Java Persistence Api – Operaciones CRUD

Objetivo

- Presentar conceptos básicos de JPA y su estructura de funcionamiento.
- Ejecutar una aplicación básica para realizar operaciones CRUD con JPA y PostgreSQL

Aprendizaje

Luego de la lectura de esta guía el alumno podrá

- Explicar los conceptos básicos de la especificación JPA
- Codificar un programa básico que realice operaciones CRUD usando EclipseLink y PostgreSQL

Contenido

Estructura de Interfaces JPA

En la guía destinada al Mapeo Objeto-Relacional se presentó los pasos para establecer una conexión a una base de datos PostgreSQL desde el IDE Eclipse.

Al momento de realizar las operaciones con los datos (CRUD) en la aplicación, se debe tener en cuenta la estructura de interfaces que presenta JPA. La figura 1 muestra un esquema simple de las mismas.

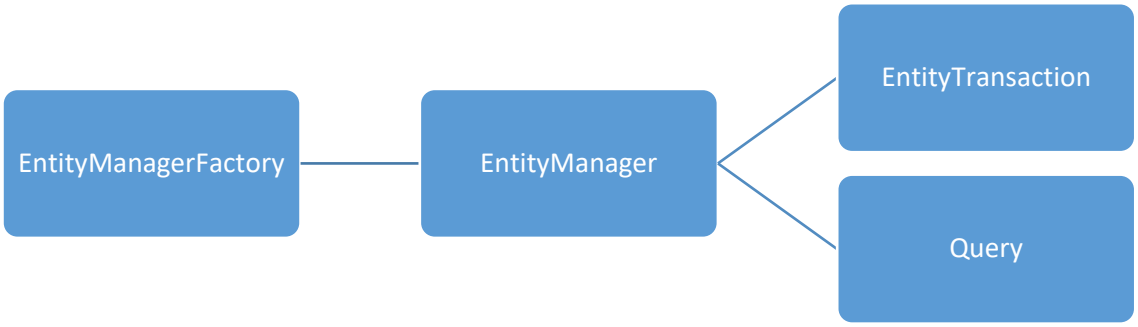

Figura 1. Estructura de Interfaces JPA

La conexión a una base de datos se representa por una instancia de EntityManager, la cual provee funcionalidad para realizar operaciones sobre la base de datos. En una aplicación web, por ejemplo, es común establecer conexiones separadas a base de datos, usando instancias separadas de EntityManager para cada Request.

EntityManagerFactory

EntityManagerFactory tiene como función principal, generar instancias de EntityManager. Un EntityManagerFactory está destinado a una base de datos específica, y provee una manera eficiente de construir múltiples instancias de EntityManager para esa base de datos. EntityManagerFactory se instancia solo una vez y sirve a toda la aplicación.

Existen, en general, dos tipos de operaciones que se ejecutan contra una base de datos:

- Aquellas que modifican su contenido (Insert, Update, Delete)
- Aquellas que consultan información (Select)

Las operaciones que modifican el contenido de la base de datos requieren transacciones activas. Estas son gestionadas por una instancia de EntityTransaction obtenida desde el EntityManager

Asimismo, una instancia de EntityManager funciona como generador de instancias de Query, las cuales son necesarias para ejecutar consultas sobre la base de datos.

Cada implementación de JPA define clases que implementan estas interfaces.

