



DE PADAWAN A JEDI:
DOMINANDO A ARTE DO
spring®

Introdução – A Nova Aliança Spring

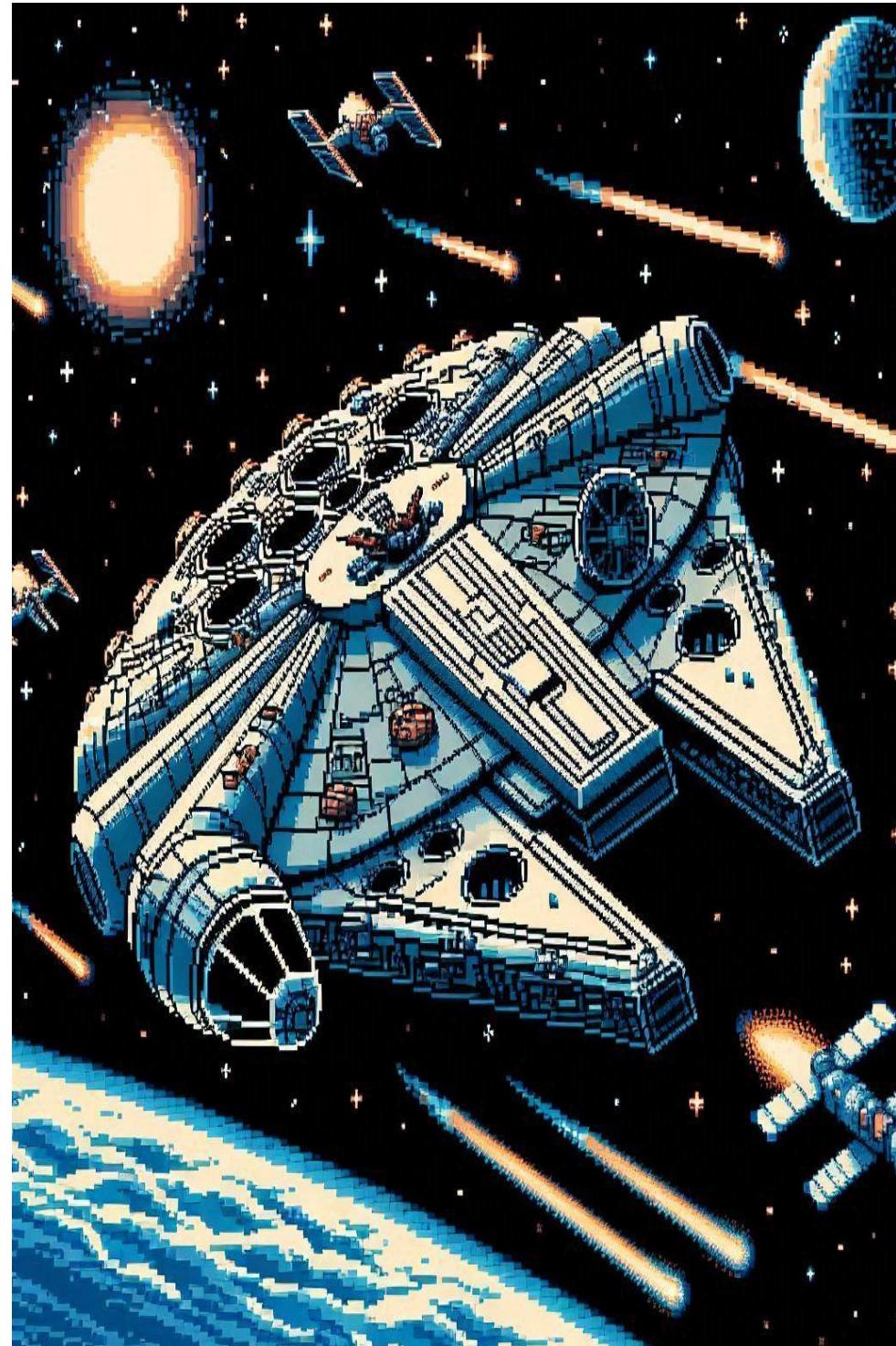
Há muito tempo, em uma galáxia não tão distante...

No vasto universo do desenvolvimento Java, construir aplicações modernas e eficientes nem sempre foi uma tarefa simples. Durante muito tempo, configurar ambientes, lidar com dependências e manter projetos organizados era um verdadeiro campo de batalha.

Foi nesse cenário que surgiu o Spring Boot, uma poderosa ferramenta capaz de reduzir drasticamente a complexidade do desenvolvimento backend em Java, acelerando a criação de aplicações robustas e escaláveis.

Este e-book é um guia prático para quem deseja dominar essa tecnologia, desde os fundamentos até o deploy em produção. A cada capítulo, você será conduzido por exemplos reais, boas práticas e conceitos explicados de forma simples e direta.

Se você já conhece Java e quer transformar seu conhecimento em aplicações prontas para o mundo real, este material foi feito para você. Aperte o cinto. Sua jornada com o Spring Boot começa agora.



PARTE I

Fundamentos Iniciais

SE TORNE UM MESTRE JEDI EM
SPRING BOOT

O que é Spring Boot e por que usar

Menos configuração, mais produtividade

Spring Boot é um framework Java que facilita a criação de aplicações modernas, principalmente APIs web. Ele elimina configurações manuais do Spring tradicional, e já vem com um servidor embarcado, suporte a REST, segurança, persistência e muito mais.

