Tutorial - JPA 2.0 com EclipseLink

Ricardo Ramos de Oliveira 7250350
Universidade de São Paulo (ICMC-USP)
JPA com EclipseLink

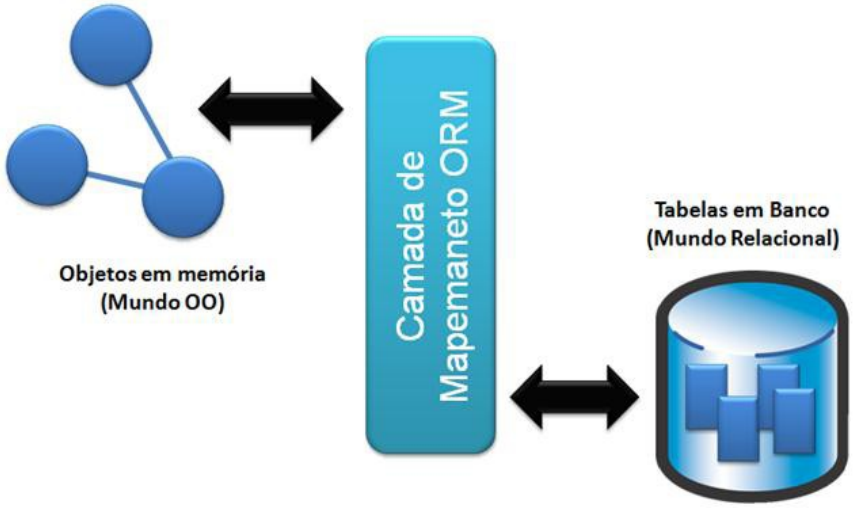
Este tutorial explica como usar o EclipseLink, a implementação de referência para a Java Persistence API (JPA). O uso de EclipseLink é demonstrado para aplicações Java stand-alone (fora do ambiente Java EE). O EclipseLink implementação 2.3.x foi utilizado para este tutorial.

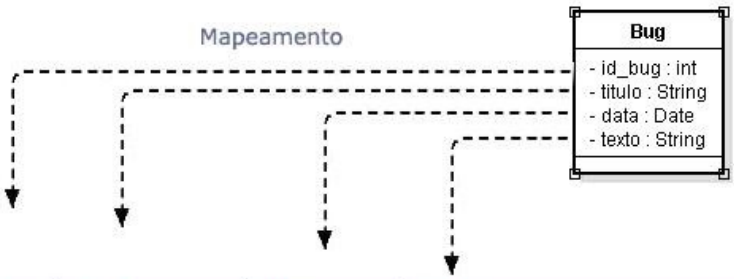
Índice

1. JPA

1.1. visão global

O processo de mapeamento de objetos Java para tabelas de banco de dados e vice-versa é chamada de "mapeamento objeto-relacional" (ORM). A Java Persistence API (JPA) é uma abordagem para ORM. Via JPA o desenvolvedor pode mapear, armazenar, atualizar e recuperar dados de bancos de dados relacionais para objetos Java e vice-versa, JPA permite ao desenvolvedor trabalhar diretamente com objetos ao invés de instruções SQL. JPA é uma especificação e implementação entre várias disponíveis. Este tutorial irá usar EclipseLink como a implementação JPA. A implementação JPA é normalmente chamado fornecedor de persistência. JPA pode ser usado em Java EE e em aplicações Java SE.





id_bug	titulo	data	texto
1	Date Bug	2007-01-13	Quando o context JSF lance este avis...
5	Value Object	2007-02-15	Quando utilizar VOs, colocar também ...
6	heheheh	2007-02-16	kwjekwejwkj
7	Muito show!!!	2007-02-16	hehehehehehe 10

1.2. Entidade

Uma classe que deve ser mantida em um banco de dados deve ser anotada com "javax.persistence.Entity". Tal classe é chamada de entidade. A JPA vai criar uma tabela para a entidade em seu banco de dados. As instâncias da classe corresponde a uma linha da tabela.

Todas as classes de entidade devem definir uma chave primária, deve ter um construtor sem argumentos e ou não tenha permissão para ser final. As chaves podem ser um campo único ou uma combinação de campos. JPA permite gerar automaticamente a chave primária no banco de dados através da anotação **@ GeneratedValue**.

