit.
  - **@DynamicPropertySource:** Configura as propriedades dinamicamente para o Spring Boot.
- 

### ♦ Banco de Dados em Memória

Use H2 ou HSQLDB apenas para cenários simples. Não cobre todas as funcionalidades reais (ex.: tipos de dados específicos).

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

---



## 7.5 Testes End-to-End

Utilize **Testcontainers** para bancos e mensageria (RabbitMQ/Kafka), simulando o ambiente real. Combine com:

- **WireMock**: Para mockar APIs externas.
- **Spring Cloud Contract**: Para gerar contratos de APIs.
- **RestAssured**: Para chamadas HTTP.

💡 **Dica**: Priorize pipelines CI/CD para executar testes automáticos.

---

## 7.6 Estratégias para Testar Comunicação entre Microserviços

- **Consumer-Driven Contracts**: Defina contratos para APIs e mensageria.
  - **Event-Driven Testing**: Teste publicação e consumo de eventos com Kafka/RabbitMQ.
  - **Simulação de Falhas**: Use bibliotecas como **Chaos Monkey** para validar resiliência.
- 

## 7.7 Limpeza e Isolamento de Dados

Sempre limpe os dados após cada teste para garantir independência:

```
@AfterEach
void limparBanco() {
    clienteRepository.deleteAll();
}
```

💡 **Dica**: Use transações e **rollback automático** no Spring Boot com `@Transactional` em testes.

---

## 7.8 Monitoramento de Consultas

Ative logs SQL para rastrear queries e identificar problemas de desempenho:

```
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

💡 **Dica**: Analise os logs para identificar N+1 e queries mal otimizadas.

---

## 7.9 Boas Práticas

- ✓ Combine Testcontainers com JUnit 5 para ambientes realistas.
- ✓ Use mocks apenas quando não testar a camada de persistência.

- ✓ Priorize testes automatizados no pipeline CI/CD.
  - ✓ Evite dependências externas que não estejam sob seu controle.
- 

## 7.10 Conclusão

Testar microserviços com JPA/Hibernate exige disciplina e boas práticas para garantir confiabilidade e escalabilidade. Neste capítulo, apresentamos estratégias robustas para testes unitários, de integração e end-to-end, utilizando ferramentas modernas como Testcontainers e Spring Boot.

No próximo capítulo, exploraremos **Estratégias de Versionamento e Migração de Banco de Dados**, incluindo o uso de Flyway e Liquibase em ambientes distribuídos.

---

# Capítulo 8. Estratégias de Versionamento e Migração de Banco de Dados

## 8.1 Introdução

Em aplicações corporativas modernas, gerenciar alterações de esquema de banco de dados é tão essencial quanto controlar versões de código-fonte. Este capítulo explora **estratégias, ferramentas e boas práticas** para versionar e migrar bancos de dados em microserviços Java, com foco na integração com JPA/Hibernate.

💡 **Dica:** Adotar versionamento de banco é indispensável para evitar conflitos entre desenvolvedores e garantir consistência entre ambientes (desenvolvimento, homologação, produção).

---

## 8.2 Problemas Comuns Sem Versionamento

- Scripts manuais e desorganizados.
  - Falta de controle de versões em ambientes diferentes.
  - Perda de histórico de alterações.
  - Falhas ao sincronizar migrações em ambientes distribuídos.
-

## 8.3 Abordagens para Versionamento

### ♦ Versionamento Manual (não recomendado)

Scripts SQL salvos em diretórios (ex.: `v1__criar_tabela.sql`), sem ferramenta de controle.

✗ Difícil de automatizar e manter.

---

### ♦ Versionamento Automatizado (recomendado)

Ferramentas de versionamento automatizam a execução de scripts, validam o histórico de execuções e permitem rollback controlado.

---

## 8.4 Ferramentas Populares

### ♦ Flyway

- Scripts em SQL ou Java (via callbacks).
- Integração fácil com Spring Boot.
- Estrutura de versionamento simples (`v1__init.sql`, `v2__add_column.sql`).

💡 **Dica:** Flyway segue um padrão linear de versionamento, ideal para equipes pequenas/médias.

---

### ♦ Liquibase

- Permite scripts XML, YAML, JSON ou SQL.
- Suporta rollback automático e múltiplos tipos de bancos.
- Mais flexível para cenários complexos.

💡 **Dica:** Ideal para bancos de dados corporativos com grande variedade de tipos de dados e alterações complexas.

---

## 8.5 Integração com Spring Boot

### ♦ Flyway

Adicione a dependência no `pom.xml`:

```
<dependency>
  <groupId>org.flywaydb</groupId>
  <artifactId>flyway-core</artifactId>
</dependency>
```

Configure no `application.properties`:

```
spring.flyway.locations=classpath:db/migration
spring.flyway.baseline-on-migrate=true
```

💡 **Dica:** `baseline-on-migrate` é útil para bases existentes, permitindo inicializar o versionamento sem falhar.

---

## ♦ Liquibase

Adicione a dependência:

```
<dependency>
  <groupId>org.liquibase</groupId>
  <artifactId>liquibase-core</artifactId>
</dependency>
```

Configure no `application.properties`:

```
spring.liquibase.change-log=classpath:db/changelog/db.changelog-
master.xml
```

💡 **Dica:** Use `db.changelog-master.xml` para gerenciar múltiplos scripts e incluir sub-changelogs.

---

## 8.6 Boas Práticas

- ✓ **Scripts Idempotentes:** Evite falhas em execuções repetidas.
  - ✓ **Scripts Pequenos:** Quebra scripts grandes em partes menores para facilitar revisão e rollback.
  - ✓ **Controle de Versões:** Armazene scripts no controle de versão (ex.: Git).
  - ✓ **Pipeline CI/CD:** Integre Flyway/Liquibase no pipeline para garantir execução automática.
  - ✓ **Ambientes Isolados:** Teste scripts em bases de dados locais antes de aplicar em produção.
  - ✓ **Backup Automático:** Faça backup antes de rodar migrações em produção.
- 

## 8.7 Integração com Hibernate

💡 **Dica:** Evite `hibernate.hbm2ddl.auto` em produção (`update`, `create`, `validate`). Use `none` e deixe Flyway/Liquibase gerenciar o schema.

```
spring.jpa.hibernate.ddl-auto=none
```

---

## 8.8 Estratégias Avançadas

- **Branch-Based Migration:** Para equipes grandes, utilize branches de migração sincronizadas com feature branches.
- **Semantic Versioning:** Adote padrões como `v1_0_0__init.sql` para indicar versões semânticas.

- **Changelogs Automatizados:** Gere changelogs automáticos a partir de modelos ER.
- 

## 8.9 Monitoramento de Versões

- **Flyway:** `flyway_schema_history` controla scripts aplicados.
  - **Liquibase:** `DATABASECHANGELOG` mantém histórico e rollback.
- 

## 8.10 Conclusão

Versionar e migrar bancos de dados com ferramentas como Flyway e Liquibase é essencial para a robustez de aplicações corporativas Java. Este capítulo apresentou boas práticas e padrões de integração com JPA/Hibernate, garantindo segurança e previsibilidade no processo de evolução do banco de dados.


No próximo capítulo, abordaremos **Padrões Avançados de Modelagem de Entidades**, explorando herança, mapeamentos complexos e otimizações de performance com Hibernate.

---

# Capítulo 9. Padrões Avançados de Modelagem de Entidades

## 9.1 Introdução

Modelar entidades de forma eficiente é um diferencial decisivo em aplicações Java corporativas. Neste capítulo, exploraremos padrões avançados de modelagem de entidades com **JPA e Hibernate**, abordando herança, mapeamentos complexos, otimizações de performance e boas práticas arquiteturais.

 **Dica:** Pense no modelo de dados como um contrato que evolui com o domínio — não apenas como tabelas isoladas.

---

## 9.2 Herança no Modelo Relacional

JPA oferece três estratégias de herança para mapear hierarquias de classes para tabelas:

### ♦ Single Table (TABLE\_PER\_CLASS)

Uma única tabela para todas as classes da hierarquia.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "tipo")
public abstract class Pagamento {
    @Id
    private Long id;
    private BigDecimal valor;
}
```

\Explicar uma regra dentro do código:

- @DiscriminatorColumn: Define a coluna que identifica o tipo de entidade.



**Vantagem:** Consultas rápidas e simples.



**Desvantagem:** Muitos campos nulos em subclasses.

---

### ♦ Joined

Cria uma tabela para cada classe da hierarquia.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Pagamento {
    @Id
    private Long id;
}
```



**Vantagem:** Normalização.



**Desvantagem:** JOINS complexos e performance inferior.

---

### ♦ Table per Class

Cada classe tem sua própria tabela e não compartilha tabelas.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Pagamento {
    @Id
    private Long id;
}
```



**Vantagem:** Independência entre classes.



**Desvantagem:** Consultas unificadas são complexas e lentas.

---

## 9.3 Mapeamentos Avançados

### ♦ Embedded Objects

Permite agrupar atributos em classes reutilizáveis.

```

@Embeddable
public class Endereco {
    private String rua;
    private String cidade;
}

@Entity
public class Cliente {
    @Embedded
    private Endereco endereco;
}

```

✓ **Vantagem:** Reuso e melhor organização.

