Projeto Verdinhas

Apresentação do Projeto

Para contextualizar melhor a aplicação dos conceitos que estamos estudando na disciplina eu vou construir com vocês passo a passo um mini-projeto chamado verdinhas, onde a ideia inicialmente é fazer o registro de plantas e suas localizações na casa.

Todas as ferramentas e configurações de ambiente necessárias já foram instaladas e realizadas na atividade de configuração de ambiente. Nós vamos utilizar o Eclipse + Spring Tool Suite, PostgreSQL e o Maven.

Apresentação da Arquitetura

Agora vamos colocar em prática alguns dos conceitos e fundamentos que vimos nas aulas anteriores para construir a arquitetura do projeto.Ele será implementado em camadas e utilizará os padrões MVC e DAO. Isto significa que nosso projeto utilizará a separação de responsabilidades, onde cada uma das 3 camadas terão funções e responsabilidades diferentes.

Cada uma das letras da sigla denotam cada uma das camada que são as seguintes:

Model: É a camada responsável por encapsular a manipulação dos dados

View: é a camada do sistema que vai conter todas as implementações relacionadas à interação com os usuários. Certo professora mas o que isso quer dizer? Quer dizer que toda a lógica que precisa ser implementada de iteração com o usuário através das telas serão implementadas nessa camada.

Control: esta é a camada responsável por gerenciar todas as implementações relacionadas à regras de negócio do sistema.

DAO, que consiste em tratar de todas as manipulações de acesso aos dados em uma camada específica, no nosso caso a camada de acesso ao banco de dados. Ele que vai se conectar com o banco de dados, enviar e receber dados. É simplesmente isso, certo.

Implementação da Camada de Persistência

De um modo geral um sistema deve ser capaz de em algum momento persistir as informações dele. Existem diversas formas de se armazenar as informações. Pode ser utilizado um sistema de arquivos, um banco de dados nosql, mas normalmente nos sistemas corporativos, que é o nosso objeto de estudo nesta disciplina, os dados são normalmente armazenados em um sistema de banco de dados relacional.

Aqui na nossa disciplina nós vamos utilizar o banco de dados PostgreSQL que é um sistema de gerenciamento de banco de dados open source muito aceito pela comunidade e que é baseado num sistema de banco de dados robusto que é o Oracle. Ele é um dos únicos que oferece suporte a recursos mais avançados como o lock de linha por exemplo, onde eu posso travar numa tabela somente um registro específico ao invés de uma tabela inteira ganhando mais performance por exemplo.

Lembra quando falamos do desenvolvimento corporativo baseado no JEE? Em que dividimos o sistema em camadas e especializados partes e blocos do sistema de acordo com a sua responsabilidade? Partindo deste fundamento numa aplicação JEE a parte do sistema responsável por lidar com o código relacionado às questões relacionada ao acesso à dados é conhecida como camada de persistência. Esta camada vai fornecer às demais camadas do sistema, de forma abstraída, todo o tratamento necessário

O que é e qual a relação entre JEE, JPA, Hibernate e Spring Data JPA?

Para se interagir com o banco de dados em aplicações Java nós utilizamos normalmente uma API chamada Java Database Connect ou JDBC. Ela fornece pra gente um conjunto de classes que fazem o envio de instruções SQL para qualquer banco de dados, e para cada banco de dados existe um driver JDBC que tem as implementações necessárias para fazer a comunicação com o banco de dados.

Quando utilizamos JDBC nós escrevemos nossas consultas utilizando SQL ANSI ou instruções específicas do banco de dados em questão. Também precisamos manipular o resultado das nossas consultas para os nossos objetos de domínio e isso pode tornar o processo de desenvolvimento um pouco mais verboso e pouco produtivo.

Neste contexto um pessoal resolveu se reunir e tentar melhorar esta iteração com o banco de dados de forma a tornar mais produtivo, utilizando algumas convenções e automatizações e foi daí que surgiu um framework conhecido como Hibernate. Este framework evoluiu e foi tão bem aceito pela comunidade que as suas ideias e conceitos foram escritos em formas de especificações de API e incorporadas no JEE como o JPA.

O JPA (Java Persistence API) é uma API do JEE que descreve como deve ser o comportamento dos frameworks de persistência em JAVA. Está certo professora mas o que é que isso significa? Significa que como uma API é uma especificação não existe um código executável que você possa baixar para utilizar, pois a API estabelece as regras, certo? Descendo para o bom nível de comentário de código o JPA é um conjunto de interfaces e existem diferentes implementações para ela. Quando chegamos em nível de implementação começamos a falar nas possíveis implementações de JPA que existem. Podemos citar: TopLink (Oracle), Hibernate, EclipseLink, etc. A mais famosa e mais bem estabelecida na comunidade é o Hibernate que foi a origem de tudo e é a implementação que vamos utilizar.

Existe ainda um outro framework que vamos utilizar chamado Spring Data JPA que é um subprojeto do ecossistema Spring,que dá pra gente uma camada de abstração e já tem várias implementações úteis e maduras para lidar com banco de dados. Entre as facilidades podemos citar:

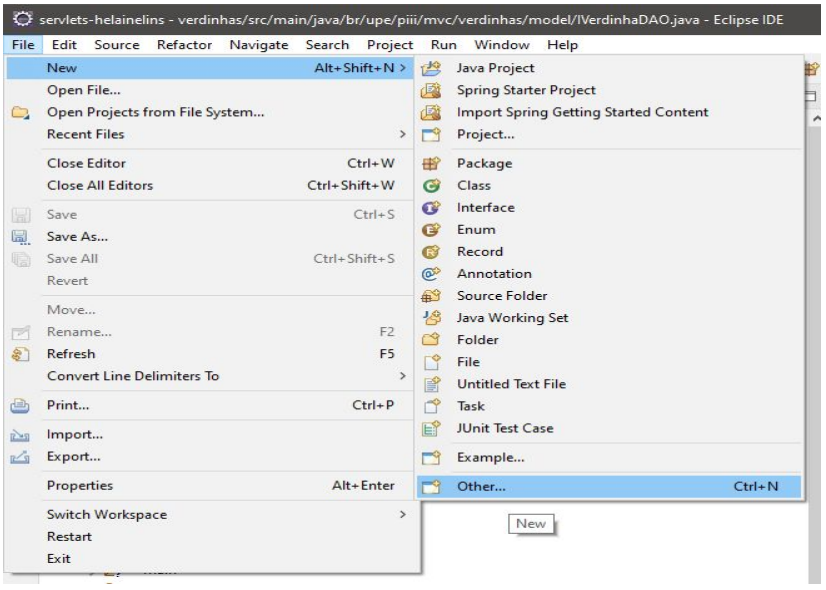
- Gerenciamento poderoso de repositórios customizados e preparados para fácil extensão.

