



Manual da UFCD: 10791 - Desenvolvimento de aplicações web em JAVA

Luís Cunha
27 de novembro de 2024

Sumário Executivo

Este manual foi desenvolvido como suporte à Unidade de Formação de Curta Duração (UFCD) 10791, intitulada "*Desenvolvimento de Aplicações Web em Java*". Tem como objetivo fornecer aos formandos uma visão abrangente e prática sobre o desenvolvimento de aplicações web modernas utilizando a linguagem Java, com foco em frameworks como o Spring Boot e o Vaadin Flow.

Ao longo dos capítulos, são abordados conceitos fundamentais, desde as bases da linguagem Java e da Programação Orientada por Objetos (POO) até tópicos avançados como a implementação de controladores REST, acesso a bases de dados com Spring Data JPA, integração com o Vaadin para interfaces ricas, e segurança com Spring Security. O manual combina teoria e prática, permitindo que os formandos desenvolvam competências robustas para criar aplicações web eficientes e seguras, seguindo boas práticas de engenharia de software.

Este documento destina-se a apoiar a aprendizagem contínua, sendo um guia completo para estudantes e profissionais que desejam aprofundar os seus conhecimentos em desenvolvimento web com Java.

Índice

Capítulo 1. Introdução à Programação Web com Java	9
História da Linguagem Java	9
Diferenças entre Java SE e Jakarta EE (antiga Java EE):	9
Resumo da Transição do Java EE para a Jakarta EE	11
Surgimento da Spring Framework.....	13
Evolução com o Spring Boot.....	13
Convivência entre Spring Boot e Jakarta EE	14
Comparação entre Spring, Spring Boot e Jakarta EE	15
Resumo:	16
Capítulo 2. Bases de Java e de Programação Orientada por Objetos	17
Motivos para aprender Java.....	17
Características principais do Java	17
Configuração do ambiente de desenvolvimento no Visual Studio Code	17
Primeiro programa em Java: "Olá, Mundo!"	19
Desafios	20
Fundamentos da Linguagem Java	20
Introdução à Programação Orientada por Objetos (POO)	26
Principais Conceitos de POO	27
Criar a Primeira Classe Java.....	29
Resumo e Exercícios	31
Estruturas de Dados e Coleções em Java	32
Manipulação de Ficheiros	38
Introdução às Funcionalidades Modernas do Java	43
Capítulo 3. Introdução ao Spring Boot	49
Conceitos Fundamentais e Benefícios.....	49
Diferenças entre Spring Framework e Spring Boot	50
Capítulo 4. Configuração do Ambiente de Desenvolvimento.....	52

Instalação do JDK, Maven e Visual Studio Code	52
Configuração das Extensões Necessárias no VS Code	54
Capítulo 5. Criação de um Projeto Spring Boot com Maven	57
Utilização do Spring Initializr para Gerar o Projeto	58
Importação e Configuração no VS Code.....	59
Outra forma de começar um projeto Spring Boot	60
Capítulo 6. Estrutura de um Projeto Spring Boot	62
Análise das Pastas e Ficheiros Gerados	62
Função de Cada Componente na Aplicação	63
Capítulo 7. Configuração de Propriedades da Aplicação no Spring Boot.....	64
Utilização do Ficheiro application.properties ou application.yml.....	64
Definição de Portas, Perfis e Outras Configurações Essenciais	65
Capítulo 8. Criação de Controladores REST no Spring Boot	68
Implementação de Endpoints Utilizando @RestController	68
Mapeamento de URLs com @RequestMapping e @GetMapping.....	69
Mapeamento de Variáveis de Caminho com @PathVariable	70
Mapeamento de Parâmetros de Requisição com @RequestParam	71
Exercício 8.1.....	72
Passo 1: Criar o Projeto com Spring Initializr	72
Passo 2: Abrir o Projeto no VS Code	72
Estrutura do Diretório.....	73
Resumo da Organização	74
Passo 3: Criar os Ficheiros do Projeto	74
Passo 4: Executar a Aplicação.....	78
Passo 5: Testar os Endpoints com o Postman	78
Dica Adicional	81
Capítulo 9. Introdução ao Spring Data JPA	82
Configuração de Acesso a Bases de Dados Relacionais	82
Criação de Entidades e Repositórios	84
Criação e Teste dos Outros Ficheiros	85
Exercício 9.1.....	87
1. Criar o Projeto com Spring Initializr	87

2. Abrir o Projeto no VS Code	89
3. Configurar a Base de Dados H2	89
4. Criar as Classes Necessárias	90
5. Testar o Projeto.....	94
Notas Finais	98
Capítulo 10. Injeção de Dependências e Componentes no Spring Boot.....	100
Conceito de Inversão de Controlo (IoC)	100
Utilização de Anotações como @Autowired e @Component	100
Capítulo 11. Testes Unitários com Spring Boot.....	104
Configuração do Ambiente de Testes	104
Criação de Testes para Controladores e Serviços	105
Capítulo 12. Princípios de Engenharia de Software no Spring Boot	109
Aplicação de Boas Práticas e Padrões de Design.....	109
Separação de Preocupações e Organização Modular do Código	110
Capítulo 13. Expansão sobre Spring Boot e Bases de Dados.....	111
Introdução ao ORM no Spring Boot	111
Definição de Entidades com POJOs no Spring Boot	116
Criação de Repositórios (Repositories) no Spring Boot	121
Definir Serviços no Spring Boot	126
Papel dos Serviços na Arquitetura MVC.....	132
Definição de Endpoints para Operações CRUD	133
Validação de Dados em Spring Boot	135
Paginação e ordenação	140
Criação de Testes para Repositórios, Serviços e Controladores REST	144
Eficiência e Desempenho com Spring Boot	147
Conclusão sobre Bases de Dados e ORM em Spring Boot.....	150
Capítulo 14. Segurança em Spring	151
Introdução ao Spring Security e Implementação Básica	151
Configuração Inicial do Spring Security.....	152
Implementação de Autenticação Simples	153
Vaadin Security e sua relação com o Spring Security.....	159
Capítulo 15. Interface Web com Vaadin Flow.....	160

Introdução ao Vaadin	160
O que é possível fazer com o Vaadin Flow	161
Configuração de uma Aplicação Vaadin Flow com Spring Boot	163
Componentes Básicos.....	167
Navegação e Layout Principal.....	169
Implementação de Manipuladores de Eventos	170
Filtragem e Debouncing de Eventos	172
Definição de Rotas com @Route	176
Navegação entre Vistas.....	176
Gestão de Recursos Estáticos em Vaadin	178
Eventos do Ciclo de Vida da Navegação.....	182
Redirecionamento e Encaminhamento durante a Navegação	183
Tratamento de Exceções de Navegação	184
Utilização da Anotação @PageTitle	187
Definição Dinâmica do Título da Página	187
Glossário de termos	189
Índice Remissivo	198
Bibliografia.....	198
Sugestões de Recursos Adicionais para Aprofundamento	199

Condições de utilização do manual

Este manual, intitulado "*Desenvolvimento de Aplicações Web em Java*", foi elaborado por Luís Simões da Cunha e destina-se exclusivamente ao uso no contexto da Unidade de Formação de Curta Duração (UFCD) 10791, promovida pela Empresa EISnt.

Todos os direitos sobre o conteúdo deste manual, incluindo texto, exemplos de código, e referências, estão reservados ao autor. É expressamente proibida a reprodução, distribuição, partilha ou utilização, parcial ou integral, deste material fora do âmbito específico da UFCD 10791, salvo autorização prévia, por escrito, do autor ou da Empresa EISnt.

Os formandos têm permissão para utilizar este manual como recurso de aprendizagem, exclusivamente durante o período da formação, podendo consultá-lo para fins educacionais e pessoais relacionados com a UFCD 10791. No entanto, qualquer uso comercial ou fora do contexto educativo da formação será considerado uma violação dos direitos de autor.

Agradece-se o respeito pelas condições de utilização estabelecidas, de modo a assegurar a integridade do material e a sua disponibilização para futuras formações.

Objetivos

Este manual foi concebido como um recurso pedagógico para apoiar os formandos na Unidade de Formação de Curta Duração (UFCD) 10791, "Desenvolvimento de Aplicações Web em Java". O principal objetivo é proporcionar um texto de suporte detalhado, que complementa os exemplos de código disponíveis no repositório **GitHub** associado à formação.

Os **exemplos** apresentados ao longo do manual são explicados cuidadosamente, com instruções claras e organizadas para que os formandos possam reconstruí-los de raiz. Adicionalmente, cada capítulo inclui links para **vídeos curtos**, onde o formador demonstra, passo-a-passo, a resolução de cada exercício. Estes vídeos servem como um complemento visual e prático, ajudando a consolidar os conceitos e metodologias apresentados no texto.

Desta forma, o manual promove uma aprendizagem integrada e estruturada, facilitando a transição entre a teoria e a prática, e garantindo que os formandos adquiram competências sólidas em desenvolvimento web com Java.

Capítulo 1. Introdução à Programação Web com Java

O Java tem desempenhado um papel fundamental no desenvolvimento de aplicações web, oferecendo uma plataforma robusta e versátil para a criação de soluções escaláveis e seguras. A sua filosofia "Write Once, Run Anywhere" permite que aplicações desenvolvidas em Java sejam executadas em diversos ambientes, tornando-o uma escolha popular entre os desenvolvedores.

História da Linguagem Java

O Java foi concebido em 1991 por uma equipa da Sun Microsystems liderada por James Gosling, Mike Sheridan e Patrick Naughton, no âmbito do "Projeto Green". Inicialmente, o objetivo era desenvolver uma linguagem para dispositivos digitais, como televisores e set-top boxes. A linguagem foi inicialmente denominada "Oak", em referência a um carvalho que se encontrava fora do escritório de Gosling. Mais tarde, foi renomeada para "Java" devido a questões de marca registada.

Em 1995, o Java foi oficialmente lançado com o lema "Write Once, Run Anywhere" (Escreva uma vez, execute em qualquer lugar), destacando a sua portabilidade e independência de plataforma. Esta característica foi possibilitada pela Máquina Virtual Java (JVM), que permite que programas Java sejam executados em qualquer sistema operativo sem necessidade de recompilação.

Diferenças entre Java SE e Jakarta EE (antiga Java EE):

O ecossistema Java é composto por várias edições, cada uma direcionada a diferentes necessidades de desenvolvimento:

- **Java SE (Standard Edition):**

Fornece as bibliotecas e APIs essenciais para a programação de uso geral. É ideal para o desenvolvimento de aplicações desktop, ferramentas de linha de comando e aplicações independentes.

- **Jakarta EE (Enterprise Edition, anteriormente Java EE):**

Baseia-se no Java SE e é projetada especificamente para aplicações empresariais que requerem uma arquitetura multcamada, distribuída e escalável. Inclui APIs adicionais para funcionalidades como servlets, JSP (Jakarta Server Pages, anteriormente JavaServer Pages), EJB

(Enterprise JavaBeans), JMS (Jakarta Message Service) e JPA (Jakarta Persistence API). Desde a sua transição para a Eclipse Foundation, Jakarta EE manteve as funcionalidades essenciais de Java EE, mas tem evoluído para oferecer maior flexibilidade e suporte a padrões modernos, como aplicações cloud-native e microserviços.

É importante notar que, embora Jakarta EE tenha adotado o prefixo ***jakarta*** nos pacotes das APIs, ainda é muito comum encontrar código legado que utiliza ***javax*** (por exemplo, *javax.servlet* ou *javax.persistence*). Nestes casos, muitos editores de código, como o **VS Code**, podem indicar que é recomendável atualizar para os pacotes ***jakarta***, especialmente em projetos novos ou em ambientes compatíveis com Jakarta EE 9 ou superior.

Resumo da Transição do Java EE para a Jakarta EE

Aspecto	Java SE (Standard Edition)	Java EE (Enterprise Edition)	Jakarta EE (Enterprise Edition)
Primeiro Lançamento	1996 (JDK 1.0)	1999 (J2EE 1.2)	2018 (Jakarta EE 8)
Base	Independente	Baseia-se no Java SE	Baseia-se no Java SE
Responsável pela Governança	Oracle Corporation	Oracle Corporation	Eclipse Foundation
Transição de Governança	-	-	Transferência da Oracle para Eclipse Foundation (2017)
Última Versão Antes da Mudança	-	Java EE 8 (2017)	Jakarta EE 8 (2018, idêntica a Java EE 8)
Mudança nos Pacotes	Não aplicável	Prefixo javax	Mudança de javax para jakarta (Jakarta EE 9, 2020)
Versões do Java SE Suportadas	Sempre atual com versões mais recentes	Até Java SE 8	A partir de Jakarta EE 9, suporte para Java SE 11 e superior
Principais APIs Incluídas	java.lang, java.util, java.io, java.nio, java.sql, etc.	javax.servlet, javax.persistence, javax.ejb, etc.	jakarta.servlet, jakarta.persistence, jakarta.ejb, etc.
Compatibilidade Retroativa	Não aplicável	Algumas alterações de API entre versões	Total compatibilidade com código de Java EE 8 em Jakarta EE 8
Foco de Uso	Aplicações standalone, desktop, ferramentas CLI	Aplicações empresariais (web, distribuídas e escaláveis)	Aplicações empresariais modernas e cloud-native
Inovação e Ciclo de Lançamento	Ciclo regular, muito ativo	Ciclo lento	Ciclo mais rápido e aberto à comunidade
Impacto da Transição para os Programadores	Não aplicável	-	Editores de código recomendam migrar de javax para jakarta quando aplicável

Explicação Adicional sobre a Transição de Java EE para Jakarta EE:

1. Origens da Transição:

A transição ocorreu em 2017, quando a Oracle transferiu a governança do Java EE para a Eclipse Foundation. A partir de 2018, a plataforma foi renomeada para **Jakarta EE**. A versão **Jakarta EE 8** foi lançada para garantir compatibilidade com Java EE 8, enquanto a versão **Jakarta EE 9**, lançada em 2020, trouxe a mudança para o prefixo jakarta.

2. Versões de Java SE Suportadas:

- **Java EE:** Suporte típico até Java SE 8.
- **Jakarta EE:** A partir de Jakarta EE 9, as versões mais modernas de Java SE, como Java SE 11, passaram a ser amplamente suportadas, com o objetivo de alinhar a plataforma com as práticas modernas de desenvolvimento.

3. Recomendações de Atualização:

Como mencionado, ainda existe muito código que utiliza o prefixo **javax**. Contudo, editores de código como o **VS Code** ou o **IntelliJ IDEA** recomendam a migração para **jakarta** nas versões mais recentes, especialmente em novos projetos ou ao trabalhar com Jakarta EE 9 ou superior.

Surgimento da Spring Framework

No final dos anos 1990 e início dos 2000, o desenvolvimento com Java EE era frequentemente percebido como complexo e pesado, devido à necessidade de configurações extensivas e à rigidez do código resultante. Em resposta a esses desafios, Rod Johnson criou a Spring Framework, introduzindo conceitos como:

- **Inversão de Controlo (IoC)**: Delegando ao framework a responsabilidade de gerir as dependências entre os componentes, promovendo um baixo acoplamento e facilitando a testabilidade.
- **Injeção de Dependências (DI)**: Permitindo que as dependências sejam fornecidas externamente, tornando o código mais modular e flexível.
- **Programação Orientada a Aspectos (AOP)**: Facilitando a separação de preocupações transversais, como logging e gestão de transações, sem poluir a lógica de negócios.

A Spring Framework rapidamente ganhou popularidade por simplificar o desenvolvimento de aplicações Java empresariais, oferecendo uma alternativa mais leve e flexível ao modelo tradicional do Java EE.

Evolução com o Spring Boot

Apesar das vantagens do Spring, a sua configuração podia ser trabalhosa, exigindo a definição manual de múltiplos ficheiros XML ou anotações detalhadas. Para simplificar este processo, foi criado o Spring Boot, lançado em 2014, com o objetivo de:

- **Autoconfiguração**: Reduzindo ou eliminando a necessidade de configurações explícitas, permitindo que os desenvolvedores iniciem projetos rapidamente com configurações padrão sensatas.
- **Aplicações Independentes**: Facilitando a criação de aplicações que podem ser executadas de forma autónoma, sem a necessidade de servidores de aplicação externos, graças à inclusão de servidores embutidos como Tomcat ou Jetty.
- **Starters POMs**: Fornecendo conjuntos de dependências pré-configuradas para diferentes funcionalidades, simplificando a gestão de dependências no Maven ou Gradle.

O Spring Boot revolucionou o desenvolvimento com Spring, tornando-o mais acessível e eficiente, permitindo que os desenvolvedores se concentrem na lógica de negócios em vez de se preocuparem com configurações complexas.

Em resumo, a Spring Framework e o Spring Boot desempenharam papéis cruciais na evolução do desenvolvimento de aplicações Java empresariais, oferecendo soluções que simplificam e

agilizam o processo de desenvolvimento, em contraste com as abordagens mais tradicionais do Java EE.

Convivência entre Spring Boot e Jakarta EE

Embora o Spring Boot seja frequentemente visto como uma alternativa ao modelo tradicional de desenvolvimento baseado em Java EE (agora Jakarta EE), é importante compreender que estas tecnologias podem coexistir e, em alguns casos, até complementar-se. Spring Boot é uma framework centrada na simplicidade e agilidade do desenvolvimento, enquanto Jakarta EE continua a ser um padrão aberto que define especificações para o desenvolvimento de aplicações empresariais robustas e escaláveis.

A principal diferença está no nível de abstração e na abordagem. O Spring Boot abstrai muitas das configurações e detalhes internos, permitindo uma experiência de desenvolvimento mais rápida e intuitiva. Por outro lado, Jakarta EE fornece especificações padronizadas que podem ser implementadas por diferentes servidores de aplicação, como WildFly, Payara ou TomEE. Graças a esta abertura, é possível integrar componentes de Jakarta EE (como JPA para persistência ou Servlets para gestão de requisições web) em aplicações Spring Boot, aproveitando o melhor de ambos os mundos. Esta convivência é particularmente útil em projetos que requerem padrões bem estabelecidos, mas que também beneficiam da agilidade e conveniência do Spring Boot.

Assim, a escolha entre Spring Boot e Jakarta EE, ou a sua combinação, dependerá das necessidades do projeto, das preferências da equipa de desenvolvimento e das exigências de escalabilidade e padronização da aplicação.

Comparação entre Spring, Spring Boot e Jakarta EE

O desenvolvimento de aplicações Java empresariais pode ser realizado através de diferentes abordagens, dependendo das necessidades do projeto e das preferências da equipa de desenvolvimento. Três opções amplamente utilizadas são a **Spring Framework**, o **Spring Boot** e o **Jakarta EE**. Embora tenham objetivos e características distintas, estas tecnologias podem complementar-se em certos contextos ou ser utilizadas isoladamente para atender a diferentes requisitos.

A **tabela na página seguinte** apresenta uma comparação entre estas três abordagens, destacando as principais diferenças em termos de origem, foco, configuração, integração com padrões Java e casos de uso típicos. Este panorama ajuda a compreender as vantagens e desvantagens de cada uma, facilitando a escolha da solução mais adequada para cada cenário de desenvolvimento.

Tabela comparativa entre Spring Framework, Spring Boot e Jakarta EE:

Aspetto	Spring Framework	Spring Boot	Jakarta EE
Origem	Criado em 2003 por Rod Johnson como alternativa ao Java EE	Criado em 2014 para simplificar a utilização do Spring Framework	Evolução do Java EE, transferido para a Eclipse Foundation em 2017
Foco Principal	Flexibilidade no desenvolvimento modular	Simplicidade e rapidez no desenvolvimento com Spring	Padrões abertos para aplicações empresariais escaláveis
Abordagem	Framework modular com configuração manual ou anotada	Autoconfiguração para reduzir configurações	Especificações padronizadas para vários fornecedores
Gestão de Dependências	Configurada manualmente (via XML ou JavaConfig)	Starters POMs para configuração simplificada	Dependências definidas com base nas especificações
Servidor de Aplicação	Requer um servidor externo (ex.: Tomcat) ou embutido manualmente	Inclui servidores embutidos (ex.: Tomcat, Jetty)	Necessita de um servidor de aplicação compatível (ex.: WildFly)
Arquitetura Suportada	Multicamadas, monólitos, microserviços	Multicamadas, microserviços, cloud-native	Multicamadas, monólitos, sistemas distribuídos
Integração com Padrões Java	Suporte para especificações Java (ex.: JPA, JMS, Servlet)	Usa Spring abstraindo especificações Java quando aplicável	Baseado em especificações Java como JPA, JMS, Servlet
Configuração	Extensa, com opções detalhadas	Simples, com valores padrão configuráveis	Moderada, dependendo das necessidades e servidor utilizado
Público-Alvo	Desenvolvedores que necessitam de flexibilidade e personalização	Desenvolvedores que procuram rapidez e simplicidade	Equipas que necessitam de padrões abertos e multiplataforma
Ecosistema de Ferramentas	Rico, com integração em vários IDEs	Rico, com suporte integrado e ferramentas específicas	Dependente do servidor de aplicação e fornecedor
Casos de Uso Típicos	Sistemas customizados com necessidade de configuração fina	Aplicações rápidas, microserviços, prototipagem ágil	Grandes sistemas corporativos com alta padronização

Resumo:

- Spring Framework: Oferece grande flexibilidade e modularidade, mas pode ser mais trabalhoso configurar.
- Spring Boot: Facilita a utilização do Spring Framework, proporcionando autoconfiguração e conveniência para projetos rápidos e modernos.
- Jakarta EE: Fornece um conjunto padronizado de APIs e especificações, promovendo interoperabilidade e conformidade com diferentes servidores de aplicação.

A escolha entre eles depende das necessidades do projeto, da experiência da equipa e das preferências em termos de configuração e padronização.

Capítulo 2. Bases de Java e de Programação Orientada por Objetos

Motivos para aprender Java

- **Portabilidade:** O lema "Escreve uma vez, executa em qualquer lugar" (Write Once, Run Anywhere - WORA) descreve bem a capacidade do Java de funcionar em qualquer sistema operativo com a Máquina Virtual Java (JVM).
- **Robustez e segurança:** Java inclui funcionalidades de gestão de memória e tratamento de exceções que ajudam a criar aplicações seguras e estáveis.
- **Comunidade e mercado:** Java continua a ser uma escolha sólida em ambientes empresariais e uma habilidade muito procurada no mercado de trabalho.

Características principais do Java

1. **Orientação a Objetos:** Java baseia-se em conceitos como classes e objetos, tornando o código mais modular e reutilizável.
2. **Fortemente Tipada:** Todas as variáveis e operações têm tipos bem definidos.
3. **Independência de Plataforma:** O código Java é compilado para bytecode, que pode ser executado em qualquer JVM.
4. **Biblioteca Rica:** Java possui uma extensa biblioteca padrão, que facilita o desenvolvimento de aplicações.

Configuração do ambiente de desenvolvimento no Visual Studio Code

Nota: aqui estamos apenas preocupados em ter a configuração básica do Java no VS Code, para exercícios destinados às pessoas que não conhecem Java, ou precisam de um “refresher”.

Passo 1: Instalar o JDK (Java Development Kit)

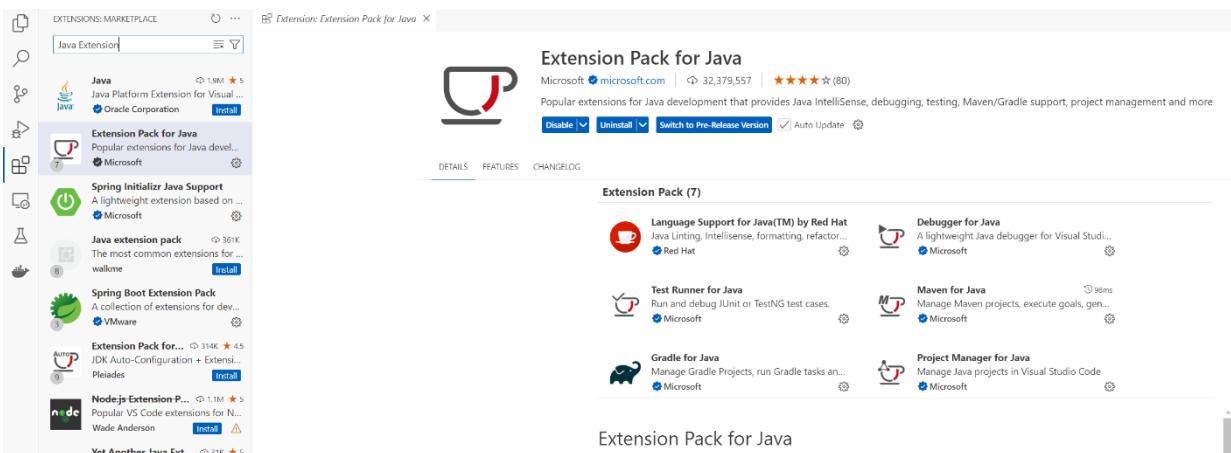
1. Aceder ao [site oficial da Oracle](#) ou à [Adoptium](#).

2. Fazer o download e instalar o JDK 21 para o sistema operativo em uso (Windows, macOS ou Linux).
3. Verificar a instalação:
4. `java -version`

O resultado deverá indicar a versão instalada, como `java 21`.

Passo 2: Configurar o VS Code

1. Instalar o VS Code a partir do [site oficial](#).
2. Adicionar a extensão para Java:
 - o No VS Code, abrir o "Extensions Marketplace" (Ctrl+Shift+X ou Cmd+Shift+X).
 - o Procurar por "**Extension Pack for Java**" e instalar (c.f. imagem em baixo).



Primeiro programa em Java: "Olá, Mundo!"

Passo 1: Criar o ficheiro

1. Criar uma nova pasta no VS Code chamada *primeiro-programa*.
 2. Criar um ficheiro chamado *Main.java* dentro da pasta.
-

Passo 2: Escrever o código

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Olá, Mundo!");  
    }  
}
```

O que este código faz?

1. *public class Main*: Define uma classe chamada *Main*.
 2. *public static void main(String[] args)*: Define o ponto de entrada do programa.
 3. *System.out.println("Olá, Mundo!");*: Imprime "Olá, Mundo!" no ecrã.
-

Passo 3: Executar o programa

1. Basta carregar no botão "Play", ou no texto "Run", mas se quiser fazer "à antiga" pode usar os passos descritos em baixo. O resultado será o mesmo.
2. Abrir o terminal no VS Code.
3. Compilar o programa:
4. *javac Main.java*

Este comando gera o ficheiro *Main.class*, que contém o bytecode.

5. Executar o programa:
6. *java Main*

O resultado será:

Olá, Mundo!

Desafios

1. Modifique o programa para imprimir o teu nome no ecrã.
2. Experimente criar outra classe, por exemplo, *Pessoa*, e fazer com que esta imprima uma mensagem diferente.

Fundamentos da Linguagem Java

Nesta secção, vamos explorar os pilares fundamentais da linguagem Java. Compreender estes conceitos é essencial para escrever programas eficazes e preparar-se para construir aplicações mais complexas.

Estrutura Básica de um Programa Java

Todo programa Java tem uma estrutura básica composta por:

1. **Classe**: A unidade fundamental do programa, onde o código é organizado.
2. **Método *main***: O ponto de entrada para execução do programa.

Exemplo:

```
public class Exemplo {  
    public static void main(String[] args) {  
        System.out.println("Bem-vindo aos fundamentos do Java!");  
    }  
}
```

- *public class Exemplo*: Define uma classe chamada *Exemplo*.
 - *public static void main(String[] args)*: Define o método principal onde o programa começa.
-

Tipos de Dados Primitivos

Java é uma linguagem **fortemente tipada**, o que significa que cada variável deve ter um tipo definido.

Tipo	Tamanho	Valor Máximo/Descrição
<i>byte</i>	8 bits	-128 a 127
<i>short</i>	16 bits	-32,768 a 32,767
<i>int</i>	32 bits	-2^{31} a $2^{31}-1$
<i>Long</i>	64 bits	-2^{63} a $2^{63}-1$
<i>float</i>	32 bits	Número decimal com precisão simples
<i>double</i>	64 bits	Número decimal com precisão dupla
<i>char</i>	16 bits	Um único carácter (Unicode)
<i>boolean</i>	1 bit	true ou false

Exemplos:

```
int idade = 25;
double salario = 1500.50;
char inicial = 'A';
boolean ativo = true;

System.out.println("Idade: " + idade);
System.out.println("Salário: " + salario);
System.out.println("Inicial: " + inicial);
System.out.println("Ativo: " + ativo);
```

Variáveis e Constantes

- **Variáveis:** Armazenam valores que podem mudar durante a execução do programa.
- **Constantes:** Valores fixos definidos com *final*.

Exemplo:

```
int altura = 180; // variável
final double PI = 3.14159; // constante

System.out.println("Altura: " + altura);
System.out.println("Valor de PI: " + PI);
```

Operadores

1. **Aritméticos:** +, -, *, /, %.
2. **Relacionais:** ==, !=, >, <, >=, <=.
3. **Lógicos:** && (e), || (ou), ! (não).

Exemplo:

```
int a = 10, b = 20;  
System.out.println(a + b); // Soma  
System.out.println(a > b); // Relacional  
System.out.println(a > 5 && b < 30); // Lógico
```

Introdução ao Controlo de Fluxo

1. Condicionais:

if, else:

```
int idade = 18;
if (idade >= 18) {
    System.out.println("Maior de idade.");
} else {
    System.out.println("Menor de idade.");
}
```

switch:

```
int dia = 2;
switch (dia) {
    case 1: System.out.println("Segunda-feira"); break;
    case 2: System.out.println("Terça-feira"); break;
    default: System.out.println("Outro dia");
}
```

Ciclos:

for:

```
for (int i = 1; i <= 5; i++) {
    System.out.println("Número: " + i);
}
```

while:

```
int i = 1;
while (i <= 5) {
    System.out.println("Número: " + i);
    i++;
}
```

do-while:

```
int i = 1;
do {
    System.out.println("Número: " + i);
    i++;
} while (i <= 5);
```

Exemplos Práticos

Soma de dois números:

```
import java.util.Scanner;

public class Soma {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Digite o primeiro número: ");
        int num1 = sc.nextInt();
        System.out.print("Digite o segundo número: ");
        int num2 = sc.nextInt();

        int soma = num1 + num2;
        System.out.println("A soma é: " + soma);
    }
}
```

Verificar se um número é par ou ímpar:

```
import java.util.Scanner;

public class ParOuImpar {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Digite um número: ");
        int numero = sc.nextInt();

        if (numero % 2 == 0) {
            System.out.println("O número é par.");
        } else {
            System.out.println("O número é ímpar.");
        }
    }
}
```

Exercícios

1. Escreva um programa que leia três números e determine o maior.
2. Crie um ciclo *for* que imprima os números de 1 a 100.
3. Escreva um programa que peça ao utilizador a sua idade e diga em que faixa etária ele se encontra:
 - o Menor de idade (<18).
 - o Adulto (18-64).
 - o Sénior (>=65).

Esta secção fornece as bases necessárias para compreender e usar Java de forma eficaz. No próximo capítulo, vamos explorar a **Programação Orientada por Objetos**, o coração da linguagem Java.

Introdução à Programação Orientada por Objetos (POO)

A Programação Orientada por Objetos (POO) é um paradigma que organiza o código em **objetos**, permitindo criar aplicações mais estruturadas, reutilizáveis e fáceis de manter. Java foi desenhada desde o início com a POO como base, tornando este conceito essencial para o domínio da linguagem.

O que é a POO?

A POO organiza o software em **objetos**, que são instâncias de **classes**. Estes objetos representam entidades do mundo real, como um "Carro", "Pessoa" ou "Produto", com as seguintes características:

- **Atributos:** Representam os dados do objeto.
- **Métodos:** Representam as ações ou comportamentos.

Principais Conceitos de POO

1. Classes e Objetos:

- o **Classe:** É um molde que define os atributos e métodos de um objeto.
- o **Objeto:** É uma instância de uma classe. Cada objeto tem os seus próprios valores para os atributos definidos na classe.

Exemplo:

```
public class Pessoa {  
    String nome; // Atributo  
    int idade;  
  
    void dizerOla() { // Método  
        System.out.println("Olá, o meu nome é " + nome);  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Pessoa pessoa = new Pessoa(); // Criar um objeto  
        pessoa.nome = "João";  
        pessoa.idade = 25;  
  
        pessoa.dizerOla();  
    }  
}
```

2. Encapsulamento:

- o Consiste em esconder os detalhes internos de uma classe e expor apenas o que é necessário.
- o Usa modificadores de acesso como *private, public, protected*.

Exemplo:

```
public class ContaBancaria {  
    private double saldo;  
  
    public void depositar(double valor) {  
        saldo += valor;  
    }  
    public double getSaldo() {  
        return saldo;  
    }  
}
```

3. Herança:

- Permite que uma classe derive de outra, reutilizando e estendendo funcionalidades.
- Usa a palavra-chave *extends*.

Exemplo:

```
public class Animal {  
    void fazerSom() {  
        System.out.println("Animal faz som.");  
    }  
}  
  
public class Cachorro extends Animal {  
    void fazerSom() {  
        System.out.println("Cachorro late.");  
    }  
}
```

4. Polimorfismo:

- Refere-se à capacidade de um objeto se comportar de diferentes formas, dependendo do contexto.
- Existem dois tipos: sobrecarga (mesmo nome, argumentos diferentes) e sobrescrita (redefinir método de uma superclasse).

Exemplo: Sobrescrita

```
Animal animal = new Cachorro();  
animal.fazerSom(); // Saída: "Cachorro late."
```

Criar a Primeira Classe Java

Vamos criar uma classe chamada "**Carro**", com atributos e métodos que representam um veículo.

Passo 1: Definir atributos e métodos

```
public class Carro {  
    String marca;  
    String modelo;  
    int ano;  
    double velocidade;  
  
    // Método para acelerar  
    void acelerar(double incremento) {  
        velocidade += incremento;  
        System.out.println("O carro acelerou para " + velocidade + " km/h.");  
    }  
  
    // Método para travar  
    void travar() {  
        velocidade = 0;  
        System.out.println("O carro travou.");  
    }  
}
```

Passo 2: Criar e usar objetos

```
public class Main {  
    public static void main(String[] args) {  
        // Criar um objeto da classe Carro  
        Carro meuCarro = new Carro();  
  
        // Definir atributos  
        meuCarro.marca = "Toyota";  
        meuCarro.modelo = "Corolla";  
        meuCarro.ano = 2020;  
  
        // Utilizar métodos  
        meuCarro.acelerar(50);  
        meuCarro.travar();  
    }  
}
```

Exemplo Prático Completo: Classe "Carro"

Vamos melhorar a classe **Carro**, aplicando **encapsulamento** e adicionando um construtor.

Código:

```
public class Carro {  
    private String marca;  
    private String modelo;  
    private int ano;  
    private double velocidade;  
  
    // Construtor  
    public Carro(String marca, String modelo, int ano) {  
        this.marca = marca;  
        this.modelo = modelo;  
        this.ano = ano;  
        this.velocidade = 0;  
    }  
  
    // Métodos getters  
    public String getMarca() {  
        return marca;  
    }  
  
    public String getModelo() {  
        return modelo;  
    }  
  
    public int getAno() {  
        return ano;  
    }  
  
    public double getVelocidade() {  
        return velocidade;  
    }  
  
    // Métodos  
    public void acelerar(double incremento) {  
        velocidade += incremento;  
        System.out.println(marca + " " + modelo + " acelerou para " + velocidade +  
" km/h.");  
    }  
  
    public void travar() {  
        velocidade = 0;  
    }  
}
```

```

        System.out.println(marca + " " + modelo + " travou.");
    }

}

public class Main {
    public static void main(String[] args) {
        // Criar objeto usando o construtor
        Carro carro1 = new Carro("Honda", "Civic", 2022);

        // Mostrar informações
        System.out.println("Carro: " + carro1.getMarca() + " " + carro1.getModelo()
+ ", Ano: " + carro1.getAno());

        // Utilizar métodos
        carro1.acelerar(60);
        carro1.travar();
    }
}

```

Resumo e Exercícios

Resumo:

- Classes e objetos organizam o código em estruturas reutilizáveis.
- Encapsulamento melhora a segurança e integridade dos dados.
- Herança permite reutilizar e estender funcionalidades.
- Polimorfismo facilita a adaptação a diferentes contextos.

Exercícios:

1. Crie uma classe *Pessoa* com os atributos *nome*, *idade* e o método *dizerIdade*, que imprime a idade.
2. Crie uma classe *Funcionario* que herda de *Pessoa* e adiciona um atributo *salario* e o método *mostrarSalario*.
3. Crie um programa onde:
 - Um objeto da classe *Funcionario* é instanciado.
 - Os métodos *dizerIdade* e *mostrarSalario* são utilizados.

A seguir, exploraremos algumas **estruturas de dados e coleções**, essenciais para lidar com informação de forma eficiente em Java.

Estruturas de Dados e Coleções em Java

Gerir e manipular dados de forma eficiente é uma habilidade essencial em qualquer linguagem de programação. Nesta secção, vamos explorar as **estruturas de dados em Java**, desde arrays básicos até ao poderoso **framework de coleções**.

Arrays: Declaração, Inicialização e Manipulação

Um **array** é uma estrutura de dados que armazena múltiplos valores do mesmo tipo.

1. Declaração e Inicialização

```
// Declaração  
int[] numeros;  
  
// Inicialização  
numeros = new int[5]; // Array com 5 posições  
  
// Declaração e inicialização combinadas  
int[] numeros2 = {10, 20, 30, 40, 50};
```

2. Acesso aos Elementos Os elementos de um array são acedidos pelos índices (começam em 0).

```
numeros[0] = 15; // Atribuir valor à primeira posição  
System.out.println(numeros2[1]); // Imprimir o segundo elemento
```

3. Percorrer Arrays Utilizamos ciclos para iterar pelos elementos de um array.

```
for (int i = 0; i < numeros2.Length; i++) {  
    System.out.println("Elemento na posição " + i + ": " + numeros2[i]);  
}  
  
// Ciclo "for-each"  
for (int numero : numeros2) {  
    System.out.println("Número: " + numero);  
}
```

Introdução à Framework de Coleções

A **framework de coleções** oferece classes e interfaces para trabalhar com estruturas de dados mais avançadas, como listas, conjuntos e mapas.

1. ArrayList

Uma **ArrayList** é uma lista dinâmica que pode crescer ou diminuir de tamanho conforme necessário.

Declaração e Operações Básicas:

```
import java.util.ArrayList;

public class ListaExemplo {
    public static void main(String[] args) {
        ArrayList<String> alunos = new ArrayList<>();

        // Adicionar elementos
        alunos.add("Ana");
        alunos.add("João");
        alunos.add("Maria");

        // Aceder elementos
        System.out.println("Primeiro aluno: " + alunos.get(0));

        // Remover elementos
        alunos.remove("João");

        // Iterar pela Lista
        for (String aluno : alunos) {
            System.out.println("Aluno: " + aluno);
        }

        // Tamanho da lista
        System.out.println("Total de alunos: " + alunos.size());
    }
}
```

2. HashSet

Um **HashSet** é uma coleção que armazena elementos únicos, ou seja, não permite duplicados.

Exemplo:

```
import java.util.HashSet;

public class ConjuntoExemplo {
    public static void main(String[] args) {
        HashSet<String> frutas = new HashSet<>();

        // Adicionar elementos
        frutas.add("Maçã");
        frutas.add("Banana");
        frutas.add("Maçã"); // Duplicado (não será adicionado)

        // Verificar a existência de um elemento
        if (frutas.contains("Banana")) {
            System.out.println("Banana está no conjunto.");
        }

        // Iterar pelo conjunto
        for (String fruta : frutas) {
            System.out.println("Fruta: " + fruta);
        }
    }
}
```

3. HashMap

Um **HashMap** armazena pares chave-valor, permitindo um acesso rápido aos valores através das suas chaves.

Exemplo:

```
import java.util.HashMap;

public class MapaExemplo {
    public static void main(String[] args) {
        HashMap<Integer, String> alunos = new HashMap<>();

        // Adicionar elementos (chave, valor)
        alunos.put(1, "Ana");
        alunos.put(2, "João");
        alunos.put(3, "Maria");

        // Aceder ao valor pela chave
        System.out.println("Aluno com ID 2: " + alunos.get(2));

        // Remover elementos
        alunos.remove(3);

        // Iterar pelo mapa
        for (Integer id : alunos.keySet()) {
            System.out.println("ID: " + id + ", Nome: " + alunos.get(id));
        }
    }
}
```

Exemplo Prático: Gerir uma Lista de Alunos

Vamos criar uma aplicação simples para gerir uma lista de alunos, utilizando uma **ArrayList**.

Código:

```
import java.util.ArrayList;
import java.util.Scanner;

public class GestaoAlunos {
    public static void main(String[] args) {
        ArrayList<String> alunos = new ArrayList<>();
        Scanner sc = new Scanner(System.in);
        int opcao;

        do {
            System.out.println("\n==== Gestão de Alunos ====");
            System.out.println("1. Adicionar aluno");
            System.out.println("2. Listar alunos");
            System.out.println("3. Remover aluno");
            System.out.println("4. Sair");
            System.out.print("Escolha uma opção: ");
            opcao = sc.nextInt();
            sc.nextLine(); // Limpar buffer do scanner

            switch (opcao) {
                case 1:
                    System.out.print("Digite o nome do aluno: ");
                    String nome = sc.nextLine();
                    alunos.add(nome);
                    System.out.println("Aluno adicionado.");
                    break;

                case 2:
                    System.out.println("Lista de alunos:");
                    for (String aluno : alunos) {
                        System.out.println("- " + aluno);
                    }
                    break;

                case 3:
                    System.out.print("Digite o nome do aluno a remover: ");
                    String nomeRemover = sc.nextLine();
                    if (alunos.remove(nomeRemover)) {
                        System.out.println("Aluno removido.");
                    } else {
                        System.out.println("Aluno não encontrado.");
                    }
            }
        }
    }
}
```

```

        break;

    case 4:
        System.out.println("A sair... ");
        break;

    default:
        System.out.println("Opção inválida. ");
    }
} while (opcao != 4);

sc.close();
}
}

```

Saída Exemplo:

```

==== Gestão de Alunos ====
1. Adicionar aluno
2. Listar alunos
3. Remover aluno
4. Sair
Escolha uma opção: 1
Digite o nome do aluno: Pedro
Aluno adicionado.

```

Resumo e Exercícios

Resumo:

- Arrays são estruturas básicas para armazenar dados de tamanho fixo.
- O framework de coleções oferece estruturas dinâmicas, como `ArrayList`, `HashSet` e `HashMap`.
- Cada estrutura tem características únicas para diferentes casos de uso:
 - **ArrayList**: Lista ordenada, permite duplicados.
 - **HashSet**: Conjunto não ordenado, sem duplicados.
 - **HashMap**: Estrutura chave-valor, rápida para pesquisa.

Exercícios:

1. Crie uma aplicação que utilize um **HashSet** para armazenar nomes de produtos únicos.
2. Desenvolva um programa que use um **HashMap** para gerir uma tabela de preços de produtos.
3. Altere o exemplo da gestão de alunos para permitir editar o nome de um aluno existente.

No próximo capítulo, exploraremos **manipulação de ficheiros** em Java, uma funcionalidade importante para armazenar e recuperar dados.

Manipulação de Ficheiros

A manipulação de ficheiros é uma funcionalidade essencial para armazenar e recuperar dados de forma persistente. Em Java, a biblioteca padrão oferece classes e métodos que facilitam a leitura e escrita em ficheiros.

Introdução à Manipulação de Ficheiros em Java

Em Java, as operações com ficheiros baseiam-se em classes da biblioteca `java.io`. As mais comuns são:

- **`File`**: Para operações básicas, como verificar se um ficheiro existe ou criar um novo ficheiro.
- **`FileWriter`**: Para escrever em ficheiros.
- **`BufferedReader`**: Para ler ficheiros de forma eficiente.

Ler e Escrever em Ficheiros

1. Criar e Verificar Ficheiros com a Classe `File`

A classe `File` permite verificar se um ficheiro existe, criar novos ficheiros e diretórios, ou até apagar ficheiros.

Exemplo:

```
import java.io.File;  
  
import java.io.IOException;  
  
public class ExemploFile {  
    public static void main(String[] args) {  
        File ficheiro = new File("inventario.txt");  
  
        try {  
            if (ficheiro.createNewFile()) {  
                System.out.println("Ficheiro criado: " + ficheiro.getName());  
            } else {  
                System.out.println("O ficheiro já existe.");  
            }  
        } catch (IOException e) {  
            System.out.println("Ocorreu um erro.");  
            e.printStackTrace();  
        }  
    }  
}
```

2. Escrever em Ficheiros com a Classe *FileWriter*

A classe *FileWriter* permite escrever texto num ficheiro, adicionando ou substituindo o conteúdo existente.

Exemplo:

```
import java.io.FileWriter;
import java.io.IOException;

public class ExemploFileWriter {
    public static void main(String[] args) {
        try {
            FileWriter escritor = new FileWriter("inventario.txt");
            escritor.write("Produto: Computador\n");
            escritor.write("Quantidade: 10\n");
            escritor.close();
            System.out.println("Dados escritos com sucesso!");
        } catch (IOException e) {
            System.out.println("Ocorreu um erro ao escrever no ficheiro.");
            e.printStackTrace();
        }
    }
}
```

3. Ler Ficheiros com a Classe *BufferedReader*

A classe *BufferedReader* lê o conteúdo de um ficheiro linha a linha, o que é útil para ficheiros grandes.

Exemplo:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ExemploBufferedReader {
    public static void main(String[] args) {
        try {
            BufferedReader leitor = new BufferedReader(new
FileReader("inventario.txt"));
            String linha;
            while ((linha = leitor.readLine()) != null) {
                System.out.println(linha);
            }
            leitor.close();
        }
    }
}
```

```

        } catch (IOException e) {
            System.out.println("Ocorreu um erro ao ler o ficheiro.");
            e.printStackTrace();
        }
    }
}

```

Exemplo Prático: Guardar e Carregar Dados de um Inventário Simples

Vamos criar um programa que permite guardar e carregar informações de um inventário simples.

Código Completo:

```

import java.io.*;
import java.util.ArrayList;
import java.util.Scanner;

public class Inventario {
    private static final String FICHEIRO = "inventario.txt";

    public static void main(String[] args) {
        ArrayList<String> inventario = new ArrayList<>();
        Scanner sc = new Scanner(System.in);
        int opcao;

        // Carregar dados do ficheiro
        carregarInventario(inventario);

        do {
            System.out.println("\n==== Inventário ===");
            System.out.println("1. Adicionar produto");
            System.out.println("2. Listar produtos");
            System.out.println("3. Guardar e sair");
            System.out.print("Escolha uma opção: ");
            opcao = sc.nextInt();
            sc.nextLine(); // Limpar o buffer do scanner

            switch (opcao) {
                case 1:
                    System.out.print("Digite o nome do produto: ");
                    String produto = sc.nextLine();
                    inventario.add(produto);
                    System.out.println("Produto adicionado.");
                    break;

                case 2:
                    System.out.println("Lista de produtos:");

```

```

        for (String item : inventario) {
            System.out.println("- " + item);
        }
        break;

    case 3:
        guardarInventario(inventario);
        System.out.println("Inventário guardado. A sair... ");
        break;

    default:
        System.out.println("Opção inválida.");
    }
} while (opcao != 3);

sc.close();
}

// Método para carregar dados do ficheiro
private static void carregarInventario(ArrayList<String> inventario) {
    try {
        BufferedReader leitor = new BufferedReader(new FileReader(FICHEIRO));
        String linha;
        while ((linha = leitor.readLine()) != null) {
            inventario.add(linha);
        }
        leitor.close();
    } catch (IOException e) {
        System.out.println("O ficheiro não existe ou está vazio.");
    }
}

// Método para guardar dados no ficheiro
private static void guardarInventario(ArrayList<String> inventario) {
    try {
        FileWriter escritor = new FileWriter(FICHEIRO);
        for (String item : inventario) {
            escritor.write(item + "\n");
        }
        escritor.close();
    } catch (IOException e) {
        System.out.println("Ocorreu um erro ao guardar o inventário.");
        e.printStackTrace();
    }
}
}

```

Funcionamento do Programa



Financiado pela
União Europeia
NextGenerationEU

1. Ao iniciar, o programa tenta carregar os dados de um ficheiro chamado `inventario.txt`.
 2. O utilizador pode:
 - Adicionar produtos.
 - Listar produtos existentes.
 - Guardar o inventário e sair.
 3. Os dados são persistidos no ficheiro `inventario.txt`.
-

Resumo e Exercícios

Resumo:

- A classe `File` permite criar e verificar ficheiros.
- `FileWriter` é usada para escrever texto em ficheiros.
- `BufferedReader` facilita a leitura linha a linha de ficheiros.

Exercícios:

1. Adapte o programa para permitir guardar a quantidade de cada produto (ex.: "Produto: Computador, Quantidade: 10").
2. Adicione uma opção para remover produtos do inventário.
3. Crie um método que calcula e imprime o número total de produtos no inventário.

Na próxima secção, exploraremos algumas **funcionalidades modernas** em Java, introduzidas a partir da versão 8 da linguagem.

Introdução às Funcionalidades Modernas do Java

Com a evolução da linguagem, Java introduziu várias funcionalidades modernas, especialmente a partir do **Java 8**, que tornam o código mais expressivo, funcional e eficiente. Estas funcionalidades incluem **expressões lambda**, **streams** para manipulação de coleções, e **interfaces funcionais**. Vamos explorar estes conceitos com exemplos práticos.

Nota: dado o carácter abrangente deste tema, neste manual optou-se por apresentar apenas uma breve introdução. Para os interessados, foi preparado um **documento adicional**, *disponível no GitHub através desta hiperligação*, com mais de 100 páginas. Este documento explora de forma mais completa, mas sempre com exemplos simples e num tom pedagógico, como e quando utilizar estas novas funcionalidades da linguagem, bem como as suas vantagens face aos recursos sintáticos já existentes.

Java 8 e Além: Um Resumo

As versões mais recentes do Java introduziram uma série de melhorias:

1. **Expressões Lambda:** Permitem passar comportamentos como argumentos para métodos, facilitando o uso de programação funcional.
2. **Streams:** Oferecem uma API para processar coleções de dados de forma declarativa.
3. **Interfaces Funcionais:** Interfaces que possuem apenas um método abstrato, sendo essenciais para lambdas.
4. **Referências a Métodos:** Tornam o código mais conciso ao referenciar diretamente métodos existentes.

Estas funcionalidades simplificam a manipulação de dados e promovem um estilo de programação mais funcional.

Expressões Lambda

Uma **expressão lambda** é uma forma compacta de implementar interfaces funcionais. Substitui classes anónimas, tornando o código mais legível.

Sintaxe Básica:

(parametros) -> { corpo da expressão }

Exemplo 1: Utilizar uma expressão lambda para ordenar uma lista

```
import java.util.ArrayList;
import java.util.Collections;

public class LambdaExemplo {
    public static void main(String[] args) {
        ArrayList<String> nomes = new ArrayList<>();
        nomes.add("Ana");
        nomes.add("João");
        nomes.add("Maria");

        // Ordenar usando uma expressão Lambda
        Collections.sort(nomes, (s1, s2) -> s1.compareTo(s2));

        System.out.println("Lista ordenada: " + nomes);
    }
}
```

Exemplo 2: Usar lambdas com Runnable

```
public class LambdaRunnable {
    public static void main(String[] args) {
        // Usar Runnable com uma classe anónima
        Runnable tarefa1 = new Runnable() {
            @Override
            public void run() {
                System.out.println("Tarefa 1 executada!");
            }
        };

        // Usar Runnable com Lambda
        Runnable tarefa2 = () -> System.out.println("Tarefa 2 executada!");

        tarefa1.run();
        tarefa2.run();
    }
}
```

Streams: Manipulação de Coleções de Forma Funcional



Financiado pela
União Europeia
NextGenerationEU

Streams são uma forma declarativa de processar coleções de dados, permitindo aplicar filtros, transformações e outras operações de forma concisa.

1. Operações Intermediárias e Terminais

- **Intermediárias:** Transformam os dados (ex.: *filter*, *map*, *sorted*).
- **Terminais:** Produzem um resultado ou efeito (ex.: *forEach*, *collect*).

Exemplo: Filtrar e Transformar Dados

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

public class StreamsExemplo {
    public static void main(String[] args) {
        List<String> nomes = new ArrayList<>();
        nomes.add("Ana");
        nomes.add("João");
        nomes.add("José");

        // Filtrar nomes que começam com "J" e transformá-los em maiúsculas
        List<String> nomesComJ = nomes.stream()
            .filter(nome -> nome.startsWith("J"))
            .map(String::toUpperCase)
            .collect(Collectors.toList());

        System.out.println("Nomes que começam com 'J': " + nomesComJ);
    }
}
```

Exemplo: Calcular Soma de Números

```
import java.util.Arrays;
import java.util.List;

public class SomaStreams {
    public static void main(String[] args) {
        List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);

        // Soma de números usando streams
        int soma = numeros.stream().reduce(0, Integer::sum);

        System.out.println("Soma: " + soma);
    }
}
```

Interface Funcional e Referências a Métodos

1. Interface Funcional



Financiado pela
União Europeia
NextGenerationEU

- Uma interface funcional é uma interface com apenas um método abstrato, usada como base para lambdas.
- Exemplo: *Runnable*, *Comparator*.

Exemplo: Criar uma Interface Funcional

```
@FunctionalInterface
interface Saudacao {
    void dizerOlá(String nome);
}

public class InterfaceFuncionalExemplo {
    public static void main(String[] args) {
        // Usar uma Lambda para implementar a interface funcional
        Saudacao saudacao = nome -> System.out.println("Olá, " + nome + "!");
        saudacao.dizerOlá("João");
    }
}
```

2. Referências a Métodos

- Simplificam lambdas referenciando métodos existentes.
- Tipos:
 - Referência a método estático: *Classe::método*.
 - Referência a método de instância: *objeto::método*.
 - Referência a um construtor: *Classe::new*.

Exemplo: Referências a Métodos

```
import java.util.Arrays;
import java.util.List;

public class ReferenciasMetodos {
    public static void main(String[] args) {
        List<String> nomes = Arrays.asList("Ana", "João", "Maria");

        // Usar uma referência a método para imprimir cada nome
        nomes.forEach(System.out::println);
    }
}
```

Exemplos Práticos

1. Ordenar uma Lista com Lambdas

```
import java.util.ArrayList;
import java.util.Collections;

public class OrdenacaoLambdas {
    public static void main(String[] args) {
        ArrayList<Integer> numeros = new ArrayList<>();
        numeros.add(5);
        numeros.add(1);
        numeros.add(3);

        // Ordenar usando uma expressão Lambda
        numeros.sort((a, b) -> a - b);

        System.out.println("Números ordenados: " + numeros);
    }
}
```

2. Filtrar Dados Usando Streams

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

public class FiltrarDados {
    public static void main(String[] args) {
        List<String> nomes = Arrays.asList("Ana", "João", "Maria", "José");

        // Filtrar nomes com mais de 3 letras
        List<String> nomesLongos = nomes.stream()
            .filter(nome -> nome.length() > 3)
            .collect(Collectors.toList());

        System.out.println("Nomes Longos: " + nomesLongos);
    }
}
```

Resumo e Exercícios

Resumo:

- **Lambdas:** Compactam a implementação de interfaces funcionais.
- **Streams:** Simplificam a manipulação de coleções, com métodos como *filter*, *map* e *reduce*.
- **Interfaces Funcionais:** Servem de base para lambdas e permitem criar comportamentos reutilizáveis.
- **Referências a Métodos:** Tornam o código mais conciso, referenciando diretamente métodos existentes.

Exercícios:

1. Crie um programa que filtra números pares de uma lista e imprime-os em ordem crescente.
2. Utilize uma interface funcional personalizada para calcular o quadrado de um número.
3. Crie um programa que transforma uma lista de nomes em maiúsculas e ordena-os alfabeticamente usando streams.

No próximo capítulo, iremos começar a explorar o Spring Boot, apresentando uma visão geral das suas principais características e benefícios, destacando como esta framework simplifica o desenvolvimento de aplicações Java modernas.

Capítulo 3. Introdução ao Spring Boot

Nota: este capítulo é só de enquadramento geral do Spring Boot, em parte recapitulando alguns aspectos já mencionados no Capítulo 1. Por isso, não há recursos suplementares (vídeos demonstrativos ou exemplos de código no Github)

Como referido ante (Capítulo 1), o *Spring Boot* é uma framework open-source baseado na Spring Framework, concebido para simplificar o desenvolvimento de aplicações Java autónomas e prontas para produção. Ao adotar convenções sensatas e configurações automáticas, o Spring Boot permite que os programadores se concentrem na lógica de negócio, reduzindo significativamente a complexidade associada à configuração inicial de projetos Spring tradicionais.

Conceitos Fundamentais e Benefícios

- **Configuração Automática (Auto-Configuration):** O Spring Boot analisa as dependências presentes no projeto e configura automaticamente os componentes necessários, eliminando a necessidade de configurações manuais extensivas.
- **Starters:** São conjuntos de dependências pré-configuradas que facilitam a inclusão de funcionalidades específicas na aplicação. Por exemplo, ao adicionar o starter *spring-boot-starter-web*, inclui-se automaticamente o suporte para desenvolvimento de aplicações web.
- **Servidor Integrado:** O Spring Boot incorpora servidores como Tomcat ou Jetty, permitindo que as aplicações sejam executadas de forma autónoma, sem necessidade de configurar servidores externos.

- **Produção Pronta:** Inclui funcionalidades como monitorização, métricas e gestão de configurações externas, essenciais para ambientes de produção.

Benefícios:

- **Rapidez no Desenvolvimento:** A configuração automática e os starters permitem iniciar projetos rapidamente, concentrando-se na lógica de negócio.
- **Redução de Código Boilerplate:** Diminui a quantidade de código repetitivo, tornando o desenvolvimento mais eficiente.
- **Facilidade de Configuração:** A abordagem "convenção sobre configuração" reduz a necessidade de configurações manuais, simplificando o processo de desenvolvimento.
- **Flexibilidade:** Embora ofereça configurações automáticas, permite personalizações conforme as necessidades específicas do projeto.

Diferenças entre Spring Framework e Spring Boot

- **Configuração:**
 - *Spring Framework:* Requer configurações detalhadas, muitas vezes através de ficheiros XML ou classes Java, para definir beans, dependências e outras propriedades da aplicação.
 - *Spring Boot:* Oferece configuração automática baseada nas dependências presentes, reduzindo significativamente a necessidade de configurações manuais.
- **Inicialização do Projeto:**
 - *Spring Framework:* A configuração inicial pode ser complexa e demorada, exigindo a definição manual de diversas dependências e configurações.
 - *Spring Boot:* Utiliza starters que agrupam dependências comuns, facilitando a criação e configuração de novos projetos.

- **Execução da Aplicação:**

- *Spring Framework*: Geralmente, as aplicações necessitam de um servidor de aplicação externo para serem executadas.
- *Spring Boot*: Inclui servidores embutidos, permitindo que as aplicações sejam executadas de forma autónoma, simplificando o processo de desenvolvimento e implantação.

- **Complexidade:**

- *Spring Framework*: Oferece uma vasta gama de funcionalidades e módulos, o que pode resultar em maior complexidade na configuração e manutenção.
- *Spring Boot*: Focado na simplicidade e rapidez, proporciona uma abordagem mais opinativa, com configurações padrão que atendem à maioria dos casos de uso, facilitando o desenvolvimento de microserviços e aplicações independentes.

Em resumo, enquanto o Spring Framework oferece uma base robusta e flexível para o desenvolvimento de aplicações Java, o Spring Boot estende essa base, simplificando e acelerando o processo de desenvolvimento através de configurações automáticas e uma abordagem mais opinativa. Esta combinação permite criar aplicações eficientes e prontas para produção com menor esforço e complexidade.

Capítulo 4. Configuração do Ambiente de Desenvolvimento

Nota: Este capítulo é acompanhado por um **vídeo** com explicações passo-a-passo (ver aqui).

Para iniciar o desenvolvimento de aplicações Java com Spring Boot no Visual Studio Code (VS Code), é fundamental configurar corretamente o ambiente de desenvolvimento. Seguem os passos detalhados:

Instalação do JDK, Maven e Visual Studio Code

- **Java Development Kit (JDK):**
 - **Descrição:** O JDK é um conjunto de ferramentas necessárias para desenvolver e executar aplicações Java.
 - **Versão Recomendada:** JDK 17 ou superior (JDK 21 recomendada, em janeiro de 2025).
 - **Instalação:**
 - **Windows/Mac/Linux:**
 - Aceda ao site oficial do *Eclipse Adoptium* e descarregue o instalador adequado ao seu sistema operativo.
 - Siga as instruções do instalador para concluir a instalação.
 - **Verificação:**
 - Após a instalação, abra o terminal ou linha de comandos e execute:

```
java -version
```

- Deverá obter uma resposta indicando a versão instalada do Java.

- **Maven:**

- **Descrição:** O Maven é uma ferramenta de automação de compilação e gestão de dependências para projetos Java.