✗ **Desvantagem:** Sem identidade própria.

---

### ◆ ElementCollection

Coleções de tipos básicos ou embutidos.

```

@Entity
public class Cliente {
    @ElementCollection
    private List<String> telefones;
}

```

💡 **Dica:** Ideal para listas simples; evite para relacionamentos complexos.

---

### ◆ Mapeamento Bidirecional

Cuidado com a bidirecionalidade:

```

@Entity
public class Pedido {
    @OneToMany(mappedBy = "pedido")
    private List<ItemPedido> itens;
}

@Entity
public class ItemPedido {
    @ManyToOne
    @JoinColumn(name = "pedido_id")
    private Pedido pedido;
}

```

💡 **Dica:** Mantenha o controle do lado "donor" (mappedBy) para evitar inconsistências.

---

## 9.4 Otimizações de Performance

- **FetchType.LAZY:** Carregamento sob demanda para evitar N+1.
- **Batch Fetching:** Configure `hibernate.default_batch_fetch_size` para otimizar coleções.
- **Second-Level Cache:** Use Redis ou Infinispan em aplicações monolíticas, mas evite em microserviços.

- **DTO Projections:** Use JPQL ou Spring Data Projections para reduzir carga.
- 

## 9.5 Soft Deletes

Em vez de deletar registros fisicamente, use flags lógicas:

```
@Entity
@SQLDelete(sql = "UPDATE cliente SET ativo = false WHERE id = ?")
@Where(clause = "ativo = true")
public class Cliente {
    @Id
    private Long id;
    private Boolean ativo = true;
}
```



**Dica:** Combina bem com auditoria e histórico de dados.

---

## 9.6 Auditoria e Histórico

Use Hibernate Envers para versionamento automático:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-envers</artifactId>
</dependency>
@Entity
@Audited
public class Cliente {
    @Id
    private Long id;
    private String nome;
}
```

---

## 9.7 Boas Práticas

- ✓ Use herança apenas quando fizer sentido no domínio.
  - ✓ Prefira `LAZY` e projeções para melhorar performance.
  - ✓ Isolar mapeamentos complexos usando DTOs ajuda a evitar acoplamentos.
  - ✓ Use `@Embeddable` para agrupamentos sem identidade própria.
  - ✓ Gerencie soft deletes e auditoria com frameworks dedicados.
- 

## 9.8 Conclusão

Neste capítulo, você aprendeu como usar estratégias avançadas de modelagem de entidades em JPA/Hibernate, incluindo herança, mapeamentos complexos e otimizações de performance. Estes padrões são fundamentais para criar modelos de dados robustos, escaláveis e fáceis de manter em ambientes corporativos.




No próximo capítulo, vamos abordar **Gerenciamento de Conexões e Pooling**, essencial para garantir a performance de aplicações Java de alta demanda.

---

## Capítulo 10. Gerenciamento de Conexões e Pooling

### 10.1 Introdução

Gerenciar conexões de banco de dados é uma das tarefas mais importantes para garantir a escalabilidade e a performance de aplicações corporativas. Este capítulo explora como configurar pools de conexão, evitar vazamentos e otimizar a performance em aplicações Java, especialmente com **JPA/Hibernate e Spring Boot**.




 **Dica:** Em ambientes de alta demanda, conexões mal gerenciadas são gargalos críticos. Um pool bem dimensionado é o segredo do sucesso.

---

### 10.2 O Problema das Conexões

Conexões são recursos limitados. Cada conexão é cara de abrir e manter, e vazamentos podem levar a falhas graves (ex.: banco de dados indisponível).

Problemas comuns:

-  Conexões abertas e não fechadas (vazamentos).
  -  Threads bloqueadas aguardando conexão.
  -  Pool mal dimensionado causando lentidão ou estouro.
- 

### 10.3 O Conceito de Connection Pool

Um **pool de conexões** é um conjunto pré-alocado de conexões abertas, gerenciadas por um framework. Ao invés de abrir/fechar a cada transação, a aplicação pega uma conexão do pool e a devolve ao final.

 **Dica:** Isso reduz o custo de overhead e melhora a performance.

---

### 10.4 Frameworks de Pooling Populares

- ♦ HikariCP
  - Padrão no Spring Boot 2.x+.

- Leve, rápido e eficiente.
- Configuração simples.

#### ♦ Apache DBCP

- Amplamente usado no passado.
- Mais pesado e menos performático.

#### ♦ C3P0

- Suporte a funcionalidades avançadas.
- Mais lento e difícil de tunar.

💡 **Dica: HikariCP** é o mais recomendado para a maioria dos casos corporativos.

---

## 10.5 Configuração do HikariCP no Spring Boot



`application.properties`

```
spring.datasource.url=jdbc:mysql://localhost:3306/meubanco
spring.datasource.username=usuario
spring.datasource.password=senha
spring.datasource.hikari.maximum-pool-size=10
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.idle-timeout=60000
spring.datasource.hikari.connection-timeout=30000
spring.datasource.hikari.max-lifetime=1800000
```

\Explicar uma regra dentro do código:

- `maximum-pool-size`: Define o número máximo de conexões simultâneas.
  - `idle-timeout`: Tempo máximo que uma conexão pode ficar ociosa antes de ser fechada.
  - `max-lifetime`: Limite máximo de tempo de vida de uma conexão.
- 

## 10.6 Monitoramento do Pool

HikariCP expõe métricas de performance integradas via **Micrometer** e **Actuator**.



**Dica:** Adicione a dependência do Actuator:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Ative o endpoint de métricas:

```
management.endpoints.web.exposure.include=health,info,metrics
```

Exemplo de métrica:

```
hikaricp.connections.active
hikaricp.connections.idle
```


---

## 10.7 Prevenção de Vazamentos

- Sempre feche as conexões!
  - Use **try-with-resources** em JDBC puro:

```
try (Connection conn = dataSource.getConnection()) {  
    // use a conexão  
}
```
  - Em JPA, a transação gerencia o ciclo de vida, mas revise configurações de timeout e rollback.
- 

## 10.8 Dimensionamento e Performance

 Ajuste o tamanho do pool de acordo com a carga da aplicação e a capacidade do banco de dados.

 **Dica:** Comece com o número de núcleos da máquina + 1 como base de cálculo.

---

## 10.9 Integração com Hibernate

Configure o DataSource do pool como fonte para o Hibernate:

```
spring.jpa.properties.hibernate.connection.provider_class=org.hibernate.  
e.hikaricp.internal.HikariCPConnectionProvider
```

---

## 10.10 Boas Práticas

- ✓ Sempre feche conexões ou transações.
  - ✓ Monitore uso e tempo de resposta.
  - ✓ Configure limites de tempo (`idle-timeout`, `connection-timeout`).
  - ✓ Evite usar múltiplos pools desnecessariamente.
  - ✓ Use pools dedicados em clusters (ex.: Kubernetes) para isolar cargas.
- 

## 10.11 Conclusão

Gerenciar conexões de forma eficiente é um pilar fundamental da performance de aplicações Java empresariais. Com o uso adequado de pools como o **HikariCP**, aliado ao monitoramento e às melhores práticas de dimensionamento, você garante estabilidade e performance mesmo em ambientes de alta concorrência.


No próximo capítulo, exploraremos **Cache de Segunda Nível com Hibernate**, aprofundando em como melhorar o desempenho e a escalabilidade das consultas em JPA.

---

# Capítulo 11. Cache de Segundo Nível com Hibernate

## 11.1 Introdução

O cache de segundo nível é um mecanismo avançado de otimização de performance em aplicações Java que utilizam Hibernate como provedor de persistência. Ele reduz o número de acessos diretos ao banco de dados ao armazenar entidades e coleções em memória ou em sistemas de cache distribuído.

 **Dica:** Use o cache de segunda nível para dados lidos com frequência e com baixo nível de atualização.

---

## 11.2 Conceitos Fundamentais

### ♦ Cache de Primeiro Nível


- Gerenciado automaticamente pelo EntityManager.
- Escopo: transação atual (session).
- Descartado ao final da transação.

### ♦ Cache de Segundo Nível

- Compartilhado entre múltiplas sessões e transações.
  - Pode ser persistido em memória ou em caches distribuídos (ex.: Redis, Ehcache, Infinispan).
  - Permite otimizar buscas repetidas.
- 

## 11.3 Estrutura do Cache de Segunda Nível

- **Entidades:** Cache para objetos mapeados como `@Entity`.
- **Coleções:** Cache para listas ou mapas relacionados.
- **Queries:** Cache de consultas HQL/JPQL com resultados pré-calculados (Query Cache).

 **Dica:** Cache de queries requer ativação adicional e é mais complexo de gerenciar.

---

## 11.4 Ativando o Cache no Hibernate

No `persistence.xml` ou `application.properties`:

```
spring.jpa.properties.hibernate.cache.use_second_level_cache=true
```

```
spring.jpa.properties.hibernate.cache.region.factory_class=org.hibernate.cache.jcache.JCacheRegionFactory
spring.jpa.properties.javax.cache.provider=org.ehcache.jsr107.EhcacheCachingProvider
```

\Explicar uma regra dentro do código:

- `hibernate.cache.region.factory_class`: Define o provedor de cache de segunda nível a ser usado.
  - `javax.cache.provider`: Define a implementação JCache (JSR-107) escolhida (ex.: Ehcache).
- 

## 11.5 Configuração com Ehcache

1 Adicione as dependências:

```
<dependency>
  <groupId>org.ehcache</groupId>
  <artifactId>ehcache</artifactId>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-jcache</artifactId>
</dependency>
```

2 Crie o arquivo ehcache.xml:

```
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.ehcache.org/v3">
  <cache alias="default">
    <expiry>
      <ttl unit="minutes">30</ttl>
    </expiry>
    <resources>
      <heap unit="entries">1000</heap>
    </resources>
  </cache>
</config>
```

3 Configure o cache provider:

```
spring.jpa.properties.javax.cache.uri=classpath:ehcache.xml
```

---

## 11.6 Anotando Entidades para Cache

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage =
CacheConcurrencyStrategy.READ_WRITE)
public class Cliente {
  @Id
  private Long id;
  private String nome;
}
```



**Dica:** READ\_WRITE é o modo mais seguro, garantindo consistência.

---

## 11.7 Cache de Consultas

Ative o cache de queries:

```
spring.jpa.properties.hibernate.cache.use_query_cache=true
```

Use a anotação `@QueryHints`:

```
@QueryHints(@QueryHint(name = "org.hibernate.cacheable", value = "true"))
```

---

## 11.8 Boas Práticas

- ✓ Use o cache apenas para entidades com leitura frequente.
  - ✓ Escolha a estratégia (`READ_ONLY`, `READ_WRITE`, `NONSTRICT_READ_WRITE`) de acordo com a frequência de atualização.
  - ✓ Monitore estatísticas de cache (via Hibernate Metrics ou Micrometer).
  - ✓ Combine com cache distribuído para alta disponibilidade (ex.: Redis).
  - ✓ Evite usar cache em dados críticos de alta transação para evitar inconsistências.
- 

## 11.9 Monitoramento

Use o Hibernate Statistics:

```
SessionFactory sessionFactory =  
entityManagerFactory.unwrap(SessionFactory.class);  
Statistics stats = sessionFactory.getStatistics();  
System.out.println("Hit Count: " +  
stats.getSecondLevelCacheHitCount());
```



**Dica:** Integre com Spring Boot Actuator para métricas automáticas.

---

## 11.10 Conclusão

O cache de segunda nível do Hibernate é uma ferramenta poderosa para reduzir a carga no banco de dados e melhorar o desempenho em leituras frequentes. Entretanto, é essencial configurar corretamente e monitorar de perto para evitar problemas de inconsistência e uso indevido.


No próximo capítulo, vamos explorar **Implementação de Auditoria e Versionamento com Hibernate Envers**, aprimorando a rastreabilidade e segurança das alterações nos dados.

---

## Capítulo 12. Implementação de Auditoria e Versionamento com Hibernate Envers

### 12.1 Introdução

Rastrear mudanças em entidades de banco de dados é essencial para garantir a segurança, a conformidade e a manutenção de um histórico de alterações. **Hibernate Envers** é uma extensão do Hibernate que fornece auditoria e versionamento de forma transparente e fácil de integrar.




 **Dica:** Auditoria é particularmente útil para sistemas financeiros, governamentais e jurídicos, onde rastrear alterações é obrigatório.

---

### 12.2 Conceito de Auditoria

A auditoria visa manter um registro de todas as alterações realizadas em entidades persistentes, incluindo operações de **inserção**, **atualização** e **deleção**.

Benefícios:

-  Histórico completo de alterações.
  -  Possibilidade de recuperar versões antigas.
  -  Transparência para auditorias externas.
- 

### 12.3 Como o Envers Funciona

Envers cria tabelas de histórico (por padrão com sufixo `_AUD`) para cada entidade auditada, registrando versões sempre que ocorre uma operação persistente.

Exemplo:

- Tabela: `cliente`
  - Tabela de auditoria: `cliente_AUD`
- 

### 12.4 Configuração do Envers

① Adicione as dependências no `pom.xml`:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-envers</artifactId>
</dependency>
```

② Ative o Envers nas configurações do Hibernate:

```
spring.jpa.properties.hibernate.integration.envers.enabled=true
```

---

## 12.5 Anotando Entidades para Auditoria

```
@Entity
@Audited
public class Cliente {
    @Id
    private Long id;
    private String nome;
}
```

\Explicar uma regra dentro do código:

- **@Audited:** Habilita o rastreamento de alterações na entidade.
- 

## 12.6 Recuperando Histórico

Use a API do Envers:

```
@Autowired
private EntityManager entityManager;

public List<Cliente> listarRevisoes(Long idCliente) {
    AuditReader auditReader = AuditReaderFactory.get(entityManager);
    List<Number> revisoes = auditReader.getRevisions(Cliente.class,
idCliente);

    List<Cliente> historico = new ArrayList<>();
    for (Number rev : revisoes) {
        Cliente cliente = auditReader.find(Cliente.class, idCliente,
rev);
        historico.add(cliente);
    }
    return historico;
}
```



**Dica:** AuditReader fornece acesso a todas as revisões de uma entidade.

---

## 12.7 Customizando a Tabela de Auditoria

Você pode renomear ou customizar a tabela de auditoria:

```
@Entity
@Audited
@AuditTable(value = "historico_cliente")
public class Cliente {
    @Id
    private Long id;
}
```

---



## 12.8 Trabalhando com Auditoria Avançada

### ♦ Campos Excluídos da Auditoria

```
@Transient
@Audited(targetAuditMode = RelationTargetAuditMode.NOT_AUDITED)
private String campoIgnorado;
```

### ♦ Auditoria de Relacionamentos

```
@OneToMany
@Audited
private List<Endereco> enderecos;
```

💡 **Dica:** Por padrão, coleções são auditadas como uma entidade separada com controle de alterações.

---

## 12.9 Estratégias de Retenção

💡 **Dica:** Implemente políticas de retenção de logs para evitar crescimento descontrolado das tabelas de auditoria. Ex.: triggers para exclusão de dados antigos ou partições na tabela `_AUD`.

---

## 12.10 Boas Práticas

- ✓ Audite apenas as entidades críticas para performance e segurança.
  - ✓ Monitore o crescimento das tabelas `_AUD` para evitar estouro de disco.
  - ✓ Combine com Soft Delete (Capítulo 9) para manter histórico de exclusões.
  - ✓ Crie índices apropriados para consultas de histórico.
  - ✓ Use DTOs para apresentar dados de auditoria ao usuário final.
- 

## 12.11 Conclusão


Com Hibernate Envers, implementa-se auditoria de forma elegante, sem reescrever regras de negócio ou poluir o código. O uso criterioso desse recurso garante rastreabilidade, segurança e conformidade regulatória em aplicações Java empresariais. No próximo capítulo, exploraremos **Boas Práticas de Arquitetura e Organização de Projetos Java**, preparando o terreno para manter a aplicação sustentável e escalável.

---

# Capítulo 13. Boas Práticas de Arquitetura e Organização de Projetos Java

## 13.1 Introdução

A organização do código e a arquitetura são fundamentais para o sucesso de uma aplicação Java corporativa. Um projeto bem estruturado facilita manutenção, testes, escalabilidade e o trabalho em equipe.

 **Dica:** Arquitetura não é apenas diagramas: é disciplina e estratégia para organizar responsabilidades.

---

## 13.2 Organização de Pacotes

Divida o projeto em pacotes claros, alinhados com o conceito de **Separation of Concerns**:

```
com.minhaempresa.minhaplicacao
```

— config	-> Configurações gerais (ex.: segurança, cache, JPA)
— controller	-> Camada de entrada (REST, MVC)
— service	-> Regras de negócio
— repository	-> Interface com a base de dados (JPA)
— model	-> Entidades do domínio
— dto	-> Objetos de transferência de dados
— exception	-> Tratamento de exceções
— util	-> Utilitários e helpers

 **Dica:** Esse padrão é conhecido como **Domain-Driven Package Structure** e melhora a legibilidade.

---

## 13.3 Camadas de Arquitetura

### ♦ Apresentação (Controller)

- Recebe requisições HTTP.
- Conversão de DTOs para modelos.

### ♦ Negócio (Service)

- Contém regras de negócio.
- Orquestra transações.

### ♦ Persistência (Repository)

- Gerencia acesso ao banco de dados.
- Usa JPA/Hibernate ou outro ORM.

### ♦ Modelo (Domain)

- Entidades, enumerações e regras de domínio.

💡 **Dica:** Use **DTOs** para evitar vazamentos do modelo de domínio na camada de apresentação.

---

## 13.4 Boas Práticas no Spring Boot

- ✓ Utilize `@Service` e `@Repository` para marcar classes de serviço e repositório.
  - ✓ Implemente exceções customizadas para erros específicos.
  - ✓ Configure **Bean Validation** (`@Valid`) para validar entradas.
  - ✓ Ative logs com SLF4J e Logback para rastreabilidade.
  - ✓ Mantenha as configurações em `application.yml` ou `application.properties` centralizadas.
- 

## 13.5 Padrões de Projeto

- **Singleton:** Para gerenciar recursos globais (ex.: conexão).
- **Factory:** Para criação de objetos complexos.
- **Builder:** Para construir objetos com múltiplos campos opcionais.
- **Observer:** Para eventos ou notificações.

💡 **Dica:** O Spring Boot já oferece suporte a muitos padrões por meio de injeção de dependência.

---

## 13.6 Gerenciamento de Dependências

Use **Maven** ou **Gradle**.

💡 **Dica:** Centralize as versões de dependências com **Dependency Management** para evitar conflitos.

---

## 13.7 Testes

- ✓ Use **JUnit 5** para testes unitários.
  - ✓ Use **Mockito** para mocks.
  - ✓ Configure perfis de teste (`application-test.properties`).
  - ✓ Integre com **Testcontainers** para testes de integração com bancos reais.
-

## 13.8 Segurança

Implemente autenticação e autorização com **Spring Security**.  
Use `@PreAuthorize` e `@Secured` para proteger métodos de serviço.

---

## 13.9 Documentação

- ✓ Integre o **Swagger/OpenAPI** para documentação de APIs RESTful.
  - ✓ Utilize comentários JavaDoc para descrever classes e métodos.
- 

## 13.10 Monitoramento

- ✓ Use Spring Boot Actuator para monitorar saúde, métricas e logs.
  - ✓ Configure logs estruturados para rastreamento.
- 

## 13.11 Conclusão

Uma boa arquitetura é a base para uma aplicação Java sustentável e robusta. Organizar pacotes, seguir boas práticas de camadas, utilizar padrões de projeto e investir em testes e monitoramento são passos essenciais para o sucesso em ambientes corporativos.

No próximo capítulo, poderemos explorar **Integração Contínua e Entrega Contínua (CI/CD) em Java**, para automatizar a qualidade e a entrega de software.

---



# Capítulo 14. Integração Contínua e Entrega Contínua (CI/CD) em Java

## 14.1 Introdução

Em um mundo onde a agilidade e a qualidade de software são cruciais, a adoção de **Integração Contínua (CI)** e **Entrega Contínua (CD)** é indispensável. Essas práticas permitem compilar, testar e implantar aplicações Java de forma automática, reduzindo erros manuais e aumentando a confiabilidade.

💡 **Dica:** CI/CD é uma cultura de automação e qualidade, não apenas uma pipeline de scripts.

---

## 14.2 Conceitos Fundamentais

### ♦ Integração Contínua (CI)

- Compila, testa e valida código sempre que novas alterações são integradas ao repositório.
- Detecta falhas rapidamente, evitando surpresas no deploy.

### ♦ Entrega Contínua (CD)

- Extensão da CI que leva o software a um ambiente pronto para produção.
  - Pode incluir deploy automático ou semi-automático.
- 

## 14.3 Ferramentas Populares

- ✓ **Jenkins** — O mais utilizado no mercado corporativo, flexível e open-source.
  - ✓ **GitLab CI/CD** — Integrado ao GitLab, ideal para projetos versionados.
  - ✓ **GitHub Actions** — Boa integração com o GitHub.
  - ✓ **CircleCI** — Rápido e fácil de integrar.
  - ✓ **Bamboo** — Focado em integrações corporativas (Atlassian).
- 

## 14.4 Estrutura de um Pipeline CI/CD

- 1 **Build**: Compilação e resolução de dependências (ex.: Maven ou Gradle).
  - 2 **Testes**: Testes unitários, integração e cobertura de código.
  - 3 **Análise Estática**: Ferramentas como SonarQube ou PMD.
  - 4 **Empacotamento**: Criação de artefatos (JAR, WAR ou Docker).
  - 5 **Deploy**: Entrega em ambiente de homologação ou produção.
- 💡 **Dica**: Separe ambientes de homologação e produção para evitar impactos diretos.
- 

## 14.5 Exemplo com Jenkins

- 1 **Instalação**: Baixe e execute o `.war`:  

```
java -jar jenkins.war
```
- 2 **Job Freestyle**:
  - Configure repositório Git.
  - Execute:  

```
mvn clean install
```
  - Adicione etapa de testes.
  - Configure notificação (e-mail ou Slack).

### 3 Pipeline Declarativo:

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        sh 'mvn clean package'
      }
    }
    stage('Test') {
      steps {
        sh 'mvn test'
      }
    }
    stage('Deploy') {
      steps {
        sh './deploy.sh'
      }
    }
  }
}
```

\Explicar uma regra dentro do código:

- **agent any:** Usa qualquer executor disponível.
  - **stages:** Define as etapas da pipeline (Build, Test, Deploy).
  - **sh:** Executa comandos de shell.
- 

## 14.6 Exemplo com GitLab CI/CD

No arquivo `.gitlab-ci.yml`:

```
stages:
  - build
  - test
  - deploy

build:
  stage: build
  script:
    - mvn clean package

test:
  stage: test
  script:
    - mvn test

deploy:
  stage: deploy
  script:
    - bash deploy.sh
```

---

## 14.7 Boas Práticas

- ✓ Configure integração com o Git para gatilhos automáticos.
  - ✓ Use SonarQube para qualidade de código.
  - ✓ Utilize Docker para criar ambientes consistentes.
  - ✓ Integre com Kubernetes para deploy escalável.
  - ✓ Automatize rollback em caso de falhas.
- 

## 14.8 Monitoramento e Alertas

- Configure notificações de falhas (Slack, e-mail, MS Teams).
  - Use dashboards como Grafana e Prometheus para métricas.
- 

## 14.9 Conclusão

CI/CD transforma o ciclo de vida de software, garantindo agilidade, qualidade e previsibilidade. Com as ferramentas certas, o time de desenvolvimento pode focar em entregar valor, enquanto a automação cuida do resto.

No próximo capítulo, podemos explorar **Testes Automatizados e Cobertura de Código**, consolidando as melhores práticas de qualidade para o ciclo de vida do software Java.

---

# Capítulo 15. Testes Automatizados e Cobertura de Código em Java

## 15.1 Introdução

Testes automatizados são a base da qualidade em aplicações corporativas Java. Garantem que o código funcione como esperado, mesmo após alterações, facilitando a manutenção e prevenindo regressões. Cobertura de código é uma métrica essencial para avaliar a eficácia dos testes.

💡 **Dica:** Testar não é apenas escrever testes, mas garantir que falhas sejam detectadas e corrigidas antes de chegarem ao usuário.

---

## 15.2 Tipos de Testes

### ♦ Testes Unitários

- Validam pequenas partes da aplicação isoladamente (métodos ou classes).
- Ferramentas: JUnit 5, Mockito.

### ♦ Testes de Integração

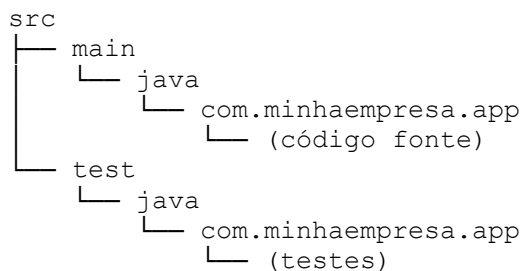
- Validam a interação entre componentes (ex.: JPA, REST).
- Ferramentas: Spring Boot Test, Testcontainers.