- Derivação dinâmica para execução de querys seguindo convenções de nomes de métodos
- Suporte transparente a auditoria e paginação

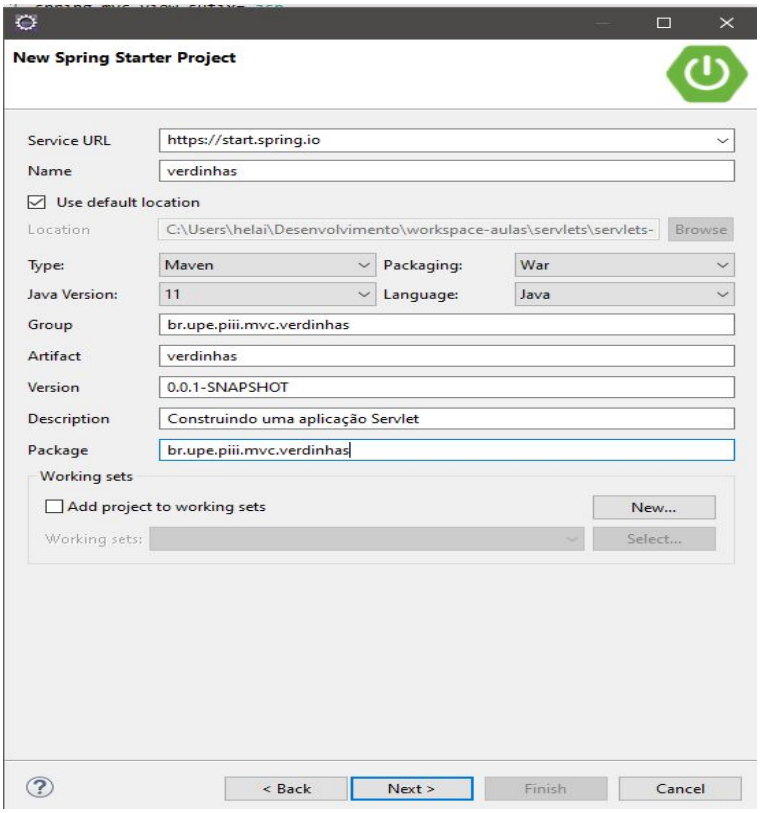
A seguir na implementação do projeto vamos ver como se faz a configuração do Spring e Spring Data JPA, implementação do DAO, mapeamento de objetos e realização de consultas.

Criando um Projeto utilizando o Spring Tool Suite (STS)

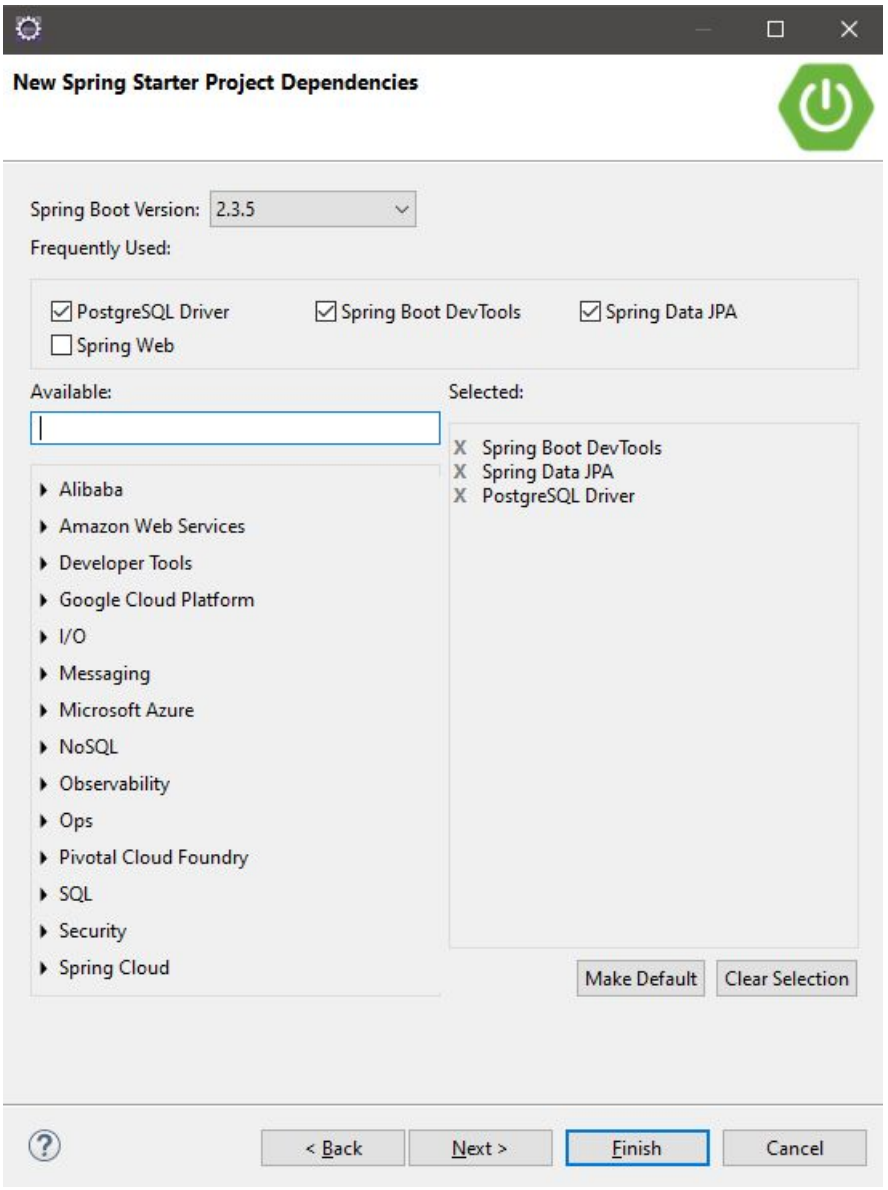
1. Abrir o menu File, selecione New e clique em Other...