Por que usar:

- Evita configurações repetitivas
- Substitui arquivos XML por anotações Java
- Ideal para microserviços e APIs REST



Criando seu primeiro projeto

Preparando seu ambiente com Spring Initializr

A maneira mais rápida de começar um projeto é pelo Spring Initializr:

- Selecione:
 - Projeto: Maven
 - Linguagem: Java
 - Dependências: Spring Web
- Clique em **Generate** e importe o projeto no seu IDE
- O arquivo principal do projeto vem assim:



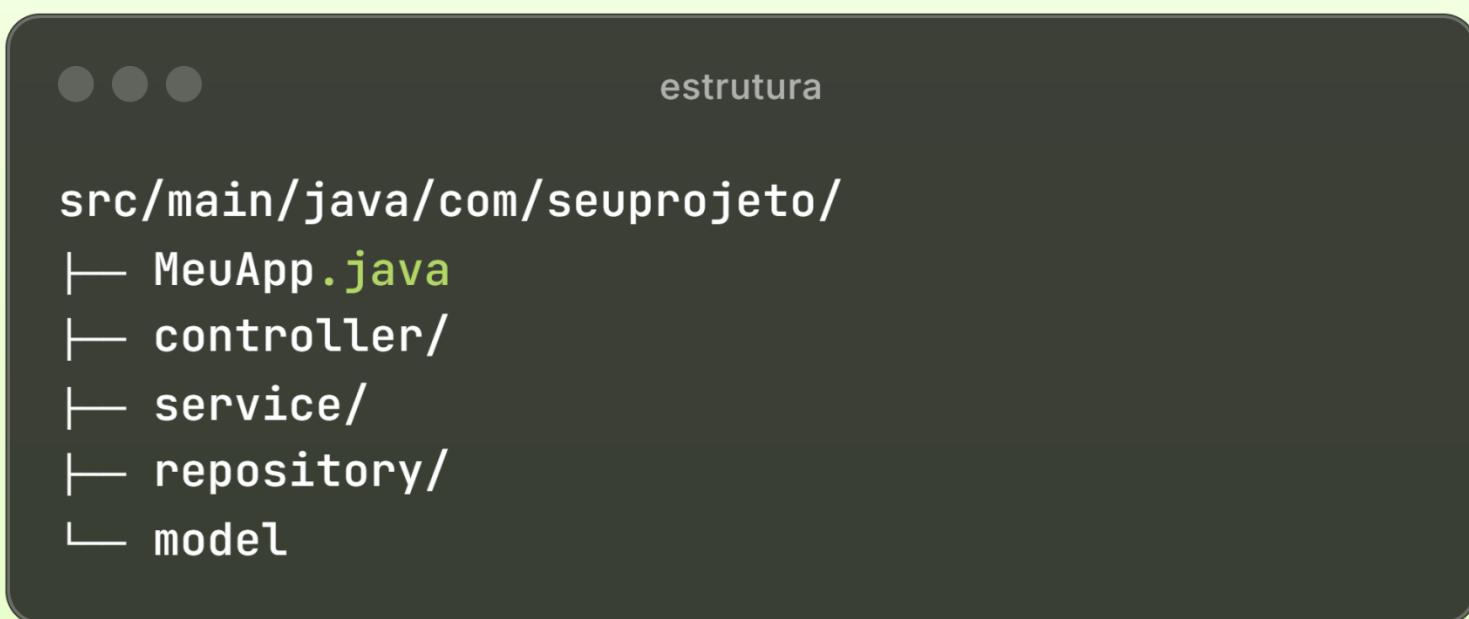
E-BOOK FEITO POR LeoMatias.java

```
1 public class MeuApp {  
2     public static void main(String[] args) {  
3         SpringApplication.run(MeuApp.class, args);  
4     }  
5 }  
6
```

Estrutura básica e boas práticas

Separando responsabilidades desde o início

A estrutura recomendada é baseada em camadas:



Funções:

- *controller*: lida com requisições HTTP
- *service*: regras de negócio
- *repository*: acesso ao banco de dados
- *model*: entidades e DTOs (objetos de transferência de dados)

Organizar bem facilita a manutenção e testes.

Criando suas primeiras rotas REST

Fazendo sua aplicação responder à web

Para criar um endpoint, usamos `@RestController` e mapeamos URLs com `@GetMapping`, `@PostMapping`, etc.

```
● ● ● E-BOOK FEITO POR LeoMatias.java  
1 @RestController  
2 @RequestMapping("/api")  
3  
4 public class OlaController {  
5  
6     @GetMapping("/ola")  
7     public String dizerOla() {  
8         return "Olá, mundo!";  
9     }  
10 }  
11  
12
```

Agora, ao acessar <http://localhost:8080/api/ola>, o navegador exibe "Olá, mundo!".

Entendendo a injeção de dependência

Automatizando o uso de componentes

Spring Boot gerencia objetos para você. Basta anotar uma classe com `@Service` ou `@Component`, e usá-la com `@Autowired`.

