

# MISSAO BACKEND

Desvendando Java e Spring Boot



Debora Paiva

# 01

## Preparação do Ambiente e Instalações Necessárias

Bem-vindo ao primeiro passo da sua jornada no desenvolvimento de back-end com Java e Spring Boot!

Antes de mergulharmos na construção de aplicações, vamos garantir que seu ambiente esteja configurado corretamente. Nesta seção, você encontrará tudo o que precisa para deixar seu sistema pronto para programar com eficiência, desde as instalações básicas até a configuração das ferramentas essenciais.

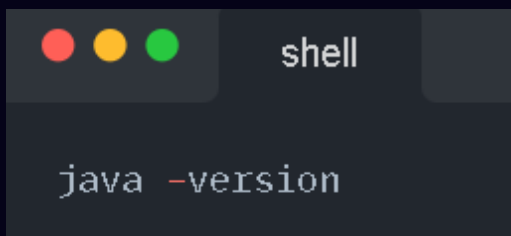
# Instalando o Java Development Kit (JDK)

O Java é a base do Spring Boot, então o primeiro passo é instalar o Java Development Kit (JDK).

**Passo 1:** Acesse o site oficial do [Oracle JDK](#) ou [OpenJDK](#) e baixe a versão mais recente.

**Passo 2:** Siga as instruções para instalar o JDK em seu sistema (Windows, macOS ou Linux).

**Passo 3:** Para verificar se a instalação foi bem-sucedida, abra o terminal ou prompt de comando e digite:

A screenshot of a terminal window with a dark background. The title bar at the top shows three colored window control buttons (red, yellow, green) on the left and the word "shell" in the center. The terminal area displays the command `java -version` in a light-colored monospaced font.

```
shell  
  
java -version
```

**Dica:** O Spring Boot é compatível com o JDK 8 e versões superiores, mas é recomendado usar o JDK 11 ou JDK 17, já que são versões de Long-Term Support (LTS).

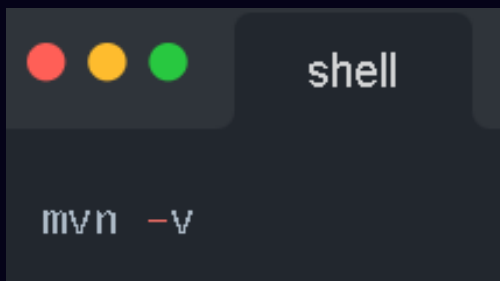
# Instalando o Maven

O Maven é uma ferramenta de gerenciamento de dependências e build, essencial para gerenciar bibliotecas e configurar o ambiente de desenvolvimento de maneira eficiente.

**Passo 1:** Baixe o Maven do [site oficial](#).

**Passo 2:** Extraia o arquivo baixado e adicione o caminho da pasta bin ao seu sistema.

**Passo 3:** Verifique a instalação digitando no terminal:



**Isso deve exibir a versão do Maven, indicando que a instalação foi concluída com sucesso.**

# Instalando uma IDE

**Para um desenvolvimento mais prático, você precisará de uma IDE (Integrated Development Environment) para escrever e executar seu código Java. As IDEs recomendadas para Java e Spring Boot são:**

- IntelliJ IDEA (versão Community é gratuita e muito popular entre desenvolvedores Java)
- Eclipse IDE (gratuita e amplamente usada)
- Spring Tool Suite (STS) (desenvolvida especificamente para o ecossistema Spring)

**Passo 1:** Baixe e instale a IDE de sua escolha.

**Passo 2:** No primeiro uso, configure o caminho do JDK nas preferências da IDE para que ela reconheça seu ambiente Java.

# Preparando o Spring Boot

Com o JDK, Maven e IDE instalados, agora você está pronto para configurar o Spring Boot.

**Passo 1:** Acesse o [Spring Initializr](#), uma ferramenta oficial que facilita a criação de novos projetos Spring Boot. Selecione as configurações:

Project: Maven

Language: Java

Spring Boot Version: Escolha a mais recente LTS

Dependencies: Adicione as dependências iniciais, como Spring Web (para APIs REST) e outras conforme necessário.

**Passo 2:** Baixe o projeto gerado e importe-o para sua IDE.

# 02

## Fundamentos Essenciais para um Sistema Backend

Neste capítulo, vamos explorar conceitos fundamentais para construir um sistema backend robusto e escalável. Usaremos como base a construção de um sistema web simples para cadastro de documentos. Vamos abordar POO (Programação Orientada a Objetos), SOLID, HTTP e Camadas, e uma introdução ao CI/CD.

# Programação Orientada a Objetos (POO)

A POO é uma abordagem de desenvolvimento onde se organiza o código em objetos. Cada objeto contém dados (atributos) e comportamentos (métodos).