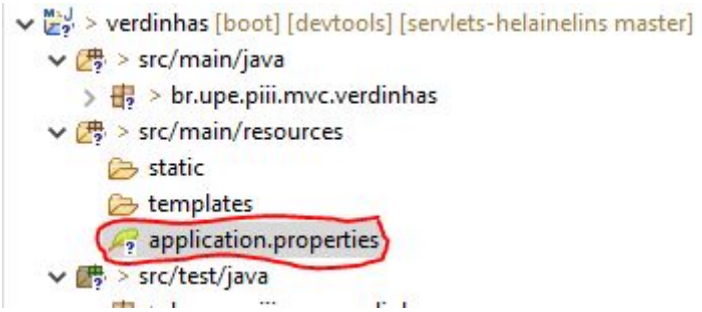
2. Preencher as informações de acordo com o seu projeto. Eu preenchi de acordo com o que julguei coerente para o projeto Verdinhas.



3. Selecionar as seguintes dependências: Spring Boot DevTools, Spring Data JPA e PostgreSQL Driver. Depois clicar em finish. O Sprint Tool

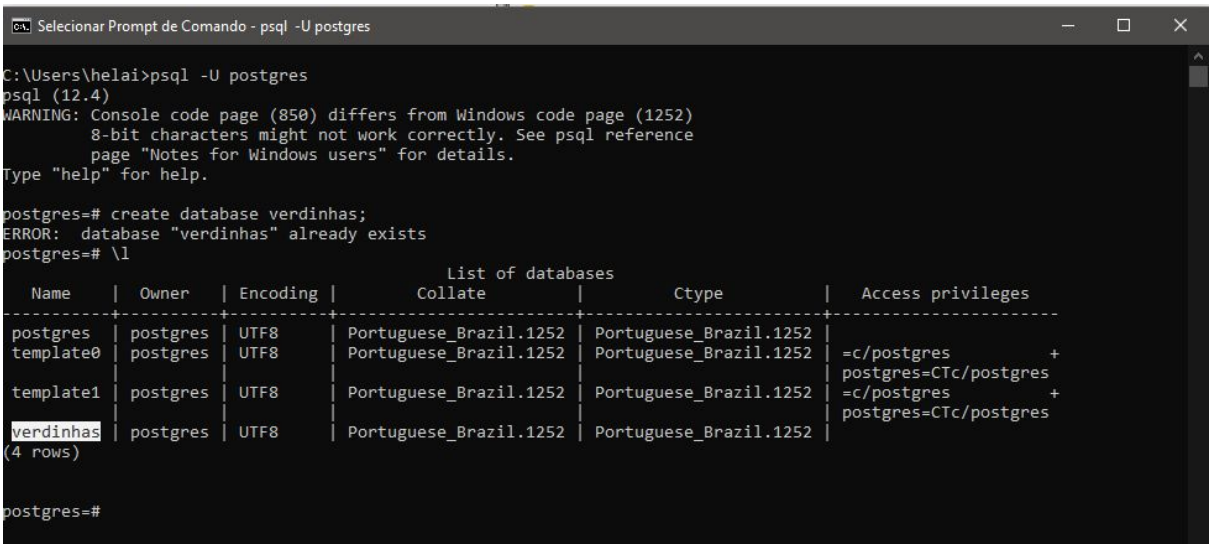


4. O STS cria dois arquivos que não vamos utilizar e eu optei por apagá-los, mas caso você decida deixá-los também não afetará o desenvolvimento do projeto. O primeiro é o ServletInicializer.java e o segundo é VerdinhasApplicationTests.java
5. Alterar o arquivo application.properties para acrescentar os dados de conexão com o banco de dados com os seus dados de acesso e alterar o nível de log




```
1
2  spring.jpa.hibernate.ddl-auto=update
3  spring.datasource.url=jdbc:postgresql://localhost:5432/verdinhas
4  spring.datasource.username=postgres
5  spring.datasource.password=postgres
6
7  logging.level.org.springframework=DEBUG
8  logging.level.org.hibernate=DEBUG
9  logging.level.org.postgresql=DEBUG
10 logging.level.root=ERROR
11
```

6. Acessar o PostgreSQL e criar o banco de dados, eu prefiro a linha de comando mas você pode fazer da mesma forma pelo PgAdmin



Criação das Entidades e Tabelas

No nosso projeto vamos trabalhar com duas entidades para podermos ilustrar ao menos um relacionamento simples entre elas. A primeira vou chamar de Verdinha que vai conter os dados de registro das plantinhas como identificador, nome e foto. A segunda entidade vai ser Local, onde vou guardar os dados e locais onde as plantinhas estão espalhadas pela casa. Ela vai conter somente um identificador, um nome e uma descrição.Então vamos lá iniciar a implementação.

1. **Criar a entidade Verdinha**
    a. Crie uma classe com o nome Verdinha na pasta src/main/java. Eu a criei em um pacote chamado br.upe.piii.mvc.verdinhas para facilitar a organização e

ajudar na compreensão de vocês, mas vocês podem fazer da forma que acharem mais conveniente quando forem criar as entidades do projeto de vocês.

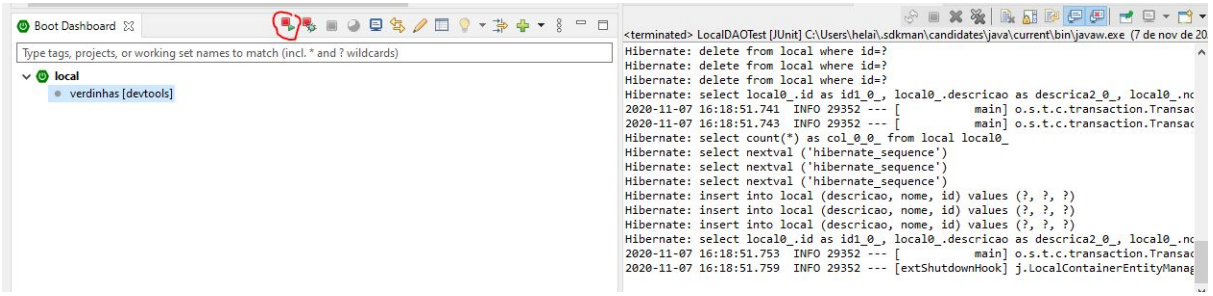
b. Adicionar a anotação @Entity na classe, que diz pro JPA que esta classe será persistida
c. Criar os atributos a serem armazenados e seus respectivos métodos gets e sets.
    i. id do tipo Long
    ii. nome do tipo String
    iii. foto do tipo byte[]
d. Informe ao JPA qual será o atributo que será utilizado como chave primária utilizando a anotação @Id acima da propriedade id. Peça ao JPA para gerar automaticamente um identificador quando salvarmos a sala adicionando a anotação @GeneratedValue(strategy= GenerationType.AUTO). Acrescente também a anotação @Lob na propriedade foto para mapear corretamente o array de bytes das fotos.

```java
package br.upe.piii.mvc.verdinhas.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Lob;

@Entity
public class Verdinha {

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

    private String nome;

    @Lob
    private byte[] foto;

    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 byte[] getFoto() {
        return foto;
    }

    public void setFoto(byte[] foto) {
        this.foto = foto;
    }

}
```

