



Dominando o Spring Boot:

Da Arquitetura à Produção

Explorando Anotações, Integrações,
Boas Práticas e Casos Reais para
Criar APIs Modernas com Java

Ana Raquel de Holanda
Estudante Java

E-book elaborado com o suporte do ChatGPT | Ano: 2025

Da Autora para Você, Dev!

Sou uma **estudante iniciante em Programação**, dedicada ao estudo da linguagem **Java** há apenas **oito meses**. Minha formação técnica teve início na **Fuctura – Escola de Desenvolvedores**, e foi intensificada através das trilhas de aprendizagem da plataforma **DIO (Digital Innovation One)**, onde mergulhei no universo das **APIs**, compreendendo desde os fundamentos até conceitos mais avançados.

Dominei o básico com esforço, avancei para o nível intermediário com curiosidade e, utilizando o poder da **Engenharia de Prompts** aliado ao **ChatGPT**, estruturei este e-book **capítulo por capítulo**, com um objetivo claro: **ajudar outros desenvolvedores e desenvolvedoras que, assim como eu, desejam ultrapassar os limites da simples Programação Orientada a Objetos (POO)**.

Cada parágrafo aqui representa noites de estudo, testes de código, revisões incansáveis e uma vontade genuína de compartilhar conhecimento.

✨ **Que este material seja o ponto de partida da sua jornada rumo a uma carreira como Engenheiro ou Engenheira de Software profissional.**

É com prazer, dedicação e um toque de poesia que deixo aqui este convite ao próximo passo.












Sumário

♥ Da Autora para Você, Dev!.....	2
📖 Capítulo 1 – Introdução ao Spring Framework.....	8
1.1 O que é o Spring Framework?.....	8
1.2 Por que usar o Spring?.....	8
1.3 Principais Módulos do Spring Framework.....	9
1.4 Ecossistema Spring.....	9
1.5 História e Evolução do Spring.....	9
1.6 Aplicações Reais com Spring.....	9
1.7 Revisão do Capítulo.....	10
📖 Capítulo 2 – Inversão de Controle (IoC) e Injeção de Dependências.....	10
2.1 O que é Inversão de Controle (IoC)?.....	10
2.2 O que é Injeção de Dependências (DI)?.....	11
2.3 Tipos de Injeção de Dependência no Spring.....	12
2.4 Ciclo de Vida dos Beans.....	12
2.5 Exemplo Completo: Serviço de Notificação com Várias Implementações.....	12
2.6 Boas Práticas 💡.....	13
2.7 Revisão do Capítulo.....	14
📖 Capítulo 3 – Spring Core: Beans e o Container.....	14
3.1 O que é o Spring Core?.....	14
3.2 O que é um Bean?.....	14
3.3 Como o Spring cria e gerencia Beans?.....	14
Principais Containers do Spring:.....	15
3.4 Declaração de Beans no Spring.....	15
✓ 1. Anotações (Spring moderno – preferido).....	15
✓ 2. Arquivo XML (Spring legado – quase não usado hoje).....	15
3.5 Anotações Essenciais no Spring Core.....	15
📌 @Component.....	15
📌 @Service.....	15
📌 @Repository.....	16
📌 @Controller.....	16
📌 @RestController.....	16
📌 @Bean.....	16
📌 @Configuration.....	17
📌 @Autowired.....	17
📌 @Qualifier.....	17
📌 @Scope.....	17
3.6 Como o Spring Descobre os Beans?.....	18
3.7 Boas Práticas 💡.....	18
3.8 Revisão do Capítulo.....	18
📖 Capítulo 4 – Spring Boot: Simplificando a Configuração.....	19
4.1 Por que Spring Boot surgiu?.....	19
4.2 O que o Spring Boot faz por você?.....	19
4.3 Estrutura mínima de um projeto Spring Boot.....	19
4.4 A anotação mágica: @SpringBootApplication.....	19

4.5 Spring Boot Starters.....	20
4.6 Embedded Servers (Servidor embutido).....	20
4.7 Arquivo de Configuração: application.properties ou application.yml.....	21
4.8 Spring Boot DevTools.....	21
4.9 Spring Initializr.....	21
4.10 Boas Práticas 💡	21
4.11 Revisão do Capítulo.....	22
📖 Capítulo 5 – Spring Boot Starters e Auto Configuration.....	22
5.1 O que são Spring Boot Starters?.....	22
5.2 Principais Starters do Spring Boot.....	22
📌 Exemplo prático: Incluindo um Starter no Maven.....	22
5.3 O que é Auto Configuration?.....	23
📌 Como funciona por trás dos panos?.....	23
5.4 Como controlar a Auto Configuration?.....	23
✅ Habilitar (Já vem habilitado por padrão):.....	23
✅ Desabilitar Auto Configuration específica:.....	23
✅ Condicionabilidade (Auto Configuration só ativa se certas classes existirem):.....	24
5.5 Spring Boot Starter Parent (Importante para Maven).....	24
5.6 Boas Práticas 💡	24
5.7 Revisão do Capítulo.....	25
📖 Capítulo 6 – Trabalhando com Spring Data JPA.....	25
6.1 O que é o Spring Data JPA?.....	25
6.2 Conceitos fundamentais antes de usar Spring Data JPA.....	25
6.3 Configurando o Spring Data JPA no projeto.....	26
📌 Adicionando a dependência Maven:.....	26
📌 Configurando o application.properties:.....	26
6.4 Criando uma Entidade (Entity).....	26
📌 Exemplo: Classe Cliente mapeada para uma tabela no banco.....	26
6.5 Criando um Repository com Spring Data JPA.....	27
6.6 Consultas Personalizadas com Spring Data JPA.....	27
6.7 Exemplo completo de um CRUD com Spring Data JPA.....	27
6.8 Boas Práticas 💡	29
6.9 Revisão do Capítulo.....	29
📖 Capítulo 7 – Criando APIs REST com Spring MVC.....	29
7.1 O que é o Spring MVC?.....	29
7.2 Principais Anotações do Spring MVC para APIs REST.....	30
7.3 Exemplo Básico de um Controller REST.....	30
7.4 Trabalhando com JSON: Serialização e Desserialização.....	30
7.5 Testando com Postman ou Insomnia.....	31
Exemplos de Requisições:.....	31
7.6 Lidando com Respostas HTTP.....	31
7.7 Tratamento de Exceções (Global Exception Handler).....	31
7.8 Boas Práticas 💡	32
7.9 Revisão do Capítulo.....	32
📖 Capítulo 8 – Validação de Dados com Spring Validation.....	32
8.1 Por que validar dados na sua API?.....	32
8.2 O que é Bean Validation?.....	33
8.3 Dependência necessária (caso esteja criando um projeto manualmente).....	33
8.4 Principais Anotações de Validação.....	33

8.5 Exemplo Prático de Entidade com Validações.....	33
8.6 Como ativar a validação no Controller?.....	34
8.7 Exemplo de Resposta de Erro de Validação.....	34
8.8 Tratamento Global de Erros de Validação.....	35
8.9 Validação em Update (PUT) também funciona.....	35
8.10 Outras validações avançadas (Validação personalizada).....	35
8.11 Boas Práticas 💡	35
8.12 Revisão do Capítulo.....	36
📖 Capítulo 9 – Segurança de API com Spring Security.....	36
9.1 O que é o Spring Security?.....	36
9.2 Principais conceitos de segurança.....	36
9.3 Adicionando o Spring Security no projeto.....	37
9.4 Segurança Padrão com Usuário em Memória.....	37
9.5 Criando um usuário personalizado (InMemoryUserDetailsManager).....	37
9.6 Personalizando Regras de Autorização.....	38
9.7 Codificação de Senhas com PasswordEncoder.....	38
9.8 Implementando Autenticação com Banco de Dados (UserDetailsService Custom).....	39
9.9 Protegendo Endpoints de API REST.....	39
9.10 Boas Práticas 💡	39
9.11 Revisão do Capítulo.....	39
📖 Capítulo 10 – Documentação de APIs REST com Springdoc OpenAPI.....	40
10.1 Por que documentar uma API REST?.....	40
10.2 O que é o Springdoc OpenAPI?.....	40
10.3 Adicionando o Springdoc OpenAPI ao projeto.....	40
📌 Dependência Maven:.....	40
10.4 Acessando o Swagger UI.....	40
10.5 Personalizando a documentação com anotações.....	41
📌 Exemplo de Controller documentado:.....	41
10.6 Exemplo de Resposta no Swagger UI.....	41
GET /clientes/{id}.....	41
10.7 Configuração adicional (se necessário).....	42
10.8 Boas Práticas 💡	42
10.9 Benefícios profissionais de usar o Springdoc OpenAPI.....	42
10.10 Revisão do Capítulo.....	42
📖 Capítulo 11 – Testes Unitários e de Integração com Spring Boot Test.....	43
11.1 Por que testar aplicações Spring Boot?.....	43
11.2 Principais tipos de testes no Spring Boot.....	43
11.3 Dependências para Testes no Spring Boot.....	43
11.4 Estrutura típica de um projeto com testes.....	44
11.5 Exemplo de Teste Unitário com JUnit e Mockito (Service Layer).....	44
📌 Service a ser testado:.....	44
📌 Teste Unitário:.....	44
11.6 Exemplo de Teste de Integração com Spring Boot.....	45
📌 Controller de exemplo:.....	45
📌 Teste de Integração usando MockMvc:.....	45
11.7 Testando Repositories com H2 Database (Teste com Banco).....	46
11.8 Mockando Dependências com Mockito.....	46
11.9 Boas Práticas 💡	47
11.10 Revisão do Capítulo.....	47
📖 Capítulo 12 – Boas Práticas de Projeto com Spring Boot.....	47


12.1 Por que seguir boas práticas?.....	47
12.2 Estrutura de pacotes recomendada.....	48
12.3 Aplicando o princípio de responsabilidade única (SRP).....	48
12.4 Boas Práticas com DTOs.....	48
12.5 Uso correto de Exceptions.....	49
12.6 Boas Práticas para Repositories.....	49
12.7 Boas Práticas para Services.....	49
12.8 Uso de Logs ao invés de System.out.println.....	50
12.9 Boas Práticas com Testes.....	50
12.10 Anotações Spring Boot: Quando usar cada uma?.....	50
12.11 Cuidados com o uso de @Autowired.....	50
12.12 Clean Code aplicado ao Spring Boot 💡	51
12.13 Revisão do Capítulo.....	51
📖 Capítulo 13 – Integração com APIs Externas: RestTemplate e WebClient.....	51
13.1 Por que integrar sua API com serviços externos?.....	51
13.2 Duas formas principais de integrar APIs REST no Spring Boot.....	52
13.3 Integração usando RestTemplate (modo clássico).....	52
13.3.1 Criando um Bean de RestTemplate:.....	52
13.3.2 Exemplo de Consumo com RestTemplate:.....	52
13.4 Integração usando WebClient (modo moderno e não bloqueante).....	53
13.4.1 Criando um Bean de WebClient:.....	53
13.4.2 Exemplo de Consumo com WebClient:.....	53
13.5 Testando integração com serviços externos.....	53
13.6 Tratamento de Erros nas Chamadas HTTP.....	54
Exemplo com RestTemplate:.....	54
Exemplo com WebClient:.....	54
13.7 Quando usar RestTemplate vs WebClient?.....	54
13.8 Revisão do Capítulo.....	54
📖 Capítulo 14 – Cache com Spring Boot.....	55
14.1 O que é Cache?.....	55
14.2 Cache no Spring Boot com Spring Cache.....	55
14.3 Configurando o Spring Cache.....	55
🔗 Dependência Maven:.....	55
14.4 Habilitando o Cache no projeto.....	55
14.5 Cache básico com @Cacheable.....	56
Exemplo:.....	56
14.6 Limpando o Cache: @CacheEvict.....	56
14.7 Atualizando o Cache: @CachePut.....	57
14.8 Cache em Métodos com Parâmetros.....	57
14.9 Implementações de Cache Suportadas pelo Spring Boot.....	57
14.10 Exemplo: Cache com Caffeine (Cache de Alta Performance).....	57
14.11 Cuidados com o uso de Cache 💡	58
14.12 Revisão do Capítulo.....	58
📖 Capítulo 15 – Mensageria com Spring Boot e RabbitMQ.....	59
15.1 O que é Mensageria?.....	59
15.2 O que é o RabbitMQ?.....	59
15.3 Dependência Maven para RabbitMQ no Spring Boot.....	59
15.4 Configurando o RabbitMQ no Spring Boot.....	59
15.5 Publicando Mensagens com RabbitTemplate.....	60