### ♦ Testes de Aceitação

- Simulam cenários reais de uso.
  - Ferramentas: Cucumber, Selenium.
- 

## 15.3 Estrutura de Testes em Java

Estruture a hierarquia de testes no projeto:



**Dica:** Mantenha os testes paralelos à estrutura de pacotes da aplicação.

---

## 15.4 JUnit 5

Configuração no Maven:

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.8.2</version>
  <scope>test</scope>
</dependency>
```

Exemplo de teste unitário:

```
@ExtendWith(MockitoExtension.class)
class ClienteServiceTest {

    @Mock
    private ClienteRepository clienteRepository;
```



```

@InjectMocks
private ClienteService clienteService;

@Test
void deveRetornarClientePorId() {
    Cliente cliente = new Cliente(1L, "Ana");

    when(clienteRepository.findById(1L)).thenReturn(Optional.of(cliente));

    Cliente resultado = clienteService.buscarClientePorId(1L);

    assertEquals("Ana", resultado.getNome());
}
}


```

\Explicar uma regra dentro do código:

- **@Mock:** Cria um mock de dependência.
  - **@InjectMocks:** Injeta os mocks na classe de teste.
  - **assertEquals:** Valida o resultado esperado.
- 

## 15.5 Mockito

Mockito é o framework padrão para mocks em Java.

 **Dica:** Use `when().thenReturn()` para simular comportamentos e `verify()` para checar interações.

---

## 15.6 Testes de Integração com Spring Boot

```

@SpringBootTest
@AutoConfigureTestDatabase
class ClienteRepositoryIntegrationTest {

    @Autowired
    private ClienteRepository clienteRepository;

    @Test
    void devePersistirCliente() {
        Cliente cliente = new Cliente(null, "Ana");
        Cliente salvo = clienteRepository.save(cliente);

        assertNotNull(salvo.getId());
    }
}

```


---

## 15.7 Cobertura de Código

### ♦ Jacoco

Adicione ao `pom.xml`:

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.8</version>
  <executions>
    <execution>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>report</id>
      <phase>verify</phase>
      <goals>
        <goal>report</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

 **Dica:** Execute `mvn verify` para gerar relatórios de cobertura.

---

## 15.8 Análise de Cobertura

- ✓ Foque na cobertura de código crítico (services, controllers).
  - ✓ Busque cobrir ao menos 80% do código relevante.
  - ✓ Não confunda cobertura com qualidade: mesmo 100% coberto pode ter falhas de lógica!
- 

## 15.9 Boas Práticas

- ✓ Use nomes descritivos para testes.
  - ✓ Separe unitários de integração.
  - ✓ Evite dependências externas desnecessárias (mock sempre que possível).
  - ✓ Automatize testes na pipeline de CI/CD (Capítulo 14).
  - ✓ Revise relatórios de cobertura regularmente.
- 

## 15.10 Conclusão

Testes automatizados e cobertura de código são a espinha dorsal da qualidade de software Java. Usando JUnit, Mockito, Spring Boot Test e Jacoco, você garante uma base sólida para entregas confiáveis e seguras.


No próximo capítulo, poderemos explorar **Mensageria Assíncrona com Java (Kafka, RabbitMQ)** para integrar sistemas de forma escalável e performática.

---

## Capítulo 16. Mensageria Assíncrona com Java (Kafka, RabbitMQ)

### 16.1 Introdução

Em sistemas modernos, a mensageria assíncrona é um pilar para desacoplar componentes, melhorar escalabilidade e garantir resiliência. Usando **Apache Kafka** e **RabbitMQ**, podemos orquestrar comunicação entre microsserviços ou aplicações monolíticas de forma confiável e eficiente.

 **Dica:** Mensageria não é apenas fila — é estratégia para lidar com alta demanda e falhas de rede.

---

### 16.2 Conceitos Fundamentais

#### ♦ Assíncrono x Síncrono

- **Síncrono:** O produtor espera a resposta do consumidor.
- **Assíncrono:** O produtor apenas envia a mensagem e segue, sem aguardar resposta.

#### ♦ Fila x Tópico

- **Fila (RabbitMQ):** Cada mensagem vai para apenas um consumidor.
  - **Tópico (Kafka):** Mensagens são publicadas e podem ser lidas por múltiplos consumidores (broadcast).
- 

### 16.3 Apache Kafka

#### 16.3.1 O que é Kafka?

- Plataforma de streaming distribuído.
- Baseada em logs de mensagens.
- Ideal para alto volume de dados (logs, eventos).

#### 16.3.2 Conceitos-Chave

- **Broker:** Nó do cluster Kafka.
  - **Producer:** Envia mensagens para o Kafka.
  - **Consumer:** Lê mensagens de tópicos.
  - **Topic:** Canal de distribuição de mensagens.
-

## 16.4 RabbitMQ

### 16.4.1 O que é RabbitMQ?

- Broker de mensageria baseado em filas.
- Suporta protocolos como AMQP, MQTT.
- Ideal para workflows de processamento.

### 16.4.2 Conceitos-Chave

- **Exchange:** Roteia mensagens para filas.
  - **Queue:** Armazena mensagens.
  - **Producer:** Envia mensagens para exchanges.
  - **Consumer:** Lê mensagens da fila.
- 

## 16.5 Configuração com Spring Boot

### ♦ Spring Kafka

pom.xml:

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>
```

**Produtor:**

```
@Autowired
private KafkaTemplate<String, String> kafkaTemplate;
```

```
public void enviarMensagem(String mensagem) {
    kafkaTemplate.send("meu-topico", mensagem);
}
```

**Consumidor:**

```
@KafkaListener(topics = "meu-topico", groupId = "meu-grupo")
public void consumirMensagem(String mensagem) {
    System.out.println("Mensagem recebida: " + mensagem);
}
```

\Explicar uma regra dentro do código:

- **@KafkaListener:** Anotação que registra o método como consumidor de mensagens do tópico especificado.
- 

### ♦ Spring AMQP (RabbitMQ)

pom.xml:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

#### Produtor:

```
@Autowired
private RabbitTemplate rabbitTemplate;

public void enviarMensagem(String mensagem) {
    rabbitTemplate.convertAndSend("minha-fila", mensagem);
}
```

#### Consumidor:

```
@RabbitListener(queues = "minha-fila")
public void consumirMensagem(String mensagem) {
    System.out.println("Mensagem recebida: " + mensagem);
}
```

---

## 16.6 Boas Práticas

- ✓ Defina padrões de nomes para tópicos e filas.
  - ✓ Monitore métricas (offsets, backlog).
  - ✓ Use **Dead Letter Queues** para tratar falhas.
  - ✓ Evite acoplamento forte — utilize DTOs e versionamento de mensagens.
  - ✓ Proteja tópicos/filas com autenticação e autorização.
- 

## 16.7 Padrões Arquiteturais

- **Event Sourcing:** Armazena estado como eventos.
  - **CQRS:** Segrega leitura e escrita para escalabilidade.
  - **Saga:** Orquestra transações distribuídas via eventos.
- 

## 16.8 Conclusão

Kafka e RabbitMQ são ferramentas poderosas para mensageria assíncrona. Combinadas ao Spring Boot, permitem construir aplicações escaláveis e resilientes.


No próximo capítulo, podemos explorar **Microserviços e Comunicação Assíncrona**, conectando toda a arquitetura distribuída com segurança e eficiência.

---

## Capítulo 17. Microsserviços e Comunicação Assíncrona


### 17.1 Introdução


Microsserviços são uma abordagem arquitetural moderna que foca na divisão de aplicações em serviços independentes, cada um responsável por uma funcionalidade específica. A comunicação assíncrona, por meio de mensageria (Kafka, RabbitMQ), é um dos principais mecanismos para garantir escalabilidade e resiliência.


 **Dica:** Microsserviços não são apenas sobre divisão técnica — são sobre autonomia, escalabilidade e evolução independente.

---

### 17.2 Microsserviços em Java

Frameworks mais comuns:  **Spring Boot** — Base para construir serviços rápidos e produtivos.

 **Quarkus** — Focado em cloud-native e performance.

 **Micronaut** — Leve e rápido para aplicações reativas.

---

### 17.3 Comunicação entre Microsserviços

#### ♦ Comunicação Síncrona (REST)

- HTTP (REST APIs) — Fácil de implementar, mas pode criar acoplamento excessivo.

#### ♦ Comunicação Assíncrona (Mensageria)

- Kafka, RabbitMQ ou outros brokers.
  - Permite escalabilidade horizontal, tolerância a falhas e menor acoplamento.
- 

### 17.4 Padrões de Comunicação Assíncrona

#### ♦ Event-Driven Architecture (EDA)

- Cada serviço emite eventos para um barramento (Kafka).
- Outros serviços reagem assinando tópicos.

## ◆ Command-Query Responsibility Segregation (CQRS)

- Separa comandos (alterações) e queries (leitura).
- Usa eventos para sincronizar estados.

## ◆ Saga Pattern

- Orquestra transações distribuídas por meio de eventos.
  - Exemplo: pedido confirmado → pagamento processado → estoque atualizado.
- 

## 17.5 Implementando com Spring Boot e Kafka

Produtor:

```
@Autowired
private KafkaTemplate<String, PedidoEvent> kafkaTemplate;

public void emitirPedidoCriado(PedidoEvent evento) {
    kafkaTemplate.send("pedidos", evento);
}
```

Consumidor:

```
@KafkaListener(topics = "pedidos")
public void processarPedidoCriado(PedidoEvent evento) {
    System.out.println("Pedido recebido: " + evento.getId());
}
```

\Explicar uma regra dentro do código:

- **PedidoEvent:** Classe DTO para transportar dados.
  - **@KafkaListener:** Consome eventos do tópico “pedidos”.
  - **KafkaTemplate:** Utilizado para enviar eventos para o broker.
- 

## 17.6 Implementando com Spring Boot e RabbitMQ

Produtor:

```
@Autowired
private RabbitTemplate rabbitTemplate;

public void emitirPedidoCriado(PedidoEvent evento) {
    rabbitTemplate.convertAndSend("pedidos.exchange", "pedido.criado",
    evento);
}
```

Consumidor:

```
@RabbitListener(queues = "pedidos.queue")
public void processarPedidoCriado(PedidoEvent evento) {
    System.out.println("Pedido recebido: " + evento.getId());
}
```

---

## 17.7 Boas Práticas

- ✓ Use DTOs para desacoplar serviços.
  - ✓ Inclua versionamento de eventos para evoluir APIs.
  - ✓ Implemente **Dead Letter Queues** para tratar falhas.
  - ✓ Monitore offset lag e métricas de filas.
  - ✓ Documente eventos e fluxos (Event Catalog).
- 

## 17.8 Observabilidade

- **Tracing distribuído:** Spring Cloud Sleuth, OpenTelemetry.
  - **Logs centralizados:** ELK Stack ou Grafana Loki.
  - **Monitoramento de filas:** Kafka Manager, RabbitMQ Management Plugin.
- 

## 17.9 Conclusão

Microserviços e comunicação assíncrona transformam aplicações Java em plataformas escaláveis e resilientes. Dominar Kafka, RabbitMQ e Spring Boot é fundamental para construir soluções corporativas de alta qualidade.

