# Tópicos Especiais em Análise e Desenvolvimento de Sistemas

Prof. Rafael Odon (rafael.alencar@prof.una.br)

**ORIENTAÇÕES PARA ENTREGA:**

- O roteiro deverá ser feito continuando **com a mesma dupla** do Roteiro 01 e  ou **individual** para quem decidiu não fazer em dupla.
- Enviar até o início da aula do dia 28/03/2013 o código e as respostas de **todas as perguntas** desse roteiro para o e-mail do professor (rafael.alencar@prof.una.br) com o título:
  - `Roteiro JPA 03 – Nome do 1º Aluno / Nome do 2º Aluno`
- Lembre-se: **todas as informações** para responder as perguntas são dadas em **sala de aula**. Preste atenção na aula e faça notas pessoais!
- As **respostas** devem ser pessoais e **escritas com suas palavras**. Não serão aceitas respostas longas e/ou **copiadas** de forma literal do material de referência ou Internet.

# Java Persistence API & Hibernate

## Roteiro de Aula 03 – 31/10/2012

Esse Roteiro se baseia no uso dos softwares:

- Netbeans 7.4.1 Java EE
- Java DB (Apache Derby)
- Java Development Kit (JDK) 1.7

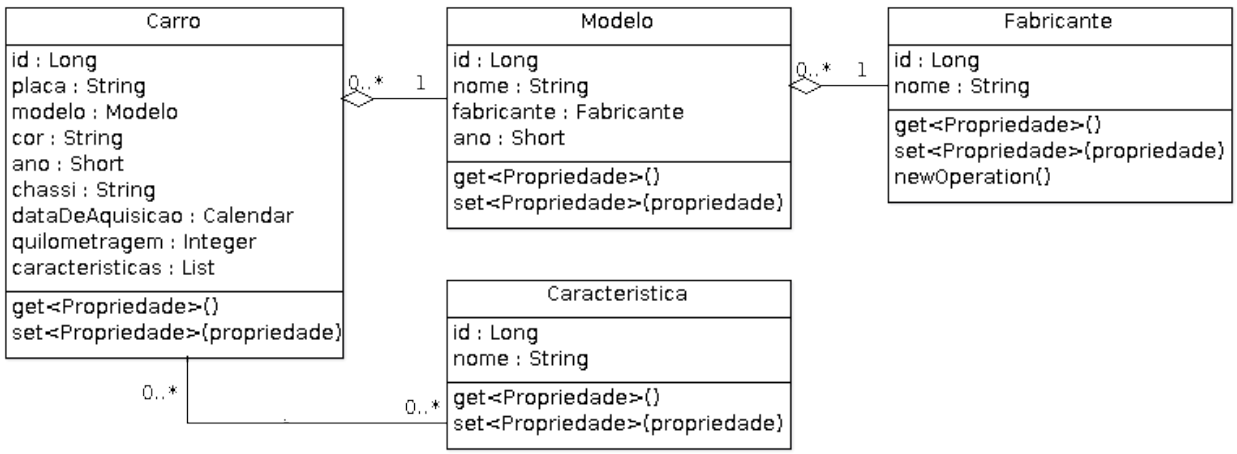
### Fase 1: Continuando de onde paramos...

No Roteiro de Aula 02, criamos diversas classes no padrão JavaBeans para representar o universo de um sistema de frota de veículos.

Fizemos com que todas fossem entidades JPA, mapeadas para tabelas em um Banco de Dados.

Relacionamos as classes umas com os outras e vimos esses relacionamentos serem refletidos no mundo relacional através de chaves estrangeiras e até mesmo de tabelas de relacionamento.

Vamos relembrar o nosso diagrama de classes:




Vamos continuar atuando nesse mesmo projeto SysFrota criado no Roteiro 02 durante esse Roteiro 03.

No Roteiro atual, criaremos Objetos de Acesso a Dados (DAO) para encapsular a lógica de persistência em classes apropriadas.

O objetivo de um DAO é prover uma interface única para operações de persistência de uma determinada Entidade tais como: *salvar*, *remover*, *buscar pelo id*, *listar todos*, *listar aquelas que atendem algum critério*, entre outras.

