



# Desenvolvimento de Sistemas

---

## Banco de dados: integração, manipulação, registros

Neste conteúdo, você verá como criar uma API com Spring Boot integrando ao banco de dados com **Spring Data JPA** e uma aplicação *web* utilizando **Spring MVC**. Será criado inicialmente o projeto na página do **Spring Initializr** e aberto o projeto na IDE NetBeans. O projeto que você desenvolverá consiste em uma API de Funcionários, no qual haverá as rotas, *endpoints*, de um CRUD para cadastrar, atualizar, excluir, buscar todos os registros e buscar um funcionário específico. Na sequência, a API será expandida, criando recursos avançados de validações de dados e personalização de mensagens de erros por *exception*. Além disso, serão criadas consultas avançadas com Spring Data JPA. Na última seção deste material, haverá a criação de uma aplicação *web* utilizando Spring MVC.

## Criando um projeto Spring Boot com banco de dados

Para começar, você precisa acessar a página do Spring Initializr (<https://start.spring.io/>), onde serão definidas todas as configurações iniciais do seu projeto:



- ◆ Em **Project**, selecione o **Maven**.
- ◆ Em **Language**, escolha o **Java**.
- ◆ Em **Spring Boot**, escolha a versão mais atualizada, a **3.0.6**.
- ◆ Em **Project Metadata**, defina alguns metadados para a criação do projeto.
  - ◆ Group: informar **com.api**
  - ◆ Artifact e Name: informar **senac**
  - ◆ Description: **informar API Rest de Projeto de Funcionários**
  - ◆ Package name: **com.api.senac**
  - ◆ Packaging: selecionar **Jar**
  - ◆ Java: indicar a versão **17**

**Project**

☐ Gradle - Groovy   ☐ Gradle - Kotlin   ☒ **Java**   ☐ Kotlin   ☐ Groovy  
☒ **Maven**

**Language****Spring Boot**

☐ 3.1.0 (SNAPSHOT)   ☐ 3.1.0 (RC1)   ☐ 3.1.0 (M2)   ☐ 3.0.7 (SNAPSHOT)  
☒ **3.0.6**   ☐ 2.7.12 (SNAPSHOT)   ☐ 2.7.11

**Project Metadata**

Group   **com.api**

Artifact   **senac**

Name   **senac**

Description   **API Rest de Projeto de Funcionários**

Package name   **com.api.senac**

Packaging   ☒ **Jar**   ☐ War

Java   ☐ 20   ☒ **17**   ☐ 11   ☐ 8

Figura 1 – Site do Spring Initializr

Fonte: Spring Initializr (s.d.)



Por fim, devem ser definidas as dependências do projeto:

- ◆ Spring Web (spring-boot-starter-web): utilizada para criação de API em Spring Boot, incluindo REST em aplicações utilizando o Spring MVC e Tomcat como um servidor-padrão.
- ◆ Spring Data JPA (spring-boot-starter-data-jpa): permite persistir dados em bancos de dados SQL usando Spring Data e Hibernate, que é uma implementação do JPA.
- ◆ MySQL Driver (mysql-connector-java) é um driver para conexão com banco de dados MySQL para Spring Boot.
- ◆ Validation (spring-boot-starter-validation) é uma dependência usada para validar os campos de um objeto antes de persistir no banco de dados.
- ◆ Lombok é uma biblioteca auxiliar para adicionar automaticamente diversos métodos, como get e set, toString e construtores, ou seja, toda vez que é adicionada a anotação `@Data` em uma classe, você não vai mais precisar escrever métodos getter, setter, equals, hashCode, construtores de classe etc.

## Criando validações e personalizando mensagens com *exceptions*

Um recurso muito importante adotado nas APIs são as validações nas requisições. O Spring Boot, por meio da dependência **Validation**, permite indicar quais atributos você quer validar em uma inclusão e/ou alteração. Por exemplo, você pode criar validações na classe **FuncionarioEntity** para os atributos CPF, *e-mail*, nome com quantidade mínima de caracteres, campos obrigatórios, entre outras. Para isso, basta apenas colocar a anotação de validação correspondente sobre o nome do atributo. A seguir, veja um resumo das principais validações.

- ◆ **@NotBlank @NotNull**: essas duas anotações realizam verificações e validam os campos onde são mapeadas para garantir que os valores não estejam em branco e nulos.

- ◆ **@CPF e @CNPJ**: valida se o campo anotado possui CPF ou CNPJ válido.
- ◆ **@Email**: valida se o campo anotado é um endereço de e-mail válido.
- ◆ **@Size**: especifica a quantidade mínima (min.) e máxima (máx.) de caracteres.
- ◆ **@Min**: valida que a propriedade anotada tem um valor não menor que o valor do atributo.
- ◆ **@Max**: valida que a propriedade anotada tem um valor não maior que o valor do atributo.

Veja como fica a classe **FuncionarioEntity** com as anotações de validação nos atributos: CPF, nome, telefone, e-mail e salário.

```
package com.api.senac.data;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;
import lombok.Data;
import org.hibernate.validator.constraints.br.CPF;

@Data
@Entity
@Table(name="Funcionario")

public class FuncionarioEntity {

    @Id
```

```
@GeneratedValue(strategy = GenerationType.AUTO)

private Integer id;


@CPF(message="CPF inválido")

private String cpf;


@Size(min=2, message="Informe ao menos 2 caracteres para o campo nome")

private String nome;


@NotBlank(message="Telefone obrigatório")

private String telefone;


@NotBlank(message="E-mail obrigatório")

@email(message="E-mail inválido")

private String email;


@NotNull(message="Salário obrigatório")

private double salario;

}
```

## Anotação @Valid

Após especificar quais atributos deverão ser validados na **FuncionarioEntity**, é preciso agora definir na **FuncionarioController** quais *endpoints* serão utilizados nas validações por meio da anotação **@Valid**. Por exemplo, você gostaria de aplicar essas validações somente quando ocorrer uma requisição de cadastrar funcionário. Para isso, é necessário apenas aplicar a anotação **@Valid** antes da anotação **@RequestBody** e adicionar a importação do pacote **import jakarta.validation.Valid**:

```
@PostMapping("/adicionar")

public ResponseEntity<FuncionarioEntity> addFuncionario(@Valid @RequestBody Func
```

```
ionarioEntity func) {  
  
    var novoFuncionario = funcionarioService.criarFuncionario(func);  
  
    return new ResponseEntity<>(novoFuncionario, HttpStatus.CREATED);  
  
}
```

## Adicionando um registro de funcionário com dados incompletos

Execute novamente a API no Postman e adicione um funcionário fazendo uma solicitação POST para **localhost:8080/funcionario/adicionar**. Observe que o nome está com apenas com um caractere e que os campos e-mail e CPF estão incorretos.

```
{  
  
  "nome": "a",  
  
  "email": "claudio",  
  
  "cpf": "123",  
  
  "salario": "1254.60",  
  
  "telefone": "51 987654"  
  
}
```

POST localhost:8080/funcionario/adicionar

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {  
2   "nome": "a",  
3   "email": "claudio",  
4   "cpf": "123",  
5   "salario": "1254.60",  
6   "telefone": "51 987654"  
7 }
```

Body Cookies Headers (4) Test Results

Status: 400 Bad Request

Pretty Raw Preview Visualize JSON