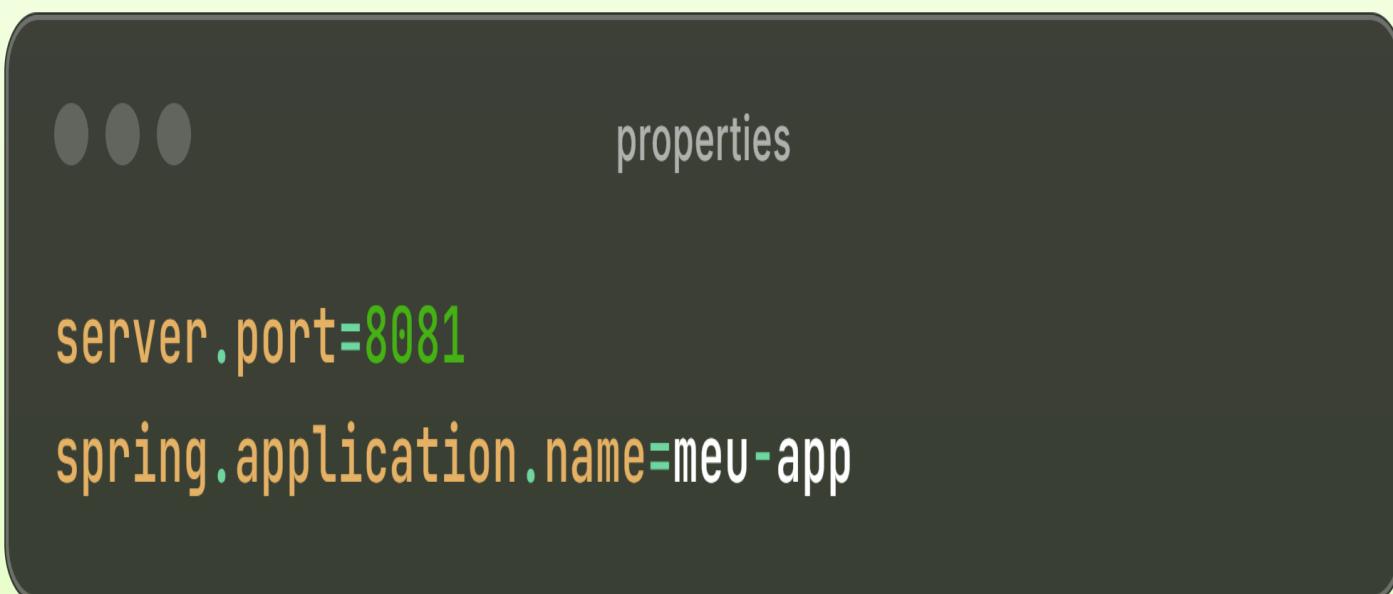
```
● ● ● E-BOOK FEITO POR LeoMatias.java  
1 @Service  
2 public class SaudacaoService {  
3     public String gerar() {  
4         return "Seja bem-vindo!";  
5     }  
6 }  
7  
8  
9  
10
```

```
● ● ● E-BOOK FEITO POR LeoMatias.java  
1 @RestController  
2 public class SaudacaoController {  
3  
4     @Autowired  
5     private SaudacaoService saudacaoService;  
6  
7     @GetMapping("/saudacao")  
8     public String get() {  
9         return saudacaoService.gerar();  
10    }  
11 }  
12  
13  
14
```

Configurando o app com properties

Centralizando configurações do projeto

O Spring Boot usa o arquivo *application.properties* para configurar o app.



Você pode:

- Alterar a porta do servidor
- Configurar acesso ao banco
- Definir perfis (dev, prod, etc.)

Usando DevTools para atualização

Salvou, atualizou – desenvolvimento em tempo real

DevTools permite recarregar o app automaticamente ao salvar, sem reinicialização. Adicione no *pom.xml*:

```
...xml  
1 <dependency>  
2   <groupId>org.springframework.boot</groupId>  
3   <artifactId>spring-boot-devtools</artifactId>  
4   <optional>true</optional>  
5 </dependency>  
6
```

Vantagens:

- Atualização automática ao editar o código
- Melhor produtividade

Reinic peace o projeto uma vez após instalar.

Empacotando e executando com .jar

Levando sua aplicação para produção (ou outro computador)

Você pode gerar um *.jar* (Java ARchive) para rodar seu projeto fora do IDE:

No terminal:

```
./mvnw clean package
```

Depois, rode:

```
java -jar target/meuapp-0.0.1-SNAPSHOT.jar
```

O app sobe como se estivesse no IDE, pronto para deploy!

PARTE 2

Persistência
De Dados
com
Spring Data
JPA

SE TORNE UM MESTRE JEDI EM
SPRING BOOT

Conectando a um banco de dados

Configurando o acesso ao seu banco em segundos

O Spring Boot facilita a conexão com bancos como MySQL, PostgreSQL ou H2 (memória). Tudo que você precisa é configurar o application.properties:

```
properties  
1 spring.datasource.url=jdbc:mysql://localhost:3306/meubanco  
2 spring.datasource.username=root  
3 spring.datasource.password=123456  
4 spring.jpa.hibernate.ddl-auto=update  
5 spring.jpa.show-sql=true
```

Dica: Para testes rápidos, use o H2:

```
properties  
1 spring.datasource.url=jdbc:h2:mem:testdb  
2 spring.h2.console.enabled=true
```

Depois acesse: <http://localhost:8080/h2-console>

Criando entidades com @Entity e @Id

Modelando suas tabelas com classes Java

Cada tabela no banco pode ser representada por uma classe com a anotação `@Entity`. Use `@Id` para definir a chave primária.

```
● ● ● EBOOK FEITO POR LeoMatias.java

1  @Entity
2  public class Usuario {
3
4      @Id
5      @GeneratedValue(strategy = GenerationType.IDENTITY)
6      private Long id;
7
8      private String nome;
9      private String email;
10
11     // Getters e Setters
12 }
```

O `@GeneratedValue` permite que o banco gere o ID automaticamente.

