



Digital Business Enablement

Prof. Gilberto Alexandre das Neves
profgilberto.neves@fiap.com.br

Spring Framework

Spring é um framework para desenvolvimento de aplicações em Java, criado em meados de 2002 por Rod Johnson, que se tornou bastante popular e adotado ao redor do mundo devido a sua simplicidade e facilidade de integração com outras tecnologias.

O framework foi desenvolvido de maneira **modular**. Em cada aplicação podemos adicionar apenas os módulos que fizerem sentido.

Existem diversos módulos no Spring, cada um com uma finalidade distinta, como por exemplo:

- o módulo **MVC**, para desenvolvimento de aplicações Web e API's Rest;
- o módulo **Security**, para lidar com controle de autenticação e autorização da aplicação;
- e o módulo **Transactions**, para gerenciar o controle transacional.

Adicionando dependências

Adicionando dependências

Vamos fazer a **validação** das informações e a **persistência** no banco de dados.

Precisaremos adicionar novos módulos do *Spring*. Temos que adicionar o driver do banco de dados, o módulo do Spring Data e o módulo de validação.

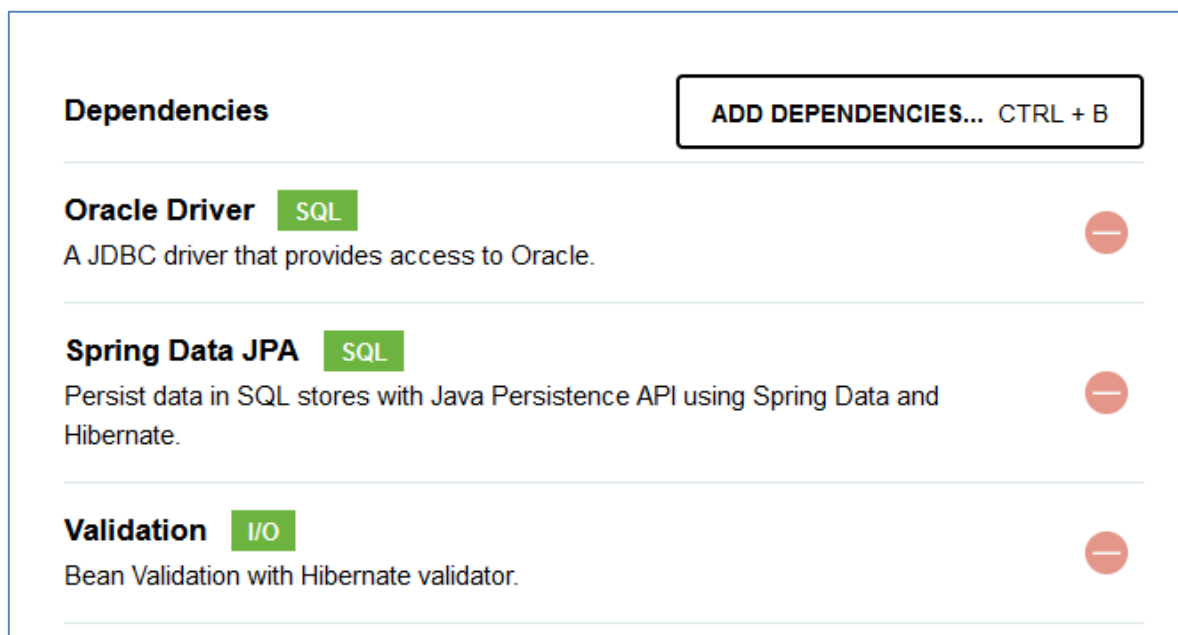
A maneira mais fácil de realizar essa tarefa de adicionar novas dependências em um projeto em andamento é utilizar o *Spring Initializr*, acesse o site: <https://start.spring.io/>

Do lado esquerda da tela confira se é um projeto **Maven**, se a linguagem utilizada é **Java**, versão do **Spring Boot** (3.0.2) e versão do **JDK** do Java (17). Não é necessário se preocupar com o nome do projeto, pacotes, etc (*Project Metadata*).

Adicionando dependências

Do lado direito da tela clique no botão **Add Dependencies...** e busque pelos módulos necessários como indicado (lembre-se, utilize o **Ctrl** ao selecionar as dependências):

- Busque pelo **Oracle Driver** (para comunicação com nosso banco de dados oracle)
- Busque pelo **Spring Data JPA** (para realizarmos a persistência no banco de dados).
- Busque pelo **Validation** (para realizarmos as validações necessários, seguindo as regras de negócios).