15.6 Consumindo Mensagens com @RabbitListener.....	60
15.7 Fluxo de Comunicação.....	61
15.8 Boas Práticas com Mensageria.....	61
15.9 Exemplo de uso prático no mundo real:.....	61
15.10 Revisão do Capítulo.....	61
 Capítulo 16 – Configuração Externa e Profiles no Spring Boot.....	62
16.1 Por que externalizar configurações?.....	62
16.2 Principais formatos de configuração no Spring Boot.....	62
16.3 Exemplo com application.properties.....	62
16.4 Exemplo com application.yml (YAML).....	63
16.5 Criando múltiplos profiles: dev, test, prod.....	63
16.6 Ativando profiles via linha de comando.....	63
Exemplo:.....	63
16.7 Ativando profiles via application.properties principal.....	63
16.8 Injetando valores de configuração com @Value.....	64
16.9 Boas práticas com Configurações.....	64
16.10 Trabalhando com @ConfigurationProperties.....	64
application.properties:.....	64
Classe Java:.....	64
16.11 Revisão do Capítulo.....	65
 Capítulo 17 – Estudo de Caso: Construindo uma API REST Completa com Spring Boot.....	65
17.1 Objetivo do Estudo de Caso.....	65
 Projeto: Cadastro de Clientes com Spring Boot.....	65
17.2 Estrutura do Projeto.....	66
17.3 Entidade Cliente (Model).....	66
17.4 DTO de Cliente (ClienteDTO).....	66
17.5 Repository.....	67
17.6 Service com Regras de Negócio.....	67
17.7 Controller REST.....	67
17.8 Configuração de Profile para Ambiente de Desenvolvimento.....	68
17.9 Validação com Bean Validation.....	68
17.10 Exemplo de Teste de Endpoint com cURL.....	69
17.11 Possíveis Extensões Futuras.....	69
 Capítulo 17.12 – Visão Arquitetural: Spring Framework Architecture.....	69
Objetivo:.....	69
 Explicação Rápida da Arquitetura – Estudo de Caso.....	70
 Dicas para Projetos Reais:.....	71
 Como usar esse Diagrama?.....	71
17.13 Conclusão do Estudo de Caso.....	71
 Conclusão Final – Obrigada por Chegar Até Aqui, Desenvolvedores! 	72
 Uma Jornada que Está Apenas Começando.....	72
 Bibliografia.....	72

Capítulo 1 – Introdução ao Spring Framework

1.1 O que é o Spring Framework?

O **Spring Framework** é um poderoso framework open-source para desenvolvimento de aplicações Java corporativas. Ele fornece uma infraestrutura abrangente para criar aplicações robustas, modulares e com baixa dependência entre componentes.

 **Dica:** O Spring é conhecido pela sua capacidade de reduzir a complexidade do desenvolvimento Java empresarial, através de conceitos como **Inversão de Controle (IoC)** e **Injeção de Dependências**.

1.2 Por que usar o Spring?

- ✓ Reduz a quantidade de código boilerplate.
 - ✓ Facilita a integração com diversos frameworks e bibliotecas.
 - ✓ Promove boas práticas de desenvolvimento orientado a interfaces.
 - ✓ Permite construção de aplicações escaláveis e testáveis.
 - ✓ Ampla comunidade e excelente documentação oficial.
-

1.3 Principais Módulos do Spring Framework

O Spring Framework é modular, o que significa que você pode utilizar apenas os componentes necessários. Os principais módulos são:

Módulo	Descrição
Spring Core	Fornece as fundações do framework (IoC e DI).
Spring AOP	Suporte para programação orientada a aspectos.
Spring Data	Simplificação de acesso a dados e repositórios.
Spring MVC	Estruturação de aplicações web seguindo o padrão MVC.
Spring Boot	Facilita a configuração, desenvolvimento e deploy de aplicações Spring.
Spring Security	Gerenciamento de autenticação, autorização e segurança em APIs.
Spring Test	Ferramentas para facilitar o teste de componentes Spring.

1.4 Ecossistema Spring

Além do Spring Framework base, o **ecossistema Spring** inclui projetos complementares como:

- **Spring Boot**

- **Spring Data JPA**
- **Spring Security**
- **Spring Cloud**
- **Spring Batch**

 **Dica:** A combinação de **Spring Framework** + **Spring Boot** é hoje o padrão de mercado para construir microsserviços, APIs REST e aplicações back-end robustas.

1.5 História e Evolução do Spring

O Spring surgiu no início dos anos 2000 como uma alternativa leve ao modelo pesado de desenvolvimento Java EE. Com o passar dos anos, o Spring evoluiu e hoje é um dos frameworks Java mais usados no mundo.

- Primeira versão: 2004
 - Marco revolucionário: **Lançamento do Spring Boot (2014)**
 - Atualizações constantes com foco em desempenho, simplicidade e integração com cloud.
-

1.6 Aplicações Reais com Spring

Empresas como Netflix, Amazon e Google utilizam Spring em larga escala. Ele é amplamente usado para:

- APIs REST
 - Microsserviços
 - Integração com bancos de dados
 - Sistemas corporativos complexos
 - Plataformas de e-commerce
-

1.7 Revisão do Capítulo

Neste capítulo, você aprendeu:

- ✓ O que é o Spring Framework.
- ✓ Por que ele é tão utilizado na indústria.
- ✓ Os principais módulos e projetos do ecossistema Spring.
- ✓ Um panorama da história e evolução do Spring.

Capítulo 2 – Inversão de Controle (IoC) e Injeção de Dependências

2.1 O que é Inversão de Controle (IoC)?

Inversão de Controle (IoC) é um princípio da engenharia de software que inverte a forma tradicional de criação e gerenciamento de objetos.

👉 **Antes (Programador controla o fluxo):**

```
java
CopiarEditar
ServicoEmail servico = new ServicoEmail();
```

👉 **Depois (O Spring controla e injeta os objetos):**

```
java
CopiarEditar
@Autowired
private ServicoEmail servico;
```

💡 **Conceito chave:** No Spring, o **container** é responsável por criar, gerenciar e entregar os objetos que a sua aplicação precisa.

2.2 O que é Injeção de Dependências (DI)?

Injeção de Dependências (DI) é o mecanismo que implementa a IoC no Spring. O Spring injeta automaticamente as dependências necessárias no seu código.

Exemplo simples:

📌 Classe de serviço:

```
java
CopiarEditar
public class ServicoEmail {
    public void enviarEmail(String mensagem) {
        System.out.println("Enviando email: " + mensagem);
    }
}
```

📌 Classe que depende do serviço:

```
java
CopiarEditar
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class NotificacaoService {
```

```

@Autowired
private ServicoEmail servicoEmail;

public void notificar(String mensagem) {
    servicoEmail.enviarEmail(mensagem);
}
}

```

🔴 Classe principal com Spring Boot:

```

java
CopiarEditar
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class MeuAppSpring {

    public static void main(String[] args) {
        SpringApplication.run(MeuAppSpring.class, args);
    }

    @Bean
    public CommandLineRunner demo(ApplicationContext ctx) {
        return args -> {
            NotificacaoService notificacaoService =
ctx.getBean(NotificacaoService.class);
            notificacaoService.notificar("Bem-vindo ao Spring!");
        };
    }
}

```

2.3 Tipos de Injeção de Dependência no Spring

Tipo	Exemplo	Quando usar
Injeção por campo (field injection)	@Autowired private ServicoEmail servico;	Rápido, mas menos testável
Injeção por construtor (constructor injection)	public NotificacaoService(ServicoEmail servico) {...}	Melhor para testes unitários ✅
Injeção por setter (setter injection)	public void setServicoEmail(ServicoEmail servico) {...}	Quando a dependência é opcional

2.4 Ciclo de Vida dos Beans

O Spring controla o ciclo de vida dos objetos, chamados de **Beans**, que são instâncias gerenciadas pelo container.

📌 Exemplos de escopos de Bean:

- **Singleton (default):** Um único bean por aplicação.
- **Prototype:** Um novo bean a cada requisição.
- **Request/Session:** Escopos para aplicações web.

💡 **Exemplo de configuração de escopo:**

```
java
CopiarEditar
@Component
@Scope("prototype")
public class MinhaClasse {
    // Cada chamada gera uma nova instância
}
```

2.5 Exemplo Completo: Serviço de Notificação com Várias Implementações

📌 Interface de notificação:

```
java
CopiarEditar
public interface Notificador {
    void notificar(String mensagem);
}
```

📌 Implementação com E-mail:

```
java
CopiarEditar
import org.springframework.stereotype.Component;

@Component
public class EmailNotificador implements Notificador {
    public void notificar(String mensagem) {
        System.out.println("Email enviado: " + mensagem);
    }
}
```

📌 Implementação com SMS:

```
java
CopiarEditar
import org.springframework.stereotype.Component;

@Component
public class SmsNotificador implements Notificador {
    public void notificar(String mensagem) {
        System.out.println("SMS enviado: " + mensagem);
    }
}
```

🔗 Serviço que usa a interface:

```
java
CopiarEditar
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class AlertaService {

    private Notificador notificador;

    @Autowired
    public AlertaService(Notificador notificador) {
        this.notificador = notificador;
    }

    public void enviarAlerta(String msg) {
        notificador.notificar(msg);
    }
}
```

🔗 Testando na aplicação principal:

```
java
CopiarEditar
@Bean
public CommandLineRunner run(AlertaService alertaService) {
    return args -> alertaService.enviarAlerta("Sistema fora do ar!");
}
```

2.6 Boas Práticas 💡

- ✅ **Prefira injeção por construtor:** facilita testes unitários e torna as dependências explícitas.
 - ✅ **Evite usar @Autowired em campos privados sem necessidade de testes.**
 - ✅ Utilize interfaces para facilitar a substituição de implementações futuras.
 - ✅ Documente suas classes com JavaDoc explicando suas responsabilidades.
 - ✅ Mantenha a separação entre **configuração, negócio e infraestrutura**.
-

2.7 Revisão do Capítulo


Neste capítulo, você aprendeu:

- ✅ O que é Inversão de Controle (IoC) e Injeção de Dependências (DI).
- ✅ Como o Spring faz a mágica de gerenciar objetos para você.
- ✅ Exemplos práticos de DI com **field, setter e constructor injection**.
- ✅ Como controlar o ciclo de vida dos beans com escopos.
- ✅ Boas práticas para melhorar desempenho e testabilidade.

Capítulo 3 – Spring Core: Beans e o Container

3.1 O que é o Spring Core?


O **Spring Core** é o núcleo do Spring Framework. Ele fornece o suporte fundamental para **Inversão de Controle (IoC)** e **Injeção de Dependências (DI)**, além de gerenciar o ciclo de vida dos objetos através do **Spring Container**.

 **Dica:** O Spring Core é o coração de toda aplicação Spring. Todo o resto (Spring Boot, Spring Data, Spring MVC) depende dos conceitos que surgem aqui.

3.2 O que é um Bean?

No Spring, um **Bean** é qualquer objeto Java que é instanciado, gerenciado e controlado pelo container Spring.

Definição prática:

 **Bean = Classe Java gerenciada pelo Spring.**

Exemplos típicos de Beans:

- Serviços de negócio
 - Repositórios de dados
 - Controladores de API
 - Configurações
-

3.3 Como o Spring cria e gerencia Beans?

O **Spring Container** é o mecanismo responsável por:

- ✓ Criar os beans
- ✓ Injetar as dependências
- ✓ Gerenciar os ciclos de vida
- ✓ Aplicar escopos e configurações

Principais Containers do Spring:

Container	Função
BeanFactory	Container básico e mais leve
ApplicationContext	Container mais robusto e com mais recursos (é o mais usado em projetos modernos)

 **Dica:** Hoje, praticamente **todas as aplicações Spring Boot** usam **ApplicationContext**.


3.4 Declaração de Beans no Spring

Existem duas formas principais de registrar Beans no Spring:

- ✓ **1. Anotações (Spring moderno – preferido)**
 - ✓ **2. Arquivo XML (Spring legado – quase não usado hoje)**
-


3.5 Anotações Essenciais no Spring Core

@Component

 **O que faz:** Marca uma classe como um Bean genérico para o Spring gerenciar.

```
@Component
public class MeuServico {
    public void executar() {
        System.out.println("Executando serviço...");
    }
}
```


@Service

 **O que faz:** Especialização de **@Component**, usada para classes de serviço da camada de negócio.

```
@Service
public class CalculadoraService {
    public int somar(int a, int b) {
        return a + b;
    }
}
```

 **Boas práticas:** Use **@Service** para dar semântica ao seu código.

@Repository

 **O que faz:** Outra especialização de **@Component**, usada para classes que acessam o banco de dados.

```
@Repository
public class ClienteRepository {
    // Métodos de acesso ao banco
}
```

💡 **Detalhe:** O Spring adiciona automaticamente o tratamento de exceções para operações de persistência com esta anotação.

📌 @Controller

👉 **O que faz:** Indica que a classe é um controlador Spring MVC.

```
@Controller
public class ClienteController {
    // Métodos que recebem requisições HTTP
}
```

📌 @RestController

👉 **O que faz:** Combina @Controller + @ResponseBody.