Por padrão, o nome da tabela corresponde ao nome da classe. Você pode mudar isso adicionando a anotação **@ Table (name = "NOMEDATABELA")**.

1.3. Persistência dos Campos

Os campos da Entidade serão salvos no banco de dados. JPA pode usar suas variáveis de instância (campos) ou o correspondente getters e setters para acessar os campos. Você não tem permissão para misturar os dois métodos. Se você quiser usar os métodos setter e getter da classe Java deve seguir as convenções de nomenclatura Java Bean. A JPA persisti por padrão todos os campos de uma entidade, se os campos não devem ser guardados deverão ser assinaladas com a anotação **@ Transient**.

Por padrão cada campo é mapeado para uma coluna com o nome do campo. Você pode alterar o nome padrão via **@Column (name = "NOMENOVACOLUNA")**.

As anotações a seguir pode ser usadas.

@Id	Identificador único ID do banco de dados
@GeneratedValue	Juntamente definido com o ID este valor será gerado automaticamente.
@Transient	O campo não será salvo no banco de dados

Tabela 1. Anotações para campos / getter e setter

1.4. Mapeamento de Relacionamento

JPA permite definir relacionamentos entre as classes, por exemplo, pode ser definida uma classe que faz parte de outra classe (composição). As classes podem ter relacionamentos 1-1, um para muitos, muitos para um, e muitos para muitos com outras classes.

A relação pode ser bidirecional ou unidirecional, por exemplo, em um relacionamento bidirecional ambas as classes armazenam uma referência de uma para a outra, enquanto no caso unidirecional apenas uma classe tem uma referência da outra classe. Dentro de um relacionamento bidirecional você precisa especificar o lado proprietário dessa relação na outra classe com o atributo "mappedBy", por exemplo, **@ ManyToMany (mappedBy = "ATRIBUTOCLASSEPROPRIETÁRIA")**.

@OneToOne
@OneToMany
@ManyToOne
@ManyToMany

Tabela 2. Anotações de Relacionamentos

1.5. Gerenciador de entidade

A entidade manager "javax.persistence.EntityManager" fornece as operações do banco de dados, por exemplo, encontrar objetos, persisti-los, remover objetos do banco de dados, etc. Em uma aplicação JavaEE o gerenciador de entidades é automaticamente inserido na aplicação web. Fora de uma aplicação JavaEE você precisa gerenciar o gerenciador de entidades.

Entidades que são gerenciadas por um Gerenciador de Entidade irá propagar automaticamente as alterações e modificações para o banco de dados (se isso acontecer dentro de uma instrução commit). Se o Gerenciador de Entidade for fechado (via close ()), as entidades gerenciadas se encontram em estados separados. Para sincronizá-los novamente com o banco de dados o Gerenciador de Entidade fornece o método merge ().

O contexto de persistência descreve todas as Entidades de um gerenciador de entidades.

1.6. Unidades de persistência

O EntityManager é criado pelo EntitiyManagerFactory que é configurado pela unidade de persistência. A unidade de persistência é descrita através do arquivo "persistence.xml" no diretório META-INF na pasta de origem do projeto. Um conjunto de entidades logicamente conectadas, serão agrupadas através de uma unidade de persistência. O arquivo "persistence.xml" define os dados de conexão ao banco de dados, por exemplo, o driver, o usuário e a senha.

2. Instalação

2.1. EclipseLink

Baixar o arquivo instalador .zip da implementação do EclipseLink no link: http://www.eclipse.org/eclipselink/downloads/. O download contém vários jar's. Precisamos dos seguintes jars:

- eclipselink.jar
- javax.persistence_*.jar

2.2. Banco de Dados Derby

Os exemplos a seguir utilizam o Apache Derby como banco de dados. O download do Derby pode ser feito através do link: http://db.apache.org/derby/. A partir deste tutorial vamos precisar do "derby.jar". Para mais detalhes sobre Derby (que não é necessário para este tutorial) consulte Apache Derby.

3. Exemplo simples

3.1. Projeto e Entidade

Crie um projeto Java "de.vogella.jpa.simple". Crie uma pasta "lib" e coloque os jar's necessários da JPA e o derby.jar nesta pasta. Adicionar as bibliotecas ao classpath do projeto.