Para obtener una instancia de EntityManagerFactory, se invoca al método createEntityManagerFactory, el cual es parte de JPA. Este método toma como argumento el nombre de la unidad de persistencia (valor establecido en el archivo persistence.xml)

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("PersUnitName");
```

Cuando se construye la instancia de EntityManagerFactory, se abre la base de datos. Si la misma no existe, una nueva base de datos es creada.

Cuando la aplicación finaliza el uso de EntityManagerFactory, éste debe ser cerrado

```
emf.close();
```

Cerrando el EntityManagerFactory, la base de datos se Cierra.

EntityManager

Una instancia de EntityManager representa una conexión a una base de datos específica. Para establecer la conexión, se puede realizar con el siguiente código:

```
EntityManager em = emf.createEntityManager();
try {
    // TODO: Código accediendo a la base de datos
}
finally {
    em.close();
}
```

La instancia de EntityManager se obtiene de EntityManagerFactory correspondiente.

EntityTransaction

Las operaciones que modifican el contenido de la base de datos (insert, update, delete) deben ser realizadas dentro de transacciones activas. La interface EntityTransaction representa y administra dichas transacciones. Cada EntityManager tiene una sola instancia de EntityTransaction asociada, la cual se invoca desde el método getTransaction.

```
try {
    em.getTransaction().begin();
    // Operaciones que modifican la base de datos van aquí
    em.getTransaction().commit();
}
finally {
```

```
    if (em.getTransaction().isActive())
        em.getTransaction().rollback();
}
```

Una transacción comienza con la llamada a “begin” y finaliza con un “commit” o “rollback” de la misma. Todas las operaciones realizadas entre un begin y un commit (o rollback) se mantienen en memoria hasta que la transacción finaliza. Si la misma finaliza con un rollback, los cambios realizados no son reflejados en la base de datos (aunque la instancia en memoria de las entidades modificadas no vuelven a su estado previo a ser modificadas)

Finalizando la transacción con un commit, genera el registro físico de los cambios en la base de datos.

Manejo de Objetos Entidad

Los Objetos Entidad son instancias en memoria de Clases de Entidad (Clases definidas por el usuario, que son persistentes). Estos Objetos Entidad son la representación de objetos almacenados físicamente en la base de datos

El ciclo de vida de los Objetos Entidad consiste en cuatro estados: *New, Managed, Removed,* y *Detached*. En la figura 2 se muestra el diagrama de estados y sus transiciones.

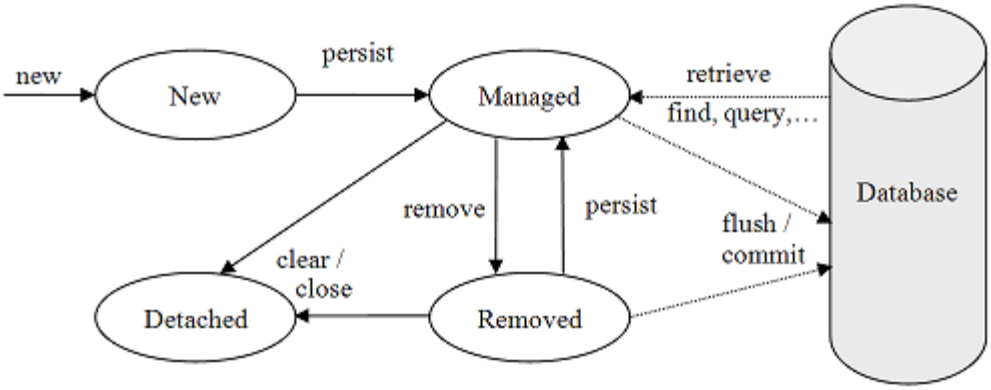

Figura 2. Estados y transiciones de Objetos Entidad

Cuando un Objeto Entidad se inicializa, si estado es **New**. En este estado, el objeto aún no está asociado con un EntityManager y no tiene representación en la base de datos.

Un Objeto Entidad se convierte en **Managed** cuando se envía a persistir en la base de datos a través del método persist de un EntityManager, el cual se invoca desde una transacción activa. Cuando la transacción realiza el commit, el EntityManager almacena el Objeto Entidad en la base de datos. Los Objetos Entidad recuperados desde la base de datos, también se encuentran en estado Managed.

Si un Objeto Entidad (en estado Managed) es modificado dentro de una transacción activa, los cambios son detectados por el correspondiente EntityManager, y la actualización se propaga a la base de datos al realizarse el commit.

Un Objeto Entidad recuperado de la base de datos puede ser marcado para ser borrado por el EntityManager a través del método remove durante una transacción activa. El Objeto Entidad cambia su estado de Managed a **Removed**, y es físicamente borrado de la base de datos cuando se realiza el commit de la transacción.

El estado **Detached** indica que el Objeto Entidad se desconectó del EntityManager que lo gestionaba. Por ejemplo, todos los objetos contenidos en un EntityManager pasan a estado Detached cuando se cierra el EntityManager. Estos pueden seguir siendo utilizados en memoria, pero no representan la última versión de los objetos persistentes en la base de datos.

Persistence Context

Persistence Context es la colección de todos los Managed Objects de un EntityManager. Si se intenta consultar un Objeto Entidad que había sido previamente recuperado, y existe en el Persistence Context, se devuelve esta instancia en lugar de acceder a la base de datos nuevamente (a excepción de consultas a través de refresh, la cual accede siempre a la base de datos)

El rol principal del Persistence Context es asegurarse que un Objeto Entidad no se represente más de una vez en memoria dentro del mismo EntityManager. Cada EntityManager administra su propio Persistence Context. Por tanto, un objeto de la base de datos puede representarse con distintos Objetos Entidad, pero solo en distintas instancias de EntityManager.

Otra forma de considerar el Persistence Context es comparar su funcionamiento como una memoria caché local de un EntityManager específico.

Por defecto, los Objetos Entidad en estado Managed que no han sido modificados o removidos durante una transacción, son mantenidos en el Persistence Context. De todas maneras, cuando uno de ellos no es utilizado por la aplicación, el garbage collector los remueve del Persistence Context.

El método *contains* permite corroborar si un Objeto Entidad específico está en el Persistence Context.