No próximo capítulo, podemos explorar **Segurança em Aplicações Java (OAuth2, JWT)**, garantindo confidencialidade, integridade e autenticação em ambientes distribuídos.

---

# Capítulo 18. Segurança em Aplicações Java (OAuth2, JWT)

## 18.1 Introdução

Segurança é um pilar fundamental em qualquer aplicação corporativa. Com a crescente adoção de microserviços e APIs, autenticação e autorização robustas são cruciais para proteger dados e serviços. O **OAuth2** e o **JSON Web Token (JWT)** são padrões amplamente adotados para implementar segurança em aplicações Java modernas.

💡 **Dica:** Segurança não é apenas adicionar uma camada de proteção — é pensar na arquitetura como um todo, considerando confidencialidade, integridade e disponibilidade.

---



## 18.2 Conceitos Fundamentais

### ♦ Autenticação x Autorização

- **Autenticação:** Verifica a identidade do usuário.
- **Autorização:** Define o que o usuário pode acessar.

### ♦ OAuth2

- Protocolo para delegação de autorização.
- Permite que aplicações (clientes) acessem recursos protegidos em nome de um usuário.

### ♦ JWT (JSON Web Token)

- Token auto-contido, assinado digitalmente.
  - Transporta informações de identidade e permissões.
  - Base para autenticação stateless em microserviços.
- 

## 18.3 Implementando OAuth2 com Spring Security

### 18.3.1 Adicionando Dependências

```
<dependency>
  <groupId>org.springframework.boot</groupId>

  <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

---

### 18.3.2 Configurando o Resource Server

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests(authorize -> authorize
                .antMatchers("/public/**").permitAll()
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer()
            .jwt();
    }
}
```

\Explicar uma regra dentro do código:

- **@EnableWebSecurity:** Habilita o módulo de segurança no Spring.

- `oauth2ResourceServer().jwt()`: Configura o servidor para validar tokens JWT.
- 

## 18.4 Gerando Tokens JWT

- Geralmente é feito por um Authorization Server (ex.: Keycloak, Auth0).
- O token é assinado (HS256 ou RS256) e contém claims como:
  - `sub`: identificador do usuário.
  - `exp`: data de expiração.
  - `roles`: lista de permissões.



**Dica:** Nunca armazene senhas ou dados sensíveis dentro do JWT.

---

## 18.5 Validando Tokens JWT

O Spring Security valida automaticamente tokens JWT (via chave pública ou segredo). Claims podem ser extraídas via `JwtAuthenticationToken`:

```
@GetMapping("/perfil")
public String getPerfil(@AuthenticationPrincipal Jwt principal) {
    return "Usuário: " + principal.getSubject();
}
```

---

## 18.6 Boas Práticas

- ✓ Use HTTPS para proteger tokens em trânsito.
  - ✓ Configure tempo de expiração curto (acesso) + refresh tokens.
  - ✓ Utilize RS256 para tokens assíncronos (chave pública/privada).
  - ✓ Implemente revogação e blacklist de tokens (via Redis ou banco).
  - ✓ Proteja endpoints sensíveis com roles e scopes.
- 

## 18.7 Integração com Microsserviços

- Cada microsserviço valida o JWT localmente (stateless).
  - Authorization Server centraliza autenticação e fornece tokens.
  - Use o padrão **API Gateway** para consolidar segurança e roteamento.
-

## 18.8 Ferramentas Recomendadas

- 🔒 **Keycloak** — Identity Provider open-source para OAuth2 e OIDC.
  - 🔒 **Auth0** — Plataforma SaaS para gerenciamento de identidade.
  - 🔒 **Spring Security OAuth2** — Framework robusto para implementar segurança corporativa.
- 

## 18.9 Conclusão

Segurança é mais do que criptografia — é pensar em todo o ciclo de vida da autenticação e autorização. Com OAuth2 e JWT, você implementa autenticação moderna e escalável em Java, seja em monólitos ou microsserviços.

No próximo capítulo, podemos explorar **Capítulo 19 — APIs REST e HATEOAS em Java**, abordando como tornar APIs mais ricas e autoexplicativas.

---

# Capítulo 19. APIs REST e HATEOAS em Java

## 19.1 Introdução

No ecossistema Java, a construção de APIs RESTful é uma prática consolidada. Entretanto, muitos desenvolvedores negligenciam a importância de torná-las verdadeiramente **autoexplicativas** e **descobertas por navegação**, conforme preconizado por Roy Fielding em sua tese de REST. É aqui que o HATEOAS (Hypermedia As The Engine Of Application State) se torna uma ferramenta poderosa.

💡 **Dica:** APIs REST não são apenas endpoints — são contratos vivos entre o cliente e o servidor, e devem evoluir de forma sustentável.

---

## 19.2 Fundamentos de REST

- **Recursos:** Representações de objetos do domínio (ex.: `/clientes`, `/pedidos`).
- **Verbos HTTP:**
  - GET: Consulta.
  - POST: Criação.
  - PUT: Atualização total.

- PATCH: Atualização parcial.
    - DELETE: Exclusão.
  - **Códigos de Status:** 200 OK, 201 Created, 204 No Content, 400 Bad Request, etc.
- 

## 19.3 O que é HATEOAS?

### Hypermedia As The Engine Of Application State:

- A resposta da API fornece **links de navegação** que orientam o cliente a próxima ação.
- Promove desacoplamento entre cliente e servidor.
- Facilita evolução da API sem quebrar clientes existentes.

Exemplo de resposta com HATEOAS:

```
{
  "id": 123,
  "nome": "Ana",
  "email": "ana@example.com",
  "_links": {
    "self": { "href": "/clientes/123" },
    "pedidos": { "href": "/clientes/123/pedidos" },
    "atualizar": { "href": "/clientes/123", "method": "PUT" }
  }
}
```

---

## 19.4 Implementando APIs REST com Spring Boot

### 19.4.1 Dependências

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

---

### 19.4.2 Exemplo de Endpoint Básico

```
@RestController
@RequestMapping("/clientes")
public class ClienteController {

    @GetMapping("/{id}")
    public ResponseEntity<ClienteDTO> buscarPorId(@PathVariable Long id) {
        ClienteDTO cliente = clienteService.buscarPorId(id);
        return ResponseEntity.ok(cliente);
    }

    @PostMapping
```

```

    public ResponseEntity<ClienteDTO> criar(@RequestBody ClienteDTO
clienteDTO) {
        ClienteDTO criado = clienteService.criar(clienteDTO);
        return ResponseEntity.status(HttpStatus.CREATED).body(criado);
    }
}

```

\Explicar uma regra dentro do código:

- **@RestController:** Indica que a classe disponibiliza endpoints REST.
  - **ResponseEntity:** Permite controlar o status HTTP da resposta.
- 

## 19.5 Adicionando HATEOAS com Spring HATEOAS

### 19.5.1 Dependência

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>

```

---

### 19.5.2 Usando RepresentationModel

```

public class ClienteModel extends RepresentationModel<ClienteModel> {
    private Long id;
    private String nome;
    private String email;

    // getters e setters
}

```

---

### 19.5.3 Construindo o Modelo com Links

```

@RestController
@RequestMapping("/clientes")
public class ClienteController {

    @Autowired
    private ClienteService clienteService;

    @GetMapping("/{id}")
    public ClienteModel buscarPorId(@PathVariable Long id) {
        ClienteDTO clienteDTO = clienteService.buscarPorId(id);
        ClienteModel model = new ClienteModel();
        model.setId(clienteDTO.getId());
        model.setNome(clienteDTO.getNome());
        model.setEmail(clienteDTO.getEmail());

        model.add(linkTo(methodOn(ClienteController.class).buscarPorId(id)).withSelfRel());
    }
}

```