Depois de criar o pacote "de.vogella.jpa.simple.model" e criar as seguintes classes.

```
@Entity
public class Todo {
      @Id
      @GeneratedValue(strategy = GenerationType.IDENTITY)
      private Long id;
      private String summary;
      private String description;
      public String getSummary() {
              return summary;
      }
      public void setSummary(String summary) {
              this.summary = summary;
      }
      public String getDescription() {
              return description;
      }
      public void setDescription(String description) {
              this.description = description;
      }
      @Override
      public String toString() {
              return "Todo [summary=" + summary + ", description=" +
description
                              + "]";
      }
}
```

3.2. Unidade de Persistência

Crie um diretório "META-INF" na sua pasta "src" e crie o arquivo "persistence.xml". Este exemplos utiliza atributos específicos do EclipseLink, por exemplo, através do parâmetro " eclipselink.ddl-generation" você pode especificar que o esquema de banco de dados será automaticamente excluído e criado.

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
      version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">

    <persistence-unit name="todos" transaction-type="RESOURCE_LOCAL">
      <class>de.vogella.jpa.simple.model.Todo</class>
      <properties>
              <property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.EmbeddedDriver" />
              <property name="javax.persistence.jdbc.url"

value="jdbc:derby:/home/vogella/databases/simpleDb;create=true />
              <property name="javax.persistence.jdbc.user"
value="test" />
              <property name="javax.persistence.jdbc.password"
value="test" />
              <!-- EclipseLink should create the database schema
automatically -->
              <property name="eclipselink.ddl-generation"
value="create-tables" />
              <property name="eclipselink.ddl-generation.output-mode"
                              value="database" />
      </properties>
    </persistence-unit>
</persistence>
```

O banco de dados especificado via "javax.persistence.jdbc.url" será criado automaticamente pelo driver Derby. Você pode querer ajustar o path, que atualmente é baseado em notações Linux e aponta para o diretório home no sistema Linux.

Para ver o SQL gerado para as bases de dados o conjunto eclipselink.ddl-generation.output modo de valor a partir de "banco de dados" para "sql-script" ou "ambas". Dois arquivos serão gerados "createDDL.jdbc 'e' dropDDL.jdbc'.

3.3. Teste sua Instalação

Crie a seguinte classe Main.java na qual criará uma nova entrada sempre que for executada. Após a primeira chamada é necessário remover a propriedade "eclipselink.ddl-generation" de persistence.xml caso contrário, você receberá um erro do EclipseLink como tentar criar o esquema de banco de dados novamente. Uma

alternativa que você poderia definir a propriedade como "drop-and-create-tables" mas isso iria apagar o seu esquema de banco de dados em cada execução.

```
package de.vogella.jpa.simple.main;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import de.vogella.jpa.simple.model.Todo;

public class Main {
      private static final String PERSISTENCE_UNIT_NAME = "todos";
      private static EntityManagerFactory factory;

      public static void main(String[] args) {
            factory =
Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
            EntityManager em = factory.createEntityManager();
            // Read the existing entries and write to console
            Query q = em.createQuery("select t from Todo t");
            List<Todo> todoList = q.getResultList();
            for (Todo todo : todoList) {
                  System.out.println(todo);
            }
            System.out.println("Size: " + todoList.size());

            // Create new todo
            em.getTransaction().begin();
            Todo todo = new Todo();
            todo.setSummary("This is a test");
            todo.setDescription("This is a test");
            em.persist(todo);
            em.getTransaction().commit();

            em.close();
      }
}
```

Execute o seu programa várias vezes para ver que o banco de dados é preenchido.

4. Exemplo de Relacionamento

Crie um projeto Java "de.vogella.jpa.eclipselink", crie novamente uma pasta "lib" e coloque os jar's necessários do JPA e o derby.jar nesta pasta. Adicione as bibliotecas ao classpath do projeto.

Criar o pacote "de.vogella.jpa.eclipselink.model" e as seguintes classes.

