2. Java Persistence API (JPA)

Índice

- Referencias
- Introducción
- Entidades y su gestión
- Mapeado objeto-relacional
- Entity manager
- Concurrencia
- Java Persistence Query Language (JP QL)
- Modelado en IBM RSA

Referencias

- Keith, M., Schincariol, M. *Pro JPA 2. Mastering the Java Persistence Api*. Apress, 2009

Introducción

- Los patrones de arquitectura multicapa pueden ser implementados con POJOs o con marcos
- Por ejemplo, para la capa de presentación, *Java Server Faces* (JSF) implementa a los controladores frontales y de aplicación, así como a unos cuantos ayudantes de vista

Introducción

- Si queremos exponer los servicios de aplicación como objetos remotos tenemos frameworks como *Enterprise Java Beans* (EJB) 3
- Si es como servicios web, disponemos de *Java API for XML Web Services* (JAX WS) y de *Java API for RESTful Web Services* (JAX RS)
- Finalmente, disponemos de *Java Persistence Api* (JPA) para implementar al almacén del dominio, y a los objetos del dominio

Introducción

- Esto es en tecnología Java, en tecnología .NET Microsoft tiene marcos similares para patrones similares
- De todos los marcos J2EE veremos JPA, ya que:
  - El almacén del dominio es complejo de implementar
  - Sin embargo, los objetos del negocio proporcionan una aproximación OO plena

Entidades y su gestión Definición

- Una *entidad* es un objeto del negocio persistente
- Sus *instancias*:
  - Tienen un identificador único
  - Son *cuasitransaccionales*
    - Son transaccionales con respecto al almacén persistente
    - No son transaccionales en memoria
  - Son *lígeros*: equivalentes a clases, y no a tablas de 500 columnas

Entidades y su gestión Metadatos

- Las entidades tienen metadatos asociados, que describen su relación entre la representación en memoria y en almacén persistente
- Puede definirse:
  - Con anotaciones
  - En un fichero XML (`PROJECT_HOME/src/META-INF/orm.xml`)

Entidades y su gestión
Metadatos

- Las anotaciones son más sencillas, al no replicar información, sin embargo, acoplan el código con los metadatos
- Los ficheros XML pueden utilizarse para anular las anotaciones
- En cualquier caso, JPA funciona bajo el principio de *configuración por excepción*: todo está ya configurado, y el usuario puede ajustar lo que necesite

Entidades y su gestión
Creación de una entidad

- Las clases Java se transforman fácilmente en entidades anotándolas
- Eso sí, necesitan, al menos, un constructor sin argumentos

Entidades y su gestión
Creación de una entidad

```
package examples.model;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Employee {
    @Id
    private int id;
    private String name;
    private long salary;
    public Employee() {}
    public Employee(int id) { this.id = id; }
```


Entidades y su gestión
Creación de una entidad

```
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public long getSalary() { return salary; }
    public void setSalary(long salary) { this.salary =
                                                salary; }
    public String toString() {return "Employee id: " +
                          getId() + " name: " + getName() +
                          " salary: " + getSalary(); }
}
```

Entidades y su gestión
Entity manager

- La gestión de entidades se hace en JPA a través del *entity manager*, que es el persistence manager de JPA
- Cuando un entity manager obtiene una referencia a una instancia de entidad, bien como parámetro de un método suyo, bien leyéndola desde la base de datos, la instancia pasa a ser *gestionada* por ese entity manager

Entidades y su gestión
Entity manager

- Al conjunto de instancias de entidades gestionadas por un entity manager se le denomina contexto de persistencia
- Sólo una instancia Java con el mismo identificador de entidad puede existir en un contexto de persistencia en un momento dado
- Los entity manager tienen que ser implementados por un *proveedor de persistencia*