```
model.add(linkTo(methodOn(PedidoController.class).listarPorCliente(id))
    .withRel("pedidos"));

    return model;
}
}
```

\Explicar uma regra dentro do código:

- `RepresentationModel`: Base para adicionar links ao recurso.
  - `linkTo(methodOn(...))`: Gera links dinamicamente usando Spring MVC.
- 

## 19.6 Boas Práticas

- ✓ Use nomes consistentes para recursos (substantivos).
  - ✓ Forneça links para as ações possíveis (CRUD e relacionados).
  - ✓ Documente a API usando Swagger/OpenAPI.
  - ✓ Utilize paginação e ordenação para coleções.
  - ✓ Gerencie versões de forma adequada (ex.: `/v1/clientes`).
- 

## 19.7 Conclusão

APIs REST com HATEOAS são poderosas para evoluir aplicações distribuídas de forma sustentável. Com Spring Boot e Spring HATEOAS, você constrói APIs autoexplicativas, fáceis de manter e evoluir.


No próximo capítulo, podemos explorar **Capítulo 20 — Padrões de Versionamento de API**, fundamental para manter a compatibilidade de longo prazo.

---

# Capítulo 20. Padrões de Versionamento de API

## 20.1 Introdução

Versionar APIs é essencial para garantir compatibilidade com clientes existentes e permitir evolução controlada. Sem versionamento, cada mudança pode causar quebras em sistemas dependentes, comprometendo a estabilidade da arquitetura corporativa.

 **Dica:** O versionamento é como uma trilha sonora para o código: ele orchestra o ritmo da evolução sem desafinar a harmonia com os clientes.

---

## 20.2 Por que versionar?

- **Evolução controlada:** Adiciona novos recursos sem quebrar clientes existentes.
  - **Compatibilidade:** Permite que consumidores escolham quando migrar para a nova versão.
  - **Governança:** Organiza a documentação e facilita a manutenção.
- 

## 20.3 Estratégias de Versionamento

### ♦ Versionamento na URL

O mais comum e explícito:

```
GET /api/v1/clientes
```

**Vantagens:**

- Fácil de identificar.
- Permite múltiplas versões coexistindo.

**Desvantagens:**

- URLs podem ficar poluídas.
  - Inviável para alguns frameworks RESTful puristas.
- 

### ♦ Versionamento via Header

Define a versão no cabeçalho HTTP:

```
Accept: application/vnd.empresa.v1+json
```

**Vantagens:**

- Mantém URLs limpas.
- Permite múltiplas representações por endpoint.

**Desvantagens:**

- Mais difícil de testar manualmente.
  - Pode confundir clientes menos experientes.
- 

### ♦ Versionamento por Media Type

Define a versão no tipo de mídia:

```
Content-Type: application/vnd.empresa.v1+json
```

**Vantagens:**

- Padronizado no HTTP.

- Permite versionar por recurso.

**Desvantagens:**

- Pouco conhecido entre desenvolvedores.
  - Ferramentas podem não suportar bem.
- 

## 20.4 Implementando Versionamento em Spring Boot

### 20.4.1 Versionamento na URL

```
@RestController
@RequestMapping("/api/v1/clientes")
public class ClienteControllerV1 {
    // implementação da versão 1
}

@RestController
@RequestMapping("/api/v2/clientes")
public class ClienteControllerV2 {
    // implementação da versão 2
}
```

\Explicar uma regra dentro do código:

- Criaremos controladores específicos para cada versão.
  - URLs diferentes para cada versão.
- 

### 20.4.2 Versionamento via Header

```
@RestController
@RequestMapping("/api/clientes")
public class ClienteController {

    @GetMapping(produces = "application/vnd.empresa.v1+json")
    public ClienteDTO listarV1() {
        // implementação versão 1
    }

    @GetMapping(produces = "application/vnd.empresa.v2+json")
    public ClienteV2DTO listarV2() {
        // implementação versão 2
    }
}
```

---

## 20.5 Boas Práticas

- ✓ Evite quebrar clientes existentes; sempre comunique depreciações.
- ✓ Documente todas as versões de forma clara.
- ✓ Adote automação para testes de múltiplas versões.
- ✓ Use OpenAPI/Swagger para descrever as versões.
- ✓ Avalie custos de manutenção antes de proliferar muitas versões.



---

## 20.6 Conclusão

Versionar APIs é uma arte que equilibra inovação e estabilidade. Cada abordagem tem seus prós e contras — o importante é escolher aquela que melhor se adapta ao seu contexto corporativo e técnico.


No próximo capítulo, podemos explorar **Capítulo 21 — Testes Automatizados em Java**, abordando JUnit, Mockito e Testcontainers para garantir qualidade contínua em sistemas corporativos.

---

## Capítulo 21. Testes Automatizados em Java (JUnit, Mockito, Testcontainers)

### 21.1 Introdução

Testes automatizados são fundamentais para garantir a qualidade e a evolução segura de qualquer aplicação corporativa. Em Java, frameworks como **JUnit**, **Mockito** e **Testcontainers** permitem criar suítes de testes robustas, confiáveis e integradas ao ciclo de desenvolvimento.

 **Dica:** Testes não são apenas para validar código — são uma documentação viva do comportamento da aplicação.

---

### 21.2 Tipos de Testes

- **Testes Unitários:** Validam classes ou métodos isolados, sem dependências externas.
  - **Testes de Integração:** Validam a interação entre múltiplos componentes.
  - **Testes de Aceitação/End-to-End:** Validam o comportamento da aplicação como um todo.
-

## 21.3 Frameworks Essenciais

### ♦ JUnit

- Framework de teste unitário mais utilizado na comunidade Java.
- Permite escrever, organizar e executar testes de forma padronizada.

### ♦ Mockito

- Framework para criar objetos simulados (mocks) e testar comportamentos isolados.

### ♦ Testcontainers

- Framework para executar containers Docker durante os testes, útil para bancos de dados e serviços externos.
- 

## 21.4 Testes Unitários com JUnit

### 21.4.1 Dependência (JUnit 5)

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.9.3</version>
  <scope>test</scope>
</dependency>
```

---

### 21.4.2 Exemplo de Teste Unitário

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculadoraTest {

    @Test
    void testSoma() {
        Calculadora calc = new Calculadora();
        int resultado = calc.somar(2, 3);
        assertEquals(5, resultado, "A soma deve ser 5");
    }
}
```

\Explicar uma regra dentro do código:

- **@Test:** Marca o método como teste.
  - **assertEquals:** Valida o resultado esperado.
-

## 21.5 Testes com Mockito

### 21.5.1 Dependência

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>5.4.0</version>
  <scope>test</scope>
</dependency>
```

---

### 21.5.2 Exemplo de Uso

```
import org.junit.jupiter.api.Test;
import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.*;

public class ClienteServiceTest {

    @Test
    void testBuscarCliente() {
        ClienteRepository repoMock = mock(ClienteRepository.class);
        when(repoMock.findById(1L)).thenReturn(Optional.of(new
        Cliente(1L, "Ana")));

        ClienteService service = new ClienteService(repoMock);
        Cliente cliente = service.buscarPorId(1L);

        assertEquals("Ana", cliente.getNome());
        verify(repoMock).findById(1L);
    }
}
```

---

## 21.6 Testes de Integração com Testcontainers

### 21.6.1 Dependência

```
<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>1.19.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>postgresql</artifactId>
  <version>1.19.0</version>
  <scope>test</scope>
</dependency>
```

---

## 21.6.2 Exemplo com PostgreSQL

```
import org.junit.jupiter.api.Test;
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

@Testcontainers
public class ClienteRepositoryIntegrationTest {

    @Container
    private static final PostgreSQLContainer<?> postgres =
        new PostgreSQLContainer<>("postgres:15-alpine")
            .withDatabaseName("testdb")
            .withUsername("test")
            .withPassword("test");

    @Test
    void testContainerRunning() {
        assertTrue(postgres.isRunning());
    }
}
```

\Explicar uma regra dentro do código:

- **@Testcontainers:** Integra o ciclo de vida do container com os testes.
  - **@Container:** Garante que o container é gerenciado pelo framework de teste.
- 

## 21.7 Boas Práticas

- ✓ Separe testes unitários e de integração em pacotes diferentes.
  - ✓ Configure pipelines de CI/CD para executar os testes automaticamente.
  - ✓ Use bancos de dados efêmeros (Docker) para evitar poluição.
  - ✓ Automatize a limpeza do ambiente após cada teste.
- 

## 21.8 Conclusão

Testes automatizados são alicerces de software de qualidade. Com JUnit, Mockito e Testcontainers, você constrói aplicações Java confiáveis, escaláveis e preparadas para evoluir.

No próximo capítulo, podemos explorar **Capítulo 22 — Boas Práticas de Deploy e CI/CD em Java**, aprofundando a integração contínua e a entrega contínua para aplicações corporativas.

---

## Capítulo 22. Estudos de Casos

### 22.1 Introdução

Chegamos à orquestra final deste e-book: **os estudos de casos**, onde teoria e prática dançam em harmonia. Cada framework, cada padrão e cada conceito aqui reunido se manifesta na rotina das empresas de tecnologia — da startup ágil à corporação robusta.

