EST – IPCB

LICENCIATURA EM ENGENHARIA INFORMÁTICA

Aplicações Distribuídas

A Minha Segunda Aplicação Multinível Spring Boot Rest MVC com Spring Data JPA

3º Ano / 1º Semestre – 2022/2023

Atividade Prática n.º5 - A Minha Segunda Aplicação Multinível Spring Boot Rest MVC com Spring Data JPA

Nesta atividade pretende-se que devolva uma segunda aplicação multinível chamada **AppVendas-SpringData-Jpa** novamente com Spring Data JPA que usa a base de dados em memória H2.

Esta atividade diferencia-se da atividade 4, por aprofundar o estudo da JPA em particular:

- O Mapeamento Objeto-Relacional e utilização das principais anotações usadas para o efeito.
- A linguagem Java Persistency Query Language (JPQL)
- Implementação de *Métodos Query usando Queries personalizadas usando a anotação* ***@Query***

Para melhor compreensão dos conteúdos deste guião sugere-se a consulta do bloco de slides: AD-2023-Modulo-4-Spring_Data.pdf.

Passo 1: Criar uma aplicação multinível Spring Boot chamada AppVendas-SpringData-Jpa:

Utilize o Spring Initializr (https://start.spring.io) para adicionar ao ficheiro pom.xml os Starter Spring Web, Spring Data JPA, Actuator, e H2.

1. Escolha o **Project** do tipo **Maven Project**.
2. Escolha a linguagem **Java**.
3. Escolha a versão do Spring Boot. Utilize a última versão estável disponível.
4. Introduza os metadados do projecto – group ID, artifact ID, name of the project, project description, e package name.
5. Escolha **Packaging** as **Jar**.
6. Escolha **Java 17 ou Java 19 (a versão mínima é Java 8)**.
7. Adicione as dependências – **Spring Web, Spring Data JPA, Actuator e H2**.
8. Clique o botão **Generate**.
9. Descomprima o ficheiro .zip.

Passo 2 (opcional): Verifique se foram adicionadas ao pom.xml as dependências do driver da base de dados H2 e do *starter* Spring Data Jpa:

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

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

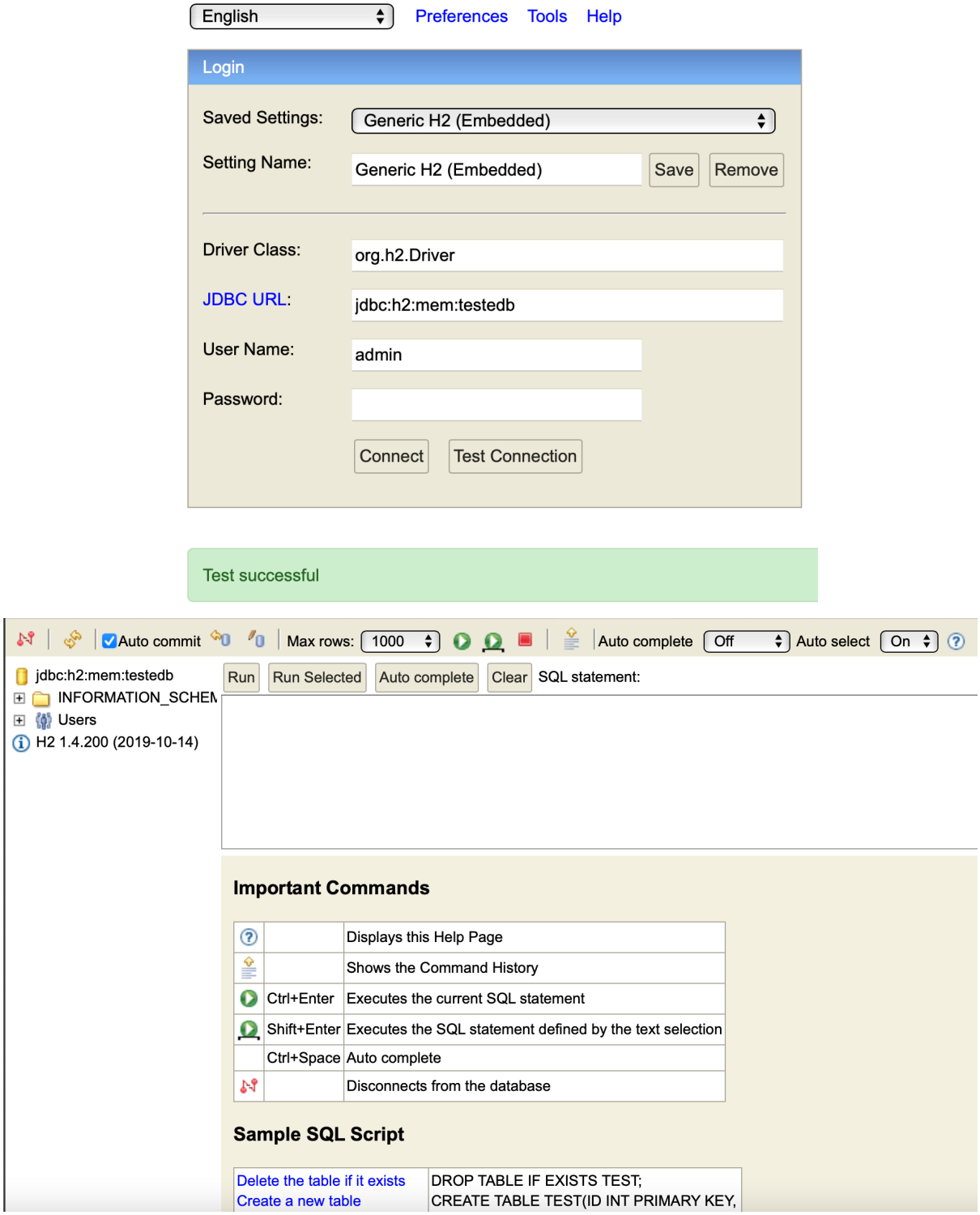
Passo 3: No ficheiro application.properties inclua as seguintes propriedades para criar uma base de dados em memória (**mem**) chamada **testedb** e uma conta de administrador com as credenciais **admin, admin**

```
spring.datasource.url=jdbc:h2:mem:colocar aqui o nome da bd
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=colocar aqui o user name
spring.datasource.password=colocar aqui a password
spring.datasource.initialization-mode=always
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

Ative a consola de administração da base de dados H2:

```
spring.h2.console.enabled=true
```

Passo 4: Execute a aplicação Spring Boot e experimente o acesso à consola da H2 acedendo ao URL da aplicação: **localhost:8080/h2-console** no JDBC URL coloque: **jdbc:h2:mem:testedb**


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:mem:testedb
User Name:
admin
Password:
Connect
Test Connection
Test successful
Auto commit
Max rows:
1000
Auto complete
Off
Auto select
On
jdbc:h2:mem:testedb
INFORMATION_SCHEM
Users
H2 1.4.200 (2019-10-14)
Run
Run Selected
Auto complete
Clear
SQL statement:
Important Commands
Displays this Help Page
Shows the Command History
Ctrl+Enter
Executes the current SQL statement
Shift+Enter
Executes the SQL statement defined by the text selection
Ctrl+Space
Auto complete
Disconnects from the database
Sample SQL Script
Delete the table if it exists
Create a new table
DROP TABLE IF EXISTS TEST;
CREATE TABLE TEST(ID INT PRIMARY KEY,

Passo 6: Considere o seguinte modelo Físico de dados:

```
CREATE TABLE CLIENTE (
    ID INTEGER PRIMARY KEY AUTO_INCREMENT,
    NOME VARCHAR(100),
    MORADA VARCHAR(100),
    DATA_NASC TIMESTAMP
);

CREATE TABLE PEDIDO (
    ID INTEGER PRIMARY KEY AUTO_INCREMENT,
    DATA_PEDIDO TIMESTAMP,
    TOTAL NUMERIC(20,2),
    CLIENTE_ID INTEGER REFERENCES CLIENTE (ID)
);

CREATE TABLE PEDIDO_ITEM (
    ID INTEGER PRIMARY KEY AUTO_INCREMENT,
    QUANTIDADE INTEGER,
    PEDIDO_ID INTEGER REFERENCES PEDIDO (ID),
    PRODUTO_ID INTEGER REFERENCES PRODUTO (ID)
);

