

Atividade Prática n.º4 - A Minha Primeira Aplicação Multinível Spring Boot Rest MVC com Spring Data JPA

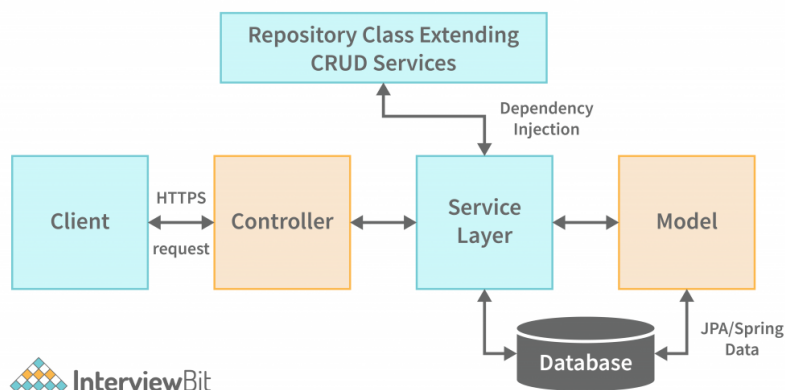
Nesta atividade pretende-se que desenvolva uma aplicação Multinível Spring Boot Rest MVC com Spring Data JPA, suportada por uma base de dados embecida H2.

NOTA: Esta atividade pode ser realizada começando de novo criando um novo projeto e seguindo as instruções ou continuando e adaptando o projeto da atividade prática n.º3.

Arquitetura do Fluxo de Trabalho do Spring Boot

Spring Boot depende fortemente da estrutura Spring, o que significa que integra quase todas as características e módulos Spring, como Spring MVC, Spring Core, etc.

Spring Boot Flow Architecture



Ao longo desta atividade, pretende-se que ao realizar os passos seguintes experimente diversas funcionalidades do Spring Boot e possa ficar a conhecer algumas características das aplicações Spring Boot MVC + Rest com Spring Data JPA.

Mais à frente, numa das próximas atividades será aprofundado o estudo/uso do Spring Data JPA.

Para melhor compreensão dos conteúdos deste guião sugere-se a consulta do bloco de slides: AD-2023-Bloco_1_Spring_e_Spring_Boot.pdf.

Passo 1: Criar uma aplicação multinível SpringBoot MVC chamada DemoMVC- Rest-SpringData-JPA:

Utilize o Spring Initializr (<https://start.spring.io>) para adicionar ao ficheiro pom.xml os Starter Spring Web, 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.

Alternativamente, pode continuar o projeto da atividade prática n.º3, adicionando ao ficheiro **pom.xml** o Starter para se usar bases de dados SQL:

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

Passo 2 (opcional): Verifique se foi adicionada a dependência do driver da base de dados.

Se continuar o projeto da atividade prática n.º3, adicione no **pom.xml**:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</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 Run Run Selected Auto complete Clear SQL statement:

INFORMATION_SCHEMA Users H2 1.4.200 (2019-10-14)

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	DROP TABLE IF EXISTS TEST;
Create a new table	CREATE TABLE TEST(ID INT PRIMARY KEY,

Passo 5: A aplicação é composta por um modelo de dados **Conta**, um controlador `@RestController` `ControladorContas`, um `@Service` `ServicoContas`, e um `@Repository` `RepositorioContas`.

Uma instância da classe **Conta** corresponderá a um registo de uma conta bancária num sistema de armazenamento de dados (e.g., Base de Dados Relacional ou NoSql). Na presente atividade utilizaremos um sistema de armazenamento em memória baseado numa base de dados em memória H2.