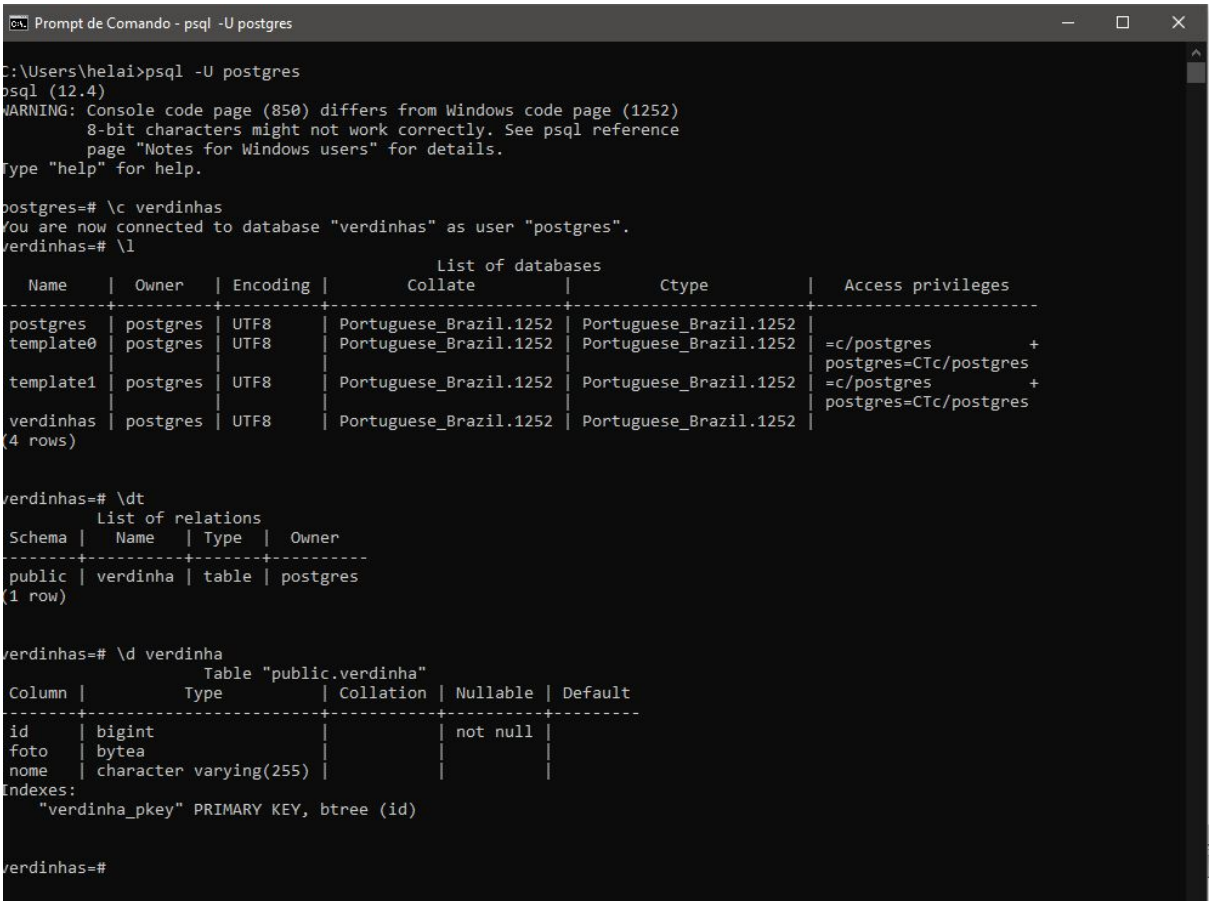
e. Não é obrigatório mas é uma boa prática acrescentar a implementação dos métodos equals e hashCode. Você pode implementar da forma que achar melhor, eu utilizei os métodos da classe java.util.Objects na implementação.

```java
@Override
public boolean equals(Object objeto) {
    boolean iguais = this == objeto;

    if (!iguais) {
        if (objeto == null || getClass() != objeto.getClass()) {
            iguais= false;
        } else {
            Verdinha verdinha = (Verdinha) objeto;
            iguais = Objects.equals(verdinha.getNome(), this.nome);
        }
    }

    return iguais;
}

@Override
public int hashCode() {
    return Objects.hash(nome);
}
```

f. Após salvar todas as alterações executar o projeto para que o spring inicialize o projeto na aba Boot Dashboard. O Spring Data irá utilizar os dados de conexões que acrescentamos no arquivo application.properties e irá realizar a conexão com a base de dados. O gerenciador de modelos do Hibernate sempre faz uma verificação de rotina antes de concluir a configuração do projeto. Ele então irá verificar que a entidade Verdinha está mapeada e que a tabela verdinha não existe no banco de dados. Como configuramos a propriedade ddl-auto toda vez que o projeto for iniciado o hibernate irá realizar todas as alterações no modelo de dados, relacionamentos e constraints automaticamente utilizando as convenções e anotações que utilizamos. Neste caso automaticamente ele irá criar a tabela verdinha para nós.



g. Verifique as alterações realizadas na base de dados para que você possa ver se tudo foi concluído conforme o esperado. Você pode usar o PGAdmin ou fazer a consulta pelo terminal de comando como eu fiz.
    i. Conectei na base executando psql -U postgres (caso o comando não seja reconhecido você pode ter esquecido de acrescentar a pasta bin dos postgresql na variável PATH)
    ii. Conectei na base de dados verdinhas utilizando o comando \c verdinhas
    iii. listei as tabelas da base de dados utilizando o comando \dt
    iv. listei as colunas da tabela utilizando o comando \d verdinha