Adicionando dependências

Agora clique no botão **Explore** e observe que ele exibe o conteúdo do arquivo *pom.xml*.

EXPLORE CTRL + SPACE

Busque pelo **XML** que adiciona as dependências que acabamos de escolher, selecione e copie para a área de transferência (**Ctrl+C**) de seu computador (o código possivelmente deve estar entre as linhas 20 e 33).

```
19 <dependencies>
20   <dependency>
21     <groupId>org.springframework.boot</groupId>
22     <artifactId>spring-boot-starter-data-jpa</artifactId>
23   </dependency>
24   <dependency>
25     <groupId>org.springframework.boot</groupId>
26     <artifactId>spring-boot-starter-validation</artifactId>
27   </dependency>
28
29   <dependency>
30     <groupId>com.oracle.database.jdbc</groupId>
31     <artifactId>ojdbc8</artifactId>
32     <scope>runtime</scope>
33   </dependency>
34   <dependency>
35     <groupId>org.springframework.boot</groupId>
```

Adicionando dependências

Volte para seu projeto no *Eclipse*, abra o arquivo **pom.xml** e cole (**Ctrl+V**) as dependências copiadas dentro do **XML** que indica as dependências do projeto (**Ctrl+Shift+F** ajuda arrumar a indentação do código).

Importante: realize essa ação sem estar com seu projeto em execução.

```
21         <groupId>org.springframework.boot</groupId>
22         <artifactId>spring-boot-starter-web</artifactId>
23     </dependency>
24
25     <dependency>
26         <groupId>org.springframework.boot</groupId>
27         <artifactId>spring-boot-starter-data-jpa</artifactId>
28     </dependency>
29     <dependency>
30         <groupId>org.springframework.boot</groupId>
31         <artifactId>spring-boot-starter-validation</artifactId>
32     </dependency>
33     <dependency>
34         <groupId>com.oracle.database.jdbc</groupId>
35         <artifactId>ojdbc8</artifactId>
36         <scope>runtime</scope>
37     </dependency>
38
39     <dependency>
40         <groupId>org.springframework.boot</groupId>
```

Salve este arquivo e aguarde o *Eclipse* realizar o download das dependências (ou mesmo, *botão direito* do mouse no projeto, escolher *Maven* e a opção *Update Project...*).

Configurando o banco de dados

Configurando o banco de dados

Execute seu projeto (pela classe **ApiApplication**) e veja que o **Console** exibe uma mensagem de erro e interrompe a execução.

```
*****  
APPLICATION FAILED TO START  
*****  
  
Description:  
  
Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.  
  
Reason: Failed to determine a suitable driver class
```

Isso acontece pois devemos configurar a conexão com o banco de dados indicando que *tipo* de banco vamos usar, o *usuário* e a *senha*.

Isto é feito no arquivo **application.properties** (que está na pasta **src/main/resources**). Abra este arquivo e adicione o código como indicado abaixo (substitua **login** e **pass** pelo *seu usuário* e *senha* de seu servidor *Oracle*).

