IGRP Web

Interagindo com a Base de Dados

Cliente	NOS*i* – IGRP WEB
Referência	
Versão	1.00
Status	

CONTEÚDO

1 Enquadramento 3
2 Objetivo 3
3 Sobre o Hibernate 4
3.1 Vantagens do Uso do JPA 5
4 Estrutura do uso da bases de dados no IGRP Web 6
5 Interação com a base de dados utilizando dao class 7
5.1 Operações de *select* 8
5.2 Operações de *insert* 11
5.3 Operações de *update* 12
5.4 Operações de *delete* 13
5.5 Mais sobre DAO class 14
6 Interação com a base de dados utilizando query 15
6.1 Operações de *select* 15
6.2 Operações de *insert* 18
6.3 Operações de *update* 19
6.4 Operações de *delete* 20
7 Transaction - RoolBack 21

1 ENQUADRAMENTO

Neste pequeno guia vamos abordar a questão da interação com a base dados no IGRP Web.

Para acessar a Base de Dados o IGRP Web utiliza o JPA (Java Persistence API), API padrão da linguagem Java, que descreve uma interface simples para frameworks de persistência de dados. No caso a framework utilizada é o Hibernate, que facilita muito o trabalho do programador. Entretanto, as tarefas de acesso à base de dados podem ser realizadas com o uso das tradicionais *querys*, para quem assim prefere, podendo tornar mais complexo o trabalho do programador.

Nas páginas que se seguem vamos tentar explicar o funcionamento, as vantagens e a implementação desta tarefa no IGRP Web.

2 OBJETIVO

Pretendemos com este manual orientar o desenvolvedor na interação com a base de dados no IGRP Web. Abordamos a preparação da aplicação para este fim e a implementação propriamente do código java da sua aplicação.

3 SOBRE O HIBERNATE

O Hibernate é um framework para mapeamento objeto-relacional, que diminui e muito a complexidade dos programas em Java, simplificando a sua implementação e permitindo a persistência de dados entre a aplicação e a BD. Este software livre e de código aberto facilita o mapeamento de atributos entre a BD relacional e o modelo de objetos da aplicação pelo uso de ficheiros XML e anotações java, permitindo a transformação de classes Java em tabelas de dados, baseando-se no modelo orientado a objetos.

Resumindo, o Hibernate cria uma camada persistente de dados entre a aplicação e a base de dados, tornando muito mais fácil e rápida a interação entre elas.

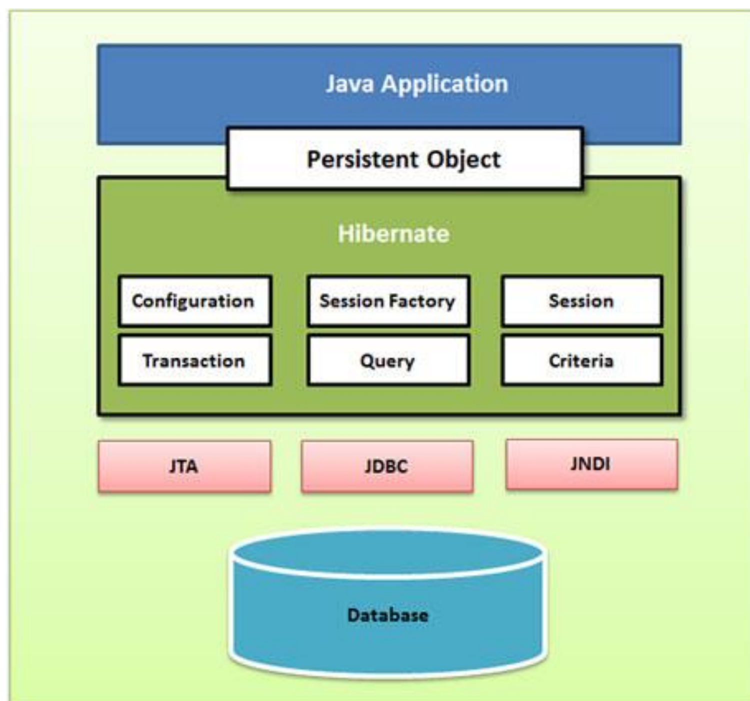

Figure 1 - Persistência de dados com Hibernate

3.1 Vantagens do Uso do JPA

1. **Independência**: utilizando JPQL ou Criteria conseguimos ter uma independência das bases de dados porque agora fica a cargo da implementação da JPA converter a consulta em SQL nativo. Agora as ações básicas de *insert*, *update* e delete também ficam por conta da implementação. Com isso podemos criar uma única aplicação que vai se comunicar com diversos bancos de dados, sem a necessidade de implementar as complexas *querys* com *joins* complicados.