```
1 {  
2   "timestamp": "2023-05-02T03:01:51.893+00:00",  
3   "status": 400,  
4   "error": "Bad Request",  
5   "message": "Validation failed for object='funcionarioEntity'. Error count: 3",  
6   "path": "/funcionario/adicionar"  
7 }
```

Figura 13 – Requisição POST no Postman

Fonte: Postman (2023)



Quando a entrada for inválida, será lançada uma exceção com uma mensagem para o cliente, nesse caso, é uma mensagem genérica que mostra três erros na validação de um objeto na classe **FuncionarioEntity**. Além disso, é retornado o *status* com código de requisição malformada 400. Veja na continuação deste conteúdo que é possível também personalizar as respostas.

## Camada *exception*

Para que seu projeto tenha uma melhor organização, inicialmente será criada uma quarta camada denominada **exception**, na qual estarão as classes para tratamento de mensagens das exceções. Será preciso criar o *package* **com.api.senac.exception** na pasta **Source Packages**, com as classes **ValidationHandler** e **ResourceNotFoundException**. A estrutura do projeto ficará da seguinte forma:

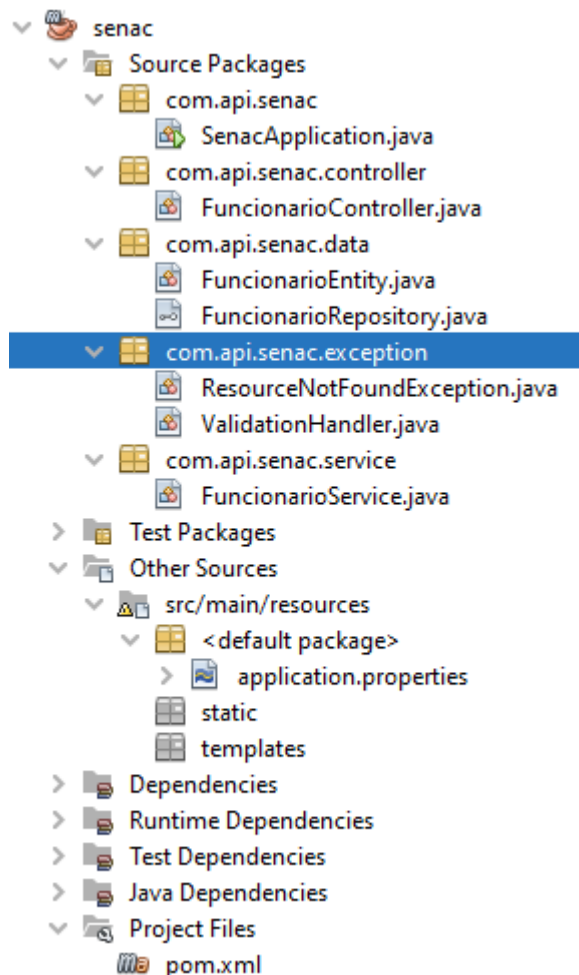


Figura 14 – Estrutura do projeto no NetBeans

Fonte: NetBeans (2023)



## Personalizando mensagens de erro em validações

É uma prática comum criar um tratamento personalizado para retornar mensagens de erros customizadas quando uma requisição é realizada e não são enviados os dados conforme as regras estabelecidas para a API. Por isso, será criada a classe `ValidationHandler` com o seguinte código:

```
package com.api.senac.exception;

import java.util.HashMap;

import java.util.Map;

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpStatus;

import org.springframework.http.HttpStatusCode;

import org.springframework.http.ResponseEntity;

import org.springframework.validation.FieldError;

import org.springframework.web.bind.MethodArgumentNotValidException;

import org.springframework.web.context.request.WebRequest;

import org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;

import org.springframework.web.bind.annotation.ControllerAdvice;

@ControllerAdvice

public class ValidationHandler extends ResponseEntityExceptionHandler {

    @Override

    protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,

        HttpHeaders headers, HttpStatus status, WebRequest request) {

        Map<String, String> errors = new HashMap<>();
```



```
ex.getBindingResult().getAllErrors().forEach((error) -> {  
  
    String fieldName = ((FieldError) error).getField();  
  
    String message = error.getDefaultMessage();  
  
    errors.put(fieldName, message);  
  
});  
  
return new ResponseEntity<Object>(errors, HttpStatus.BAD_REQUEST);  
  
}  
  
}
```

- ◆ A anotação **@ControllerAdvice** é responsável pelo gerenciamento global das exceções.
- ◆ A classe **ValidationHandler** estende a classe **ResponseEntityExceptionHandler**, que é uma classe-base conveniente quando se deseja implementar esse tipo de tratamento de exceção centralizado, pois fornece métodos para lidar com exceções internas do Spring MVC.
- ◆ **@Override** sobrescreve uma tratativa da exceção **MethodArgumentNotValidException** (que é disparado quando uma requisição desrespeita alguma validação aplicada no objeto) para pegar as informações lançadas na exceção e retorná-las dentro do objeto de erros.

Agora será retornado o exemplo anterior. Execute novamente a API no Postman para adicionar um funcionário fazendo uma solicitação POST no endereço **localhost:8080/funcionario/adicionar**.

```
{  
  
    "nome": "a",  
  
    "email": "claudio",  
  
    "cpf": "123",  
  
    "salario": "1254.60",  
  
    "telefone": "51 987654"
```

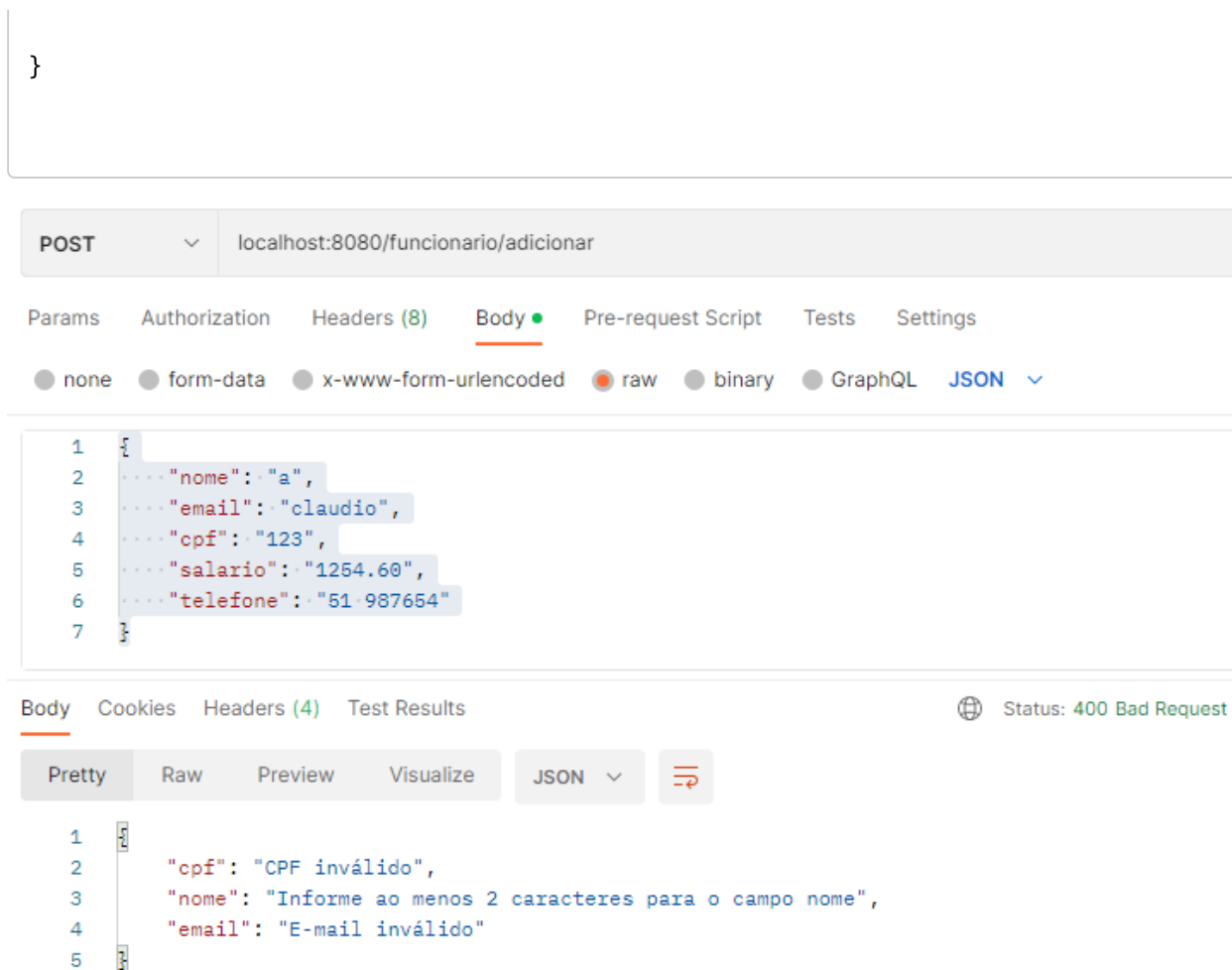


Figura 15 – Requisição POST no Postman

Fonte: Postman (2023)

Observe agora a API retornando as mensagens personalizadas nas anotações da classe **FuncionarioEntity**.

## Gerando mensagens de erro com *exception*

Na camada de serviço, há a classe **FuncionarioService**, que tem o método **getFuncionarioId**, que tem por objetivo buscar um funcionário pelo ID informado. Entretanto, pode ocorrer de ele não ser encontrado ou ser informado um ID inexistente. Como visto anteriormente, a mensagem de erro está sendo produzida de forma geral. Entretanto, você pode criar uma exceção personalizada informando, por exemplo, que o funcionário não foi localizado com o ID informado. Nesse cenário, crie outra exceção **ResourceNotFoundException**:

