Instituto Politécnico de Castelo Branco
Escola Superior de Tecnologia

JPA e Spring Data

Unidade Curricular
Aplicações Internet Distribuídas / Aplicações Distribuídas

Licenciatura em Engenharia Informática
Mestrado em Desenvolvimento de Software e Sistemas Interactivos
UTC Informática
Est-IPCB
Ano Letivo 2022/2023

Prof.º Doutor Alexandre Fonte
(adf@ipcb.pt)

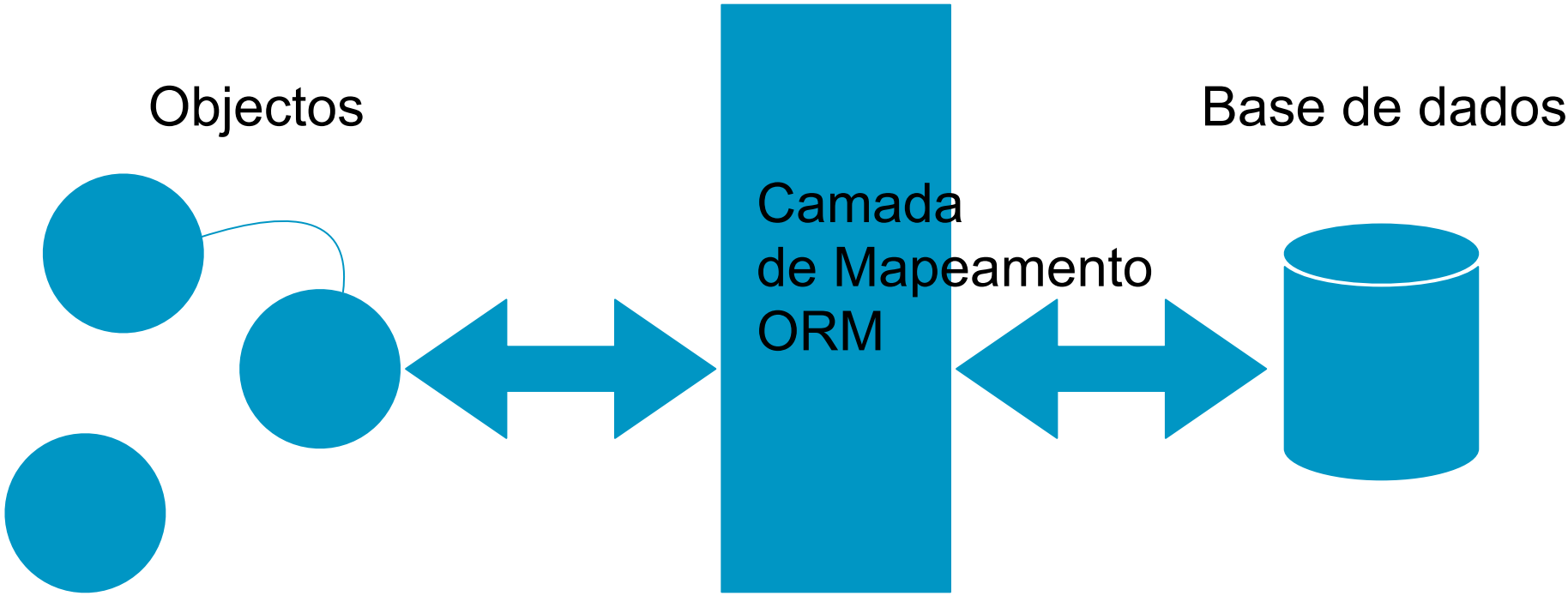
Versão: 20 outubro de 2022

Sumário

- Java Persistence API
- Spring Data
- Spring Data JPA

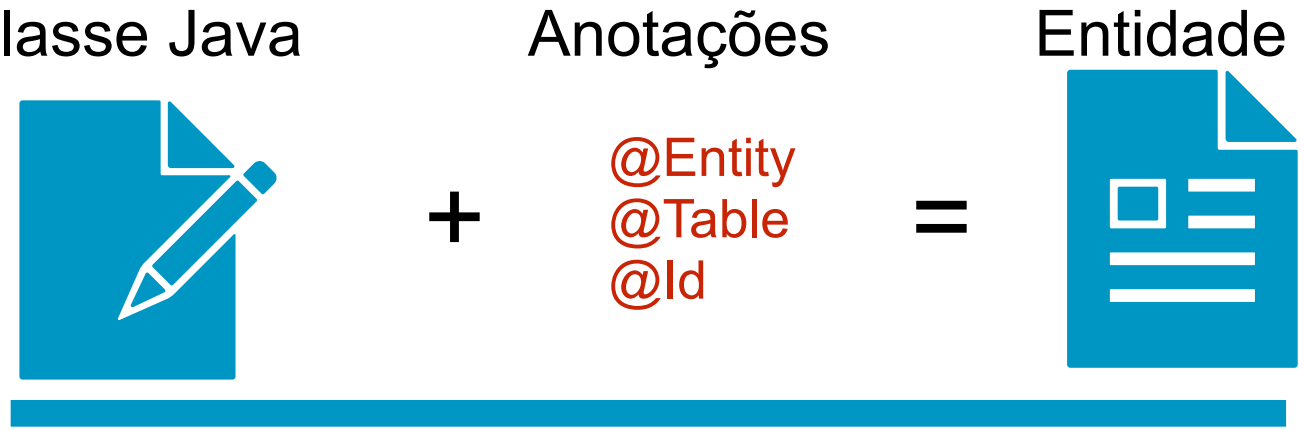
ORM (Object Relational Mapper)

- Técnica para relacionar um objecto com os dados que ele representa na base de dados.