- **Instalação:**

- **Windows/Mac/Linux:**

- Aceda ao site oficial do [Apache Maven](#) e descarregue a versão mais recente.
 - Siga as instruções de instalação fornecidas na página para o seu sistema operativo.

- **Verificação:**

- Após a instalação, abra o terminal ou linha de comandos e execute:

```
mvn -version
```

- Deverá obter uma resposta indicando a versão instalada do Maven.

- **Visual Studio Code (VS Code):**

- **Descrição:** O VS Code é um editor de código-fonte leve e altamente extensível, ideal para desenvolvimento em diversas linguagens, incluindo Java.

- **Instalação:**

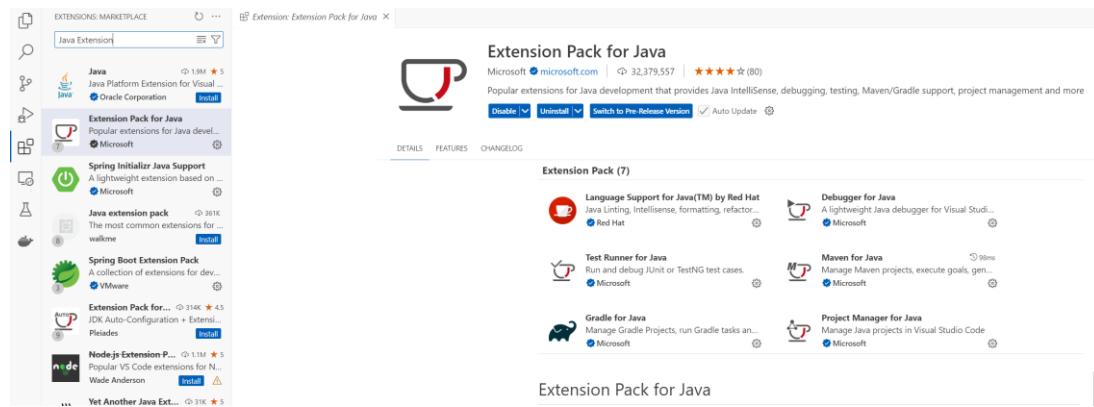
- **Windows/Mac/Linux:**

- Aceda ao site oficial do [Visual Studio Code](#) e descarregue o instalador adequado ao seu sistema operativo.
 - Siga as instruções do instalador para concluir a instalação.

Configuração das Extensões Necessárias no VS Code

Para aprimorar o desenvolvimento em Java no VS Code, é recomendável instalar as seguintes extensões:

- **Extension Pack for Java** (**já deve ter instalado, se seguiu o manual até agora**):
 - **Descrição:** Este pacote inclui um conjunto de extensões essenciais para o desenvolvimento Java, proporcionando suporte completo para edição, execução, depuração e testes de aplicações Java.
 - **Instalação:**
 - No VS Code, clique no ícone de extensões na barra lateral esquerda.
 - Pesquise por "Extension Pack for Java" e clique em "Instalar" (mas note, por favor, que deve **já ter instalado**, porque **fizemos isso no Capítulo 2**).
 - Este pacote inclui as seguintes extensões:
 - **Language Support for Java™ by Red Hat:** Fornece suporte à linguagem Java, incluindo realce de sintaxe, autocompletar e verificação de erros.
 - **Debugger for Java:** Permite depurar aplicações Java diretamente no VS Code.
 - **Java Test Runner:** Facilita a execução e depuração de testes unitários em Java.
 - **Maven for Java:** Oferece suporte integrado ao Maven, permitindo a gestão de dependências e execução de tarefas Maven.
 - **Java Dependency Viewer:** Permite visualizar e gerir as dependências do projeto de forma intuitiva.
 - **Visual Studio IntelliCode:** Fornece sugestões inteligentes de código baseadas em inteligência artificial.



- **Spring Boot Extension Pack:**

- **Descrição:** Este pacote é essencial para o desenvolvimento com Spring Boot, oferecendo ferramentas que facilitam a criação, execução e depuração de aplicações Spring Boot.
- **Instalação:**
 - No VS Code, na aba de extensões, pesquise por "Spring Boot Extension Pack" e clique em "Instalar".
 - Este pacote inclui as seguintes extensões:
 - **Spring Boot Tools:** Fornece suporte para funcionalidades específicas do Spring Boot, como execução e depuração de aplicações.
 - **Spring Initializr Java Support:** Facilita a criação de novos projetos Spring Boot através do Spring Initializr.
 - **Spring Boot Dashboard:** Oferece uma interface visual para gerir e monitorizar aplicações Spring Boot.

The screenshot shows the VS Code Extension Marketplace. On the left, there's a sidebar with icons for file operations like Open, Save, Find, and others. A search bar at the top has 'Spring' typed into it. Below the search bar, a list of extensions is shown, starting with 'Spring Boot Tools' and 'Spring Boot Extension Pack'. The 'Spring Boot Extension Pack' is highlighted with a blue border. To its right, a detailed view of the extension is displayed. The title is 'Spring Boot Extension Pack' by VMware, with a rating of 4.5 stars from 18 reviews. It's described as a collection of extensions for developing Spring Boot applications. Below this, there's a section for 'Extension Pack (3)' which includes 'Spring Boot Tools' and 'Spring Initializr Java Support'. At the bottom, there's a section for 'VS Code Spring Boot Application Development Extension Pack'.

Com estas configurações, o seu ambiente de desenvolvimento estará preparado para criar aplicações Java com Spring Boot de forma eficiente no Visual Studio Code.

Para mais informações e recursos adicionais, consulte a [documentação oficial do Visual Studio Code para Java](#).

Capítulo 5. Criação de um Projeto Spring Boot com Maven

Nota: Este capítulo é acompanhado por um **vídeo** com explicações passo-a-passo (ver aqui).

O que é o Maven

O **Maven** e o **Gradle** são ferramentas de automação de build amplamente utilizadas em projetos Java, ajudando a gerir dependências, compilar código e empacotar aplicações. Neste curso, utilizaremos o **Maven**, que é conhecido pela sua simplicidade e abordagem baseada em convenções, através do ficheiro de configuração pom.xml. O **Gradle**, por outro lado, é uma opção muito popular, especialmente em projetos Android, onde é a ferramenta de build padrão. Ambas as ferramentas estão disponíveis no **Spring Initializr**, uma aplicação web que facilita a configuração inicial de projetos Spring Boot, permitindo selecionar o tipo de ferramenta de build, as dependências e outras configurações iniciais. O Maven foi escolhido para este curso por ser tradicionalmente mais usado no contexto de aplicações Spring Boot, mas o Gradle também seria uma alternativa válida.

Para iniciar o desenvolvimento de uma aplicação Spring Boot utilizando o Maven no Visual Studio Code (VS Code), siga os passos detalhados que se descrevem seguidamente.

Utilização do Spring Initializr para Gerar o Projeto

O Spring Initializr é uma ferramenta que facilita a criação de projetos Spring Boot, permitindo configurar rapidamente as dependências e definições iniciais.

- **Através do VS Code (depois de ter instalado a extensão do Spring Boot referida no capítulo anterior):**
 - a. **Abrir a Paleta de Comandos:**
 - No VS Code, pressione *Ctrl + Shift + P* (ou *Cmd + Shift + P* no macOS) para abrir a Paleta de Comandos.
 - b. **Iniciar o Spring Initializr:**
 - Digite "Spring Initializr" na barra de pesquisa e selecione "Spring Initializr: Generate a Maven Project".
 - c. **Configurar o Projeto:**
 - Siga as instruções fornecidas para configurar o projeto:
 - **Group ID:** Por exemplo, *com.exemplo*.
 - **Artifact ID:** Por exemplo, *meu-projeto-springboot*.
 - **Versão do Spring Boot:** Selecione a versão mais recente e estável.
 - **Dependências:** Adicione as dependências necessárias, como "Spring Web" para desenvolvimento de aplicações web.
 - d. **Selecionar o Diretório de Destino:**
 - Escolha a pasta onde o projeto será criado e confirme.
 - e. **Importar o Projeto:**
 - Após a geração, o VS Code poderá solicitar a importação do projeto Maven.
 - Confirme a importação para que as dependências sejam resolvidas automaticamente.

Importação e Configuração no VS Code

- **Abrir o Projeto:**
 - Se o projeto não for aberto automaticamente, vá ao menu "Ficheiro" > "Abrir Pasta..." e selecione o diretório do projeto recém-criado.
- **Verificar as Dependências:**
 - No painel lateral, expanda a secção "Maven" para visualizar as dependências do projeto.
 - Certifique-se de que todas as dependências foram carregadas corretamente.
- **Configurar o JDK:**
 - Assegure-se de que o VS Code está configurado para utilizar o JDK correto:
 - Acesse as configurações (*CtrlL + ,* ou *Cmd + ,* no macOS).
 - Procure por "java.home" e defina o caminho para a instalação do JDK no seu sistema.
- **Executar a Aplicação:**
 - No explorador de arquivos, navegue até *src/main/java* e localize a classe principal anotada com *@SpringBootApplication*.
 - Clique com o botão direito do rato sobre o ficheiro e selecione "Run Java" para iniciar a aplicação.
- **Testar a Aplicação:**
 - Abra um navegador e aceda a *http://localhost:8080* para verificar se a aplicação está em execução.

Seguindo estes passos, terá um projeto Spring Boot configurado com Maven no VS Code, pronto para o desenvolvimento.

Outra forma de começar um projeto Spring Boot

Além da utilização da funcionalidade "Java: Create Java Project" no VS Code, uma das formas mais comuns e práticas de iniciar um projeto com Spring Boot é através do site **Spring Initializr** (start.spring.io). Este site fornece uma interface intuitiva para gerar um projeto Spring Boot personalizado.

2. Aceder ao Spring Initializr:

- Abra o navegador e aceda ao site start.spring.io.

3. Configurar o Projeto:

- Selecione as opções desejadas:

- **Project:** Escolha **Maven**.
- **Language:** Escolha **Java**.
- **Spring Boot Version:** Certifique-se de que seleciona a versão recomendada ou mais recente.
- **Project Metadata:** Preencha os campos como o **Group**, **Artifact**, e **Name**, por exemplo:
 - **Group ID:** Por exemplo, *com.exemplo*.
 - **Artifact ID:** Por exemplo, *meu-projeto-springboot*.
 - **Versão do Spring Boot:** Selecione a versão mais recente e estável.
 - **Dependências:** Adicione as dependências necessárias, como "Spring Web" para desenvolvimento de aplicações web.

4. Download do Projeto:

- Após configurar o projeto, clique em "**Generate**" para fazer o download do ficheiro *.zip*.

5. Importar no VS Code:

- Extraia o ficheiro *.zip* para uma pasta no seu computador.
- No VS Code, abra a pasta do projeto (*File > Open Folder*) e certifique-se de que as dependências são carregadas automaticamente pelo Maven.

Esta abordagem com o Spring Initializr é especialmente útil para configurar rapidamente os projetos com as dependências exatas que serão utilizadas, poupando tempo na configuração manual e garantindo que o projeto tem uma estrutura inicial consistente e adequada para o desenvolvimento.

Com estas duas opções (a partir do VS Code, ou no site Spring Initializr), @ programador@ pode escolher a abordagem que melhor se adequa às suas preferências ou ao fluxo de trabalho da sua equipa.

Capítulo 6. Estrutura de um Projeto Spring Boot

Nota: Este capítulo é acompanhado por um **vídeo** com explicações passo-a-passo (ver aqui).

Ao criar um projeto Spring Boot, uma estrutura de diretórios padrão é gerada para organizar o código e os recursos da aplicação de forma eficiente.

Análise das Pastas e Ficheiros Gerados

A estrutura típica de um projeto Spring Boot é a seguinte:

```
meu-projeto/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── exemplo/
│   │   │           └── MeuProjetoApplication.java
│   │   └── resources/
│   │       ├── application.properties
│   │       └── static/
│   │           └── templates/
│   └── test/
│       ├── java/
│       │   └── com/
│       │       └── exemplo/
│       │           └── MeuProjetoApplicationTests.java
│       └── resources/
└── .gitignore
└── mvnw
└── mvnw.cmd
└── pom.xml
└── README.md
```

Função de Cada Componente na Aplicação

- ***src/main/java/***: Contém o código-fonte principal da aplicação.
 - ***com/exemplo/MeuProjetoApplication.java***: Classe principal da aplicação, anotada com `@SpringBootApplication`, que serve como ponto de entrada.
- ***src/main/resources/***: Armazena recursos como ficheiros de configuração e templates.
 - ***application.properties***: Ficheiro de configuração da aplicação, onde se definem propriedades como parâmetros de conexão a bases de dados, portas de servidor, entre outros.
 - ***static/***: Diretório para recursos estáticos, como ficheiros CSS, JavaScript e imagens.
 - ***templates/***: Armazena templates Thymeleaf ou de outros motores de template para renderização de páginas dinâmicas.
- ***src/test/java/***: Contém as classes de teste para a aplicação.
 - ***com/exemplo/MeuProjetoApplicationTests.java***: Classe de teste gerada automaticamente para testar o contexto da aplicação.
- ***.gitignore***: Ficheiro que especifica quais ficheiros ou pastas devem ser ignorados pelo controlo de versão Git.
- ***mvnw* e *mvnw.cmd***: Scripts do Maven Wrapper que permitem construir o projeto sem necessidade de uma instalação prévia do Maven no sistema.
- ***pom.xml***: Ficheiro de configuração do Maven que define as dependências, plugins e outras configurações do projeto.
- ***README.md***: Ficheiro de documentação que geralmente contém informações sobre o projeto, como instruções de instalação e uso.

Esta organização modular facilita a manutenção e escalabilidade da aplicação, permitindo uma separação clara entre o código-fonte, recursos estáticos, templates e testes.

Capítulo 7. Configuração de Propriedades da Aplicação no Spring Boot

Nota: Este capítulo é acompanhado por um **vídeo** com explicações passo-a-passo (ver aqui).

No Spring Boot, a configuração da aplicação é centralizada em ficheiros de propriedades, permitindo uma gestão eficiente de parâmetros como portas de servidor, perfis de ambiente e outras definições essenciais. No **contexto deste Curso**, iremos utilizar o ficheiro **application.properties** para configurar as nossas aplicações Spring Boot. Este formato é simples e direto, baseando-se em pares chave-valor, o que facilita a aprendizagem e compreensão inicial. Apesar de o application.yml ser também uma opção válida, optamos pelo application.properties para garantir maior clareza e simplicidade nas configurações ao longo do curso.

Utilização do Ficheiro *application.properties* ou *application.yml*

O Spring Boot permite a configuração através de dois formatos principais:

- **application.properties**: Ficheiro de propriedades tradicional, com pares chave-valor.
 - *Exemplo:*

```
server.port=8081
spring.application.name=MinhaAplicacao
```
- **application.yml**: Ficheiro YAML que suporta uma estrutura hierárquica, facilitando a organização de configurações mais complexas.

- *Exemplo:*

```
server:  
  port: 8081  
spring:  
  application:  
    name: MinhaAplicacao
```

Independentemente de se utilizar o ficheiro `application.properties` ou `application.yml`, o local onde deve ser colocado o ficheiro de configuração é o diretório `src/main/resources/` do projeto, para serem corretamente reconhecidos pelo Spring Boot.

Definição de Portas, Perfis e Outras Configurações Essenciais

- **Configuração da Porta do Servidor:**

Por padrão, o Spring Boot inicia o servidor na porta 8080. Para alterar esta porta, adicione a seguinte propriedade:

- *No application.properties:*

```
server.port=8081
```

- *No application.yml:*

```
server:  
  port: 8081
```

Esta configuração define a porta na qual o servidor embutido irá escutar.

- **Configuração de Perfis de Ambiente:**

Perfis permitem definir diferentes configurações para diversos ambientes (desenvolvimento, teste, produção).

- **Definir o Perfil Ativo:**

- *No application.properties:*

```
spring.profiles.active=desenvolvimento
```

- *No application.yml:*

```
spring:
```

```
profiles:
```

```
active: desenvolvimento
```

- **Configurações Específicas por Perfil:**

Pode criar ficheiros de propriedades específicos para cada perfil, como *application-desenvolvimento.properties* ou *application-producao.yml*.

- *Exemplo no application-desenvolvimento.properties:*

```
server.port=8081
```

- *Exemplo no application-producao.yml:*

```
server:
```

```
port: 80
```

Ao ativar um perfil, o Spring Boot carrega as configurações correspondentes, permitindo uma adaptação dinâmica ao ambiente de execução.

- **Outras Configurações Essenciais:**

- **Nome da Aplicação:**

- *No application.properties:*

```
spring.application.name=MinhaAplicacao
```

- *No application.yml:*

```
spring:
```

```
application:
```

```
name: MinhaAplicacao
```

- **Configurações de Base de Dados:**

- *No application.properties:*

```
spring.datasource.url=jdbc:mysql://localhost:3306/meubanco
```

```
spring.datasource.username=usuario
```

```
spring.datasource.password=senha
```

- *No application.yml:*

```
spring:
```

```
datasource:
```

```
url: jdbc:mysql://localhost:3306/meubanco
```

```
username: usuario
```

```
password: senha
```

Estas configurações permitem definir parâmetros cruciais para o funcionamento da aplicação, assegurando que se comporta conforme o esperado em diferentes ambientes e cenários.

Para mais detalhes sobre as propriedades disponíveis e suas utilizações, consulte a [documentação oficial do Spring Boot](#).

Capítulo 8. Criação de Controladores REST no Spring Boot

Notas:	1. Este capítulo é acompanhado por um vídeo com explicações passo-a-passo (ver aqui).
	2. A descrição de como fazer o(s) exercício(s), com todos os passos, está no final do capítulo.
	3. O código com a resolução do(s) exercício(s) está no GitHub

No desenvolvimento de APIs RESTful com Spring Boot, os controladores são componentes fundamentais que gerem as requisições HTTP e produzem as respostas adequadas.

Implementação de Endpoints Utilizando `@RestController`

A anotação `@RestController` é uma especialização de `@Controller` que combina `@Controller` e `@ResponseBody`, indicando que os métodos da classe produzem respostas diretamente no corpo da resposta HTTP, em formato JSON ou XML, sem recorrer a views.

Exemplo de um controlador REST simples:

```
package com.exemplo.demo.controlador;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class SaudacaoControlador {

    @GetMapping("/saudacao")
    public String obterSaudacao() {
        return "Olá, bem-vindo à nossa API!";
    }
}
```

Neste exemplo, a classe `SaudacaoControlador` está anotada com `@RestController`, indicando que é um controlador REST. O método `obterSaudacao` é mapeado para o endpoint `/api/saudacao` e responde a requisições GET com uma mensagem de saudação.

Mapeamento de URLs com `@RequestMapping` e `@GetMapping`

O Spring Boot oferece diversas anotações para mapear URLs a métodos de controlador, facilitando a definição de endpoints na aplicação.

- **`@RequestMapping`**: Utilizada para mapear requisições HTTP a classes ou métodos específicos. Pode ser aplicada a nível de classe para definir um caminho base e a nível de método para caminhos específicos.

Exemplo:

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

    @RequestMapping("/produtos")
    public List<Produto> listarProdutos() {
        // Lógica para obter a lista de produtos
    }
}
```

Aqui, todas as requisições para `/api/produtos` serão tratadas pelo método `ListarProdutos`.

- **@GetMapping**: Uma especialização de **@RequestMapping** para o método HTTP GET. Simplifica o mapeamento de requisições GET a métodos específicos.

Exemplo:

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

    @GetMapping("/produtos")
    public List<Produto> listarProdutos() {
        // Lógica para obter a lista de produtos
    }
}
```

Neste caso, **@GetMapping** é utilizado para mapear requisições GET para o método *listarProdutos*.

Mapeamento de Variáveis de Caminho com **@PathVariable**

Para endpoints que requerem parâmetros dinâmicos na URL, utiliza-se **@PathVariable** para extrair valores do caminho da requisição.

Exemplo:

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

    @GetMapping("/produtos/{id}")
    public Produto obterProduto(@PathVariable Long id) {
        // Lógica para obter o produto pelo ID
    }
}
```

Aqui, *{id}* na URL representa uma variável de caminho que será passada ao método *obterProduto*.

Mapeamento de Parâmetros de Requisição com `@RequestParam`

Para capturar parâmetros de consulta na URL, utiliza-se `@RequestParam`.

Exemplo:

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

    @GetMapping("/produtos")
    public List<Produto> buscarProdutos(@RequestParam String nome) {
        // Lógica para buscar produtos pelo nome
    }
}
```

Neste exemplo, uma requisição para `/api/produtos?nome=Computador` passará o valor "Computador" para o parâmetro `nome` do método `buscarProdutos`.

Conclusão

A utilização das anotações `@RestController`, `@RequestMapping`, `@GetMapping`, `@PathVariable` e `@RequestParam` permite a criação de controladores REST de forma clara e eficiente no Spring Boot, facilitando o desenvolvimento de APIs robustas e bem estruturadas.

Para aprofundar o conhecimento sobre o mapeamento de URLs e a criação de controladores REST no Spring Boot, recomenda-se a leitura dos seguintes recursos:

- [Spring @RequestMapping - Baeldung](#)
- [Spring Boot - Trabalhando com Controllers | Blog da TreinaWeb](#)
- [Spring @GetMapping and @PostMapping with Examples - HowToDoInJava](#)

Estes artigos fornecem explicações detalhadas e exemplos práticos que complementam os conceitos apresentados.

Exercício 8.1

Passo 1: Criar o Projeto com Spring Initializr

6. Aceder ao Spring Initializr:

- Abra o navegador e aceda ao site oficial: start.spring.io.

7. Configurar o Projeto:

- **Project:** Escolha **Maven** (recomendado para iniciantes).
- **Language:** Selecione **Java**.
- **Spring Boot Version:** Escolha a versão estável mais recente (geralmente recomendada no site).
- **Group:** Introduza *com.example*.
- **Artifact:** Introduza *demo*.
- **Name:** Introduza *demo*.
- **Dependencies:** Clique no botão **Add Dependencies** e selecione:
 - **Spring Web** (necessário para criar endpoints REST).

8. Gerar o Projeto:

- Clique em **Generate** para descarregar o projeto como um ficheiro *.zip*.

9. Extrair o Projeto:

- Extraia o ficheiro *.zip* para uma pasta no seu computador, por exemplo, *C:\Projetos\demo*.
-

Passo 2: Abrir o Projeto no VS Code

10. Abrir o VS Code:

- Abra o **Visual Studio Code**.

11. Abrir a Pasta do Projeto:

- No menu, selecione **File > Open Folder** (ou **Abrir Pasta**).
- Navegue até à pasta onde extraiu o projeto (*C:\Projetos\demo*) e selecione-a.

12. Esperar que as Dependências sejam Carregadas:

- O VS Code detectará automaticamente o ficheiro *pom.xml* e iniciará o carregamento das dependências do Maven.
- Se solicitado, instale as extensões recomendadas para Java.

Estrutura do Diretório

Quando se cria um projeto Spring Boot, a estrutura típica será algo como:

```
src
└── main
    ├── java
    │   └── com.example.demo
    │       ├── controller
    │       ├── model
    │       └── DemoApplication.java
    └── resources
        ├── static
        ├── templates
        └── application.properties
```

Onde Colocar os Ficheiros:

Começamos por explicar o que cada ficheiro faz. A explicação detalhada de como criar cada ficheiro, dentro da pasta apropriada, é apresentada no passo seguinte ([Passo 3](#)).

Ficheiro *Produto* (Modelo):

`src/main/java/com/example/demo/model/Produto.java`

Este ficheiro pertence ao pacote *model*, que armazena classes relacionadas ao domínio da aplicação, como modelos de dados ou entidades.

Ficheiro *ProdutoControlador* (Controlador):

`src/main/java/com/example/demo/controller/ProdutoControlador.java`

Este ficheiro deve estar no pacote *controller*, que é responsável por gerir as requisições HTTP e fornecer as respostas adequadas.

Ficheiro *DemoApplication* (Classe Principal):

`src/main/java/com/example/demo/DemoApplication.java`

Esta classe é gerada automaticamente pelo Spring Initializr e contém o método *main*, que inicia a aplicação Spring Boot.

Ficheiros de Configuração (Opcional, para configurações adicionais):

src/main/resources/application.properties

Use este ficheiro para adicionar configurações da aplicação, como a porta do servidor ou outras propriedades.

Resumo da Organização

Ficheiro	Localização	Descrição
Produto.java	<i>src/main/java/com/example/demo/model</i>	Define o modelo de dados (classe do domínio).
ProdutoControlador.java	<i>src/main/java/com/example/demo/controller</i>	Gere as requisições HTTP (endpoints REST).
DemoApplication.java	<i>src/main/java/com/example/demo</i>	Classe principal que inicia a aplicação Spring Boot.
application.properties	<i>src/main/resources</i>	Configuração da aplicação (ex.: porta do servidor, propriedades adicionais).

Passo 3: Criar os Ficheiros do Projeto

Adicionar o Modelo (*Produto.java*):

No VS Code, navegue para a pasta *src/main/java/com/example/demo*.

Clique com o botão direito na pasta *demo*, selecione **New Folder** e crie uma subpasta chamada *model*.

Clique com o botão direito na nova pasta *model* e selecione **New File**.

Nomeie o ficheiro como *Produto.java*.

Copie e cole o seguinte código no ficheiro (página seguinte, para facilitar copy-paste):

```
package com.example.demo.model;

public class Produto {
    private Long id;
    private String nome;
    private double preco;

    public Produto(Long id, String nome, double preco) {
        this.id = id;
        this.nome = nome;
        this.preco = preco;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public double getPreco() {
        return preco;
    }

    public void setPreco(double preco) {
        this.preco = preco;
    }
}
```

Adicionar o Controlador (*ProdutoControlador.java*):

No VS Code, navegue para *src/main/java/com/example/demo*.

Clique com o botão direito na pasta *demo*, selecione **New Folder** e crie uma subpasta chamada *controller*.

Clique com o botão direito na nova pasta *controller* e selecione **New File**.

Nomeie o ficheiro como *ProdutoControlador.java*.

Copie e cole o seguinte código no ficheiro (página seguinte):

```

package com.example.demo.controller;

import com.example.demo.model.Produto;
import org.springframework.web.bind.annotation.*;

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

@RestController
@RequestMapping("/api/produtos")
public class ProdutoControlador {

    private List<Produto> produtos = new ArrayList<>();

    public ProdutoControlador() {
        produtos.add(new Produto(1L, "Computador", 1200.00));
        produtos.add(new Produto(2L, "TecLado", 50.00));
        produtos.add(new Produto(3L, "Rato", 25.00));
    }

    @GetMapping
    public List<Produto> ListarProdutos() {
        return produtos;
    }

    @GetMapping("/{id}")
    public Produto obterProduto(@PathVariable Long id) {
        return produtos.stream()
            .filter(produto -> produto.getId().equals(id))
            .findFirst()
            .orElse(null);
    }

    @PostMapping
    public Produto adicionarProduto(@RequestBody Produto produto) {
        produtos.add(produto);
        return produto;
    }

    @GetMapping("/filtrar")
    public List<Produto> filtrarProdutos(@RequestParam String nome) {
        return produtos.stream()
            .filter(produto ->
                produto.getNome().toLowerCase().contains(nome.toLowerCase()))
            .collect(Collectors.toList());
    }
}

```

Configuração Adicional (Opcional):

Verifique o ficheiro *application.properties* em *src/main/resources*. Por padrão, não é necessário alterar nada, mas pode definir a porta do servidor (ex.: *server.port=8081*).

Passo 4: Executar a Aplicação

Executar o Projeto:

No VS Code, abra a classe principal *DemoApplication.java* localizada em *src/main/java/com/example/demo/DemoApplication.java*.

Clique no botão **Run** (ou pressione **F5**) para iniciar o servidor.

Passo 5: Testar os Endpoints com o Postman

O **Postman** é uma ferramenta popular para testar APIs REST. Aqui estão os passos detalhados para testar os endpoints REST criados no exemplo:

1. Instalar o Postman

- Faça o download e instale o **Postman** a partir do site oficial: postman.com/downloads.
-

2. Abrir o Postman

- Após instalar, abra o Postman e siga os passos abaixo para testar cada endpoint.

3. Testar os Endpoints REST

(a) Listar todos os produtos

- **Método HTTP:** *GET*
- **URL:** *http://localhost:8080/api/produtos*
- **Passos no Postman:**
 - a. Clique em **New Request**.
 - b. Selecione o método *GET* no menu suspenso.
 - c. Introduza a URL no campo correspondente.
 - d. Clique em **Send**.
 - e. O Postman exibirá a lista de produtos em formato JSON, por exemplo:

```
[  
  {  
    "id": 1,  
    "nome": "Computador",  
    "preco": 1200.00  
  },  
  {  
    "id": 2,  
    "nome": "Teclado",  
    "preco": 50.00  
  },  
  {  
    "id": 3,  
    "nome": "Rato",  
    "preco": 25.00  
  }]
```

(b) Obter um produto pelo ID

- **Método HTTP:** *GET*
- **URL:** *http://localhost:8080/api/produtos/{id}*
- **Exemplo:** *http://localhost:8080/api/produtos/1*
- **Passos no Postman:**
 - Crie uma nova requisição e selecione o método *GET*.
 - Substitua *{id}* pelo ID desejado (ex.: *1*).
 - Clique em **Send**.
 - O Postman retornará o produto correspondente em formato JSON:

```
{  
    "id": 1,  
    "nome": "Computador",  
    "preco": 1200.00  
}
```

(c) Adicionar um novo produto

- **Método HTTP:** *POST*
- **URL:** *http://localhost:8080/api/produtos*
- **Corpo (Body):** Envie um objeto JSON com os dados do produto.
 - Exemplo:

```
{  
    "id": 4,  
    "nome": "Monitor",  
    "preco": 200.00  
}
```

- **Passos no Postman:**

- Crie uma nova requisição e selecione o método *POST*.
- Introduza a URL no campo correspondente.
- Clique no separador **Body**, selecione a opção **raw**, e no menu à direita escolha o formato **JSON**.
- Cole o JSON acima no campo de texto.
- Clique em **Send**.
- O Postman retornará o JSON do produto adicionado:

```
{  
    "id": 4,  
    "nome": "Monitor",  
    "preco": 200.00  
}
```

(d) Filtrar produtos pelo nome

- **Método HTTP:** *GET*
- **URL:** *http://localhost:8080/api/produtos/filtrar?nome={nome}*
- **Exemplo:** *http://localhost:8080/api/produtos/filtrar?nome=Computador*
- **Passos no Postman:**
 - Crie uma nova requisição e selecione o método *GET*.
 - Substitua *{nome}* pelo termo de pesquisa (ex.: *Computador*).
 - Clique em **Send**.
 - O Postman exibirá a lista de produtos que correspondem ao termo, por exemplo:

```
[  
  {  
    "id": 1,  
    "nome": "Computador",  
    "preco": 1200.00  
  }  
]
```

Dica Adicional

Se algum endpoint não funcionar, certifique-se de que:

- A aplicação Spring Boot está em execução no terminal.
- A URL e o método HTTP estão corretos.
- O servidor está configurado na porta *8080* (ou a porta definida no *application.properties*).