Salvando e consultando dados

CRUD automático sem escrever SQL

Crie um repositório que herda *JpaRepository* e o Spring gera todos os métodos padrão (CRUD) para você.



EBOOK FEITO POR LeoMatias.java

```
1 public interface UsuarioRepository extends JpaRepository<Usuario, Long> {}
```

No controller, você pode salvar e buscar usuários:



EBOOK FEITO POR LeoMatias.java

```
1 @RestController
2 @RequestMapping("/usuarios")
3 public class UsuarioController {
4
5     @Autowired
6     private UsuarioRepository repo;
7
8     @PostMapping
9     public Usuario salvar(@RequestBody Usuario usuario) {
10         return repo.save(usuario);
11     }
12
13     @GetMapping("/{id}")
14     public ResponseEntity<Usuario> buscar(@PathVariable Long id) {
15         return repo.findById(id)
16             .map(ResponseEntity::ok)
17             .orElse(ResponseEntity.notFound().build());
18     }
19 }
```

Usando relacionamentos

Ligando suas entidades como no mundo real

Você pode criar relações entre tabelas com anotações como `@ManyToOne` e `@OneToMany`.

Exemplo: Um usuário pode ter vários pedidos ->

```
EBOOK FEITO POR LeoMatias.java  
1 @Entity  
2 public class Pedido {  
3     @Id @GeneratedValue  
4     private Long id;  
5     private String produto;  
6  
7     @ManyToOne  
8     private Usuario usuario;  
9 }
```

```
EBOOK FEITO POR LeoMatias.java  
1 @Entity  
2 public class Usuario {  
3     @Id @GeneratedValue  
4     private Long id;  
5     private String nome;  
6  
7     @OneToMany(mappedBy = "usuario")  
8     private List<Pedido> pedidos;  
9 }
```

OBS: Lembre-se de usar `mappedBy` para indicar o lado que "possui" a relação.

Paginação e ordenação com Pageable

Devolvendo dados em partes, do jeito certo

Para evitar retornar muitos registros de uma vez, use *Pageable*.

```
EBOOK FEITO POR LeoMatias.java  
1 @GetMapping  
2 public Page<Usuario> listar(Pageable pageable) {  
3     return repo.findAll(pageable);  
4 }
```

Você pode controlar a URL:

```
bash  
/usuarios?page=0&size=5&sort=nome,asc
```

→ Isso retorna os 5 primeiros usuários ordenados por nome crescente.

Paginação e ordenação com Pageable

Devolvendo dados em partes, do jeito certo
(Pt. 2)

Se quiser configurar um valor padrão de paginação:

```
● ● ● EBOOK FEITO POR LeoMatias.java

1 @Configuration
2 public class ConfigPaginacao extends WebMvcConfigurerAdapter {
3     @Override
4     public void addArgumentResolvers(List<HandlerMethodArgumentResolver> resolvers) {
5         PageableHandlerMethodArgumentResolver resolver = new
6             PageableHandlerMethodArgumentResolver();
7         resolver.setFallbackPageable(PageRequest.of(0, 10));
8         resolvers.add(resolver);
9     }
}
```

PARTE 3

APIs Reais

e

Boas Práticas

SE TORNE UM MESTRE JEDI EM
SPRING BOOT

Separando entidades de DTOs

Mantenha seu modelo limpo e evite vazamentos de dados

Em APIs reais, é uma boa prática usar DTOs (Data Transfer Objects) para evitar expor diretamente as entidades do banco. Isso te dá mais controle sobre o que entra e sai da API.

Exemplo:

```
● ● ● EBOOK FEITO POR LeoMatias.java  
1 public class UsuarioDTO {  
2     private String nome;  
3     private String email;  
4 }
```

No Controller:

```
● ● ● EBOOK FEITO POR LeoMatias.java  
1 @PostMapping  
2 public ResponseEntity<?> criar(@RequestBody UsuarioDTO dto) {  
3     Usuario u = new Usuario();  
4     u.setNome(dto.getNome());  
5     u.setEmail(dto.getEmail());  
6     repo.save(u);  
7     return ResponseEntity.ok("Usuário salvo!");  
8 }
```

Separar entidades de DTOs ajuda na validação, segurança e manutenção do código.

Validando dados com Bean Validation

Garanta que só dados corretos entrem no seu sistema

Com javax.validation, você pode validar automaticamente os dados recebidos. **Exemplo com anotações no DTO:**

```
● ● ● EBOOK FEITO POR LeoMatias.java

1 public class UsuarioDTO {
2     @NotBlank(message = "Nome é obrigatório")
3     private String nome;
4
5     @Email(message = "E-mail inválido")
6     private String email;
7 }
```

No Controller:

```
● ● ● EBOOK FEITO POR LeoMatias.java

1 @PostMapping
2 public ResponseEntity<?> criar(@RequestBody @Valid UsuarioDTO dto) {
3     // salvar usuário
4     return ResponseEntity.ok("OK");
5 }
```

Erro de validação? O Spring devolve um 400 Bad Request com as mensagens!