Usado em APIs REST para retornar objetos JSON diretamente.

```
@RestController
public class ApiController {
    @GetMapping("/hello")
    public String hello() {
        return "Olá Spring Boot!";
    }
}
```

📌 @Bean

👉 **O que faz:** Registra manualmente um Bean dentro de uma classe de configuração.

📌 **Exemplo de classe de configuração:**

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {

    @Bean
    public MeuServico meuServico() {
        return new MeuServico();
    }
}
```


💡 **Quando usar @Bean:**

Quando você quer criar e controlar manualmente a instância de um objeto, ou quando a classe vem de uma biblioteca externa que não tem anotações Spring.


@Configuration

 **O que faz:** Informa ao Spring que a classe contém métodos de configuração (normalmente com métodos anotados com @Bean).

```
@Configuration
public class MinhaConfiguracao {
    // Métodos @Bean
}
```

 **Dica:** Sempre que você precisar registrar Beans manualmente, a classe precisa ser anotada com @Configuration.

@Autowired

 **O que faz:** Faz o Spring injetar automaticamente uma dependência.


```
@Component
public class PedidoService {

    @Autowired
    private ClienteRepository clienteRepository;

    // Uso de clienteRepository dentro da classe
}
```


 **Onde pode ser usada:** Em atributos, construtores ou métodos setters.

@Qualifier

 **O que faz:** Especifica qual Bean o Spring deve injetar quando há múltiplas implementações do mesmo tipo.

```
@Autowired
@Qualifier("emailNotificador")
private Notificador notificador;
```

@Scope

 **O que faz:** Define o escopo de um Bean (singleton, prototype, request, session, etc).

```
@Component
@Scope("prototype")
public class MeuBeanDinamico {
    // Um novo Bean a cada requisição
}
```

3.6 Como o Spring Descobre os Beans?

Por padrão, o Spring faz um **Component Scan** para buscar todas as classes anotadas com `@Component`, `@Service`, `@Repository`, etc.

📌 Exemplo:

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

👉 A anotação `@SpringBootApplication` inclui automaticamente o **Component Scan** para o pacote atual e seus subpacotes.

3.7 Boas Práticas 💡

- ✅ Prefira usar anotações ao invés de XML (mais moderno e legível).
 - ✅ Use `@Component`, `@Service`, `@Repository` e `@Controller` de forma semântica.
 - ✅ Evite usar o mesmo tipo de Bean com o mesmo nome para não confundir o container.
 - ✅ Use `@Qualifier` sempre que houver mais de uma implementação da mesma interface.
-

3.8 Revisão do Capítulo

Neste capítulo, você aprendeu:

- ✅ O que é o Spring Core e seu papel na arquitetura.
 - ✅ Como o Spring cria e gerencia Beans.
 - ✅ O funcionamento de cada anotação fundamental (`@Component`, `@Service`, `@Bean`, etc).
 - ✅ Como o Spring realiza o Component Scan.
 - ✅ As melhores práticas de organização de Beans.
-

💡 **Próximo Passo:** No próximo capítulo, vamos explorar **Spring Boot: Simplificando a Configuração**, onde veremos como o Spring Boot revolucionou o desenvolvimento Spring.

Capítulo 4 – Spring Boot: Simplificando a Configuração

4.1 Por que Spring Boot surgiu?

O **Spring Framework**, apesar de poderoso, tinha um problema nas suas versões iniciais: exigia **muita configuração manual**. Era necessário criar arquivos XML extensos, definir beans manualmente e configurar servidores de aplicação.

O **Spring Boot** nasceu com um objetivo claro:

- ✓ **Eliminar a configuração excessiva (boilerplate)**
- ✓ **Simplificar o desenvolvimento**
- ✓ **Acelerar o time-to-market de aplicações Java modernas**

 **Dica:** O Spring Boot segue o conceito de "**convenção sobre configuração**" (Convention over Configuration).

4.2 O que o Spring Boot faz por você?

- ✓ Configuração automática de Beans
 - ✓ Embedded servers (Tomcat, Jetty, Undertow)
 - ✓ Dependências centralizadas via "**Spring Boot Starters**"
 - ✓ Monitoramento com Actuator
 - ✓ Exposição de endpoints REST de forma rápida
 - ✓ Profiles para configurações de ambientes diferentes (dev, test, prod)
-

4.3 Estrutura mínima de um projeto Spring Boot

 Estrutura típica de diretórios:

```
src
├── main
│   ├── java
│   │   └── com.exemplo.meuprojeto
│   │       └── MinhaAplicacao.java
│   └── resources
│       ├── application.properties
│       └── static / templates / etc
```

4.4 A anotação mágica: @SpringBootApplication

 **O que faz:**

É uma anotação de alto nível que combina:

- `@Configuration` → Declara uma classe de configuração
- `@EnableAutoConfiguration` → Ativa a configuração automática do Spring Boot
- `@ComponentScan` → Faz o Spring escanear pacotes para encontrar Beans

🔗 Exemplo:

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

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

💡 **Dica:** A classe com `@SpringBootApplication` deve ficar no nível mais alto do seu pacote base para que o Component Scan encontre todos os componentes.

4.5 Spring Boot Starters

Starters são pacotes prontos que incluem todas as dependências necessárias para uma funcionalidade.

Starter	Para que serve
spring-boot-starter-web	Criar APIs REST
spring-boot-starter-data-jpa	Integração com banco de dados via JPA/Hibernate
spring-boot-starter-security	Segurança e autenticação
spring-boot-starter-test	Testes com JUnit e Mockito

🔗 Exemplo de dependência Maven:

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

4.6 Embedded Servers (Servidor embutido)

Com Spring Boot, **você não precisa mais instalar e configurar o Tomcat separadamente.**

✓ O servidor web já vem embutido no JAR executável.

🔗 Para rodar sua aplicação:

```
mvn spring-boot:run
```

Ou simplesmente:

```
java -jar target/minha-aplicacao.jar
```

4.7 Arquivo de Configuração: `application.properties` ou `application.yml`

Este arquivo controla configurações do projeto.

📌 Exemplo de configurações:

```
server.port=8081
spring.datasource.url=jdbc:mysql://localhost:3306/meubanco
spring.jpa.hibernate.ddl-auto=update
```

💡 **Dica:** Para projetos maiores, prefira o formato YAML (`application.yml`) pela melhor organização visual.

4.8 Spring Boot DevTools

Ferramenta opcional que permite:

- ✓ Reload automático da aplicação durante o desenvolvimento
- ✓ Cache de templates desativado para ver alterações imediatas
- ✓ Experiência de desenvolvimento mais ágil

📌 Adicionando ao Maven:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <scope>runtime</scope>
</dependency>
```

4.9 Spring Initializr

👉 Ferramenta oficial para gerar projetos Spring Boot rapidamente:

🌐 Site: <https://start.spring.io/>

✓ Escolha versão, dependências e gere o projeto pronto para importar no Eclipse, IntelliJ ou VS Code.

4.10 Boas Práticas 💡


- ✓ Centralize configurações sensíveis em arquivos externos (ou use variáveis de ambiente).
- ✓ Sempre defina o **profile ativo** (exemplo: dev, test, prod).

- ✓ Mantenha o `application.properties` ou `application.yml` bem organizado.
 - ✓ Utilize **Spring Boot Starters** para reduzir o risco de conflitos de dependências.
-

4.11 Revisão do Capítulo

Neste capítulo, você aprendeu:

- ✓ Por que o Spring Boot revolucionou o desenvolvimento Java.
 - ✓ Como criar uma aplicação mínima com Spring Boot.
 - ✓ O papel de `@SpringBootApplication`.
 - ✓ Como usar Starters, embedded servers e configurar o `application.properties`.
 - ✓ Ferramentas como **DevTools** e **Spring Initializr** para agilizar seu desenvolvimento.
-

 **Próximo Passo:** No próximo capítulo, vamos explorar os detalhes dos **Spring Boot Starters** e **Auto Configuration**, aprofundando o controle sobre o comportamento da sua aplicação.

Capítulo 5 – Spring Boot Starters e Auto Configuration

5.1 O que são Spring Boot Starters?

Os **Starters** do Spring Boot são **pacotes de dependências pré-configurados** que facilitam a inclusão de tecnologias específicas em seu projeto.

 **Objetivo:**


- 👉 Reduzir a complexidade da configuração de bibliotecas externas.
 - 👉 Fornecer um conjunto de dependências testadas e compatíveis entre si.
-

5.2 Principais Starters do Spring Boot

Starter	Finalidade
<code>spring-boot-starter-web</code>	Desenvolvimento de aplicações web e APIs REST
<code>spring-boot-starter-data-jpa</code>	Integração com bancos de dados usando JPA/Hibernate
<code>spring-boot-starter-thymeleaf</code>	Desenvolvimento de páginas web com Thymeleaf
<code>spring-boot-starter-security</code>	Autenticação e autorização de aplicações
<code>spring-boot-starter-test</code>	Inclui JUnit, Mockito e outras ferramentas de teste


Exemplo prático: Incluindo um Starter no Maven

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

 **Dica:** Sempre use apenas os Starters realmente necessários, evitando dependências desnecessárias.

5.3 O que é Auto Configuration?

Auto Configuration (Configuração Automática) é o mecanismo que permite ao Spring Boot **configurar automaticamente os Beans, DataSources, Servlets e outras dependências** com base nas bibliotecas que estão no classpath.

 Exemplo:

Se você inclui o **starter-web**, o Spring Boot automaticamente:

- ✓ Configura o Tomcat embutido
 - ✓ Cria um `DispatcherServlet`
 - ✓ Define configurações básicas de JSON, Jackson, etc.
-

Como funciona por trás dos panos?

O Spring Boot usa o arquivo:

`META-INF/spring.factories`

Onde ele registra quais classes de configuração devem ser carregadas de forma automática.

Exemplo interno do Boot:

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration
```

 **Você não precisa decorar isso, apenas entender que o Spring Boot faz essa magia para você.**

5.4 Como controlar a Auto Configuration?

✓ Habilitar (Já vem habilitado por padrão):

`@SpringBootApplication`

Como essa anotação contém internamente:

`@EnableAutoConfiguration`

✓ Desabilitar Auto Configuration específica:

Suponha que você queira desabilitar a configuração automática de JPA:

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

✓ Condicionabilidade (Auto Configuration só ativa se certas classes existirem):

O Spring Boot usa anotações internas como:

- `@ConditionalOnClass`
- `@ConditionalOnMissingBean`
- `@ConditionalOnProperty`

👉 Exemplo de uso interno do Spring Boot:

Se você adiciona o Spring Web, o Boot só ativa a configuração web **se detectar o `DispatcherServlet` no classpath**.

5.5 Spring Boot Starter Parent (Importante para Maven)

É comum que o `pom.xml` use o **Starter Parent**:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.2.0</version>
</parent>
```

✓ O que ele faz?

- Gerencia versões de todas as dependências

- Define configurações de build padrão
 - Inclui configurações de plugins do Maven
-


5.6 Boas Práticas

- ✓ Use **Starters** sempre que possível para garantir dependências testadas.
 - ✓ Só adicione Starters necessários (evite inflar o projeto com dependências desnecessárias).
 - ✓ Conheça os Starters mais usados (Web, Data JPA, Security, Validation e Test).
 - ✓ Se precisar, **desative Auto Configurations** que não são úteis para seu projeto.
-

5.7 Revisão do Capítulo

Neste capítulo, você aprendeu:

- ✓ O que são os **Spring Boot Starters**.
 - ✓ Como eles reduzem a complexidade da configuração de dependências.
 - ✓ O conceito de **Auto Configuration** e como o Spring Boot configura o projeto automaticamente.
 - ✓ Como controlar e até desativar configurações automáticas quando necessário.
 - ✓ O papel do **Spring Boot Starter Parent** no gerenciamento de dependências Maven.
-

 **Próximo Passo:** No próximo capítulo, vamos explorar **Spring Data JPA**, onde você aprenderá como mapear entidades Java para tabelas no banco de dados de forma prática e eficiente.

Capítulo 6 – Trabalhando com Spring Data JPA

6.1 O que é o Spring Data JPA?

Spring Data JPA é um projeto da família Spring que facilita o acesso a bancos de dados relacionais através da **Java Persistence API (JPA)**, eliminando a necessidade de escrever SQL repetitivo.

 **Dica:** Ele permite que você **crie repositórios com poucos códigos**, usando apenas interfaces e convenções de nomenclatura.