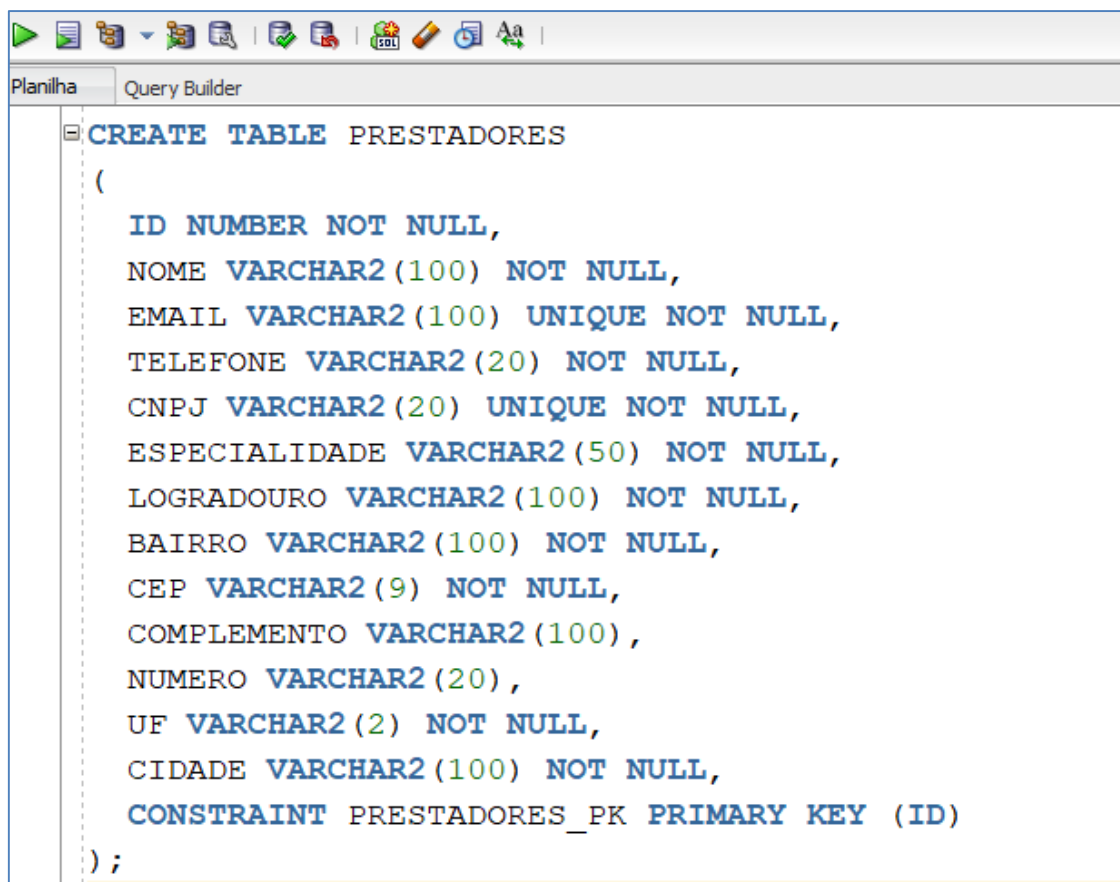
```
application.properties X  
1 spring.datasource.url=jdbc:oracle:thin:@oracle.fiap.com.br:1521:ORCL  
2 spring.datasource.username=login  
3 spring.datasource.password=pass  
4
```

Configurando o banco de dados

Agora que nossa conexão está funcionando, vamos ao próximo passo.

Nossa funcionalidade de *Cadastro de Prestador* deve adicionar um registro à tabela **PRESTADORES** no banco de dados.

Vamos utilizar o **SQL Developer** e criar essa tabela, faça como indica a codificação abaixo:

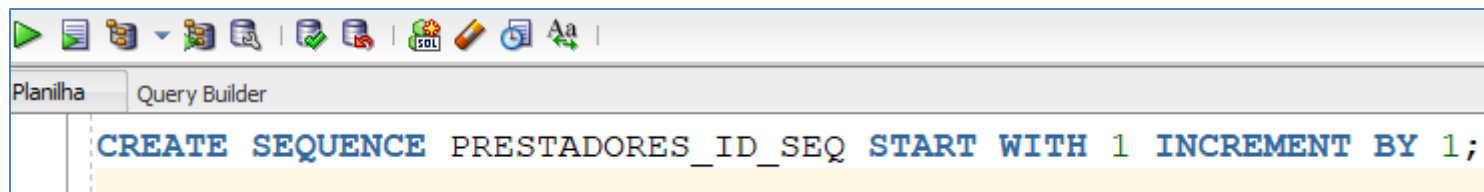


```
CREATE TABLE PRESTADORES
(
  ID NUMBER NOT NULL,
  NOME VARCHAR2(100) NOT NULL,
  EMAIL VARCHAR2(100) UNIQUE NOT NULL,
  TELEFONE VARCHAR2(20) NOT NULL,
  CNPJ VARCHAR2(20) UNIQUE NOT NULL,
  ESPECIALIDADE VARCHAR2(50) NOT NULL,
  LOGRADOURO VARCHAR2(100) NOT NULL,
  BAIRRO VARCHAR2(100) NOT NULL,
  CEP VARCHAR2(9) NOT NULL,
  COMPLEMENTO VARCHAR2(100),
  NUMERO VARCHAR2(20),
  UF VARCHAR2(2) NOT NULL,
  CIDADE VARCHAR2(100) NOT NULL,
  CONSTRAINT PRESTADORES_PK PRIMARY KEY (ID)
);
```