JPA (Java Persistence API)

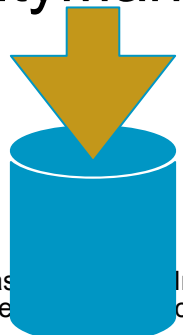
- Especificação Java (JSR-338) de como os frameworks ORM se devem comportar



Java Persistence Query Language

EntityManagerFactory

EntityManager



Base de dados

JPA (Java Persistence API)

- Conceito de Entidade JPA
  - -Uma Entidade JPA é a unidade básica de persistência
  - -É uma tabela numa base de dados.
  - -Uma entidade é representada por uma classe Java.
  - -O estado de uma instância de uma entidade é persistido/ corresponde a uma entrada/registo na tabela da base de dados.
  - -Está sujeita às Operações CRUD (Create, Read, Update e Delete)
  - -A base de dados é referida como: **Data Source ou Data Store**
    - **Nota:** Na camada inferior é usada a API JDBC, neste caso a base de dados é um **JDBC Resource**

Anatomia de uma Entidade
Anotações JPA

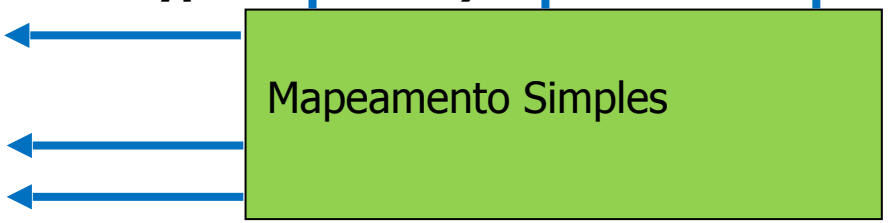
- As anotações **@Entity** e **@Id** são os requisitos mínimos de metadados para tornar uma classe Java numa entidade
  - A classe torna-se numa entidade se for decorada com **@Entity.** O contentor após a encontrar, reconhece-a como uma classe persistente e procura por outras anotações na classe.
  - A anotação **@Id** permite especificar o atributo identificador para a entidade, correspondente à chave primária.
  - A classe entidade não pode ter um constructor com parâmetros, nem pode ser do tipo final.
- Cada campo corresponde a uma coluna de uma tabela na base de dados.
- Um campo da classe anotado com **@Transient** será ignorado pelo contentor
- Por omissão, não é requerido especificar os nomes das tabelas, das colunas ou chave forasteira em que a entidade é mapeada. Se for necessário usar as anotações:
  - **@Table(name="nome da tabela")**
  - **@Column (name="nome da coluna")**
  - **@JoinColumn(name="nome chave forasteira")**

Exemplo Básico de uma Classe Entidade e do seu Mapeamento

Tabela User

id	primeiroNome	ultimoNome
1	Alexandre	Fonte
2	José	Pereira

```
@Entity
@Table(name="User")
  public class Utilizador{
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
@Column(name="primeiroNome")
private String firstName;
private String ultimoNome;
@Transient
private String codigoPostal;
//Getters e Setters
}
```



Nota: O campo codigoPostal anotado com @Transient não será persistido para a tabela na base de dados.

Anotações @Table, @Column
Mais alguns detalhes

- **@Table** pode ter também um atributo *schema*.

```
@Entity
@Table(name="UTILIZADOR", schema="ESTCB")
public class Utilizador {
        // fields, getters and setters
}
```

O elemento schema serve para distinguir conjuntos de entidades

- **@Column** pode ter vários atributos como *name*, *length*, *nullable*, e *unique*

```
@Column(name="PRIMEIRO_NOME", length=50, nullable=false, unique=false)
private String firstName;
```

"The name element specifies the name of the column in the table. The length element specifies its length. The nullable element specifies whether the column is nullable or not, and the unique element specifies whether the column is unique.

If we don't specify this annotation, the name of the column in the table will be the name of the field".

Problemas com os formatos das “Datas”

- The **java.sql** package contains JDBC types that are aligned with the types defined by the SQL standard:

  -**Date** corresponds to the DATE SQL type, which is only a date without time.

  -**Time** corresponds to the TIME SQL type, which is a time of the day specified in hours, minutes and seconds.

  -**Timestamp** includes information about date and time with precision up to nanoseconds and corresponds to the TIMESTAMP SQL type.

- The type **java.util.Date** contains both date and time information, up to millisecond precision. **But it doesn't directly relate to any SQL type**.

- This is why we need another annotation to specify the desired SQL type.

java.sql.Date vs java.utilDate

Date Type	java.util	java.sql
Date Time	2016-29-02 12:10:15.0	2016-29-02 12:10:15.0
Time Stamp	2016-29-02 12:10:15.0	2016-29-02 12:10:15.0
Date	2016-29-02 00:00:00.0	2016-29-02
Time	1970-01-01 12:10:15.0	12:10:15.0

java.util.Calendar
também sofre de problemas semelhantes.

Créditos: https://www.baeldung.com/hibernate-date-time

Problemas com os formatos das “Datas”

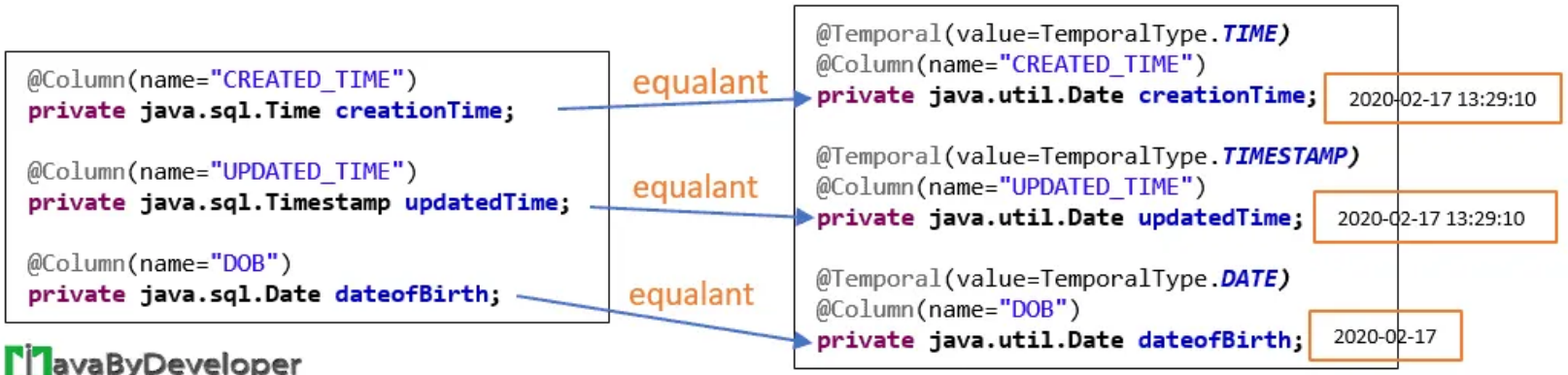
- **@Temporal** annotation solves the one of the major issue of converting the date and time values from Java object to compatible database type.
- In JPA or Hibernate, **@Temporal should only be set on a java.util.Date or java.util.Calendar property.**
- JPA supports:

```
1. @Temporal(TemporalType.TIMESTAMP)
2. @Temporal(TemporalType.DATE)
3. @Temporal(TemporalType.TIME)
```