## Princípios Básicos da POO

- **Classes e Objetos:** Uma classe é um molde para criar objetos. Por exemplo, a classe `Usuario` define o que todo usuário possui, como nome, email e senha. Cada novo usuário é um objeto criado a partir dessa classe.
- **Herança:** Permite que uma classe "filha" herde características de uma classe "pai". Exemplo: `UsuarioComum` e `UsuarioAdm` podem herdar atributos e métodos da classe `Usuario`.
- **Encapsulamento:** Controla o acesso aos dados de um objeto, garantindo segurança e organização. No nosso sistema, senha deve ser acessível apenas através de métodos seguros, como `alterarSenha()`.
- **Polimorfismo:** Permite usar um método de forma genérica, adaptando o comportamento. No caso, tanto `UsuarioComum` quanto `UsuarioAdm` podem ter o método `login()`, mas com permissões e funcionalidades ajustadas para cada tipo de usuário.

**Exemplo Prático:** Imagine que criamos uma classe `Usuario` para estruturar os dados básicos de todos os usuários. A partir dela, podemos criar `UsuarioComum` e `UsuarioAdm`, com funcionalidades específicas para cada um.



# SOLID: Princípios para um Código Limpo e Organizado

Os princípios SOLID ajudam a manter o código organizado e fácil de manter. Veja cada princípio aplicado ao nosso sistema de cadastro de documentos:

- **Single Responsibility Principle (SRP):** Cada classe deve ter uma única responsabilidade. Por exemplo, DocumentoService deve se concentrar na lógica de documentos, enquanto UsuarioService cuida da lógica de usuários.
- **Open/Closed Principle (OCP):** As classes devem ser fáceis de estender sem modificar o código existente. Podemos criar novas funcionalidades sem alterar o código base, mantendo a estabilidade do sistema.
- **Liskov Substitution Principle (LSP):** Classes derivadas devem ser substituíveis pela classe base. Nosso UsuarioComum deve funcionar sempre que um Usuario for necessário, sem causar problemas.
- **Interface Segregation Principle (ISP):** Crie interfaces específicas para cada função. Exemplo: podemos ter uma interface DocumentoActions com métodos específicos para criação e edição de documentos, em vez de uma interface única para tudo.
- **Dependency Inversion Principle (DIP):** As classes devem depender de abstrações, não de implementações concretas. No Spring Boot, usamos injeção de dependência para trabalhar com interfaces, facilitando testes e substituições de componentes.

**Exemplo Prático:** No nosso sistema, UsuarioService e DocumentoService têm responsabilidades bem definidas. A utilização de interfaces permite que substituamos as implementações desses serviços sem impacto no sistema.

# HTTP e Camadas no Backend

**Para comunicação entre o cliente (navegador) e o servidor (onde o sistema backend roda), usamos o protocolo HTTP. O HTTP permite enviar e receber dados de forma estruturada.**

## **Estrutura do Sistema em Camadas**

No desenvolvimento backend, organizamos o sistema em camadas, cada uma com uma função específica:

- **Controlador (Controller):** Recebe e responde às solicitações HTTP. Exemplo: quando um usuário envia uma solicitação para criar um documento, o DocumentoController interpreta a solicitação e chama os serviços apropriados.
- **Serviço (Service):** Contém a lógica de negócios, ou seja, define o que acontece em cada ação. Exemplo: DocumentoService decide como criar, alterar ou excluir documentos.
- **Repositório (Repository):** Lida com a comunicação com o banco de dados. Ele grava, lê, atualiza ou remove dados do banco.

**Exemplo Prático:** Quando o cliente envia uma solicitação para criar um documento, o DocumentoController recebe essa solicitação, o DocumentoService valida os dados e chama o DocumentoRepository para armazená-los no banco de dados.

# CI/CD: Integração e Entrega Contínua

CI/CD é uma metodologia para automatizar testes e entregas de código, garantindo que cada atualização seja testada e esteja pronta para ser lançada.

- **CI (Continuous Integration):** Sempre que uma nova parte do código é desenvolvida, ela é automaticamente testada e integrada ao sistema. Isso evita problemas e integra mudanças de forma mais rápida.
- **CD (Continuous Deployment/Delivery):** Com a entrega contínua, o código validado é automaticamente disponibilizado para produção (ambiente final), facilitando atualizações rápidas e seguras.

**Exemplo Prático:** No nosso sistema, cada vez que um novo recurso, como uma função de "esqueci minha senha", é adicionado, ele passa por testes automáticos. Se aprovado, é automaticamente implementado no ambiente de produção, garantindo que tudo esteja em ordem e atualizado.

# 03

## Estruturando o Sistema

Neste capítulo, vamos colocar em prática os conceitos aprendidos, criando as bases do nosso sistema de cadastro de documentos com Spring Boot. Vamos explorar as classes essenciais como Entity, Controller, DTO, Repository, entre outras.

Esse guia fornecerá um passo a passo simples para estruturar um backend do zero.