6.2 Conceitos fundamentais antes de usar Spring Data JPA

Antes de começar, é importante entender:

- ✓ **JPA (Java Persistence API):** Especificação Java para mapeamento objeto-relacional (ORM).
 - ✓ **Hibernate:** Uma das implementações mais populares de JPA.
 - ✓ **Entidade (Entity):** Classe Java que representa uma tabela no banco de dados.
 - ✓ **Repository:** Interface onde você define as operações de acesso aos dados.
-

6.3 Configurando o Spring Data JPA no projeto

Adicionando a dependência Maven:

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

<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
</dependency>
```

(Exemplo com MySQL, mas pode ser usado com PostgreSQL, H2, etc.)

Configurando o `application.properties`:

```
spring.datasource.url=jdbc:mysql://localhost:3306/meubanco
spring.datasource.username=root
spring.datasource.password=senha

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

Explicação de cada propriedade:

- **ddl-auto=update** → Atualiza o schema automaticamente
 - **show-sql=true** → Mostra os comandos SQL executados
 - **format_sql=true** → Formata o SQL para melhor leitura no console
-

6.4 Criando uma Entidade (Entity)

Exemplo: Classe Cliente mapeada para uma tabela no banco

```
import jakarta.persistence.Entity;
```

```
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Cliente {

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

    private String nome;
    private String email;

    // Getters e Setters
}
```

💡 Explicação das anotações:

Anotação	Função
@Entity	Indica que a classe é uma entidade JPA
@Id	Identifica o campo chave primária
@GeneratedValue	Controla a geração automática do ID

6.5 Criando um Repository com Spring Data JPA

Você **não precisa escrever nenhuma implementação!** Basta criar uma interface:

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface ClienteRepository extends JpaRepository<Cliente, Long> {
    // Métodos CRUD já disponíveis automaticamente
}
```

💡 Spring Data JPA já fornece métodos prontos:

Método	Descrição
findAll()	Retorna todos os registros
findById(Long id)	Busca por ID
save(Cliente cliente)	Salva ou atualiza
deleteById(Long id)	Remove por ID

6.6 Consultas Personalizadas com Spring Data JPA


Você pode criar **métodos de consulta apenas declarando os nomes de forma estratégica:**

```
List<Cliente> findByNome(String nome);
List<Cliente> findByEmailContaining(String email);
```


Exemplos de consulta possíveis com apenas métodos de interface:

- `findByNomeLike(String nome)`

- `findByNomeOrEmail(String nome, String email)`
- `findByNomeStartingWith(String prefixo)`

 **Dica:** Consulte a documentação oficial para a lista completa de convenções de query.

6.7 Exemplo completo de um CRUD com Spring Data JPA

 Exemplo de Service:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class ClienteService {

    @Autowired
    private ClienteRepository clienteRepository;

    public List<Cliente> listarTodos() {
        return clienteRepository.findAll();
    }

    public Cliente salvar(Cliente cliente) {
        return clienteRepository.save(cliente);
    }

    public void deletar(Long id) {
        clienteRepository.deleteById(id);
    }

    public Cliente buscarPorId(Long id) {
        return clienteRepository.findById(id).orElse(null);
    }
}
```

 Exemplo de Controller REST:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

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

    @Autowired
    private ClienteService clienteService;

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



```
@PostMapping
public Cliente criar(@RequestBody Cliente cliente) {
    return clienteService.salvar(cliente);
}

@GetMapping("/{id}")
public Cliente buscarPorId(@PathVariable Long id) {
    return clienteService.buscarPorId(id);
}

@DeleteMapping("/{id}")
public void deletar(@PathVariable Long id) {
    clienteService.deletar(id);
}
}
```


6.8 Boas Práticas

- ✓ Crie uma camada de serviço entre Controller e Repository (não chame o Repository diretamente do Controller em aplicações reais).
 - ✓ Valide os dados antes de persistir (veremos isso no capítulo de validação).
 - ✓ Use DTOs (Data Transfer Objects) para evitar expor entidades diretamente nas APIs (prática comum em aplicações profissionais).
 - ✓ Controle as transações de forma declarativa (com `@Transactional`, que veremos nos próximos capítulos).
-

6.9 Revisão do Capítulo

Neste capítulo, você aprendeu:

- ✓ O que é o Spring Data JPA e sua relação com o JPA e o Hibernate.
 - ✓ Como configurar o acesso ao banco de dados no Spring Boot.
 - ✓ Como criar entidades e repositórios com mínimo de código.
 - ✓ Como realizar consultas personalizadas apenas declarando métodos em interfaces.
 - ✓ Um exemplo completo de CRUD usando Service, Repository e Controller.
-

 **Próximo Passo:** No próximo capítulo, vamos começar a desenvolver **APIs REST com Spring MVC**, criando endpoints que consomem e produzem JSON.

Capítulo 7 – Criando APIs REST com Spring MVC

7.1 O que é o Spring MVC?

Spring MVC é o módulo do Spring responsável por criar **aplicações web** e **APIs REST**.

 **MVC significa:**

Model – View – Controller, um padrão arquitetural onde:

Camada	Função
Model	Representa os dados e a lógica de negócio
View	Representa a interface de usuário (em APIs REST, geralmente não usada)
Controller	Camada que recebe requisições, processa e retorna respostas

No caso de APIs REST, a **View** é substituída por respostas em **JSON** ou **XML**.

7.2 Principais Anotações do Spring MVC para APIs REST

Anotação	Função
<code>@RestController</code>	Define a classe como um Controller REST
<code>@RequestMapping</code>	Define o caminho base das requisições
<code>@GetMapping</code> , <code>@PostMapping</code> , <code>@PutMapping</code> , <code>@DeleteMapping</code>	Mapeiam métodos HTTP
<code>@PathVariable</code>	Captura variáveis de rota
<code>@RequestBody</code>	Lê o corpo da requisição (ex: JSON)
<code>@ResponseBody</code>	Indica que o retorno será serializado (já incluído em <code>@RestController</code>)

7.3 Exemplo Básico de um Controller REST

```
import org.springframework.web.bind.annotation.*;

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

    @GetMapping
    public String listar() {
        return "Listando todos os clientes";
    }

    @PostMapping
    public String criar(@RequestBody String cliente) {
        return "Criando o cliente: " + cliente;
    }
}
```

```

    @GetMapping("/{id}")
    public String buscar(@PathVariable Long id) {
        return "Buscando cliente com ID: " + id;
    }

    @DeleteMapping("/{id}")
    public String deletar(@PathVariable Long id) {
        return "Deletando cliente com ID: " + id;
    }
}

```

7.4 Trabalhando com JSON: Serialização e Desserialização

Quando o Spring Boot detecta a presença do **Jackson** (uma biblioteca JSON), ele automaticamente converte objetos Java em JSON e vice-versa.

🔗 Exemplo de Entidade Cliente sendo retornada:

```

@GetMapping("/{id}")
public Cliente buscarPorId(@PathVariable Long id) {
    return clienteService.buscarPorId(id);
}

```

👉 Resultado no navegador ou Postman:

```

{
  "id": 1,
  "nome": "Ana",
  "email": "ana@example.com"
}

```

7.5 Testando com Postman ou Insomnia

Exemplos de Requisições:

Método	Endpoint	Corpo da Requisição
GET	/clientes	-
POST	/clientes	{"nome": "Ana", "email": "ana@example.com"}
GET	/clientes/1	-
DELETE	/clientes/1	-

💡 **Dica:** Instale e utilize o **Postman** ou o **Insomnia** para testar suas APIs de forma visual e rápida.

7.6 Lidando com Respostas HTTP

Por padrão, o Spring retorna **HTTP Status 200 OK**, mas podemos personalizar:

🔗 Exemplo com ResponseEntity:

```
import org.springframework.http.ResponseEntity;

@GetMapping("/{id}")
public ResponseEntity<Cliente> buscar(@PathVariable Long id) {
    Cliente cliente = clienteService.buscarPorId(id);
    if (cliente != null) {
        return ResponseEntity.ok(cliente);
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

7.7 Tratamento de Exceções (Global Exception Handler)

📌 Exemplo de Exception personalizada:

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class ClienteNaoEncontradoException extends RuntimeException {
    public ClienteNaoEncontradoException(String mensagem) {
        super(mensagem);
    }
}
```

📌 Exemplo de Exception Handler global:

```
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ClienteNaoEncontradoException.class)
    public String handleNotFound(ClienteNaoEncontradoException ex) {
        return ex.getMessage();
    }
}
```

7.8 Boas Práticas 💡

- ✅ Use DTOs (Data Transfer Objects) para separar a camada de entidade da API.
 - ✅ Nunca retorne entidades diretamente em aplicações grandes.
 - ✅ Documente os endpoints com Swagger/OpenAPI (que veremos mais adiante).
 - ✅ Sempre trate erros e valide dados de entrada (veremos isso nos próximos capítulos).
-

7.9 Revisão do Capítulo

Neste capítulo, você aprendeu:

- ✓ Como criar Controllers REST com Spring MVC.
 - ✓ As anotações principais para mapear requisições HTTP.
 - ✓ Como o Spring Boot transforma objetos Java em JSON.
 - ✓ Como lidar com códigos de resposta HTTP e exceções.
 - ✓ Como testar suas APIs com Postman ou Insomnia.
-

💡 **Próximo Passo:** No próximo capítulo, vamos aprender como **validar os dados das requisições** utilizando **Bean Validation** e anotações como `@Valid` e `@NotNull`.

Capítulo 8 – Validação de Dados com Spring Validation

8.1 Por que validar dados na sua API?

Toda API profissional precisa validar os dados antes de:

- ✓ Persistir no banco
- ✓ Processar regras de negócio
- ✓ Retornar respostas para o cliente

Exemplo de problemas comuns sem validação:

- Cadastro de cliente com e-mail inválido
- Campos obrigatórios faltando
- Dados com tamanho fora do padrão esperado

💡 **Dica:** Uma API mal validada é uma API vulnerável e sujeita a erros em produção.

8.2 O que é Bean Validation?

Bean Validation é uma especificação Java para validar objetos de forma declarativa, usando **anotações diretamente nos atributos das classes**.

👉 O Spring Boot já integra automaticamente o **Hibernate Validator**, que é a implementação de Bean Validation mais popular.

8.3 Dependência necessária (caso esteja criando um projeto manualmente)

Se você usa o **Spring Boot Starter Web**, o suporte a validação já está incluso.

Mas, se precisar adicionar manualmente:

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

8.4 Principais Anotações de Validação

Anotação	Função
@NotNull	Campo não pode ser nulo
@NotBlank	Campo de String não pode ser vazio ou nulo
@Size(min, max)	Define tamanho mínimo e máximo
@Email	Valida formato de e-mail
@Pattern(regexp)	Valida com base em expressão regular
@Min, @Max	Define valores numéricos mínimo/máximo

8.5 Exemplo Prático de Entidade com Validações

```
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;

public class ClienteDTO {

    @NotBlank(message = "Nome é obrigatório")
    @Size(min = 3, max = 50, message = "Nome deve ter entre 3 e 50 caracteres")
    private String nome;

    @Email(message = "E-mail deve ser válido")
    @NotBlank(message = "E-mail é obrigatório")
    private String email;

    // Getters e Setters
}
```

8.6 Como ativar a validação no Controller?

Basta usar a anotação **@Valid** no parâmetro do método:

```
import jakarta.validation.Valid;
import org.springframework.web.bind.annotation.*;
```

```

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

    @PostMapping
    public String criar(@RequestBody @Valid ClienteDTO cliente) {
        return "Cliente criado: " + cliente.getNome();
    }
}

```

👉 Resultado:

Se o campo "nome" for enviado vazio, o Spring retornará um erro HTTP **400 Bad Request** com a mensagem personalizada.

8.7 Exemplo de Resposta de Erro de Validação

Se a requisição for inválida, a resposta será algo como:

```

{
  "timestamp": "2025-06-19T15:30:00.123",
  "status": 400,
  "errors": [
    "Nome é obrigatório",
    "E-mail deve ser válido"
  ]
}

```

8.8 Tratamento Global de Erros de Validação

Podemos criar um **ExceptionHandler global** para personalizar o formato das respostas de erro:

```

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

import java.util.List;
import java.util.stream.Collectors;

@RestControllerAdvice
public class GlobalExceptionHandler {


    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<List<String>>
handleValidationErrors(MethodArgumentNotValidException ex) {
        List<String> errors = ex.getBindingResult()
            .getFieldErrors()
            .stream()
            .map(error -> error.getDefaultMessage())
            .collect(Collectors.toList());

        return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
    }
}


```

```
}  
}
```

8.9 Validação em Update (PUT) também funciona

 Exemplo:

```
@PutMapping("/{id}")  
public String atualizar(@PathVariable Long id, @RequestBody @Valid ClienteDTO  
cliente) {  
    return "Cliente atualizado com sucesso!";  
}
```

 **Dica:** Valide dados tanto no POST quanto no PUT. Nunca confie cegamente nos dados que vêm do cliente.

8.10 Outras validações avançadas (Validação personalizada)

Você pode criar anotações de validação customizadas, mas esse tópico é avançado. Se quiser, posso criar um capítulo extra só para isso.