```
boolean isManaged = em.contains(employee);
```

El Persistence Context puede limpiarse usando el método *clear*, de la manera siguiente:

```
em.clear();
```

Cuando un Persistence Context se limpia, todas sus entidades en estado Managed pasan a estado Detached, y los cambios hechos en Objetos Entidad que no fueron confirmados en la base de datos son descartados.

Operaciones CRUD con JPA

El término CRUD es un acrónimo de **C**reate, **R**ead, **U**pdate, **D**elete, y representa las acciones que pueden realizarse sobre una base de datos. En esta sección se presenta la forma de llevar adelante estas acciones desde una aplicación Java, usando una implementación de JPA, contra una base de datos.

Almacenar Objetos Entidad

Nuevos objetos entidad se almacenan en una base de datos ya sea invocando explícitamente el método *persist*, o implícitamente, como resultado de una operación de inserción en cascada

De manera explícita, el siguiente código almacena una instancia de un objeto Empleado en la base de datos.

```
Empleado elEmpleado = new Empleado("Juan", "Pérez");
  em.getTransaction().begin();
  em.persist(elEmpleado);
  em.getTransaction().commit();
```

La instancia del objeto Empleado se construye normalmente como un objeto Java, y su estado es **New**. Una llama explícita a *persist* asocia el objeto con el EntityManager (*em*) y cambia su estado a **Managed**. El nuevo objeto se almacena en la base de datos cuando se realiza el commit de la transacción.

Si el parámetro enviado en el método *persist* no es una instancia de una clase de entidad, ocurre una excepción de tipo *IllegalArgumentException*. Esto es debido a que sólo clases de entidad (u objetos “persistibles” embebidos) pueden ser almacenadas en la base de datos.

Si no hay una transacción activa cuando se llama al método *persist*, ocurre una excepción de tipo *TransactionRequiredException*, debido a que las operaciones que modifican la base de datos requieren una transacción activa.

Si la base de datos contiene otra entidad del mismo tipo, con la misma clave primaria, ocurre una excepción de tipo *EntityExistException*. Esta excepción puede ocurrir en la llamada a *persist* (si el objeto está en el persistence context) o en la llamada al commit.

Existe un tipo de clases llamadas “embebidas”, las cuales son tipos definidos por el usuario, persistentes, y que funcionan como valores de una instancia de una clase. Las instancias de una clase embebida solo pueden almacenarse en la base de datos como parte de un objeto que las contiene. Dichas clases no tienen un identificador o clave primaria, por lo que no se pueden compartir con distintos objetos, y no pueden ser consultadas directamente.

El siguiente código muestra el almacenamiento de una instancia Empleado con una referencia a una instancia de Dirección, la cual es una clase Embebida.