2. **Reduz a necessidade de conhecer SQL:** conhecer pelo menos o básico de SQL é fundamental para qualquer programador. Todavia você consegue trabalhar com a JPA sem esse conhecimento.

3. **Dispensa a conversão de *querys* em objetos:** as implementações da JPA são capazes de realizar essa tarefa que consome tempo demais, quando realizado braçalmente e com o JDBC.

4. **Otimização automática:** nem sempre temos o trabalho de otimizar uma consulta. Porém, esse é um ponto crítico porque muda de implementação para implementação e nem sempre temos o resultado desejado. Caso você precise ter um sistema em que as consultas sejam sempre otimizadas, use o JPA.

5. **Cache de dados:** Cache de dados é muito útil para diminuir a quantidade de requisição ao banco de dados, o chace de dados também faz com que o desempenho da sua aplicação melhore caso você tenha cuidado com o custo de RAM necessário.

4 ESTRUTURA DO USO DA BASES DE DADOS NO IGRP WEB

Conforme pode ver na figura abaixo, o IGRP Web, utiliza duas bases de dados, uma para a framework, e outra para as aplicações. Este documento trata de questões relativas a base de dados para aplicações.

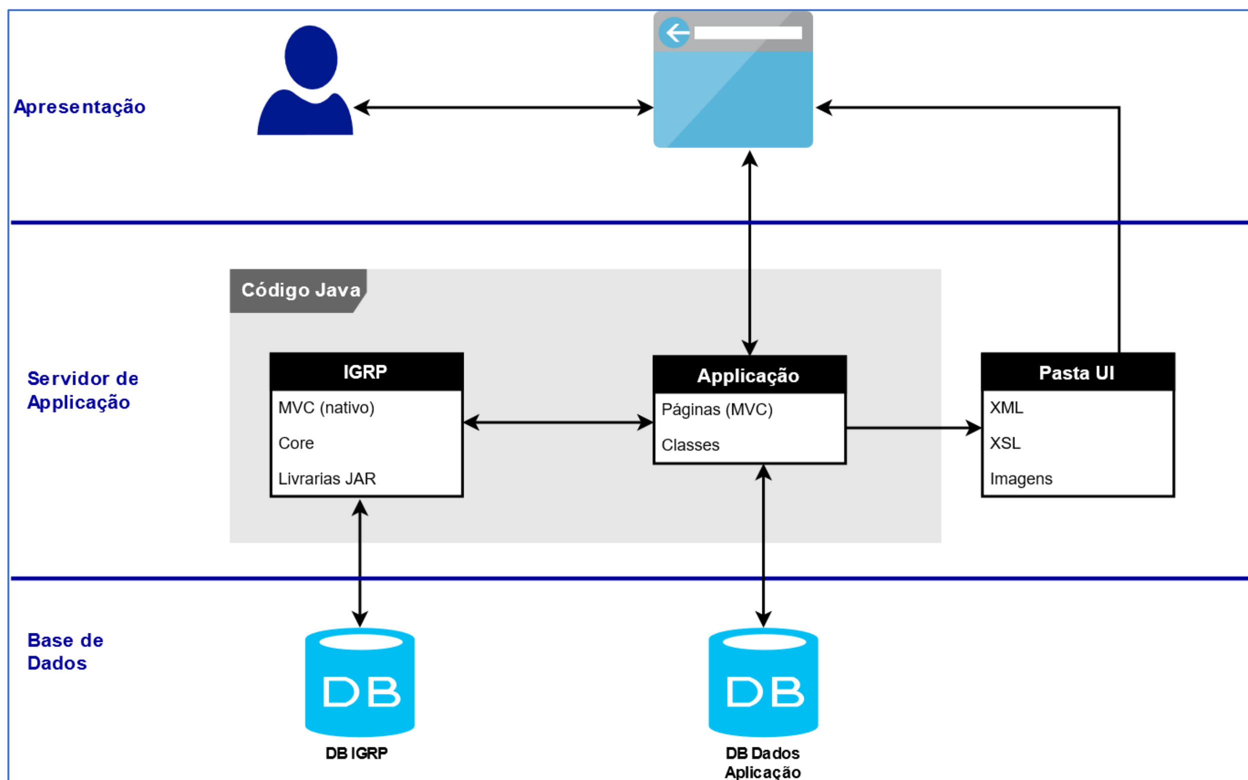

Figure 2 - Esquema de bases de dados

5 INTERAÇÃO COM A BASE DE DADOS UTILIZANDO DAO CLASS