8.11 Boas Práticas

- ✓ Sempre valide dados de entrada antes de qualquer lógica de negócio.
 - ✓ Retorne mensagens de erro claras e amigáveis ao consumidor da API.
 - ✓ Nunca confie em dados vindos da web.
 - ✓ Centralize o tratamento de exceções para evitar código repetitivo.
-

8.12 Revisão do Capítulo

Neste capítulo, você aprendeu:

- ✓ O que é Bean Validation.
 - ✓ Como validar dados usando anotações como `@NotNull`, `@NotBlank`, `@Email`, etc.
 - ✓ Como ativar a validação nos Controllers com `@Valid`.
 - ✓ Como personalizar a resposta de erro de validação.
 - ✓ Boas práticas de segurança e robustez para APIs REST.
-


 **Próximo Passo:** No próximo capítulo, vamos avançar para o tema de **Spring Security**, onde aprenderemos a proteger a sua API com autenticação e autorização.

Capítulo 9 – Segurança de API com Spring Security

9.1 O que é o Spring Security?

Spring Security é o framework oficial do ecossistema Spring para:

- ✓ **Autenticação** (Quem é você?)
- ✓ **Autorização** (O que você pode fazer?)
- ✓ Proteção contra ataques como **CSRF**, **XSS** e **session fixation**

 **Dica:** Ele é utilizado por grandes empresas para proteger APIs REST, sistemas web e aplicações corporativas.


9.2 Principais conceitos de segurança

Conceito	Significado
Autenticação	Verificar a identidade de quem está acessando
Autorização	Controlar o acesso aos recursos com base em permissões
UserDetailsService	Interface do Spring para buscar dados do usuário
PasswordEncoder	Mecanismo para criptografar e validar senhas

9.3 Adicionando o Spring Security no projeto

 Dependência Maven:

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

 Após adicionar, sua API já estará protegida com **autenticação básica (Basic Auth)** automaticamente!

9.4 Segurança Padrão com Usuário em Memória

Por padrão, o Spring Boot cria um usuário com:

- ✓ Usuário: `user`
- ✓ Senha: Gerada no console no startup

 **Dica:** Esse comportamento é apenas para testes iniciais.

Exemplo de proteção automática:

Endpoint	Requer Autenticação?
/clientes	✓ Sim
/	✓ Sim

9.5 Criando um usuário personalizado (InMemoryUserDetailsManager)

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;
```

```
@Configuration
public class SecurityConfig {

    @Bean
    public InMemoryUserDetailsManager userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("ana")
            .password("1234")
            .roles("USER")
            .build();

        return new InMemoryUserDetailsManager(user);
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .authorizeHttpRequests()
            .anyRequest().authenticated()
            .and()
            .httpBasic();

        return http.build();
    }
}
```

Explicação das Anotações e Métodos:

Anotação	Função
@Configuration	Indica uma classe de configuração
@Bean	Registra um bean no Spring
SecurityFilterChain	Corrente de filtros de segurança
InMemoryUserDetailsManager	Gerencia usuários em memória
http.authorizeHttpRequests()	Configura regras de autorização
.httpBasic()	Define que será usado Basic Auth

9.6 Personalizando Regras de Autorização

📌 Exemplo para proteger somente certos endpoints:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests()
        .requestMatchers("/admin/**").hasRole("ADMIN")
        .requestMatchers("/public/**").permitAll()
        .anyRequest().authenticated()
        .and()
        .httpBasic();

    return http.build();
}
```

9.7 Codificação de Senhas com PasswordEncoder

Nunca armazene senhas em texto puro!

📌 Exemplo usando BCrypt:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

📌 Gerando uma senha criptografada:

```
String senhaCriptografada = passwordEncoder.encode("1234");
```

9.8 Implementando Autenticação com Banco de Dados (UserDetailsService Custom)

Se quiser, posso desenvolver um capítulo separado só para **segurança com JDBC ou JPA**, onde os usuários ficam salvos no banco.

💡 **Dica:** Em sistemas profissionais, o Spring Security quase sempre é integrado com o banco de dados.

9.9 Protegendo Endpoints de API REST

Para proteger suas APIs REST:

- ✅ Use o **Spring Security com HTTP Basic ou JWT**
- ✅ Valide todas as requisições

- ✓ Crie perfis de usuário (ROLE_USER, ROLE_ADMIN)
 - ✓ Retorne status apropriados: **401 Unauthorized** ou **403 Forbidden**
-


9.10 Boas Práticas

- ✓ Sempre use criptografia de senhas (BCrypt, Argon2).
 - ✓ Nunca exponha endpoints administrativos publicamente.
 - ✓ Crie políticas de CORS se sua API for consumida por front-ends externos.
 - ✓ Não confie apenas em segurança de rede, proteja cada endpoint!
-

9.11 Revisão do Capítulo

Neste capítulo, você aprendeu:

- ✓ O que é o Spring Security.
 - ✓ Como proteger suas APIs com autenticação básica.
 - ✓ Como criar usuários em memória.
 - ✓ Como configurar filtros de segurança.
 - ✓ Como usar codificação de senhas.
 - ✓ Regras básicas de autorização para APIs REST.
-

 **Próximo Passo:** No próximo capítulo, vamos aprender a **Documentar APIs com Springdoc OpenAPI e Swagger UI**, facilitando o consumo das suas APIs por outros desenvolvedores.


Capítulo 10 – Documentação de APIs REST com Springdoc OpenAPI

10.1 Por que documentar uma API REST?

Documentação não é luxo, é **necessidade profissional**.

Uma API bem documentada:

- ✓ Facilita a vida de outros desenvolvedores que vão consumir seus endpoints.
- ✓ Garante que sua API seja compreendida sem depender de longas reuniões.
- ✓ Permite a geração automática de **testes manuais interativos** através de ferramentas como o **Swagger UI**.

 **Dica:** Documentar sua API é um sinal de maturidade técnica.

10.2 O que é o Springdoc OpenAPI?


Springdoc OpenAPI é uma biblioteca que gera automaticamente a documentação da sua API Spring Boot com base nas anotações do projeto.

Ele segue o padrão **OpenAPI Specification (antigo Swagger Specification)**.

10.3 Adicionando o Springdoc OpenAPI ao projeto

Dependência Maven:

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.5.0</version>
</dependency>
```

 **Importante:** Essa dependência já inclui o Swagger UI por padrão.

10.4 Acessando o Swagger UI

Após executar o projeto, acesse no navegador:

`http://localhost:8080/swagger-ui/index.html`

 Isso abrirá uma interface web para **testar e visualizar** todos os seus endpoints.

10.5 Personalizando a documentação com anotações

Além da geração automática, podemos personalizar a documentação usando anotações:

Anotação	Função
@Operation	Documenta um endpoint específico
@Parameter	Descreve os parâmetros da requisição
@ApiResponse	Documenta as possíveis respostas HTTP
@Tag	Agrupar endpoints por categorias

Exemplo de Controller documentado:

```
import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.tags.Tag;
import org.springframework.web.bind.annotation.*;
```

```

@RestController
@RequestMapping("/clientes")
@Tag(name = "Clientes", description = "Operações relacionadas a clientes")
public class ClienteController {

    @GetMapping("/{id}")
    @Operation(summary = "Buscar Cliente por ID", description = "Retorna um
cliente específico pelo ID fornecido.")
    public Cliente buscar(@PathVariable Long id) {
        // Lógica para buscar cliente
        return new Cliente();
    }

    @PostMapping
    @Operation(summary = "Criar novo Cliente", description = "Adiciona um novo
cliente ao banco de dados.")
    public Cliente criar(@RequestBody Cliente cliente) {
        // Lógica para criar cliente
        return cliente;
    }
}

```

10.6 Exemplo de Resposta no Swagger UI

Ao acessar o Swagger, você verá algo assim:

GET /clientes/{id}

- **Summary:** Buscar Cliente por ID
 - **Description:** Retorna um cliente específico pelo ID fornecido.
 - **Parameters:**
 - id → Path Variable → Long
 - **Responses:**
 - **200 OK** → Cliente retornado com sucesso
 - **404 Not Found** → Cliente não encontrado
-

10.7 Configuração adicional (se necessário)

Caso queira customizar o título, descrição global, etc.:

 Exemplo de Configuração Global OpenAPI:

```

import io.swagger.v3.oas.models.OpenAPI;
import io.swagger.v3.oas.models.info.Info;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

```

```
@Configuration
public class OpenApiConfig {

    @Bean
    public OpenAPI customOpenAPI() {
        return new OpenAPI()
            .info(new Info()
                .title("API de Gestão de Clientes")
                .version("1.0")
                .description("Documentação da API criada com Spring Boot e
OpenAPI."));
    }
}
```

10.8 Boas Práticas

- ✓ Documente sempre os principais endpoints da sua API.
 - ✓ Use descrições claras e objetivas para cada operação.
 - ✓ Inclua exemplos de request e response quando possível.
 - ✓ Mantenha a documentação sincronizada com a evolução do código.
-


10.9 Benefícios profissionais de usar o Springdoc OpenAPI

- ✓ Facilita o consumo da sua API por frontends (React, Angular, etc.)
 - ✓ Melhora a comunicação entre times (back-end e front-end)
 - ✓ Dá um ar de profissionalismo ao projeto
 - ✓ Permite exportar um arquivo OpenAPI (`openapi.json` ou `openapi.yaml`) para gerar SDKs automáticos em outras linguagens
-

10.10 Revisão do Capítulo

Neste capítulo, você aprendeu:

- ✓ O que é o Springdoc OpenAPI.
 - ✓ Como adicionar o Swagger UI no seu projeto Spring Boot.
 - ✓ Como documentar endpoints com anotações.
 - ✓ Como personalizar e configurar a documentação da sua API.
 - ✓ Como tornar sua API mais amigável para outros desenvolvedores.
-

 **Próximo Passo:** No próximo capítulo, vamos aprender a criar **Testes Unitários e Testes de Integração com Spring Boot Test**, utilizando ferramentas como **JUnit** e **Mockito**.

Capítulo 11 – Testes Unitários e de Integração com Spring Boot Test

11.1 Por que testar aplicações Spring Boot?

Testar é sinônimo de **qualidade**, **confiabilidade** e **manutenção fácil**.

- ✓ Reduz bugs antes de ir para produção
- ✓ Garante que futuras mudanças não quebrem o sistema (regressão)
- ✓ Dá confiança para fazer refatorações
- ✓ Documenta o comportamento esperado da aplicação

 **Dica:** Empresas sérias não colocam código em produção sem testes automatizados.

11.2 Principais tipos de testes no Spring Boot

Tipo de Teste	Objetivo
Teste Unitário	Testa métodos isolados (sem dependências externas)
Teste de Integração	Testa vários componentes juntos (ex: Controller + Service + Repository)
Teste de API/Endpoint	Testa chamadas REST (geralmente usando MockMvc ou TestRestTemplate)

11.3 Dependências para Testes no Spring Boot

Inclua no seu `pom.xml`:

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

Essa dependência inclui:

- ✓ JUnit Jupiter (JUnit 5)
 - ✓ Mockito
 - ✓ Spring Test
 - ✓ AssertJ
 - ✓ Hamcrest
-

11.4 Estrutura típica de um projeto com testes

src


```
└─ main
    └─ java
└─ test
    └─ java
```

📌 Suas classes de teste devem ficar dentro da pasta **src/test/java**.

11.5 Exemplo de Teste Unitário com JUnit e Mockito (Service Layer)

Vamos testar um Service isolado.

📌 Service a ser testado:

```
import org.springframework.stereotype.Service;

@Service
public class CalculadoraService {
    public int somar(int a, int b) {
        return a + b;
    }
}
```

📌 Teste Unitário:

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

public class CalculadoraServiceTest {

    CalculadoraService service = new CalculadoraService();

    @Test
    void deveSomarDoisNumeros() {
        int resultado = service.somar(2, 3);
        assertEquals(5, resultado);
    }
}
```

💡 Explicação:

Este é um **teste puro**, sem Spring. Foca apenas na **lógica da classe isolada**.

11.6 Exemplo de Teste de Integração com Spring Boot

Agora vamos testar **camadas integradas** (ex: Controller + Service + Repository).