```
Empleado elEmpleado = new Empleado ("Juan", "Pérez");
Dirección laDireccion = new Direccion ("Chilecito", "La Rioja");
elEmpleado.setAddress(laDireccion);
em.getTransaction().begin();
em.persist(elEmpleado);
em.getTransaction().commit();
```

Las instancias de clases embebidas son almacenadas automáticamente, embebidas en el objeto entidad que las contiene.

Insertar Objetos Entidad referenciados

Los casos más comunes de modelo de datos presentan clases persistentes con referencia a otras clases persistentes. Siguiendo el ejemplo recientemente mostrado, en el caso de que la clase Dirección sea persistente, el código anterior no almacenaría la referencia a la instancia de Dirección.

Para evitar una referencia inconsistente en la base de datos, se genera una excepción de tipo *IllegalStateException* en el commit si un Objeto Entidad persistente debe almacenarse en la base de datos en una transacción y hace referencia a otro Objeto Entidad que no será almacenado en la base de datos al final de transacción.

Es responsabilidad de la aplicación verificar que cuando se almacene un objeto en la base de datos, se almacene toda su estructura de objetos referenciados. Esto se puede hacer a través de persistencia explícita de cada objeto, o estableciendo una política de inserción en cascada.

Agregando a un atributo de referencia la anotación *CascadeType.PERSIST* (o *CascadeType.ALL*, que incluye a *PERSIST*), se indica que las operaciones *persist* deben contemplar los objetos referenciados por ese atributo (o atributos en el caso de colecciones)

```
@Entity
class Empleado {
     :
    @OneToOne(cascade=CascadeType.PERSIST)
    private Direccion direccion;
     :
}
```

En el ejemplo, la case *Empleado* contiene un atributo *dirección* que referencia a la clase Direccion, la cual es otra Clase de Entidad. Cuando se persiste una instancia de *Empleado*, automáticamente se procede a la inserción en cascada de la instancia de *Direccion*, la cual es también insertada (en caso de que no exista en la base de datos) sin necesidad de invocar de manera separada a *persist*.

Recuperar Objetos

JPA provee distintas maneras de recuperar los objetos de la base de datos. Esta recuperación no requiere una transacción activa debido a que no modifica el contenido de la base de datos.

El Persistence Context funciona como una memoria caché de los Objetos Entidad recuperados. Si un objeto consultado no está en el contexto, se construye y completa sus atributos desde la base de datos. Luego este objeto se agrega al contexto como Objeto Entidad en estado Managed, y se remite a la aplicación.

Recuperar Objetos por clave primaria

Cada Objeto Entidad se identifica de manera unívoca a través de su clave primaria. El siguiente código muestra la obtención de un objeto de tipo *Empleado*, cuya clave primaria es *1*