Vamos considerar a DB relacional do modelo abaixo e implementar as operações de leitura e escrita. Assim, todos os exemplos que neste documento abordaremos dizem respeito a esta DB.

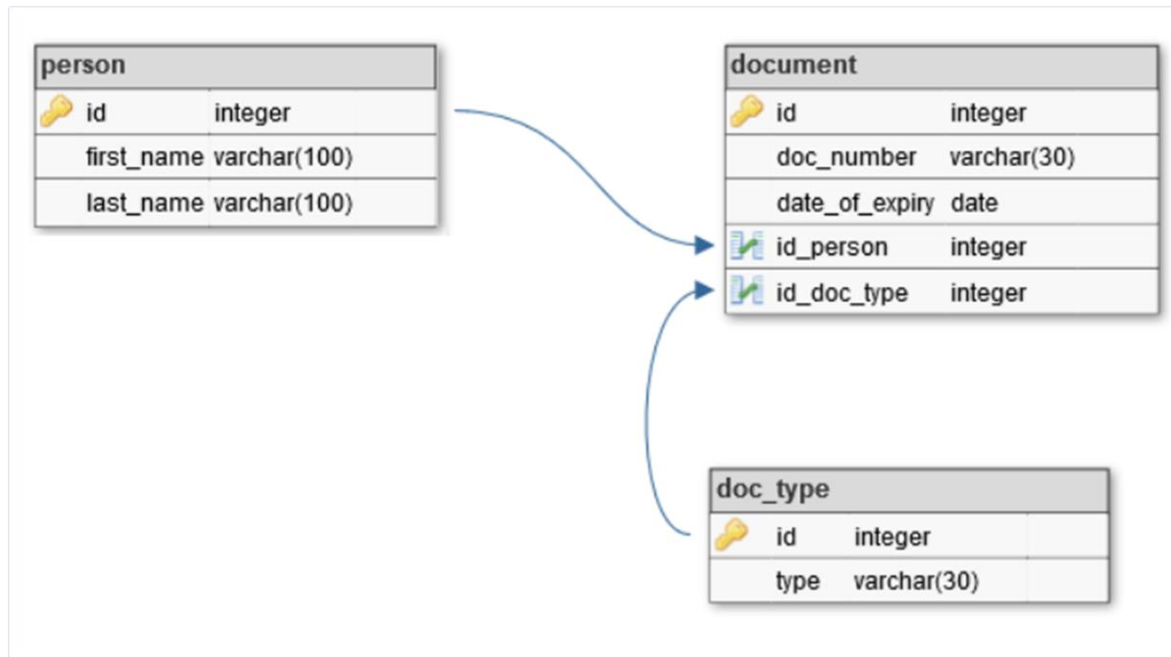

Figure 3 – Modelo ER

Abordamos a preparação da aplicação para uso de sessão Hibernate com uma package DAO, no documento IGRP Web – Mapeamento DAO.

5.1 Operações de *select*

O IGRP Web oferece um conjunto de métodos para procura de informações na DB, que passamos a abordar a seguir.

Método *findOne* – utilizado para selecionar um objeto/linha consoante o id [chave primária da tabela]. Recebe como parâmetro uma string ou um inteiro.

Ex:

```
/*----#start-code(index)----*/
String personId = Core.getParam( "p_id" );
Person person = new Person().findOne( personId );
//...
```

Método *findAll* – utilizado para selecionar todas a linhas de uma tabela. Neste caso, os dados devem ser recebidos em uma lista e deve ser feito o seguinte *import*.

Ex:

```
/*----#start-code(packages_import)----*/
import java.util.List;
/*----#end-code----*/
```

```
/*----#start-code(index)----*/
List<Person> personList = new Person().findAll();
//...
```

Os métodos *find*, *one*, *all*, *andWhere*, *orWhere* e *orderBy* – combinam-se em instruções de modo a produzir o resultado de uma pesquisa personalizada como mostraremos mais a frente.