Sendo assim, para cada Entidade da nossa aplicação, fabricaremos um DAO respectivo: FabricanteDAO, ModeloDAO, CaracteristicaDAO e CarroDAO. Feito isso, qualquer classe que precise realizar operações de persistência deve utilizar o DAO da respectiva entidade.

Não é uma boa estratégia criar métodos e classes antes que eles tenham alguma utilização. No entanto, para fins didáticos e pensando no sistema web SysFrota que será feita mais adiante, vamos preparar nossos DAOs previamente e testá-los brevemente na nossa classe *Bootstrap*.

## Fase 2: Criando a classe JPAUtil

Na classe *Bootstrap* do Roteiro 02, fizemos diversas operações de persistência com nossas entidades. Para tanto foi preciso ter disponível um *EntityManager*. Para criar um EntityManager usamos os seguinte trecho de código:

```
    ...
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("db_sysfrota");
    EntityManager em = emf.createEntityManager();


    ...

    em.close();
    emf.close();
}
```

Perceba que a criação do EntityManager depende do **nome da Unidade de Persistência**. Dessa forma, agora que vamos criar nossos DAOs, não seria uma boa ideia ter esse mesmo código repetido em diversas classes, muito menos ter o nome da Unidade de Persistência espalhado em diversos pontos da aplicação. Vamos então criar uma classe utilitária responsável dentre outras coisas pela criação do EntityManager, a JPAUtil.

1. Crie a classe JPAUtil no pacote *sysfrota.persistence*
2. O código da JPAUtil deve prover uma forma simples de obter um EntityManager e outra de fechar os recursos alocados. Uma sugestão de implementação segue abaixo. Cuide dos *imports* e dos demais detalhes tais como o nome da unidade de persistência, e deixe sua classe JPAUtil funcional.

```
...
public class JPAUtil {

    private static final EntityManagerFactory emf = Persistence.createEntityManagerFactory("NOME DA UNIDADE");
    private static ThreadLocal<EntityManager> manager = new ThreadLocal<EntityManager>();

    public static EntityManager getEntityManager(){
        EntityManager em = JPAUtil.manager.get();
        if (em == null || !em.isOpen()) {
            em = JPAUtil.emf.createEntityManager();
            JPAUtil.manager.set(em);
        }
        return em;
    }

    public static void close() {
        emf.close();
    }
}
```

3. Observe que na sugestão acima a classe JPAUtil não precisa ser instanciada para obtermos um EntityManager, bastando para isso acessar o método estático *getEntityManager().*

4. Para testar, modifique sua classe de Bootstrap para utilizar a JPAUtil ao invés do antigo código de obtenção do EntityManager e também para o fechamento dos recursos no fim do método:

```
    ...

    EntityManager em = JPAUtil.getEntityManager();

    em.getTransaction().begin();

    ...

    JPAUtil.close();
}
```

**Observação:** a arquitetura do JPA indica utilizar uma instância de EntityManager por transação, e nunca reutilizar um mesma instância de Entity Manager em duas transações ao mesmo tempo. No desenvolvimento da aplicação Web esse assunto voltará a ser abordado!

## Fase 3: Criando o DAO de Fabricante

1. Crie a classe FabricanteDAO no pacote *sysfrota.persistence.dao*

2. Faça-a ter um EntityManager como membro privado, obtido da JPAUtil. Insira também nessa classe o método *salvar* como no exemplo abaixo:

```
public class FabricanteDAO {

    private EntityManager em = JPAUtil.getEntityManager();

    public Fabricante salvar(Fabricante fabricante) {
        Fabricante retorno = em.merge(fabricante);
        return retorno;
    }
}
```

3. Pronto! Nosso DAO já passou a ser útil. Se alguma classe externa precisar salvar/atualizar uma instância de Fabricante, basta instanciar FabricanteDAO e chamar seu método *salvar*. Perceba que com isso a classe externa não precisa conhecer o EntityManager, muito menos realizar o controle transacional.

