

# Desenvolvimento de Software para Web



UFC - Universidade Federal do Ceará

André Meireles  
[andre@crateus.ufc.br](mailto:andre@crateus.ufc.br)

# JPA com Spring Data

# ORM , JPA e Hibernate

## **ORM - Object-Relational Mapping**

É um padrão criado para permitir especificar como o as classes e seus relacionamentos em uma aplicação orientada a objetos deve ser mapeados para um modelo relacional

## **JPA - Java Persistence API**

É o conjunto de interfaces Java que especifica um padrão para implementação de ORM em Java

## **Hibernate**

É a implementação de JPA mais popular do mundo, ela dá suporte a diversos servidores de banco de dados

# Executando o servidor o MySQL

Para executar um servidor MySQL utilizando Docker:

```
# docker run --name xampp-fbd -p 8081:80 -p 3306:3306 -d tomsik68/xampp
```

Acesse o PHPMyAdmin pelo navegador no endereço <http://localhost:8081/phpmyadmin/>

Na aba SQL do PHPMyAdmin, execute esse comando para permitir o acesso remoto ao usuário desejado:

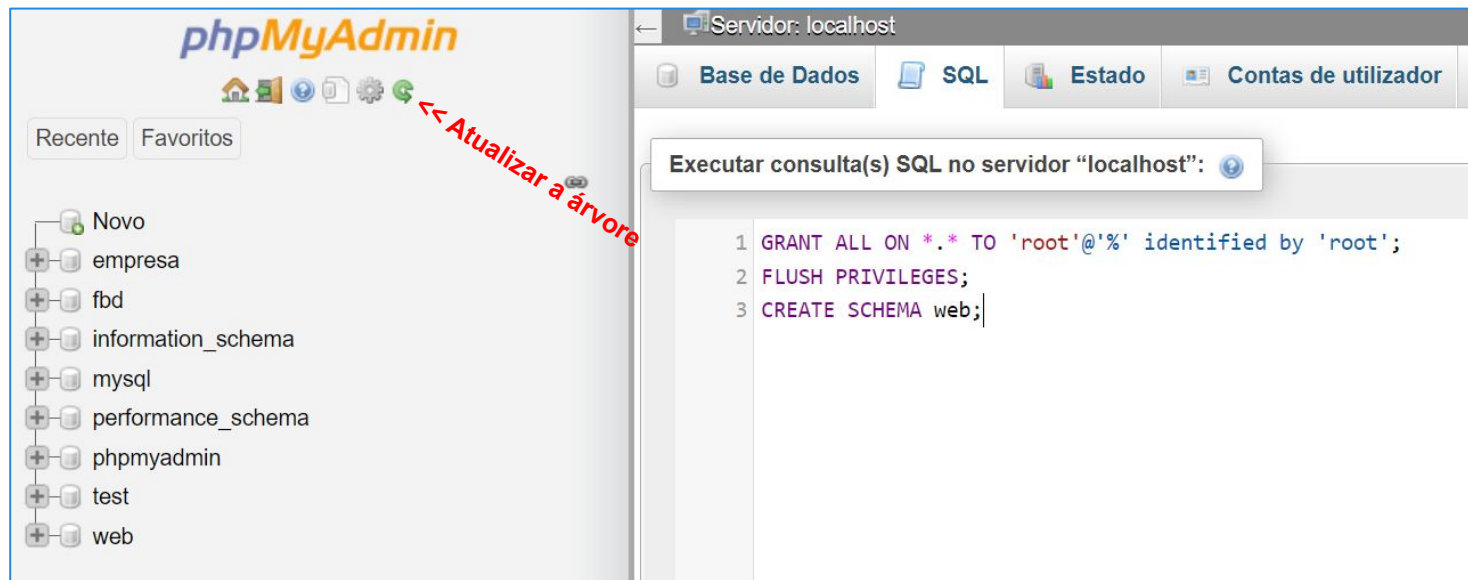
```
GRANT ALL ON *.* TO 'root'@'%' identified by 'root';  
FLUSH PRIVILEGES;
```

Crie o esquema que você utilizará para a aplicação:

```
CREATE SCHEMA web;
```

# Executando o servidor o MySQL

Após executar os comandos, atualiza a árvore e verifique se o esquema foi criado:



The screenshot displays the phpMyAdmin web interface. On the left, the 'Servidor: localhost' tree shows a list of databases: 'Novo', 'empresa', 'fbd', 'information\_schema', 'mysql', 'performance\_schema', 'phpmyadmin', 'test', and 'web'. A red arrow points to the 'Atualizar a árvore' button. The right pane shows the 'SQL' tab with the following commands executed:

```
1 GRANT ALL ON *.* TO 'root'@'%' identified by 'root';
2 FLUSH PRIVILEGES;
3 CREATE SCHEMA web;
```

The output of the execution is visible below the SQL text.