- O método ***find*** é o primeiro a ser utilizado nas combinações e marca o início da personalização de um *select*.
- Pelo meio utilizamos os método ***andWhere***, ***orWhere***, responsável por definir/filtar os resultados que procuramos e em seguida o método ***orderBy*** para a sua ordenação.
- Finalmente utilizamos os métodos ***one*** e ***all*** que especificam a quantidade de dados pretendida.

Os exemplos abaixo mostram a forma como se pode estruturar o código com estes métodos, todavia, para facilitar a compreensão, a seguir aos exemplos abordamos os métodos *andWhere* e *orderBy*.

Ex1: selecionar todos os person com firstName igual a "Lia" ordenados por de forma descendente por lastName.

```
/*----#start-code(index)----*/
String firstName = "Lia";
List<Person> personList = new Person().find()
                                .andWhere( "firstName","=", firstName )
                                .orderBy( "lastName" )
                                .all();
/*----#end-code----*/
```

Ex2: selecionar todas as linhas de person com um determinado com determinado docType [recebido como parâmetro] ordenadas por firstName e seguidamente por lastName.

```
/*----#start-code(packages_import)----*/
import nosi.core.webapp.databse.helpers.ORDERBY; // if use orderBy(column, ascOrDesc );
import java.util.List;
/*----#end-code----*/
```

```
/*----#start-code(index)----*/
String docType = Core.getParam( "p_doc_type" );
List<Document> documentList = new Person().find()
                                .andWhere( "idDocumentType.typeName","=", docType )
                                .orderBy( "firstName" )
                                .orderBy( "lastName", ORDERBY.ASC )
                                .all();
/*----#end-code----*/
```

Note o primeiro argumento do método ***andWhere*** referenciando um atributo de outra classe. Neste caso o atributo idDocumentType pertence à classe Document e é chave estrangeira referente à classe DocType [tabela doc_type] onde se encontra o atributo typeName [coluna type_name].

O método ***orderBy*** por defeito [quando utilizado com um único parâmetro] faz a ordenação descendente. Porém, se preferir definir a forma como deve ser a ordenação, faça o import da classe ORDERBY e utilize o método de dois parâmetros com os valores ORDERBY.***ASC*** ou ORDERBY.***DESC*** no segundo parâmetro.

Ex3: selecionar um person de acordo com o firstName e lastName enviados por parâmetro.

```
/*----#start-code(index)----*/
String fistName = Core.getParam( "p_first_name" );
String lastName = Core.getParam( "p_last_name" );
Person person = new Person().find();
if( fistName != null )
        person.andWhere("firstName","=", fistName);
if( lastName != null )
```

```
            person.andWhere("lastName","=", lastName);
document.one();
/*----#end-code----*/
```

Sobre método *andWhere*

Primeiro argumento - sempre se refere a alguma coluna da nossa tabela na BD. Se esta coluna for uma chave estrangeira, podemos referir uma coluna na tabela correspondente adicionando o nome desta à string inicial, ficando separados por ponto, como pode ver no exemplo anterior.

Segundo argumento - este é o operador a ser usado na filtragem de dados. Os valores válidos para métodos andWhere de 2 e 3 parâmetros estão indicados na tabela abaixo.

Terceiro argumento – valor [*date*, *number*, *object*] a ser comparado com o valor da coluna especificada no primeiro argumento, de acordo com o operador do segundo argumento, geralmente uma variável.

A tabela a seguir mostra os valores que podemos enviar como segundo parâmetro dos dois métodos.

Método	Segundo parâmetro	
andWhere - 2 parâmetros	"is null"	"is not null"
andWhere - 3 parâmetros	"="	"!="
	">"	"<"
	">="	"<="
	"like"	"not like"

5.2 Operações de *insert*

Inserimos um objecto/linha person do tipo Person na BD com o código:

```
person.insert();
```

O método *insert* retorna o objeto que pretendemos registar na BD. Em caso de sucesso o objeto vem com o **id** [chave primária] preenchido. Caso contrário, o método *hasError* terá o valor true.