Tratando erros com @ControllerAdvice

Centralize o tratamento de exceções de forma elegante

Em vez de tratar erro a erro em cada controller, você pode capturar tudo em um lugar só com `@ControllerAdvice`.

```
E-BOOK FEITO POR LeoMatias.java

1  @ControllerAdvice
2  public class TratamentoGlobal {
3
4      @ExceptionHandler(MethodArgumentNotValidException.class)
5      public ResponseEntity<?> handleValidacao(MethodArgumentNotValidException ex) {
6          List<String> erros = ex.getBindingResult()
7              .getFieldErrors()
8              .stream()
9              .map(e -> e.getField() + ": " + e.getDefaultMessage())
10             .toList();
11
12          return ResponseEntity.badRequest().body(erros);
13      }
14
15      @ExceptionHandler(EntityNotFoundException.class)
16      public ResponseEntity<?> naoEncontrado(EntityNotFoundException ex) {
17          return ResponseEntity.status(404).body("Registro não encontrado");
18      }
19 }
```

Agora, toda vez que um erro ocorrer, ele será tratado de forma padronizada e clara.

Documentando sua API

Deixe sua API navegável e comprehensível com uma interface web

O Swagger (via SpringDoc) gera automaticamente uma interface para testar e entender os endpoints da sua API. Adicione no *pom.xml*:

```
...xml  
1 <dependency>  
2   <groupId>org.springdoc</groupId>  
3   <artifactId>springdoc-openapi-ui</artifactId>  
4   <version>1.7.0</version>  
5 </dependency>  
6
```

Acesse no navegador:

<http://localhost:8080/swagger-ui.html>

ou

/swagger-ui/index.html (versões recentes)

Você verá uma tela interativa com todos os endpoints organizados, com exemplos de entrada e saída.

→ Útil para desenvolvedores, testers, e até clientes que querem testar a API.

PARTE 4

Segurança
com
Spring Security

SE TORNE UM MESTRE JEDI EM
SPRING BOOT

Sobre a autenticação e autorização

Entendendo quem é você e o que pode fazer

Autenticação é **quem você é**. Autorização é **o que você pode fazer**. Spring Security oferece uma estrutura completa para tratar as duas. Exemplo de configuração básica com autenticação em memória:

```
● ● ● E-BOOK FEITO POR LeoMatias.java  
1 @Bean  
2 public UserDetailsService userDetailsService() {  
3     UserDetails user = User.withDefaultPasswordEncoder()  
4         .username("admin")  
5         .password("1234")  
6         .roles("ADMIN")  
7         .build();  
8  
9     return new InMemoryUserDetailsManager(user);  
10 }
```

E ao acessar qualquer rota protegida, será necessário login.

→ Comece testando com autenticação em memória antes de usar banco de dados.

Protegendo rotas com roules

Controle o acesso a cada endpoint da API

Você pode restringir o acesso a determinados endpoints com base nas roles (funções) dos usuários.

```
● ● ● E-BOOK FEITO POR LeoMatias.java  
1 @Override  
2 protected void configure(HttpSecurity http) throws Exception {  
3     http  
4         .authorizeRequests()  
5         .antMatchers("/admin/**").hasRole("ADMIN")  
6         .antMatchers("/user/**").hasAnyRole("USER", "ADMIN")  
7         .anyRequest().authenticated()  
8         .and()  
9         .httpBasic();  
10 }
```

Exemplo de Controller:

```
● ● ● E-BOOK FEITO POR LeoMatias.java  
1 @GetMapping("/admin/dados")  
2 public String dadosAdmin() {  
3     return "Somente admin vê isso";  
4 }
```

→ Assim, você define claramente quem pode acessar o quê.

JWT: Autenticação com tokens

Deixe sua API stateless com segurança moderna

Autenticar com JWT (JSON Web Tokens) é ideal para APIs REST, pois evita sessões e facilita integrações com front-ends e mobile.

Fluxo básico:

1. Usuário envia login e senha → recebe token JWT.
2. Frontend envia esse token em cada requisição.
3. Back-end valida o token antes de responder.

Exemplo do corpo do token JWT (de forma simplificada):

```
... json  
1 {  
2   "sub": "usuario123",  
3   "roles": ["ROLE_USER"],  
4   "exp": 1735689600  
5 }
```

O token é assinado e não pode ser alterado sem invalidação.

→ Frameworks como o spring-boot-starter-oauth2-resource-server ajudam na implementação.