- **These types are equivalent of**

```
1. java.sql.Timestamp
2. java.sql.Date
3. java.sql.Time
```

O melhor é usar sempre as classes correspondentes java.sql ou os novos tipos do pacote **java.time!**



JavaByDeveloper

Créditos: https://javabydeveloper.com/temporal/

Estratégias de Geração da Identidade

- Os identificadores únicos podem ser gerados:
  - **Pela aplicação:** Código que cria a entidade e atualiza o identificador
  - **Pelo fornecedor JPA:** decorando o identificador com `@GeneratedValue` este é Gerado na base de dados
- Estratégias
  - **GenerationType.AUTO:** É o fornecedor de persistência que escolhe a estratégia mais adequada de acordo com a base de dados.
  - **GenerationType.IDENTITY:** Os valores a serem atribuídos ao identificador único serão gerados pela coluna de auto incremento da base de dados.
  - **GenerationType.SEQUENCE:** Os valores serão gerados a partir de uma sequência (sequence) da base de dados criada para o efeito.
  - **GenerationType.TABLE:** Com a opção TABLE é necessário criar uma tabela para gestão das chaves primárias. Pouco recomendada, a tabela criada precisa de ser acedida com frequência.

Geração da Identidade: Exemplos

```
@Entity
public class Livro {
    @Id
    @GeneratedValue(
        strategy = GenerationType.SEQUENCE,
        generator = "sequence_id_livros"
    )
    @SequenceGenerator(
        name =  "sequence_id_livros",
        sequenceName = “sequence_livro”,
        initialValue =  5
    )
    private Long id;

    // restantes dos atributos e métodos
}
```

Define um gerador da sequência referenciado pelo nome.
O valor inicial é 5.

Exemplo criação de uma sequência numa BD Oracle:

```
SEQUENCE sequence_livro START WITH 5 INCREMENT BY 2;
```

Geração da Identidade: Exemplos

```
@Entity
public class Livro {
    @Id
    @GeneratedValue(
        strategy = GenerationType.TABLE,
        generator = "tabela_id_livros"
    )
    @TableGenerator(
        name =  "tabela_id_livros",
        table = "ids_livro",
    )
    private Long id;

    // restantes dos atributos e métodos
}
```

Define um gerador da tabela referenciado pelo nome.