```
package com.api.senac.exception;
```

```
import org.springframework.http.HttpStatus;

import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND)

public class ResourceNotFoundException extends RuntimeException {

    public ResourceNotFoundException(String mensagem) {

        super(mensagem);

    }

}
```

- ◆ A anotação **@ResponseStatus** é responsável por indicar que a classe retornará com *status* em “http”, de acordo com o informado, nesse caso, *not found*.
- ◆ A classe de exceção **ResourceNotFoundException** estende o comportamento da superclasse **RuntimeException** que gerencia as exceções em tempo de execução
- ◆ Há apenas o construtor que recebe uma mensagem e informa ao construtor da superclasse para notificar quando a exceção ocorrer.

Agora com a configuração da *exception* definida, é preciso alterar o método **getFuncionarioId** para que use o método **orElseThrow** em vez de **orElse**, que será responsável por enviar um objeto da classe **ResourceNotFoundException**, passando a mensagem “Funcionário não localizado”.

```
public FuncionarioEntity getFuncionarioId(Integer funcId) {
    return funcionarioRepository.findById(funcId).orElseThrow(() -> new Resource
NotFoundException("Funcionário não encontrado " + funcId));
}
```

Agora você testará os ajustes realizados. Por isso, faça uma solicitação GET para o endereço **localhost:8080/funcionario/pesquisa/99999**, passando o ID do funcionário **99999** no final da URL.

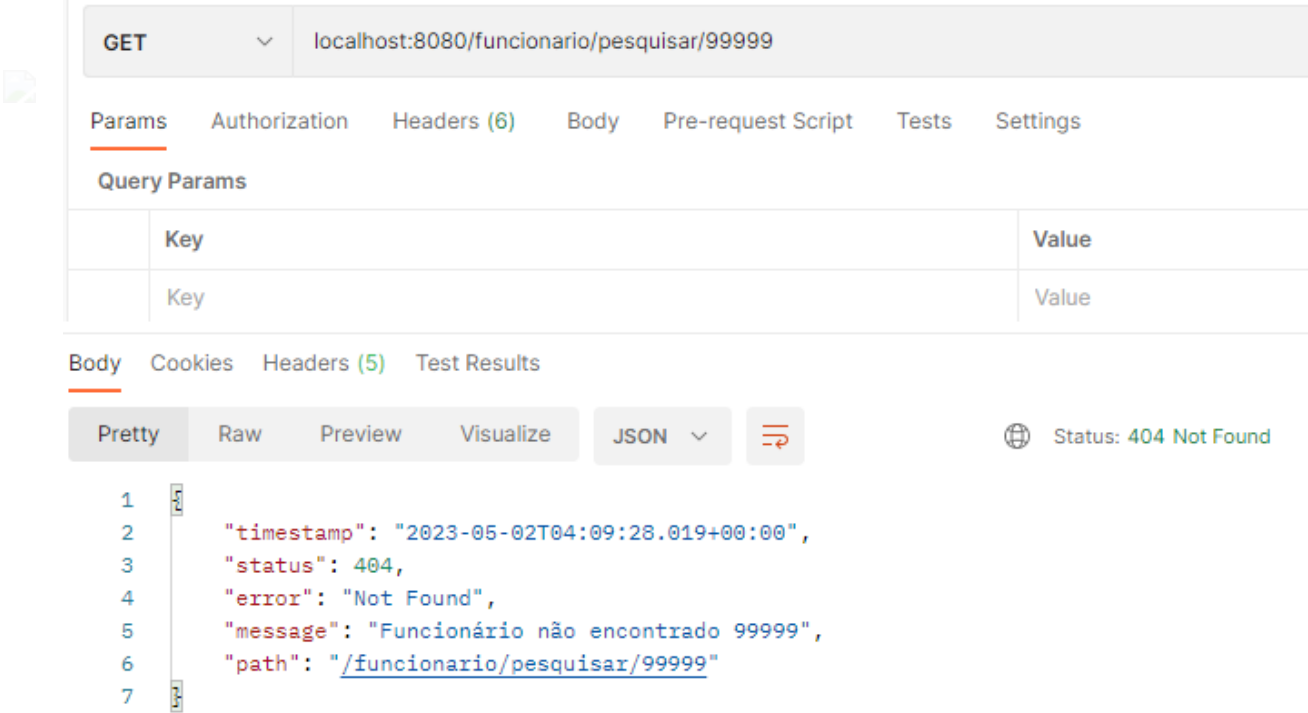


Figura 16 – Requisição GET no Postman

Fonte: Postman (2023)

Observe agora a API retornando a mensagem personalizada `Funcionário não encontrado 99999`, que foi definida no método `getFuncionarioId`.

Incremente o projeto de livros do desafio anterior, criando validações para o nome e o autor como campos requeridos.

## Criando consultas avançadas

O **Spring Data JPA** disponibiliza métodos para criar *queries* a partir do nome de métodos personalizados. Esse é um recurso extremamente flexível e poderoso. No entanto, para gerar essas consultas, existem algumas regras que devem ser seguidas por meio de palavras-chaves e atributos que indicam quais consultas serão utilizadas.

Retorno	Palavra de busca	Atributo	Operador
FuncionarioEntity	findBy	Nome	And, Or
List>FuncionarioEntity>		<i>E-mail</i>	StartingWith, EndingWith, Containing
		Salário	OrderBy
			Asc, Desc

Para melhor entender como criar métodos personalizados de consultas, veja alguns exemplos com maior detalhamento.

Exemplo 1: Localizar um funcionário pelo seu nome.

Para isso, defina o retorno do método, em seguida, a palavra para sinalizar a busca, por fim, o nome do atributo que será utilizado. Essa pesquisa somente retornará caso o nome informado seja igual ao nome armazenado no banco de dados.

```
FuncionarioEntity findByNome(String nome);
```

Exemplo 2: Localizar funcionários pelo seu nome ou e-mail em uma mesma pesquisa.

Defina agora o tipo de retorno do método, como uma lista, em seguida, a palavra-chave “findBy”, após especificar o nome do atributo do primeiro atributo “nome” a ser pesquisado, seguido do operador “Or” e, por fim, o último atributo “email”.

```
List<FuncionarioEntity> findByNomeOrEmail(String nome, String email);
```

Exemplo 3: Encontrar nomes de funcionários que começam com um valor.

Use o operador “StartingWith”, que em SQL seria algo como “WHERE nome LIKE ‘%value’”.

```
List<FuncionarioEntity> findByNomeStartingWith(String nome)
```

Exemplo 4: Pesquisar nomes de funcionários que terminem com um valor.

Use o operador “EndingWith”, que em SQL seria algo como “WHERE nome LIKE ‘value%’”.