Segurança extra: CORS e CSRF

Proteja sua API contra origens maliciosas e ataques invisíveis

1. CORS (Cross-Origin Resource Sharing):

Controla quem pode consumir sua API em um navegador. Útil ao integrar com front-ends hospedados em domínios diferentes.

```
● ● ● E-BOOK FEITO POR LeoMatias.java  
1 @Bean  
2 public WebMvcConfigurer corsConfigurer() {  
3     return new WebMvcConfigurer() {  
4         @Override  
5         public void addCorsMappings(CorsRegistry registry) {  
6             registry.addMapping("/**")  
7                 .allowedOrigins("http://meu-frontend.com")  
8                 .allowedMethods("GET", "POST", "PUT", "DELETE");  
9         }  
10    };  
11 }
```

2. CSRF (Cross-Site Request Forgery):

É importante para apps web com sessão. Para APIs REST com JWT, normalmente desativamos:

```
● ● ● E-BOOK FEITO POR LeoMatias.java  
1 http.csrf().disable();
```

→ Desative CSRF somente se sua API for stateless e usar tokens com autenticação.

PARTES

Testes

e

Qualidade de
Código

SE TORNE UM MESTRE JEDI EM
SPRING BOOT

Testes unitários com JUnit e Mockito

Teste seus métodos sem depender de nada externo

Os **testes unitários** verificam se métodos individuais funcionam corretamente. No Spring Boot, usamos **JUnit** como framework de testes e **Mockito** para simular comportamentos de dependências. Exemplo real:

→ Vamos testar um serviço que calcula descontos:

```
● ● ● E-BOOK FEITO POR LeoMatias.java  
1 @Service  
2 public class DescontoService {  
3     public double aplicarDesconto(double valor, double percentual) {  
4         return valor - (valor * percentual);  
5     }  
6 }
```

Testando com JUnit:

```
● ● ● E-BOOK FEITO POR LeoMatias.java  
1 @SpringBootTest  
2 public class DescontoServiceTest {  
3  
4     @Autowired  
5     private DescontoService descontoService;  
6  
7     @Test  
8     public void deveAplicarDescontoCorretamente() {  
9         double resultado = descontoService.aplicarDesconto(100.0, 0.1);  
10        Assertions.assertEquals(90.0, resultado);  
11    }  
12 }
```

Testes unitários com JUnit e Mockito

Teste seus métodos sem depender de nada externo (Pt. 2)

Exemplo com Mockito:

→ Suponha que o DescontoService chame um repositório:

```
● ● ● E-BOOK FEITO POR LeoMatias.java  
1 public class DescontoService {  
2     private final ClienteRepository repo;  
3  
4     public DescontoService(ClienteRepository repo) {  
5         this.repo = repo;  
6     }  
7  
8     public double aplicarDescontoSeVip(Long idCliente, double valor) {  
9         Cliente cliente = repo.findById(idCliente).orElseThrow();  
10        return cliente.isVip() ? valor * 0.9 : valor;  
11    }  
12 }
```

Teste com Mockito:

```
● ● ● E-BOOK FEITO POR LeoMatias.java  
1 @ExtendWith(MockitoExtension.class)  
2 public class DescontoServiceTest {  
3  
4     @Mock  
5     private ClienteRepository repo;  
6  
7     @InjectMocks  
8     private DescontoService service;  
9  
10    @Test  
11    void deveAplicarDescontoParaClienteVip() {  
12        Cliente vip = new Cliente("João", true);  
13        Mockito.when(repo.findById(1L)).thenReturn(Optional.of(vip));  
14  
15        double valorFinal = service.aplicarDescontoSeVip(1L, 100.0);  
16  
17        Assertions.assertEquals(90.0, valorFinal);  
18    }  
19 }
```

Testando controllers

Teste a camada web sem precisar subir tudo

O `@WebMvcTest` permite testar os controllers isoladamente, sem carregar todo o contexto da aplicação. Junto com `MockMvc`, conseguimos simular requisições HTTP reais. **Exemplo real:**

→ Controller de cadastro de cliente:

```
● ● ● E-BOOK FEITO POR LeoMatias.java

1 @RestController
2 @RequestMapping("/clientes")
3 public class ClienteController {
4     @PostMapping
5     public ResponseEntity<String> cadastrar(@RequestBody Cliente cliente) {
6         return ResponseEntity.status(HttpStatus.CREATED).body("Cliente cadastrado");
7     }
8 }
```

Testando controllers

Teste a camada web sem precisar subir tudo (Pt. 2)

Teste com MockMvc:

```
● ● ● E-BOOK FEITO POR LeoMatias.java

1 @WebMvcTest(ClienteController.class)
2 public class ClienteControllerTest {
3
4     @Autowired
5     private MockMvc mockMvc;
6
7     @Test
8     void deveCadastrarCliente() throws Exception {
9         String clienteJson = """
10             {
11                 "nome": "João",
12                 "email": "joao@email.com"
13             }
14             """;
15
16         mockMvc.perform(post("/clientes")
17                         .contentType(MediaType.APPLICATION_JSON)
18                         .content(clienteJson))
19                         .andExpect(status().isCreated())
20                         .andExpect(content().string("Cliente cadastrado"));
21     }
22 }
```

Esse teste simula uma requisição **HTTP POST** e verifica se a resposta está correta — sem subir banco, serviços ou repositórios.