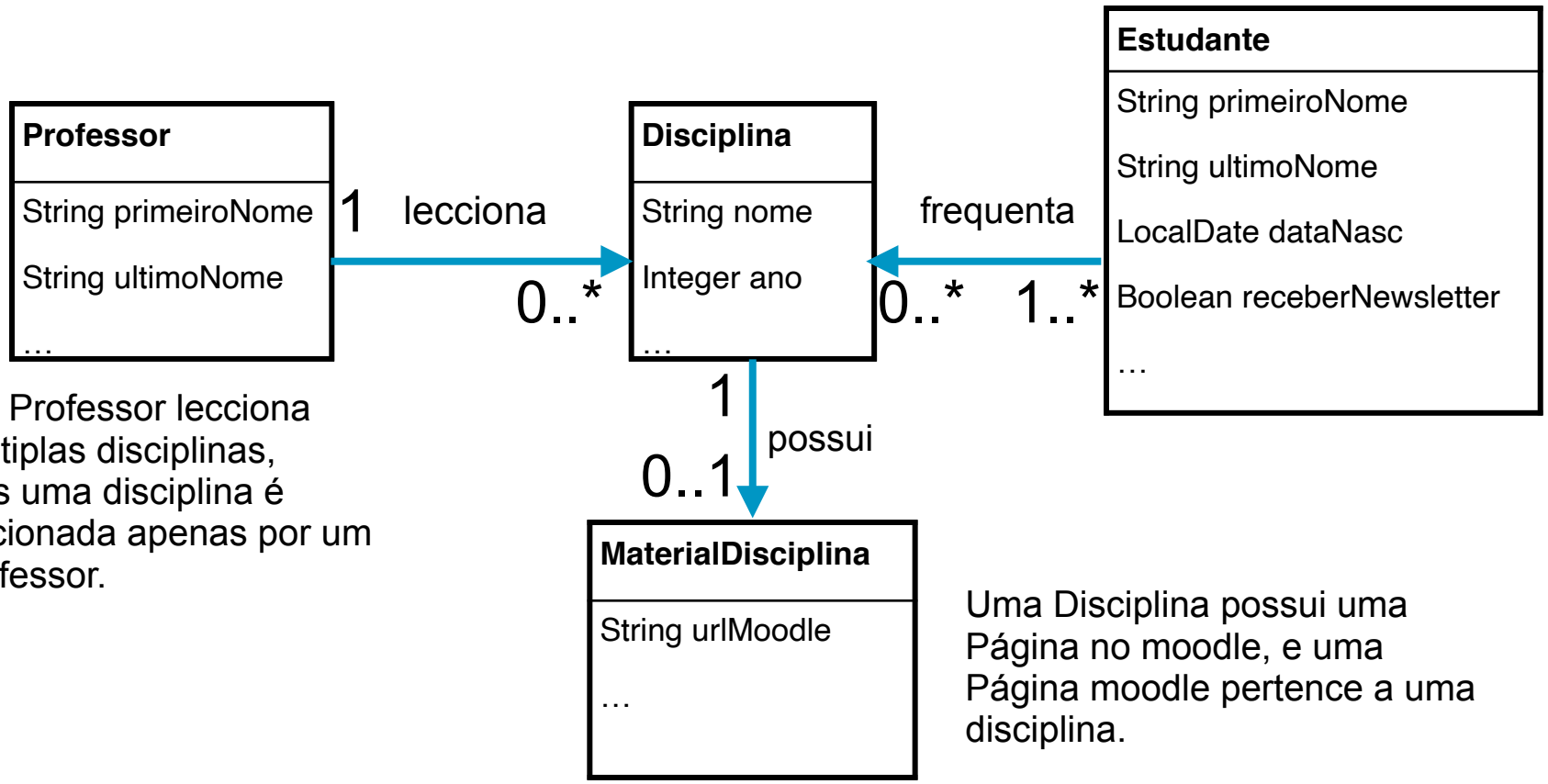
Relacionamentos entre Entidades

- One-To-One:
  - Uma instância (linha de uma tabela) relaciona-se com uma única instância de outra entidade (linha de outra tabela).
- One-To-Many:
  - Uma instância (linha de uma tabela) relaciona-se com mais de que uma instância de outra entidade.
  - Mas, uma instância da outra entidade apenas se relaciona com uma instância da 1.ª.
- Many-To-One:
  - Múltiplas instâncias relacionam-se com uma instância. Mas, uma instância da outra entidade apenas se relaciona com uma instância da 1.ª.
- Many-To-Many:
  - Múltiplas para Múltiplas

```
@Entity
public class Owner {
@Id int id;
String name;
@Column(name="PHONE_NUM")
 String phoneNumber;
@OneToOne Address address;
@OneToMany (mappedBy="owner")
List<Pet> pets; ... }

@Entity
@Table(name="PET_INFO")
public class Pet {
@Id
@Column(name="ID")
int licenseNumber;
String name;
PetType type;
@ManyToOne
@JoinColumn(name="OWNER_ID")
Owner owner; ... }
```

Exemplo Modelo de Dados

Um estudante pode frequentar várias disciplinas e uma disciplina pode ser seguida por vários estudantes.



Um Professor lecciona múltiplas disciplinas, mas uma disciplina é leccionada apenas por um Professor.

Uma Disciplina possui uma Página no moodle, e uma Página moodle pertence a uma disciplina.

Exemplo: Mapeamentos One to Many/Many-to-One

```
@Entity
public class Professor
  {
    @Id
    private Long id;
    private String primeiroNome;
    private String ultimoNome;
    ...
    @OneToMany(mappedBy="professor",fetch = FetchType.EAGER)
    private List<Disciplina> disciplinas;
  }

@Entity
public class Disciplina
  {
    @Id
    private Long id;
    private String nome;
    ...
    @ManyToOne
    @JoinColumn(name="PROFESSOR_ID",
     referencedColumnName="ID")
    private Professor professor;
  }
```

Here, the value of **mappedBy** is the name of the association-mapping attribute on the owning side (no caso a disciplina).

Opcional no lado do cliente:
Relacionamento **one-to-many**.
Define uma propriedade do tipo List, Set ou Collection decorada com @OneToMany.
O atributo **mappedBy** é obrigatório no caso de uma associação bidireccional, isto é se queremos oferecer ao professor a lista de disciplinas que ele lecciona. Este terá que corresponder ao atributo professor da Disciplina

JoinColumn: It simply means that our Disciplina entity will have a foreign key column named disciplina_id that references the primary attribute id of our Professor entity.

Exemplo: Mapeamentos One to Many/Many-to-One

- Inserts do Exemplo

```
insert into professor(id, primeiro_nome, ultimo_nome) values(1,'Alexandre','Fonte');
insert into professor(id, primeiro_nome, ultimo_nome) values(2,'Vasco','Soares');

insert into disciplina (id, nome, professor_id) values(1,'Aplicações Distribuídas',1);
insert into disciplina (id, nome, professor_id) values(2,'Arquitectura Internet',1);
insert into disciplina (id, nome, professor_id) values(3,'Redes de Computadores',2);
insert into disciplina (id, nome, professor_id) values(4,'Redes Alargadas',2);
```

Exemplo: Mapeamentos One to Many/Many-to-One

- Comentários *Owning Side* e Bidireccionalidade
  - It's a good practice to put the owning side of a relationship in the class/table **where the foreign key** will be held na base de dados.
  - We keep our **@ManyToOne** mapping on the Disciplina entity. However, we also map a list of Disciplinas to the Professor entity.
  - What's important to note here is the use of the **mappedBy** flag in the **@OneToMany** annotation on the referencing side.
  - Without it, we wouldn't have a two-way relationship. We'd have two one-way relationships. Both entities would be mapping foreign keys for the other entity.
  - With it, we're telling JPA that the field is already mapped by another entity. It's mapped by the Professor field of the Disciplina entity.

Exemplo: Mapeamentos One to Many/Many-to-One

- Comentários Carregamento *Lazy vs Eager*
  - With all our relationships mapped, it's wise to avoid impacting the software's memory by putting too many entities in it if unnecessary.
  - Imagine that Disciplina is a heavy object, and we load all Professor objects from the database for some operation.
  - This can be devastating for the application's performance.
  - Thankfully, **JPA thought ahead and made One-to-Many relationships load lazily by default**. In our example, that would mean until we call on the `Professor#disciplinas` method, the disciplinas are not being fetched from the database.
  - **By contrast, Many-to-One relationships are eager by default**, meaning the relationship is loaded at the same time the entity is.

Exemplo: Mapeamentos One to Many/Many-to-One

- Comentários Optionality
  - A relationship may be optional or mandatory.
  - Considering the *One-to-Many* side - it is always optional, and we can't do anything about it.
  - The *Many-to-One* side, on the other hand, offers us the option of making it *mandatory*.

```
@ManyToOne(optional = false)
@ManyToOne(optional = true)
```

Obrigatório
Opcional

Exemplo: Mapeamentos One to Many/Many-to-One

- Comentários Optionality

```
@ManyToOne(optional = false)
```

```
insert into professor(id, primeiro_nome, ultimo_nome) values(1,'Alexandre','Fonte');
insert into professor(id, primeiro_nome, ultimo_nome) values(2,'Vasco','Soares');

insert into disciplina (id, nome) values(1,'Aplicações Distribuídas');
insert into disciplina (id, nome) values(2,'Arquitectura Internet');
insert into disciplina (id, nome) values(3,'Redes de Computadores');
insert into disciplina (id, nome) values(4,'Redes Alargadas');
```

```
insert into disciplina (id, nome) values(1,'Aplicações Distribuídas');
```

```
NULL not allowed for column "PROFESSOR_ID"; SQL statement:
insert into disciplina (id, nome) values(1,'Aplicações Distribuídas') [23502-214] 23502/23502 (Help)
```

```
insert into disciplina (id, nome) values(2,'Arquitectura Internet');
```

```
NULL not allowed for column "PROFESSOR_ID"; SQL statement:
insert into disciplina (id, nome) values(2,'Arquitectura Internet') [23502-214] 23502/23502 (Help)
```

```
insert into disciplina (id, nome) values(3,'Redes de Computadores');
```

```
NULL not allowed for column "PROFESSOR_ID"; SQL statement:
insert into disciplina (id, nome) values(3,'Redes de Computadores') [23502-214] 23502/23502 (Help)
```