Entidades y su gestión
Entity manager

- Los entity managers se crean a través de factorías
- La configuración del entity manager se hace a través de la factoría que lo creó, y se define a través de la *unidad de persistencia*
- Dicha unidad determina las características de las entidades gestionadas por el entity manager

Entidades y su gestión
Entity manager

- Así, hay una correspondencia uno a uno entre las unidades de persistencia y las factorías de entidades
- Las unidades de persistencia tienen nombre para distinguir las factorías de entidades, dando a la aplicación posibilidad de elegir entre distintas opciones

Entidades y su gestión
Entity manager

```
package examples.model;
import java.util.Collection;
import javax.persistence.EntityManager;
import javax.persistence.TypedQuery;

public class EmployeeService {

    protected EntityManager em;
    public EmployeeService(EntityManager em) {  this.em = em;
    }
```


Entidades y su gestión
Entity manager

```
      public Employee createEmployee(int id, String name,
long salary) {
        Employee emp = new Employee(id);
        emp.setName(name);
        emp.setSalary(salary);
        em.persist(emp);
        return emp;
    }

    public void removeEmployee(int id) {
        Employee emp = findEmployee(id);
        if (emp != null) {
            em.remove(emp);
        }
    }
```


Entidades y su gestión
Entity manager

```
      public Employee raiseEmployeeSalary(int id, long
raise) {
        Employee emp = em.find(Employee.class, id);
        if (emp != null) {
            emp.setSalary(emp.getSalary() + raise); }
        return emp;  }

    public Employee findEmployee(int id) {
        return em.find(Employee.class, id);   }

    public List<Employee> findAllEmployees() {
        TypedQuery query = em.createQuery("SELECT e FROM
                           Employee e", Employee.class);
        return query.getResultList();
    }
}
```


Entidades y su gestión
Entity manager

```
package examples.client;
import java.util.Collection;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import examples.model.Employee;
import examples.model.EmployeeService;

public class EmployeeTest {

    public static void main(String[] args) {
        EntityManagerFactory emf =

Persistence.createEntityManagerFactory("EmployeeService");
        EntityManager em = emf.createEntityManager();
        EmployeeService service = new EmployeeService(em);
```

Entidades y su gestión
Entity manager

```
        //  create and persist an employee
         em.getTransaction().begin();
         Employee emp = service.createEmployee(158, "John
Doe", 45000);
         em.getTransaction().commit();
         System.out.println("Persisted " + emp);

         // find a specific employee
         emp = service.findEmployee(158);
         System.out.println("Found " + emp);
```


Entidades y su gestión
Entity manager

```
        // find all employees
         Collection<Employee> emps =
service.findAllEmployees();
         for (Employee e : emps)
             System.out.println("Found Employee: " + e);

         // update the employee
         em.getTransaction().begin();
         emp = service.raiseEmployeeSalary(158, 1000);
         em.getTransaction().commit();
         System.out.println("Updated " + emp);
```


Entidades y su gestión
Entity manager

```
        // remove an employee
         em.getTransaction().begin();
         service.removeEmployee(158);
         em.getTransaction().commit();
         System.out.println("Removed Employee 158");

         // close the EM and EMF when done
         em.close();
         emf.close();
    }
}
```


Entidades y su gestión
Despliegue

- Las unidades de persistencia se definen en un fichero XML llamado `PROJECT_HOME/src/META-INF/persitence.xml`
- Las unidades de persistencia tienen nombre para que puedan ser invocadas por las aplicaciones
- Así un único fichero puede tener varias configuraciones de unidades de persistencia distintas

Entidades y su gestión
Despliegue

- La información incluida en el fichero varía si estamos utilizando JPA en el contexto de un servidor de aplicaciones o no

Entidades y su gestión
Despliegue

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
    <persistence-unit name="EmployeeService" transaction-type="RESOURCE_LOCAL">

        <class>examples.model.Employee</class>
         <class>examples.model.Departement</class>