4. Perceba que utilizamos o método *merge* para salvar, ao invés do já conhecido método *persist.* A diferença básica é que o *persist* sempre tenta fazer um INSERT no banco de dados, e o MERGE é mais esperto, e dentre outras coisas, sabe quando deve ser feito um INSERT ou um UPDATE.

5. Vamos agora criar os métodos *carregarPeloId*, *remover* e *listarTodos*, de acordo com o código abaixo:

```
public class FabricanteDAO {
    ...

    public Fabricante carregarPeloId(Long id) {
        return em.find(Fabricante.class, id);
    }
```

```
    public void remover(Fabricante fabricante) {
        em.remove(fabricante);
    }

    public List<Fabricante> listarTodos() {
        Query query = em.createQuery("FROM Fabricante ORDER BY nome");
        return query.getResultList();
    }
}
```

6. Perceba que o método *listarTodos* introduziu o uso da JPQL (Java Persistence Query Language), uma linguagem de consulta muito parecida com o SQL, mas que tem sua sintaxe baseada nos nomes das classes de entidade Java, bem como o nome de suas propriedades. Isso afasta o usuário do trabalho sujo de conhecer os nomes de tabelas e colunas do banco de dados!

7. Modifique sua classe *Bootstrap* para **utilizar o DAO de Fabricante** construído. Utilize-a para salvar os fabricantes criados, e também para listá-los e imprimí-los em ordem alfabética. Lembre-se de abrir e fechar as transações adequadamente para que as informações sejam persistidas.

```
...
public class Bootstrap {

    private static FabricanteDAO fabricanteDAO = new FabricanteDAO();

    public static void main(String[] args) throws Exception {
        ...

        Fabricante fiat = new Fabricante("Fiat");
        fiat = fabricanteDAO.salvar(fiat);

        ...
    }
}
```

**RESPONDA:**

I. Qual o objetivo da classe JPAUtil? Se o método de obtenção do EntityManager não fosse estático, que implicações isso traria? Que padrões de projeto essa classe pode representar?

II. Baseado nos conhecimentos obtidos até aqui, e observando atentamente a implementação do DAO de Fabricante obtida nessa fase e responda:
a) Que métodos do EntityManager são usados para salvar uma entidade nova?
b) Que método do EntityManager é usado para atualizar uma entidade já existente?
c) Que método do EntityManager é usado para remover uma entidade? Que parâmetros ele pede? Que tipo de comando SQL é gerado por esse método?
d) Que método do EntityManager é usado para criar uma consulta com JPQL? O que ele retorna? Que parâmetro ele pede? O que mais precisou ser feito para obtermos uma lista de entidades dessa consulta?
e) Que método do EntityManager é usado para buscar uma Entidade pelo seu Id? Que parâmetros ele pede? Como o JPA sabe qual é o Id de uma entidade?

III. Que tipos de operações necessitam de controle transacional? Como esse controle é feito via EntityManager?

IV. Baseado no exemplo da JPQL do método *listarTodos* do FabricanteDAO, tente imaginar como seria uma JPQL que trouxesse fabricantes de id menor que 10, ordenado por id. Escreva-a aqui.

## Fase 4: *Criando o DAO de Modelo*

Baseado na implementação da classe FabricanteDAO, você já tem condições de criar outros DAO. Mãos à obra!

1. Crie no pacote *sysfrota.persistence.dao* a classe ModeloDAO.
2. Inclua nessa classe métodos de mesmo nome daqueles incluídos em FabricanteDAO (*carregarPeloId, salvar, remover, listarTodo)* observando que agora estamos lidando com a entidade *Modelo*. Modifique o que for necessário no código desses métodos.
3. Faça com que o método *listarTodos* do ModeloDAO traga os modelos ordenados primeiramente por ano e depois por nome.
4. Crie ainda no ModeloDAO um novo método, chamado *listarDoFabricante*, cujo objetivo é listar apenas os Modelos de um determinado Fabricante. Veja o código sugerido abaixo e não se esqueça do import de *javax.persistence.Query*:

```
public class ModeloDAO{
    ...

    public List<Modelo> listarDoFabricante(Fabricante fabricante) {
        Query query = em.createQuery("FROM Modelo m WHERE m.fabricante = :f ORDER BY m.nome");
        query.setParameter("f", fabricante);
        return query.getResultList();
    }
}
```