Este guia detalhado assegura que os utilizadores conseguem testar corretamente todos os endpoints REST com o Postman.

Solução do Exercício no Repositório do GitHub:

[Pasta Geral do Repositório](#)

[Pasta Específica do Exercício](#)

[Hiperligação para Descarregar Apenas Pasta do Exercício](#)

Capítulo 9. Introdução ao Spring Data JPA

Notas:	1. Este capítulo é acompanhado por um vídeo com explicações passo-a-passo (ver aqui).
	2. A descrição de como fazer o(s) exercício(s), com todos os passos, está no final do capítulo.
	3. O código com a resolução do(s) exercício(s) está no GitHub

O Spring Data JPA é um projeto do Spring que simplifica a implementação de repositórios baseados na JPA (Java Persistence API), facilitando o acesso a bases de dados relacionais e reduzindo a necessidade de código boilerplate.

Configuração de Acesso a Bases de Dados Relacionais

Para configurar o acesso a uma base de dados relacional no seu projeto Spring Boot, siga os passos abaixo:

1. Adicionar Dependências no *pom.xml*:

Certifique-se de que o seu ficheiro *pom.xml* inclui as dependências necessárias para o Spring Data JPA e o driver da base de dados que pretende utilizar. Por exemplo, para o PostgreSQL:

```
<dependencies>
    <!-- Dependência do Spring Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <!-- Dependência do driver PostgreSQL -->
```

```

<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
<!-- Outras dependências -->
</dependencies>

```

Esta configuração adiciona o suporte ao Spring Data JPA e ao driver do PostgreSQL ao seu projeto.

2. Configurar as Propriedades de Conexão:

No ficheiro *application.properties* ou *application.yml*, defina as propriedades de conexão à base de dados. Utilizando o formato *.properties*:

```

spring.datasource.url=jdbc:postgresql://localhost:5432/nome_da_base_de_dados
spring.datasource.username=seu_utilizador
spring.datasource.password=sua_senha
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

```

Estas propriedades configuram a URL de conexão, o nome de utilizador, a senha, o comportamento de atualização do esquema (*ddl-auto*) e a exibição das instruções SQL executadas.

Criação de Entidades e Repositórios

1. Definir Entidades JPA:

As entidades representam as tabelas da base de dados e são geralmente definidas como **POJOs** (*Plain Old Java Objects*), ou seja, classes Java simples que não dependem de frameworks externas, mas são enriquecidas com anotações JPA, como `@Entity`. Isto permite que a entidade seja mapeada para uma tabela na base de dados de forma clara e intuitiva.

Exemplo:

```
package com.exemplo.demo.entidade;

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

@Entity
public class Produto {

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

    private String nome;
    private Double preco;

    // Construtores, getters e setters
}
```

Neste exemplo, a classe `Produto` é uma entidade JPA que será mapeada para uma tabela `produto` na base de dados. O campo `id` é a chave primária, gerada automaticamente.

2. Criar Repositórios:

Os repositórios são interfaces que permitem a interação com a base de dados. O Spring Data JPA fornece a interface `JpaRepository` que pode ser estendida para criar repositórios específicos.

Exemplo:

```
package com.exemplo.demo.repository;

import com.exemplo.demo.entidade.Produto;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ProdutoRepository extends JpaRepository<Produto,
Long> {
    // Métodos de consulta personalizados podem ser definidos aqui
}
```

Ao estender `JpaRepository`, o `ProdutoRepository` herda métodos CRUD (Create, Read, Update, Delete) padrão, como `save()`, `findById()`, `findAll()`, `deleteById()`, entre outros.

Criação e Teste dos Outros Ficheiros

Até este ponto, procurou-se explicar os elementos essenciais para compreender o funcionamento do JPA no contexto de um projeto Spring Boot: a configuração da base de dados, a criação de um repositório e a definição de uma entidade na forma de um **POJO** (*Plain Old Java Object*). Este conceito refere-se a uma classe simples em Java que não depende de frameworks específicas, permitindo que as entidades sejam estruturadas de forma clara e intuitiva. Para evitar tornar o texto repetitivo e sobrecarregar a leitura, opta-se por deixar a descrição completa de todos os ficheiros na solução do exercício 9.1, disponível no GitHub, que complementa este capítulo com um guia mais detalhado.

Considerações Finais

Com estas configurações, a sua aplicação Spring Boot estará pronta para interagir com uma base de dados relacional utilizando o Spring Data JPA. Esta abordagem simplifica a implementação de operações de persistência, permitindo que concentre os seus esforços na lógica de negócio da aplicação.

Dado que o Java empresarial é amplamente utilizado no desenvolvimento de backends para aplicações baseadas em bases de dados, os temas tratados nesta secção têm uma importância especial. Por isso, serão explorados em maior profundidade no capítulo final dedicado ao Spring Boot neste manual.

Nesse capítulo, iremos abordar conceitos essenciais como ORM (Object-Relational Mapping), a criação e implementação de Serviços (Services), e o Mapeamento de Relacionamentos Entre Entidades. Isto inclui configurações avançadas como @OneToMany, @ManyToMany, e @ManyToOne, entre outros. Além disso, exploraremos tópicos mais avançados que, embora possam ir além do que seria esperado num curso introdutório, foram incluídos como referência futura devido à escassez de material em Português sobre o assunto.

Para complementar, foram criados exemplos práticos e acessíveis, disponíveis no GitHub, que demonstram estes conceitos de forma clara e aplicada. Assim, mesmo tópicos mais complexos podem ser compreendidos e utilizados com confiança no desenvolvimento das suas aplicações.

Exercício 9.1

1. Criar o Projeto com Spring Initializr

Aceder ao Spring Initializr:

Aceda a [Spring Initializr](#) no navegador (ou use o método de criar o Projeto no VS Code, como preferir)

Configurar o Projeto:

Project: Escolha **Maven**.

Language: Selecione **Java**.

Spring Boot Version: Use a mais recente e estável.

Group: Mantenha (ou insira) *com.example*.

Artifact: Mantenha (ou insira) *demo*.

Name: Mantenha (ou insira) *demo*.

Dependencies (lado direito): Adicione as seguintes:

- **Spring Web:** Para criar controladores REST.
- **Spring Data JPA:** Para integração com base de dados.
- **H2 Database:** Para usar uma base de dados embutida.

(nota: podíamos usar outra base de dados, como PostgreSQL, MySQL, MariaDB, MongoDB, etc, mas a base de dados H2 vem embutida no SpringBoot e evita a complexidade de gerir um SGBD independente: outro(s) exemplo(s) usaremos outros SGBD)

Versão de Java: A que tiver instalada no seu computador (pode abrir uma linha de comandos e escrever “java -version” para ver o número de versão: em janeiro de 2025, a versão 21 é a recomendada, mas por defeito está escolhida da 17).



Project

Gradle - Groovy Gradle - Kotlin Maven Java Kotlin Groovy

Spring Boot

3.4.2 (SNAPSHOT) 3.4.1 3.3.8 (SNAPSHOT) 3.3.7

Project Metadata

Group: com.example

Artifact: demo

Name: demo

Description: Solution for Intro Chapter on JPA

Package name: com.example.demo

Packaging: Jar War

Java: 23 21 17

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Web WEB

Build web, including RESTful applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JPA SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

H2 Database SQL

Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Gerar e Descarregar:

Não se esqueça de escolher as dependências (Spring Web, Spring Data JPA, e H2 Database).

Clique em **Generate** para descarregar o ficheiro *.zip*.

Extrair o Projeto:

Extraia o *.zip* para uma pasta, por exemplo, *C:\Projetos\demo*.

2. Abrir o Projeto no VS Code

3. Abrir o VS Code:

- Abra o Visual Studio Code.

4. Abrir a Pasta do Projeto:

- Vá em **File > Open Folder** (ou **Abrir Pasta**) e selecione a pasta extraída.

5. Esperar Carregamento de Dependências:

- O VS Code detectará o ficheiro *pom.xml* e carregará as dependências do Maven automaticamente.

3. Configurar a Base de Dados H2

Editar o Ficheiro *application.properties*:

Navegue até *src/main/resources/application.properties* e insira as configurações do H2:

```
spring.datasource.url=jdbc:h2:file:./data/testdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=update
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
spring.jpa.show-sql=true
```

Salvar o Ficheiro:

Estas configurações ativam o console do H2 e garantem persistência dos dados no disco (*./data/testdb*).

4. Criar as Classes Necessárias

Criar a Entidade *Produto*:

Local: *src/main/java/com/example/demo/model/Produto.java*

(Precisa de criar a pasta “model”, dentro de “demo” e colocar lá o ficheiro “Produto.java”)

Conteúdo:

```
package com.example.demo.model;

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

@Entity
public class Produto {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
    private Double preco;

    public Produto() {}

    public Produto(String nome, Double preco) {
        this.nome = nome;
        this.preco = preco;
    }

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

No repositório do GitHub, os Getters e Setters estão escritos. Aqui opta-se por não os colocar, porque ocupam muito espaço e é código muito repetitivo. Pode pedir ao VS Code para colocar os getters e os setters automaticamente: pressione **Ctrl+Shift+P** (ou **Cmd+Shift+P** no macOS) para abrir a **Paleta de Comandos**, e escreva "**Generate Getters and Setters**". Após escolher esta opção, uma janela pop-up aparecerá para selecionar os campos desejados (escolha todos).

Há mesmo uma biblioteca muito popular, chamada [Lombok](#) concebida para os programadores não serem obrigados a escrever getters, setters, e outros blocos de repetitivos deste tipo, tornando o código menos verboso e mais elegante.

Criar o Repositório *ProdutoRepository*:

Local: *src/main/java/com/example/demo/repository/ProdutoRepository.java*

(Precisa de criar a pasta “repository”, dentro de “demo” e colocar lá o ficheiro “ProdutoRepository.java”)

Conteúdo:

```
package com.example.demo.repository;

import com.example.demo.model.Produto;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ProdutoRepository extends
    JpaRepository<Produto, Long> {
}
```

Criar o Controlador *ProdutoControlador*:

Local: *src/main/java/com/example/demo/controller/ProdutoControlador.java*

(Precisa de criar a pasta “controller”, dentro de “demo” e colocar lá o ficheiro “ProdutoControlador.java”)

Conteúdo:

```
package com.example.demo.controller;

import com.example.demo.model.Produto;
import com.example.demo.repository.ProdutoRepositorio;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/produtos")
public class ProdutoControlador {

    private final ProdutoRepositorio produtoRepositorio;

    public ProdutoControlador(ProdutoRepositorio produtoRepositorio) {
        this.produtoRepositorio = produtoRepositorio;
    }

    @GetMapping
    public List<Produto> ListarProdutos() {
        return produtoRepositorio.findAll();
    }

    @GetMapping("/{id}")
    public ResponseEntity<Produto> obterProduto(@PathVariable Long id) {
        return produtoRepositorio.findById(id)
            .map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
    }

    @PostMapping
    public Produto adicionarProduto(@RequestBody Produto produto) {
        return produtoRepositorio.save(produto);
    }
}
```

Adicionar Dados de Teste:

Local: *src/main/java/com/example/demo/DatabaseLoader.java*

(Aqui só tem de criar o ficheiro “DatabaseLoader.java” diretamente na pasta “demo”)

Conteúdo:

```
package com.example.demo;

import com.example.demo.modelo.Produto;
import com.example.demo.repository.ProdutoRepository;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class DatabaseLoader {

    @Bean
    CommandLineRunner initDatabase(ProdutoRepository
produtoRepository) {
        return args -> {
            produtoRepository.save(new Produto("Teclado", 50.0));
            produtoRepository.save(new Produto("Rato", 30.0));
            produtoRepository.save(new Produto("Monitor", 200.0));
        };
    }
}
```

5. Testar o Projeto

Antes de realizar qualquer teste, é essencial **executar a aplicação Spring Boot** para garantir que o servidor está ativo e os endpoints estão disponíveis.

Passo 1: Executar a Aplicação

Abrir a Classe Principal:

No VS Code, navegue até `src/main/java/com/example/demo/DemoApplication.java`.

Executar a Aplicação:

Clique no texto **Run** por cima do método `main()`, ou no ícone de "play" que aparece na parte superior do editor. Também pode pressionar **F5** para iniciar.

Conferir a Execução no Terminal:

Verifique no terminal do VS Code se a aplicação foi iniciada com sucesso. Deve aparecer uma mensagem semelhante a esta:

Started DemoApplication in 2.345 seconds (JVM running for 2.564)

Confirmar a Porta do Servidor:

Por padrão, a aplicação Spring Boot é executada na porta **8080**. A URL base será:

`http://localhost:8080`

Passo 2: Verificar a Base de Dados H2

Aceder à Consola da Base de Dados H2:

No navegador, abra:

`http://localhost:8080/h2-console`

Insira as credenciais:

JDBC URL: `jdbc:h2:file:./data/testdb`

User: `sa`

Password: (deixe em branco).

Deve obter “Test successful”, num fundo verde, como indicado em baixo:

English ▾ Preferences Tools Help

Login

Saved Settings: Generic H2 (Embedded) ▾

Setting Name: Generic H2 (Embedded) Save Remove

Driver Class: org.h2.Driver

JDBC URL: jdbc:h2:file:./data/testdb

User Name: sa

Password:

Connect Test Connection

Test successful

Consultar os Dados:

No console H2, execute o comando:

```
SELECT * FROM PRODUTO;
```

Como indicado na imagem em baixo (faça “Run” para executar):

The screenshot shows the H2 Database Console interface. On the left, there is a sidebar with database connections and schemas: 'jdbc:h2:file:./data/testdb', 'PRODUTO', 'INFORMATION_SCHEMA', 'Users', and 'H2 2.3.232 (2024-08-11)'. The main area has a toolbar with 'Run', 'Run Selected', 'Auto complete', 'Clear', and 'SQL statement:'. Below the toolbar, the SQL statement 'SELECT * FROM PRODUTO;' is written. A preview window below displays the results of the query:

ID	NOME	PRECO
1	Teclado	50.0
2	Rato	30.0
3	Monitor	200.0

(3 rows, 1 ms)

An 'Edit' button is located at the bottom of the preview window.

Passo 3: Testar os Endpoints REST com o Postman

Abrir o Postman:

Certifique-se de que o Postman está instalado e aberto.

Configurar e Testar os Endpoints:

1. Listar Todos os Produtos

Método: GET

URL: <http://localhost:8080/api/produtos>

Resultado esperado:

```
[{"id":1, "nome":"Teclado", "preco":50.0}, {"id":2, "nome":"Rato", "preco":30.0}, {"id":3, "nome":"Monitor", "preco":200.0}]
```

2. Adicionar um Novo Produto

Método: POST

URL: <http://localhost:8080/api/produtos>

Body (JSON):

```
{  
    "nome": "Cadeira",  
    "preco": 150.0  
}
```

Resultado esperado:

```
{  
    "id": 4,  
    "nome": "Cadeira",  
    "preco": 150.0  
}
```

3. Consultar um Produto Específico

Método: GET

URL: <http://localhost:8080/api/produtos/4>

Resultado esperado:

```
{  
    "id": 4,  
    "nome": "Cadeira",  
    "preco": 150.0  
}
```

4. Atualizar um Produto

Método: PUT

URL: <http://localhost:8080/api/produtos/4>

Body (JSON):

```
{  
    "nome": "Cadeira Atualizada",  
    "preco": 160.0  
}
```

Resultado esperado:

```
{  
    "id": 4,  
    "nome": "Cadeira Atualizada",  
    "preco": 160.0  
}
```

5. Remover um Produto

Método: *DELETE*

URL: *http://localhost:8080/api/produtos/4*

Resultado esperado:

Código de estado **204 No Content**.

Notas Finais

- Certifique-se de que a aplicação **continua em execução** durante os testes.
- Verifique os logs no terminal para eventuais erros.
- Depois de testar os endpoints, pode confirmar os resultados consultando a base de dados H2 novamente.

Solução do Exercício 9.1 (no GitHub):

Código e solução passo-a-passo no GitHub

Restantes Exercícios do Capítulo:

Enquanto o Exercício 9.1 foi acompanhado de instruções detalhadas diretamente no manual, os exercícios seguintes, do 9.2 ao 9.6, terão as suas instruções completas disponíveis no repositório GitHub, no link associado a cada um dele, em baixo.

Esta abordagem foi escolhida para manter o manual conciso e evitar repetição excessiva, ao mesmo tempo que se procura proporcionar uma ampla variedade de exemplos práticos

A hiperligação oficial para o repositório completo do curso é esta.

As hiperligações em baixo permitem descarregar apenas a pasta com os ficheiros de cada exercício (usando a ferramenta “download-directory” da equipa “Refined GitHub”). Como é uma ferramenta não-oficial, se deixar de funcionar, aceda ao código usando a hiperligação oficial, dada acima.

Exercício 9.2 (Gestão de Clientes e Pedidos – Conceitos de JPA)

Código e solução passo-a-passo no GitHub

Exercício 9.3 (API para Gestão de Livros, sem UI)

Código e solução passo-a-passo no GitHub

Exercício 9.4 (UI em HTML/CSS/JS para API de Gestão de Livros)

Código e solução passo-a-passo no GitHub

Exercício 9.5 (UI em ReactJS para API de Gestão de Livros)

Código e solução passo-a-passo no GitHub

Exercício 9.6 (UI em Vaadin Flow para API de Gestão de Livros)

Código e solução passo-a-passo no GitHub

Capítulo 10. Injeção de Dependências e Componentes no Spring Boot

Nota: Este capítulo é acompanhado por um **vídeo** com explicações passo-a-passo (ver aqui).

No desenvolvimento de aplicações com Spring Boot, a Inversão de Controlo (IoC) e a Injeção de Dependências (DI) são conceitos fundamentais que promovem um código mais modular, testável e de fácil manutenção.

Conceito de Inversão de Controlo (IoC)

A Inversão de Controlo é um princípio de design onde a responsabilidade de criar e gerir as dependências dos objetos é delegada a um container ou framework, em vez de ser gerida pelo próprio código da aplicação. No contexto do Spring, o container IoC é responsável por instanciar, configurar e gerir o ciclo de vida dos objetos, conhecidos como *beans*. Este padrão promove um baixo acoplamento e facilita a substituição de implementações, melhorando a flexibilidade e testabilidade do código.

Utilização de Anotações como `@Autowired` e `@Component`

O Spring Boot utiliza diversas anotações para facilitar a configuração e gestão de dependências.

- **`@Component`:** Anotação que indica que uma classe é um componente gerido pelo Spring. Classes anotadas com `@Component` são automaticamente detectadas e registadas pelo container IoC durante a varredura de componentes (*component scanning*).

Exemplo:

```
package com.exemplo.servico;

import org.springframework.stereotype.Component;

@Component
public class ServicoEmail {
    // Implementação do serviço de email
}
```

Neste exemplo, a classe *ServicoEmail* é marcada como um componente Spring, tornando-a elegível para injeção em outras partes da aplicação.

- **@Autowired:** Anotação utilizada para injetar automaticamente dependências em componentes geridos pelo Spring. Pode ser aplicada a campos, construtores ou métodos setter.

Exemplos:

- **Injeção em Campo:**

```
@Component
public class ServicoNotificacao {

    @Autowired
    private ServicoEmail servicoEmail;

    // Métodos que utilizam servicoEmail
}
```

Aqui, o *ServicoEmail* é injetado diretamente no campo *servicoEmail*.

- **Injeção via Construtor:**

```
@Component
public class ServicoNotificacao {

    private final ServicoEmail servicoEmail;

    @Autowired
    public ServicoNotificacao(ServicoEmail servicoEmail) {
        this.servicoEmail = servicoEmail;
    }

    // Métodos que utilizam servicoEmail
}
```

Neste caso, o *ServicoEmail* é injetado através do construtor da classe.

- **Injeção via Método Setter:**

```
@Component
public class ServicoNotificacao {

    private ServicoEmail servicoEmail;

    @Autowired
    public void setServicoEmail(ServicoEmail servicoEmail) {
        this.servicoEmail = servicoEmail;
    }

    // Métodos que utilizam servicoEmail
}
```

Aqui, a injeção é realizada através de um método setter.

Considerações Importantes:

- **Autowiring por Tipo:** O `@Autowired` realiza a injeção com base no tipo da classe. Se houver mais de uma implementação do mesmo tipo, pode ocorrer ambiguidade. Para resolver isso, utiliza-se a anotação `@Qualifier` para especificar qual bean deve ser injetado.
- **Ciclo de Vida dos Beans:** O container Spring gera o ciclo de vida dos beans, incluindo a sua criação, inicialização e destruição, conforme as necessidades da aplicação.
- **Testabilidade:** A injeção de dependências facilita a substituição de implementações reais por mocks ou stubs durante os testes, permitindo a realização de testes unitários mais eficazes.

A compreensão e utilização adequadas de IoC e DI no Spring Boot são essenciais para o desenvolvimento de aplicações robustas e de fácil manutenção.

Para aprofundar o conhecimento sobre Inversão de Controlo e Injeção de Dependências no Spring, recomenda-se a leitura dos seguintes recursos:

- [*Inversão de Controle e Injeção de Dependência com Spring | Baeldung*](#)
- [*Spring @Autowired Annotation - GeeksforGeeks*](#)
- [*Spring @Component Annotation - Baeldung*](#)

Estes artigos fornecem explicações detalhadas e exemplos práticos que complementam os conceitos apresentados.

Capítulo 11. Testes Unitários com Spring Boot

Nota: Este capítulo é acompanhado por um **vídeo** com explicações passo-a-passo (ver aqui).

Os testes unitários são fundamentais para assegurar a qualidade e a robustez das aplicações. No contexto do Spring Boot, frameworks como JUnit e Mockito facilitam a criação e execução desses testes, permitindo verificar o comportamento de componentes individuais de forma isolada.

Configuração do Ambiente de Testes

1. Dependências Necessárias:

No ficheiro *pom.xml* do seu projeto Maven, certifique-se de que as seguintes dependências estão incluídas:

```
<dependencies>
    <!-- Dependência do Spring Boot para testes -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <!-- Outras dependências -->
</dependencies>
```

A dependência *spring-boot-starter-test* inclui o JUnit, Mockito e outras bibliotecas úteis para testes, proporcionando um ambiente completo para a criação de testes unitários.

2. Estrutura de Diretórios para Testes:

No diretório `src/test/java/`, mantenha uma estrutura de pacotes semelhante à do código-fonte principal (`src/main/java/`). Por exemplo, se a sua classe de serviço estiver em `com.exemplo.servico.MinhaClasse`, o teste correspondente deve estar em `src/test/java/com/exemplo/servico/MinhaClasseTest.java`.

Criação de Testes para Controladores e Serviços

1. Testes de Serviços:

Para testar serviços, é comum utilizar o Mockito para criar *mocks* das dependências, permitindo isolar a lógica do serviço.

Exemplo:

```
package com.exemplo.servico;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.when;

import com.exemplo.repositorio.ProdutoRepositorio;
import com.exemplo.entidade.Produto;

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

public class ProdutoServicoTest {

    @Mock
    private ProdutoRepositorio produtoRepositorio;

    @InjectMocks
    private ProdutoServico produtoServico;
```

```

    @BeforeEach
    public void setup() {
        MockitoAnnotations.openMocks(this);
    }

    @Test
    public void testObterProdutoPorId() {
        Produto produto = new Produto(1L, "Produto Teste", 10.0);

        when(produtoRepositorio.findById(1L)).thenReturn(Optional.of(produto));
    }

    Produto resultado = produtoServico.obterProdutoPorId(1L);

    assertEquals("Produto Teste", resultado.getNome());
    assertEquals(10.0, resultado.getPreco());
}
}

```

Neste exemplo, o `ProdutoRepositorio` é mockado para simular o comportamento esperado, permitindo testar o método `obterProdutoPorId` do `ProdutoServico` de forma isolada.

2. Testes de Controladores:

Para testar controladores REST, o Spring Boot oferece suporte para testes *mock* de MVC, permitindo simular requisições HTTP e verificar as respostas.

Exemplo:

```

package com.exemplo.controlador;

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.jsonPath;

import com.exemplo.servico.ProdutoServico;
import com.exemplo.entidade.Produto;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.web.servlet.MockMvc;

import java.util.Optional;

import static org.mockito.Mockito.when;

@WebMvcTest(ProdutoControlador.class)
public class ProdutoControladorTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private ProdutoServico produtoServico;

    @Test
    public void testObterProdutoPorId() throws Exception {
        Produto produto = new Produto(1L, "Produto Teste", 10.0);
        when(produtoServico.obterProdutoPorId(1L)).thenReturn(produto);

        mockMvc.perform(get("/api/produtos/1"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.nome").value("Produto Teste"))
    }
}

```

```
        .andExpect(jsonPath("$.preco").value(10.0));
    }
}
```

Aqui, o *ProdutoServico* é mockado para retornar um produto específico, e o *MockMvc* é utilizado para simular uma requisição GET ao endpoint */api/produtos/1*, verificando se a resposta contém os dados esperados.

Considerações Finais

A implementação de testes unitários em aplicações Spring Boot é essencial para garantir a qualidade e a confiabilidade do software. Utilizando ferramentas como JUnit e Mockito, é possível criar testes eficazes para controladores e serviços, assegurando que cada componente funcione conforme o esperado.

Para aprofundar os seus conhecimentos, recomenda-se a leitura dos seguintes recursos:

- [*Testes realmente unitários no Spring Boot*](#)
- [*Como testar serviços, endpoints e repositórios com o SpringBoot*](#)
- [*Curso Iniciando com Testes com SpringBoot*](#)

Estes materiais oferecem explicações detalhadas e exemplos práticos que complementam os conceitos apresentados.

Capítulo 12. Princípios de Engenharia de Software no Spring Boot

Nota: Este capítulo é acompanhado por um **vídeo** com explicações passo-a-passo (ver aqui).

O Spring Boot é uma ferramenta poderosa que, quando utilizada em conjunto com boas práticas de engenharia de software, permite o desenvolvimento de aplicações robustas, escaláveis e de fácil manutenção.

Aplicação de Boas Práticas e Padrões de Design

A adoção de padrões de design comprovados é essencial para resolver problemas comuns de forma eficiente e promover um código mais limpo e organizado.

- **Padrão Singleton:** Garante que uma classe tenha apenas uma única instância durante o ciclo de vida da aplicação. No Spring, os beans são, por padrão, singletons, assegurando que componentes como serviços e repositórios sejam reutilizados em vez de recriados.
- **Padrão Factory:** Fornece uma interface para criar objetos, permitindo que subclasses decidam quais classes instanciar. No Spring, o padrão Factory é frequentemente utilizado para a criação de beans através de métodos anotados com `@Bean`.
- **Padrão Strategy:** Permite selecionar algoritmos em tempo de execução, promovendo flexibilidade. No Spring Boot, este padrão é utilizado em componentes como serviços de autenticação, onde diferentes estratégias podem ser aplicadas conforme necessário.

- **Padrão Template Method:** Define o esqueleto de um algoritmo na superclasse, permitindo que as subclasses implementem passos específicos. O Spring utiliza este padrão em diversos componentes, facilitando a reutilização de código e a implementação de novas funcionalidades sem necessidade de refatorações extensivas.

Separação de Preocupações e Organização Modular do Código

A separação de preocupações (*Separation of Concerns*) é um princípio que visa dividir a aplicação em partes distintas, cada uma responsável por uma funcionalidade específica. Esta abordagem facilita a manutenção, a escalabilidade e a compreensão do sistema.

- **Arquitetura em Camadas:** Dividir a aplicação em camadas, como apresentação, negócio e acesso a dados, assegura que cada camada tenha responsabilidades bem definidas, promovendo um baixo acoplamento e uma alta coesão.
- **Padrão MVC (Model-View-Controller):** O Spring Boot adota o padrão MVC para separar a lógica de negócio (Model), a interface com o utilizador (View) e o controle de fluxo (Controller), facilitando o desenvolvimento e a manutenção da aplicação.
- **Modularização:** Estruturar o projeto em módulos independentes permite que diferentes equipas trabalhem simultaneamente em componentes distintos, melhora a reutilização de código e facilita a gestão de dependências. No Spring Boot, é possível criar projetos multi-módulo utilizando o Maven, organizando o código de forma modular e escalável.
- **Arquitetura Hexagonal:** Também conhecida como Arquitetura de Portas e Adaptadores, esta abordagem promove a independência da lógica de negócio em relação a detalhes de implementação, como frameworks e bases de dados. No Spring Boot, a Arquitetura Hexagonal pode ser implementada para criar aplicações mais flexíveis e adaptáveis a mudanças.

Conclusão

A aplicação de princípios sólidos de engenharia de software, aliada ao uso de padrões de design e à organização modular do código, é fundamental para o sucesso de projetos desenvolvidos com Spring Boot. Estas práticas conduzem a sistemas mais robustos, fáceis de manter e preparados para evoluir conforme as necessidades do negócio.

Capítulo 13. Expansão sobre Spring Boot e Bases de Dados

Nota: Este capítulo é acompanhado por um **vídeo** com explicações passo-a-passo (ver aqui).

Introdução ao ORM no Spring Boot

O **Mapeamento Objeto-Relacional (ORM)** é uma técnica que permite converter dados entre sistemas incompatíveis, especificamente entre linguagens de programação orientadas a objetos e bases de dados relacionais. No contexto do desenvolvimento de aplicações Java, o ORM facilita a interação com a base de dados, permitindo que os programadores trabalhem com objetos Java em vez de escrever consultas SQL diretamente.

Definição de ORM e sua Importância no Desenvolvimento de Aplicações Java

O ORM atua como uma ponte entre o modelo de dados orientado a objetos e o modelo relacional das bases de dados. Ao utilizar ORM, os programadores podem mapear classes Java para tabelas de bases de dados, e atributos de classes para colunas, permitindo operações de criação, leitura, atualização e eliminação (CRUD) de forma mais intuitiva e alinhada com a programação orientada a objetos.

Vantagens de Utilizar ORM em Projetos Spring Boot

- **Produtividade Aumentada:** O ORM reduz a necessidade de escrever consultas SQL manuais, permitindo que os programadores se concentrem na lógica de negócio.
- **Manutenção Simplificada:** Alterações no modelo de dados podem ser geridas através de modificações nas classes Java correspondentes, mantendo a consistência entre o código e a base de dados.
- **Portabilidade:** O ORM abstrai as especificidades dos diferentes sistemas de gestão de bases de dados, facilitando a migração entre diferentes fornecedores de bases de dados.
- **Segurança:** Ao utilizar ORM, é possível prevenir vulnerabilidades como injeção de SQL, uma vez que as consultas são geradas de forma segura pelo framework.

Visão Geral das Ferramentas ORM Suportadas pelo Spring Boot, com Destaque para o Hibernate

O Spring Boot integra-se de forma eficaz com várias ferramentas ORM, proporcionando flexibilidade na escolha da solução mais adequada para cada projeto.

- **Hibernate:** É uma das implementações ORM mais populares no ecossistema Java. O Hibernate oferece uma vasta gama de funcionalidades, incluindo caching, gestão de transações e suporte a consultas avançadas através da HQL (Hibernate Query Language). No Spring Boot, o Hibernate é frequentemente utilizado como a implementação padrão do JPA (Java Persistence API).
- **Spring Data JPA:** Parte do ecossistema Spring, o Spring Data JPA simplifica o desenvolvimento de repositórios JPA, fornecendo funcionalidades como consultas derivadas e criação automática de consultas, facilitando a implementação de operações de acesso a dados.
- **MyBatis:** Embora não seja um ORM completo, o MyBatis permite mapear instruções SQL, procedimentos armazenados e expressões complexas para objetos Java, oferecendo maior controlo sobre as consultas executadas.

A integração do Spring Boot com estas ferramentas permite aos programadores selecionar a abordagem que melhor se adapta às necessidades do seu projeto, beneficiando de uma configuração simplificada e de uma forte comunidade de suporte.

Se quiser Saber Mais sobre ORM:

A utilização de ORM no desenvolvimento de aplicações Java com Spring Boot proporciona uma abordagem mais intuitiva e eficiente para a gestão de dados persistentes. Ao abstrair a complexidade das interações com a base de dados, o ORM permite que os programadores se concentrem na lógica de negócio, promovendo um desenvolvimento mais ágil e seguro.

Para aprofundar os seus conhecimentos sobre ORM e sua aplicação no Spring Boot, recomenda-se a leitura dos seguintes recursos:

- [*Spring Boot with Hibernate*](#)
- [*Mapeamento Objeto-Relacional \(ORM\) em Java: Simplificando o Acesso a Bancos de Dados*](#)
- [*O que é um ORM – o significado das ferramentas de mapeamento relacional de objetos de banco de dados*](#)

Estes materiais oferecem explicações detalhadas e exemplos práticos que complementam os conceitos apresentados.

Configuração do Ambiente para ORM no Spring Boot

Para integrar o ORM na sua aplicação Spring Boot, é essencial configurar corretamente as dependências e as propriedades de conexão com a base de dados. Vamos abordar cada um desses passos detalhadamente.

1. Revisão das Dependências Necessárias no *pom.xml* para Suporte a JPA e Hibernate

No Maven, o ficheiro *pom.xml* é utilizado para gerir as dependências do projeto. Para adicionar suporte ao JPA e ao Hibernate, inclua as seguintes dependências:

- **Spring Boot Starter Data JPA:** Esta dependência inclui as bibliotecas necessárias para trabalhar com JPA e Hibernate.
- **Driver da Base de Dados:** Dependendo do sistema de gestão de base de dados (SGBD) que pretende utilizar, será necessário adicionar a dependência correspondente ao driver.

Por exemplo, se estiver a utilizar o MySQL, o seu *pom.xml* deverá conter:

```
<dependencies>
    <!-- Dependência para Spring Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- Dependência para o driver do MySQL -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>

    <!-- Dependência para testes -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Se optar por outro SGBD, substitua a dependência do driver pelo correspondente ao seu banco de dados.