Ex:

```
/*----#start-code(save)----*/

Person person = new Person();

person.setFirstName( model.getFirstName() );
person.setLastName ( model.getLastName() );

person = person.insert();

if ( person != null && !person.hasError() )
        Core.setMessageSuccess( "Information saved successfully. Id: "+ person.getId() );
else
        Core.setMessageError( person.getError().toString() );

/*----#end-code----*/
```

5.3 Operações de *update*

A atualização de um objeto/linha person do tipo Person na BD é feita com o código a seguir.

```
person.update();
```

O método *update* retorna o objeto atualizado em caso de sucesso. Em caso de erro o método *hasError* retorna o valor true.

Ex: atualizando o objecto com um determinado id

```
/*----#start-code(save)----*/

String personId = Core.getParam( "p_id" );
Person person = new Person().findOne( personId );
person.setFirstName( model.getFirstName() );
person.setLastName ( model.getLastName() );

person = person.update();

if( person != null && !person.hasError() ){
	Core.setMessageSuccess( "Information updated successfully. Id: "+ person.getId() );
} else
	Core.setMessageError( person.getError().toString() );

/*----#end-code----*/
```

5.4 Operações de *delete*

A exclusão de um objeto/linha person do tipo Person na DB é feita assim.

```
person.delete( person.getId() );
```

O método *delete* retorna um valor booleano – true para sucesso e false para erro.

Ex: eliminando um objeto pelo id

```
/*----#start-code(index)----*/

String personId = Core.getParam( "p_id" );
Person person = new Person().findOne( personId );

boolean del = person.delete( person.getId() );

if ( del == true )
        Core.setMessageSuccess( "Information deleted successfully.");
else
        Core.setMessageError( "Error deleting data from DB!" );

/*----#end-code----*/
```

Ex: eliminando objeto após eliminar dependências

```
/*----#start-code(index)----*/

int personId = Core.getParamInt( "p_id" );
Person person = new Person().findOne( personId );
List<Document> listDoc = new Document().find().andWhere ( "idPerson", "=", person.getId() ).all();
// eliminando dependências
listDoc.forEach( doc -> {
      doc.delete( doc.getDocumentId() );
});
// eliminando o objeto
person.delete( person.getId() );

/*----#end-code----*/
```

5.5 Mais sobre DAO class

Quando utilizamos uma DAO class com Hibernate o tipo de dados dos atributos de uma classe que corresponde a id de outra classe [chave estrangeira] é do mesmo tipo da class em que o id se encontra como pode ver no exemplo a seguir.

```
public class Document extends BaseActiveRecord<Document> implements Serializable {
            private integer id;
            private String number;
            private Person person; // person id
            private DocType docType; // document type id
}
```

Assim, a *field* do id recebe uma instância da classe correspondente como pode ver no código a seguir e não o valor do id.

```
/*----#start-code(index)----*/
Person person = new Person();
person.setLastName ( model.getLast_name() );
person = person.insert();
Document doc = new Document();
doc.setNumber( model.getNumber() );
doc.setPerson( person );
doc.setDocType( new DocType().find().andWhere("type","=", "Passport").one() );
doc.insert();
/*----#end-code----*/
```

Logo, podemos aceder ao objeto a que um id se refere pelo método *getter* correspondente.

Ex:

```
/*----#start-code(index)----*/
String docId = Core.getParam( "p_id" );
Document document = new Document().findOne( docId );
Person person = document.getPerson();
/*----#end-code----*/
```

6 INTERAÇÃO COM A BASE DE DADOS UTILIZANDO QUERY

6.1 Operações de *select*

Selecionando uma linha da DB

Geralmente selecionamos informações na base de dados utilizando a classe **Core.*query***, que retorna uma *QueryInterface*, que pode ser executada ou carregada diretamente no *model*.

Ex: selecionar a linha da tabela person cujo id é recebido como parâmetro

```
/*----#start-code(index)----*/
Integer personId = Core.getParamInt( "p_id" );
String dbConn = "demo_conn";
QueryInterface query = Core.query( dbConn, "SELECT id, first_name, last_name"
                                          +" FROM public.person" )
                                         .where( "id=:person_id" )
                                         .addInt( "person_id", personId );
model.load( query );
/*----#end-code----*/
```

O código acima é equivalente a:

```
/*----#start-code(index)----*/
Integer personId = Core.getParamInt( "p_id" );
String dbConn = "demo_conn";
QueryInterface query = Core.query( dbConn, "SELECT id, first_name, last_name"
                                                  +" FROM public.person"
                                                  +" WHERE id="+ personId.toString() );

ResultSet.Record  result = query.getSingleRecord();
if ( result != null ) {
        model.setId( result.getInt( "id" ) );
        model.setFirst_name( result.getString( "first_name" ) );
        model.setLast_name( result.getString( "last_name" ) );
}
/*----#end-code----*/
```

Os dois códigos acima mostram maneiras diferentes de criar a mesma *query*. O primeiro carrega os dados da *query* em um formulário cujos nomes são iguais aos esperados no resultado [id, first_name, last_name].