Teste de repositórios em memória

Testes reais de banco, sem depender do seu ambiente

O `@DataJpaTest` cria um ambiente de teste leve, usando banco de dados em memória (geralmente H2) e apenas os componentes do Spring Data JPA. Exemplo com repositório de cliente:

```
● ● ● E-BOOK FEITO POR LeoMatias.java

1 @Entity
2 public class Cliente {
3     @Id @GeneratedValue
4     private Long id;
5     private String nome;
6     private String email;
7
8     // getters, setters, construtores
9 }
10
11 public interface ClienteRepository extends JpaRepository<Cliente, Long> {
12     Optional<Cliente> findByEmail(String email);
13 }
```

Teste de repositórios em memória

Testes reais de banco, sem depender do seu ambiente (Pt. 2)

Testando com banco em memória:

```
● ● ● E-BOOK FEITO POR LeoMatias.java

1 @DataJpaTest
2 public class ClienteRepositoryTest {
3
4     @Autowired
5     private ClienteRepository repository;
6
7     @Test
8     void deveEncontrarClientePorEmail() {
9         Cliente c = new Cliente("João", "joao@email.com");
10        repository.save(c);
11
12        Optional<Cliente> resultado = repository.findByEmail("joao@email.com");
13
14        Assertions.assertTrue(resultado.isPresent());
15        Assertions.assertEquals("João", resultado.get().getNome());
16    }
17 }
```

Esse tipo de teste é ótimo para garantir que suas consultas personalizadas funcionem de verdade.

PARTE 6

Deploy
e
Produção

SE TORNE UM MESTRE JEDI EM
SPRING BOOT

Preparando o app para produção

De app local para um serviço estável e confiável

Antes de fazer o deploy, é essencial garantir que sua aplicação esteja pronta para produção. Isso inclui:

- Uso do perfil prod
- Logs bem configurados
- Banco de produção separado
- Senhas e chaves sensíveis fora do código

Exemplo prático:

```
... properties  
1 # application-prod.properties  
2 spring.datasource.url=jdbc:postgresql://localhost:5432/app_prod  
3 spring.datasource.username=${DB_USER}  
4 spring.datasource.password=${DB_PASS}  
5 spring.jpa.hibernate.ddl-auto=validate  
6 spring.jpa.show-sql=false  
7 server.port=8080
```

DICA: Nunca use `create` ou `update` em produção no `ddl-auto`. Prefira `validate` ou use migrations com Flyway ou Liquibase.

Configurando perfis de ambiente

Um app, múltiplos contextos

O Spring Boot permite configurar diferentes ambientes usando o conceito de "**profiles**". Você pode ter:

- application-dev.properties
- application-prod.properties
- application-test.properties

E ativá-los assim:

```
... bash  
# Ambiente de produção  
java -Dspring.profiles.active=prod -jar minha-aplicacao.jar
```

Ou no application.properties padrão:

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

Configurando perfis de ambiente

Um app, múltiplos contextos (Pt. 2)

Exemplo de uso condicional com `@Profile`:

```
● ● ● E-BOOK FEITO POR LeoMatias.java  
1 @Service  
2 @Profile("dev")  
3 public class DevEmailService implements EmailService {  
4     public void enviar(String msg) {  
5         System.out.println("Simulado envio de e-mail: " + msg);  
6     }  
7 }
```

```
● ● ● E-BOOK FEITO POR LeoMatias.java  
1 @Service  
2 @Profile("prod")  
3 public class SmtpEmailService implements EmailService {  
4     public void enviar(String msg) {  
5         // envia e-mail de verdade  
6     }  
7 }
```

Monitoramento com o Actuator

Veja o que seu app está fazendo sem abrir o código

O **Spring Boot Actuator** fornece endpoints para monitorar a saúde, métricas e status da aplicação. Então, habilitando-o:

```
...                                     xml  
1 <!-- pom.xml -->  
2 <dependency>  
3   <groupId>org.springframework.boot</groupId>  
4   <artifactId>spring-boot-starter-actuator</artifactId>  
5 </dependency>
```

Alguns endpoints comuns:

- `/actuator/health` → Verifica se o app está vivo
- `/actuator/info` → Mostra dados customizados
- `/actuator/metrics` → Coleta de métricas

Monitoramento com o Actuator

Veja o que seu app está fazendo sem abrir o código
(Pt. 2)

Então, personalizando:

```
... properties
1 management.endpoints.web.exposure.include=health,info,metrics
2 management.endpoint.health.show-details=always
```

DICA: Você pode integrar o Actuator com **Prometheus**, **Grafana** e serviços como **New Relic** ou **Datadog** para monitoramento avançado.

Containerizando o app com o Docker

Sua aplicação rodando em qualquer lugar, sempre igual

O Docker permite empacotar sua aplicação com tudo que ela precisa para rodar, em qualquer ambiente. Criando um Dockerfile simples:

```
... dockerfile
1 FROM openjdk:17-jdk-alpine
2 COPY target/meuapp.jar app.jar
3 ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Build e execução:

```
... bash
# Compila a aplicação
./mvnw clean package -DskipTests

# Constrói a imagem Docker
docker build -t meuapp .

# Roda a imagem
docker run -p 8080:8080 meuapp
```

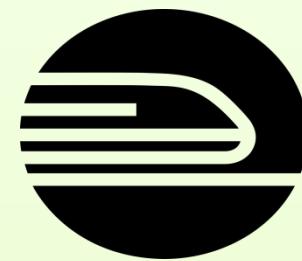
Deploy na nuvem

Publicando seu app de verdade

Existem várias opções gratuitas e fáceis para deploy de apps Java. As mais populares são:

→ Railway

- Crie um projeto no Railway
- Suba seu projeto no GitHub
- Conecte Railway ao seu repositório
- Configure as variáveis de ambiente (como SPRING_PROFILES_ACTIVE=prod)
- Ele cuida do build e deploy pra você



→ Render

- Ótimo para APIs com Docker
- Também pode usar Java direto com comandos personalizados



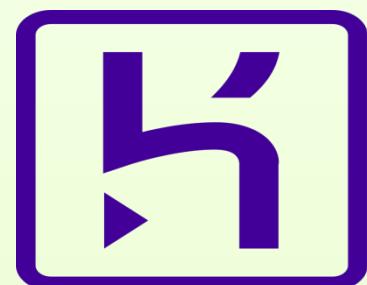
```
... bash  
./mvnw spring-boot:run
```

Deploy na nuvem

Publicando seu app de verdade (Pt. 2)

→ Heroku (com Docker ou JAR)

```
... bash  
# Login e criação  
heroku login  
heroku create nome-app  
  
# Deploy de JAR  
heroku deploy:jar target/meuapp.jar --app nome-app
```



Variáveis de ambiente:

- Todas essas plataformas permitem definir variáveis como:

```
... env  
SPRING_PROFILES_ACTIVE=prod  
DB_URL=  
DB_USER=  
DB_PASS=
```

→ DICA: Use [fly.io](#), [Vercel](#) (frontend) ou [DigitalOcean](#) para opções mais profissionais e com controle total do ambiente.

PARTE EXTRA

Apêndices



Extras

SE TORNE UM MESTRE JEDI EM
SPRING BOOT

Boas práticas em aplicações

O que diferencia um código que funciona de um código que impressiona

Organização de pacotes é bastante essencial ->

- Use o padrão por domínio (e não apenas por tipo):

```
estrutura

com.seuprojeto
└── usuario
    ├── Usuario.java
    ├── UsuarioController.java
    ├── UsuarioService.java
    ├── UsuarioRepository.java
    └── UsuarioDTO.java
```

Aplique a injeção de dependência ->

- Prefira **construtores** ao invés de `@Autowired` em atributos:

```
E-BOOK FEITO POR LeoMatias.java

1 @Service
2 public class ProdutoService {
3     private final ProdutoRepository repo;
4
5     public ProdutoService(ProdutoRepository repo) {
6         this.repo = repo;
7     }
8 }
```

Sobre DTOs e validação ->

- Evite expor entidades diretamente. Use DTOs para entrada e saída.
- Valide entradas com `@Valid`, `@NotBlank`, etc.

Ferramentas úteis para o dia a dia

Superpoderes para desenvolvedores Spring

Uma seleção de ferramentas que economizam tempo, reduzem erros e aumentam sua produtividade no desenvolvimento com Spring Boot.

→ TESTES

- **Testcontainers**: para rodar banco real em container durante testes
- **Postman** ou **Insomnia**: testar endpoints REST
- **Mockoon**: simular APIs externas

→ PRODUTIVIDADE

- **Spring Boot DevTools**: recarregamento automático durante desenvolvimento
- **Lombok**: reduz boilerplate (@Getter, @Builder, @AllArgsConstructor)
- **Spring Initializr**: <https://start.spring.io> – crie projetos rapidamente

→ MONITORAMENTO E LOG

- **Spring Boot Admin**: painel visual para monitorar múltiplas instâncias
- **Logback** com ELK Stack (Elasticsearch + Logstash + Kibana)
- **Actuator + Prometheus + Grafana** para métrica e alertas

Próximos passos para a glória

De Padawan a Mestre Jedi: A estrada para o domínio do ecossistema Spring

Um guia de especialização contínua para quem quer se destacar no mercado com conhecimento técnico aprofundado.

► 1. Domine segurança com Spring Security

- Autenticação com JWT
- OAuth2 e login social
- RBAC (controle de acesso por papéis)

► 2. Aprenda sobre arquitetura avançada

- Microsserviços com Spring Cloud
- API Gateway com Spring Cloud Gateway
- Circuit Breakers com Resilience4j

► 3. Explore mensageria e eventos

- RabbitMQ / Kafka
- Event-driven architecture
- Sagas e orquestração de fluxos distribuídos

► 4. Banco de dados avançado

- Flyway ou Liquibase para versionamento
- Cache com Redis
- Elasticsearch para busca full-text



AGRADECIMENTO

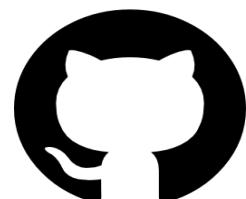
SE TORNE UM MESTRE JEDI EM
SPRING BOOT

OBRIGADO POR LER ATÉ AQUI!

E chega ao fim a nossa jornada estelar...

Este E-book foi gerado por IA (ChatGPT) e diagramado por um humano (eu mesmo, Léo Matias). O passo a passo se encontra no Github do professor da DIO. O link para meu Github pessoal também se encontrará linkado.

Esse conteúdo foi gerado com fins didáticos de construção, não foi realizado uma validação cuidadosa humana no conteúdo e pode conter erros gerados por uma IA. Espero que tenham gostado, até mais!



<https://github.com/felipeAguiarCode/prompts-recipe-to-create-a-ebook>

<https://github.com/leomatiazzz>