2. Configuração do Datasource e Propriedades de Conexão com a Base de Dados no *application.properties* ou *application.yml*

O Spring Boot utiliza ficheiros de propriedades para configurar diversas definições da aplicação, incluindo as informações de conexão com a base de dados.

- **Ficheiro *application.properties*:**

Para configurar a conexão com o MySQL, adicione as seguintes linhas:

```
# URL de conexão com a base de dados
spring.datasource.url=jdbc:mysql://localhost:3306/nome_da_base_de_dados

# Nome de utilizador da base de dados
spring.datasource.username=seu_utilizador

# Palavra-passe da base de dados
spring.datasource.password=sua_palavra_passe

# Driver JDBC a ser utilizado
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Estratégia de geração do esquema da base de dados
spring.jpa.hibernate.ddl-auto=update

# Dialeto do Hibernate para o MySQL
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

Certifique-se de substituir *nome_da_base_de_dados*, *seu_utilizador* e *sua_palavra_passe* pelos valores correspondentes à sua configuração.

- **Ficheiro *application.yml*:**

Alternativamente, se preferir utilizar YAML, a configuração seria:

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/nome_da_base_de_dados
    username: seu_utilizador
    password: sua_palavra_passe
    driver-class-name: com.mysql.cj.jdbc.Driver
  jpa:
    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQLDialect
```

Novamente, substitua os valores conforme a sua configuração.

Explicação das Propriedades Configuradas:

- *spring.datasource.url*: Define a URL de conexão JDBC para a base de dados.
- *spring.datasource.username* e *spring.datasource.password*: Especificam as credenciais de acesso à base de dados.
- *spring.datasource.driver-class-name*: Indica o driver JDBC a ser utilizado para a conexão.
- *spring.jpa.hibernate.ddl-auto*: Determina a estratégia de geração do esquema da base de dados. Valores comuns incluem:
 - *none*: Não faz nenhuma alteração no esquema.
 - *update*: Atualiza o esquema conforme as entidades definidas.
 - *create*: Cria o esquema a cada inicialização, eliminando os dados existentes.
 - *create-drop*: Cria o esquema no início e elimina-o no encerramento da aplicação.
- *spring.jpa.properties.hibernate.dialect*: Especifica o dialeto do Hibernate, que define as particularidades do SQL para o SGBD em uso.

Considerações Adicionais:

- **Pooling de Conexões**: O Spring Boot, por padrão, utiliza o HikariCP como pool de conexões, garantindo eficiência na gestão das conexões com a base de dados.
- **Segurança**: Evite armazenar credenciais sensíveis diretamente nos ficheiros de propriedades. Considere o uso de variáveis de ambiente ou serviços de gestão de segredos para proteger essas informações.
- **Perfis de Ambiente**: Utilize perfis do Spring (*spring.profiles.active*) para definir configurações específicas para diferentes ambientes (desenvolvimento, teste, produção), permitindo uma gestão mais flexível das propriedades.

Para mais detalhes sobre a configuração de propriedades no Spring Boot, consulte a [documentação oficial](#).

Com estas configurações, a sua aplicação Spring Boot estará preparada para utilizar JPA e Hibernate, permitindo o mapeamento objeto-relacional de forma eficiente e integrada.

Definição de Entidades com POJOs no Spring Boot

No contexto do Spring Boot e JPA (Java Persistence API), as entidades são representadas por POJOs (Plain Old Java Objects), que correspondem às tabelas da base de dados. Cada instância de uma entidade representa uma linha na tabela associada.

Conceito de POJOs e seu Papel como Entidades JPA

POJOs são classes Java simples que não dependem de qualquer framework específico, mantendo-se independentes e fáceis de utilizar. No JPA, os POJOs são enriquecidos com anotações que definem o mapeamento entre a classe e a tabela da base de dados, permitindo que o JPA gerencie a persistência desses objetos de forma transparente.

Utilização de Anotações JPA para Mapear Classes Java às Tabelas do Banco de Dados

As anotações JPA são utilizadas para configurar o mapeamento entre as classes Java e as tabelas da base de dados. As principais anotações incluem:

- `@Entity`: Indica que a classe é uma entidade JPA e será mapeada para uma tabela na base de dados.
- `@Table(name = "nome_tabela")`: Especifica o nome da tabela na base de dados; se omitida, o nome da classe será utilizado.
- `@Id`: Define o atributo que será a chave primária da entidade.
- `@GeneratedValue(strategy = GenerationType.IDENTITY)`: Configura a estratégia de geração automática de valores para a chave primária.
- `@Column(name = "nome_coluna")`: Especifica o nome da coluna correspondente ao atributo; se omitida, o nome do atributo será utilizado.

Exemplo de uma Entidade Simples:

```
import jakarta.persistence.*;  
  
@Entity  
@Table(name = "clientes")  
public class Cliente {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    @Column(name = "nome", nullable = false)
```

```

    private String nome;

    @Column(name = "email", unique = true)
    private String email;

    // Construtores, getters e setters
}

```

Mapeamento de Relacionamentos entre Entidades

Em sistemas reais, as entidades frequentemente possuem relacionamentos entre si. O JPA permite mapear esses relacionamentos utilizando anotações específicas:

- **@OneToOne**: Relacionamento um-para-um, onde uma entidade está associada a exatamente uma outra entidade.
- **@OneToMany**: Relacionamento um-para-muitos, onde uma entidade está associada a múltiplas instâncias de outra entidade.
- **@ManyToOne**: Relacionamento muitos-para-um, onde múltiplas instâncias de uma entidade estão associadas a uma única instância de outra entidade.
- **@ManyToMany**: Relacionamento muitos-para-muitos, onde múltiplas instâncias de uma entidade estão associadas a múltiplas instâncias de outra entidade.

Exemplos de Relacionamentos:

Relacionamento **@OneToOne**:

Suponha que cada cliente tenha um único endereço:

```

@Entity
public class Cliente {

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

    private String nome;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "endereco_id", referencedColumnName = "id")
    private Endereco endereco;

    // Construtores, getters e setters
}

```

```

@Entity
public class Endereco {

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

    private String rua;

    private String cidade;

    // Construtores, getters e setters
}

```

Relacionamento @OneToMany e @ManyToOne:

Um cliente pode ter múltiplas encomendas, e cada encomenda pertence a um cliente:

```

@Entity
public class Cliente {

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

    private String nome;

    @OneToMany(mappedBy = "cliente", cascade = CascadeType.ALL,
    orphanRemoval = true)
    private List<Encomenda> encomendas = new ArrayList<>();

    // Construtores, getters e setters
}

@Entity
public class Encomenda {

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

    private String descricao;

    @ManyToOne
    @JoinColumn(name = "cliente_id", nullable = false)

```

```

    private Cliente cliente;

    // Construtores, getters e setters
}

```

Relacionamento @ManyToMany:

Uma encomenda pode conter múltiplos produtos, e um produto pode estar presente em múltiplas encomendas:

```

@Entity
public class Encomenda {

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

    private String descricao;

    @ManyToMany
    @JoinTable(
        name = "encomenda_produto",
        joinColumns = @JoinColumn(name = "encomenda_id"),
        inverseJoinColumns = @JoinColumn(name = "produto_id")
    )
    private Set<Produto> produtos = new HashSet<>();

    // Construtores, getters e setters
}

@Entity
public class Produto {

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

    private String nome;

    private Double preco;

    @ManyToMany(mappedBy = "produtos")
    private Set<Encomenda> encomendas = new HashSet<>();
}

```

```
// Construtores, getters e setters  
}
```

Considerações Importantes:

- **Chave Estrangeira:** Utilize a anotação `@JoinColumn` para definir a coluna que representa a chave estrangeira no relacionamento.
- **Cascade:** O atributo `cascade` define operações em cascata que propagam alterações da entidade principal para as entidades relacionadas.
- **Orphan Removal:** O atributo `orphanRemoval = true` permite que entidades órfãs sejam automaticamente removidas.

Criação de Repositórios (Repositories) no Spring Boot

No desenvolvimento de aplicações Spring Boot, os repositórios desempenham um papel crucial na abstração das operações de acesso a dados, permitindo interagir com a base de dados de forma simples e eficiente.

Introdução ao Spring Data JPA e à Interface `JpaRepository`

O Spring Data JPA é um módulo do Spring que facilita a implementação de repositórios baseados na JPA (Java Persistence API). A interface `JpaRepository` é uma extensão das interfaces `CrudRepository` e `PagingAndSortingRepository`, fornecendo um conjunto completo de métodos para operações CRUD (Create, Read, Update, Delete), paginação e ordenação.

Criação de Repositórios para Operações CRUD Básicas

Para criar um repositório no Spring Boot, basta definir uma interface que estenda `JpaRepository`. Por exemplo, considerando uma entidade `Cliente`:

```
import org.springframework.data.jpa.repository.JpaRepository;  
  
public interface ClienteRepository extends JpaRepository<Cliente, Long> {  
    // Métodos personalizados podem ser adicionados aqui  
}
```

Neste exemplo, `Cliente` é a entidade que o repositório irá manipular, e `Long` é o tipo da chave primária da entidade. Ao estender `JpaRepository`, o `ClienteRepository` herda métodos como `save()`, `findById()`, `findAll()`, `deleteById()`, entre outros, permitindo realizar operações CRUD sem a necessidade de implementar esses métodos manualmente.

Definição de Métodos de Consulta Personalizados Utilizando a Convenção de Nomes do Spring Data JPA

O Spring Data JPA permite a criação de métodos de consulta personalizados baseados na convenção de nomes dos métodos. Ao definir um método no repositório com um nome específico, o Spring Data JPA gera automaticamente a implementação da consulta correspondente.

Exemplos de Métodos de Consulta Personalizados:

Consulta por Atributo Simples:

Para encontrar clientes pelo nome:

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

Este método recupera todos os clientes cujo atributo `nome` corresponde ao parâmetro fornecido.

Consulta com Condições Compostas:

Para encontrar clientes por nome e idade:



Financiado pela
União Europeia
NextGenerationEU

```
List<Cliente> findByNomeAndIdade(String nome, Integer idade);
```

Este método recupera clientes cujo *nome* e *idade* correspondem aos parâmetros fornecidos.

Consulta com Ordenação:

Para encontrar clientes por cidade, ordenados pelo nome:

```
List<Cliente> findByCidadeOrderByNomeAsc(String cidade);
```

Este método recupera clientes da cidade especificada, ordenados pelo nome em ordem ascendente.

Consulta com Condições de Comparação:

Para encontrar clientes com idade maior que um determinado valor:

```
List<Cliente> findByIdadeGreaterThan(Integer idade);
```

Este método recupera clientes cuja idade é superior ao valor fornecido.

Palavras-chave Comuns Utilizadas nas Convenções de Nomes:

- *And*: Combina condições.
- *Or*: Define condições alternativas.
- *Between*: Verifica se um valor está dentro de um intervalo.
- *LessThan* / *GreaterThan*: Compara valores numéricos ou datas.
- *Like*: Pesquisa por padrões em strings.
- *IsNull* / *IsNotNull*: Verifica se um atributo é nulo ou não.
- *OrderBy*: Define a ordenação dos resultados.

Estas convenções permitem a criação de consultas complexas de forma declarativa e intuitiva, sem a necessidade de escrever SQL manualmente. Para consultas mais elaboradas ou específicas, o Spring Data JPA também suporta o uso da anotação *@Query*, onde é possível definir consultas JPQL ou SQL nativas conforme necessário.

Para mais detalhes sobre a criação de repositórios e métodos de consulta personalizados, consulte a [documentação oficial do Spring Data JPA](#).

Para implementar uma funcionalidade de busca que retorne todas as entradas na base de dados contendo os termos introduzidos, como no exemplo "Luís Cunha" correspondendo a "Luís Simões da Cunha", podemos utilizar o Spring Data JPA com a anotação `@Query` e a cláusula `LIKE` do SQL.

Passos para Implementar a Busca:

Definir o Repositório com o Método de Busca Personalizado:

No repositório da entidade, utilize a anotação `@Query` para definir uma consulta JPQL que utilize a cláusula `LIKE` para verificar se os termos fornecidos estão contidos nos campos desejados.

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import java.util.List;

public interface PessoaRepository extends JpaRepository<Pessoa, Long> {

    @Query("SELECT p FROM Pessoa p WHERE " +
           "LOWER(p.nome) LIKE LOWER(CONCAT('%', :termo, '%')) " +
           "OR LOWER(p.sobrenome) LIKE LOWER(CONCAT('%', :termo,
            '%'))")
    List<Pessoa> buscarPorTermo(@Param("termo") String termo);
}
```

Explicação:

- `LOWER(p.nome)`: Converte o valor do campo `nome` para minúsculas, permitindo uma busca case-insensitive.
- `LIKE LOWER(CONCAT('%', :termo, '%'))`: Verifica se o termo fornecido está contido no campo, ignorando maiúsculas e minúsculas. Os caracteres % são curingas que representam zero ou mais caracteres antes e depois do termo.
- A consulta verifica tanto o campo `nome` quanto o campo `sobrenome`, retornando resultados que correspondam ao termo em qualquer um dos campos.

3. Utilizar o Método de Busca no Serviço ou Controlador:

No serviço ou controlador da aplicação, chame o método `buscarPorTermo` passando o termo de busca fornecido pelo utilizador.

```
@RestController
@RequestMapping("/pessoas")
public class PessoaController {
```

```

@Autowired
private PessoaRepository pessoaRepository;

@GetMapping("/buscar")
public List<Pessoa> buscarPessoas(@RequestParam String termo) {
    return pessoaRepository.buscarPorTermo(termo);
}

```

Explicação:

- O método `buscarPessoas` recebe um parâmetro `termo` da requisição HTTP e utiliza o repositório para buscar as pessoas cujos nomes ou sobrenomes contenham o termo fornecido.

Considerações Importantes:

- Busca por Múltiplos Termos:**

Para permitir a busca por múltiplos termos, como "Luís Cunha", é necessário dividir a string de entrada em palavras individuais e construir a consulta para verificar cada termo separadamente.

```

@Query("SELECT p FROM Pessoa p WHERE " +
        "LOWER(p.nome) LIKE LOWER(CONCAT('%', :termo1, '%')) " +
        "AND LOWER(p.sobrenome) LIKE LOWER(CONCAT('%', :termo2, '%'))")
List<Pessoa> buscarPorTermos(@Param("termo1") String termo1,
                               @Param("termo2") String termo2);

```

Neste exemplo, a consulta verifica se o primeiro termo está contido no nome e o segundo termo está contido no sobrenome.

- Performance:**

O uso de cláusulas `LIKE` com curingas (%) no início e no fim pode impactar a performance da consulta, especialmente em bases de dados grandes. Certifique-se de que os campos utilizados nas buscas estejam indexados para melhorar o desempenho.

- **Segurança:**

Ao construir consultas dinâmicas, evite a concatenação direta de strings para prevenir vulnerabilidades como SQL Injection. Utilize sempre parâmetros nomeados, como demonstrado nos exemplos acima.

Para mais detalhes sobre a criação de consultas personalizadas com `@Query`, consulte a [documentação oficial do Spring Data JPA](#).

Definir Serviços no Spring Boot

A **camada de serviço** no Spring Boot desempenha um papel crucial na arquitetura de uma aplicação, servindo como intermediária entre os **controladores** (controllers) e os **repositórios** (repositories).

Relação entre Serviços e Repositórios:

- **Encapsulamento da Lógica de Negócio:** Os serviços contêm a lógica de negócios da aplicação, enquanto os repositórios são responsáveis pelo acesso e manipulação dos dados no banco de dados. Essa separação permite que a lógica de negócios seja independente dos detalhes de persistência.
- **Injeção de Dependências:** Os serviços utilizam os repositórios para realizar operações de persistência. No Spring Boot, isso é feito por meio da injeção de dependências, geralmente utilizando a anotação `@Autowired` ou através de injeção pelo construtor.

Exemplo de Implementação:

4. Entidade `Produto`:

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

@Entity
public class Produto {

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

    private String nome;
    private Double preco;

    // Getters e setters
}
```

5. Repositório `ProdutoRepository`:

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

@Repository
public interface ProdutoRepository extends JpaRepository<Produto, Long> {
    // Métodos de consulta personalizados, se necessário
}
```

6. Serviço *ProdutoService*:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.Optional;

@Service
public class ProdutoService {

    private final ProdutoRepository produtoRepository;

    @Autowired
    public ProdutoService(ProdutoRepository produtoRepository) {
        this.produtoRepository = produtoRepository;
    }

    public List<Produto> ListarTodos() {
        return produtoRepository.findAll();
    }

    public Optional<Produto> buscarPorId(Long id) {
        return produtoRepository.findById(id);
    }

    public Produto salvar(Produto produto) {
        // Lógica de negócio adicional pode ser adicionada aqui
        return produtoRepository.save(produto);
    }

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

Considerações Importantes:

- **Anotações Específicas:** Embora `@Service` e `@Repository` sejam especializações de `@Component`, elas fornecem semântica clara e funcionalidades adicionais, como tratamento específico de exceções em `@Repository`.
- **Boas Práticas:** Utilizar anotações específicas como `@Service` e `@Repository` melhora a clareza e a manutenção do código, indicando claramente o papel de cada classe na aplicação.
- **Tratamento de Exceções:** A anotação `@Repository` também facilita o tratamento de exceções relacionadas à persistência, convertendo exceções específicas de banco de dados em exceções não verificadas do Spring.

Conclusão:

A interação entre serviços e repositórios no Spring Boot é fundamental para uma arquitetura bem estruturada, promovendo a separação de responsabilidades, facilitando a manutenção e permitindo a escalabilidade da aplicação.

A **arquitetura MVC (Model-View-Controller)** é um padrão de design de software que separa uma aplicação em três componentes principais: Modelo, Visão e Controlador. Essa divisão facilita a organização do código, a manutenção e a escalabilidade da aplicação.

Componentes do MVC:

7. Modelo (Model):

- **Responsabilidade:** Gerenciar os dados e a lógica de negócios da aplicação.
- **Funções:**
 - Acessar e manipular os dados, geralmente interagindo com o banco de dados.
 - Aplicar as regras de negócio.
 - Notificar a Visão sobre mudanças nos dados, permitindo a atualização da interface.

8. Visão (View):

- **Responsabilidade:** Apresentar os dados ao utilizador de forma adequada.
- **Funções:**
 - Exibir a interface gráfica ou textual para o utilizador.
 - Receber as interações do utilizador e encaminhá-las ao Controlador.
 - Atualizar a apresentação conforme as mudanças nos dados do Modelo.

9. Controlador (Controller):

- **Responsabilidade:** Intermediar a interação entre o Modelo e a Visão.
- **Funções:**
 - Receber as entradas do utilizador a partir da Visão.
 - Processar essas entradas, aplicando a lógica necessária.
 - Atualizar o Modelo com base nas interações do utilizador.
 - Solicitar à Visão que atualize a interface conforme as mudanças no Modelo.

Interação entre os Componentes:

- O **utilizador** interage com a **Visão** (por exemplo, clicando em botões ou inserindo dados).
- A **Visão** encaminha essas interações ao **Controlador**.
- O **Controlador** processa as entradas e, se necessário, faz alterações no **Modelo**.
- O **Modelo** atualiza seus dados e notifica a **Visão** sobre as mudanças.
- A **Visão** então atualiza a interface apresentada ao utilizador, refletindo o estado atual do **Modelo**.

Benefícios da Arquitetura MVC:

- **Separação de Responsabilidades:** Cada componente tem uma função bem definida, o que facilita o desenvolvimento e a manutenção.
- **Reutilização de Código:** Componentes podem ser reutilizados em diferentes partes da aplicação ou em projetos distintos.
- **Facilidade de Testes:** A separação permite testar cada componente isoladamente, melhorando a qualidade do software.
- **Escalabilidade:** A arquitetura modular facilita a adição de novas funcionalidades sem comprometer o sistema existente.

No contexto do **Spring Boot**, o padrão MVC é amplamente utilizado para o desenvolvimento de aplicações web. Os **Controladores** são implementados com classes anotadas com `@Controller` ou `@RestController`, que lidam com as requisições HTTP e interagem com os **Serviços** (camada de negócio) e os **Repositórios** (camada de acesso a dados). As **Visões** podem ser representadas por templates Thymeleaf, JSPs ou outros mecanismos de template, responsáveis por renderizar o conteúdo a ser exibido ao utilizador.

Para uma introdução mais detalhada ao padrão MVC, você pode consultar o artigo "[Introdução ao Padrão MVC](#)".

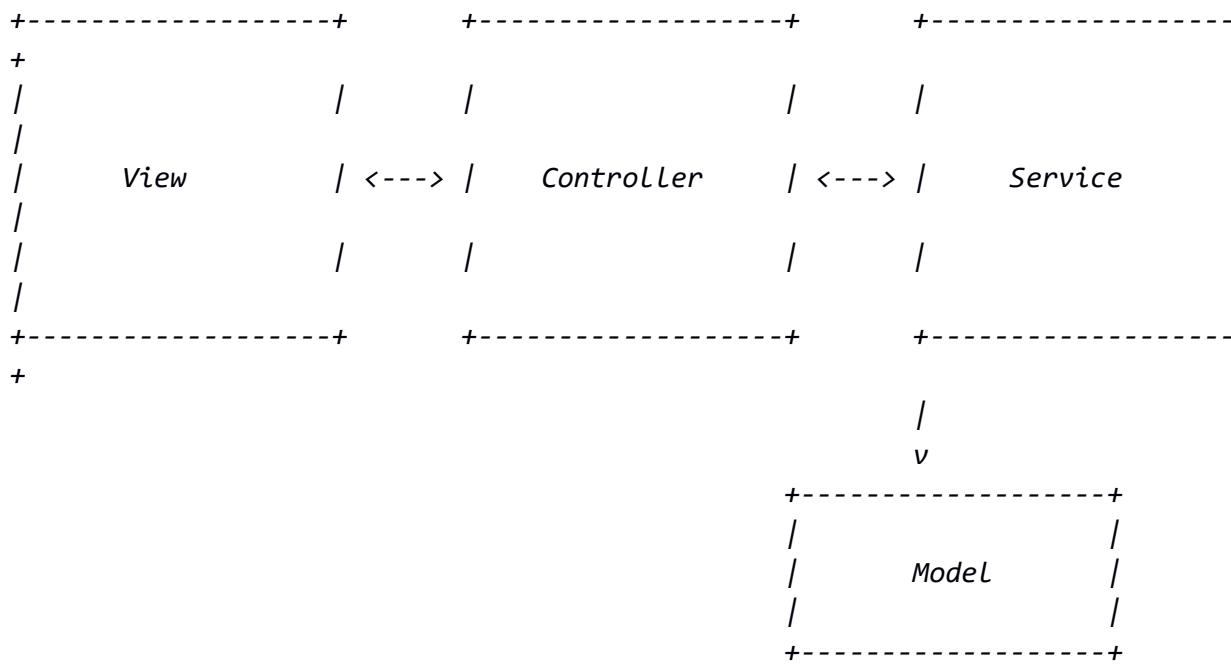
Portanto, arquitetura **Model-View-Controller (MVC)** organiza uma aplicação em três componentes principais:

1. **Modelo (Model):** Responsável pela lógica de negócio e acesso aos dados.
2. **Visão (View):** Encarregada da apresentação dos dados ao utilizador.
3. **Controlador (Controller):** Intermediário que processa as interações do utilizador, manipula o modelo e atualiza a visão.

No contexto do **Spring Boot**, a camada de **Serviço (Service)** é introduzida para encapsular a lógica de negócio, promovendo uma separação mais clara entre as responsabilidades.

O diagrama que se apresenta em seguida ilustra a arquitetura Model-View-Controller (MVC), adaptada ao contexto do Spring Boot. Este modelo organiza a aplicação em componentes bem definidos, com a adição da camada de Serviço (Service) para encapsular a lógica de negócio. Assim, o fluxo de interação entre o utilizador, a interface, e as camadas responsáveis pelo processamento e manipulação de dados é claramente estruturado, promovendo uma separação eficiente de responsabilidades.

Diagrama Representativo:



Descrição do Fluxo:

- Interação do Utilizador:** O utilizador interage com a **Visão** (por exemplo, preenchendo um formulário ou clicando num botão).
- Processamento pelo Controlador:** A **Visão** envia a solicitação ao **Controlador**, que interpreta a ação do utilizador.
- Lógica de Negócio no Serviço:** O **Controlador** delega a lógica de negócios à camada de **Serviço**, que contém as regras e operações específicas da aplicação.
- Interação com o Modelo:** O **Serviço** interage com o **Modelo** (entidades e repositórios) para acessar ou manipular os dados necessários.
- Atualização da Visão:** Após o processamento, o **Controlador** atualiza a **Visão** com as informações resultantes, refletindo as mudanças para o utilizador.

Papel dos Serviços na Arquitetura MVC

- **Encapsulamento da Lógica de Negócio:** Os **Serviços** isolam as regras de negócio, permitindo que o **Controlador** se concentre no fluxo de aplicação e na interação com a **Visão**.
- **Reutilização e Manutenção:** Ao centralizar a lógica de negócio nos **Serviços**, promove-se a reutilização de código e facilita-se a manutenção e testes da aplicação.
- **Interação com o Modelo:** Os **Serviços** interagem diretamente com o **Modelo**, realizando operações de leitura e escrita nos dados, geralmente por meio de repositórios.

Considerações:

- **Separação de Responsabilidades:** A introdução da camada de **Serviço** aprimora a separação de responsabilidades, tornando a aplicação mais modular e escalável.
- **Flexibilidade:** Essa estrutura permite que mudanças na lógica de negócio sejam implementadas nos **Serviços** sem impactar diretamente os **Controladores** ou a **Visão**.

Na arquitetura **Model-View-Controller (MVC)**, os **Controladores REST** desempenham um papel fundamental na exposição de dados e funcionalidades da aplicação por meio de APIs RESTful.

Criação de Controladores REST utilizando `@RestController`

No Spring Boot, a anotação `@RestController` é utilizada para definir classes que manipulam requisições HTTP e produzem respostas JSON ou XML, facilitando a criação de serviços web RESTful.

Exemplo de um Controlador REST:

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

import java.util.List;

@RestController
@RequestMapping("/api/produtos")
public class ProdutoController {

    @Autowired
    private ProdutoService produtoService;

    // Métodos para operações CRUD serão definidos aqui
}
```

Definição de Endpoints para Operações CRUD

As operações CRUD correspondem às ações de **Criar (Create)**, **Ler (Read)**, **Atualizar (Update)** e **Deletar (Delete)** recursos.