Passo 5.1 Realize o seguinte:

- Crie a classe **Conta**.
- Anote a classe **Conta** com a anotação JPA `@Entity`, o campo id com `@Id` e que a estratégia de geração da chave primária é auto-incremento.
- Adicione à classe os *getters* e *setters*, o *constructor* por omissão e o construtor que inicializa todas as propriedades, excepto o ID da conta.

```
@Entity
public class Conta {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String titular;
    private String morada;
    private long nif;
    private long pin;
    private double saldo;
    //getters e setters

    ...

}
```

Passo 5.2 De seguida realize o seguinte:

- Crie uma classe `ControladorContas` e anote com `@RestController`
- Crie um serviço `ServicoContas` e anote com `@Service`
- Crie uma interface `RepositorioContas` e anote-a com `@Repository`

Nota: Se realizou a atividade n.º2 somente precisa de remover a classe `RepositorioContas` e criar a Interface `ReposiorioContas`.

```
@RestController
public class ControladorContas
{
    // Métodos que executam as ações
}

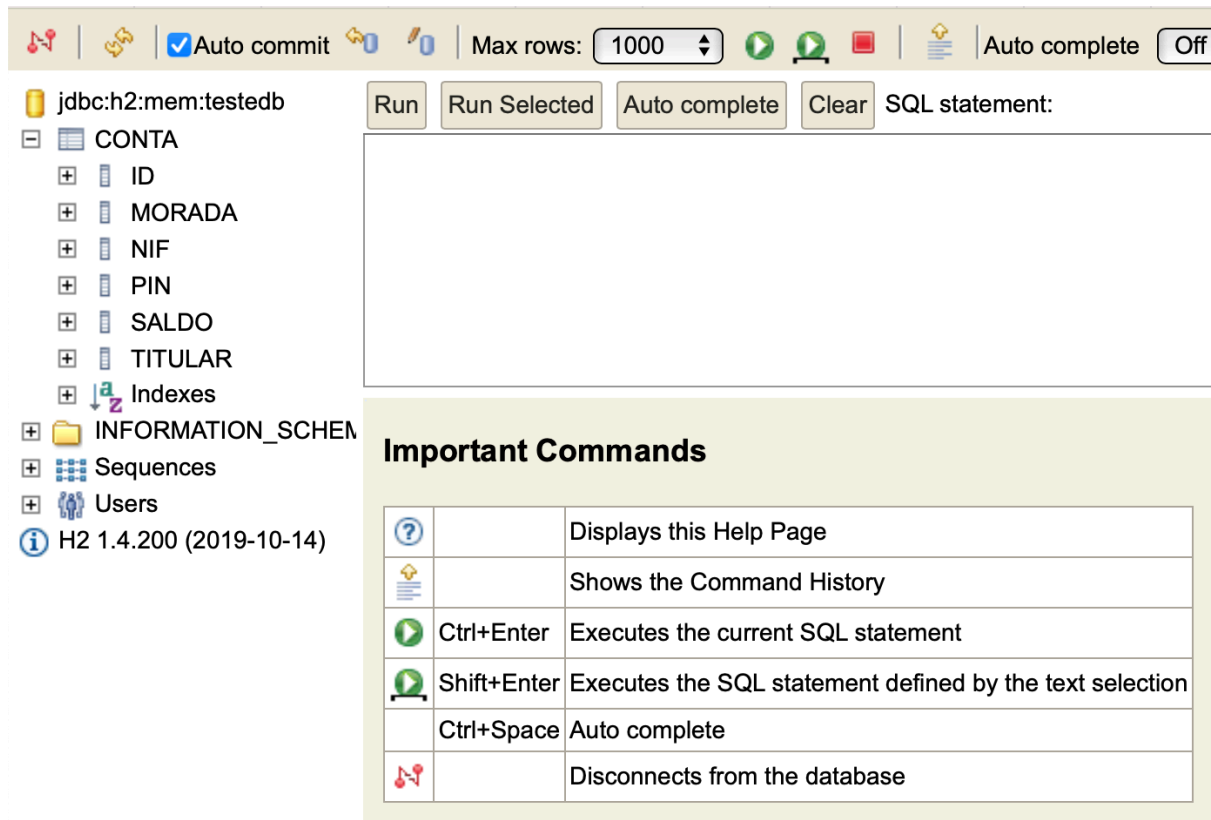
...
```

O Controlador REST a implementar “embrulha” o serviço `ServicoContas` com uma camada web/REST.

O serviço permite realizar as operações CRUD à conta bancária. Para tal este usa um repositório de dados.

O repositório realiza os acessos/as consultas ao sistema de dados utilizado.

Passo 6 (opcional): Execute a aplicação e aceda à consola da H2 e verifique que a tabela CONTA foi criada pelo Spring Boot.