        <validation-mode>NONE</validation-mode>

         <properties>
            <property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.ClientDriver"/>
            <property name="javax.persistence.jdbc.url"
value="jdbc:derby://localhost:1527/EmpServDB;create=true"/>
            <property name="javax.persistence.jdbc.user" value="APP"/>
            <property name="javax.persistence.jdbc.password" value="APP"/>
            <!-- enable this property to see SQL and other logging -->
            <!-- property name="eclipselink.logging.level" value="FINE"/ -->
```


Entidades y su gestión
Despliegue

```
        <property name="eclipselink.ddl-generation" value="drop-and-create-
        tables"/>
        <property name="eclipselink.create-ddl-jdbc-file-name"
        value="createDDL_ddlGeneration.jdbc"/>
        <property name="eclipselink.drop-ddl-jdbc-file-name"
        value="dropDDL_ddlGeneration.jdbc"/>
        <property name="eclipselink.ddl-generation.output-mode"
        value="both"/>
         </properties>
    </persistence-unit>
</persistence>
```


Entidades y su gestión
Despliegue

- Las últimas anotaciones del fichero anterior permiten crear el esquema JPA
- Dicho esquema se crea cada vez que se llama a `Persistence.createEntityManagerFactory()`

Mapeado objeto-relacional
Anotaciones de persistencia

- Las anotaciones de persistencia pueden aplicarse a tres niveles:
  - Clase
  - Método
  - Campo
- Deben situarse:
  - En la misma línea que el elemento anotado
  - Sobre el elemento anotado

Mapeado objeto-relacional
Anotaciones de persistencia

- Las anotaciones pueden ser:
  - Lógicas: indican información del dominio
  - Físicas: indican información del mapeado a la base de datos

Mapeado objeto-relacional
Modo de acceso

- La implementación de JPA tiene que acceder en ejecución al estado mapeado de una instancia de entidad
- Hay tres *modos de acceso*:
  - Acceso a campo
  - Acceso a propiedad
  - Acceso mixto

Mapeado objeto-relacional
Modo de acceso

- En el *acceso a campo* la implementación JPA accede a los campos de la entidad directamente
  - Los campos deben ser `protected`, `package` o `private`
  - Los clientes acceden a través de setters/getters
  - La propia clase solo debería acceder a sus campos directamente en el constructor
  - Ejemplo:

Mapeado objeto-relacional
Modo de acceso

```
@Entity
public class Employee {
    @Id
    private int id;
    private String name;
    private long salary;
    public Employee() {}
    public Employee(int id) { this.id = id; }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
……
}
```


Mapeado objeto-relacional
Modo de acceso

- En el *acceso a propiedad* la implementación JPA accede a los campos a través de los setters/getters de la clase
  - Dichos setters/getters son obligatorios, y pueden ser `public` o `protected`
  - La anotación debe hacerse en el getter
  - Ejemplo:

Mapeado objeto-relacional
Modo de acceso

```
@Entity
public class Employee {
    private int id;
    private String name;
    private long salary;

    @Id
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
……
}
```


Mapeado objeto-relacional
Modo de acceso

- En el *acceso mixto* se combinan ambos tipos de acceso
  - Puede ser útil para modificar clases ya hechas
  - La anotación `@Access` permite invalidar el tipo de acceso previamente definido

Mapeado objeto-relacional
Mapeado a tabla

- Las únicas anotaciones necesarias para mapear una entidad a una tabla son:
  - `@Entity`
  - `@Id`
- En ese caso, los nombres de la tabla y las columnas coinciden con los de la clase y sus atributos

Mapeado objeto-relacional
Mapeado a tabla

- Para cambiar el nombre de la tabla, simplemente hay que indicarlo en la anotación:

```
@Entity
@Table(name="EMP")
public class Employee {….}
```


Mapeado objeto-relacional
Mapeado de tipos simples

- Los tipos simples de Java se mapean directamente
- Opcionalmente (por fines de documentación) se pueden anotar con la etiqueta `@Basic`
- Más adelante veremos que los atributos pueden anotarse para definir las columnas de la tabla
- Ahora nos centraremos en el acceso

Mapeado objeto-relacional
Mapeado de tipos simples

- Para indicar el nombre de la columna en la base de datos, simplemente hay que indicarlo en la anotación:

```
@Entity
public class Employee {
 @Id
 @Column(name="EMP_ID")
 private int id;
 private String name;
……
}
```

Mapeado objeto-relacional
Mapeado de tipos simples

- Los tipos simples también pueden cargarse bajo demanda (`LAZY/EAGER`):

```
@Entity
public class Employee {
……
@Basic(fetch=FetchType.LAZY)
private String comments;
…… }
```

- La anotación `@Transient` permite especificar atributos que no tienen que ser almacenados en la base de datos

Mapeado objeto-relacional
Mapeado de la clave primaria

- Toda entidad mapeada a la base de datos debe tener un mapeado a un clave primaria en la tabla
- Se puede incluir distintas estrategias de generación de valores de claves. Por ej:

```
@Entity
public class Employee {
 @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
 private int id;
 …… }
```


Mapeado objeto-relacional
Mapeado de la clave primaria

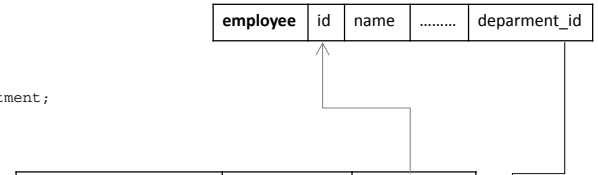
- ¿Cómo se accede al identificador de una instancia de entidad recién creada? Haciendo un `getId()` sobre la entidad, después del commit

Mapeado objeto-relacional
Mapeado de relaciones

- Las relaciones pueden ser:
  - N a 1
  - 1 a N
  - 1 a 1
  - M a N

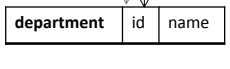
Mapeado objeto-relacional
Mapeado de relaciones

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    …….
    @ManyToOne
    private Department department;
…… }
```




```
@Entity
public class Department {
    @Id private int id;
    private String name;

   @OneToMany(mappedBy="department")
    private Collection<Employee> employees;
……}
```



Si no pone el atributo `mappedBy`, JPA crea esta tabla de join para soportar la relación `@OneToMany`

Mapeado JPA de una relación N a 1 y 1 a N

Mapeado objeto-relacional
Mapeado de relaciones

- Relaciones N a 1
  - Se utiliza la anotación `@ManyToOne`
  - Ejemplo: muchos empleados pertenecen a un único departamento

Mapeado objeto-relacional
Mapeado de relaciones

```
@Entity
public class Employee {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String name;
    private long salary;

    @ManyToOne
    private Department department;

……..
}
```


Mapeado objeto-relacional
Mapeado de relaciones

```
@Entity
public class Department {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String name;

    public int getId() {
        return id;
    }

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

Mapeado objeto-relacional
Mapeado de relaciones

- La columna que soporta la referencia de la clave extranajera en la relación N a 1, se llama *columna de join* en JPA
- Si ya existe el esquema de base de datos, y difiere del nombre del atributo puede utilizarse la anotación `@JoinColumn` para referirse a dicha columna

Mapeado objeto-relacional
Mapeado de relaciones

```
@Entity
public class Employee {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String name;
    private long salary;

    @ManyToOne
    @JoinColumn(name="id_departamento")
    private Department department;