1. Criar um Novo Produto:

```
@PostMapping  
public ResponseEntity<Produto> criarProduto(@RequestBody Produto produto) {  
    Produto novoProduto = produtoService.salvar(produto);  
    return ResponseEntity.status(201).body(novoProduto);  
}
```

2. Listar Todos os Produtos:

```
@GetMapping  
public ResponseEntity<List<Produto>> listarProdutos() {  
    List<Produto> produtos = produtoService.listarTodos();  
    return ResponseEntity.ok(produtos);  
}
```

3. Buscar Produto por ID:

```
@GetMapping("/{id}")  
public ResponseEntity<Produto> buscarProdutoPorId(@PathVariable Long id) {  
    Produto produto = produtoService.buscarPorId(id);  
    return ResponseEntity.ok(produto);  
}
```

4. Atualizar Produto Existente:

```
@PutMapping("/{id}")  
public ResponseEntity<Produto> atualizarProduto(@PathVariable Long id,  
@RequestBody Produto produto) {  
    Produto produtoAtualizado = produtoService.atualizar(id, produto);  
    return ResponseEntity.ok(produtoAtualizado);  
}
```

5. Deletar Produto:

```
@DeleteMapping("/{id}")  
public ResponseEntity<Void> deletarProduto(@PathVariable Long id) {  
    produtoService.deletar(id);  
    return ResponseEntity.noContent().build();  
}
```

Manipulação de Respostas HTTP e Códigos de Status Apropriados

É essencial que os controladores REST retornem respostas HTTP com códigos de status adequados para refletir o resultado das operações:

- **201 Created:** Utilizado ao criar um novo recurso com sucesso.
- **200 OK:** Indica que a requisição foi bem-sucedida e o recurso solicitado está sendo retornado.
- **204 No Content:** Usado quando uma operação, como deletar, é bem-sucedida, mas não há conteúdo a ser retornado.
- **404 Not Found:** Indica que o recurso solicitado não foi encontrado.
- **400 Bad Request:** Utilizado quando a requisição é inválida ou contém erros.

Exemplo de Manipulação de Exceções:

```
@GetMapping("/{id}")
public ResponseEntity<Produto> buscarProdutoPorId(@PathVariable Long id) {
    try {
        Produto produto = produtoService.buscarPorId(id);
        return ResponseEntity.ok(produto);
    } catch (ProdutoNaoEncontradoException e) {
        return ResponseEntity.status(404).body(null);
    }
}
```

Considerações Finais

A implementação adequada de controladores REST no Spring Boot, utilizando `@RestController`, permite a criação de APIs RESTful robustas e eficientes, facilitando a interação com clientes e outras aplicações.

Para aprofundar seu conhecimento na criação de APIs RESTful com Spring Boot, você pode consultar o seguinte recurso:

- [CRUD REST utilizando Spring Boot 2, Hibernate, JPA, e MySQL](#)

Validação de Dados em Spring Boot

A **validação de dados** é uma etapa crucial no desenvolvimento de aplicações Spring Boot, garantindo que as informações fornecidas pelos utilizadores atendam aos critérios esperados antes de serem processadas ou armazenadas.

Utilização de Anotações de Validação nas Entidades

O Spring Boot integra o **Bean Validation**, permitindo a utilização de anotações diretamente nas classes de entidade para definir restrições nos campos.

Exemplo de Entidade com Anotações de Validação:

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;

@Entity
public class Usuario {

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

    @NotBlank(message = "O nome é obrigatório.")
    @Size(min = 2, max = 50, message = "O nome deve ter entre 2 e 50
caracteres.")
    private String nome;

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

    @NotNull(message = "A idade é obrigatória.")
    private Integer idade;

    // Getters e setters
}
```

Principais Anotações de Validação:

- `@NotNull`: Assegura que o campo não seja nulo.
- `@NotBlank`: Garante que o campo não seja nulo nem vazio, aplicável a strings.
- `@Size(min = x, max = y)`: Define o tamanho mínimo e máximo para strings, coleções ou arrays.
- `@Email`: Valida se o campo possui um formato de e-mail válido.

Configuração do Bean Validation no Spring Boot

Para habilitar o Bean Validation, é necessário adicionar a dependência correspondente no arquivo `pom.xml`:

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

Com essa dependência, o Spring Boot automaticamente reconhece e aplica as anotações de validação nas entidades.

Tratamento de Erros de Validação e Retorno de Mensagens Informativas ao Cliente

Ao receber dados de entrada, como em requisições HTTP, é fundamental validar os dados e fornecer feedback adequado ao cliente em caso de erros.

Exemplo de Controlador com Validação:

```
import org.springframework.http.ResponseEntity;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/usuarios")
public class UsuarioController {

    @PostMapping
    public ResponseEntity<Usuario> criarUsuario(@Validated @RequestBody
    Usuario usuario) {
        // Lógica para salvar o utilizador
        return ResponseEntity.status(201).body(usuario);
    }
}
```

Tratamento Global de Erros de Validação:

Para capturar e personalizar as respostas de erros de validação, pode-se utilizar a anotação `@ControllerAdvice` juntamente com métodos anotados com `@ExceptionHandler`.

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

import java.util.HashMap;
import java.util.Map;

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>>
    handleValidationExceptions(MethodArgumentNotValidException ex) {
        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult().getFieldErrors().forEach(error ->
            errors.put(error.getField(), error.getDefaultMessage())
        );
        return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
    }
}
```

Com essa configuração, quando ocorrer um erro de validação, o cliente receberá uma resposta contendo os campos inválidos e as respectivas mensagens de erro, facilitando a correção dos dados enviados.

Considerações Finais

A validação eficaz dos dados nas entidades assegura a integridade e consistência das informações na aplicação. Utilizando as anotações de validação e configurando o tratamento adequado de erros, é possível fornecer feedback claro e útil aos clientes, melhorando a experiência de uso e a robustez do sistema.

Para aprofundar seu conhecimento sobre validação de dados no Spring Boot, você pode consultar o seguinte recurso:

- [Validation in Spring Boot - Baeldung](#)

O tratamento eficaz de exceções numa API REST é essencial para fornecer respostas claras e consistentes aos clientes, melhorando a experiência do utilizador e facilitando a depuração. No Spring Boot, a anotação `@ControllerAdvice` permite a implementação de manipuladores de exceções globais, centralizando o tratamento de erros num único ponto.

Implementação de Manipuladores de Exceções Globais com `@ControllerAdvice`

A anotação `@ControllerAdvice` designa uma classe como um manipulador global de exceções para todos os controladores da aplicação. Dentro dessa classe, métodos anotados com `@ExceptionHandler` podem ser definidos para tratar tipos específicos de exceções.

Exemplo de Implementação:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;

import java.time.LocalDateTime;

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse>
    handleResourceNotFoundException(ResourceNotFoundException ex, WebRequest
request) {
        ErrorResponse errorResponse = new ErrorResponse(
            HttpStatus.NOT_FOUND.value(),
            ex.getMessage(),
            request.getDescription(false),
            LocalDateTime.now()
        );
        return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
    }

    // Outros manipuladores de exceções podem ser adicionados aqui
}
```

Personalização de Respostas de Erro para Diferentes Tipos de Exceções

Para fornecer respostas de erro informativas, é útil definir uma estrutura padrão para as mensagens de erro.

Definição de uma Classe de Resposta de Erro:

```
import java.time.LocalDateTime;

public class ErrorResponse {
    private int statusCode;
    private String message;
    private String details;
    private LocalDateTime timestamp;

    // Construtores, getters e setters
}
```

Com essa estrutura, é possível retornar informações detalhadas sobre o erro, incluindo o código de status HTTP, uma mensagem descritiva, detalhes adicionais e o timestamp do ocorrido.

Exemplo de Manipulador para Exceções Genéricas:

```
@ExceptionHandler(Exception.class)
public ResponseEntity<ErrorResponse> handleGlobalException(Exception ex,
WebRequest request) {
    ErrorResponse errorResponse = new ErrorResponse(
        HttpStatus.INTERNAL_SERVER_ERROR.value(),
        "Ocorreu um erro interno no servidor.",
        request.getDescription(false),
        LocalDateTime.now()
    );
    return new ResponseEntity<>(errorResponse,
    HttpStatus.INTERNAL_SERVER_ERROR);
}
```

Melhoria da Experiência do Cliente com Mensagens de Erro Claras e Consistentes

Ao centralizar o tratamento de exceções e personalizar as respostas de erro, a API REST oferece aos clientes mensagens mais claras e consistentes, facilitando a compreensão e resolução de problemas.

Benefícios:

- Consistência:** Todas as respostas de erro seguem um formato uniforme, independentemente do ponto da aplicação onde o erro ocorreu.
- Clareza:** Mensagens de erro descritivas ajudam os clientes a entenderem exatamente o que deu errado.

- **Depuração Facilitada:** Detalhes adicionais, como timestamps e descrições, auxiliam na identificação e correção de problemas.

Para aprofundar seu conhecimento sobre o tratamento de exceções no Spring Boot, você pode consultar os seguintes recursos:

- [*Como fazer tratamento de exceções globalmente em Java Spring Boot*](#)
- [*Tratamento de erros personalizados para APIs REST com Spring Boot*](#)

Paginação e ordenação

A **paginação e ordenação** são fundamentais em APIs REST para gerenciar grandes volumes de dados, melhorando a performance e a experiência do utilizador. No Spring Boot, o **Spring Data JPA** oferece suporte integrado para essas funcionalidades por meio das classes *Pageable* e *Page*.

Implementação de Paginação e Ordenação nos Endpoints REST

Para implementar paginação e ordenação, é necessário ajustar o repositório e o controlador da aplicação.

1. Ajuste do Repositório:

Certifique-se de que o repositório estenda *JpaRepository*, que já inclui suporte a paginação e ordenação.

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

public interface ProdutoRepository extends JpaRepository<Produto, Long> {
    // Métodos de consulta adicionais, se necessário
}
```

2. Ajuste do Controlador:

No controlador, utilize os parâmetros `page`, `size` e `sort` nas requisições para controlar a paginação e a ordenação.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/produtos")
public class ProdutoController {

    @Autowired
    private ProdutoRepository produtoRepository;

    @GetMapping
    public Page<Produto> listarProdutos(
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "10") int size,
        @RequestParam(defaultValue = "id,asc") String[] sort) {

        Sort.Direction direction = Sort.Direction.fromString(sort[1]);
        Pageable pageable = PageRequest.of(page, size, Sort.by(direction,
        sort[0]));
        return produtoRepository.findAll(pageable);
    }
}
```

Utilização das Classes `Pageable` e `Page` do Spring Data JPA

- **`Pageable`:** Interface que encapsula informações sobre a página solicitada, como número da página, tamanho e ordenação.
- **`Page`:** Interface que representa uma página de dados, contendo a lista de elementos e informações adicionais, como total de páginas e total de elementos.

Configuração de Parâmetros de Paginação e Ordenação nas Requisições HTTP

Os parâmetros de paginação e ordenação podem ser configurados diretamente nas requisições HTTP:

- **page**: Número da página (iniciando em 0).
- **size**: Quantidade de registros por página.
- **sort**: Campo e direção para ordenação, no formato *campo, direção* (por exemplo, *nome, asc* ou *preco, desc*).

Exemplo de Requisição HTTP:

`GET /api/produtos?page=1&size=5&sort=nome,asc`

Essa requisição retorna a segunda página (índice 1) com 5 produtos por página, ordenados pelo nome em ordem crescente.

Considerações Finais sobre Paginação e Ordenação:

A implementação de paginação e ordenação nos endpoints REST melhora a eficiência da aplicação e proporciona uma melhor experiência ao utilizador, permitindo o consumo de dados de forma mais controlada e organizada.

Para aprofundar seu conhecimento sobre paginação e ordenação no Spring Boot, você pode consultar os seguintes recursos:

- [Spring Boot Pagination and Sorting Example - HowToDoInJava](#)
- [Spring Boot Pagination and Sorting example - BezKoder](#)

A realização de **testes de integração** em aplicações Spring Boot é fundamental para assegurar que os diversos componentes do sistema funcionem harmoniosamente. Esses testes validam a interação entre módulos, como repositórios, serviços e controladores REST, garantindo a integridade da aplicação.

Configuração do Ambiente de Testes para a Camada de Persistência

Para configurar o ambiente de testes, é essencial incluir as dependências necessárias no arquivo `pom.xml`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
```



Financiado pela
União Europeia
NextGenerationEU

```
<artifactId>h2</artifactId>
<scope>test</scope>
</dependency>
```

A dependência *spring-boot-starter-test* inclui bibliotecas como JUnit, Mockito e Hamcrest, essenciais para a criação de testes. A inclusão do H2 permite a utilização de um banco de dados em memória durante os testes.

Criação de Testes para Repositórios, Serviços e Controladores REST

1. Testes de Repositórios:

Para testar a camada de persistência, utiliza-se a anotação `@DataJpaTest`, que configura um ambiente de teste otimizado para JPA.

```
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 ProdutoRepositoryTest {

    @Autowired
    private ProdutoRepository produtoRepository;

    @Test
    public void deveSalvarProduto() {
        Produto produto = new Produto("Produto Teste", 100.0);
        Produto salvo = produtoRepository.save(produto);
        assertThat(salvo.getId()).isNotNull();
    }
}
```

2. Testes de Serviços:

Para a camada de serviços, utiliza-se `@SpringBootTest` para carregar o contexto completo da aplicação.

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest
public class ProdutoServiceTest {

    @Autowired
    private ProdutoService produtoService;

    @Test
    public void deveRetornarProdutoPorId() {
```

```

        Produto produto = produtoService.buscarPorId(1L);
        assertThat(produto).isNotNull();
    }
}

```

3. Testes de Controladores REST:

Para testar controladores REST, o *MockMvc* é uma ferramenta eficaz que permite simular requisições HTTP.

```

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
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;

@SpringBootTest
@AutoConfigureMockMvc
public class ProdutoControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void deveRetornarListaDeProdutos() throws Exception {
        mockMvc.perform(get("/api/produtos"))
            .andExpect(status().isOk());
    }
}

```

Utilização de Banco de Dados em Memória (e.g., H2) para Testes

O H2 é um banco de dados em memória amplamente utilizado em testes por sua leveza e facilidade de configuração. Para configurá-lo, adicione as seguintes propriedades no *application.properties* ou *application.yml*:

```

spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true

```

Essa configuração direciona a aplicação para utilizar o H2 como banco de dados durante os testes, garantindo um ambiente isolado e controlado.

Considerações Finais

A implementação de testes de integração robustos assegura que os componentes da aplicação interajam corretamente, elevando a qualidade e a confiabilidade do software. Utilizar bancos de dados em memória, como o H2, facilita a criação de um ambiente de teste eficiente e reproduzível.

Para aprofundar seu conhecimento sobre testes de integração com Spring Boot, considere consultar os seguintes recursos:

- [*Implementando Testes de Integração com Spring Boot e JUnit*](#)
- [*Testes de integração com TestContainers e Spring Boot*](#)

Eficiência e Desempenho com Spring Boot

Para garantir o desempenho eficiente de uma aplicação Spring Boot que utiliza o Hibernate como provedor de persistência, é fundamental adotar estratégias adequadas de carregamento de dados, implementar mecanismos de cache e monitorar as consultas geradas.

Estratégias de Carregamento de Dados: Eager vs. Lazy Loading

No Hibernate, as associações entre entidades podem ser carregadas de duas maneiras:

- **Eager Loading (Carregamento Antecipado):** As associações são carregadas imediatamente junto com a entidade principal, resultando em consultas mais complexas e potencialmente trazendo dados desnecessários.
- **Lazy Loading (Carregamento Tardio):** As associações são carregadas sob demanda, ou seja, somente quando acessadas pela primeira vez, permitindo consultas mais leves e carregamento de dados apenas quando necessário.

Configuração no Código:

```
@Entity  
public class Pedido {  
  
    @OneToMany(fetch = FetchType.LAZY)  
    private List<Item> itens;  
  
    // getters e setters  
}
```

No exemplo acima, a coleção `itens` será carregada apenas quando acessada, devido ao `FetchType.LAZY`.

Boas Práticas:

- Utilize **Lazy Loading** para associações que nem sempre são necessárias na operação atual, evitando sobrecarga desnecessária.
- Empregue **Eager Loading** apenas quando tiver certeza de que a associação será sempre utilizada, garantindo que os dados estejam disponíveis imediatamente.

Utilização de Caches para Melhorar o Desempenho

O Hibernate oferece dois níveis de cache para otimizar o acesso aos dados:

- **First-Level Cache (Cache de Primeiro Nível):** Ativado por padrão, é específico para a sessão atual e armazena entidades já carregadas, evitando consultas repetidas durante a mesma sessão.

- **Second-Level Cache (Cache de Segundo Nível):** Compartilhado entre sessões, requer configuração adicional e é útil para dados frequentemente acessados, reduzindo a carga no banco de dados.

Configuração do Second-Level Cache:

1. Adicionar Dependência de um Provedor de Cache:

```
<dependency>
    <groupId>org.ehcache</groupId>
    <artifactId>ehcache</artifactId>
</dependency>
```

2. Configurar o Hibernate para Utilizar o Cache:

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

3. Anotar Entidades para Cache:

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage =
        CacheConcurrencyStrategy.READ_WRITE)
public class Produto {
    // campos, getters e setters
}
```

Monitorização e Otimização de Consultas Geradas pelo Hibernate

Para assegurar que o Hibernate está gerando consultas eficientes, é importante monitorar e otimizar as operações de banco de dados:

- Ativar Logs de Consultas:

```
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
Logging.Level.org.hibernate.SQL=DEBUG
Logging.Level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

Essas configurações permitem visualizar as consultas SQL geradas e os parâmetros utilizados, facilitando a identificação de possíveis ineficiências.

- Utilizar Ferramentas de Monitoramento:

Ferramentas como o **Hibernate Statistics** podem fornecer métricas detalhadas sobre o desempenho das consultas, número de execuções e tempo gasto, auxiliando na identificação de gargalos.

- **Evitar o Problema de N+1 Consultas:**

Ocorre quando, ao carregar uma entidade principal, o Hibernate executa consultas adicionais para cada entidade relacionada. Para mitigar esse problema, utilize **fetch joins** ou **entity graphs** para carregar as associações necessárias de forma eficiente.

Exemplo de Fetch Join:

```
String jpql = "SELECT p FROM Pedido p JOIN FETCH p.itens WHERE p.id = :id";
Pedido pedido = entityManager.createQuery(jpql, Pedido.class)
    .setParameter("id", pedidoId)
    .getSingleResult();
```

Considerações Finais

A adoção de estratégias adequadas de carregamento de dados, a implementação de caches eficientes e a monitorização contínua das consultas geradas pelo Hibernate são práticas essenciais para otimizar o desempenho de aplicações Spring Boot. A compreensão e aplicação dessas técnicas garantem uma interação mais eficiente com o banco de dados, resultando em sistemas mais responsivos e escaláveis.

Para aprofundar seu conhecimento sobre essas práticas, considere consultar os seguintes recursos:

- [Eager/Lazy Loading in Hibernate - Baeldung](#)
- [Hibernate Performance Tuning Tips - Vlad Mihalcea](#)
- [The JPA and Hibernate Second-Level Cache - Vlad Mihalcea](#)

Conclusão sobre Bases de Dados e ORM em Spring Boot

Concluímos nossa jornada pelo uso de **ORM (Mapeamento Objeto-Relacional)** com **Spring Boot**, abordando desde conceitos fundamentais até práticas avançadas que aprimoram a eficiência e a segurança das aplicações.

Recapitulação dos Conceitos Abordados

- Introdução ao ORM e Spring Boot:** Compreendemos como o ORM facilita a interação entre aplicações Java e bancos de dados relacionais, e como o Spring Boot simplifica a configuração e o desenvolvimento de aplicações robustas.
- Configuração do Spring Data JPA:** Exploramos a integração do Spring Data JPA para gerenciar operações de persistência, incluindo a configuração de dependências e propriedades essenciais.
- Definição de Entidades e Relacionamentos:** Aprendemos a mapear classes Java para tabelas de banco de dados, definindo relacionamentos como um-para-um, um-para-muitos e muitos-para-muitos.
- Criação de Repositórios:** Implementamos interfaces de repositório para realizar operações CRUD, aproveitando a simplicidade e a flexibilidade do Spring Data JPA.
- Implementação de Serviços:** Discutimos a importância da camada de serviço na arquitetura, encapsulando a lógica de negócio e interagindo com os repositórios.
- Exposição de Dados via REST Controllers:** Desenvolvemos controladores REST para expor endpoints que permitem a interação com as entidades, seguindo o padrão arquitetural MVC.
- Validação de Dados nas Entidades:** Utilizamos anotações de validação para garantir a integridade dos dados, fornecendo feedback claro em caso de erros.
- Tratamento de Exceções na API REST:** Implementamos manipuladores globais de exceções para fornecer respostas consistentes e informativas aos clientes.
- Paginação e Ordenação de Resultados:** Aplicamos técnicas de paginação e ordenação para otimizar a apresentação de grandes volumes de dados.
- Testes de Integração com Spring Boot:** Configuramos testes de integração para assegurar o funcionamento correto dos componentes da aplicação, utilizando bancos de dados em memória.
- Considerações sobre Desempenho e Boas Práticas:** Adotamos estratégias de carregamento de dados, implementamos caches e monitoramos consultas para melhorar o desempenho da aplicação.

Capítulo 14. Segurança em Spring

Nota: Este capítulo é acompanhado por um **vídeo** com explicações passo-a-passo (ver aqui).

A segurança é um componente essencial no desenvolvimento de aplicações web, assegurando a proteção de dados sensíveis e a integridade dos sistemas. No contexto do Spring Boot, o **Spring Security** destaca-se como um framework robusto e flexível para a implementação de mecanismos de autenticação e autorização.

Introdução ao Spring Security e Implementação Básica

Conceito de Segurança em Aplicações Web

A segurança em aplicações web visa proteger informações confidenciais e garantir que apenas utilizadores autorizados acedam a recursos específicos. Os principais desafios incluem a prevenção de acessos não autorizados, proteção contra ataques como CSRF (Cross-Site Request Forgery) e XSS (Cross-Site Scripting), e a gestão eficaz de sessões de utilizador.

Introdução ao Spring Security

O Spring Security é um framework do ecossistema Spring projetado para lidar com autenticação e autorização em aplicações Java. Oferece funcionalidades como:

- **Autenticação e Autorização:** Verificação da identidade dos utilizadores e definição de permissões de acesso a recursos.
- **Proteção Contra Ameaças Comuns:** Defesas integradas contra ataques CSRF, XSS e outros.
- **Integração com Diversos Mecanismos de Autenticação:** Suporte a autenticação em memória, bases de dados, LDAP, entre outros.

Configuração Inicial do Spring Security

Para integrar o Spring Security num projeto Spring Boot, siga os passos abaixo:

Adicionar a Dependência no `pom.xml`:

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

Configurar a Segurança da Aplicação:

Crie uma classe de configuração que estenda `WebSecurityConfigurerAdapter` e anote-a com `@EnableWebSecurity`.

```
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSec
urity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityC
onfigurerAdapter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .permitAll()
            .and()
            .Logout()
            .permitAll();
    }
}
```

Esta configuração básica assegura que todas as requisições exigem autenticação, com um formulário de login padrão fornecido pelo Spring Security.

Implementação de Autenticação Simples

Para definir utilizadores e roles em memória:

A autenticação em memória, como a apresentada neste exemplo, é uma solução simples e prática para ambientes de desenvolvimento ou aplicações pequenas. Entre as vantagens estão a facilidade de implementação e configuração rápida, sem a necessidade de bases de dados ou serviços externos. Contudo, apresenta desvantagens significativas: as credenciais são armazenadas diretamente no código e, muitas vezes, sem encriptação adequada, tornando esta abordagem inadequada para ambientes de produção. Além disso, não é escalável, dificultando a gestão de um número elevado de utilizadores ou integrações mais complexas. Para sistemas em produção, é essencial optar por mecanismos mais seguros e robustos, como a autenticação baseada em bases de dados ou serviços externos (LDAP, OAuth2, etc.).

Configurar Utilizadores em Memória:

Adicione o seguinte método à classe *SecurityConfig*:

```
import  
org.springframework.security.config.annotation.authentication.builders  
.AuthenticationManagerBuilder;  
import  
org.springframework.security.crypto.password.NoOpPasswordEncoder;  
  
@Override  
protected void configure(AuthenticationManagerBuilder auth) throws  
Exception {  
    auth.inMemoryAuthentication()  
        .passwordEncoder(NoOpPasswordEncoder.getInstance())  
        .withUser("user").password("password").roles("USER")  
        .and()  
        .withUser("admin").password("admin").roles("ADMIN");  
}
```

Nota: O uso de *NoOpPasswordEncoder* não é recomendado em ambientes de produção, pois as senhas são armazenadas em texto simples. Utilize encoders como *BCryptPasswordEncoder* para maior segurança.

Definir Permissões de Acesso

Altere o método `configure(HttpSecurity http)` para restringir o acesso com base nas roles:

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .authorizeRequests()  
            .antMatchers("/admin/**").hasRole("ADMIN")  
            .antMatchers("/user/**").hasRole("USER")  
            .anyRequest().authenticated()  
            .and()  
        .formLogin()  
            .permitAll()  
            .and()  
        .logout()  
            .permitAll();  
}
```

Com esta configuração, endpoints que começam com `/admin/` exigem que o utilizador tenha a role "ADMIN", enquanto aqueles que começam com `/user/` requerem a role "USER".

Esta implementação básica permite proteger a aplicação com autenticação em memória, adequada para ambientes de desenvolvimento ou aplicações simples. Para aplicações em produção, recomenda-se a integração com um sistema de gestão de utilizadores mais robusto, como uma base de dados ou um serviço de diretório LDAP.

Segurança Baseada em LDAP

Para implementar um sistema de gestão de utilizadores robusto baseado num serviço de diretório LDAP (Lightweight Directory Access Protocol) utilizando Spring Boot, é necessário configurar a aplicação para autenticar e autorizar utilizadores através do LDAP.

Passos para Implementar a Autenticação LDAP com Spring Boot:

Adicionar Dependências Necessárias:

No ficheiro `pom.xml` do seu projeto Maven, inclua as seguintes dependências:

```
<dependencies>
    <!-- Dependência principal do Spring Boot -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <!-- Suporte para LDAP no Spring -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-ldap</artifactId>
    </dependency>
    <!-- Biblioteca para integração do Spring Security com LDAP -->
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-ldap</artifactId>
    </dependency>
    <!-- Implementação do servidor LDAP embutido para testes -->
    <dependency>
        <groupId>com.unboundid</groupId>
        <artifactId>unboundid-ldapsdk</artifactId>
    </dependency>
</dependencies>
```

Configurar o Servidor LDAP:

Para fins de desenvolvimento e testes, pode utilizar um servidor LDAP embutido. Crie um ficheiro *application.properties* com as seguintes configurações:

```
# Configurações do servidor LDAP embutido
spring.ldap.embedded.base-dn=dc=example,dc=com
spring.ldap.embedded.ldif=classpath:users.ldif
spring.ldap.embedded.port=8389
```

Crie um ficheiro *users.ldif* no diretório *src/main/resources* para definir os utilizadores e grupos:

```
dn: dc=example,dc=com
objectclass: top
objectclass: domain
dc: example

dn: ou=groups,dc=example,dc=com
objectclass: top
objectclass: organizationalUnit
ou: groups

dn: ou=users,dc=example,dc=com
objectclass: top
objectclass: organizationalUnit
ou: users

dn: uid=user,ou=users,dc=example,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: User
sn: User
uid: user
userPassword: password

dn: uid=admin,ou=users,dc=example,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Admin
sn: Admin
uid: admin
userPassword: admin
```

Nota: As senhas no ficheiro LDIF estão em texto simples para simplicidade. Em ambientes de produção, utilize senhas encriptadas.

Configurar a Segurança da Aplicação:

Crie uma classe de configuração de segurança para definir como a aplicação irá autenticar os utilizadores através do LDAP (como os imports são colocados automaticamente pelo IDE, e para caberem numa linha, optou-se por usar um tamanho de letra pequeno):

```
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .permitAll()
            .and()
            .logout()
            .permitAll();
    }

    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .ldapAuthentication()
            .userDnPatterns("uid={0},ou=users")
            .groupSearchBase("ou=groups")
            .contextSource()
            .url("Ldap://localhost:8389/dc=example,dc=com")
            .and()
            .passwordCompare()
            .passwordEncoder(new BCryptPasswordEncoder())
            .passwordAttribute("userPassword");
    }
}
```

Nesta configuração, a aplicação está configurada para autenticar utilizadores contra o servidor LDAP embutido, procurando utilizadores no padrão *uid={0},ou=users* e comparando as senhas utilizando *BCryptPasswordEncoder*.

Executar e Testar a Aplicação:

Com as configurações acima, ao iniciar a aplicação, o servidor LDAP embutido será inicializado com os utilizadores definidos no ficheiro *users.ldif*. Ao aceder à aplicação, será apresentada uma página de login onde poderá autenticar-se com as credenciais definidas (por exemplo, *user/password* ou *admin/admin*).

Considerações Importantes:

- **Segurança das Senhas:** Em ambientes de produção, assegure-se de que as senhas armazenadas no LDAP estão encriptadas e que a aplicação utiliza um codificador de senhas adequado, como *BCryptPasswordEncoder*.
- **Configurações de Produção:** Para ambientes de produção, utilize um servidor LDAP robusto e configure conexões seguras (por exemplo, LDAPS) para proteger a comunicação entre a aplicação e o servidor LDAP.
- **Gestão de Grupos e Roles:** Configure adequadamente os grupos no LDAP para refletir as roles e permissões necessárias na aplicação, permitindo uma gestão centralizada de acessos.

Para mais detalhes e exemplos sobre a integração do Spring Security com LDAP, consulte a documentação oficial do Spring:

Vaadin Security e sua relação com o Spring Security

Ao desenvolver uma aplicação com **Vaadin** e **Spring Boot**, é recomendável utilizar os auxiliares de segurança integrados do Vaadin em conjunto com o **Spring Security**. Essa abordagem oferece uma integração mais harmoniosa e eficiente entre a interface do utilizador e os mecanismos de segurança.

Vantagens de Utilizar os Auxiliares de Segurança do Vaadin:

- **Integração Simplificada:** Os auxiliares de segurança do Vaadin permitem configurar o controlo de acesso baseado em vistas com uma configuração mínima do Spring Security. Isso facilita a definição de permissões diretamente nas classes de vista, utilizando anotações como `@AnonymousAllowed`, `@PermitAll` e `@RolesAllowed`.
- **Desenvolvimento Mais Rápido e Seguro:** Ao aproveitar os auxiliares do Vaadin, pode implementar rapidamente mecanismos de segurança robustos, reduzindo a complexidade e o potencial de erros na configuração manual do Spring Security.