# Dependências requeridas: pom.xml

Adicione as seguintes dependências ao pom.xml do seu projeto.

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

# Configuração do application.properties

O arquivo `/src/main/resources/application.properties` deve ser utilizado para definir propriedades da aplicação, é nele onde as credenciais de acesso ao banco de dados devem ser informadas da seguinte forma:

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/web
spring.datasource.username=root
spring.datasource.password=root
```

# Mapeamento básico de Entidades

Para que as classes sejam mapeadas como tabelas do banco relacional, é preciso utilizar algumas anotações, veja a seguir:

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity << Declara a entidade, ela será mapeada como uma tabela no BD
public class User {

    @Id << Informa a chave primária da entidade
    @GeneratedValue << Define que os valores da chave devem ser automaticamente controlados pelo Hibernate
    private int id;
    private String name;
    private String password;

    ...
}
```



# Spring Repository

Para realizar operações de CRUD (Create, Read, Update e Delete) é preciso definir um repositório para a entidade:

```
import org.springframework.data.repository.CrudRepository;
```

```
public interface UserRepository extends CrudRepository<User, Integer> {  
  
}
```

<< Declara o repositório de User

<< Classe do atributo ID de User  
<< Classe da entidade do repositório (User)

# Acessando o Repository

Como o repositório é um Component, você pode acessá-los usando `@Autowired` a partir de quaisquer `Service` ou `Controller` da sua aplicação

```
@RestController
@RequestMapping("/api/user")
public class UserRestController {

    @Autowired << O Spring injeta o repositório neste objeto
    UserRepository userRepository;

    @GetMapping
    Iterable<User> getUsers() {
        return userRepository.findAll(); << Utilize as operações de repositório para recuperar objetos do BD
    }

    @PostMapping
    User addUser(@RequestBody User user) {
        return userRepository.save(user); << Utilize as operações de repositório para salvar objetos no BD
    }
}
```

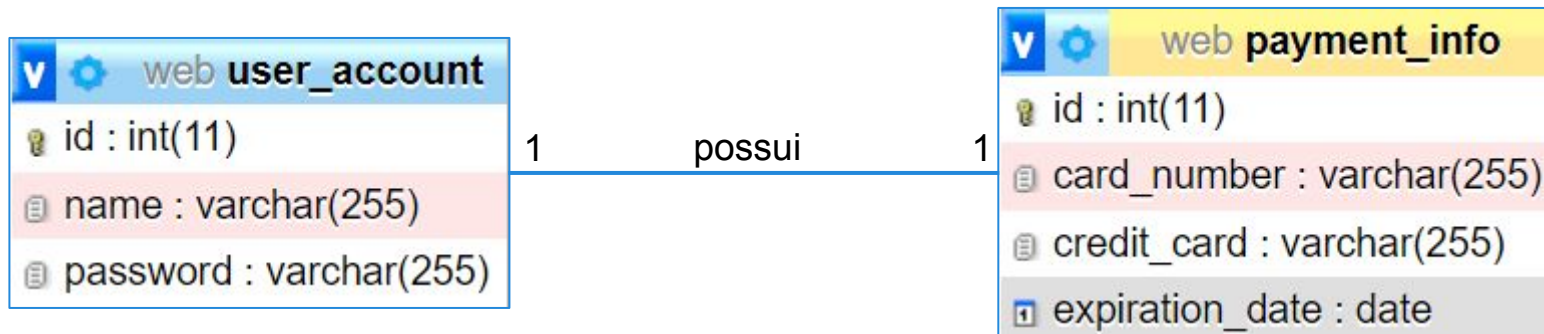
# JPA - Relacionamientos entre entidades