```
List<FuncionarioEntity> findByNameEndingWith(String nome);
```

Exemplo 5: Descobrindo quais nomes de funcionários contêm um valor.

Use o operador “Containing”, que em SQL seria algo como “WHERE nome LIKE ‘%value%’”.

```
List<FuncionarioEntity> findByNameContaining(String nome);
```

Exemplo 6: Gerar uma listagem de funcionários ordenados pelo seu nome de forma ascendente.

Não use um filtro, e sim um retorno de uma listagem ordenada. Para isso, continue usando palavra-chave “findBy”, seguida do operador “OrderBy”, acrescido dos atributos que estão sendo ordenados, nesse caso, “nome”, e o tipo de ordenação “Asc”.

```
List<FuncionarioEntity> findByOrderByNomeAsc();
```


Exemplo 7: Gerar a mesma listagem de funcionários, porém de forma decrescente do nome.

Basta usar o operador “Desc”.

```
List<FuncionarioEntity> findByOrderByNomeDesc();
```

Por fim, veja como ficou a interface “FuncionarioRepository”.

```
package com.api.senac.data;
```



```
import java.util.List;

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

import org.springframework.stereotype.Repository;

@Repository

public interface FuncionarioRepository extends JpaRepository<FuncionarioEntity, Integer> {

    FuncionarioEntity findByName(String nome);

    List<FuncionarioEntity> findByNameOrEmail(String nome, String email);

    List<FuncionarioEntity> findByNameStartingWith(String nome);

    List<FuncionarioEntity> findByNameEndingWith(String nome);

    List<FuncionarioEntity> findByNameContaining(String nome);

    List<FuncionarioEntity> findByNameOrderByNomeAsc();

    List<FuncionarioEntity> findByNameOrderByNomeDesc();

}
```

## Testando os métodos de consulta avançada

Antes de iniciar os testes na API de Funcionários, é importante ressaltar que os métodos criados serão utilizados de acordo com a necessidade na camada **Service**. Para exemplificar, será criado um método na classe **FuncionarioService**, chamando **getFuncionarioPorNome**, que deverá receber um nome e retornar uma lista de funcionários que tenham, em qualquer parte dos nomes dos funcionários, o valor informado. Será usado o método de pesquisa **findByNameContaining**.

```
public List<FuncionarioEntity> getFuncionarioPorNome(String nome) {

    return funcionarioRepository.findByNameContaining(nome);

}
```



O próximo passo agora é criar na camada **Controller** uma rota ou *endpoint* para solicitar uma pesquisa, chamada “pesquisar-nome”, passando o nome em *string*, que chamará o serviço **getFuncionarioPorNome**, pelo nome e retornando uma lista com os nomes de funcionários encontrados.

```
@GetMapping("/pesquisar-nome/{nome}")

public ResponseEntity<List> getPesquisarPorNomeFuncionarios(@PathVariable String
nome) {

    List<FuncionarioEntity> funcionarios = funcionarioService.getFuncionarioPorN
ome(nome);

    return new ResponseEntity<>(funcionarios, HttpStatus.OK);

}
```

No Postman, agora teste a pesquisa criada. Para isso, faça uma solicitação GET para o endereço **localhost:8080/funcionario/pesquisa-nome/Senac**, passando parte de um nome do funcionário **Senac** no final da URL.



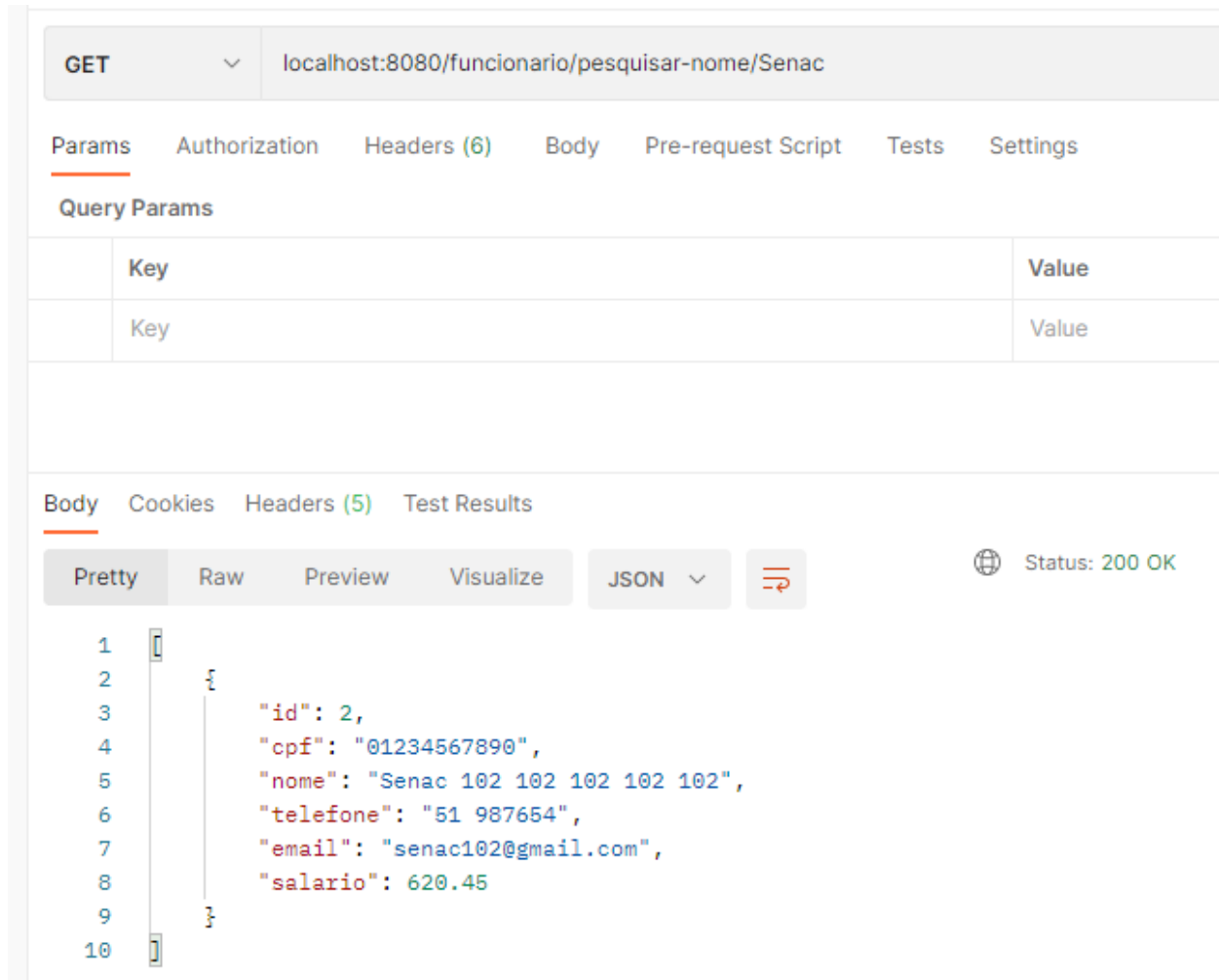


Figura 17 – Requisição GET no Postman

Fonte: Postman (2023)

## Utilizando a anotação @Query

Uma das formas que o **Spring Data** provê para executar uma *query* é por meio da anotação `@Query`, que só pode ser utilizada nas interfaces *repository*. Neste projeto, é **FuncionarioRepository**. A chamada dos métodos anotados por `@Query` acionará a execução da instrução encontrada. Com isso, é possível executar SQL nativo.

**@Query com SQL Nativo:** para utilizar instruções nativas em SQL, é necessário indicar com `true` o atributo `nativeQuery`. Para exemplificar, será criada, em **FuncionarioRepository**, uma consulta `findAllMaiorSalario`, que retornará uma lista dos funcionários com maior salário. Para isso, use a anotação `@Query` e defina na propriedade “value” a query em SQL “select MAX(salario) from funcionário”.