Exemplo: Mapeamentos One-to-One

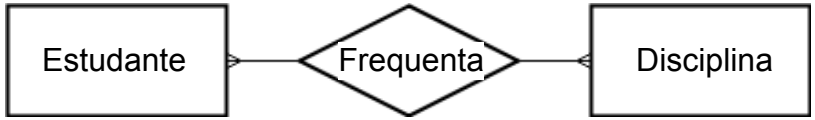
```
@Entity
public class Disciplina
  {
    @Id
    private int id;
    private String nome;
    ...
    @ManyToOne
    @JoinColumn(name="DISCIPLINA_ID",
     referencedColumnName="ID")
    private Professor professor;

    @OneToOne(mappedBy = "disciplina")
    private MaterialDisciplina material;
  }

@Entity
public class MaterialDisciplina
  {
    @Id
    private int id;
    private String urlMoodle;
    ...
   @OneToOne(optional = false)
   @JoinColumn(name = "DISCIPLINA_ID", referencedColumnName = "ID")
   private Disciplina disciplina;
  }
```

Here, the value of **mappedBy** is the name of the association-mapping attribute on the owning side.

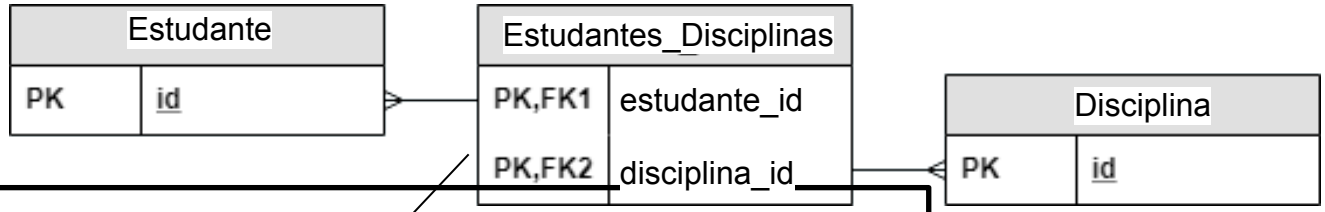
JoinColumn: It simply means that our MaterialDisciplina entity will have a foreign key column named disciplina_id that references the primary attribute id of our Disciplina entity.

Exemplo: Many-to-Many

- Neste caso ambos os lados podem relacionar-se com múltiplas instâncias do outro lado.
- Um estudante pode frequentar muitas disciplinas e Muitos estudantes podem frequentar a mesma disciplina.



- A criação das relações é com as chaves forasteiras. Neste caso é preciso criar uma **@Join table** referenciando ambos os lados para criar a Tabela que representa o relacionamento many-to-many. **JoinColumns** configura a(s) FK do owning side. **@InverseJoinColumns** as do lado de referenciação.




```
@Entity
class Disciplina {
    @Id
    Long id;
    @ManyToMany
    @JoinTable(
        name = "estudantes_disciplinas",
        joinColumns = @JoinColumn(name = "estudante_id"),
            inverseJoinColumns = @JoinColumn(name =
     "disciplina_id"))
    List<Estudante> estudantes;
    // additional properties
    // standard constructors, getters, and setters
}
```

Many-to-Many relationships are lazy by default.

```
@Entity
class Estudante {
    @Id
    Long id;
    @ManyToMany(mappedby="estudante
s"
    List<Disciplina> disciplinas;
    // additional properties
    // standard constructors,
getters, and setters
}
```

Java Persistence Query Language (JPQL)

Java Persistence Query Language (JPQL)

- Linguagem definida pela JPA para suportar **queries baseadas em entidades**
- Partilha muitos aspetos comuns ao SQL, com a diferença de que as estruturas principais são entidades e campos, em vez das tabelas e colunas das BDs
- As **queries JPQL** podem incluir parâmetros de entrada e retornam entidades ou objetos Java

Exemplo Query Simples JPQL

```
"SELECT o FROM Cliente o"
```

Java Persistence Query Language (JPQL)
Exemplos

Exemplo Query Simples

```
"SELECT o FROM Cliente o"
```

Exemplo Queries Simples usando parâmetros Named

Parâmetro Named
Sintaxe :<nome>

```
"SELECT o FROM Inventario o WHERE o.ano=:ano"
"SELECT o FROM Inventario o WHERE o.ano=:ano AND o.mes=:mes"

"SELECT o FROM Cliente o WHERE o.nome like '%:nome%'"
```

Exemplo Query Simples com 2 parâmetros Posicionais

```
"SELECT p FROM Jogador p WHERE p.idade BETWEEN ?1 AND ?2"
```

Exemplo Query Complexa

```
"SELECT o.id, o.quantity, o.item, " + "i.id, i.name,
i.description "
+ "FROM Order o, Item i "
+ "WHERE (o.quantity > 25) AND (o.item = i.id)"
```

Parâmetro Posicional
Sintaxe ?1 ?2, etc

Spring Data

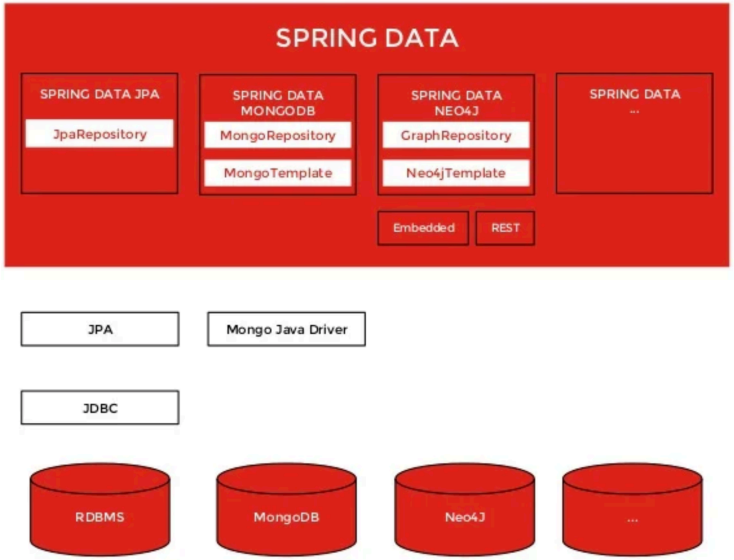
Spring Data: O que é?