2. **Criar a entidade Local**
    a. Criar uma classe com o nome Local, adicione a anotação @Entity na classe
    b. Criar os atributos a serem armazenados e seus respectivos métodos gets e sets.
        i. id do tipo Long (acrescente as anotações @Id e @GeneratedValue(strategy= GenerationType.AUTO)
        ii. nome do tipo String
        iii. descricao do tipo String

    d. Acrescentar a implementação dos métodos equals e hashCode
    e. Salvar todas as alterações
    f. Executar o projeto para que o spring realize a atualização da base de dados com as alterações que foram realizadas. Você pode repetir os mesmos passos que fizemos na entidade Verdinha

```java
package br.upe.piii.mvc.verdinhas.model;

import javax.persistence.Entity;

@Entity
public class Local {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public Long id;
    public String nome;
    public String descricao;

    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 String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

}
```

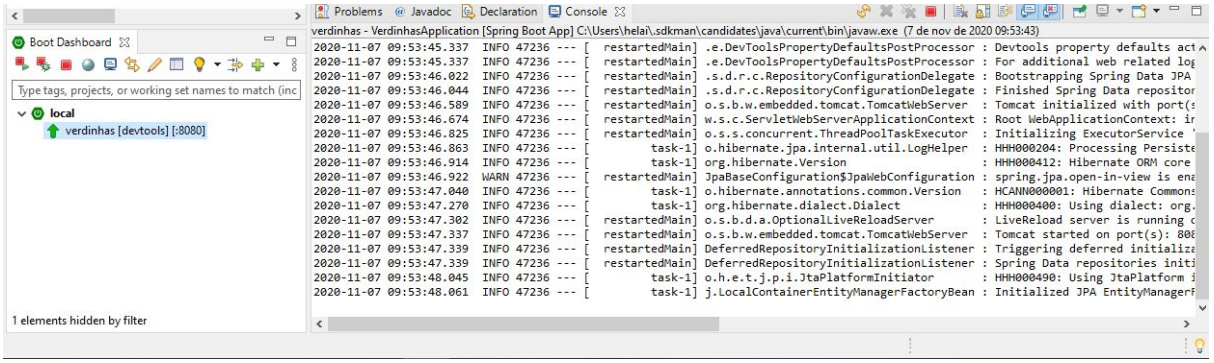
3. Atualizar a entidade Verdinha com o relacionamento para a entidade Local

a. Acrescentar um atributo chamado local na classe Verdinha e gere os métodos get e set correspondentes.

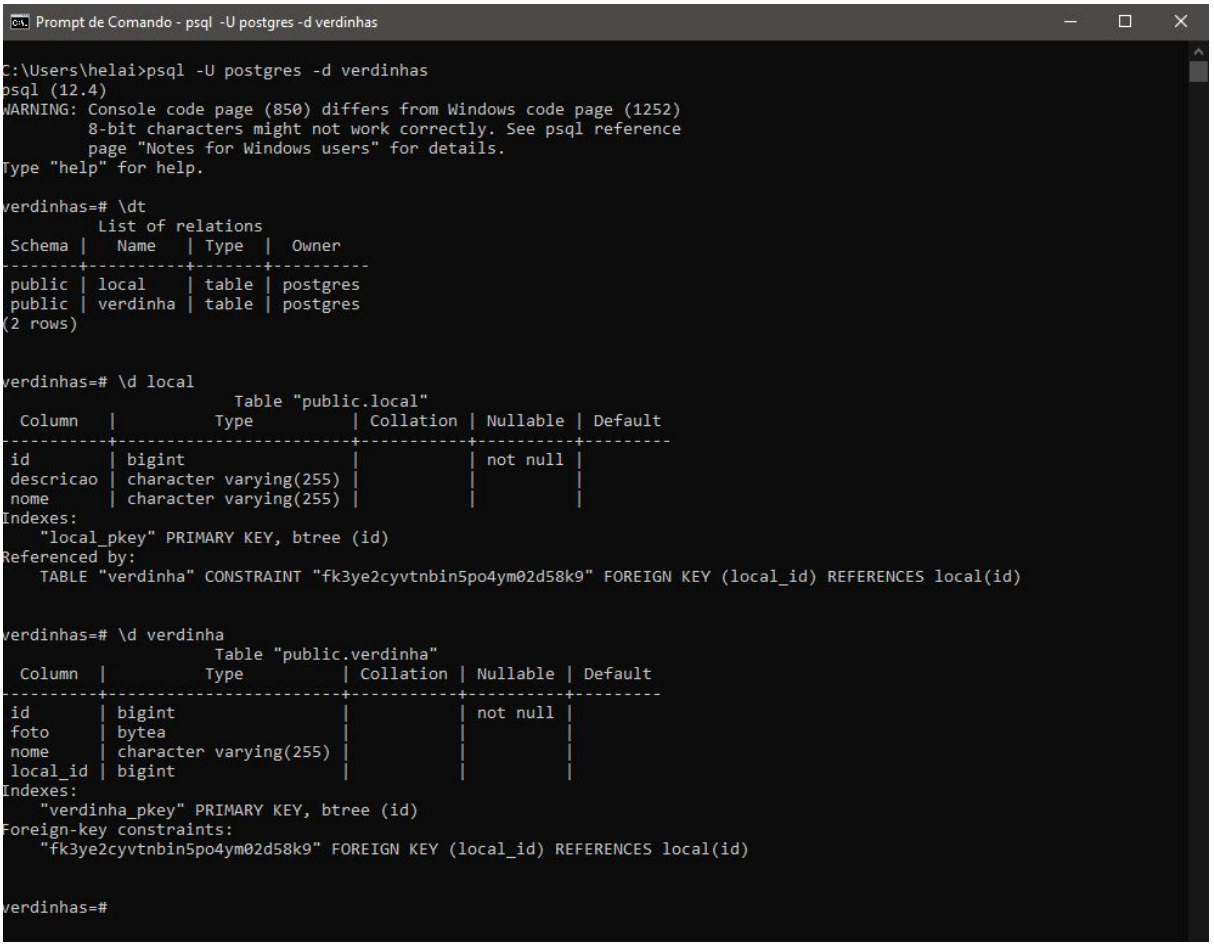
b. O tipo de relacionamento que temos entre as entidades é o seguinte: Como uma Verdinha só pode estar em um Local e um Local pode conter várias Verdinhas, este é um tipo de relacionamento um para muitos como vocês já devem ter visto na aula de banco de dados. Agora vamos informar ao JPA o tipo de relacionamento entre a entidade Verdinha e Local. Acrescentar a anotação @ManyToOne.

```java
@Entity
public class Verdinha {

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

    private String nome;

    @Lob
    private byte[] foto;

    @ManyToOne
    public Local local;

    public Local getLocal() {
        return local;
    }

    public void setLocal(Local local) {
        this.local = local;
    }
```

c. Salvar as alterações e executar o projeto para que o Hibernate realize a atualização dos relacionamentos.



d. Lembre-se de conectar a base de dados como fizemos nos passos anteriores e lista as informações da tabela verdinha e você verá que o Hibernate criou uma chave estrangeira na tabela verdinha que aponta para a tabela local.