📌 Controller de exemplo:

```
import org.springframework.web.bind.annotation.*;

@RestController
```

```

@RequestMapping("/calculadora")
public class CalculadoraController {

    @GetMapping("/soma")
    public int somar(@RequestParam int a, @RequestParam int b) {
        return a + b;
    }
}

```

Teste de Integração usando MockMvc:

```

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.test.web.servlet.MockMvc;

import static
    org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
    org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static
    org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;

@WebMvcTest(CalculadoraController.class)
public class CalculadoraControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void deveRetornarSomaDeDoisNumeros() throws Exception {
        mockMvc.perform(get("/calculadora/soma?a=2&b=3"))
            .andExpect(status().isOk())
            .andExpect(content().string("5"));
    }
}

```

Explicação das Anotações:

Anotação	Função
@WebMvcTest	Carrega apenas os componentes Web (Controller, etc)
@Autowired MockMvc	Ferramenta para simular chamadas HTTP na aplicação

11.7 Testando Repositories com H2 Database (Teste com Banco)

 Exemplo usando H2 (Banco de dados em memória):

```

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

```

Exemplo de Teste de Repository:

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

import static org.assertj.core.api.Assertions.assertThat;

@DataJpaTest
public class ClienteRepositoryTest {

    @Autowired
    private ClienteRepository clienteRepository;

    @Test
    void deveSalvarCliente() {
        Cliente cliente = new Cliente();
        cliente.setNome("Ana");
        cliente.setEmail("ana@example.com");

        Cliente salvo = clienteRepository.save(cliente);
        assertThat(salvo.getId()).isNotNull();
    }
}
```

Explicação:

O Spring inicializa um banco H2 apenas durante o teste e limpa tudo ao final.

11.8 Mockando Dependências com Mockito

Para testes unitários onde você **não quer instanciar o Repository real**, use Mockito:

```
import static org.mockito.Mockito.*;

import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

public class ClienteServiceTest {

    @Mock
    private ClienteRepository clienteRepository;

    @InjectMocks
    private ClienteService clienteService;

    public ClienteServiceTest() {
        MockitoAnnotations.openMocks(this);
    }

    @Test
    void deveChamarSaveDoRepository() {
        Cliente cliente = new Cliente();
        clienteService.salvar(cliente);
        verify(clienteRepository).save(cliente);
    }
}
```

}


11.9 Boas Práticas

- ✓ Separe bem testes unitários de testes de integração.
 - ✓ Automatize os testes em pipelines CI/CD.
 - ✓ Use banco de dados em memória para não afetar o ambiente de produção.
 - ✓ Sempre cubra casos de sucesso e erro.
-

11.10 Revisão do Capítulo

Neste capítulo, você aprendeu:

- ✓ A diferença entre **testes unitários** e **testes de integração**.
 - ✓ Como testar Services, Controllers e Repositories.
 - ✓ Como usar **MockMvc** para testar endpoints REST.
 - ✓ Como isolar dependências usando **Mockito**.
 - ✓ Como usar banco H2 para testes seguros com Repository.
-

 **Próximo Passo:** No próximo capítulo, vamos aprender sobre **Boas Práticas com Spring Boot**, abordando tópicos como estruturação de pacotes, Clean Code e Design Patterns no contexto do Spring.

Capítulo 12 – Boas Práticas de Projeto com Spring Boot

12.1 Por que seguir boas práticas?

Desenvolver código funcional é só o primeiro passo.

O verdadeiro diferencial de um desenvolvedor intermediário para um sênior é:

- ✓ Estruturar bem o projeto
- ✓ Tornar o código limpo, legível e testável
- ✓ Seguir padrões amplamente aceitos pela comunidade Spring
- ✓ Facilitar a manutenção e evolução da aplicação

💡 **Dica:** Código mal organizado vira dívida técnica que se acumula com o tempo.

12.2 Estrutura de pacotes recomendada

Uma estrutura de projeto Spring Boot comum e organizada segue o seguinte padrão:

```
src/main/java
├── com
│   └── suaempresa
│       └── seuprojeto
│           ├── config
│           ├── controller
│           ├── dto
│           ├── exception
│           ├── model
│           ├── repository
│           ├── service
│           └── util
```

Pacote	Responsabilidade
config	Arquivos de configuração (Security, Swagger, etc)
controller	Controllers REST
dto	Data Transfer Objects
exception	Classes de exceção personalizada e handlers
model	Entidades JPA
repository	Interfaces Spring Data JPA
service	Regras de negócio
util	Classes utilitárias

12.3 Aplicando o princípio de responsabilidade única (SRP)

Cada classe deve ter **apenas uma responsabilidade clara**.

🔗 Exemplo:

❌ **Errado:** Um Controller que acessa diretamente o banco.

✅ **Correto:** Controller → chama Service → Service chama Repository.

12.4 Boas Práticas com DTOs

✅ Nunca exponha diretamente as entidades JPA nos endpoints.

✅ Use DTOs para controlar os campos de entrada e saída da API.

🔗 Exemplo de um DTO:

```
public class ClienteDTO {
    private String nome;
    private String email;
```

```
    // Getters e Setters  
}
```

12.5 Uso correto de Exceptions

✓ Crie exceções customizadas para regras de negócio.

📌 Exemplo:

```
@ResponseStatus(HttpStatus.NOT_FOUND)  
public class ClienteNaoEncontradoException extends RuntimeException {  
    public ClienteNaoEncontradoException(String mensagem) {  
        super(mensagem);  
    }  
}
```

✓ Centralize o tratamento de exceções usando `@RestControllerAdvice`.

12.6 Boas Práticas para Repositories

✓ Extenda sempre de `JpaRepository`.

✓ Use nomes de métodos descritivos.

✓ Evite colocar regras de negócio no repository.

📌 Exemplo correto:

```
public interface ClienteRepository extends JpaRepository<Cliente, Long> {  
    Optional<Cliente> findByEmail(String email);  
}
```

12.7 Boas Práticas para Services

✓ Service deve conter **regra de negócio**, nunca lógica de controle web.

📌 Exemplo:

```
@Service  
public class ClienteService {  
  
    @Autowired  
    private ClienteRepository clienteRepository;  
  
    public Cliente salvar(Cliente cliente) {  
        // Regras de negócio aqui  
        return clienteRepository.save(cliente);  
    }  
}
```

12.8 Uso de Logs ao invés de `System.out.println`

Substitua prints por um **Logger profissional**.

🔴 Exemplo com SLF4J:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@Service
public class ClienteService {

    private static final Logger logger =
        LoggerFactory.getLogger(ClienteService.class);

    public Cliente salvar(Cliente cliente) {
        logger.info("Salvando cliente: {}", cliente.getNome());
        return clienteRepository.save(cliente);
    }
}
```

12.9 Boas Práticas com Testes

- ✓ Nomeie as classes de teste com sufixo `Test`.
- ✓ Crie testes para cada camada (Controller, Service, Repository).
- ✓ Use mocks quando testar Service isoladamente.

12.10 Anotações Spring Boot: Quando usar cada uma?

Anotação	Função
@Service	Classe de regra de negócio
@Repository	Acesso a dados
@Controller / @RestController	Camada web
@Component	Caso geral
@Configuration	Arquivos de configuração
@Autowired	Injeção de dependência
@Bean	Criação manual de beans

12.11 Cuidados com o uso de @Autowired

- ✓ Prefira **injeção por construtor**, para garantir imutabilidade e facilitar testes unitários.

🔴 Exemplo:

```
@Service
public class ClienteService {
```

```
private final ClienteRepository clienteRepository;

public ClienteService(ClienteRepository clienteRepository) {
    this.clienteRepository = clienteRepository;
}
}
```


12.12 Clean Code aplicado ao Spring Boot

- ✓ Nomeie variáveis e métodos de forma descritiva
 - ✓ Não deixe código comentado morto no projeto
 - ✓ Mantenha métodos curtos e objetivos
 - ✓ Siga padrões de nomenclatura convencionados
-

12.13 Revisão do Capítulo

Neste capítulo, você aprendeu:

- ✓ Estruturar pacotes de forma profissional.
 - ✓ Aplicar o **Single Responsibility Principle (SRP)**.
 - ✓ Usar DTOs corretamente.
 - ✓ Centralizar tratamento de erros.
 - ✓ Configurar o uso correto de logs.
 - ✓ Seguir boas práticas de Clean Code e Spring Boot.
-

 **Próximo Passo:** No próximo capítulo, vamos abordar **Integração com APIs Externas usando RestTemplate e WebClient**, permitindo que sua aplicação Spring Boot consuma serviços de terceiros.

Capítulo 13 – Integração com APIs Externas: RestTemplate e WebClient

13.1 Por que integrar sua API com serviços externos?

No mundo real, quase nenhuma aplicação trabalha isolada.

- ✓ Integração com serviços de pagamento
- ✓ Consumo de APIs públicas (exemplo: OpenWeather, ViaCEP, etc)
- ✓ Comunicação entre microserviços

 **Dica:** Saber integrar APIs externas é uma habilidade muito valorizada no mercado.

13.2 Duas formas principais de integrar APIs REST no Spring Boot

Tecnologia	Estilo	Quando usar
RestTemplate	Bloqueante, tradicional	Projetos legados ou simples
WebClient	Não bloqueante, reativo	Projetos modernos, alta performance


13.3 Integração usando RestTemplate (modo clássico)

13.3.1 Criando um Bean de RestTemplate:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class AppConfig {

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

 **Explicação da Anotação:**

Anotação	Função
@Configuration	Indica uma classe de configuração Spring
@Bean	Expõe o RestTemplate para injeção

13.3.2 Exemplo de Consumo com RestTemplate:

```
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class ViaCepService {

    @Autowired
    private RestTemplate restTemplate;

    public String buscarEndereco(String cep) {
        String url = "https://viacep.com.br/ws/" + cep + "/json/";
        return restTemplate.getForObject(url, String.class);
    }
}
```

👉 **Explicação:** Fazemos um GET para a API do ViaCEP e recebemos uma String com o JSON da resposta.

13.4 Integração usando WebClient (modo moderno e não bloqueante)

13.4.1 Criando um Bean de WebClient:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.client.WebClient;

@Configuration
public class WebClientConfig {

    @Bean
    public WebClient webClient() {
        return WebClient.builder().build();
    }
}
```

13.4.2 Exemplo de Consumo com WebClient:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;

@Service
public class ViaCepWebClientService {

    @Autowired
    private WebClient webClient;

    public Mono<String> buscarEndereco(String cep) {
        String url = "https://viacep.com.br/ws/" + cep + "/json/";
    }
}
```

```

        return webClient.get()
            .uri(url)
            .retrieve()
            .bodyToMono(String.class);
    }
}

```

👉 Explicação:

Termo	Significado
WebClient	Cliente HTTP reativo
Mono<String>	Tipo reativo que devolve um único valor (String) futuramente
.retrieve()	Executa a requisição
.bodyToMono()	Converte o corpo da resposta

13.5 Testando integração com serviços externos

💡 **Dica:** Evite consumir diretamente serviços externos em testes unitários.

📌 Boas práticas:

- ✅ Use **WireMock** para simular APIs externas em testes.
 - ✅ Ou, em ambiente de desenvolvimento, consuma APIs reais apenas em ambientes de homologação.
-

13.6 Tratamento de Erros nas Chamadas HTTP

- ✅ Trate erros de status como 404, 500, etc.

Exemplo com RestTemplate:

```

try {
    String resposta = restTemplate.getForObject(url, String.class);
} catch (HttpClientErrorException e) {
    // Trata erro 4xx
} catch (HttpServerErrorException e) {
    // Trata erro 5xx
}

```

Exemplo com WebClient:

```

webClient.get()
    .uri(url)
    .retrieve()
    .onStatus(HttpStatus::is4xxClientError, response -> Mono.error(new
RuntimeException("Erro 4xx")))
    .onStatus(HttpStatus::is5xxServerError, response -> Mono.error(new
RuntimeException("Erro 5xx")))
    .bodyToMono(String.class);

```

13.7 Quando usar RestTemplate vs WebClient?

Critério	RestTemplate	WebClient
Simplicidade	✓ Mais simples	✗ Pouco verboso
Projetos legados	✓ Recomendado	✗ Não usual
Projetos modernos	✗ Antigo	✓ Melhor escolha
Performance escalável	✗ Não recomendado	✓ Ideal

💡 **Fato:** O **RestTemplate** está sendo descontinuado gradativamente, o **WebClient** é o futuro.

13.8 Revisão do Capítulo

Neste capítulo, você aprendeu:

- ✓ A importância de integrar APIs externas.
 - ✓ Como usar o **RestTemplate** (modo tradicional).
 - ✓ Como usar o **WebClient** (modo reativo).
 - ✓ Como tratar erros de forma profissional.
 - ✓ Boas práticas para testes e tratamento de falhas.
-

💡 **Próximo Passo:** No próximo capítulo, vamos explorar **Cache com Spring Boot**, onde você aprenderá a otimizar o desempenho da sua aplicação armazenando respostas de consultas pesadas.

Capítulo 14 – Cache com Spring Boot

14.1 O que é Cache?

Cache é uma técnica para **armazenar dados em memória** temporariamente, evitando chamadas repetitivas ao banco de dados ou a APIs externas.

- ✓ Reduz o tempo de resposta
- ✓ Diminui o consumo de recursos de banco
- ✓ Melhora a escalabilidade da aplicação

💡 **Dica:** Imagine um sistema de consulta de CEP: se o mesmo CEP for consultado várias vezes, é melhor buscar o resultado de um cache do que acessar a API ViaCEP novamente.