# Criando a Classe Entity

A Entity representa a estrutura de dados do nosso sistema e mapeia as tabelas no banco de dados.

No exemplo, criemos uma classe Documento com campos como numeroDeContrato, nomeCliente, celularCliente, etc.

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

@Entity
public class Documento {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String numeroDeContrato;
    private String nomeCliente;
    private String celularCliente;
    private Double valorContrato;
    private String inicioVigencia;
    private String terminoVigencia;
    private String emailCliente;

    // Getters e Setters
```

Aqui, **@Entity** indica que esta classe será uma tabela no banco, e **@Id** define a chave primária. Essa classe Documento armazena dados essenciais para cada contrato.

# Definindo o Repository

O Repository é a camada responsável pela comunicação com o banco de dados. Em Spring Boot, usamos o JpaRepository para criar repositórios com operações básicas, como salvar e buscar dados.

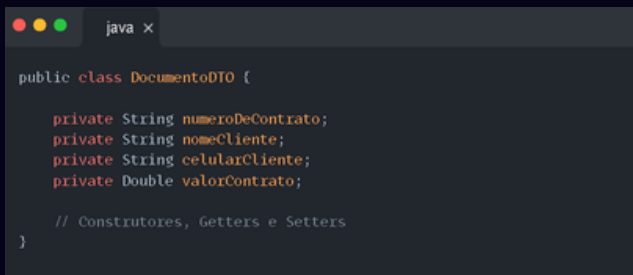
```
import org.springframework.data.jpa.repository.JpaRepository;

public interface DocumentoRepository extends JpaRepository<Documento,
Long> {
    // Métodos personalizados, se necessário
}
```

Com essa interface DocumentoRepository, temos acesso a métodos como save(), findById(), findAll(), entre outros, sem precisar escrever SQL.

# Criando o DTO (Data Transfer Object)

O DTO é uma classe que simplifica os dados enviados entre o frontend e o backend. Isso ajuda a não expor diretamente a entidade.

A screenshot of a Java IDE window titled 'java x'. The code defines a public class 'DocumentoDTO' with four private attributes: 'numeroDeContrato' (String), 'nomeCliente' (String), 'celularCliente' (String), and 'valorContrato' (Double). Below the attributes, there is a comment '// Construtores, Getters e Setters' followed by a closing curly brace '}'.

```
public class DocumentoDTO {  
  
    private String numeroDeContrato;  
    private String nomeCliente;  
    private String celularCliente;  
    private Double valorContrato;  
  
    // Construtores, Getters e Setters  
}
```

Esse DocumentoDTO contém apenas os dados que queremos expor ou receber, deixando a estrutura mais limpa e segura.

# Configurando o Controlle

O Controller é responsável por definir os endpoints da aplicação e direcionar as solicitações para o serviço apropriado. No caso, vamos criar o DocumentoController para gerenciar documentos.

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

import java.util.List;

@RestController
@RequestMapping("/documentos")
public class DocumentoController {

    @Autowired
    private DocumentoService documentoService;

    @PostMapping
    public DocumentoDTO criarDocumento(@RequestBody DocumentoDTO documentoDTO) {
        return documentoService.salvarDocumento(documentoDTO);
    }

    @GetMapping
    public List<DocumentoDTO> listarDocumentos() {
        return documentoService.listarTodos();
    }
}
```

Aqui, o `@RestController` define que esta classe é um controlador REST, e `@RequestMapping("/documentos")` define a URL base para os endpoints.

O método `criarDocumento` usa `@PostMapping` para receber um novo documento e `listarDocumentos` usa `@GetMapping` para listar todos os documentos cadastrados.



# Implementando o Service

A camada de serviço contém a lógica de negócios e chama o repositório para acessar o banco de dados. Criamos o DocumentoService para implementar as operações.

```
java

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.stream.Collectors;

@Service
public class DocumentoService {

    @Autowired
    private DocumentoRepository documentoRepository;

    public DocumentoDTO salvarDocumento(DocumentoDTO documentoDTO) {
        Documento documento = new Documento();
        documento.setNumeroDeContrato(documentoDTO.getNumeroDeContrato());
        documento.setNomeCliente(documentoDTO.getNomeCliente());
        documento.setCelularCliente(documentoDTO.getCelularCliente());
        documento.setValorContrato(documentoDTO.getValorContrato());

        documentoRepository.save(documento);
        return documentoDTO;
    }

    public List<DocumentoDTO> listarTodos() {
        return documentoRepository.findAll()
            .stream()
            .map(doc -> new DocumentoDTO(doc.getNumeroDeContrato(), doc.getNomeCliente(),
                doc.getCelularCliente(), doc.getValorContrato()))
            .collect(Collectors.toList());
    }
}
```

Esse DocumentoService possui métodos para salvar e listar documentos. Ele converte o DocumentoDTO em Documento para salvar no banco e, ao buscar os dados, converte Documento de volta para DocumentoDTO.