## JPA Relationship Types

- **OneToOne** - A unique reference from one object to another, inverse of a OneToOne.
- **ManyToOne** - A reference from one object to another, inverse of a OneToMany.
- **OneToMany** - A Collection or Map of objects, inverse of a ManyToOne.
- **ManyToMany** - A Collection or Map of objects, inverse of a ManyToMany.
- **Embedded** - A reference to a object that shares the same table of the parent.
- **ElementCollection** - JPA 2.0, a Collection or Map of Basic or Embeddable objects, stored in a separate table

# JPA - Relacionamento 1x1

Suponha o seguinte relacionamento para representar que um usuário possui dados de pagamento e aquele dado de pagamento pertence apenas àquele usuário:




# JPA - Relacionamento 1x1

Utilize a anotação **@OneToOne** para implementar relacionamentos 1x1


```
@Entity
public class UserAccount {

    @Id
    @GeneratedValue
    private int id;
    private String name;
    private String password;
    @OneToOne
    private PaymentInfo paymentInfo;
```



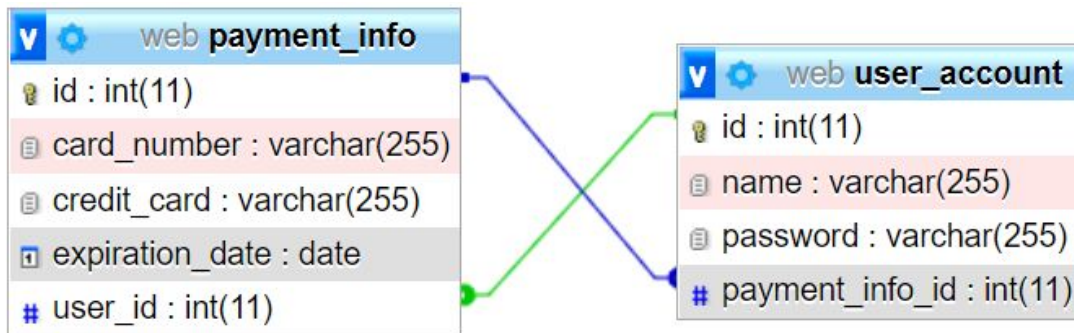
```
@Entity
public class PaymentInfo {

    @Id
    @GeneratedValue
    private int id;
    @Enumerated(EnumType.STRING)
    private CreditCard creditCard;
    private String cardNumber;
    @Temporal(TemporalType.DATE)
    private Date expirationDate;
    @OneToOne
    private UserAccount user;
```



# JPA - Relacionamento 1x1

Porém, se a anotação for colocada nas duas entidades, a chave estrangeira será criada nas duas tabelas.



Essa solução está **ERRADA**.

Como sabemos, nos relacionamentos 1x1, a chave estrangeira deve ficar apenas na entidade mais fraca.

# JPA - Relacionamento 1x1

Para evitar a criação das chaves estrangeiras nas duas tabelas, você pode **remover o atributo de uma das tabelas** ou utilizar **MappedBy** para indicar que a chave estrangeira será definida por outra entidade

```
@Entity
public class UserAccount {

    @Id
    @GeneratedValue
    private int id;
    private String name;
    private String password;
    @OneToOne (mappedBy = "user")
    private PaymentInfo paymentInfo;
```

```
@Entity
public class PaymentInfo {

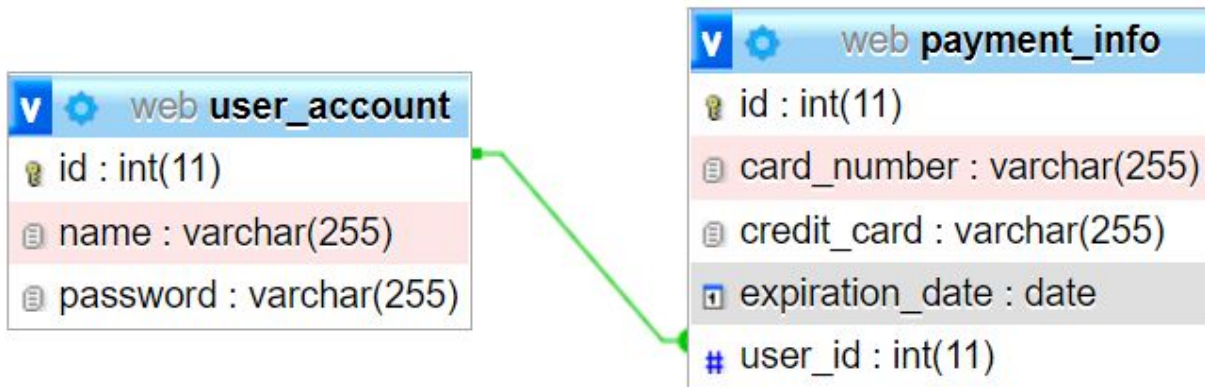
    @Id
    @GeneratedValue
    private int id;
    @Enumerated(EnumType.STRING)
    private CreditCard creditCard;
    private String cardNumber;
    @Temporal(TemporalType.DATE)
    private Date expirationDate;
    @OneToOne
    private UserAccount user;
```