Já no segundo a query é executada pelo método ***getSingleRecord*** e o resultado é guardado na variável result. Em seguida os dados dessa variável são introduzidos no formulário.

Como pode perceber a classe **ResultSet.Record** possui um conjunto de métodos para extrair seus dados: *getInt*, *getString*, *getShort*, *getLong*, *getDouble*, *getBigDecimal*, entre outros.

Nos exemplos acima procuramos um objeto de acordo com o id recebido por parâmetro. Se quiséssemos realizar a mesma operação baseados no nome e apelido introduzidos num formulário nosso código podia ficar assim.

```
/*----#start-code(index)----*/
String dbConn = "demo_conn";
QueryInterface query = Core.query( dbConn, "SELECT id, first_name, last_name"
                                           +" FROM public.person" )
                                          .where( "first_name=:firstN AND last_name=:lastN" )
                                          .addString( "firstN", model.getFirst_name() );
                                          .addString( "lastN", model.getLast_name() );
model.load( query );
/*----#end-code----*/
```

Selecionando várias linhas

Quando selecionamos várias linhas da DB a forma de construir as *querys* é igual. Porém o resultado deve ser carregado em um componente capaz de receber várias linhas [ao contrário do formulário].

Ex:

```
/*----#start-code(index)----*/
model.loadTable_person( Core.query( "demo_conn", "SELECT id, first_name, last_name"
                                           +" FROM public.person" ) );
/*----#end-code----*/
```

Já se quisermos guardar o resultado em uma variável devemos invocar o método ***getResultList*** que retorna uma lista de *Tuple* e o nosso código fica assim:

```
/*----#start-code(packages_import)----*/
import java.util.List;
import javax.persistence.Tuple;
/*----#end-code----*/
```

```
/*----#start-code(index)----*/

String dbConn = "demo_conn";
List<Tuple> tupleList = Core.query( dbConn, "SELECT id, first_name, last_name"
                                                  +" FROM public.person" ).getResultList();
// getting data from tuple
if ( tupleList != null ) {
```

```
        tupleList.forEach( tuple -> {
                Integer id = Core.toInt( tuple.get( "id" ).toString() ); // the method get returns an object
                String firstName = tuple.get( "first_name" ).toString();
                String lastName = tuple.get( "last_name" ).toString();
        });
}
/*----#end-code----*/
```

Agora imagine que pretendemos selecionar todos os documentos de um determinado tipo pertencentes a pessoas com determinado nome. Considere que os dados para a procura são introduzidos em um campo *text* e um *combobox* num formulário.

```
/*----#start-code(search)----*/
String dbConn = "demo_conn";
QueryInterface query = Core.query( dbConn, "SELECT D.* FROM document D, person P, doc_type T"
                            + " WHERE D.id_person  = P.id and P.first_name =:name"
                            + " AND D.id_doc_type = T.id and T.type =:type ")
                            .addString( "name", model.getName())
                            .addInt( "type", model.getDoc_Type());
ResultSet result = query. getResultList();
/*----#end-code----*/
```

6.2 Operações de *insert*

O IGRP Web conta com o método **Core.*insert***, que é responsável pelas operações de registo de dados. A instrução no caso deve ser terminada com o método **execute** que retorna um *ResultSet*.

Ex:

```
/*----#start-code(save)----*/

String connection="demo_conn", schema="public", table="person";
ResultSet result =  Core.insert(connection, schema, table)
                                        .addString( "first_name", model.getFirstName() )
                                        .addString( "last_name", model.getLastName() )
                                        .execute();
if ( ! result.hasError() )
            Core.setMessageSuccess( "Person data inserted. Id: "+ result.getInt( "id" ) );
else
            Core.setMessageError();
/*----#end-code----*/
```

Podemos verificar o resultado da operação pelo método ***hasError*** e aceder as informações gravadas na DB através de outros métodos que a classe *ResultSet* oferece como *getInt*, *getString*, etc.