```
package de.vogella.jpa.eclipselink.model;

import java.util.ArrayList;
import java.util.List;

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

@Entity
public class Family {
	@Id
	@GeneratedValue(strategy = GenerationType.TABLE)
	private int id;
	private String description;

	@OneToMany(mappedBy = "family")
	private final List<Person> members = new ArrayList<Person>();

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getDescription() {
		return description;
	}

	public void setDescription(String description) {
```

```
		this.description = description;
	}

	public List<Person> getMembers() {
		return members;
	}

}

package de.vogella.jpa.eclipselink.model;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
import javax.persistence.OneToMany;
import javax.persistence.Transient;

@Entity
public class Person {
	@Id
	@GeneratedValue(strategy = GenerationType.TABLE)
	private String id;
	private String firstName;
	private String lastName;

	private Family family;

	private String nonsenseField = "";

	private List<Job> jobList = new ArrayList<Job>();

	public String getId() {
		return id;
	}

	public void setId(String Id) {
```

```
                this.id = Id;
        }

        public String getFirstName() {
                return firstName;
        }

        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }

        // Leave the standard column name of the table
        public String getLastName() {
                return lastName;
        }

        public void setLastName(String lastName) {
                this.lastName = lastName;
        }

        @ManyToOne
        public Family getFamily() {
                return family;
        }

        public void setFamily(Family family) {
                this.family = family;
        }

        @Transient
        public String getNonsenseField() {
                return nonsenseField;
        }

        public void setNonsenseField(String nonsenseField) {
                this.nonsenseField = nonsenseField;
        }

        @OneToMany
        public List<Job> getJobList() {
                return this.jobList;
```

```
	}

	public void setJobList(List<Job> nickName) {
		this.jobList = nickName;
	}

}

package de.vogella.jpa.eclipselink.model;

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

@Entity
public class Job {
	@Id
	@GeneratedValue(strategy = GenerationType.TABLE)
	private int id;
	private double salery;
	private String jobDescr;

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public double getSalery() {
		return salery;
	}

	public void setSalery(double salery) {
		this.salery = salery;
	}

	public String getJobDescr() {
		return jobDescr;
```

```
	}

	public void setJobDescr(String jobDescr) {
		this.jobDescr = jobDescr;
	}

}
```

Crie o arquivo "persistence.xml" em "src / META-INF". Lembre-se de mudar o caminho para o banco de dados.

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
	version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
    <persistence-unit name="people" transaction-
type="RESOURCE_LOCAL">

	<class>de.vogella.jpa.eclipselink.model.Person</class>
	<class>de.vogella.jpa.eclipselink.model.Family</class>
	<class>de.vogella.jpa.eclipselink.model.Job</class>

	<properties>
		<property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.EmbeddedDriver" />
		<property name="javax.persistence.jdbc.url"

value="jdbc:derby:/home/vogella/databases/relationsshipDb;create=true
" />
		<property name="javax.persistence.jdbc.user"
value="test" />
		<property name="javax.persistence.jdbc.password"
value="test" />
		<!-- EclipseLink should create the database schema
automatically -->
		<property name="eclipselink.ddl-generation"
value="create-tables" />
```

```
            <property name="eclipselink.ddl-generation.output-mode"
                              value="database" />
      </properties>
    </persistence-unit>
</persistence>
```

A seleção a seguir é implementado como um teste JUnit. Para mais informações consulte Veja um Tutorial do JUnit. O método setup () irá criar um arquivo de entradas de alguns testes. Após as entradas de teste são criados, eles serão lidos e no campo uma das entradas é alterado e salvo no banco de dados.