14.2 Cache no Spring Boot com Spring Cache

Spring Cache é um módulo oficial do Spring para implementação de caching de forma simples e rápida.

14.3 Configurando o Spring Cache

Dependência Maven:

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

14.4 Habilitando o Cache no projeto

Adicione a anotação `@EnableCaching` na sua classe principal (Application):

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cache.annotation.EnableCaching;

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

Explicação:

Anotação	Função
<code>@EnableCaching</code>	Ativa o suporte a cache no projeto Spring

14.5 Cache básico com `@Cacheable`

Exemplo:

```
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

@Service
public class ClienteService {

    @Cacheable("clientes")
    public Cliente buscarPorId(Long id) {
        System.out.println("Consultando o banco de dados...");
        return clienteRepository.findById(id).orElseThrow();
    }
}
```

```
}  
}
```

💡 Explicação da Anotação @Cacheable:

Atributo	Função
"clientes"	Nome da região/cache que vai armazenar os dados
Resultado:	Na primeira chamada, busca do banco. Nas próximas, busca direto do cache.

14.6 Limpando o Cache: @CacheEvict

🔗 Exemplo de como limpar o cache após salvar ou excluir um cliente:

```
import org.springframework.cache.annotation.CacheEvict;  
import org.springframework.stereotype.Service;  
  
@Service  
public class ClienteService {  
  
    @CacheEvict(value = "clientes", allEntries = true)  
    public void salvar(Cliente cliente) {  
        clienteRepository.save(cliente);  
    }  
}
```

💡 Explicação:

Toda vez que salvar um cliente, o cache da região "clientes" será limpo.

14.7 Atualizando o Cache: @CachePut

🔗 Exemplo:

```
import org.springframework.cache.annotation.CachePut;  
import org.springframework.stereotype.Service;  
  
@Service  
public class ClienteService {  
  
    @CachePut(value = "clientes", key = "#cliente.id")  
    public Cliente atualizar(Cliente cliente) {  
        return clienteRepository.save(cliente);  
    }  
}
```

Anotação	Função
@CachePut	Atualiza o cache após executar o método

14.8 Cache em Métodos com Parâmetros

Exemplo: Caching em métodos que recebem parâmetros (exemplo: CEP)

```
@Cacheable(value = "enderecos", key = "#cep")
public Endereco buscarEndereco(String cep) {
    // Lógica de consulta externa
}
```



Dica: O Spring gera uma chave de cache única com base no parâmetro cep.

14.9 Implementações de Cache Suportadas pelo Spring Boot

Tecnologia	Indicação
SimpleCacheManager (padrão)	Armazena tudo em memória, apenas para testes
EhCache / Caffeine	Boa para produção
Redis	Melhor escolha para ambientes distribuídos e aplicações em nuvem

14.10 Exemplo: Cache com Caffeine (Cache de Alta Performance)



Dependência:

```
<dependency>
  <groupId>com.github.ben-manes.caffeine</groupId>
  <artifactId>caffeine</artifactId>
</dependency>
```



Configuração:

```
import com.github.benmanes.caffeine.cache.Caffeine;
import org.springframework.cache.CacheManager;
import org.springframework.cache.caffeine.CaffeineCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.concurrent.TimeUnit;

@Configuration
public class CacheConfig {

    @Bean
    public CacheManager cacheManager() {
        CaffeineCacheManager cacheManager = new
        CaffeineCacheManager("clientes");
        cacheManager.setCaffeine(Caffeine.newBuilder()
            .expireAfterWrite(10, TimeUnit.MINUTES)
            .maximumSize(100));
        return cacheManager;
    }
}
```


14.11 Cuidados com o uso de Cache

- ✓ Valide se o cache realmente faz sentido para o seu caso.
 - ✓ Sempre tenha uma estratégia de invalidação (expiração ou limpeza manual).
 - ✓ Cuidado com memória: cache infinito pode causar problemas de performance.
-

14.12 Revisão do Capítulo

Neste capítulo, você aprendeu:

- ✓ O que é cache e por que usá-lo.
 - ✓ Como configurar o Spring Cache.
 - ✓ Como usar as anotações `@Cacheable`, `@CacheEvict` e `@CachePut`.
 - ✓ Como trabalhar com caches de alto desempenho (ex: Caffeine, Redis).
 - ✓ Boas práticas para gestão de cache.
-

 **Próximo Passo:** No próximo capítulo, vamos explorar **Mensageria com Spring Boot**, aprendendo a usar o **RabbitMQ** para comunicação assíncrona entre serviços.

Capítulo 15 – Mensageria com Spring Boot e RabbitMQ

15.1 O que é Mensageria?

Mensageria é um modelo de comunicação **assíncrona** entre sistemas.

Em vez de serviços conversarem em tempo real via HTTP, eles trocam mensagens através de uma **fila (queue)**.

- ✓ Permite desacoplar serviços
- ✓ Garante resiliência (caso o destinatário esteja offline)
- ✓ Suporta picos de tráfego com menos risco de sobrecarga

 **Dica:** Empresas que trabalham com **microserviços**, **e-commerce** e **sistemas de processamento em lote** usam mensageria intensivamente.

15.2 O que é o RabbitMQ?

O **RabbitMQ** é um dos **Message Brokers** mais usados no mundo Java.


Ele implementa o protocolo **AMQP (Advanced Message Queuing Protocol)**, fornecendo:

- ✓ Fila de mensagens
 - ✓ Exchange (roteador de mensagens)
 - ✓ Sistema de confirmação de entrega (ACK)
 - ✓ Persistência opcional de mensagens
-

15.3 Dependência Maven para RabbitMQ no Spring Boot

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

15.4 Configurando o RabbitMQ no Spring Boot

 Exemplo de configuração de filas, exchanges e bindings:

```
import org.springframework.amqp.core.*;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RabbitMQConfig {

    public static final String QUEUE = "fila_clientes";
    public static final String EXCHANGE = "exchange_clientes";
    public static final String ROUTING_KEY = "clientes.routingkey";

    @Bean
    public Queue queue() {
        return new Queue(QUEUE);
    }

    @Bean
    public TopicExchange exchange() {
        return new TopicExchange(EXCHANGE);
    }

    @Bean
    public Binding binding(Queue queue, TopicExchange exchange) {
        return BindingBuilder.bind(queue).to(exchange).with(ROUTING_KEY);
    }
}
```

 **Explicação das Anotações:**

Anotação	Função
@Configuration	Classe de configuração Spring
@Bean	Instancia e registra os objetos no contexto Spring

15.5 Publicando Mensagens com RabbitTemplate

🔗 Exemplo de Producer (quem envia a mensagem):

```
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class ClienteProducer {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    public void enviarMensagem(String mensagem) {
        rabbitTemplate.convertAndSend(
            RabbitMQConfig.EXCHANGE,
            RabbitMQConfig.ROUTING_KEY,
            mensagem
        );
    }
}
```

💡 **Explicação:**

O método `convertAndSend` envia a mensagem para o **Exchange**, que roteia até a **Queue**.

15.6 Consumindo Mensagens com @RabbitListener

🔗 Exemplo de Consumer (quem recebe a mensagem):

```
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Service;

@Service
public class ClienteConsumer {

    @RabbitListener(queues = RabbitMQConfig.QUEUE)
    public void consumirMensagem(String mensagem) {
        System.out.println("Mensagem recebida: " + mensagem);
        // Aqui você pode processar a mensagem, salvar no banco, etc.
    }
}
```

💡 **Explicação da Anotação:**

Anotação	Função
@RabbitListener	Faz a classe escutar mensagens da fila indicada

15.7 Fluxo de Comunicação

👉 **Producer** → **Exchange** → **Queue** → **Consumer**

- ✓ Producer envia
 - ✓ Exchange roteia
 - ✓ Queue armazena
 - ✓ Consumer consome
-

15.8 Boas Práticas com Mensageria

- ✓ Não use mensageria para fluxos que exigem resposta síncrona imediata.
 - ✓ Garanta **idempotência** no consumo (evitar duplicações).
 - ✓ Mantenha logs detalhados de envio e consumo.
 - ✓ Se o processamento da mensagem falhar, implemente **dead-letter queues** para tratar falhas.
-

15.9 Exemplo de uso prático no mundo real:

- ✓ Notificações de envio de e-mails
 - ✓ Processamento assíncrono de pedidos em um e-commerce
 - ✓ Registro de logs e auditorias
 - ✓ Integração entre microserviços
-

15.10 Revisão do Capítulo

Neste capítulo, você aprendeu:

- ✓ O que é mensageria.
 - ✓ Como o RabbitMQ funciona dentro do Spring Boot.
 - ✓ Como enviar e consumir mensagens com `RabbitTemplate` e `@RabbitListener`.
 - ✓ Como configurar queues, exchanges e bindings.
 - ✓ Boas práticas para produção.
-


💡 **Próximo Passo:** No próximo capítulo, vamos estudar sobre **Configuração Externa no Spring Boot**, entendendo como trabalhar com arquivos `.properties`, `.yaml` e o poderoso recurso **Spring Profiles**.

Capítulo 16 – Configuração Externa e Profiles no Spring Boot

16.1 Por que externalizar configurações?

Separar configurações do código é um princípio essencial para:

- ✓ Facilitar a mudança de ambiente (dev, homologação, produção)
- ✓ Manter dados sensíveis fora do código-fonte
- ✓ Tornar a aplicação mais flexível e segura

 **Dica:** Nunca coloque senhas de banco de dados ou tokens de API hardcoded dentro da aplicação.


16.2 Principais formatos de configuração no Spring Boot

Formato	Extensão	Exemplo
Properties	.properties	application.properties
YAML	.yaml	application.yml

16.3 Exemplo com application.properties

 Exemplo simples:

```
server.port=8081
spring.datasource.url=jdbc:mysql://localhost:3306/minha_base
spring.datasource.username=root
spring.datasource.password=senha123
```

 **Dica:** O Spring Boot procura automaticamente um arquivo `application.properties` ou `application.yml` na pasta `/src/main/resources`.

16.4 Exemplo com application.yml (YAML)

```
server:
  port: 8081

spring:
  datasource:
    url: jdbc:mysql://localhost:3306/minha_base
    username: root
    password: senha123
```



Diferença:

O YAML é mais legível para estruturas hierárquicas.

16.5 Criando múltiplos profiles: dev, test, prod



Exemplo de múltiplos arquivos de configuração:

```
src/main/resources/
├── application.properties
├── application-dev.properties
├── application-test.properties
└── application-prod.properties
```



Conteúdo de cada arquivo:

application-dev.properties:

```
server.port=8080
spring.datasource.url=jdbc:mysql://localhost:3306/dev_db
```

application-prod.properties:

```
server.port=80
spring.datasource.url=jdbc:mysql://servidor_producao:3306/prod_db
```

16.6 Ativando profiles via linha de comando

Exemplo:

```
java -jar minha-api.jar --spring.profiles.active=prod
```



Explicação:

O Spring Boot carregará o arquivo `application-prod.properties`.

16.7 Ativando profiles via `application.properties` principal



No `application.properties` padrão:

```
spring.profiles.active=dev
```



Dica: Nunca deixe o profile de produção ativo no repositório de desenvolvimento!

16.8 Injetando valores de configuração com `@Value`



Exemplo:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class ConfigExample {

    @Value("${server.port}")
    private int porta;

    public void mostrarPorta() {
        System.out.println("Aplicação rodando na porta: " + porta);
    }
}
```

Anotação	Função
@Value	Injecta o valor de uma propriedade externa

16.9 Boas práticas com Configurações

- ✓ Separe configurações sensíveis usando **variáveis de ambiente** ou serviços de vault.
 - ✓ Para projetos maiores, centralize configurações com **Spring Cloud Config Server**.
 - ✓ Documente suas configurações: outros devs precisam entender o que cada chave faz.
-

16.10 Trabalhando com @ConfigurationProperties

📌 Exemplo para mapear várias configurações em uma única classe:

application.properties:

```
app.titulo=Minha API Java
app.versao=1.0.0
```

Classe Java:

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

@Component
@ConfigurationProperties(prefix = "app")
public class AppProperties {

    private String titulo;
    private String versao;

    // Getters e Setters
}
```


💡 **Explicação:**

Anotação	Função
@ConfigurationProperties	Mapeia múltiplas propriedades com o mesmo prefixo
@Component	Faz a classe ser gerenciada pelo Spring

16.11 Revisão do Capítulo

Neste capítulo, você aprendeu:

- ✓ A importância de externalizar configurações.
 - ✓ Diferença entre `.properties` e `.yaml`.
 - ✓ Como criar e ativar perfis de ambiente (dev, test, prod).
 - ✓ Como injetar configurações com `@Value` e `@ConfigurationProperties`.
 - ✓ Boas práticas de segurança e organização de configurações.
-

 **Próximo Passo:** No próximo capítulo, vamos mergulhar no tema **Estudo de Caso: Construindo uma API REST Completa com Spring Boot**, iremos consolidar tudo o que aprendemos nos 15 capítulos anteriores.