Configurando o banco de dados

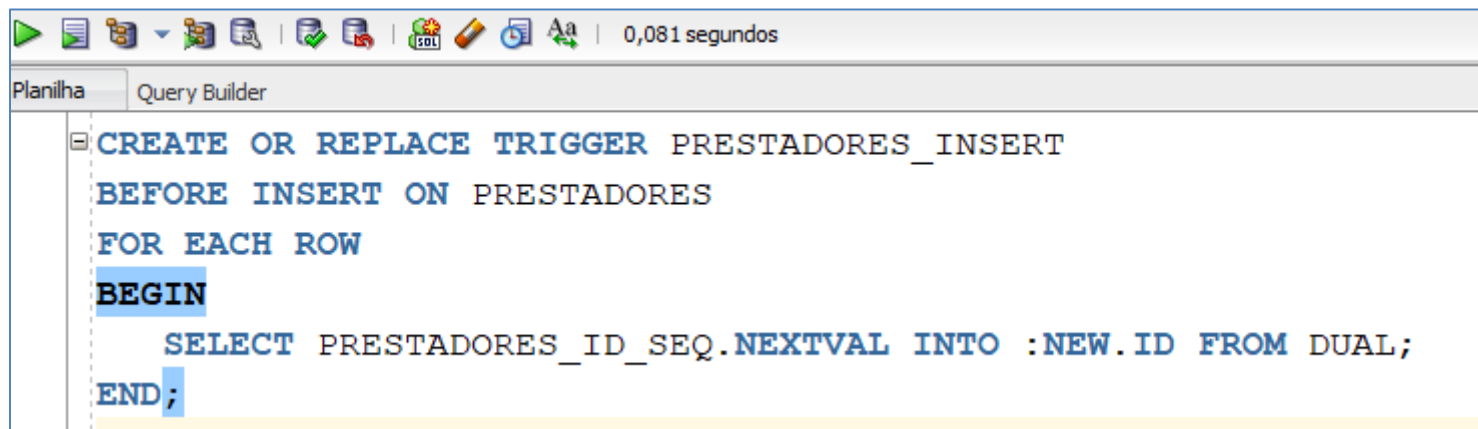
Para não precisar entrar com o código da chave primária manualmente vamos criar uma **Sequence** e em seguida uma **Trigger** para ativá-la.

Digite e execute o código como indicado abaixo:



A screenshot of a SQL query editor window. The title bar shows various icons and the text '0,081 segundos'. The window has two tabs: 'Planilha' and 'Query Builder'. The 'Query Builder' tab is active, and the text area contains the following SQL code: `CREATE SEQUENCE PRESTADORES_ID_SEQ START WITH 1 INCREMENT BY 1;`

```
CREATE SEQUENCE PRESTADORES_ID_SEQ START WITH 1 INCREMENT BY 1;
```



A screenshot of a SQL query editor window. The title bar shows various icons and the text '0,081 segundos'. The window has two tabs: 'Planilha' and 'Query Builder'. The 'Query Builder' tab is active, and the text area contains the following SQL code: `CREATE OR REPLACE TRIGGER PRESTADORES_INSERT BEFORE INSERT ON PRESTADORES FOR EACH ROW BEGIN SELECT PRESTADORES_ID_SEQ.NEXTVAL INTO :NEW.ID FROM DUAL; END;`

```
CREATE OR REPLACE TRIGGER PRESTADORES_INSERT  
BEFORE INSERT ON PRESTADORES  
FOR EACH ROW  
BEGIN  
    SELECT PRESTADORES_ID_SEQ.NEXTVAL INTO :NEW.ID FROM DUAL;  
END;
```

Entidades JPA

I Entidades JPA

Agora vamos continuar a implementar a funcionalidade de cadastros de prestadores.

Vamos voltar para **PrestadorController.java**. Nele, estamos recebendo um **DTO**, representado pelo **record** *DadosCadastroPrestador* e dando um *system.out*.

O que precisamos fazer, porém, é pegar o objeto e salvá-lo no banco de dados.

O ***Spring Data JPA*** utiliza o **JPA** como ferramenta de mapeamento de objeto relacional. Por isso, criaremos uma entidade **JPA** para representar uma tabela no banco de dados.

No pacote **prestador** vamos criar a classe **Prestador.java**.

Vamos criar os *atributos*, *getters/setters* e um *construtor vazio* para esta classe.

Atenção para o atributo **endereco** que será um objeto da classe **Endereco** que vamos criar na sequência.