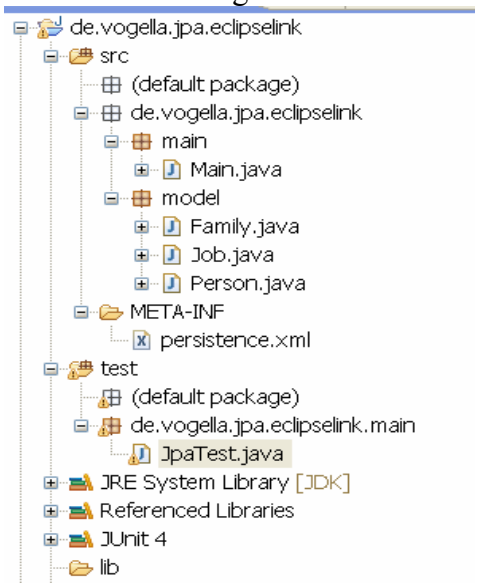
```
package de.vogella.jpa.eclipselink.main;

import static org.junit.Assert.assertTrue;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

import org.junit.Before;
import org.junit.Test;

import de.vogella.jpa.eclipselink.model.Family;
import de.vogella.jpa.eclipselink.model.Person;

public class JpaTest {

      private static final String PERSISTENCE_UNIT_NAME = "people";
      private EntityManagerFactory factory;

      @Before
      public void setUp() throws Exception {
            factory =
Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
            EntityManager em = factory.createEntityManager();

            // Begin a new local transaction so that we can persist
a new entity
            em.getTransaction().begin();

            // Read the existing entries
```

```
		Query q = em.createQuery("select m from Person m");
		// Persons should be empty

		// Do we have entries?
		boolean createNewEntries = (q.getResultList().size() ==
0);

		// No, so lets create new entries
		if (createNewEntries) {
			assertTrue(q.getResultList().size() == 0);
			Family family = new Family();
			family.setDescription("Family for the Knopfs");
			em.persist(family);
			for (int i = 0; i < 40; i++) {
				Person person = new Person();
				person.setFirstName("Jim_" + i);
				person.setLastName("Knopf_" + i);
				em.persist(person);
				// Now persists the family person
relationship
				family.getMembers().add(person);
				em.persist(person);
				em.persist(family);
			}
		}

		// Commit the transaction, which will cause the entity
to
		// be stored in the database
		em.getTransaction().commit();

		// It is always good practice to close the
EntityManager so that
		// resources are conserved.
		em.close();

	}

	@Test
	public void checkAvailablePeople() {
```

```
		// Now lets check the database and see if the created
entries are there
		// Create a fresh, new EntityManager
		EntityManager em = factory.createEntityManager();

		// Perform a simple query for all the Message entities
		Query q = em.createQuery("select m from Person m");

		// We should have 40 Persons in the database
		assertTrue(q.getResultList().size() == 40);

		em.close();
	}

	@Test
	public void checkFamily() {
		EntityManager em = factory.createEntityManager();
		// Go through each of the entities and print out each
of their
		// messages, as well as the date on which it was
created
		Query q = em.createQuery("select f from Family f");

		// We should have one family with 40 persons
		assertTrue(q.getResultList().size() == 1);
		assertTrue(((Family)
q.getSingleResult()).getMembers().size() == 40);
		em.close();
	}

	@Test(expected = javax.persistence.NoResultException.class)
	public void deletePerson() {
		EntityManager em = factory.createEntityManager();
		// Begin a new local transaction so that we can persist
a new entity
		em.getTransaction().begin();
		Query q = em
				.createQuery("SELECT p FROM Person p
WHERE p.firstName = :firstName AND p.lastName = :lastName");
		q.setParameter("firstName", "Jim_1");
		q.setParameter("lastName", "Knopf_!");
```

```
                Person user = (Person) q.getSingleResult();
                em.remove(user);
                em.getTransaction().commit();
                Person person = (Person) q.getSingleResult();
                // Begin a new local transaction so that we can persist
a new entity

                em.close();
        }
}
```

O projeto agora deve ser semelhante ao seguinte.



Você deve ser capaz de executar com sucesso os testes JUnit.

5. Agradecimentos

http://www.icmc.usp.br/~ricardoramos/

6. Perguntas e Discussão

Antes de postar perguntas, consulte o FAQ http://www.icmc.usp.br/~ricardoramos/. Se você tiver dúvidas ou encontrar um erro neste artigo, por favor envie um e-mail para: ricardoramos.usp@gmail.com.

7. Links e Literatura

7.1. Java Persistence API Recursos

http://www.eclipse.org/eclipselink/ EclipseLink
http://java.sun.com/developer/technicalArticles/J2SE/Desktop/persistenceapi/ Usando o Java Persistence API em aplicações desktop
http://www.ibm.com/developerworks/java/library/j-typesafejpa/index.html dinâmico, consultas typesafe em JPA 2.0 usando a API Criteria

7.2. vogella Recursos

Eclipse RCP Treinamento Junte-se a minha formação Eclipse RCP para se tornar um especialista RCP em 5 dias (Formação em alemão)
Android tutorial de introdução ao Android Programação
GWT Programa Tutorial em Java e compilar para JavaScript e HTML
Eclipse RCP Tutorial criar aplicativos nativos em Java
JUnit Tutorial Teste o seu aplicativo
Git Tutorial Coloque tudo que você tem sob o sistema de controle de versão distribuído