6.3 Operações de *update*

As operações de atualização são realizadas pelos métodos **Core.*update*** e **execute** e funciona de forma similar à inserção.

Ex:

```
/*----#start-code(save)----*/
String connection="demo_conn", schema="public", table="person";
Integer personId = Core.getParamInt("p_id");
ResultSet result =  Core.update(connection, schema, table)
                                        .addString( "first_name", model.getFirst_name() )
                                        .addString( "last_name", model.getFirst_name()  )
                                        .where( "id =:person_id" )
                                        .addInt( "person_id", personId )
                                        .execute();
if ( ! result.hasError() )
            Core.setMessageSuccess( "Person data updated. Id: "+ result.getInt( "id" ) );
else
            Core.setMessageError();
/*----#end-code----*/
```

Novamente podemos verificar o resultado da operação pelo método ***hasError*** e aceder as informações gravadas na DB através de outros métodos que a classe *ResultSet* oferece como *getInt*, *getString*, etc.

6.4 Operações de *delete*

Eliminamos uma linha da DB com os métodos **Core.*delete*** e **execute** e recebemos um *ResultSet* como retorno.

Ex:

```
/*----#start-code(save)----*/
String connection="demo_conn", schema="public", table="person";
Integer personId = Core.getParamInt("p_id");
ResultSet result =  Core.delete( connection, schema, table )
                                .where( "id =:person_id" )
                                .addInt( "person_id", personId )
                                .execute();
if ( ! result.hasError() )
        Core.setMessageSuccess( "Person data deleted. Id: "+ result.getInt( "id" ) );
else
        Core.setMessageError();
/*----#end-code----*/
```

De forma similar aos casos de *insert* e *update* o resultado da operação pode ser verificado pelo método ***hasError*** do *ResultSet* e as informações da linha eliminada são acessíveis pelos métodos *getInt*, *getString*, entre outros.

7 TRANSACTION - ROOLBACK

Quando temos uma sucessão de operações com a base de dados interdependentes entre si estamos diante de um ***transaction,*** onde os erros que podem acontecer acarretam a necessidade de desfazer operações, que possam ter sido executadas com sucesso. Assim, podemos encerrar um ***transaction*** de duas formas:

1. com ***commit***, salvando permanentemente todas as alterações (*insert*, *update*, *delete*) realizadas;
2. com ***roolback*** para desfazer as alterações realizadas, caso haja falha em alguma operação do *transaction*.

Nos exemplos abaixo [com DAO e *query*] a falha na inserção de um **person** ou de um **document** desfaz a outra operação mesmo que esta tenha tido sucesso.

Utilizando DAO

Imports

```
/*----#start-code(packages_import)----*/
import org.hibernate.Session;
import org.hibernate.Transaction;
import nosi.webapps.igrp_tutorials.dao.Document; // dao class
import nosi.webapps.igrp_tutorials.dao.Person; // dao class
/*----#end-code----*/
```

Implementação

```
Session session = Core.getSession("connectionName");
Transaction transaction = null;
try {
	transaction = session.getTransaction();
	transaction.begin();
	Person person = new Person();
	person.setFirstName("Mónica");
	person.setLastName("Semedo");
	session.persist(person);
	Document document = new Document();
	document.setNumber("XY125463-S");
	document.setPersonId(person);
	session.persist(document);
	transaction.commit();
} catch (Exception e) {
	if(transaction != null)
	transaction.rollback();
} finally {
```

```
		if (session != null) {
			session.close();
		}
}
```

Utilizando query

```
QueryInterface query = Core.transaction();
try {
		query.begin();
		ResultSet rp = query.insert("person")
						.addString("first_name","Mónica")
						.addString("last_name","Semedo")
						.execute();
		ResultSet rc = query.insert("document")
						.addDouble("doc_number","XY125463-S")
						.addInt("id_person", Core.toInt( rp.getKeyValue().toString()) )
						.execute();
		query.commit();
} catch (SQLException e) {
		e.printStackTrace();
		try {
			query.roolback();
		} catch (SQLException e1) {
			e1.printStackTrace();
		}
} finally {
		try {
			query.closeConnection();
		} catch (SQLException e) {
			e.printStackTrace();
		}
}
```