A Classe Prestador

```
Prestador.java x
1 package br.com.reformas.api.prestador;
2
3 import br.com.reformas.api.endereco.Endereco;
4
5 public class Prestador {
6
7     private Long id;
8     private String nome;
9     private String email;
10    private String telefone;
11    private String cnpj;
12    private Especialidade especialidade;
13    private Endereco endereco;
14
15    public Prestador() {
16    }
17
18    public Long getId() {
19        return id;
20    }
21
22    public void setId(Long id) {
23        this.id = id;
24    }
25
26    public String getNome() {
27        return nome;
28    }
29
```

A Classe Prestador

```
30 public void setNome(String nome) {  
31     this.nome = nome;  
32 }  
33  
34 public String getEmail() {  
35     return email;  
36 }  
37  
38 public void setEmail(String email) {  
39     this.email = email;  
40 }  
41  
42 public String getTelefone() {  
43     return telefone;  
44 }  
45  
46 public void setTelefone(String telefone) {  
47     this.telefone = telefone;  
48 }  
49  
50 public String getCnpj() {  
51     return cnpj;  
52 }  
53  
54 public void setCnpj(String cnpj) {  
55     this.cnpj = cnpj;  
56 }  
57
```



```
58 public Especialidade getEspecialidade() {  
59     return especialidade;  
60 }  
61  
62 public void setEspecialidade(Especialidade especialidade) {  
63     this.especialidade = especialidade;  
64 }  
65  
66 public Endereco getEndereco() {  
67     return endereco;  
68 }  
69  
70 public void setEndereco(Endereco endereco) {  
71     this.endereco = endereco;  
72 }  
73 }
```

A Classe Endereco

No pacote **endereco** vamos criar a classe **Endereco.java**. Vamos criar os *atributos*, *getters/setters* e um *construtor vazio* para esta classe.

```
1 package br.com.reformas.api.endereco;
2
3 public class Endereco {
4
5     private String logradouro;
6     private String bairro;
7     private String cep;
8     private String cidade;
9     private String uf;
10    private String numero;
11    private String complemento;
12
13    public Endereco() {
14    }
15
16    public String getLogradouro() {
17        return logradouro;
18    }
19
20    public void setLogradouro(String logradouro) {
21        this.logradouro = logradouro;
22    }
23
24    public String getBairro() {
25        return bairro;
26    }
```

A Classe Endereco

```
27
28= public void setBairro(String bairro) {
29     this.bairro = bairro;
30 }
31
32= public String getCep() {
33     return cep;
34 }
35
36= public void setCep(String cep) {
37     this.cep = cep;
38 }
39
40= public String getCidade() {
41     return cidade;
42 }
43
44= public void setCidade(String cidade) {
45     this.cidade = cidade;
46 }
47
48= public String getUf() {
49     return uf;
50 }
51
52= public void setUf(String uf) {
53     this.uf = uf;
54 }
55
```

```
56 public String getNumero() {  
57     return numero;  
58 }  
59  
60 public void setNumero(String numero) {  
61     this.numero = numero;  
62 }  
63  
64 public String getComplemento() {  
65     return complemento;  
66 }  
67  
68 public void setComplemento(String complemento) {  
69     this.complemento = complemento;  
70 }  
71 }
```

Entidade JPA

Vamos adicionar as *anotações* da **JPA** para transformar a classe **Prestador** em uma entidade.

```
13 @Table(name = "prestadores")
14 @Entity(name = "Prestador")
15 public class Prestador {
16
17     @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
18     private Long id;
19     private String nome;
20     private String email;
21     private String telefone;
22     private String cnpj;
23
24     @Enumerated(EnumType.STRING)
25     private Especialidade especialidade;
26
27     @Embedded
28     private Endereco endereco;
29 }
```

I Entidade JPA

Vamos usar **@Embeddable** do **JPA** para que **Endereco** fique em uma classe separada, mas faça parte da mesma tabela de **Prestadores** junto ao banco de dados.

Para que isso funcione, vamos acessar a classe **Endereco** e adicionar, no topo do código, a anotação **@Embeddable** logo acima da classe.

```
3 import jakarta.persistence.Embeddable;
4
5 @Embeddable
6 public class Endereco {
7
8     private String logradouro;
9     private String bairro;
10    private String cep;
11    private String cidade;
12    private String uf;
13    private String numero;
14    private String complemento;
```

Interfaces Repository

Interfaces Repository

Para fazer a persistência, pegar o objeto **Prestador** e salvar no banco de dados, o **Spring Data** tem o **Repository**, que são *interfaces*. O *Spring* já nos fornece a implementação.