```
@Query(value="select MAX(salario) from Funcionario", nativeQuery=true)  
List<FuncionarioEntity> findAllMaiorSalario();
```

Veja como ficou a classe **FuncionarioRepository**.

```
package com.api.senac.data;  
  
import java.util.List;  
  
import org.springframework.data.jpa.repository.JpaRepository;  
  
import org.springframework.data.jpa.repository.Query;  
  
import org.springframework.stereotype.Repository;  
  
@Repository  
  
public interface FuncionarioRepository extends JpaRepository<FuncionarioEntity, Integer> {  
  
    FuncionarioEntity findByNome(String nome);  
  
    List<FuncionarioEntity> findByNomeOrEmail(String nome, String email);  
  
    List<FuncionarioEntity> findByNomeStartingWith(String nome);  
  
    List<FuncionarioEntity> findByNomeEndingWith(String nome);  
  
    List<FuncionarioEntity> findByNomeContaining(String nome);  
  
    List<FuncionarioEntity> findByOrderByNomeAsc();  
  
    List<FuncionarioEntity> findByOrderByNomeDesc();  
  
    @Query(value="select * from Funcionario", nativeQuery=true)  
  
    List<FuncionarioEntity> findAllMaiorSalario();  
  
}
```

Na classe de serviço **FuncionarioService**, crie o método **getMaioresSalarios**, que retornará uma lista de funcionários com maiores salários, para isso, use o método de pesquisa com SQL nativo **findAllMaiorSalario**, criado na classe **FuncionarioRepository**.



```
public List<FuncionarioEntity> listarTodosFuncionarios() {  
    return funcionarioRepository.findAll();  
}
```

Por fim, é preciso criar na camada **Controller** uma rota ou endpoint para solicitar uma pesquisa, chamada **maiores-salarios**, que chamará o serviço **getMaioresSalarios**, retornando uma lista com os nomes dos funcionários com maiores salários.

```
@GetMapping("/maiores-salarios")  
  
public ResponseEntity<List<FuncionarioEntity>> getMaioresSalarios() {  
    List<FuncionarioEntity> funcionarios = funcionarioService.getMaioresSalarios()  
();  
    return new ResponseEntity<>(funcionarios, HttpStatus.OK);  
}
```

No Postman, teste a pesquisa criada. Para isso, faça uma solicitação GET para o endereço **localhost:8080/funcionario/maiores-salarios**.

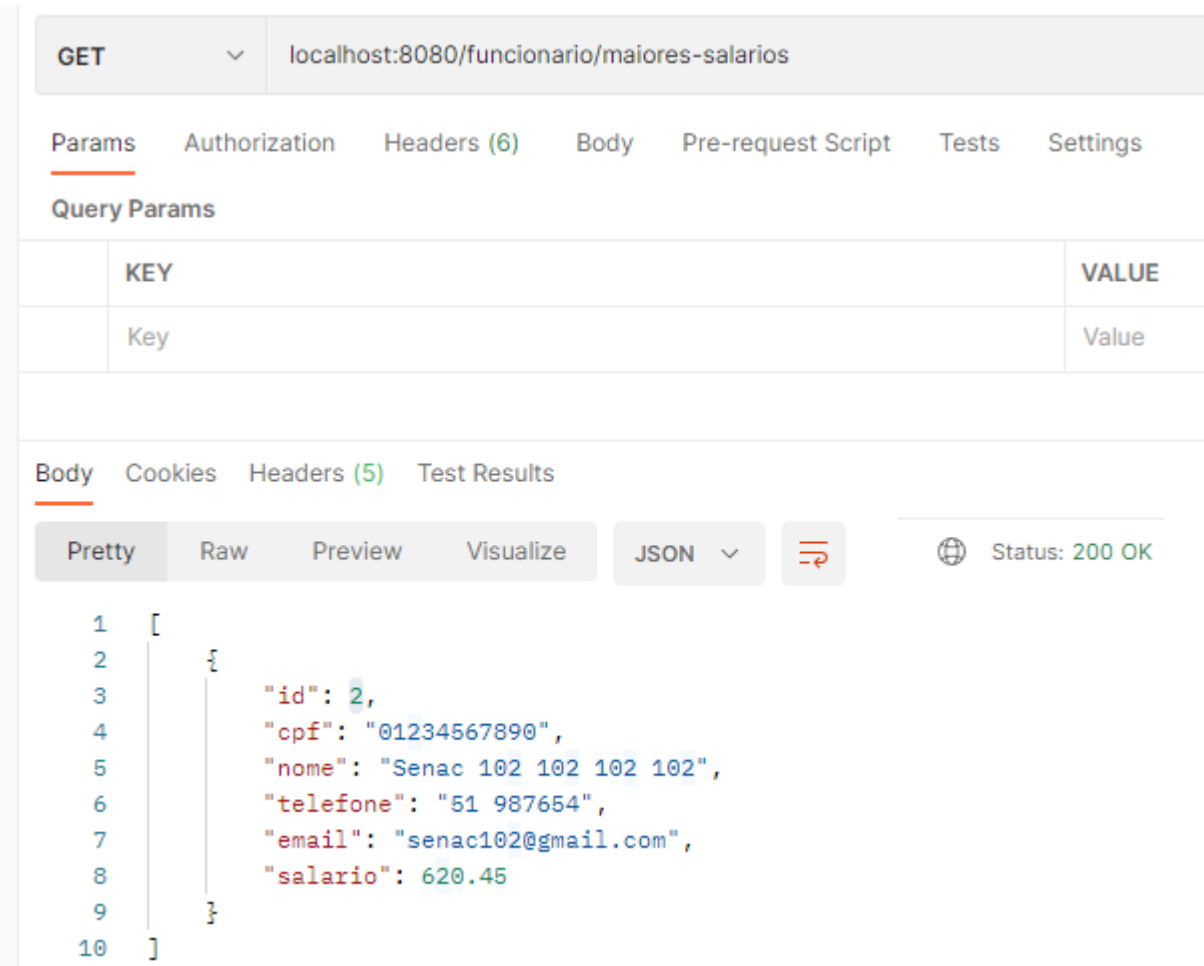


Figura 18 – Requisição GET no Postman

Fonte: Postman (2023)

## Criando uma aplicação *web* com Spring MVC

Agora que você tem a API criada, pode utilizar todos os serviços criados em uma aplicação *web* utilizando o Spring MVC. Essa aplicação será um CRUD de funcionários, em que você irá listar, cadastrar, alterar e excluir. Para iniciar, é necessário compreender que a construção de um projeto Spring MVC pode ser realizada a partir do Spring Boot, mas, para isso, serão necessários três passos:

Adicionar a dependência da biblioteca do **Thymeleaf** no arquivo “`pow.xml`”.

Criar uma classe *controller* específica com rotas para o Spring MVC, usando a anotação `@Controller`.

Criar uma classe *controller* específica com rotas para o Spring MVC, usando a anotação `@Controller`.

A seguir, serão aprofundados cada um desses passos.



## Inserir dependência da biblioteca Thymeleaf no arquivo “pom.xml”

Para incluir uma dependência na biblioteca Thymeleaf de um projeto já criado, use o gerenciador de dependências do Maven.

- ◆ Abra o arquivo pom.xml no NetBeans – ele fica dentro de Project Files na estrutura de arquivos. Esse arquivo é responsável por gerenciar as dependências do seu projeto.

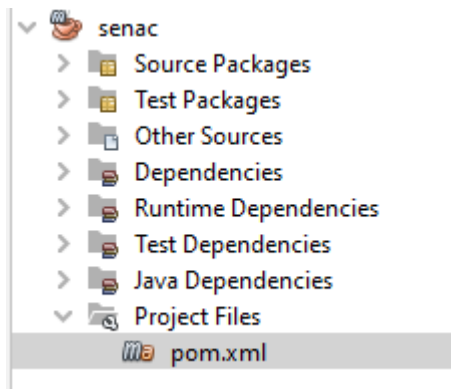


Figura 19 – Arquivo pom.xml

Fonte: NetBeans (2023)

- ◆ Localize a seção <dependencies> do arquivo “pom.xml” – ela estará aproximadamente na linha 19. É aqui que você precisa adicionar as novas dependências. Nesse caso, use o Thymeleaf na versão 3.0.6, com o seguinte código em XML:

```
<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-thymeleaf</artifactId>
>

    <version>3.0.6</version>

</dependency>
```



- ◆ Cole o código XML da dependência dentro da seção <dependencies> do arquivo “pom.xml”. Certifique-se de que a dependência esteja dentro das tags <dependency> e </dependency>. Tenha cuidado para não apagar as dependências que já estão lá. Suas dependências devem ficar assim:

Arraste para ver o restante do código

```
<?xml> version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    https://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion> 4.0.0 </modelVersion>

  <parent>

    <groupId> org.springframework.boot </groupId>

    <artifactId> spring-boot-starter-parent </artifactId>

    <version> 3.0.6 </version>

    <relativePath/> <!-- lookup parent from repository -->

  </parent>

  <groupId> com.api </groupId>

  <artifactId> senac </artifactId>

  <version> 0.0.1-SNAPSHOT </version>

  <name> senac </name>

  <description> API Rest de Projeto de Funcionários </
```



description>

<properties>

<java.version> 17 </java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot </groupId>

<artifactId>spring-boot-starter-data-jpa </ar  
tifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot </groupId>

<artifactId> spring-boot-starter-validation  
</artifactId>

</dependency>

<dependency>

<groupId> org.springframework.boot </groupId>

<artifactId>spring-boot-starter-web </artifa  
ctId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</



artifactId>

```
        <version>3.0.6</version>
    </dependency>
```

```
<dependency>
```

```
    <groupId> com.mysql </groupId>
```

```
    <artifactId> mysql-connector-j </artifactId>
```

>

```
    <scope> runtime </scope>
```

```
</dependency>
```

```
<dependency>
```

```
    <groupId> org.projectlombok </groupId>
```

```
    <artifactId> lombok </artifactId>
```

```
    <optional> true < /optional>
```

```
</dependency>
```

```
<dependency>
```

```
    <groupId> org.springframework.boot </groupI
```

d>

```
    <artifactId> spring-boot-starter-test </art
```

ifactId>

```
    <scope> test </scope>
```

```
</dependency>
```

```
</dependencies>
```

```
<build>
```

```
    <plugins>
```





```
<plugin>

    <groupId> org.springframework.boot </gr
oupId>

    <artifactId> spring-boot-maven-plugin
</artifactId>

    <configuration>

        <excludes>

            <exclude>

                <groupId> org.projectlombok
</groupId>

                <artifactId> lombok </artifa
ctId>

            </exclude>

        </excludes>

    </configuration>

</plugin>

</plugins>

</build>

</project>
```

- ◆ Salve o arquivo pom.xml. O Maven agora baixará automaticamente a nova dependência para o seu projeto.
- ◆ Para atualizar o projeto no NetBeans com as novas dependências, clique com o botão direito do *mouse* no nome do projeto e selecione a opção “Clean and Build” (Limpar e Construir).

◆ Pronto, agora você pode usar a nova dependência em seu projeto Spring.



## Criando um controller específico para o Spring MVC

Crie uma classe **FuncController** em **com.api.senac.controller** com o código a seguir. Observe que foi usada uma anotação especial **@Controller**, proveniente do pacote “org.springframework.stereotype” do Spring. Essa anotação é necessária para que o sistema busque nessa classe qual método deve ser executado a partir da URL. Em seguida, conecte automaticamente a classe **FuncionarioService** em no controlador para usar os métodos declarados implementados nele. Isso é injeção de dependência pela anotação **@Autowired**. Além disso, há cinco métodos que serão detalhados na sequência do conteúdo durante a criação do CRUD de funcionários.

```
package com.api.senac.controller;
import com.api.senac.data.FuncionarioEntity;
import com.api.senac.service.FuncionarioService;
import jakarta.validation.Valid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;

@Controller

public class FuncController {
    @Autowired
    FuncionarioService funcionarioService;
    @GetMapping("/")
    public String viewHomePage(Model model) {
        model.addAttribute("listarFuncionarios", funcionarioService.listarTodosFuncionarios());
        return "index";
    }

    @GetMapping("/deletarFuncionario/{id}")
    public String deletarFuncionario(@PathVariable(value = "id") Integer id) {
        funcionarioService.deletarFuncionario(id);
        return "redirect:/";
    }

    @GetMapping("/criarFuncionarioForm")
    public String criarFuncionarioForm(Model model) {
```

```
FuncionarioEntity func = new FuncionarioEntity();
model.addAttribute("funcionario", func);
return "inserir";
}

@PostMapping("/salvarFuncionario")

public String salvarFuncionario(@Valid @ModelAttribute("funcionario") FuncionarioEntity func, BindingResult result) {

    if (result.hasErrors()) {

        return "inserir";

    }

    if (func.getId()==null) {

        funcionarioService.criarFuncionario(func);

    } else {

        funcionarioService.atualizarFuncionario(func.getId(), func);

    }

    return "redirect:/";

}

@GetMapping("/atualizarFuncionarioForm/{id}")

public String atualizarFuncionarioForm(@PathVariable(value = "id") Integer id, Model model) {

    FuncionarioEntity func = funcionarioService.getFuncionarioId(id);

    model.addAttribute("funcionario", func);

    return "atualizar";

}
}
```

## Criando um CRUD de funcionários

No NetBeans, na pasta “Other Sources/src/main/resources/templates”, clique com o botão direito do *mouse* e selecione New > HTML File e crie três arquivos com nomes “index.html”, “atualizar.html” e “inserir.html”. Veja como ficará a estrutura do projeto a seguir:

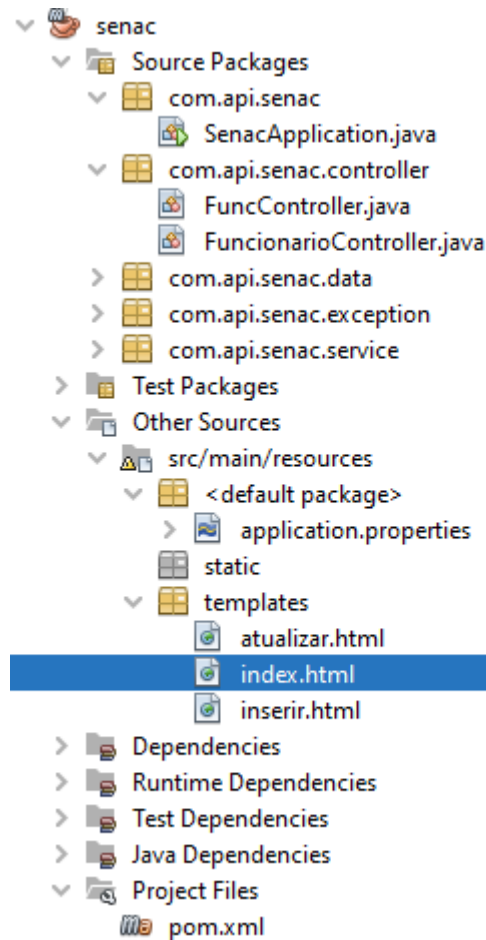


Figura 20 – Estrutura da camada View

Fonte: NetBeans (2023)

**Inserir funcionário:** será chamada uma página com formulário de cadastro de funcionário e estudado como realizar a postagem dos valores digitados pelo usuário para o *back-end*. Primeiramente, analise na classe “FuncController” o método criado anteriormente **criarFuncionarioForm()**, que responderá pela URL “/criarFuncionarioForm”.