Indica que a chave será mapeada pelo atributo user na outra entidade

# JPA - Relacionamento 1x1

Desta forma, apenas a tabela **payment\_info** terá chave estrangeira para **user\_account**.

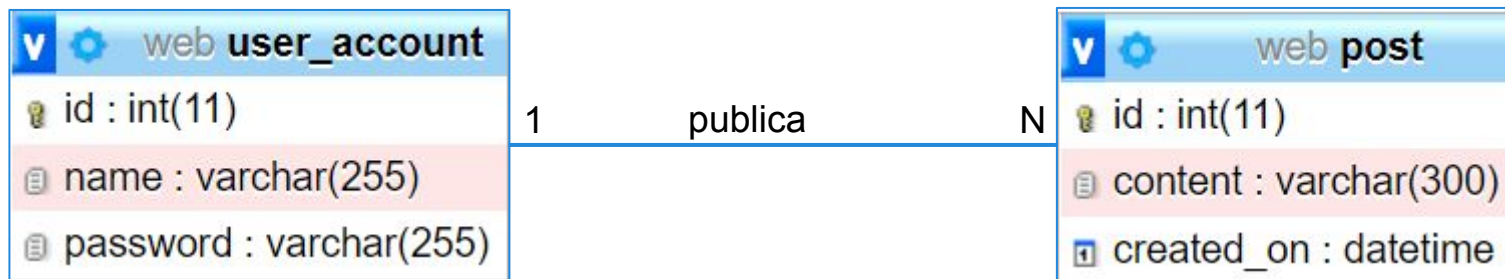
Como o **UserAccount** ainda possui o atributo **paymentInfo**, ao recuperar um objeto **UserAccount**, o atributo **paymentInfo** estará automaticamente acessível.





# JPA - Relacionamento 1xN ou Nx1


Suponha o seguinte relacionamento para representar que um **usuário** publica vários **posts** e que cada post pertence a um único usuário:



# JPA - Relacionamento 1xN ou Nx1

Utilize a anotação **@ManyToOne** ou **@OneToMany** para implementar relacionamentos 1 para muitos

Você deve colocar a anotação **@ManyToOne** na entidade que representa lado N da relação, no atributo que referencia a entidade que representa o lado 1



```
@Entity
public class Post {

    @Id
    @GeneratedValue
    private int id;

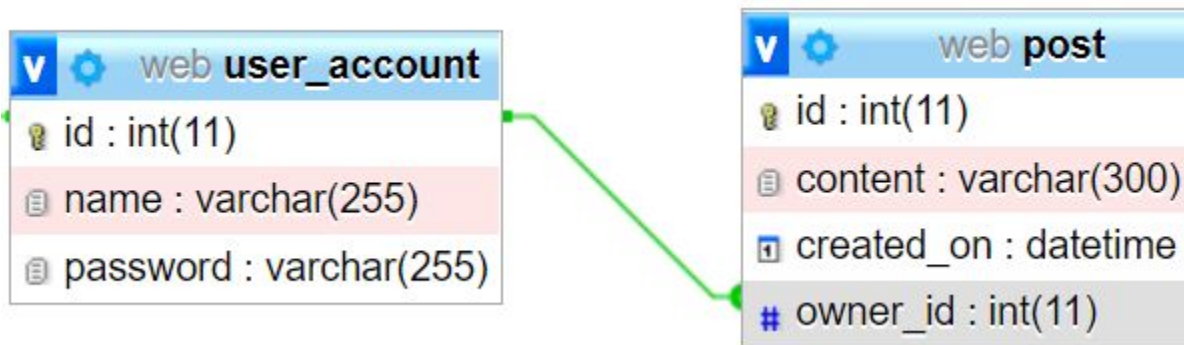
    @ManyToOne
    private UserAccount owner;

    @Column(length = 300)
    private String content;

    @Temporal(TemporalType.TIMESTAMP)
    private Date createdOn;
```