5. Observe a JPQL utilizada no último método criado. Algumas coisas precisam ser observadas:
   I. Foi possível dar um apelido para a entidade *Modelo*, que no caso foi *m.*
   II. Identificamos um parâmetro de consulta (o fabricante) através da convenção dois pontos seguidos de um texto, que no caso foi *:f*
   III. Através do método setParameter da Query, pudemos informar qual foi o valor do parâmetro *:f* fazendo a chamada query.setParameter("f", fabricante); O primeiro parâmetro do método setParameter permite mencionar o nome do parâmetro a ser substituído, sem os dois pontos. O segundo parâmetro permite indicar qual objeto irá substituir esse parâmetro na JPQL.
   IV. Perceba que não utilizamos Chaves Estrangeiras como num SQL Comum. Fizemos referência direta à propriedade fabricante de Modelo, que é uma instância de *Fabricante.*
6. Proponha um novo método *listarDoFabricanteNoAno(Fabricante fabricante, Short ano)* no DAO de Modelo capaz de listar todos os modelos de um determinado Fabricante num dado Ano. Utilize o método *setParameter* da Query para indicar o fabricante e o ano.
7. Modifique sua classe Bootstrap para utilizar o DAO de Modelo construído. Utilize-a para salvar os modelos, e também para listá-los e imprimí-los. Teste o método novo criado no passo anterior.

## Fase 4: *Criando os DAO de Carro e Característica*

1. Crie os DAO de *Carro* e *Caracteristica*. Eles devem conter pelo menos os mesmos métodos da classe FabricanteDAO: *salvar, carregarPeloId, remover e listarTodos.* Lembre-se de modificar todo os parâmetros, retornos e JPQLs para referenciar as entidades corretas, de acordo com o DAO criado.

**DESAFIO EXTRA:** Você achou muito repetitivo criar os 4 DAOs do nosso sistema? Os 4 métodos fundamentais *salvar, carregarPeloId, remover e listarTodos* não ficaram extremamente parecidos?

Você seria capaz de criar um DAO genérico para prover esses métodos básicos para quaisquer classes? Onde e como ele seria usado? Dica: utilize Generics e Reflection e após tentar pesquise soluções consagradas na Internet.

2. Proponha três novos métodos no DAO de *Carro* para listar carros segundo alguns critérios de sua escolha;

**RESPONDA:**

I. O que faz cada um dos métodos propostos por você no DAO de *Carro*? Qual foi a JPQL utilizada e qual foi o SQL gerado no console por cada um dos métodos*?*

### Fase 5: *Explorando mais a JPQL*

A JPQL permite fazer diversas operações correlatas ao SQL, porém de uma forma ligeiramente diferente. Vamos dar alguns exemplos. Preste atenção nas JPQLs apresentadas para abstrair alguns conceitos importantes.

1. Adicione à classe FabricanteDAO o método *quantidadeModelos* abaixo, cujo objetivo é retornar quantos modelos um fabricante possui de forma simples. Teste seu uso na classe *Bootstrap*.

```
public Long quantidadeModelos(Fabricante fabricante){
    Query query = em.createQuery("SELECT COUNT(m) FROM Modelo m WHERE m.fabricante = :f");
    query.setParameter("f", fabricante);
    Long quantidade = (Long) query.getSingleResult();
    return quantidade;
}
```

2. Adicione à classe CarroDAO o método *somaQuilometragem* abaixo, cujo objetivo é retornar a soma de todas as quilometragens de todos os veículos da frota. Teste seu uso na classe *Bootstrap*.

```
public Long somaQuilometragem(){
    Query query = em.createQuery("SELECT SUM(c.quilometragem) FROM Carro c");
    Long soma = (Long) query.getSingleResult();
    return soma;
}
```