Vamos criar uma nova **interface** em no pacote **prestador**. O nome será **PrestadorRepository**. Vamos herdar de uma interface chamada **JpaRepository**, usando **extends**. Entre **<>**, passaremos dois tipos de objeto. O primeiro será o tipo da entidade trabalhada pelo *repository*, **Prestador**, e o tipo do atributo da chave primária da entidade, **Long**. A interface está criada:

```
PrestadorRepository.java X
1 package br.com.reformas.api.prestador;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4
5 public interface PrestadorRepository extends JpaRepository<Prestador, Long>{
6
7 }
8
```


Interfaces Repository

Agora já podemos utilizar o *repository* no *controller* **PrestadorController**.

Vamos declarar o *repository* como um atributo da classe.

Precisamos avisar ao *Spring* que esse novo atributo precisa ser instanciado. Faremos a *injeção de dependências* inserindo a anotação **@Autowired** acima do atributo.

No método cadastrar, vamos chamar o método que fará o *insert* na tabela do banco de dados.

```
PrestadorController.java X
1 package br.com.reformas.api.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
12
13 @RestController
14 @RequestMapping("/prestadores")
15 public class PrestadorController {
16
17     @Autowired
18     private PrestadorRepository repository;
19
20     @PostMapping
21     public void cadastrar(@RequestBody DadosCadastroPrestador dados) {
22         repository.save(new Prestador(dados));
23     }
24 }
```

Interfaces Repository

Precisamos criar na classe **Prestador** um construtor que receba como parâmetro o **DTO DadosCadastroPrestador**:

```
public Prestador(DadosCadastroPrestador dados) {  
    this.nome = dados.nome();  
    this.email = dados.email();  
    this.telefone = dados.telefone();  
    this.cnpj = dados.cnpj();  
    this.especialidade = dados.especialidade();  
    this.endereco = new Endereco(dados.endereco());  
}
```

Para finalizar, precisamos criar na classe **Endereco** um construtor que receba como parâmetro o **DTO DadosEndereco**.

```
public Endereco(DadosEndereco dados) {  
    this.logradouro = dados.logradouro();  
    this.bairro = dados.bairro();  
    this.cep = dados.cep();  
    this.cidade = dados.cidade();  
    this.uf = dados.uf();  
    this.numero = dados.numero();  
    this.complemento = dados.complemento();  
}
```

Agora só falta testar:

- Inicie o projeto (classe **ApiApplication**)
- Envie as informações a serem preenchidas pelo **Insomnia**
- Confira (no **SQL Developer**) se as informações foram gravadas.

Realize outros testes, mude as informações no **JSON** do **Insomnia** e preencha outros prestadores na tabela do banco de dados.

Praticando...

Agora implemente a funcionalidade de cadastro de clientes:

- Crie a tabela **Clientes** no banco de dados, uma ***sequence*** e um ***trigger*** para ela.
- Crie a classe **Cliente** e a transforme em uma entidade ***JPA***.
- Crie uma *interface* **ClienteRepository** que seja herdada da *interface* *JpaRepository*.
- Modifique a classe **ClienteController** para utilizar o método **save** do *Repository*.
- Envie requisições pelo **Insomnia** e preencha alguns registros na tabela **Clientes**.

Questionário

1. Informações como tipo de banco de dados, login e senha incluímos no arquivo:
 - a) pom.xml
 - b) ApiApplication da pasta src/main/java
 - c) application.properties da pasta src/main/java
 - d) application.properties da pasta src/main/resources

2. Qual a principal finalidade da *interface* Repository do *Spring Data*?
 - a) Indicar que o código da tabela terá preenchimento automático
 - b) Indicar que uma classe é uma entidade do banco de dados
 - c) Efetuar a validação com o banco de dados
 - d) Efetuar a persistência com o banco de dados




Java com Spring, Hibernate e Eclipse. Anil Hemrajani. Pearson, 2013.


Spring MVC: Domine o principal framework web Java. Alberto Souza. Série Caelum.

Site: alura.com.br

Até breve!

Questionário

1. Informações como tipo de banco de dados, login e senha incluímos no arquivo:
 - a) pom.xml
 - b) ApiApplication da pasta src/main/java
 - c) application.properties da pasta src/main/java
 -  d) application.properties da pasta src/main/resources

2. Qual a principal finalidade da *interface* Repository do *Spring Data*?
 - a) Indicar que o código da tabela terá preenchimento automático
 - b) Indicar que uma classe é uma entidade do banco de dados
 - c) Efetuar a validação com o banco de dados
 -  d) Efetuar a persistência com o banco de dados