……
}
```


Mapeado objeto-relacional
Mapeado de relaciones

- Relaciones 1 a N
  - Se utiliza la anotación `@OneToMany`
  - Si la relación es unidireccional, al no poder colocarse la columna de join en el lado N, se crea automáticamente una tabla de join para poder soportar las referencias del lado 1
  - Si es bidireccional, no es necesaria la tabla de join, y la columna de join se crea en el lado N
    - Esto fuerza a que el poseedor esté en el lado N

Mapeado objeto-relacional
Mapeado de relaciones

- Por tanto, habrá que poner el atributo `mappedBy` en el lado 1
- Las modificaciones en la relación sólo afectan a los cambios en la entidad del lado N
  - Así, si insertamos un empleado en un departamento, no afecta a la base de datos
  - Deberíamos asignar el departamento al empleado (puede hacerse en el mismo `Department::addEmployee()`)
- Si no lo hacemos, tendremos dos relaciones unidireccionales con una tabla de join y una columna de join
  - Esto puede provocar falta de integridad si no se hacen actualizaciones en ambos lados desde el código Java

Mapeado objeto-relacional
Mapeado de relaciones

```
@Entity
public class Employee {
    @Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String name;
    private long salary;

    @ManyToOne
    private Department department;

……..……
}
```


Mapeado objeto-relacional
Mapeado de relaciones

```
@Entity
public class Department {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String name;

       @OneToMany(mappedBy="department")
       private Collection<Employee> employees;

……..……..
}
```


Mapeado objeto-relacional
Mapeado de relaciones

- Relaciones 1 a 1
  - Se utiliza la anotación `@OneToOne`
  - Al igual, que en el caso N a 1, también existe columna de join
  - Ejemplo: empleado y plaza de aparcamiento

Mapeado objeto-relacional
Mapeado de relaciones

```
@Entity
public class Employee {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String name;
    private long salary;

    @OneToOne
    private ParkingSpace parkingSpace;

……..……
}
```

Mapeado objeto-relacional
Mapeado de relaciones

```
@Entity
public class ParkingSpace {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private int lot;
    private String location;

    public int getId() {
        return id;
    }

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


Mapeado objeto-relacional
Mapeado de relaciones

- Relaciones 1 a 1 bidireccionales
  - Se implementan con dos relaciones 1 a 1
  - Hay que indicar quien es el *poseedor* de la relación (es decir, quien contiene la columna de join)
  - Se hace a través del atributo `mappedBy` en el lado que no es el poseedor de la relación
  - Si no se incluyera el atributo, habría dos relaciones unidireccionales con dos columnas de join
    - Esto puede generar falta de integridad, si no se hacen actualizaciones en ambos lados desde el código Java

Mapeado objeto-relacional
Mapeado de relaciones

```
@Entity
public class ParkingSpace {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private int lot;
    private String location;

    @OneToOne(mappedBy="parkingSpace")
    private Employee employee

……..……}
```


Mapeado objeto-relacional
Mapeado de relaciones

- Relaciones M a N
  - Se utiliza la anotación `@ManyToMany`
  - Sea o no unidireccional, aquí siempre se crea automáticamente una tabla de join
  - Si es bidireccional, hay que establecer un poseedor de la relación, que será el que actualice automáticamente la tabla de join
    - Así, si el poseedor es empleado y se da de baja un proyecto, se desvinculan
    - Por el contrario, si de proyecto se elimina un empleado, no se desvinculan

Mapeado objeto-relacional
Mapeado de relaciones

- El poseedor de la relación será el referenciado por el atributo `mappedBy`
- Si no lo hacemos, tendremos dos relaciones unidireccionales con dos tablas de join
  - Esto puede provocar falta de integridad si no se hacen actualizaciones en ambos lados desde el código Java
- Se puede referenciar explícitamente la tabla de join con la anotación `@JoinTable`, en la entidad que posea la relación, y por tanto, la tabla de join

Mapeado objeto-relacional
Mapeado de relaciones

```
@Entity
public class Employee {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String name;
    private long salary;

    @ManyToMany
    private Collection<Project> projects;

……..……
}
```


Mapeado objeto-relacional
Mapeado de relaciones

```
@Entity
public class Project {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String name;

    @ManyToMany(mappedBy="projects")
    private Collection<Employee> employees;

……..……
}
```