3. Adicione à classe CarroDAO o método *carrosComMaiorQuilometragem*, cujo objetivo é retornar a lista de carros que possuem a maior quilometragem da frota. Teste seu uso na classe *Bootstrap*.

```
public List<Carro> carrosComMaiorQuilometragem(){
    Query query = em.createQuery("FROM Carro c WHERE "
            + "c.quilometragem = (SELECT MAX(c2.quilometragem) FROM Carro c2)");
    List<Carro> lista = query.getResultList();
    return lista;
}
```

4. Adicione à classe FabricanteDAO o método *fabricantesVigentes*, cujo objetivo é retornar a lista de fabricantes que possuem veículos fabricados no ano corrente. Teste seu uso na classe *Bootstrap*.

```
public List<Fabricante> fabricantesVigentes(){
    Query query = em.createQuery("FROM Fabricante f WHERE EXISTS "
            + "(FROM Carro c WHERE c.modelo.fabricante = f AND c.ano = :a)");
    short anoVigente = (short) Calendar.getInstance().get(Calendar.YEAR);
    query.setParameter("a", anoVigente);
    List<Fabricante> lista = query.getResultList();
    return lista;
}
```

**RESPONDA:**

I. Qual a principal diferença entre o SQL e o JPQL?

II. Onde foram parar os JOINs do SQL na JPQL? Que construções equivalem a JOINs na JPQL? Escolha uma consulta do Roteiro que faça JOIN, execute-a e cole aqui o SQL gerado para exemplificar.

III. Nos exemplos que vimos, a JPQL retorna listas de entidades mapeadas ou resultados simples como números. Imagine uma situação onde a JPQL devesse retornar alguma informação complexa mas que não se tratasse de uma entidade/tabela. Nesse sentido, descubra para que serve o SELECT NEW na JPQL, e como utilizá-lo, dando um exemplo.

IV. Encontre na internet a documentação/tutorial oficial da Oracle sobre JPQL no Java EE 6. Cole aqui o link e mencione alguma funcionalidade nova que você descobriu por lá. Dicas: outras funções agregadas, funções para pegar data corrente, funções para manipular Strings, entre outras.

V. Qual o maior problema do uso da JPQL? Como podemos contornar esse problema? Pesquise e descubra para que serve a *API Criteria Builder do JPA.* Pesquise também a vantagem de se utilizar *API Metamodel do JPA* e que problemas isso veio a resolver.

## Fase 6: Promovendo *controle transacional adequado*

A maneira correta de lidar com transações é tentando fazer o commit do bloco de operações de persistência e realizando o rollback caso ocorra algum erro. Em Java, o mecanismo de captura de exceção nos permite refletir essa ideia.

1. **Abra a transação** à partir da sua instância de EntityManager.
2. Defina um **try/catch** que irá **envolver as operações de persitência que fazem parte da sua transação** (adapte para o seu código).
3. **Realize o commit** das operações antes de fechar o bloco **try**
4. Caso seja capturada alguma RuntimeException (exceção em tempo de execução) **realize o rollback** da transação e faça o tratamento adequado (no nosso caso, apenas repassar a exceção para cima).

```
...
    em.getTransaction().begin();
    try{
        // aqui vão as operações de persitência que fazem parte da transação
        fabricanteDAO.salvar(fiat);
        modeloDAO.salvar(uno);
        carroDAO.salvar(meuCarro);

        em.getTransaction().commit();
    }catch(RuntimeException e){
        em.getTransaction().rollback();
        throw e;
    }
...
```

5. O código acima pode ser adaptado tanto para envolver todas as operações de persitência da sua classe Boostrap, quanto para envolver o código dos diversos métodos que realizam escrita no banco em cada um de seus DAOs. Mas tenha em mente que com os recursos disponibilizados não pode haver aninhamento de transações, ou seja, uma transação não pode ser criada dentro de outra.

6. No desenvolvimento da aplicação Web iremos controlar o mecanismo transacional por requisição Web, de modo que cada clique do usuário na tela gere uma transação gerenciada por um filtro de uso geral.