Capítulo 17 – Estudo de Caso: Construindo uma API REST Completa com Spring Boot

17.1 Objetivo do Estudo de Caso

Neste capítulo, vamos **consolidar tudo o que aprendemos nos 16 capítulos anteriores** através da construção de uma API REST real:

Projeto: Cadastro de Clientes com Spring Boot

Requisitos da API:

- ✓ CRUD completo (Create, Read, Update, Delete)
- ✓ Validação com Bean Validation
- ✓ Persistência com JPA e Hibernate
- ✓ Uso de DTOs
- ✓ Camadas organizadas (Controller, Service, Repository)
- ✓ Configuração de perfis
- ✓ Cache com Spring Cache
- ✓ Logs com SLF4J
- ✓ Integração com RabbitMQ (Opcional)

17.2 Estrutura do Projeto

```
src/main/java
├── com
│   └── empresa
│       └── clienteapi
│           ├── config
│           ├── controller
│           ├── dto
│           ├── exception
│           ├── model
│           ├── repository
│           ├── service
│           └── util
```

17.3 Entidade Cliente (Model)

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Cliente {

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

    private String nome;

    private String email;

    private String telefone;

    // Getters e Setters
}
```




Explicação:

Usamos **JPA** com a anotação `@Entity` para mapear o Cliente no banco.

17.4 DTO de Cliente (ClienteDTO)

```
public class ClienteDTO {
    private String nome;
    private String email;
    private String telefone;

    // Getters e Setters
}
```


 **Boas práticas:** Nunca exponha entidades diretamente nos endpoints.

17.5 Repository

```
import org.springframework.data.jpa.repository.JpaRepository;

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

17.6 Service com Regras de Negócio

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class ClienteService {

    @Autowired
    private ClienteRepository repository;

    public Cliente salvar(Cliente cliente) {
        return repository.save(cliente);
    }

    public List<Cliente> listarTodos() {
        return repository.findAll();
    }

    public Cliente buscarPorId(Long id) {
        return repository.findById(id).orElseThrow(() -> new
RuntimeException("Cliente não encontrado"));
    }

    public void deletar(Long id) {
        repository.deleteById(id);
    }
}
```

17.7 Controller REST

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;

import java.util.List;

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

```

@Autowired
private ClienteService service;

@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Cliente salvar(@RequestBody Cliente cliente) {
    return service.salvar(cliente);
}

@GetMapping
public List<Cliente> listar() {
    return service.listarTodos();
}

@GetMapping("/{id}")
public Cliente buscarPorId(@PathVariable Long id) {
    return service.buscarPorId(id);
}

@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deletar(@PathVariable Long id) {
    service.deletar(id);
}
}

```

17.8 Configuração de Profile para Ambiente de Desenvolvimento

application-dev.properties:

```

server.port=8080
spring.datasource.url=jdbc:h2:mem:clientesdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=update
spring.profiles.active=dev

```



Banco em Memória: Utilizamos o **H2** apenas para desenvolvimento.

17.9 Validação com Bean Validation

Exemplo de validação no DTO:

```

import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;

public class ClienteDTO {

    @NotBlank(message = "Nome é obrigatório")

```

```
private String nome;

@email(message = "Email inválido")
private String email;

@NotBlank(message = "Telefone é obrigatório")
private String telefone;

// Getters e Setters
}
```

Anotações de Validação:

- @NotBlank → Campo não pode ser vazio
 - @Email → Formato de e-mail válido
-

17.10 Exemplo de Teste de Endpoint com cURL

```
curl -X POST http://localhost:8080/api/clientes \
-H "Content-Type: application/json" \
-d '{"nome":"Ana","email":"ana@email.com","telefone":"99999-9999"}'
```

17.11 Possíveis Extensões Futuras

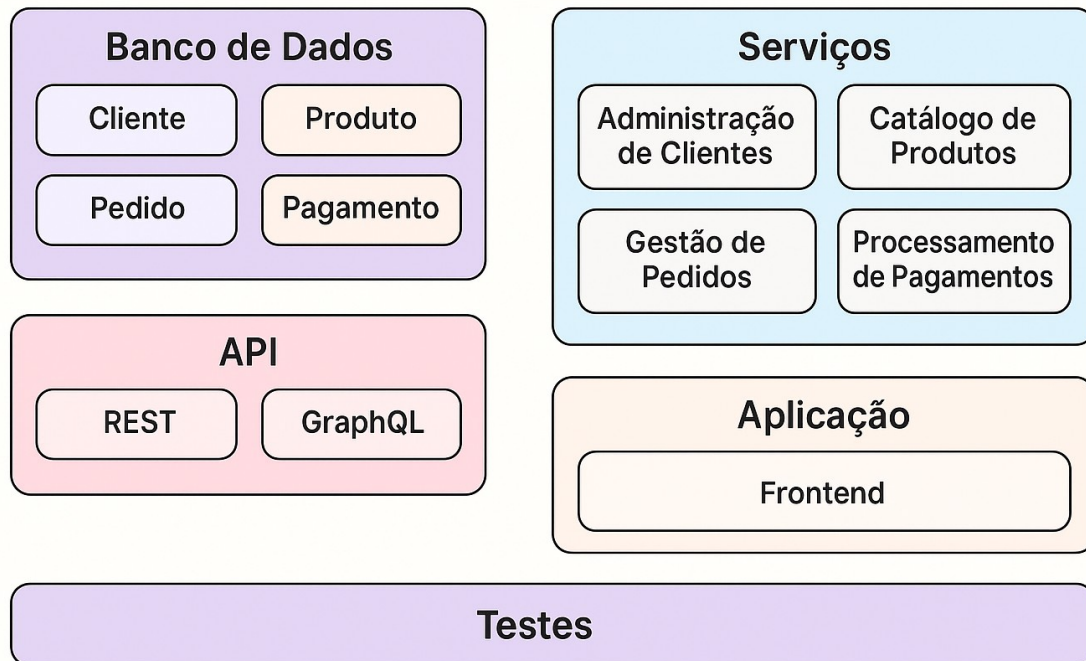
- ✓ Implementar Cache no método de listagem
- ✓ Integrar com RabbitMQ para notificação de novos clientes
- ✓ Expor métricas com Spring Boot Actuator
- ✓ Adicionar autenticação com Spring Security

17.12 – Visão Arquitetural: Spring Framework Architecture

Objetivo:

Apresentar uma visão arquitetural clara e objetiva de como os módulos e componentes do **Spring Framework** interagem dentro de uma aplicação real (como o nosso Estudo de Caso da API de Clientes).

Estudo de caso abordado no Capítulo 17



Explicação Rápida da Arquitetura – Estudo de Caso

A imagem representa a **arquitetura lógica e modular da aplicação construída no Estudo de Caso desse Capítulo**. Ela divide os componentes principais da aplicação Spring Boot em **5 grandes blocos funcionais**, cada um com sua responsabilidade clara dentro do sistema.

Camada / Bloco	Responsabilidade	Exemplos Práticos
Banco de Dados	Representa o modelo de persistência de dados. Contém todas as entidades mapeadas com JPA/Hibernate que são persistidas no banco relacional (MySQL, PostgreSQL, etc).	Tabelas: Cliente, Produto, Pedido, Pagamento
Serviços (Service Layer)	Onde ficam todas as regras de negócio da aplicação. Aqui tratamos as lógicas de manipulação de dados, validação e processamento antes de expor os dados à camada de API.	Administração de Clientes, Gestão de Pedidos, Catálogo de Produtos, Processamento de Pagamentos
API (Camada de Exposição)	Responsável por abrir os endpoints para consumo externo. Pode ser via REST ou	Exemplo: GET <code>/api/clientes</code> , POST

Camada / Bloco	Responsabilidade	Exemplos Práticos
Aplicação (Frontend ou Camadas Adjacentes)	GraphQL, dependendo da estratégia de exposição de serviços escolhida.	/api/pedidos, ou uma consulta via GraphQL
	Interface que o usuário final irá interagir. Pode ser um Angular , React , Vue.js ou até um cliente mobile.	Exemplo: Um painel web de administração de clientes
Testes (QA e Validação)	Toda a camada de testes automatizados. Inclui Testes Unitários , Testes de Integração , Testes de Contrato para garantir a robustez do sistema.	Exemplo: Testes com JUnit, Mockito, TestContainers



Dicas para Projetos Reais:

- Você pode facilmente escalar esse modelo modular para incluir autenticação (Spring Security), monitoramento (Spring Actuator) ou mensageria (RabbitMQ).
- Separar bem cada camada (Banco, Serviços, API, Aplicação e Testes) melhora a **manutenibilidade**, **testabilidade** e **escalabilidade** do projeto.



Como usar esse Diagrama?

- Inclua-o como parte de sua documentação técnica.
- Apresente em entrevistas para demonstrar seu domínio sobre arquitetura Spring Boot.
- Use como guia para estruturar novos projetos futuros.

17.13 Conclusão do Estudo de Caso

Neste capítulo, você aplicou **todos os conceitos práticos** vistos ao longo do e-book:



JPA



DTO



Service Layer



Repository



Controller REST



Profiles



Validação



Camadas organizadas



Boas práticas Spring Boot



Desafio Final: Pegue este projeto, suba no GitHub e comece a melhorar!

Adicione autenticação, pague os resultados, integre com um frontend...

Conclusão Final – Obrigada por Chegar Até Aqui, Desenvolvedores!

Chegar até o fim deste e-book não foi apenas um avanço técnico, mas uma verdadeira jornada de transformação no seu conhecimento em **Java e Spring Boot**.

Ao longo dos 17 capítulos, você explorou com profundidade:

- ✓ Os fundamentos da arquitetura Spring Boot
 - ✓ Como criar APIs REST robustas e escaláveis
 - ✓ O poder das anotações como `@RestController`, `@Service`, `@Repository`, `@Autowired`, `@Cacheable`, entre outras
 - ✓ Persistência de dados com **JPA e Hibernate**
 - ✓ Boas práticas de projeto com **DTOs**, **Service Layer** e **tratamento de exceções centralizado**
 - ✓ Técnicas modernas de **consumo de APIs externas** com **WebClient**
 - ✓ Implementação de **Cache** para performance
 - ✓ **Mensageria assíncrona** com **RabbitMQ**
 - ✓ Gerenciamento de configurações com **Profiles**
 - ✓ Monitoramento e exposição de métricas com o **Spring Boot Actuator**
 - ✓ E, por fim, consolidou tudo com um **Estudo de Caso prático**, unindo teoria e prática!
-

Uma Jornada que Está Apenas Começando...

Este e-book foi construído capítulo a capítulo, com muito zelo, atenção aos detalhes e sempre com foco em te ajudar a alcançar o **nível intermediário avançado** em desenvolvimento Java com Spring Boot.

Você agora tem ferramentas reais para:

- ✓ Trabalhar em empresas que utilizam microserviços
- ✓ Contribuir com APIs REST profissionais
- ✓ Participar de projetos que exigem escalabilidade, segurança e performance

Dica Final:

Nunca pare de praticar.

Refatore este projeto.

Implemente autenticação com Spring Security.

Explore Spring Cloud.

Experimente testes unitários e de integração.



Bibliografia

SPRING.IO. *Spring Framework Documentation*. Disponível em: <https://spring.io/projects/spring-framework>. Acesso em: 19 jun. 2025.

SPRING.IO. *Spring Boot Reference Documentation*. Disponível em: <https://spring.io/projects/spring-boot>. Acesso em: 19 jun. 2025.

SPRING.IO. *Spring Guides: Building a RESTful Web Service*. Disponível em: <https://spring.io/guides/gs/rest-service/>. Acesso em: 19 jun. 2025.

SPRING.IO. *Spring Boot Caching Documentation*. Disponível em: <https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.caching>. Acesso em: 19 jun. 2025.

SPRING.IO. *Messaging with RabbitMQ using Spring Boot*. Disponível em: <https://spring.io/guides/gs/messaging-rabbitmq/>. Acesso em: 19 jun. 2025.

SPRING.IO. *Spring Boot Actuator Documentation*. Disponível em: <https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html>. Acesso em: 19 jun. 2025.

SPRING.IO. *Spring Data JPA Reference Documentation*. Disponível em: <https://spring.io/projects/spring-data-jpa>. Acesso em: 19 jun. 2025.

ORACLE. *Java Platform, Enterprise Edition: Java EE 8 API Documentation*. Disponível em: <https://javaee.github.io/javaee-spec/javadocs/>. Acesso em: 19 jun. 2025.

ORACLE. *The Java™ Tutorials*. Disponível em: <https://dev.java/learn/>. Acesso em: 19 jun. 2025.