```
Empleado elEmpleado = em.find(Empleado.class, 1);
```

No es necesaria la conversión al tipo de dato *Empleado* debido a que el método *find* devuelve una instancia del mismo que fue enviado como parámetro. Si la clase especificada como parámetro no es una Clase de Entidad, ocurre una excepción de tipo *IllegarArgumentException.*

Si el EntityManager ya contenía un objeto como el que se pretende consultar, no se requiere consulta a la base de datos, y se retorna el objeto del Persistence Context. De otra forma, se realiza la consulta a la base de datos, se construye el objeto y se remite a la aplicación. Si no se encuentra el objeto en la base de datos, se devuelve *null*.

Política de recuperación Eager y Lazy

Recuperar objetos de la base de datos puede causar que se recuperen Objetos Entidad adicionales. Por defecto, una consulta automáticamente recupera en cascada todos los atributos que no sean colecciones o Maps (es decir, relaciones one-to-one y many-to-one). Este tipo de recuperación se denomina *eager* (del inglés, ansioso). Por lo tanto, cuando un objeto se recupera, es posible navegar a todos sus atributos que no sean colecciones o Maps. De forma teórica, en casos extremos podría causar la recuperación de toda la base de datos en memoria, lo que resultaría problemático e inaceptable.

Un atributo de referencia persistente puede excluirse de esta recuperación en cascada usando el tipo de recuperación *lazy* (del inglés, perezoso)

```
@Entity
class Empleado {
   :
  @ManyToOne(fetch=FetchType.LAZY)
```

```
  private Empleado manager;
   :
}
```

Indicando la recuperación de modo *lazy*, excluye el atributo para ser recuperado en cascada.

Cuando un Objeto Entidad se recupera, todos sus atributos persistentes se inicializan. Una referencia anotada como *lazy* inicializa un objeto referenciado vacío (a menos que el objeto ya esté referenciado en el Persistence Context). Un objeto vacío (o *hollow* en inglés) solo contiene el valor de la clave primaria, mientras que el resto de sus atributos no se inicializan hasta que se intenta acceder a uno de ellos. En el ejemplo anterior, cuando se recupera una instancia `Empleado`, su atributo manager podría hacer referencia a una instancia de Empleado vacía.

Por otra parte, la política por defecto para recuperar colecciones y Maps es *lazy*. Por lo tanto, al recuperar un Objeto Entidad con atributo de tipo colección, los objetos referenciados a través de dicha colección no son recuperados. Esto puede cambiarse definiendo explícitamente esos atributos con el modo de recuperación *eager*.

```
@Entity
class Empleado {
   :
  @ManyToMany(fetch=FetchType.EAGER)
  private List<Proyecto> proyectos;
   :
}
```

En el ejemplo anterior, cuando una instancia de *Empleado* se recupera, todas sus instancias de *Proyecto* referenciadas son recuperadas también.

De una manera u otra, la política de recuperación afecta el desempeño de la aplicación. Recuperar en modo *eager* minimiza los accesos a la base de datos mejorando la performance, pero puede recuperar innecesariamente objetos que no serán usados, lo que reduce la performance.

Recuperación por Consultas

La forma más flexible de recuperar objetos de la base de datos es usar consultas. JPA tiene un lenguaje específico de consultas llamado JPQL (Java Persistence Query Language). Este lenguaje permite realizar tanto consultas simples, como complejas.

En JPA 2 existen dos tipos de interfaces de consulta. La antigua *Query* y la nueva *TypedQuery*, que extiende la funcionalidad de la primera. Cuando no se conoce el tipo de dato que devolverá la consulta, se utiliza *Query*. Mientras que cuando se espera un tipo de valor específico, generalmente se usa la interfaz *TypedQuery*.

Como la mayoría de las operaciones en JPA, la ejecución de consultas requiere de un EntityManager, que sirve tanto para la interfaz *Query* como para la interfaz *TypedQuery*.