7. Atualizar a entidade Local com o relacionamento para a entidade Verdinha

e. O JPA permite que acrescentemos um relacionamento bi-direcional, ou seja que acrescentemos também o relacionamento da entidade Local com a entidade Verdinha. Veja que quem possui a chave de relacionamento é a entidade Verdinha, mas podemos exprimir através deste tipo de relacionamento que a entidade Local pode trazer consigo a lista das Verdinhas que as utilizam, facilitando o acesso aos dados de relacionamento

na hora de implementar métodos de negócio por exemplo. Acrescentar um atributo chamado verdinhas do tipo List<Verdinha> na classe Local e gere os métodos get e set correspondentes.

f. Informar ao JPA o relacionamento bi-direcional utilizando a anotação “inversa” à que utilizamos na classe Verdinha que é @OneToMany

```
package br.upe.piii.mvc.verdinhas.model;

import java.util.List;

@Entity
public class Local {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public Long id;
    public String nome;
    public String descricao;

    @OneToMany
    public List<Verdinha> verdinhas;

    public List<Verdinha> getVerdinhas() {
        return verdinhas;
    }

    public void setVerdinhas(List<Verdinha> verdinhas) {
        this.verdinhas = verdinhas;
    }
```

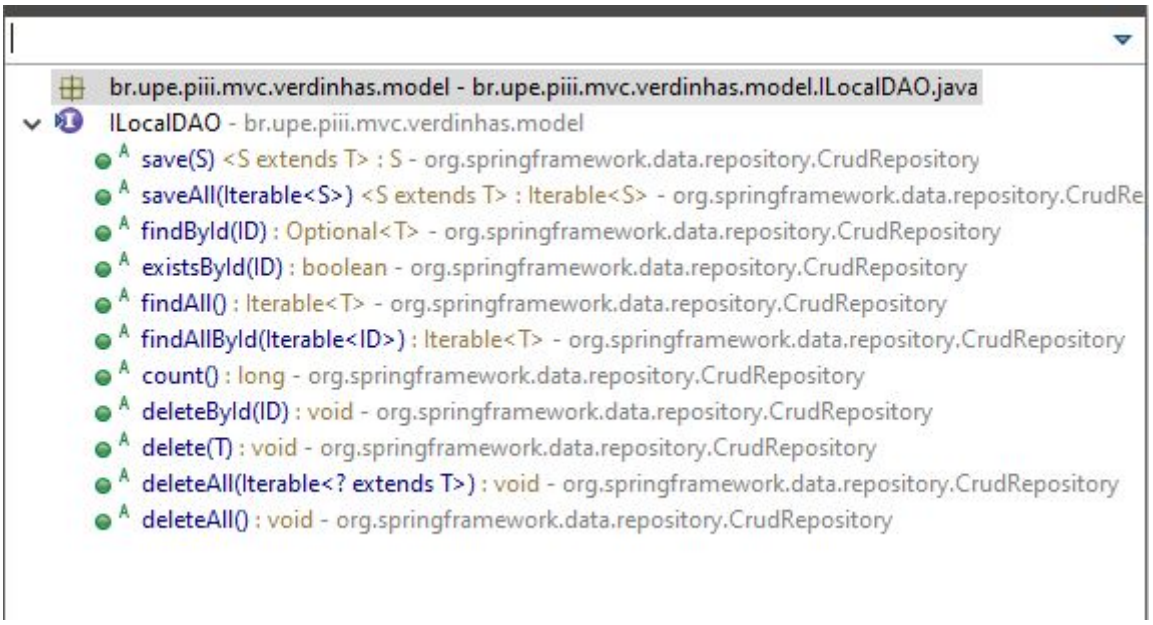
g. Salvar as alterações e executar o projeto.

h. Lembre-se de conectar a base de dados como fizemos nos exemplos anteriores e verifique que o Hibernate não realizou mais nenhuma alteração, pois o mapeamento que acrescentamos é apenas para adicionar mais “conveniência” à implementação, ela não altera a estrutura física das tabelas. Salve todas as alterações realizadas, execute o projeto, conecte a base de dados e liste as propriedades das tabelas para constatar que não foi alterado nada físico nas estruturas das tabelas.

Criação dos DAOs

Agora nós vamos criar os DAOs das duas entidades para criarmos o CRUD para cada uma das Entidades. Mas professora o que danado é CRUD? Esta é uma sigla que vem das iniciais das quatro operações básicas sobre entidades em um sistema de cadastro que é C-Create é a operação de criar e inserir registros, Restore que é a busca , Update que é a operação de atualizar valores e D-Delete que é a operação de apagar registros.

Lembra que eu comentei que o projeto Spring Data JPA traz algumas facilidades de implementação que agilizam e facilitam muito o processo de desenvolvimento ? A interface `org.springframework.data.repository.CrudRepository` utiliza um recursos de acrescenta a implementação destes métodos em tempo de execução



1. **Criar o IVerdinhaDAO**
   a. Crie uma interface chamada IVerdinhaDAO no seu pacote model e herde da interface org.springframework.data.repository.CrudRepository.
   b. Informe ao Spring quem é a sua entidade e qual o tipo do seu identificador. Para isso acrescente a tipagem dos parâmetros genéricos na interface da seguinte maneira:

```
package br.upe.piii.mvc.verdinhas.model;

import org.springframework.data.repository.CrudRepository;

public interface IVerdinhaDAO extends CrudRepository<Verdinha, Long> {

}
```

   c. E pronto, seu repositório está criado com toda a implementação padrão com o mínimo de esforço.