```
@GetMapping("/criarFuncionarioForm")

public String criarFuncionarioForm(Model model) {

    FuncionarioEntity func = new FuncionarioEntity();

    model.addAttribute("funcionario", func);
```

```
return "inserir";
```

```
}
```

O método recebe um parâmetro do tipo “Model”, que será usado para transportar dados para a camada de visão. Nesse caso, inclua em “model” o atributo “funcionário” com o objeto “func” que foi criado. Por fim, retorne para a visão “inserir”. Agora é preciso um formulário em “html” para que o usuário possa realizar o cadastro de um funcionário. Para isso, insira no arquivo “inserir.html” o seguinte bloco de código:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Gerenciamento de Funcionários</title>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO" crossorigin="anonymous">
  </head>
  <body>
    <div class="container">
      <h1>Gerenciamento de Funcionários</h1>
      <hr>
      <h2>Adicionar</h2>
      <form action="#" th:action="@{/salvarFuncionario}" th:object="${funcionario}" method="POST">
        <input type="text" th:field="*{nome}" placeholder="Insira o nome" class="form-control mt-4 col-4">
        <span class="text-danger" th:if="${#fields.hasErrors('nome')}" th:errors="*{nome}"></span>
        <input type="text" th:field="*{cpf}" placeholder="Insira o CPF" class="form-control mt-4 col-4">
        <span class="text-danger" th:if="${#fields.hasErrors('cpf')}" th:errors="*{cpf}"></span>
        <input type="text" th:field="*{email}" placeholder="Insira o Email" class="form-control mt-4 col-4">
        <span class="text-danger" th:if="${#fields.hasErrors('email')}" th:errors="*{email}"></span>
        <input type="text" th:field="*{telefone}" placeholder="Insira o Telefone" class="form-control mt-4 col-4">
        <span class="text-danger" th:if="${#fields.hasErrors('telefone')}" th:errors="*{telefone}"></span>
        <input type="number" th:field="*{salario}" placeholder="Insira o Salário" class="form-control mt-4 col-4">
        <span class="text-danger" th:if="${#fields.hasErrors('salario')}" th:errors="*{salario}"></span>
        <button type="submit" class="btn btn-info mt-4"> Salvar</button>
      </form>
    </div>
  </body>
</html>
```

```
        <a th:href="@{/}"> Retornar</a>
    </div>
</body>
</html>
```

A página conta com um formulário com campos simples. Veja alguns dos atributos Thymeleaf presentes:

- ◆ **th:action="@{/salvarFuncionario}"**: indica que a ação do formulário irá para a URL terminada em “/salvarfuncionario”.
- ◆ **th:object="\${funcionario}" data-binding**: com o objeto do tipo Funcionário fornecido por “Model”.
- ◆ **th:field="\*{nome}"**: associa um elemento de *input* com uma propriedade do objeto Funcionário associado ao formulário.

No método “salvarFuncionario”, da classe “FuncController”, existem características importantes.

```
@PostMapping("/salvarFuncionario")

public String salvarFuncionario(@Valid @ModelAttribute("funcionario") FuncionarioEntity func, BindingResult result) {

    if (result.hasErrors()) {
        return "inserir";
    }

    if (func.getId()==null) {
        funcionarioService.criarFuncionario(func);
    } else {
        funcionarioService.atualizarFuncionario(func.getId(), func);
    }

    return "redirect:/";
}
```

A anotação `@PostMapping` indica que o método é acessível apenas por requisições do tipo POST e responderá à URL terminada em “/salvarFuncionario” (a mesma do atributo “action” no formulário). Já `@ModelAttribute` indica que o objeto

“func” presente no parâmetro do método conterá as informações que o usuário digitou no formulário ao realizar a postagem. Outra característica são as validações por meio da anotação `@Valid`, que normalmente é usada antes do objeto a ser validado. Lembre-se de que dentro da classe “FuncionarioEntity” existem anotações que validam os dados enviados. Suponha que, na requisição, o campo “nome” tenha sido informado em branco ou com menos de dois caracteres.

```
@Size(min=2, message="Informe ao menos 2 caracteres para o campo nome")
private String nome;
```

Quando alguma restrição é violada, os erros de violação serão adicionados ao objeto “BindingResult”, presente nos parâmetros do método. Em seguida, é possível verificar se há erros no objeto “BindingResult”. O método “hasErrors()” é usado para verificar se há algum erro no objeto “BindingResult” após o processo de validação ter sido acionado. Caso ocorra, redirecione para a visão “inserir.html” para que as mensagens de validação de erros sejam exibidas ao lado de cada campo, como no caso do nome, por meio da marcação do Thymeleaf.

```
<input type= "text" th:field=
"*{nome}" placeholder=
"Insira o nome" class=
"form-control mt-4 col-4">
<span class=
"text-danger" th:if="${#fields.hasErrors('nome')}}" th:errors="*{nome}"></span>
```

- ◆ **th:if = "\${#fields.hasErrors('nome')}}"**: habilita o componente em HTML, por ter ocorrido erro com atributo “nome”.
- ◆ **th:errors="\*{nome}"**: exibe o erro gerado na violação.

Veja o erro sendo exibido na visão “inserir.html”:

# Gerenciamento de Funcionários



## Adicionar

Informe ao menos 2 caracteres para o campo nome

[Retornar](#)

Figura 21 – Formulário de adicionar funcionário

Fonte: Senac EAD (2023)

Esse método pode ser usado tanto para criar um novo funcionário como para atualizar um existente. Por isso, é necessário criar uma condição, no exemplo, está sendo validado se o ID do funcionário está sendo enviado. Se for enviado nulo, será chamado o serviço de “criarFuncionario”, caso contrário, o “atualizarFuncionario”, que são serviços já definidos anteriormente durante a criação da API. Por fim, é feito um redirecionamento para a raiz do projeto “viewHomePage”, que é o método para exibir uma lista de funcionários na visão “index.html”.

**Atualizar funcionário:** analise na classe “FuncController” o método criado anteriormente **atualizarFuncionarioForm**, que responderá pela URL “/atualizarFuncionarioForm/{id}”. Essa rota deverá receber dois parâmetros: o ID de um funcionário pelo PathVariable e o objeto Model. Depois, deve ser incluída, em um mapa



de informações, uma entrada pelo método “addAttribute()” da classe Model, em que serão informados uma chave (“funcionario”) e o valor (o objeto func que foi localizado pelo ID informado no método getFuncionarioId). Por fim, retorne para a visão “atualizar”.

```
@GetMapping("/atualizarFuncionarioForm/{id}")
public String atualizarFuncionarioForm(@PathVariable(value = "id") Integer id, Model model) {
    FuncionarioEntity func = funcionarioService.getFuncionarioId(id);
    model.addAttribute("funcionario", func);
    return "atualizar";
}
```

Agora é preciso um formulário para que o usuário possa realizar a atualização de um funcionário. Para isso, insira no arquivo “atualizar.html” o seguinte bloco de código:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Gerenciamento de Funcionários
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCW98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPKFOJwJ8ERdknLPMO" crossorigin="anonymous">
  </head>
  <body>
    <div class="container">
      <h1>Gerenciamento de Funcionários
      <hr>
      <h2>Atualizar
      <form action="#" th:action="@{/salvarFuncionario}" th:object="${funcionario}" method="POST">
        <input type="text" th:field="*{nome}" placeholder="Insira o nome" class="form-control mb-4 col-4">
        <input type="text" th:field="*{cpf}" placeholder="Insira o CPF" class="form-control mb-4 col-4">
        <input type="text" th:field="*{email}" placeholder="Insira o Email" class="form-control mb-4 col-4">
        <input type="text" th:field="*{telefone}" placeholder="Insira o Telefone" class="form-control mb-4 col-4">
        <input type="text" th:field="*{salario}" placeholder="Insira o Salário" class="form-control mb-4 col-4">
        <input type="hidden" th:field="*{id}" />
        <button type="submit" class="btn btn-info col-2"> Atualizar</button>
      </form>
      <hr>
      <a th:href="@{/}"> Retornar</a>
```