Compatibilidade com Recursos de Segurança Avançados:

Embora o Vaadin forneça auxiliares de segurança que simplificam a integração com o Spring Security, ele é totalmente compatível com soluções de segurança avançadas no ecossistema Java, incluindo:

- **LDAP (Lightweight Directory Access Protocol):** Pode configurar a sua aplicação para autenticar utilizadores através de um serviço LDAP, aproveitando as capacidades do Spring Security para integração com diretórios corporativos.
- **Outros Mecanismos de Autenticação:** Além do LDAP, o Vaadin, em conjunto com o Spring Security, permite a implementação de diversos mecanismos de autenticação, como OAuth2, JWT (JSON Web Token) e autenticação baseada em bases de dados, oferecendo flexibilidade para atender às necessidades específicas da sua aplicação.

Em resumo, ao utilizar o Vaadin com Spring Boot, é preferível aproveitar os auxiliares de segurança do Vaadin em conjunto com o Spring Security. Essa abordagem proporciona uma integração mais fluida, simplifica a configuração e mantém a compatibilidade com recursos de segurança avançados, garantindo uma aplicação segura e eficiente.

Capítulo 15. Interface Web com Vaadin Flow

Nota: Este capítulo é acompanhado por um **vídeo** com explicações passo-a-passo (ver aqui).

Introdução ao Vaadin

O **Vaadin Flow** é uma das abordagens disponibilizadas pelo Vaadin para o desenvolvimento de aplicações web modernas e responsivas. Esta framework destaca-se por permitir que toda a interface de utilizador (UI) seja construída em Java, eliminando a necessidade de escrever diretamente código HTML ou JavaScript. No entanto, o Vaadin, como um todo, oferece outras alternativas, como o Vaadin Hilla, que combina TypeScript no cliente com Java no servidor para quem prefere separar a lógica da UI e da aplicação.

Tradicionalmente, as UIs de aplicações web podem ser criadas utilizando frameworks baseados em HTML, CSS e JavaScript (como React, Angular ou Vue), ou em frameworks server-side mais tradicionais (como JSP ou Thymeleaf). O Vaadin Flow propõe um caminho diferente, abstraindo a complexidade da integração entre cliente e servidor, permitindo que os programadores se concentrem exclusivamente na lógica de negócios e na interação com componentes, enquanto o framework trata da comunicação cliente-servidor e da renderização no navegador.

Um dos principais aspectos diferenciadores do Vaadin Flow é a sua integração perfeita com o Spring Boot, tirando proveito do ecossistema Spring e proporcionando uma experiência de desenvolvimento mais fluida. Além disso, o Vaadin inclui uma biblioteca robusta de componentes de UI prontos a usar, como botões, tabelas, formulários e gráficos, altamente personalizáveis e ideais para criar interfaces ricas e responsivas.

Neste capítulo, exploraremos como o Vaadin Flow pode ser utilizado para criar interfaces de utilizador de forma eficiente e elegante. Abordaremos os conceitos fundamentais da framework, como configurar um projeto, definir vistas, gerir eventos e implementar funcionalidades como navegação e estilização, demonstrando como esta tecnologia permite o desenvolvimento rápido e intuitivo de aplicações web robustas e modernas.

O que é possível fazer com o Vaadin Flow

Para criar uma interface de utilizador (UI) numa aplicação Vaadin Flow com Spring Boot, pode utilizar componentes Java fornecidos pelo Vaadin para construir a estrutura e funcionalidade da sua aplicação. O Vaadin Flow permite desenvolver aplicações web modernas inteiramente em Java, sem a necessidade de escrever HTML ou JavaScript diretamente.

Passos para Criar uma UI com Vaadin Flow:

Definir uma Vista Principal: Crie uma classe Java que servirá como a vista principal da sua aplicação. Utilize a anotação `@Route` para mapear esta vista a uma rota específica.

```
import com.vaadin.flow.component.button.Button;
import com.vaadin.flow.component.orderedLayout.VerticalLayout;
import com.vaadin.flow.component.textfield.TextField;
import com.vaadin.flow.router.Route;

@Route("")
public class MainView extends VerticalLayout {

    public MainView() {
        TextField nomeField = new TextField("O seu nome");
        Button saudacaoButton = new Button("Cumprimentar",
            e -> add(new Paragraph("Olá, " + nomeField.getValue() +
        "!")));
        add(nomeField, saudacaoButton);
    }
}
```

Neste exemplo, ao aceder à raiz da aplicação ("/"), o utilizador verá um campo de texto para inserir o nome e um botão que, ao ser clicado, exibe uma saudação personalizada.

Adicionar Componentes e Layouts: Utilize componentes como `TextField`, `Button` e layouts como `VerticalLayout` para organizar a interface. Os componentes do Vaadin são altamente personalizáveis e permitem a criação de UIs interativas de forma declarativa em Java.

Gerir Eventos: Adicione *listeners* de eventos aos componentes para definir comportamentos interativos. Por exemplo, o botão no exemplo acima possui um ouvinte que adiciona um parágrafo com a saudação ao layout quando clicado.

Navegação entre Vistas: Para criar uma aplicação com múltiplas vistas, defina várias classes anotadas com `@Route`, cada uma representando uma vista diferente. Utilize o método `UI.getCurrent().navigate("rota")` para navegar entre elas.

```
@Route("outra-vista")
public class OutraVista extends VerticalLayout {
    public OutraVista() {
        add(new Text("Esta é outra vista."));
    }
}
```

Para navegar para "outra-vista":

```
Button navegarButton = new Button("Ir para Outra Vista",
    e -> UI.getCurrent().navigate("outra-vista"));
```

Estilização: Personalize o estilo da sua aplicação utilizando temas e folhas de estilo CSS. O Vaadin permite a integração de estilos para garantir que a UI corresponde à identidade visual desejada.

Ao seguir estes passos e utilizar os recursos fornecidos pelo Vaadin, pode desenvolver interfaces de utilizador modernas e responsivas de forma eficiente, aproveitando o poder do Java no desenvolvimento web.

Configuração de uma Aplicação Vaadin Flow com Spring Boot

O Vaadin Flow é uma framework que permite aos programadores desenvolver aplicações web inteiramente em Java. Em vez de escrever HTML, CSS e JavaScript, a interface do utilizador é construída a partir de componentes de UI em Java, de forma semelhante ao que é feito com Swing e JavaFX. O código HTML, as folhas de estilo CSS e os JavaScripts ainda estão presentes e acessíveis quando necessário, mas são abstraídos por uma API rica em Java.

O Flow atua como um controlo remoto para os elementos HTML no navegador. Estes elementos HTML podem ser simples elementos `<div>` ou elementos mais complexos, como um componente web `<vaadin-grid>`. O Flow controla os atributos, propriedades e filhos desses elementos, podendo até executar invocações JavaScript personalizadas. No lado do servidor, existem objetos Java correspondentes com os quais o programador interage. O Flow gera a sincronização entre os objetos Java do lado do servidor e os elementos HTML do lado do cliente.

Ao construir uma aplicação com o Vaadin Flow, o programador pode concentrar-se na lógica da interface do utilizador em Java, enquanto o Flow trata da comunicação cliente-servidor e da renderização no navegador. Isto permite uma abordagem mais produtiva no desenvolvimento de aplicações web modernas, aproveitando a robustez e familiaridade do ecossistema Java.

Para começar a utilizar o Vaadin Flow com Spring Boot, é recomendável seguir tutoriais que abordem desde a configuração inicial do projeto até à implementação de funcionalidades específicas, garantindo uma compreensão sólida de como integrar o Vaadin Flow em aplicações Spring Boot.

Para iniciar o desenvolvimento de uma aplicação web com o Vaadin Flow e o Spring Boot, siga os seguintes passos:

1. Criação do Projeto

A forma mais simples de iniciar um projeto com Vaadin e Spring Boot é utilizar o [Vaadin Start](#). Esta ferramenta permite configurar e descargar um projeto inicial personalizado. Alternativamente, pode utilizar o [Spring Initializr](#) para configurar um projeto Spring Boot e, em seguida, adicionar as dependências do Vaadin manualmente.

2. Estrutura do Projeto

Um projeto típico com Vaadin e Spring Boot inclui as seguintes partes:

- **Dependências:** As bibliotecas necessárias são geridas no ficheiro `pom.xml` (para projetos Maven) ou `build.gradle` (para projetos Gradle).
- **Classe Principal:** Uma classe anotada com `@SpringBootApplication` que contém o método `main` para iniciar a aplicação.
- **Vistas (views):** Classes Java que representam as diferentes páginas ou componentes da interface do utilizador, anotadas com `@Route`.

3. Adicionar Dependências

Para integrar o Vaadin com o Spring Boot, adicione as seguintes dependências ao seu ficheiro `pom.xml`:

(por favor, ver página seguinte, para uma visão global do ficheiro numa única página)

```

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-bom</artifactId>
            <version>${vaadin.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>com.vaadin</groupId>
        <artifactId>vaadin-spring-boot-starter</artifactId>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
        <plugin>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-maven-plugin</artifactId>
            <version>${vaadin.version}</version>
            <executions>
                <execution>
                    <goals>
                        <goal>prepare-frontend</goal>
                        <goal>build-frontend</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>

```

Certifique-se de substituir \${vaadin.version} pela versão específica do Vaadin que pretende utilizar.

4. Definir a Classe Principal

Crie uma classe principal para a sua aplicação:

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

A anotação `@SpringBootApplication` ativa as configurações automáticas do Spring Boot, incluindo a configuração do Vaadin.

5. Criar Vistas com Vaadin

No Vaadin, as vistas são definidas como classes Java anotadas com `@Route`. Por exemplo:

```
@Route("")  
public class MainView extends VerticalLayout {  
    public MainView() {  
        add(new Text("Bem-vindo ao MainView."));  
    }  
}
```

Esta classe define uma vista que será apresentada na raiz da aplicação.

6. Executar a Aplicação

Para iniciar a aplicação, execute o método `main` da classe principal. O Spring Boot iniciará um servidor web incorporado e a aplicação estará disponível no navegador.

7. Implementar Funcionalidades Adicionais

À medida que desenvolve a aplicação, pode adicionar mais vistas, componentes e lógica de negócios conforme necessário. O Vaadin oferece uma ampla gama de componentes de interface do utilizador que podem ser utilizados para criar interfaces ricas e interativas.

Seguindo estes passos, os alunos poderão criar e desenvolver aplicações web utilizando o Vaadin com o Spring Boot, aproveitando as vantagens de ambos os frameworks para construir interfaces de utilizador modernas e responsivas.

Para criar uma interface de utilizador (UI) com o Vaadin Flow, é fundamental compreender os componentes e layouts que o framework oferece. O Vaadin Flow permite construir UIs de forma declarativa em Java, utilizando uma vasta gama de componentes prontos para uso.

Componentes Básicos

O Vaadin fornece mais de 40 componentes de UI que facilitam a construção de aplicações. Estes componentes são elementos HTML personalizados, baseados nos padrões de Web Components do W3C, e podem ser utilizados diretamente em Java. Por exemplo, para criar um botão:

```
Button button = new Button("Clique-me!");
```

Este código cria um botão com o rótulo "Clique-me!".

Layouts

Os layouts determinam como os componentes são posicionados na janela do navegador. Os layouts mais comuns são *HorizontalLayout*, *VerticalLayout* e *Div*. Por exemplo, para adicionar componentes a um layout horizontal:

```
HorizontalLayout layout = new HorizontalLayout();
layout.add(component1, component2);
```

Este código adiciona *component1* e *component2* ao layout horizontal, posicionando-os lado a lado.

Eventos de UI

Para adicionar funcionalidade à aplicação, é possível associar ouvintes de eventos aos componentes. Por exemplo, para adicionar um ouvinte de clique a um botão:

```
button.addClickListener(event -> {
    // Ação a ser executada quando o botão é clicado
});
```

Este código define uma ação a ser executada quando o botão é clicado.

Exemplo Prático: Lista de Contactos

Vamos criar uma vista que exibe uma lista de contactos. Utilizaremos um *Grid* para mostrar os contactos e um *TextField* para filtrar a lista.

(por favor, ver página seguinte, para uma visão global do ficheiro numa única página)

```

@Route("")
@PageTitle("Contactos | Vaadin CRM")
public class ListView extends VerticalLayout {
    Grid<Contact> grid = new Grid<>(Contact.class);
    TextField filterText = new TextField();

    public ListView() {
        addClassName("list-view");
        setSizeFull();
        configureGrid();
        add(getToolbar(), grid);
    }

    private void configureGrid() {
        grid.addclassNames("contact-grid");
        grid.setSizeFull();
        grid.setColumns("firstName", "lastName", "email");
        grid.addColumn(contact ->
contact.getStatus().getName()).setHeader("Status");
        grid.addColumn(contact ->
contact.getCompany().getName()).setHeader("Empresa");
        grid.getColumns().forEach(col -> col.setAutoWidth(true));
    }

    private HorizontalLayout getToolbar() {
        filterText.setPlaceholder("Filtrar por nome... ");
        filterText.setClearButtonVisible(true);
        filterText.setValueChangeMode(ValueChangeMode.LAZY);
        Button addContactButton = new Button("Adicionar contacto");
        HorizontalLayout toolbar = new HorizontalLayout(filterText,
addContactButton);
        toolbar.addClassNames("toolbar");
        return toolbar;
    }
}

```

Este exemplo cria uma vista principal que exibe uma grelha de contactos e uma barra de ferramentas com um campo de filtro e um botão para adicionar novos contactos.

Navegação e Layout Principal

Aplicações geralmente possuem uma vista principal com um menu que permite a navegação entre diferentes sub-vistas. O *AppLayout* do Vaadin facilita a criação desse layout principal, incluindo cabeçalho, menu e área de conteúdo. Por exemplo:

```
public class MainView extends AppLayout {  
    private final Tabs menu;  
    private H1 viewTitle;  
  
    public MainView() {  
        setPrimarySection(Section.DRAWER);  
        addToNavbar(true, createHeaderContent());  
        menu = createMenu();  
        addToDrawer(createDrawerContent(menu));  
    }  
  
    private Component createHeaderContent() {  
        HorizontalLayout layout = new HorizontalLayout();  
        layout.setId("header");  
        layout.setWidthFull();  
        layout.setSpacing(false);  
        layout.setAlignItems(FlexComponent.Alignment.CENTER);  
        layout.add(new DrawerToggle());  
        viewTitle = new H1();  
        layout.add(viewTitle);  
        layout.add(new Image("images/user.svg", "Avatar"));  
        return layout;  
    }  
  
    private Tabs createMenu() {  
        // Implementação do menu  
    }  
  
    private Component createDrawerContent(Tabs menu) {  
        // Implementação do conteúdo do drawer  
    }  
}
```

Este código cria uma vista principal com um cabeçalho que inclui um botão para alternar o menu (drawer), um título de vista e um avatar de utilizador.

Seguindo estas diretrizes, irá poder construir interfaces de utilizador eficazes utilizando o Vaadin Flow com Spring Boot, aproveitando os componentes e layouts fornecidos pela framework para criar aplicações web modernas e responsivas.

Implementação de Manipuladores de Eventos

Para lidar com a interação do utilizador, o Vaadin Flow utiliza programação orientada a eventos. Quando um utilizador interage com a interface no navegador, são gerados eventos que o Flow encaminha para o lado do servidor, permitindo que o código da aplicação os processe.

Cada tipo de interação do utilizador gera um tipo específico de evento. Por exemplo, clicar num botão gera um evento `ClickEvent<Button>`. Para processar estes eventos, é necessário implementar um ouvinte que trate o evento específico. Os ouvintes de eventos podem ser implementados de várias formas:

- **Classes Locais:** Implementando a interface `ComponentEventListener` para o tipo de evento.

```
Button button = new Button("Clique-me!");
class MeuOuvinteDeClique implements
ComponentEventListener<ClickEvent<Button>> {
    int contador = 0;
    @Override
    public void onComponentEvent(ClickEvent<Button> evento) {
        evento.getSource().setText("Clicou " + (++contador) + " vezes");
    }
}
button.addClickListener(new MeuOuvinteDeClique());
add(button);
```

- **Expressões Lambda:** Uma forma concisa de implementar ouvintes para interfaces funcionais.

```
Button button = new Button("Clique-me!", evento ->
    evento.getSource().setText("Clicado!"));
add(button);
```

- **Classes Anónimas:** Permitem implementar ouvintes de eventos sem a necessidade de classes nomeadas.

```
Button button = new Button("Clique-me!", new
ComponentEventListener<ClickEvent<Button>>() {
    int contador = 0;
    @Override
    public void onComponentEvent(ClickEvent<Button> evento) {
        evento.getSource().setText("Clicou " + (++contador) + " vezes");
    }
});
add(button);
```

- **Referências a Métodos:** Direccionam eventos para métodos específicos.

```

class BarraDeBotoes extends HorizontalLayout {
    public BarraDeBotoes() {
        add(new Button("OK", this::ok));
        add(new Button("Cancelar", this::cancelar));
    }
    public void ok(ClickEvent<Button> evento) {
        evento.getSource().setText("OK!");
    }
    public void cancelar(ClickEvent<Button> evento) {
        evento.getSource().setText("Cancelado!");
    }
}
add(new BarraDeBotoes());

```

Estas abordagens permitem que os programadores escolham a forma mais adequada para implementar ouvintes de eventos, dependendo do contexto e das necessidades da aplicação.

Eventos Personalizados

Além dos eventos padrão fornecidos pelos componentes do Vaadin, é possível criar eventos personalizados para componentes específicos. Para isso, deve-se estender a classe *ComponentEvent* e utilizar a anotação *@DomEvent* para associar o evento a um evento DOM específico. Por exemplo:

```

@DomEvent("change")
public class EventoDeAlteracao extends ComponentEvent<TextField> {
    public EventoDeAlteracao(TextField source, boolean fromClient) {
        super(source, fromClient);
    }
}

```

Para adicionar um ouvinte para este evento personalizado:

```

textField.addListener(EventoDeAlteracao.class, evento -> {
    // Lógica a ser executada quando o evento ocorre
});

```

Esta abordagem permite a criação de componentes altamente personalizados que podem responder a eventos específicos definidos pelo programador.

Filtragem e Debouncing de Eventos

Em situações onde eventos são disparados com alta frequência, como durante a digitação num campo de texto, é importante controlar a taxa de eventos enviados para o servidor. O Vaadin permite configurar filtros e definir configurações de "debounce" para limitar a frequência de eventos processados. Por exemplo:

```
@DomEvent(value = "input",
           debounce = @DebounceSettings(timeout = 500,
                                         phases = DebouncePhase.TRAILING))
public class EventoDeInput extends ComponentEvent<TextField> {
    private String valor;
    public EventoDeInput(TextField source, boolean fromClient,
                         @EventData("element.value") String valor) {
        super(source, fromClient);
        this.valor = valor;
    }
    public String getValor() {
        return valor;
    }
}
```

Neste exemplo, o evento *EventoDeInput* é enviado para o servidor 500 milissegundos após o último input do utilizador, evitando sobrecarga no processamento de eventos.

Ao compreender e aplicar estes conceitos, poderá desenvolver aplicações web interativas e responsivas utilizando o Vaadin Flow com Spring Boot, aproveitando os recursos de manipulação de eventos fornecidos pelo framework.

Para criar uma vista principal numa aplicação Vaadin Flow com Spring Boot, é comum utilizar o componente *AppLayout*. Este componente permite estruturar a aplicação com um cabeçalho, um menu de navegação e uma área de conteúdo, facilitando a navegação entre diferentes sub-vistas.

Passos para Implementar a Vista Principal:

Definir a Classe da Vista Principal: Crie uma classe que estende *AppLayout*. Esta classe servirá como a estrutura principal da aplicação.

```
@Route("")
public class MainView extends AppLayout {
    private final Tabs menu;
    private H1 viewTitle;

    public MainView() {
        setPrimarySection(Section.DRAWER);
        addToNavbar(true, createHeaderContent());
        menu = createMenu();
        addToDrawer(createDrawerContent(menu));
    }
}
```

Neste exemplo, a anotação `@Route("")` define que esta é a vista padrão da aplicação, acessível na raiz do URL.

Criar o Conteúdo do Cabeçalho: Implemente um método para criar o conteúdo do cabeçalho, incluindo elementos como o título da vista e um botão para alternar a visibilidade do menu lateral.

```
private Component createHeaderContent() {
    HorizontalLayout Layout = new HorizontalLayout();
    Layout.setId("header");
    Layout.setWidthFull();
    Layout.setSpacing(false);
    Layout.setAlignItems(FlexComponent.Alignment.CENTER);
    Layout.add(new DrawerToggle());
    viewTitle = new H1();
    Layout.add(viewTitle);
    Layout.add(new Image("images/user.svg", "Avatar"));
    return Layout;
}
```

Este método cria um layout horizontal para o cabeçalho, incluindo um botão de alternância (*DrawerToggle*), um título (*H1*) e uma imagem de avatar.

Configurar o Menu de Navegação: Implemente um método para criar o menu de navegação, utilizando componentes como *Tabs* e *Tab* para representar as diferentes vistas da aplicação.

```
private Tabs createMenu() {
    final Tabs tabs = new Tabs();
    tabs.setOrientation(Tabs.Orientation.VERTICAL);
    tabs.add(createMenuItem());
    return tabs;
}

private Component[] createMenuItem() {
    return new Tab[]{
        createTab("Início", HomeView.class),
        createTab("Sobre", AboutView.class),
        createTab("Contacto", ContactView.class)
    };
}

private Tab createTab(String text, Class<? extends Component>
navigationTarget) {
    final Tab tab = new Tab();
    RouterLink Link = new RouterLink(text, navigationTarget);
    Link.setTabIndex(-1);
    tab.add(Link);
    return tab;
}
```

Este conjunto de métodos cria um menu vertical com itens que permitem a navegação para diferentes vistas, como "Início", "Sobre" e "Contacto".

Definir o Conteúdo do Menu Lateral (Drawer): Implemente um método para criar o conteúdo do menu lateral, que incluirá o menu de navegação criado anteriormente.

```
private Component createDrawerContent(Tabs menu) {
    VerticalLayout Layout = new VerticalLayout();
    Layout.setSizeFull();
    Layout.add(menu);
    return Layout;
}
```

Este método adiciona o menu de navegação a um layout vertical que ocupa toda a altura disponível no menu lateral.

Atualizar o Título da Vista: Implemente um método para atualizar o título da vista com base na navegação atual.

```
private void updateViewTitle(String title) {  
    viewTitle.setText(title);  
}
```

Este método permite que o título exibido no cabeçalho seja atualizado conforme o utilizador navega entre diferentes vistas.

Segundo estes passos, é possível criar uma estrutura de aplicação organizada, com um cabeçalho, menu de navegação e área de conteúdo, proporcionando uma experiência de utilizador intuitiva e eficiente.

Para implementar a navegação e o roteamento numa aplicação Vaadin Flow com Spring Boot, é essencial compreender como mapear vistas para URLs específicos e gerir a navegação entre elas. O Vaadin Flow facilita este processo através de anotações e componentes dedicados.

Definição de Rotas com `@Route`

No Vaadin, cada vista é uma classe que representa uma parte lógica da interface do utilizador. Para mapear uma vista a um URL específico, utiliza-se a anotação `@Route`. Por exemplo:

```
@Route("home")
public class HomeView extends VerticalLayout {
    public HomeView() {
        add(new Text("Bem-vindo à HomeView!"));
    }
}
```

Neste exemplo, ao navegar para `http://localhost:8080/home`, a aplicação exibirá o conteúdo definido em `HomeView`. Se a anotação `@Route` for utilizada sem parâmetros, a vista será associada à raiz do URL.

Navegação entre Vistas

A navegação entre vistas pode ser realizada de várias formas:

- **Utilizando o Componente `RouterLink`:** Este componente cria um link que aponta para uma rota específica.

```
RouterLink Link = new RouterLink("Ir para Home", HomeView.class);
add(Link);
```

Ao clicar no link, o utilizador será direcionado para a vista `HomeView` sem recarregar a página, e o URL no navegador será atualizado.

- **Através do Método `UI.navigate`:** Este método permite navegar programaticamente para uma rota específica.

```
Button button = new Button("Ir para Home", event ->
    UI.getCurrent().navigate("home"));
add(button);
```

Ao clicar no botão, a aplicação navegará para a rota "home".

Parâmetros de Rota

Para passar parâmetros através da URL, pode-se definir rotas com parâmetros. Por exemplo:

```
@Route("user/:userID")
public class UserView extends VerticalLayout implements BeforeEnterObserver {
    @Override
    public void beforeEnter(BeforeEnterEvent event) {
        String userId =
event.getRouteParameters().get("userID").orElse("Desconhecido");
        add(new Text("ID do Utilizador: " + userId));
    }
}
```

Ao navegar para `http://localhost:8080/user/123`, a vista exibirá "ID do Utilizador: 123".

Layouts Aninhados e Vistas Principais

Para criar uma estrutura de navegação mais complexa, como uma aplicação com um menu e várias sub-vistas, pode-se utilizar layouts aninhados. Por exemplo:

```
@Route(value = "main", Layout = MainLayout.class)
public class MainView extends VerticalLayout {
    public MainView() {
        add(new Text("Esta é a vista principal."));
    }
}
```

Neste caso, `MainLayout` é uma classe que define o layout comum (como um menu ou cabeçalho), e `MainView` é uma sub-vista que será renderizada dentro desse layout.

Ciclo de Vida da Navegação

O Vaadin fornece interfaces como `BeforeEnterObserver` e `AfterNavigationObserver` que permitem executar lógica antes ou depois da navegação para uma vista. Por exemplo:

```
@Route("secure")
public class SecureView extends VerticalLayout implements BeforeEnterObserver {
    @Override
    public void beforeEnter(BeforeEnterEvent event) {
        if (!isUserAuthenticated()) {
            event.forwardTo("Login");
        }
    }
}
```

Este exemplo verifica se o utilizador está autenticado antes de permitir o acesso à vista "secure". Se não estiver, redireciona-o para a página de login.

Gestão de Recursos Estáticos em Vaadin

Para gerir recursos estáticos numa aplicação Vaadin Flow com Spring Boot, é fundamental compreender como armazenar e aceder a ficheiros como imagens, folhas de estilo CSS e scripts JavaScript. A estrutura do projeto e a localização dos recursos variam ligeiramente entre projetos Spring e não Spring.

Localização dos Recursos em Projetos Spring Boot

Em projetos que utilizam Spring Boot, os recursos estáticos devem ser colocados em diretórios específicos para serem servidos corretamente pela aplicação. A tabela abaixo resume onde colocar diferentes tipos de recursos:

Tipo de Ficheiro	Localização do Ficheiro no Projeto
Ficheiros CSS	<code>/src/main/frontend/my-styles/</code>
JavaScript, TypeScript e Lit templates	<code>/src/main/frontend/src/</code>
Ficheiros estáticos (imagens, ícones, documentos)	<code>/src/main/resources/META-INF/resources/</code>

Por exemplo, para adicionar uma imagem chamada `fLower.jpg`, deve colocá-la em:

`/src/main/resources/META-INF/resources/img/fLower.jpg`

Desta forma, a imagem estará acessível na aplicação através do caminho `/img/fLower.jpg`.

Importação de Recursos no Código

Para utilizar estes recursos na aplicação, é necessário importá-los ou referenciá-los adequadamente no código Java:

- **Ficheiros CSS:** Utilize a anotação `@CssImport` para importar folhas de estilo.
`@CssImport("./my-styles/styles.css")`
- **Ficheiros JavaScript ou TypeScript:** Utilize a anotação `@JsModule` para importar scripts.
`@JsModule("./src/my-script.js")`
- **Ficheiros Estáticos (Imagens, Documentos):** Utilize componentes como `Image` ou `Anchor` para referenciar recursos estáticos.
`Image image = new Image("img/fLower.jpg", "A fLower");`

Ao seguir estas diretrizes, os recursos serão corretamente servidos e integrados na aplicação Vaadin com Spring Boot.

Carregamento Programático de Recursos

Para carregar recursos de forma programática, como imagens ou documentos, pode-se utilizar a classe `StreamResource` em conjunto com um `InputStreamFactory`. Isto é útil quando o conteúdo é dinâmico ou gerado em tempo real.

Este método permite carregar recursos diretamente do classpath da aplicação:

```
StreamResource resource = new StreamResource("flower.jpg", () ->
    getClass().getResourceAsStream("/META-INF/resources/img/flower.jpg"));
Image image = new Image(resource, "A flower");
```

Ao estruturar corretamente os recursos estáticos e referenciá-los adequadamente no código, garante-se que a aplicação Vaadin Flow com Spring Boot serve e utiliza esses recursos de forma eficiente. Para mais detalhes e exemplos, consulte a [documentação oficial do Vaadin](#).

Atualização de Parâmetros de URL sem Navegação

Para manter a integridade dos links diretos (deep linking) numa aplicação Vaadin Flow com Spring Boot, é por vezes necessário atualizar os parâmetros da URL sem efetuar uma navegação completa. Isto é particularmente útil em cenários onde o estado da aplicação deve refletir-se na URL, permitindo que os utilizadores partilhem ou guardem links que reproduzam o estado atual da interface.

O Vaadin Flow permite a atualização dos parâmetros da URL sem recarregar a página, utilizando a API de Histórico do navegador. Ao manipular o estado do histórico, é possível modificar a URL apresentada ao utilizador sem desencadear uma navegação ou recarregamento da página.

(por favor, ver página seguinte, para uma visão global do ficheiro numa única página)

Exemplo Prático:

Considere uma vista que edita informações de um utilizador. Ao selecionar um utilizador para edição, pretende-se que a URL reflita essa seleção, permitindo que o link seja partilhado ou guardado para acesso direto posterior.

```
@Route("user")
public class UserEditor extends VerticalLayout implements
HasUrlParameter<Integer> {

    public void editUser(User user) {
        // Lógica para editar o utilizador
        createFormForUser(user);
        // Atualizar os parâmetros da URL
        updateQueryParameters(user);
    }

    private void updateQueryParameters(User user) {
        String deepLinkingUrl = RouteConfiguration.forSessionScope()
            .getUrl(UserEditor.class, user.getId());
        // Atualiza a URL no navegador sem recarregar a página
        getUI().ifPresent(ui -> ui.getPage().getHistory()
            .replaceState(null, deepLinkingUrl));
    }

    @Override
    public void setParameter(BeforeEvent event, @OptionalParameter Integer
id) {
        if (id != null) {
            // Lógica para carregar o utilizador com o ID fornecido
            createFormForUser(new User("Utilizador " + id, id));
        }
    }

    private void createFormForUser(User user) {
        // Implementação do formulário de edição do utilizador
        add(new Paragraph("Utilizador: " + user.getId()));
    }
}
```

Neste exemplo, ao chamar o método `editUser`, a aplicação atualiza a interface para refletir o utilizador selecionado e modifica a URL no navegador para incluir o ID do utilizador. A utilização de `getPage().getHistory().replaceState(null, deepLinkingUrl)` permite alterar a URL sem recarregar a página, mantendo a experiência do utilizador fluida.