```
Query q1 = em.createQuery("SELECT c FROM Pais c");
TypedQuery<Pais> q2 = em.createQuery("SELECT c FROM Pais c", Pais.class);
```

Ambas interfaces definen dos métodos para ejecutar consultas SELECT:

- *getSingleResult*, para usar cuando se espera recuperar exactamente un objeto
- *getResultList*, para uso general en cualquier otro escenario

Los códigos siguientes muestran la realización de consultas usando JPQL

```
TypedQuery<Pais> consulta= em.createQuery("SELECT c FROM Pais c", Pais.class);
List<Pais> resultado = consulta.getResultList();
Query consulta = em.createQuery("SELECT c FROM Pais c");
List resultado = consulta.getResultList();
```

Para búsquedas más refinadas, la sintaxis de JPQL es simple de extender. El siguiente código muestra la recuperación de datos agregando filtro por un atributo de la clase

```
Query consulta = em.createQuery("SELECT c FROM Pais c WHERE c.nombre= 'Argentina'");
```

La colección resultante funciona como cualquier colección en Java, y puede ser recorrida con cualquier estructura repetitiva

```
for (Pais c : resultado) {
      System.out.println(c.getNombre());
}
```

Para más información sobre consultas JPQL remitirse al siguiente link: Estructura de consultas JPQL

Actualizar Objetos

Modificar Objetos Entidad existentes en la base de datos se basa en persistencia transparente. Esto significa que los cambios son detectados y manejados automáticamente

Una vez que el Objeto Entidad es recuperado de la base de datos (indistintamente de la forma de recuperación), puede modificarse en memoria dentro de una transacción activa. El objeto es físicamente actualizado cuando se produce el commit de la transacción. Si en la transacción ocurre un rollback, los cambios son descartados.

Configurando la inserción en cascada (CascadeType.PERSIST o CascadeType.ALL) implica que los cambios se verán reflejados incluyendo los objetos modificados.

```
Empleado elEmpleado= em.find(Empleado.class, 1);
em.getTransaction().begin();
elEmpleado.setApodo("Juan Pintor");
em.getTransaction().commit();
```

Como se visualiza, una actualización se lleva a cabo modificando un Objeto Entidad dentro de una transacción activa. No se requiere un método específico del EntityManager para llevar a cabo esta acción.

De la misma manera que para recuperar objetos, es posible utilizar consultas JPQL para actualizar objetos en la base de datos. Más información en el siguiente link: JPQL Update Queries

Borrar Objetos

El borrado de objetos de la base de datos, se realiza explícitamente invocando al método *remove* del EntityManager, o implícitamente como resultado de una operación en cascada.

Para borrar un objeto, primero debe ser recuperado de la base de datos (indistintamente del método de consulta), y luego, en una transacción activa, invocar al método *remove* del EntityManager. El objeto es removido físicamente de la base de datos cuando se realiza el commit de la transacción. Objetos embebidos, contenidos dentro del Objeto Entidad, también son borrados. Si la transacción no se realiza y ocurre un rollback, el objeto no se borra de la base de datos.

```
Empleado elEmpleado = em.find(Empleado.class, 1);
  em.getTransaction().begin();
  em.remove(elEmpleado);
  em.getTransaction().commit();
```

Si el parámetro enviado al método *remove* no es una instancia de una Clase de Entidad (o si es una Entidad en estado Detached), ocurre una excepción de tipo *IllegarArgumentException*.

Si no hay una transacción activa cuando se llama al método *remove*, ocurre una excepción de tipo *TransactionRequiredException*, debido a que operaciones que modifican el contenido de la base de datos requieren una transacción activa.

Anotando un campo de referencia con CascadeType.REMOVE (o CascadeType.ALL, que incluye a REMOVE) se indica que una operación de borrado debe borrar en cascada aquellos objetos referenciados.

```
@Entity
class Empleado {
     :
    @OneToOne(cascade=CascadeType.REMOVE)
    private Direccion direccion;
     :
}
```

En el ejemplo anterior, cuando una instancia de Empleado es borrada, y debido a la anotación de borrado en cascada, automáticamente se borrará la instancia de Direccion asociada al objeto Empleado. Adicionalmente, de manera recursiva se borrarán todos los objetos referenciados de Direccion, y demás clases asociadas, si hubiera).

De la misma forma que con las actualizaciones, es posible borrar objetos mediante consultas. Más información en el siguiente link: JPQL Delete Queries