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

Passo 7: Interface `RepositorioContas` tem que *extends* uma das interfaces *Repository*: `Repository`, `JpaRepository`, `CrudRepository` ou `PagingAndSortingRepository`. Considere na implementação a `JpaRepository`.

```
public interface RepositorioContas extends JpaRepository<tipo  
de entidade, tipo da chave primária> {
```

```
...  
}
```

Passo 8: Declarar 6 métodos query que permitem as seguintes consultas:

- insere uma conta
- procura pelo Id
- retorna o número total de contas
- procura pelo titular
- procura pelo titular e morada

- procura pelo saldo maior do que, por exemplo 1000.0 euros

Complete o estudo realizando as seguintes consultas:

- retorna o número de contas de um titular
- apaga uma conta pelo Id
- procura pelo titular tipo (like) com ordenação alfabética
- procura pelo saldo menor ou igual do que 1000.0 euros, mas devolve apenas no máximo 3 contas. Dica: coloque a *keyword* Max3 entre find...By
- procura pelo saldo maior do que por exemplo 1000.0 euros, mas devolve apenas as top 3 contas ordenadas por ordem decrescente. Dica: coloque a *keyword* Top3 entre find...By
- atualiza o nome do titular de uma conta com um determinado Id.

Nota: Sobre a atualização de 1 registo: A JPA Repository não suporta explicitamente um método merge() como na JPA nativa. Contudo, o método save(Conta conta) internamente verifica se a conta já existe; caso esta exista faz o merge das atualizações. Senão cria nova conta.

Nesta implementação:

- Considere que os nomes dos métodos seguem as convenções/palavras-chave definidas em:

<https://docs.spring.io/spring-data/data-jpa/docs/current/reference/html/#jpa.query-methods.query-creation>

<https://docs.spring.io/spring-data/data-jpa/docs/current/reference/html/#repository-query-keywords>

- Os métodos do não devem retornar null. Para dar corpo a este requisito utilize sempre que possível a classe Optional introduzida no Java 8.

Passo 9 (Opcional): Mostrar na consola o SQL gerado pelo Spring Data JPA.

```
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Passo 10 (Opcional): Configure a aplicação para persistir os dados da Base de Dados H2 num ficheiro na área de trabalho do projeto.

Adapte a configuração:

```
spring.datasource.url=jdbc:h2:mem:testedb
```

Para:

```
spring.datasource.url=jdbc:h2:file:./testedb
```

Abra a pasta do projecto e verifique se foram criados os ficheiros da base de dados testedb.

Passo 11: Implementar o serviço ServicoContas

De momento, este serviço simplesmente reencaminha os pedidos para as operações CRUD chamando os correspondentes métodos do **RepositorioContas**.

Passo 12: Implementar um Application/Command Line Runner

Implementar um Runner que usa o ServicoContas para inserir 5 contas na base de dados, com saldos de 1000.O, 10000.O e sucessivamente maiores.

Aceda à consola da base de dados H2 e verifique se os registos foram criados.

Passo 13: Implementar no ControladorContas os métodos ação CRUD seguintes anotados com **@PostMapping**, **@GetMapping**, **@PutMapping** e **@Delete**, respetivamente:

Altere a implementação da atividade prática n.º2, por forma a respeitar as recomendações REST (siga as instruções do docente):

//URI: POST /contas

@PostMapping

Conta criar(@RequestBody Conta conta) ;

Este método cria uma conta e retorna a conta criada.

//URI: GET /contas/{id}

@GetMapping

Optional<Conta> consultar(@PathVariable("id") long id) ;

Este método permite a consulta do registo da conta id.

//URI: PUT /contas/{id}

@PutMapping

void editar(@RequestBody Conta conta, @PathVariable("id") long id) ;

Este método permite atualizar o registo da conta id.

Nota: A JPA Repository não suporta explicitamente um método merge() como na JPA nativa. O método save(Conta conta) internamente se verificar que a conta já existe faz o merge das atualizações.

//URI: DELETE /contas/{id}

@DeleteMapping

void remover(@PathVariable("id") long id) ;

Este método permite remover o registo da conta id.

Implemente os restantes métodos correspondentes às restantes operações suportadas pelo repositório de dados.

Passo 14: Teste a implementação

Métodos Get

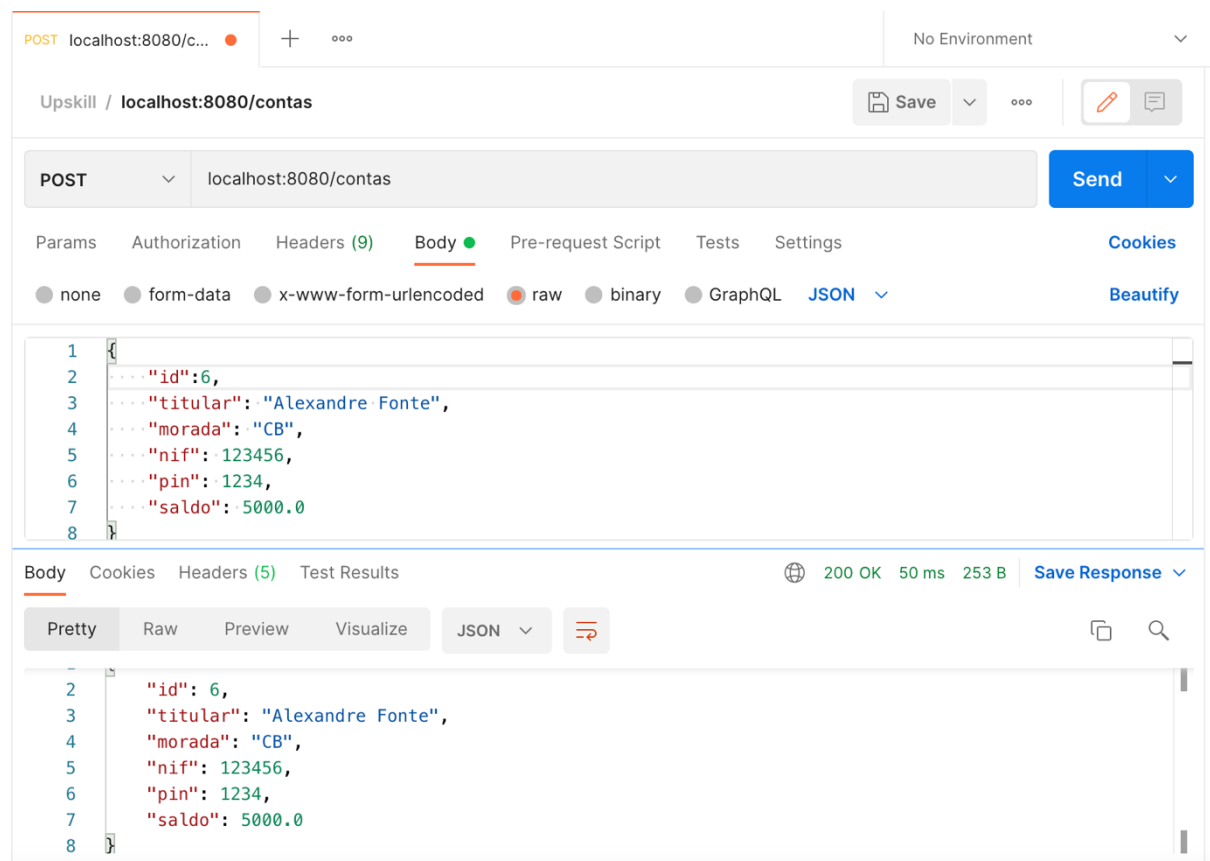
Para testar os métodos Get precisa apenas de um simples browser.

Na figura seguinte usando um browser Web, ilustra-se o resultado esperado para uma invocação do método ação consulta pelo Id.



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.



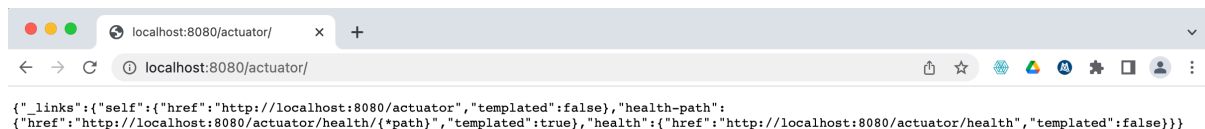
Passo 15: Adicione um Actuator à aplicação MVC e verifique a “saúde” da aplicação.

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