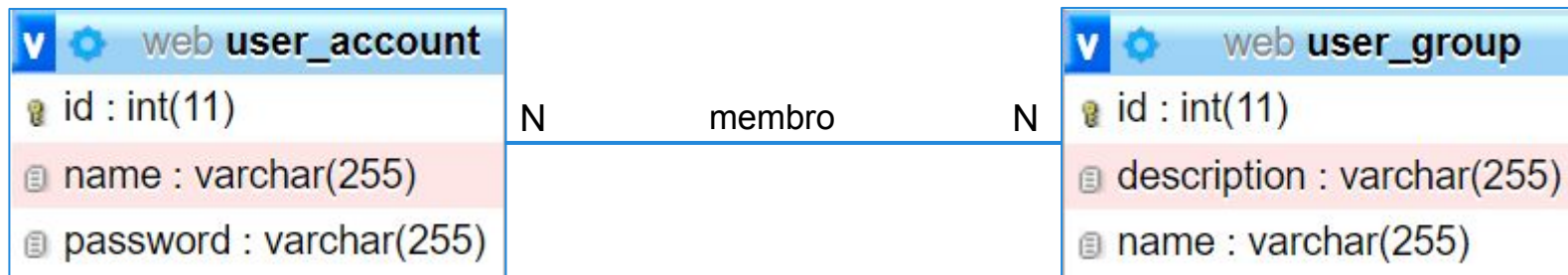
# JPA - Relacionamento 1xN ou Nx1

Desta forma, a tabela **post** terá a chave estrangeira para **user\_account**.



# JPA - Relacionamento NxN

Suponha o seguinte relacionamento para representar que um **usuário** pode ser **membro** de vários **grupos** e que cada **grupo** pode ter vários membros




# JPA - Relacionamento NxN

Utilize uma coleção (ex.: List) e a anotação **@ManyToMany** para implementar relacionamentos muitos para muitos

Você deve colocar a anotação **@ManyToMany** na entidade que deseja utilizar para acessar a lista de objetos da outra entidade.

Se desejar ter a lista de objetos nas duas entidades, uma delas deve ter **mappedBy** para evitar que o Hibernate crie duas tabelas de relacionamento



```
@Entity
public class UserGroup {

    @Id
    @GeneratedValue
    private int id;

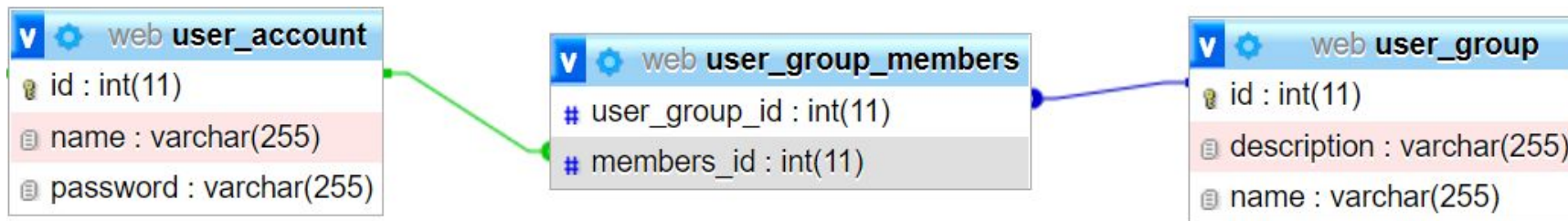
    @ManyToMany
    private List<UserAccount> members;

    private String name;

    private String description;
```