CREATE TABLE PRODUTO (
    ID INTEGER PRIMARY KEY AUTO_INCREMENT,
    DESCRICAO VARCHAR(100),
    PRECO_UNITARIO NUMERIC(20,2)
);
```

Como vimos antes podemos delegar no Spring Boot para criar o Modelo físico a partir das classes entidade em Java. Alternativamente, podemos criar um script SQL na pasta de recursos do projeto que será executado durante o start-up da App.

Posteriormente, será necessário realizar manualmente o mapeamento JPA, criando as correspondentes classes entidade devidamente anotadas.

Passo 7: Na pasta **`resources`**, crie um novo ficheiro **`schema.sql`** com as expressões SQL.

- Execute a aplicação. Terá ocorrido um erro "Produto" not found :-)
- Corrija o SQL.

Passo 8: Aceda à Consola da H2 e verifique se as tabelas foram criadas.

Passo 9: Crie as Classes Modelo Cliente, Pedido, Produto e PedidoItem.

Passo 10: Realize o mapeamento entre as Entidades Cliente e Pedido usando as anotações JPA estudadas:

- @Entity, @Table, @Id, @Column
- @OneToMany, @ManyToOne, @JoinColumn.

Exemplo parcial de implementação

```
@Entity
@Table(name="CLIENTE")
public class Cliente {
    @Id
    @Column(name="ID")
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    @Column(name="NOME,length=100)
    private String nome;

    //Setters e Getters
}

@Entity
@Table(name="PEDIDO")
public class Pedido {
    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private Timestamp data_pedido;
    private Double total;
    @ManyToOne
    @JoinColumn(name="CLIENTE_ID,referencedColumnName="ID")
    private Cliente cliente;

    //Setters e Getters
}
```

Passo 11: Crie os Repositórios das Entidades Cliente e Pedido do tipo JpaRepository: RepositorioCliente e RepositorioPedido.

Passo 12: No método run() de um ApplicationRunner insira dois registos Cliente e dois registos Pedido.

Passo 13: Crie as classes Entidade Pedido_Item e Produto, e faça também o mapeamento JPA dos Pedidos e Produtos. De seguida, declare as interfaces Repositório.

Implementação Parcial

```
@Entity
@Table(name = "PEDIDO_ITEM")
public class ItemPedido {
    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    private Integer id;
    private BigDecimal quantidade;
    @JoinColumn(name = "pedido_id")
    @ManyToOne
    private Pedido pedido;
    @ManyToOne
    @JoinColumn(name = "produto_id")
    private Produto produto;
    //Getters e Setters
}

@Entity
public class Produto {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String descricao;
    private BigDecimal preco_unitario;
…
}
```

Passo 14: Pretende-se que ensaie a criação de queries personalizadas JPQL e nativas usando a anotação @Query.

- Crie um método que cria um Cliente. A JPQL não suporta *inserts*. Defina um método save().
- Usando uma query JPQL consulte todos os clientes e ordene a consulta pelo Id.
- Usando uma query JPQL consulte os clientes pelo nome usando a cláusula LIKE.

- Usando uma query nativa insira um novo Cliente. Este método precisa de ser anotado com @Modifying. Poderá também ser necessária a anotação @Transactional
- Usando uma query JPQL atualize o nome de um Cliente pelo Id. Este método deve retornar void. Este método precisa de ser anotado com @Modifying(clearAutomatically = true) e @Transactional.
- Repita, agora usando uma query Nativa, consulte os clientes pelo nome usando a cláusula LIKE.
- Usando uma query Nativa insira um novo Pedido.
- Usando uma query JPQL apague um cliente pelo seu Id. Este método precisa de ser anotado com @Modifying.
- Usando uma query nativa atualize a data do Pedido pelo Id. Este método deve retornar void.

Precisa de declarar as queries e especificar os métodos query nos repositórios.

Ensaie as chamadas dos métodos num Application Runner.

Nos passos seguintes pretende-se que faça uma consulta com os relacionamentos JPA, designadamente quando consultar o registo de um cliente, a Lista dos pedidos de um cliente sejam carregados da Base de dados e quando consultar um pedido que seja carregado o cliente.

A JPA pensou nos problemas/vantagens e por omissão as relações One-to-Many carregam junto preguiçosamente (LAZY). No nosso exemplo, isso significa que por omissão ao consultarmos um Cliente, os pedidos não são carregados da base de dados. Apenas mais tarde a pedido (on-demand).

Pelo contrário, as relações Many-to-One por omissão carregam junto *eagerly*, o que significa que a relação é carregada ao mesmo tempo que a entidade o é.

Como podemos configurar/personalizar estes carregamentos?

Usamos o atributo fetch (trazer junto) das anotações @OneToMany ou @ManyToOne para se definir explicitamente a forma escolhida.