- Mundo dos Dados em crescimento
  - -Necessidade de novos mecanismos de processamento
    - •Por exemplo, as Base de dados relacionais não escalam
  - -Necessidade de mecanismos especializados
    - •Suportam casos específicos de uso ou tipos de dados específicos
  - -A desvantagem: o aparecimento de várias tecnologias de armazenamento trouxe diferentes APIs, diferentes configurações, e diferentes métodos
    - •JDBC e JPA são standard mas específicos de bases de dados SQL
    - •No caso das bases de dados No-Sql, "não há" formas standard…
- O projecto Spring Data "aims at providing a consistent data access layer for various data-stores, right from relational to no-sql databases".
  - -Não é uma ESPECIFICAÇÃO ou Norma
  - -É uma abstração

A persistencia Poliglota tem-se tornado prevalente (SQL, NoSQL, Big Data)

Módulos Spring Data

- *"The Spring-Data is an umbrella project having number of sub-projects or modules all aimed towards providing uniform abstractions and uniform utility methods for the Data Access Layer in an application and support wide range of databases and datastores."*
- Consultar: *https://spring.io/projects/spring-data*
  - Os principais módulos são:
  - spring-data-commons
  - spring-data-jpa
  - spring-data-ldap
  - spring-data-mongo
  - Outros: spring-data-azure-cosmos-db



Spring Data
Principais Interfaces (1) - Interface Genérica

- Repository
  - Principal interface de abstração Spring Data.
  - Permite ao Spring poder obter o tipo de Domínio/Modelo (T) e o tipo do seu ID com que o *developer* irá trabalhar e para o Spring ajudar a descobrir outras interfaces que estendem esta interface.
  - As interfaces que estendem Repository, podem expor os métodos CRUD declarando os métodos com a mesma assinatura dos declarados na CrudRepository (ver slide seguinte)

```
public interface Repository<T,ID extends Serializable> {}
```

Spring Data
Principais Interfaces (1) - Interface Genérica

- CrudRepository

  -Interface que disponibiliza as operações CRUD (Create, Read, Update e Delete) **genéricas** sobre a entidade que está a ser gerida.

  -Estende a interface Repository. Recebe o tipo da entidade e o tipo do campo id da entidade.

```
public interface CrudRepository<T,ID extends
Serializable> extends Repository {
<S extends T> S save(S entity);
T findOne(ID primaryKey);
Iterable<T> findAll();
Long count();
void delete(T entity);
Boolean exists(ID primaryKey);
…}
```

1. Saves the given entity.
2. Returns the entity identified by the given ID.
3. Returns all entities.
4. Returns the number of entities.
5. Deletes the given entity.
6. Indicates whether an entity with the given ID exists.

Disponibiliza
outros métodos além destes.

Spring Data
Principais Interfaces (1) - Interface Genérica

- Paginação e Ordenação

  -Estende a CrudRepository para oferecer as operações de paginação.

```
public interface PagingAndSortingRepository<T,ID extends
Serializable> extends CrudRepository {
Iterable<T> findAll(Sort sort);
Page<T> findAll(Pageable pageable);
}
```

Exemplo de Utilização:

```
PagingAndSortingRepository<Utilizador,Long> repository=//… get access to a
bean
Page<Utilizador> utilizadores=repository.findAll(PageRequest.of(1,20));
```

No exemplo, é retornada uma página com 20 Users.

Spring Data
Principais Interfaces (2) - Interfaces Específicas

- JpaRepository
  - JPA specific extension of Repository.
  - Consultar: https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html
- MongoRepository
  - Mongo specific Repository interface.
  - Consultar: https://docs.spring.io/spring-data/mongodb/docs/current/api/org/springframework/data/mongodb/repository/MongoRepository.html

Introdução ao Spring Data JPA
Repositórios de Dados

- Abstrai a camada de acesso aos dados (DAL - Data Access Layer)
- Usa o conceito de Repositórios JPA e CRUD
  - Conjunto de Interfaces que definem métodos ***query***
  - ***"No need to write native queries anymore!"*** *No limite não precisamos de implementar a camada de acesso ou de escrever SQL. Apenas de declarar os métodos!*

```
@Repository
public interface EmployeeRepository extends CrudRepository<Employee, Long> {
  List<Employee> findAll();
  Optional<Employee> findById(Long employeeId);
  List<Employee> findByFirstNameAndLastName(String firstName, String
lastName);
  List<Employee> findByDepartmentNameAndAgeLessThanOrderByDateOfJoiningDesc(
String departmentName, int maxAge);
}
```

Spring Data JPA
Métodos Query

- O nome dos métodos declarados nas interfaces repositório são convertidos pelo Spring para “low-level SQL”.
- Os gestores de entidade continuam a existir, mas em background…
- As interfaces repositório estendem as interfaces **CrudRepository** ou **JpaRepository** ou **PagingAndSortingRepository**

```
@Repository
public interface ContaRepositorio
                extends CrudRepository<Conta, Long> {
}
```

Where:

- **T:** Domain type that repository manages (Generally the Entity/Model class name)
- **ID:** Type of the id of the entity that repository manages (Generally the wrapper class of your @Id that is created inside the Entity/Model class)

Spring Data JPA
*Métodos Query - **Queries automáticas personalizadas** criadas a partir dos nomes dos métodos!*

- Quando o Spring Data cria um novo Repositório analisa todos os métodos declarados e tenta automaticamente gerar queries a partir dos nomes dos métodos

- Já vimos que a lista completa de palavras-chave está disponível em:

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

```
@Repository
public interface ContaRepositorio extends
CrudRepository<Conta, Long> {
    public Conta findById(long id);
    public Conta findByTitular(String titular);
}
```

Queries geradas pelo Spring

```
SELECT c from Conta WHERE u.Id = ?1
SELECT c from Conta WHERE u.Titular = ?1
```

Spring Data JPA

Métodos Query - Queries personalizadas usando a anotação ***@Query***

- Por omissão usa JPQL

```
@Query("SELECT u FROM User u WHERE u.status = 1")
Collection<User> findAllActiveUsers();
```

- Também pode ser usado SQL nativo, colocando o atributo **`nativeQuery=true`**

```
@Query(
  value = "SELECT * FROM USERS u WHERE u.status = 1",
  nativeQuery = true)
Collection<User> findAllActiveUsersNative();
```

Créditos:
https://www.baeldung.com/spring-data-jpa-query

Spring Data JPA
Métodos Query - Queries personalizadas usando a anotação ***@Query***

- Declaração da Ordenação numa Query

  *-We can pass an additional parameter of type **Sort** to a Spring Data method declaration that has the **@Query** annotation. It'll be translated into the ORDER BY clause that gets passed to the database.*

No caso de Queries nativas não é possível usar um parâmetro Sort.

```
@Query(value = "SELECT u FROM User u")
List<User> findAllUsers(Sort sort);

//Ordenação pelo campo “nome” direção ascendente
userRepository.findAllUsers(Sort.by("name"));
```

Alternativamente, usar a cláusula ORDER BY diretamente na Query

```
@Query(value = "SELECT u FROM User u ORDER BY Id")
List<User> findAllUsers();
```

Spring Data JPA
Métodos Query - Queries personalizadas usando a anotação ***@Query***

- Paginação

  -Pagination allows us to return just a subset of a whole result in a Page. This is useful, for example, when navigating through several pages of data on a web page.

  -Another advantage of pagination is that the amount of data sent from server to client is minimized. By sending smaller pieces of data, we can generally see an improvement in performance.

  - ***Em JPQL é direto:***

```
@Query(value = "SELECT u FROM User u ORDER BY id")
Page<User> findAllUsersWithPagination(Pageable pageable);
```

  - ***Em SQL nativo é preciso declarar um atributo countQuery***

```
@Query(
  value = "SELECT * FROM Users ORDER BY id",
  countQuery = "SELECT count(*) FROM Users",
  nativeQuery = true)
Page<User> findAllUsersWithPagination(Pageable pageable);
```

Spring Data JPA

Métodos Query - Queries personalizadas usando a anotação ***@Query***

- Declaração da Ordenação numa **Query Nativa não é suportada**

Spring Data JPA does not currently support dynamic sorting for native queries, because it would have to manipulate the actual query declared, which it cannot do reliably for native SQL. You can, however, use native queries for pagination by specifying the count query yourself, as shown in the following example:

```
public interface UserRepository extends JpaRepository<User, Long> {

  @Query(value = "SELECT * FROM USERS WHERE LASTNAME = ?1",
    countQuery = "SELECT count(*) FROM USERS WHERE LASTNAME = ?1",
    nativeQuery = true)
  Page<User> findByLastname(String lastname, Pageable pageable);
}
```

Como usar:

```
Pageable firstPageWithTwoElements = PageRequest.of(0, 2);
Pageable secondPageWithFiveElements = PageRequest.of(1, 5);
Page<User> allUsers = userRepository.findByLastname("Fonte",
firstPageWithTwoElements);
```

Ref: https://docs.spring.io/spring-data/data-jpa/docs/current/reference/html/#jpa.query-methods.at-query

Spring Data JPA
Métodos Query - Queries personalizadas usando a anotação ***@Query***

- Parametrização das Queries

  -There are two possible ways that we can pass method parameters to our query: indexed and named parameters.

  -In this section, we'll cover indexed parameters.

  - ***Parâmetros Posicionais (ordem dos parâmetros no método é a mesma dos da query):***

```
@Query("SELECT u FROM User u WHERE u.status = ?1 and u.name = ?2")
User findUserByStatusAndName(Integer status, String name);
```

```
@Query(
  value = "SELECT * FROM Users u WHERE u.status = ?1",
  nativeQuery = true)
User findUserByStatusNative(Integer status);
```

Este é método por omissão usado pelo Spring Data JPA.

Spring Data JPA

Métodos Query - Queries personalizadas usando a anotação ***@Query***

- Parametrização das Queries (Cont.)

  -There are two possible ways that we can pass method parameters to our query: indexed and named parameters.

  -In this section, we'll cover indexed parameters.

  - ***Parâmetros nomeados (usa-se a anotação @Param)***

```
@Query("SELECT u FROM User u WHERE u.status = :status and u.name = :name")
User findUserByUserStatusAndUserName(@Param("status") Integer userStatus,
  @Param("name") String userName);
```

Spring Data JPA

Métodos Query - Queries personalizadas usando a anotação ***@Query com @Modifying***

- **@Modifying** is used to enhance the **@Query** annotation so that we can execute not only SELECT queries, but also **INSERT, UPDATE, DELETE**.

Spring Data JPA

Métodos Query - Queries personalizadas usando a anotação ***@Query com @Modifying***

- Atualizações de Dados

  -The repository method that modifies the data has two differences in comparison to the select query — it has the @Modifying annotation and, of course, the JPQL query uses update instead of select

```
@Modifying
@Query("update User u set u.status = :status where u.name = :name")
int updateUserSetStatusForName(@Param("status") Integer status,
  @Param("name") String name);
```

```
@Modifying
@Query(value = "update Users u set u.status = ? where u.name = ?",
  nativeQuery = true)
int updateUserSetStatusForNameNative(Integer status, String name);
```

Nota: @Modifying poderá necessitar de ser parametrizado com o atributo **@Modifying(clearAutomatically = true)** para limpar automaticamente o contexto de persistência após a query ser executada.