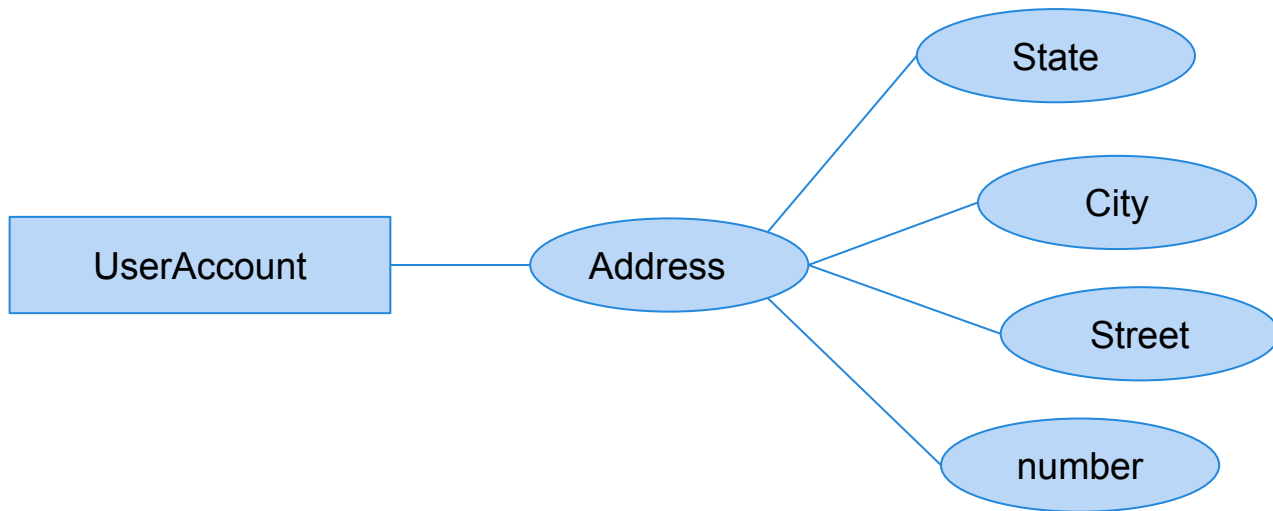
# JPA - Relacionamento NxN

Desta forma, será criada uma tabela de relacionamento com as chaves estrangeiras para duas tabelas do relacionamento.



# JPA - Atributo Composto

Suponha uma entidade `UserAccount` que possui um atributo da classe `Address`, e que a classe `Address` é composta por vários atributos primitivos:




# JPA - Atributo Composto

Utilize as anotações **@Embeddable** e **@Embedded** para modelar um atributo composto

```
@Embeddable
public class Address {


    @Column(length = 2)
    private String state;
    private String city;
    private String streetName;
    @Column(length = 10)
    private String number;
```



Declara que a classe pode ser um atributo composto

```
@Entity
public class UserAccount {

    @Id
    @GeneratedValue
    private int id;
    private String name;
    private String password;
    @OneToOne (mappedBy = "user")
    private PaymentInfo paymentInfo;
    @Embedded
    private Address address;
```