Considerações Importantes:

- **Manutenção do Estado da Aplicação:** Ao atualizar a URL para refletir o estado atual, garante-se que os utilizadores podem partilhar ou guardar links que reproduzem exatamente o mesmo estado da interface quando revisitados.
- **Utilização da API de Histórico:** A manipulação do histórico do navegador através de `replaceState` ou `pushState` permite modificar a URL sem desencadear uma navegação completa, preservando o estado atual da aplicação.
- **Implementação de `HasUrlParameter`:** Ao implementar a interface `HasUrlParameter`, a vista pode processar parâmetros da URL quando o utilizador acede diretamente através de um link, assegurando que o estado correto é restaurado.

Ao aplicar estas práticas, a aplicação torna-se mais robusta e amigável, proporcionando uma experiência de utilizador consistente e permitindo a partilha eficaz de links diretos para estados específicos da aplicação.

No desenvolvimento de aplicações web com Vaadin Flow e Spring Boot, é crucial compreender o ciclo de vida da navegação para garantir uma experiência de utilizador fluida e segura. O Vaadin Flow oferece eventos específicos que permitem interagir com o processo de navegação, possibilitando a execução de ações antes e depois da transição entre vistas. Iremos abordar este tema na secção seguinte.

Eventos do Ciclo de Vida da Navegação

BeforeLeaveEvent: Este evento é disparado antes de a navegação sair da vista atual. Implementando a interface *BeforeLeaveObserver*, é possível controlar ou impedir a saída da vista, útil para verificar alterações não guardadas ou confirmar ações do utilizador.

```
@Route("edit")
public class EditView extends VerticalLayout implements
BeforeLeaveObserver {
    @Override
    public void beforeLeave(BeforeLeaveEvent event) {
        if (hasUnsavedChanges()) {
            // Lógica para notificar o utilizador ou cancelar a
            // navegação
            event.postpone();
        }
    }
}
```

BeforeEnterEvent: Disparado antes de entrar numa nova vista, permite verificar permissões, redirecionar ou modificar o destino da navegação. Implementando a interface *BeforeEnterObserver*, pode-se controlar o acesso à vista.

```
@Route("admin")
public class AdminView extends VerticalLayout implements
BeforeEnterObserver {
    @Override
    public void beforeEnter(BeforeEnterEvent event) {
        if (!isUserAuthorized()) {
            event.forwardTo("access-denied");
        }
    }
}
```

AfterNavigationEvent: Este evento ocorre após a conclusão da navegação, sendo ideal para atualizar componentes da interface ou executar ações que dependem da vista atual. Implementando a interface *AfterNavigationObserver*, é possível reagir a mudanças na navegação.

```
@Route("dashboard")
public class DashboardView extends VerticalLayout implements
AfterNavigationObserver {
    @Override
    public void afterNavigation(AfterNavigationEvent event) {
        // Atualizar dados ou componentes após a navegação
        refreshData();
    }
}
```

Redirecionamento e Encaminhamento durante a Navegação

Durante os eventos de navegação, é possível alterar o destino utilizando métodos como `rerouteTo()` e `forwardTo()`.

- **`rerouteTo()`**: Altera o destino da navegação sem modificar a URL no navegador. Útil para redirecionamentos internos onde a URL visível deve permanecer inalterada.

```
@Override  
public void beforeEnter(BeforeEnterEvent event) {  
    if (someCondition()) {  
        event.rerouteTo("alternative-view");  
    }  
}
```

- **`forwardTo()`**: Encaminha a navegação para uma nova vista e atualiza a URL no navegador. Adequado quando é necessário refletir a nova vista na URL.

```
@Override  
public void beforeEnter(BeforeEnterEvent event) {  
    if (anotherCondition()) {  
        event.forwardTo("new-view");  
    }  
}
```

Compreender e utilizar estes eventos permite um controlo refinado sobre o fluxo de navegação na aplicação, assegurando que os utilizadores têm acesso adequado às vistas e que a interface responde de forma previsível às suas ações.

Para mais detalhes, consulte a [documentação oficial do Vaadin sobre o ciclo de vida da navegação](#).

Tratamento de Exceções de Navegação

No desenvolvimento de aplicações web com Vaadin Flow e Spring Boot, é essencial implementar um tratamento eficaz de exceções de roteamento para garantir uma experiência de utilizador robusta e informativa. O Vaadin Flow oferece mecanismos que permitem lidar com exceções não tratadas durante a navegação, apresentando vistas de erro apropriadas aos utilizadores.

Quando ocorre uma exceção não tratada durante a navegação, o Vaadin Flow procura uma classe que implemente a interface `HasErrorParameter<T extends Exception>` correspondente ao tipo de exceção lançada. Estas classes atuam como alvos de exceção, funcionando de maneira semelhante aos alvos de navegação regulares, mas geralmente não possuem uma anotação `@Route` específica, pois são exibidas para URLs arbitrárias.

Exemplo de Implementação:

Considere uma exceção personalizada `AccessDeniedException` que é lançada quando um utilizador tenta aceder a uma área restrita da aplicação. Para tratar esta exceção e fornecer uma resposta adequada ao utilizador, pode-se criar uma classe que implemente `HasErrorParameter<AccessDeniedException>`:

```
@Tag(Tag.DIV)
@ParentLayout(MainLayout.class)
public class AccessDeniedExceptionHandler extends Component implements
HasErrorParameter<AccessDeniedException> {

    @Override
    public int setErrorParameter(BeforeEnterEvent event,
ErrorParameter<AccessDeniedException> parameter) {
        getElement().setText("Acesso negado: não tem permissões suficientes
para aceder a esta área.");
        return HttpServlet.SC_FORBIDDEN;
    }
}
```

Neste exemplo, quando uma `AccessDeniedException` é lançada durante a navegação, o utilizador é direcionado para uma vista que exibe a mensagem "Acesso negado: não tem permissões suficientes para aceder a esta área." e o código de resposta HTTP 403 (Forbidden) é retornado.

Redirecionamento para Vistas de Erro

Durante os eventos de navegação, como *BeforeEnterEvent* ou *BeforeLeaveEvent*, é possível redirecionar para uma vista de erro registada para uma exceção específica utilizando o método *rerouteToError()*. Por exemplo:

```
public class AuthenticationHandler implements BeforeEnterObserver {  
  
    @Override  
    public void beforeEnter(BeforeEnterEvent event) {  
        Class<?> target = event.getNavigationTarget();  
        if (!currentUserMayEnter(target)) {  
            event.rerouteToError(AccessDeniedException.class);  
        }  
    }  
  
    private boolean currentUserMayEnter(Class<?> target) {  
        // Implementação da Lógica de verificação de permissões  
        return false;  
    }  
}
```

Neste cenário, se o utilizador não tiver permissão para aceder à vista de destino, a navegação é redirecionada para a vista de erro associada à *AccessDeniedException*.

Considerações Importantes:

- **Ordem de Correspondência de Exceções:** As exceções são correspondidas primeiro pela causa e depois pelo supertipo da exceção. O Vaadin Flow implementa por defeito o *RouteNotFoundError* para *NotFoundException* (erro 404) e o *InternalServerError* para *Exception* (erro 500).
- **Personalização de Vistas de Erro:** É possível substituir os manipuladores de exceção padrão estendendo-os e definindo layouts parentais conforme necessário. As vistas de erro podem definir *ParentLayouts* para manter a consistência da interface.
- **Eventos de Navegação:** Os eventos *BeforeNavigationEvent* e *AfterNavigationEvent* são disparados durante a navegação normal e também quando ocorre uma exceção, permitindo a execução de lógica adicional conforme necessário.

Ao implementar um tratamento de exceções de roteamento eficaz, assegura-se que a aplicação responde de forma adequada a erros, fornecendo feedback claro aos utilizadores e mantendo a integridade da experiência de navegação.

Para mais detalhes, consulte a [documentação oficial do Vaadin sobre tratamento de exceções de roteamento](#).

Para obter todas as rotas registadas numa aplicação Vaadin Flow com Spring Boot, pode utilizar a classe *RouteConfiguration*. Esta classe fornece métodos que permitem aceder às rotas disponíveis, facilitando a construção de menus de navegação dinâmicos ou outras funcionalidades que dependam das rotas definidas na aplicação.

Obtenção de Todas as Rotas Registadas

Para recuperar uma lista de todas as rotas registadas, utilize o método *getAvailableRoutes()* da seguinte forma:

```
List<RouteData> rotas =  
    RouteConfiguration.forSessionScope().getAvailableRoutes();
```

O objeto *RouteData* contém informações relevantes sobre cada rota definida, como o template da rota, parâmetros e layouts parentais.

Filtragem de Rotas por Layout Parental

Se pretender obter apenas as rotas associadas a um layout parental específico, pode filtrar a lista obtida:

```
List<RouteData> rotas =  
    RouteConfiguration.forSessionScope().getAvailableRoutes();  
List<RouteData> rotasDoMeuLayout = rotas.stream()  
    .filter(routeData ->  
        MeuLayoutParental.class.equals(routeData.getParentLayout()))  
    .collect(Collectors.toList());
```

Este código filtra as rotas cuja classe de layout parental corresponde a *MeuLayoutParental*.

Considerações Importantes

- **Escopo da Sessão:** O método *forSessionScope()* obtém a configuração de rotas para a sessão atual. Se necessitar de obter rotas num contexto diferente, como a nível de aplicação, utilize o método *forApplicationScope()*.
- **Construção de Menus Dinâmicos:** Ao obter a lista de rotas registadas, pode construir menus de navegação que refletem dinamicamente as vistas disponíveis na aplicação, melhorando a experiência do utilizador.

Para mais detalhes, consulte a [documentação oficial do Vaadin sobre obtenção de rotas registadas](#).

Utilização da Anotação `@PageTitle`

Para definir o título da página durante a navegação numa aplicação Vaadin Flow com Spring Boot, existem duas abordagens principais: utilizar a anotação `@PageTitle` ou implementar a interface `HasDynamicTitle`. Estas abordagens são mutuamente exclusivas; utilizar ambas na mesma classe resultará numa exceção em tempo de execução.

A forma mais simples de definir o título da página é através da anotação `@PageTitle` na classe do componente:

```
@PageTitle("Página Inicial")
@Route("inicio")
public class PaginaInicialView extends Div {
    public PaginaInicialView() {
        setText("Bem-vindo à Página Inicial");
    }
}
```

Neste exemplo, ao navegar para a rota "`inicio`", o título da página será definido como "Página Inicial". A anotação `@PageTitle` é lida apenas a partir do alvo de navegação atual; superclasses e vistas parentais não são consideradas.

Definição Dinâmica do Título da Página

Como alternativa, é possível definir o título da página em tempo de execução implementando a interface `HasDynamicTitle`:

```
@Route("artigo")
public class ArtigoView extends Div implements HasDynamicTitle,
HasUrlParameter<Long> {
    private String titulo = "";

    @Override
    public String getPageTitle() {
        return titulo;
    }

    @Override
    public void setParameter(BeforeEvent event, Long parameter) {
        if (parameter != null) {
            titulo = "Artigo #" + parameter;
        }
    }
}
```

```
    } else {
        titulo = "Lista de Artigos";
    }
}
```

Neste caso, o título da página é definido dinamicamente com base no parâmetro da URL. Por exemplo, ao navegar para `/artigo/1`, o título será "Artigo #1".

Considerações Importantes

- **Exclusividade das Abordagens:** Não combine `@PageTitle` e `HasDynamicTitle` na mesma classe, pois isso causará uma exceção em tempo de execução.
- **Herança de Títulos:** A anotação `@PageTitle` não é herdada de superclasses ou layouts parentais; deve ser definida diretamente na classe do componente de navegação.

Ao aplicar estas práticas, assegura-se que o título da página reflete corretamente o conteúdo apresentado, melhorando a experiência do utilizador e a otimização para motores de busca (SEO).

Continua, em construção

Glossário de termos

Acesso a Bases de Dados

Operações realizadas para conectar, consultar, atualizar ou eliminar dados numa base de dados relacional ou não-relacional.

Annotations em Java

Metadados no código que fornecem instruções ou informações adicionais ao compilador ou a frameworks (ex.: `@Override`, `@Service`).

Anotações do Spring (`@Service`, `@Component`, `@Repository`)

Declarações em código que informam ao Spring como gerir as classes no contexto da aplicação.

API (Application Programming Interface)

Um conjunto de regras que permite que diferentes sistemas ou programas comuniquem entre si.

Arquitetura Cloud-Native

Um estilo de desenvolvimento de software que utiliza tecnologias específicas para criar aplicações otimizadas para ambientes de cloud computing.

Arquitetura MVC (Model-View-Controller)

Um modelo que organiza uma aplicação em camadas para separar a lógica de negócio, a interface e o controlo das interações.

Arquivo POM (Project Object Model)

Um ficheiro XML utilizado pelo Maven para descrever as dependências, plugins e configurações do projeto.

Array

Uma estrutura de dados que armazena múltiplos valores do mesmo tipo, organizados em posições numeradas chamadas índices.

ArrayList

Uma lista dinâmica em Java que pode crescer ou diminuir de tamanho conforme necessário, permitindo armazenar elementos de forma flexível.

Autenticação e Autorização

Processos que garantem que o utilizador é quem diz ser (autenticação) e que tem permissões para realizar determinadas ações (autorização).

BCryptPasswordEncoder

Uma classe do Spring Security usada para encriptar palavras-passe de forma segura.

CamelCase

Uma convenção de nomenclatura em que a primeira palavra é escrita em minúsculas e as subsequentes começam com letra maiúscula (ex.: minhaVariavel).

Ciclo de Vida de uma Aplicação Spring Boot

O conjunto de etapas que uma aplicação Spring Boot passa, desde a inicialização até o encerramento.

Classe BufferedReader

Uma classe em Java usada para ler texto de forma eficiente a partir de ficheiros ou outros fluxos de entrada, linha por linha.

Classe de Serviço (Service)

Uma classe no Spring Boot usada para implementar a lógica de negócios da aplicação, separando-a das camadas de controlo e persistência.

Classe HashMap

Uma estrutura de dados em Java que mapeia chaves a valores, permitindo acesso rápido a elementos através das suas chaves.

Classe HashSet

Uma implementação de *Set* em Java que armazena elementos únicos e utiliza uma tabela de dispersão (hash table) para acesso rápido.

Classe Scanner

Uma classe em Java utilizada para ler a entrada do utilizador ou de ficheiros, permitindo interpretar dados como strings, inteiros ou outros tipos.

Classe

Um modelo ou definição que descreve os atributos (dados) e métodos (ações) de um objeto.

Código Boilerplate

Código repetitivo e padrão que é necessário para configurar ou iniciar certas funcionalidades, mas que raramente varia de aplicação para aplicação.

Coleções em Java (ArrayList, HashSet, HashMap)

Estruturas de dados que permitem armazenar e manipular coleções de elementos.

Compilador Java (javac)

Ferramenta que traduz o código Java escrito pelo programador para bytecode executável pela Máquina Virtual Java (JVM).

Configuração Automática (Auto-Configuration)

Um recurso do Spring Boot que configura automaticamente os componentes da aplicação com base nas dependências presentes no projeto.

Configuração Modular

O ato de dividir uma aplicação em módulos independentes que podem ser configurados separadamente para maior flexibilidade e reutilização.

Configuração por Convenção

Um princípio que reduz a necessidade de configurações explícitas, assumindo valores padrão sensatos, como no Spring Boot.

Configurador de Dependências (Maven)

Uma ferramenta de automação de build que gera dependências e processos de compilação em projetos Java.

Construtor

Um método especial de uma classe usado para criar e inicializar novos objetos.

Controlador (Controller)

No padrão MVC, é o componente responsável por gerir as interações do utilizador e a lógica que conecta os dados ao que é mostrado na interface.

Controlador REST (@RestController)

Uma anotação no Spring que define uma classe como um controlador para lidar com requisições HTTP em APIs REST.

CORS (Cross-Origin Resource Sharing)

Uma política de segurança que controla como os recursos numa página web podem ser acessados por scripts de outro domínio.

CRUD (Create, Read, Update, Delete)

As quatro operações básicas realizadas numa base de dados: criar, ler, atualizar e apagar registo.

Depuração

O processo de encontrar e corrigir erros ou bugs num programa através de ferramentas ou técnicas específicas.

Encapsulamento

Um princípio da programação orientada a objetos que protege os dados de uma classe, permitindo o acesso apenas através de métodos específicos (getters e setters).

EndPoint REST

Uma URL específica numa API REST que permite executar uma ação, como obter ou atualizar dados, utilizando métodos HTTP.

Enterprise JavaBeans (EJB)

Um componente da plataforma Jakarta EE que permite o desenvolvimento de aplicações empresariais escaláveis e seguras.

Execução Independente

A capacidade de uma aplicação, como as criadas com Spring Boot, ser executada sem dependências externas, como servidores de aplicação adicionais.

Expressões Lambda

Funções anónimas em Java, introduzidas no Java 8, que tornam o código mais compacto e funcional.

Extension Pack for Java

Um conjunto de extensões para o Visual Studio Code que facilita o desenvolvimento em Java, incluindo suporte para Spring Boot.

Ficheiros de Configuração (*application.properties*, *application.yml*)

Ficheiros utilizados no Spring Boot para definir parâmetros de configuração da aplicação, como portas, perfis e conexões de base de dados.

FileWriter

Uma classe em Java que facilita a escrita de texto em ficheiros, permitindo criar ou modificar ficheiros no sistema.

Framework

Um conjunto de ferramentas e bibliotecas que ajuda a desenvolver software de forma mais estruturada e eficiente (ex.: Spring Boot, Vaadin).

Gateway API

Um ponto centralizado que gera todas as chamadas para uma API, incluindo autenticação, autorização e transformação de dados.

Gerador de Dependências

Uma ferramenta ou funcionalidade, como o Spring Initializr, que facilita a inclusão de dependências específicas num projeto de forma automática.

Gradle

Uma ferramenta de automação de build semelhante ao Maven, mas que utiliza scripts Groovy ou Kotlin para configuração.

H2 Database

Uma base de dados relacional leve, usada frequentemente para testes ou desenvolvimento local.

Herança

Um conceito da programação orientada a objetos onde uma classe herda atributos e métodos de outra, promovendo a reutilização de código.

HTTP Verbs (GET, POST, PUT, DELETE)

Métodos usados no protocolo HTTP para especificar ações em recursos de uma API.

IDE (Integrated Development Environment)

Um ambiente de desenvolvimento que integra ferramentas como editores de código, depuradores e terminais (ex.: VS Code, IntelliJ IDEA).

Injeção de Dependências (Dependency Injection)

Um padrão de design onde as dependências de uma classe são fornecidas pelo framework, promovendo baixo acoplamento e maior modularidade.

Interface

Um tipo em Java que define métodos que uma classe deve implementar, ajudando a criar sistemas mais flexíveis.

Interfaces Funcionais

Interfaces em Java com apenas um método abstrato, usadas frequentemente em expressões lambda.

Inversão de Controlo (IoC)

Um princípio de design onde o controlo do fluxo do programa é delegado a um framework, permitindo maior flexibilidade e modularidade.

Iteração

A repetição de instruções num programa, geralmente com estruturas como *for*, *while* ou *do-while*.

Jakarta EE

Uma plataforma para desenvolvimento de aplicações empresariais em Java, que evoluiu do Java EE.

Jakarta Message Service (JMS)

Uma API que facilita a comunicação assíncrona entre componentes ou sistemas utilizando mensagens.

Jakarta Server Pages (JSP)

Uma tecnologia baseada em Java para criar páginas web dinâmicas, permitindo integrar HTML e código Java.

Java SE (Standard Edition)

A edição padrão da linguagem Java que fornece as ferramentas básicas para o desenvolvimento de aplicações.

JPA (Java Persistence API)

Uma especificação que permite mapear objetos Java para tabelas de bases de dados relacionais, facilitando a manipulação de dados.

JSON (JavaScript Object Notation)

Um formato leve e legível para troca de dados entre aplicações, frequentemente usado em APIs REST.

JWT (JSON Web Token)

Um formato compacto de token usado para transmitir informações entre partes de forma segura, frequentemente utilizado em autenticação.

Mapa de Rotas

Um conjunto de definições que especifica quais URLs ou caminhos numa aplicação web estão associados a determinados controladores ou ações.

Método

Um conjunto de instruções que executa uma tarefa específica dentro de uma classe.

Modelo (Model)

No padrão MVC, é a camada que gera os dados, a lógica de negócios e as regras da aplicação.

MVC (Model-View-Controller)

Um padrão que organiza uma aplicação em três partes: o Modelo (dados e lógica), a Vista (interface do utilizador) e o Controlador (gestão de interações).

OAuth 2.0

Um protocolo padrão para autenticação e autorização que permite o acesso seguro a recursos protegidos.

Objeto

Uma instância de uma classe que representa algo do mundo real, com atributos (estado) e métodos (comportamento).

ORM (Object-Relational Mapping)

Uma técnica que mapeia objetos em código para tabelas em bases de dados, permitindo manipular dados como se fossem objetos.

Paginação

Uma técnica para dividir grandes conjuntos de dados em páginas menores, facilitando a navegação e reduzindo a carga de processamento.

PascalCase

Uma convenção de nomenclatura onde todas as palavras começam com letra maiúscula, geralmente usada para nomes de classes (ex.: MinhaClasse).

Perfis de Ambiente (Spring Profiles)

Uma funcionalidade do Spring Boot que permite criar configurações específicas para diferentes ambientes, como desenvolvimento, teste ou produção.

POJO (Plain Old Java Object)

Um objeto Java simples, sem dependências externas, frequentemente usado para representar dados.

Polimorfismo

A capacidade de um método ou interface funcionar de várias formas, adaptando-se ao contexto.

Postman

Uma ferramenta que permite testar APIs de forma fácil, simulando requisições HTTP como GET, POST, PUT e DELETE.

Referências a Métodos

Um recurso que permite referenciar diretamente métodos ou construtores, tornando o código mais conciso.

Repositório (Repository)

Uma interface que fornece métodos para acesso e manipulação de dados em bases de dados, usada no Spring Data JPA.

REST (Representational State Transfer)

Um estilo de arquitetura para construir APIs que utilizam os métodos padrão do HTTP (GET, POST, PUT, DELETE).

Servidores Embutidos (Embedded Servers)

Servidores de aplicação, como Tomcat ou Jetty, integrados nas aplicações Spring Boot, permitindo que sejam executadas de forma autónoma.

Servlets

Componentes Java que processam requisições HTTP e geram respostas, sendo a base para muitas aplicações web.

Sobrecarga de Métodos

Criar métodos com o mesmo nome numa classe, mas que aceitam diferentes números ou tipos de parâmetros.

Spring Boot Dashboard

Uma ferramenta visual no Visual Studio Code ou IntelliJ IDEA que facilita a gestão e monitorização de aplicações Spring Boot.

Spring Boot Tools

Uma extensão que adiciona funcionalidades específicas ao desenvolvimento Spring Boot, como autocompletar e integração com o Spring Initializr.

Spring Boot

Uma framework que simplifica o desenvolvimento de aplicações Java, oferecendo configurações automáticas e ferramentas integradas.

Spring Initializr

Uma ferramenta online que permite criar rapidamente projetos Spring Boot, configurando dependências e definições iniciais.

Spring Security

Um módulo do Spring que oferece soluções robustas para autenticação, autorização e proteção de aplicações.

Starters (Spring Boot Starters)

Conjuntos de dependências pré-configuradas no Spring Boot para adicionar funcionalidades como segurança, web ou acesso a bases de dados.

Streams em Java

Uma API para processar coleções de dados de forma declarativa, introduzida no Java 8.

String

Uma classe em Java usada para representar e manipular texto.

Teste de Integração

Testes que verificam a interação entre diferentes componentes de uma aplicação para garantir que trabalham juntos corretamente.

Teste Unitário

Testes automáticos que verificam o funcionamento correto de partes individuais do código, como métodos ou funções.

Tipo Primitivo

Tipos básicos de dados em Java, como números inteiros (*int*), decimais (*double*), caracteres (*char*) e valores lógicos (*boolean*).

TLS (Transport Layer Security)

Um protocolo de segurança que encripta a comunicação entre servidores e clientes, garantindo privacidade e integridade dos dados.

Tomcat

Um servidor de aplicação embutido no Spring Boot, usado para processar requisições web.

Vaadin Flow

Um módulo do Vaadin que permite criar interfaces web ricas utilizando apenas Java, eliminando a necessidade de HTML ou JavaScript.

Vaadin

Um framework Java para criar interfaces web dinâmicas e modernas, usado frequentemente com Spring Boot.

Validação de Dados

O processo de garantir que os dados fornecidos pelo utilizador cumprem regras específicas antes de serem processados ou armazenados.

Variáveis de Ambiente

Valores configurados no sistema operativo ou na aplicação para definir o comportamento da aplicação sem alterar o código, como credenciais de base de dados ou URLs de APIs.

Variável

Um local na memória do computador onde são armazenados valores que podem mudar durante a execução de um programa.

Vista (View)

No padrão MVC, é a camada responsável pela interface do utilizador, exibindo dados e capturando as suas interações.

WildFly (Servidor de Aplicação)

Um servidor de aplicação open-source utilizado para implementar, gerir e executar aplicações Java empresariais.

YAML (YAML Ain't Markup Language)

Um formato de ficheiro legível por humanos, utilizado para configurar aplicações de forma hierárquica, como no Spring Boot.

Índice Remissivo

- @Autowired, 5, 99, 100, 101, 102, 106, 123, 125, 126, 131, 140, 143, 144
- @Component, 5, 99, 100, 101, 102, 127, 188
- @GetMapping, 4, 68, 69, 70, 71, 77, 92, 123, 132, 133, 140
- @PathVariable, 4, 70, 71, 77, 92, 132, 133
- @Repository, 85, 91, 125, 127, 188
- @RequestMapping, 4, 68, 69, 70, 71, 77, 92, 122, 131, 135, 140
- @RestController, 4, 68, 69, 70, 71, 77, 92, 122, 128, 131, 133, 135, 140, 190
- @Route, 6, 160, 161, 163, 165, 167, 172, 175, 176, 179, 181, 183, 186
- @Service, 126, 127, 188
- application.properties, 4, 62, 63, 64, 65, 66, 67, 73, 74, 78, 81, 83, 89, 113, 144, 155, 191
- application.yml, 4, 64, 65, 66, 67, 83, 113, 144, 191
- ArrayList, 33, 36, 37, 40, 41, 44, 45, 47, 77, 117, 188, 189
- Autenticação, 6, 108, 150, 151, 152, 153, 154, 158, 188, 191, 193, 195
- BufferedReader, 38, 39, 41, 42, 189
- Bytecode, 17, 19, 189
- Configuração automática, 49, 50, 190
- Construtores, 84, 100, 116, 117, 118, 119, 138, 194
- Controladores REST, 2, 4, 5, 68, 71, 87, 105, 131, 133, 141, 143, 144, 149
- Debugger for Java, 54
- Encapsulamento, 27, 30, 31, 125, 131, 190
- Engenharia de Software, 2, 5, 108, 109
- Eventos do ciclo de vida, 6, 181
- Expressões lambda, 43, 44, 169, 191, 192
- Extension Pack for Java, 18, 54, 191
- File, 38, 42, 60, 72, 74, 76, 89, 95
- FileWriter, 38, 39, 41, 42, 191
- HashMap, 35, 37, 136, 189
- HashSet, 34, 37, 118, 189
- Herança, 28, 31, 187, 191
- Injeção de Dependências (DI), 13, 99
- Interfaces funcionais, 43, 44, 48, 169, 192
- Inversão de Controlo (IoC), 5, 13, 99, 192
- Jakarta EE, 3, 9, 10, 11, 12, 14, 15, 16, 191, 192
- Java SE, 3, 9, 11, 12, 192
- Java Test Runner, 54
- JDK, 4, 11, 17, 18, 52, 59
- JVM, 9, 17, 94, 189
- Manipulação de Ficheiros, 3, 37, 38
- Maven, 4, 13, 52, 53, 54, 57, 58, 59, 60, 63, 72, 87, 89, 103, 109, 112, 154, 163, 164, 188, 190, 191
- Model-View-Controller (MVC), 129, 131
- Modificadores de acesso, 27
- Navegação entre vistas, 6, 161, 175
- Objeto, 26, 27, 28, 29, 31, 46, 80, 110, 111, 114, 149, 185, 189, 193, 194
- Padrões de design, 5, 108, 109, 199
- Permissões, 150, 153, 157, 158, 181, 183, 184, 188
- Polimorfismo, 28, 31, 194
- Spring Framework, 3, 4, 13, 15, 16, 49, 50, 51
- Spring Security, 2, 5, 6, 150, 151, 154, 157, 158, 189, 195
- Streams, 43, 44, 45, 47, 48, 195
- Testes, 5, 54, 63, 98, 102, 103, 104, 105, 107, 112, 128, 131, 141, 142, 143, 144, 145, 149, 154, 155, 191, 195
- Testes de controladores REST, 144
- Testes de repositórios, 143
- Testes de serviços, 104, 143
- Testes unitários, 5
- Vaadin Security, 6, 158

Bibliografia

Coelho, P. (2016). *Programação em Java: Curso completo* (5.ª ed. atualizada). FCA.

Duarte, A. (2021). *Practical Vaadin*. Apress.

Marcelino, M. J., & Mendes, A. J. (2016). *Fundamentos de programação em Java* (4.ª ed. atualizada e aumentada). FCA.

Martins, F. M. (2017). *Java 8: POO + construções funcionais*. FCA.

Musib, S. (2022). *Spring Boot in practice*. Manning Publications Co.

Turnquist, G. L. (2023). *Learning Spring Boot 3.0: Build Modern, Cloud-Native, and Distributed Systems Using Spring Boot* (3rd ed.). Packt Publishing.

Sugestões de Recursos Adicionais para Aprofundamento

Para expandir seu conhecimento e habilidades, considere explorar os seguintes recursos:

- **Documentação Oficial do Spring:** Uma fonte abrangente e detalhada sobre os diversos projetos Spring.
- **Documentação Oficial do Vaadin:** Estão sempre a aparecer novidades, uma das quais a possibilidade de usar assistentes de IA (Vaadin Copilot) para assistir na elaboração do UI.
- **Tutoriais e Artigos Especializados:** Plataformas como Baeldung oferecem tutoriais aprofundados sobre Spring Boot e JPA.
- **Cursos Online:** Plataformas como Udemy e Coursera disponibilizam cursos focados em Spring Boot e desenvolvimento de APIs RESTful.
- **Repositórios no GitHub:** Explore projetos open-source para ver implementações práticas e padrões de design em uso.