2. **Criar o ILocalDAO**
   a. Crie uma interface chamada ILocalDAO no seu pacote model e herde da interface org.springframework.data.repository.CrudRepository.
   b. Informe ao Spring quem é a sua entidade e qual o tipo do seu identificador como fizemos no IVerdinhaDAO:

```
1  package br.upe.piii.mvc.verdinhas.model;
2
3  import org.springframework.data.repository.CrudRepository;
4
5  public interface ILocalDAO extends CrudRepository<Local, Long> {
6
7  }
```

   c. E pronto, o outro repositório está criado.

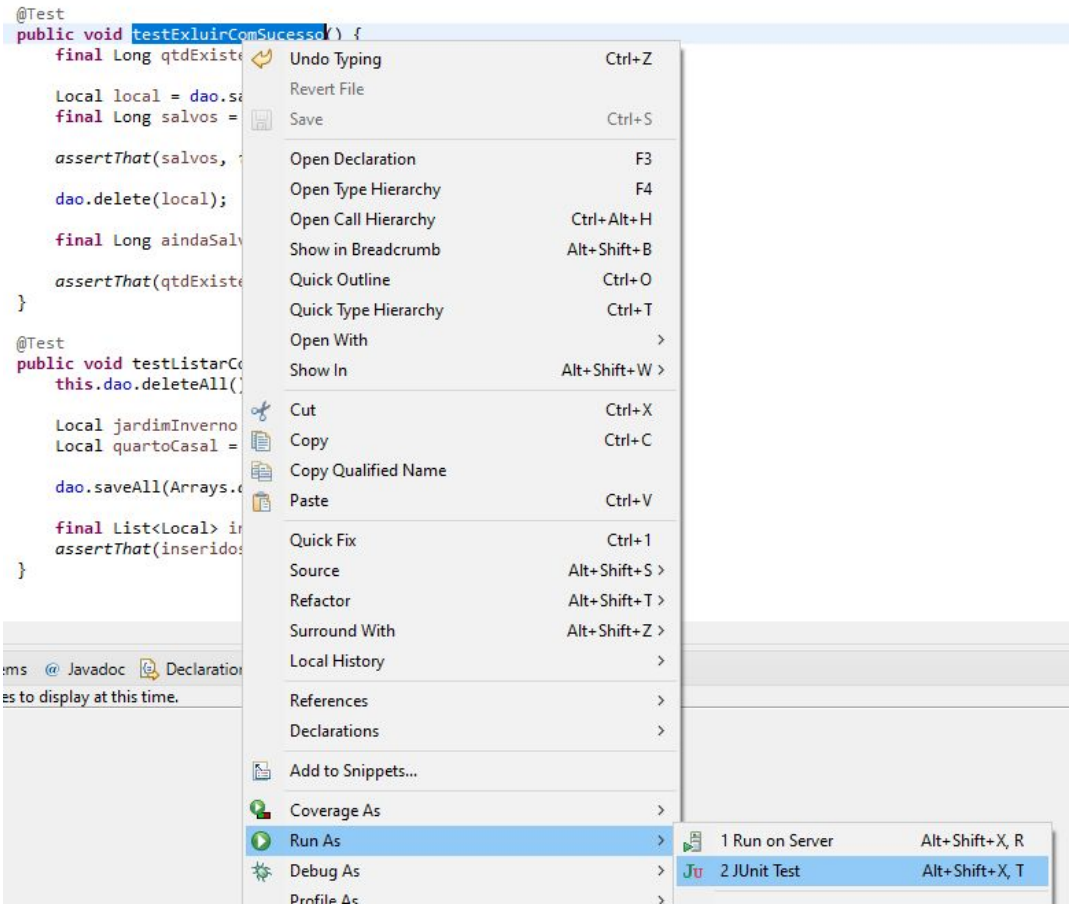
Construindo testes utilizando JUnit e @DataJpaTest

Agora vamos construir alguns testes unitários para que possamos verificar se os métodos estão funcionando como o esperado. A ideia aqui não é em nenhum momento validar se a implementação do Spring e do Hibernate estão corretas, mas sim permitir que você materialize o resultado do que construímos juntos até aqui.

O Spring Boot permite que criemos classes de testes de maneira simples utilizando o JUnit, e com poucas configurações através da utilização de algumas anotações. Vamos passar o olho em algumas delas para que eu te explique o propósito e necessidade delas.

- @RunWith(SpringRunner.class) : Realiza a integração das configurações entre o Spring e o JUnit e inicializa o contexto da aplicação
- @DataJpaTest: Anotação do Spring Boot que faz a configuração automática de um contexto transacional baseado em JPA para a realização dos testes acessando a base de dados.
- @AutoConfigureTestDatabase(replace = Replace.NONE): Por padrão o Spring Boot utiliza o banco de dados em memória H2, para que você possa visualizar os dados na base de dados que criamos incluímos esta anotação e com isso o Spring Boot passa a utilizar a conexão com o banco de dados. Ele vai ler as configurações de conexões do arquivo application.properties
- @Rollback(false): O DataJPATest por convenção cria uma transação para a execução dos testes e ao final da execução ele aplica um rollback para que as informações não sejam salvas na base de dados, o que em alguns contextos poderia “sujar” a base de testes. Como neste caso queremos que os dados permaneçam na base e você possa visualizar que tudo está funcionando utilizamos esta anotação para que ele deixe de aplicar o rollback.
- @Autored: É através desta anotação que o Spring vai instanciar, inicializar e realizar a injeção de objetos, no nosso caso aqui o Dao que vamos testar.
- @Test: Esta anotação informa ao JUnit que o método anotado é um teste a ser executado