```
</div>
</body>
</html>
```

Observe que a visão de atualização de funcionário tem as mesmas características da visão de inserir. Entretanto é necessário informar o ID do funcionário pela marcação do Thymeleaf `th:field="**{id}"` e defini-la como oculta pela propriedade `"hidden"`, da seguinte forma:

```
<input type="hidden" th:field="**{id}" />
```

No restante, requisição e validações serão iguais.

# Gerenciamento de Funcionários

## Atualizar

Senac 102 102 102 102

01234567890

senac102@gmail.com

51 987654

620.45

Atualizar

[Retornar](#)

Figura 22 – Formulário de atualizar funcionário

Fonte: Senac EAD (2023)

**Listagem de funcionários:** em “FuncController”, há o método criado anteriormente **viewHomePage**. A presença da anotação “@GetMapping” é que fará, quando o usuário digitar `http://localhost:8080/`, que uma lista de funcionários seja exibida na visão “index”. Observe que o método tem como parâmetro um objeto do tipo `Model`, que incluirá em um mapa de informações uma entrada pelo método “`addAttribute()`”, em que é informada uma chave “`listarFuncionarios`” e como valor a lista de funcionários obtida pelo método `listarTodosFuncionarios()` da camada de serviço `funcionarioService`. Por fim, o redirecionamento para a visão “`index.html`”.

```
@GetMapping("/")

public String viewHomePage(Model model) {

    model.addAttribute("listarFuncionarios", funcionarioService.listarTodosFuncionarios());

    return "index";

}
```

Agora é necessária uma listagem de funcionários com as operações de alterar e excluir, além do botão de adicionar um novo funcionário. Para isso, insira no arquivo “`index.html`” o seguinte bloco de código:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="utf-8">
        <title>Gerenciamento de Funcionários
        <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPM0" crossorigin="anonymous">
    </head>
    <body>
        <div class="container my-2">
            <h1>Lista de Funcionários
            <a th:href="@{/criarFuncionarioForm}" class="btn btn-primary btn-sm mb-3"> Adicionar </a>
            <table border="1" class="table table-striped table-responsive-md">
                <thead>
                    <tr>
                        <th>ID
                        <th>Nome
                        <th>CPF
```

```

        <th>Email
        <th>Telefone
        <th>Salário
        <th>Operações
    </tr>
</thead>
<tbody>
    <tr th:each="func : ${listarFuncionarios}">
        <td th:text="${func.id}">
        <td th:text="${func.nome}">
        <td th:text="${func.cpf}">
        <td th:text="${func.email}">
        <td th:text="${func.telefone}">
        <td th:text="${func.salario}">
        <td> <a th:href="@{/atualizarFuncionarioForm/{id}(id=${func.
id}})" class="btn btn-primary">Atualisar</a>
            <a th:href="@{/deletarFuncionario/{id}(id=${func.id}})"
class="btn btn-danger">Excluir</a>
        </td>
    </tr>
</tbody>
</table>
</div>
</body>
</html>

```

A primeira alteração acontece no tag html, que recebe um atributo para reconhecer os atributos especiais de Thymeleaf. A segunda alteração é que, para chamar uma rota para cadastrar, será usada a marcação **th:href**. Para cadastrar, será usado: `th:href="@{/criarFuncionarioForm}"`.

```

        <a th:href="@{/criarFuncionarioForm}" class="btn btn-primary btn-sm mb
-3"> Adicionar </a>

```

E como os dados da listagem de funcionários serão exibidos? A marcação **th:each** é usada para iterar a lista de funcionários. O **func : \${listarFuncionarios}** significa o valor para cada atributo, ou seja, “para cada elemento no resultado da lista **\${listarFuncionarios}**, repita esse fragmento da configuração do modelo desse elemento em uma variável chamada **func**”. O próximo passo agora é exibir cada atributo do objeto por meio do marcador **th:text="\${func.id}"**



Por fim, agora é possível fazer a chamada de rotas usando o marcador **th:ref** e passando parâmetros usando **{func.id}**.

Sintaxe: `th:href="@{/rota/parâmetro(parâmetro=${obj.atributo})}"`

**Exemplo para chamar uma requisição de atualização de funcionário:** aqui há a rota “atualizarFuncionarioForm” da classe “FuncController”, que recebe como parâmetro o ID do funcionário, busca as informações e retorna para a visão “atualizar”. Esse método foi detalhado na seção “Atualizar Funcionário”.

```
<a th:href="@{/atualizarFuncionarioForm/{id}(id=${func.id})}" class="btn btn-primary">Atualizar</a>
```

**Exemplo para excluir um funcionário:** no código a seguir, há o bloco responsável por excluir um funcionário, por meio do ID do funcionário.

```
<a th:href="@{/deletarFuncionario/{id}(id=${func.id})}" class="btn btn-danger">Excluir</a>
```

No momento em que a rota é requisitada, o método “deletarFuncionario” presente na classe “FuncController” recebe como parâmetro o ID do funcionário. Logo a seguir, é chamada a classe de serviço com o método de “deletarFuncionario”, no qual o objetivo é excluí-lo do banco de dados. Por fim, é feito um redirecionamento para a raiz do projeto “viewHomePage”, que é o método para carregar e exibir uma lista de funcionários na visão “index.html”.

```
@GetMapping("/deletarFuncionario/{id}")

public String deletarFuncionario(@PathVariable(value = "id") Integer id) {

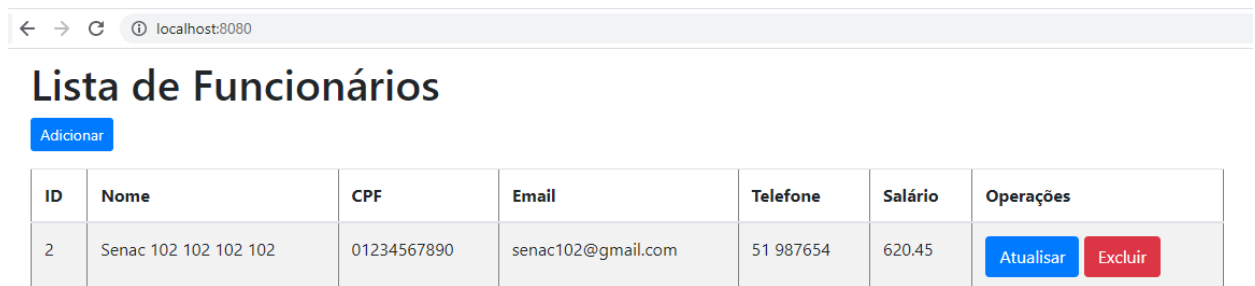
    funcionarioService.deletarFuncionario(id);

    return "redirect:/";

}
```



**Testando a aplicação:** para testar, clique com o botão direito do *mouse* sobre a classe “SenacApplication.java” e escolha “Run File”. A aba *output* do NetBeans deve mostrar um *log* na aba *output*, indicando que o servidor interno Tomcat do Spring está rodando. Note ainda que o processo não encerra, segue rodando, mas você pode interromper clicando no botão “stop” na lateral esquerda dessa aba. Agora com servidor rodando, você pode, enfim, testar por completo as funcionalidades, primeiro acessando “http://localhost:8080/” via navegador de internet.



ID	Nome	CPF	Email	Telefone	Salário	Operações
2	Senac 102 102 102 102	01234567890	senac102@gmail.com	51 987654	620.45	<button>Atualisar</button> <button>Excluir</button>

Figura 23 – Lista de Funcionários

Fonte: Senac EAD (2023)

Crie uma aplicação *web* Spring MVC que consuma os serviços REST de livros desenvolvidos nos desafios anteriores.

## Encerramento

Neste conteúdo, você aprendeu como criar uma API com Spring Boot, integrando ao banco de dados com Spring Data JPA, e criar consultas avançadas, além do desenvolvimento de uma aplicação *web* utilizando o Spring MVC.