```
@ManyToOne(fetch = FetchType.EAGER)
@ManyToOne(fetch = FetchType.LAZY
```

Ensaio Fetch EAGER (passos 15 a 17)

Passo 15 - Em adição ao passo 10, declare o mapeamento também na entidade Cliente e defina o *fetchtype* como EAGER.

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

Passo 16: Ensaie o método query findById() e verifique o conteúdo da Lista de pedidos for carregada juntamente.

Passo 17: Pode também fazer o fetch do cliente quando consulta por um pedido:

```
@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "CLIENTE_ID")
private Cliente cliente;
```

Ensaio Fetch LAZY (passos 18 a 20)

Passo 18: Em adição ao passo 17, declare o *fetchtype* como LAZY.

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

Passo 19: Ensaie o método query findById() e verifique o conteúdo da Lista de pedidos. OPS!!! Erro!

Neste caso precisa de fazer uma consulta on-demand subsequente para que os pedidos sejam carregados.

Passo 20: O fetch tem que ser realizado num pedido subsequente à parte usando uma Query apropriada:

```
@Query("select c from Cliente c left join fetch
c.pedidos where c.id =:id")
Cliente findClienteByIdFetchPedidos(@Param("id")
BigInteger id);
```

Um relacionamento pode ser *optional* ou obrigatório.

- Considerando o lado One-to-Many - é sempre opcional, e não podemos fazer nada quanto a isso.
- O lado Many-to-One, por outro lado, oferece-nos a opção de o tornar obrigatório.
  @ManyToOne(optional = false)
  @ManyToOne(optional = true)

Passo 21: Implementar num ControladorCliente os métodos ação CRUD devidamente anotados com @PostMapping, @GetMapping, @PutMapping e @Delete, etc similares à implementação da ficha 4 (ver de seguida):

- Injete no ControladorCliente o repositório de dados dos Cliente.
- Sugere-se as seguintes assinaturas para os métodos Ação.

//URI: POST /clientes
@PostMapping(("/clientes")
```
Cliente criar(@RequestBody Cliente cliente);
```
Este método cria um cliente e retorna o cliente criado.

//URI: GET /clients
@GetMapping("/clientes")
```
List<Cliente> findAll();
```
Este método retorna a lista de todos os clientes.

//URI: GET /clientes/{id}
@GetMapping("/clientes/{id}")
```
Optional<Cliente> consultar(@PathVariable("id") long id);
```
Este método permite a consulta do registo da conta id.

//URI: GET /clientes/nome/{nome}
@GetMapping("/clientes/nome/{nome}")
```
Optional<Cliente> consultar(@PathVariable("nome") String nome);
```

//URI: PUT /clientes/{id}
@PutMapping("/clientes/{id}")
```
void editar(@RequestBody Cliente cliente, @PathVariable("id")
long id);
```
Este método permite atualizar o registo do cliente com o id.
Nota: A JPA Repository não suporta explicitamente um método merge() como na JPA nativa. O método save(Cliente cliente) internamente se verificar que a conta já existe faz o merge das atualizações.

//URI: DELETE /clientes/{id}
@DeleteMapping("/clientes/{id}")
```
void remover(@PathVariable("id") long id);
```
Este método permite remover o registo da conta id.

- Implemente também os métodos correspondentes às mesmas operações suportadas pelo repositório de dados dos Pedidos. Se necessário implemente os métodos e as Queries em falta.

Passo 22: Teste a implementação usando o Browser e/ou a Ferramenta Postman.

Métodos Get

Para testar os métodos Get precisa apenas de um simples browser.

Métodos Post, Put, e Delete

Para testar estes métodos precisará de usar a ferramenta de testes de APIs REST (e.g., Postman) ou de usar o comando cURL.

Passo 23(opcional): Adicione um Actuator à aplicação e verifique a "saúde" da aplicação.

Para ativar um Spring Boot Actuator, apenas precisamos de adicionar a dependência do spring-boot-actuator ao gestor de pacotes (pom.xml):

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

Efetue as seguintes configurações no ficheiro application.properties com as informações gerais a mostrar sobre a aplicação :

```
## Configuring info endpoint for Atuator
info.app.name= Minha Primeira Aplicação Multinível Spring Boot Rest MVC
info.app.description=Esta aplicação ilustra o desenvolvimento de uma aplicação multinível Spring Boot
info.app.version=1.0.0
## Expose all actuator endpoints
management.endpoints.web.exposure.include=*
```

FIM