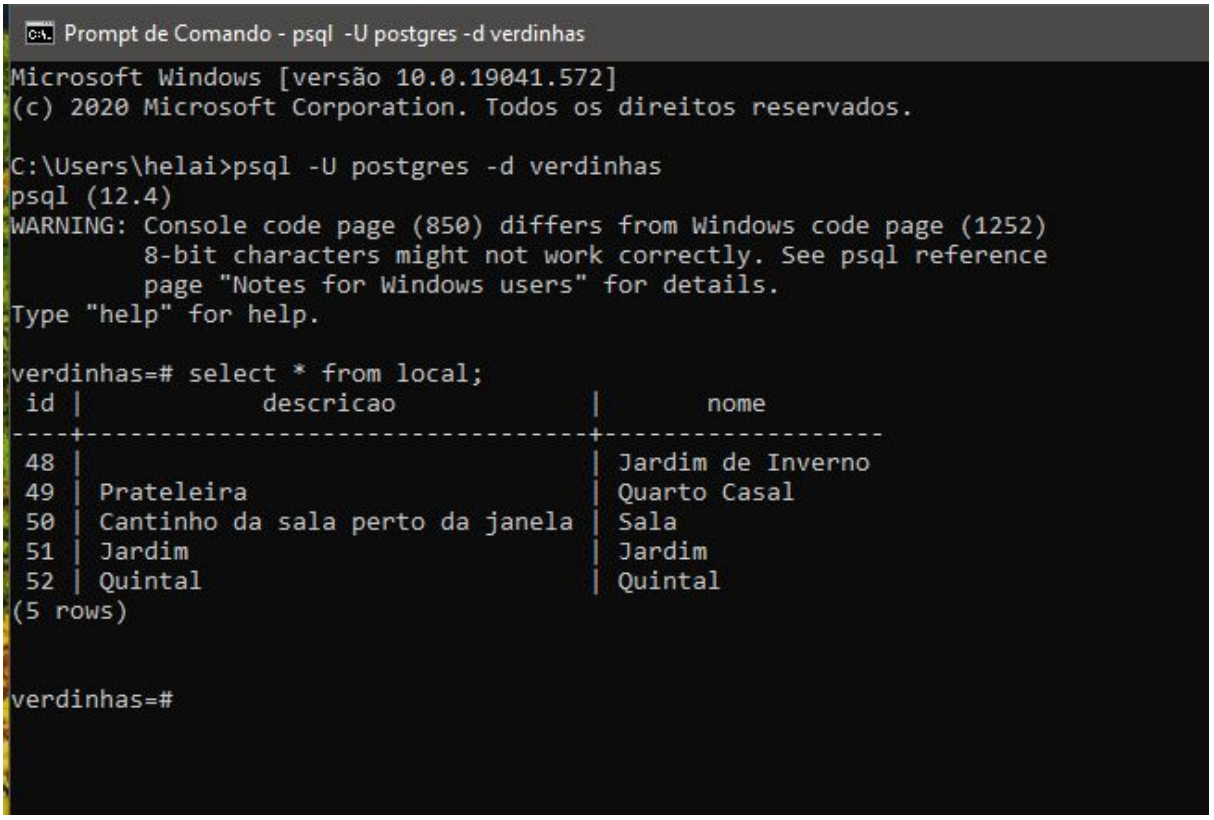
Aqui no nosso exemplo eu criei métodos de testes bem simplórios, apenas para materializar a nossa implementação até o momento e que a persistência no banco de dados está sendo realizada. Foram 4 métodos de teste um para cada operação do CRUD. Cada um dos métodos deve estar com a anotação @Test e depois de criados você pode executá-los pelo eclipse, clicando com o botão direito no nome do método, escolhendo a opção Run As.. e depois em JUnit Test.



1. **Criando a classe de testes para ILocalDAO**
    a. Criei a classe LocalDAOTest e acrescentei as anotações: @RunWith(SpringRunner.class), @DataJpaTest, @AutoConfigureTestDatabase(replace = Replace.NONE) e @Rollback(false)
    b. Acrescentei um atributo privado chamado dao do tipo ILocalDAO que será o repositório a ser testado
    c. Criei um teste para inclusão: a implementação faz uma contagem no repositório antes de inserir, insere 3 instâncias de Local e depois realiza a contagem para ver se ela foi incrementada de 3. Após executar este método de teste você poderá, como nos passos anteriores, executar uma consulta e ver que as três instâncias que inserimos estão presentes entre os registros da tabela local da nossa base de dados.
    d. Criei o teste para alteração: no método é inserida uma instância de Local. Depois fiz uma alteração na descrição e verifico se a alteração foi realizada conforme o esperado.
    e. Criei o teste para exclusão, realizei uma contagem de registros no início, fiz uma inserção de instância e em seguida a apaguei. Depois verifiquei se a contagem de registros estava igual a do início.
    f. Criei mais um teste para a listagem de registros, onde iniciei limpando a base, depois inseri dois registros, fiz a listagem e verifiquei se a quantidade era a mesma que havia inserido.
    g. O resultado final ficou como na imagem abaixo

```java
import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.MatcherAssert.assertThat;

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

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;
import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase.Replace;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.test.annotation.Rollback;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace = Replace.NONE)
@Rollback(false)
public class LocalDAOTest {

    @Autowired
    private ILocalDAO dao;

    public void testeIncluirComSucesso() {

    @Test
    public void testAlterarComSucesso() {
        Local local = new Local("Sala", "Cantinho da sala perto da janela");
        dao.save(local);

        String novaDescricao = "Na estante";
        local.setDescricao(novaDescricao);

        Local localAlterado = dao.save(local);

        assertThat(localAlterado.getDescricao(), is(equalTo(novaDescricao)));
    }

    @Test
    public void testExluirComSucesso() {
        final Long qtdExistentes = dao.count();

        Local local = dao.save(new Local("Terraço", "Área coberta"));
        final Long salvos = dao.count();

        assertThat(salvos, is(equalTo(qtdExistentes + 1)));

        dao.delete(local);

        final Long aindaSalvos = dao.count();

        assertThat(qtdExistentes, is(equalTo(aindaSalvos)));
    }

    @Test
    public void testListarComSucesso() {
        this.dao.deleteAll();

        Local jardimInverno = new Local("Jardim de Inverno", "");
        Local quartoCasal = new Local("Quarto Casal", "Prateleira");

        dao.saveAll(Arrays.asList(jardimInverno, quartoCasal));

        final List<Local> inseridos = (List<Local>) dao.findAll();
        assertThat(inseridos.size(), is(equalTo(2)));
    }
}
```

h. Também podemos ver um resultado de consulta a base de dados após a realização de alguns testes


Prompt de Comando - psql -U postgres -d verdinhas
Microsoft Windows [versão 10.0.19041.572]
(c) 2020 Microsoft Corporation. Todos os direitos reservados.
C:\Users\helai>psql -U postgres -d verdinhas
psql (12.4)
WARNING: Console code page (850) differs from Windows code page (1252)
8-bit characters might not work correctly. See psql reference
page "Notes for Windows users" for details.
Type "help" for help.
verdinhas=# select * from local;
id | descricao | nome
48 | | Jardim de Inverno
49 | Prateleira | Quarto Casal
50 | Cantinho da sala perto da janela | Sala
51 | Jardim | Jardim
52 | Quintal | Quintal
(5 rows)
verdinhas=#