Spring Data JPA

Métodos Query - Queries personalizadas usando a anotação ***@Query com @Modifying***

- Inserções de Dados

  -To perform an insert operation, we have to both apply @Modifying and use a native query since INSERT is not a part of the JPA interface:

Insert existe apenas nas Queries nativas

```
@Modifying
@Query(
  value =
    "insert into Users (name, age, email, status)
     values (:name, :age, :email, :status)", nativeQuery = true)
void insertUser(@Param("name") String name, @Param("age") Integer age,
  @Param("status") Integer status, @Param("email") String email);
```

- Eliminação de Dados também exige @Modifying

```
@Modifying
@Query(
  value =
    "DELETE FROM Users u WHERE u.name=:name", nativeQuery = true)
void deleteUser(@Param("name") String name);
```

A minha segunda Aplicação Spring Boot com Spring Data JPA

- **<u>Atividade Prática n.º5:</u>** Criar uma aplicação multinível chamada **AppVendas-SpringData-Jpa** novamente com Spring Data Jpa que usa a base de dados em memória H2.

A minha primeira Aplicação Spring Boot com Spring Data JPA

- **Passo 1:** Abra o ficheiro application.properties e inclua as seguintes propriedades para criar uma base de dados
  - Nome: testedb
  - Utilizador: admin
  - Password: admin
- No final ative a consola da base de dados H2.

```
spring.datasource.url=jdbc:h2:mem:testedb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=admin
spring.datasource.password=admin
spring.datasource.initialization-mode=always
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
```

Execute a aplicação e aceda à consola da H2: http://localhost:8080/h2-console. Respeite o nome da datasource.

A minha segunda Aplicação Spring Boot com Spring Data JPA

Passo 2: Considere o seguinte modelo Físico de dados.

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

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

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

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

Como vimos antes podemos delegar no Spring Boot para criar o Modelo a partir das classes entidade em Java.

Alternativamente, podemos também criar um script SQL na pasta de recursos que será executado durante o start-up da App.

Posteriormente, será necessário realizar manualmente o mapeamento JPA.

A minha segunda Aplicação Spring Boot com Spring Data JPA

- **Passo 3:** Na pasta resources, crie um novo ficheiro **schema.sql** com as expressões SQL. Execute a aplicação.
  - Terá ocorrido um erro "Produto" not found :-)
  - Corrija o SQL.
- **Passo 4:** Aceda à Consola da H2 e verifique se as tabelas foram criadas.
- **Passo 5:** Crie as Classes Modelo Cliente, Pedido, Produto e ItemProduto
- **Passo 6:** Faça o mapeamento entre as Entidades Cliente e Pedido usando as anotações JPA estudadas:
  - @Entity, @Table, @Id, @Column
  - @ManyToOne, @JoinColumn, @OneToMany,

A minha segunda Aplicação Spring Boot com Spring Data JPA

- **Passo 6** - Resolução parcial

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

    //Setters e Getters
}
```

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

    //Setters e Getters
}
```

Opcional no lado do cliente:
Relacionamento **one-to-many**.
Define uma propriedade do tipo List, Set ou Collection decorada com @OneToMany.
O atributo mappedBy é obrigatório no caso de uma associação bidireccional.

Obrigatório no lado do pedido:
Relacionamento **many-to-one**.
1) Em vez de Integer cliente_id, a JPA para criar o relacionamento entre Objectos precisa de uma variável do tipo Cliente.
2) Além disso, em vez de @Column usa-a a anotação @JoinColumn para indicarmos o nome da coluna da tabela correspondente à Chave forasteira.

A minha segunda Aplicação Spring Boot com Spring Data JPA

- **Passo 7:** Criar os Repositórios das Entidades Cliente e Pedido do tipo JpaRepository: `RepositorioCliente` e `RepositorioPedido`.
- **Passo 8:** No método run() de um ApplicationRunner insira dois registos Cliente e dois registos Pedido.
- **Passo 9:** Crie as classes Entidade Produto e PedidoItem e faça também o mapeamento JPA Pedidos e Produtos. De seguida, declare as interfaces Repositório.

A minha segunda Aplicação Spring Boot com Spring Data JPA

- **Passo 9** - Resolução parcial

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

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

A minha segunda Aplicação Spring Boot com Spring Data JPA

- **Passo 10:** Pretende-se que ensaie a criação de queries personalizadas usando a anotação @Query.
  - Usando uma query JPQL consulte todos os clientes e ordene pelo seu Id.
  - Usando uma query JPQL consulte os clientes pelo nome usando a cláusula LIKE
  - Repita, agora usando uma query Nativa, consulte os clientes pelo nome usando a cláusula LIKE
  - Usando uma query Nativa insira um novo Pedido.
  - Usando uma query JPQL apague um cliente pelo seu nome.

Precisa de declarar as queries e especificar
os métodos query nos repositórios.
Ensaie as chamadas dos métodos no Application Runner.
Ou desenvolva uma controlador REST que usa os repositórios.

A minha segunda Aplicação Spring Boot com Spring Data JPA

- Nos passos seguintes pretende-se que faça **uma consulta com os relacionamentos JPA**, designadamente quando consultar o registo de um cliente, a Lista dos pedidos de um cliente também sejam carregados da Base de dados.
- Existem duas formas de realizar isto:
  - 1. Carregar os pedidos conjuntamente com os restantes campo (i.e. eagerly). **QUAL é a desvantagem desta FORMA?**
  - 2. Carregar on-demand (i.e. lazily) apenas quando chamamos um método subsequente.
- Usamos o atributo **fetch** (trazer junto) das anotações @OneToMany ou @ManyToOne para se definir a forma escolhida.
  - Exemplos:

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

A minha segunda Aplicação Spring Boot com Spring Data JPA

Ensaio Fetch EAGER

- **Passo 11** - Em adição ao passo 6, declare o mapeamento também na Entidade Cliente e defina o fetch como EAGER.

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

- **Passo 12:** Ensaie o método query findById() e verifique o conteúdo da Lista de pedidos carregada juntamente.
- **Passo 13:** Pode também fazer o fetch do cliente quando consulta por um pedido:

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

A minha segunda Aplicação Spring Boot com Spring Data JPA

Ensaio Fetch LAZY

- **Passo 14** - Em adição ao passo 6, declare o fetch como LAZY.

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

- **Passo 15:** Ensaie o método query findById() e verifique o conteúdo da Lista de pedidos. **OPS!!! Erro!**
- Neste caso precisa de fazer uma consulta on-demand subsequente para que os pedidos sejam carregados.
- **Passo 16**: O fetch tem que ser realizado num pedido subsequente à parte usando uma Query apropriada:

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

Questões