💡 **Dica:** Entender o que é mais utilizado pelas empresas é como observar as partituras de uma orquestra; cada nota tem seu tempo, e cada framework, seu lugar.

---

### 22.2 A Sinfonia da Arquitetura Moderna

Após explorar fundamentos de Java, APIs REST, ORM, versionamento, testes e muito mais, podemos vislumbrar um **ecossistema de práticas consolidadas**. Segue o que predomina nas empresas:

---

#### ♦ Frameworks ORM (JPA e Hibernate)

- **Uso predominante:** Hibernate (via JPA) reina soberano no mundo corporativo, devido à sua robustez e integração com frameworks como Spring Boot.
  - **Vantagem:** Facilita o mapeamento objeto-relacional e a escrita de queries personalizadas.
  - **Curiosidade:** Alguns bancos de dados legados ainda exigem JDBC puro, mas Hibernate/JPA é a norma.
- 

#### ♦ Spring Boot como Plataforma Principal

- **Uso predominante:** O Spring Boot domina o cenário para APIs RESTful, microsserviços e até integrações de mensageria (Kafka, RabbitMQ).
  - **Vantagem:** Produtividade e ecossistema maduro.
  - **Prática comum:** Uso de Starter Projects para acelerar o desenvolvimento.
-

### ♦ APIs REST e HATEOAS

- **Uso predominante:** APIs RESTful representam o padrão de integração. O uso de HATEOAS ainda é moderado, mas ganha força em sistemas que exigem evolutividade e autoexploração.
  - **Vantagem:** Facilita manutenção e extensibilidade.
  - **Prática comum:** Uso de `linkTo()` do Spring HATEOAS e documentação via Swagger/OpenAPI.
- 

### ♦ Versionamento de API

- **Uso predominante:** Versionamento via URL (`/api/v1/recursos`) é o mais adotado pela sua simplicidade e clareza.
  - **Vantagem:** Facilidade de manutenção e suporte a múltiplas versões.
  - **Nota:** Versionamento via header é usado em APIs mais sofisticadas ou em empresas que priorizam contratos REST estritos.
- 

### ♦ Testes Automatizados

- **Uso predominante:**
    - **JUnit 5:** Para testes unitários.
    - **Mockito:** Para mockar dependências em testes unitários.
    - **Testcontainers:** Para testes de integração robustos com banco de dados.
  - **Vantagem:** Confiança nas entregas e integração com pipelines de CI/CD.
  - **Curiosidade:** Startups às vezes começam apenas com testes unitários, evoluindo para integração à medida que o projeto cresce.
- 

## 22.3 Integração Contínua e Entrega Contínua (CI/CD)

- **Ferramentas principais:** Jenkins, GitLab CI, GitHub Actions.
  - **Uso predominante:** Integração automática de testes e deploys automatizados com Docker e Kubernetes.
  - **Prática comum:** Pipelines com steps de build, testes (unitários + integração) e deploy em ambientes de homologação e produção.
-


## 22.4 Estudos de Casos Reais

Abaixo, alguns **cenários reais** baseados em práticas observadas em empresas de tecnologia:

---

### Startup Ágil (5-10 devs)


- **Stack:** Spring Boot + Hibernate (JPA) + PostgreSQL.
- **Testes:** JUnit + Mockito (testes unitários).
- **Versionamento:** URL (/api/v1/).
- **Deploy:** Heroku ou AWS com pipelines GitHub Actions.

 **Dica:** Menos burocracia, mais produtividade.

---

### Empresa de Médio Porte (50-100 devs)


- **Stack:** Spring Boot + JPA + Redis (cache) + Kafka (mensageria).
- **Testes:** JUnit + Mockito + Testcontainers.
- **Versionamento:** URL + Header para clientes externos.
- **Deploy:** Kubernetes em cloud privada com GitLab CI/CD.

 **Dica:** Aqui, a automação de testes e o uso de containers são mandatórios.

---

### Corporação Multinacional (200+ devs)

- **Stack:** Microsserviços Spring Boot + Hibernate + Oracle + Elasticsearch.
- **Testes:** JUnit + Mockito + Testcontainers + integração com SonarQube (cobertura de código).
- **Versionamento:** URL + Media Type para suportar múltiplos contratos de APIs.
- **Deploy:** Jenkins Pipeline + Docker + Kubernetes (multi-ambientes).

 **Dica:** Versionamento sofisticado, alta governança de testes e pipelines extensos.

---

## 22.5 Conclusão

As empresas de tecnologia — sejam startups ou gigantes corporativos — convergem em torno de alguns pilares essenciais: **Spring Boot**, **JPA/Hibernate**, **APIs REST**, **testes automatizados** e **versionamento de APIs**. Este compêndio é, portanto, um retrato vivo das práticas modernas que mantêm a sinfonia corporativa afinada.

No próximo capítulo, podemos explorar temas como **boas práticas de segurança**, **monitoramento** ou até **mensageria corporativa**.

## Conclusão do E-Book Java Nível Intermediário

Chegamos ao final desta trilha de aprendizado — uma jornada que começou com os alicerces de Java e se expandiu pelos labirintos das melhores práticas, frameworks modernos e padrões arquiteturais que guiam o desenvolvimento de software nas empresas de tecnologia.

Ao longo dos **22 capítulos**, percorremos:

- ✓ **Fundamentos de Java** — revisitando conceitos essenciais, como sintaxe, orientação a objetos e melhores práticas de codificação.
- ✓ **Frameworks ORM** — mergulhamos no poder do Hibernate e JPA para manipulação de bancos de dados relacionais.
- ✓ **Spring Boot** — a pedra angular do desenvolvimento moderno de APIs RESTful, microsserviços e integrações corporativas.
- ✓ **APIs REST e Versionamento** — aprendemos a construir APIs limpas, escaláveis e a versionar recursos para evoluir sem quebrar clientes.
- ✓ **Testes Automatizados** — apresentando JUnit, Mockito e Testcontainers para garantir qualidade contínua.
- ✓ **Integração Contínua e Entrega Contínua (CI/CD)** — orquestrando pipelines de build, testes e deploy.
- ✓ **Estudos de Casos** — aplicando o conhecimento teórico em cenários reais, conectando a sinfonia do aprendizado com o palco da indústria.

💡 **Dica Final:** Continue praticando! A excelência é forjada no compasso da disciplina e da curiosidade.

---

**Obrigada por chegar até aqui, desenvolvedores!** Vocês são a orquestra que transforma códigos em sinfonias, e o palco corporativo espera por vocês.

Sigam em frente com coragem, pois o futuro da tecnologia é escrito por quem ousa aprender e criar. 🚀


Se quiser, podemos avançar para temas como segurança corporativa, mensageria assíncrona ou até mesmo integrações com nuvem — a decisão é sua!

---



## Bibliografia

- BLOCH, Joshua. **Effective Java**. 3. ed. Boston: Addison-Wesley, 2018.
- BAUER, Christian; KING, Gavin. **Java Persistence with Hibernate**. 2. ed. Greenwich: Manning, 2015.
- GAMMA, Erich et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. Boston: Addison-Wesley, 1994.
- MARTIN, Robert C. **Clean Code: A Handbook of Agile Software Craftsmanship**. Upper Saddle River: Prentice Hall, 2009.
- WALLS, Craig. **Spring in Action**. 6. ed. Shelter Island: Manning Publications, 2022.
- BECK, Kent. **Test-Driven Development: By Example**. Boston: Addison-Wesley, 2003.
- HIBERNATE ORM. **Hibernate ORM Documentation**. Disponível em: <https://hibernate.org/orm/documentation/>. Acesso em: 4 jun. 2025.
- SPRING BOOT. **Spring Boot Reference Documentation**. Disponível em: <https://spring.io/projects/spring-boot>. Acesso em: 4 jun. 2025.
- SPRING DATA JPA. **Spring Data JPA Documentation**. Disponível em: <https://spring.io/projects/spring-data-jpa>. Acesso em: 4 jun. 2025.
- SPRING SECURITY. **Spring Security Documentation**. Disponível em: <https://spring.io/projects/spring-security>. Acesso em: 4 jun. 2025.
- JUNIT 5. **JUnit 5 User Guide**. Disponível em: <https://junit.org/junit5/>. Acesso em: 4 jun. 2025.
- MOCKITO. **Mockito Official Documentation**. Disponível em: <https://site.mockito.org/>. Acesso em: 4 jun. 2025.
- TESTCONTAINERS. **Testcontainers Documentation**. Disponível em: <https://www.testcontainers.org/>. Acesso em: 4 jun. 2025.
- SWAGGER/OPENAPI. **Swagger Documentation**. Disponível em: <https://swagger.io/>. Acesso em: 4 jun. 2025.
- DEV.JAVA. **Learning Path**. Disponível em: <https://dev.java/learn/>. Acesso em: 4 jun. 2025.

 **Observação:** Além destas fontes formais, a elaboração deste e-book contou com a vivência prática da autora como estudante da Fuctura, assinante da Dio, e o uso da Engenharia de Prompts com o ChatGPT para a construção dinâmica deste material.

---