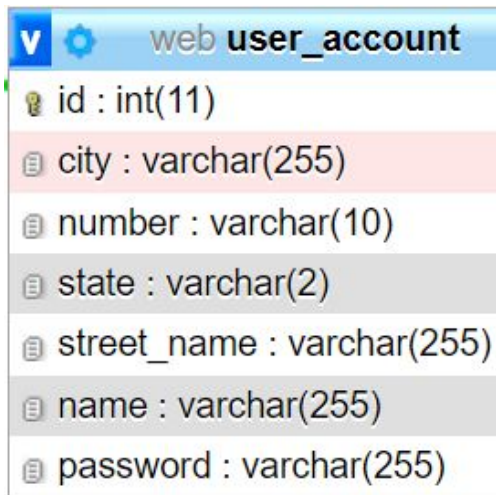


Define **address** como atributo composto de **UserAccount**



# JPA - Atributo Composto

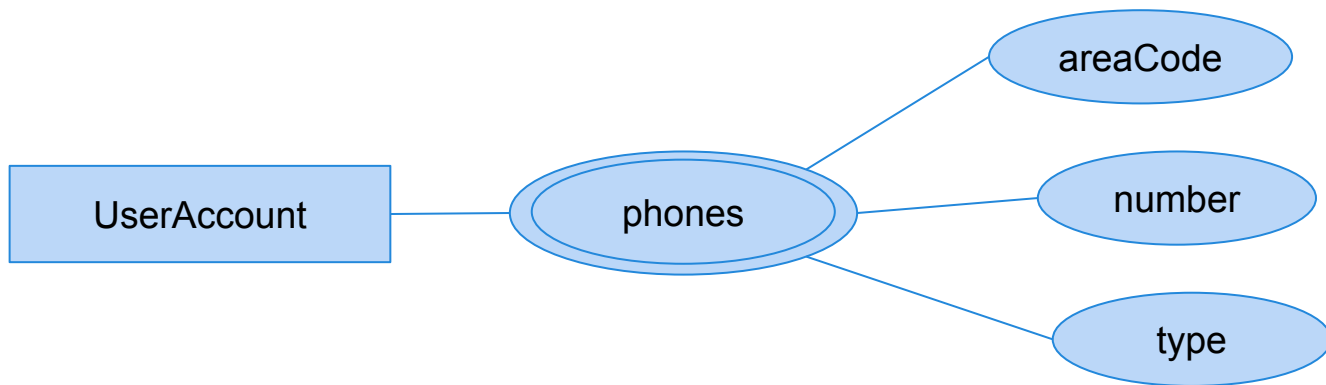
Desta forma, os campos da entidade **Address** ficarão na tabela **user\_account**



web user_account	
id	int(11)
city	varchar(255)
number	varchar(10)
state	varchar(2)
street_name	varchar(255)
name	varchar(255)
password	varchar(255)

# JPA - Atributo Multivalorado

Suponha uma entidade **UserAccount** que possui um atributo **phones**, que é uma coleção de objetos da classe **Phone**



# JPA - Atributo Multivalorado

Utilize as anotações **@ElementCollection** para modelar um atributo multivalorado. Se o atributo for também composto, utilize **@Embeddable**

```
@Embeddable
public class Phone {

    @Enumerated(EnumType.STRING)
    private PhoneType type;

    @Column(length = 2)
    private String areaCode;

    @Column(length = 9)
    private String phoneNumber;
```

```
@Entity
public class UserAccount {

    @Id
    @GeneratedValue
    private int id;
    private String name;
    private String password;
    @OneToOne(mappedBy = "user")
    private PaymentInfo paymentInfo;
    @Embedded
    private Address address;
    @ElementCollection
    private List<Phone> phones;
```

Defina a lista de **Phone** como atributo multivalorado

# JPA - Atributo Multivalorado

Suponha uma entidade **UserGroup** que possui um atributo **tags**, que é uma coleção de Strings




# JPA - Atributo Multivalorado

Se o atributo for de tipo primitivo, String ou Date, não precisa utilizar **@Embeddable**

```
@Entity
public class UserGroup {

    @Id
    @GeneratedValue
    private int id;
    @ManyToMany
    private List<UserAccount> members;
    private String name;
    private String description;
    @ElementCollection
    private List<String> tags;
```



# JPA - Atributo Multivalorado

Desta forma, novas tabelas serão criadas para os atributos multivalorados



# Consultas com Spring Data

Uma forma simples de definir consultas básicas com Spring Data é utilizar [Query creation from method names](#)

Por exemplo:

```
public interface UserRepository extends CrudRepository<UserAccount, Integer> {  
  
    List<UserAccount> findByNameAndPassword(String name, String password);  
  
    List<UserAccount> findByNameOrEmail(String name, String email);  
  
}
```

# Consultas com Spring Data

Para consultas avançadas, recomenda-se usar **JPQL** ou **Native Query** através da anotação [@Query](#)

```
public interface UserRepository extends CrudRepository<UserAccount, Integer> {
```

```
    . . .
```

```
    @Query("select u from UserAccount u where u.name like %?1")  
    List<UserAccount> findByNameEndsWith(String name);
```

 JPQL

```
    @Query(value = "SELECT * FROM user_account u "  
        + "JOIN user_accountphone p ON p.user_account_id = u.id "  
        + "WHERE p.area_code = 'CE'", nativeQuery = true)  
    List<UserAccount> findByCearaPhone(String name);
```

 Native Query



# Links importantes

- <https://hibernate.org/orm/>
- [https://en.wikibooks.org/wiki/Java\\_Persistence/Relationships#JPA\\_Relationship\\_Types](https://en.wikibooks.org/wiki/Java_Persistence/Relationships#JPA_Relationship_Types)
- <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation>
- <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.at-query>
- <https://spring.io/guides/gs/accessing-data-jpa/>
- <https://spring.io/guides/gs/accessing-data-mysql/>