Mapeado objeto-relacional
Mapeado de relaciones

- Carga dinámica de relaciones
  - Al igual que los atributos simples, las relaciones pueden cargarse bajo demanda
  - Puede especificarse para cualquier cardinalidad
  - Si es una relación de cardinalidad 1, la carga es impaciente
  - En el caso es perezosa, pero depende de la implementación
  - En relaciones bidireccionales, puede ser impaciente en un lado, y perezosa en el otro

Mapeado objeto-relacional
Mapeado de relaciones

```
@Entity
public class Employee {
    @Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String name;
    private long salary;

    @OneToOne(fetch=FetchType.LAZY)
    private ParkingSpace parkingSpace;

……..……
}
```


Mapeado objeto-relacional
Claves primarias compuestas

- Las claves primarias compuestas se definen como clases individuales, que se vinculan las clases a las que identifican
- Deben:
  - Incluir método `equals()`
  - Incluir método `hashCode()`
  - Ser públicas
  - Ser serializables
  - Tener un constructor sin argumentos

Mapeado objeto-relacional
Claves primarias compuestas

- La clase puede vincularse a su clave compuesta:
  - Referenciando a la clave con la anotación `@IdClass`
  - Incluyéndola como un objeto incrustado a través de la anotación `@EmbeddedId`

Mapeado objeto-relacional
Claves primarias compuestas

```
public class EmployeeId {
  private String contry;
  private int id;

  public EmployeeId() {}
  public EmployeeId(String country, int id)
  {
    this.country= country;
    this.id= id;
  }
  public getCountry() { return country; }
  public getId() { return id; }
```

Mapeado objeto-relacional
Claves primarias compuestas

```
public boolean equals(Object o)
{ return (
             o instanceof EmployeeId) &&
             country.equals((EmployeeId)o).getCountry()) &&
             Id == ((EmployeeId)o).getId());
}

public int hashCode()
{
       return country.hashCode() + id;
}
}
```


Mapeado objeto-relacional
Claves primarias compuestas

```
@Entity
@IdClass(EmployeeId.class)
public class Employee {
    @Id private String country;
    @Id private int id;
    private String name;
    private long salary;

    public Employee() {}
    public Employee(String country, int id)
       {
         this.country= country;
         this.id= id;
       }
……….
}
```


Mapeado objeto-relacional
Claves primarias compuestas

```
@Entity
public class Employee {
    @EmbeddedId private EmployeeId id;
    private String name;
    private long salary;

    public Employee() {}
    public Employee(String country, int id)
       {
         this.id= new EmployeeId(country, id)
       }
    public String getCountry(){ return id.getCountry();}
    public int getId() { return id.getId(); }

……..
}
```


Mapeado objeto-relacional
Claves primarias compuestas

```
……..
EmployeeId id= new EmployeeId("España", 73);
Employee emp= em.find(Employee.class, id);
```

Mapeado objeto-relacional
Mapeado de clases asociación

- Las clases de asociación son el equivalente orientado a objetos de los atributos de las relaciones M a N
- Su representación en JPA se hace:
  - Definiendo una clase que represente a la clase asociación:
    - Con los atributos de la relación
    - Una clave compuesta por las claves de las dos entidades de la relación M a N
    - Dos relaciones 1 a N de las clases a la clase relación, la cual es la poseedora de las columnas de join de ambas relaciones

Mapeado objeto-relacional
Mapeado de clases asociación

```
@Entity
public class Employee {
       @Id private int id;
……
       OneToMany(mappedBy="employee")
       private Collection<ProjectAssingment> assingments;
…… }

@Entity
public class Project {
       @Id private int id;
……
       @OneToMany(mappedBy="project")
       private Collection<ProjectAssingment> projects;
…… }
```


Mapeado objeto-relacional
Mapeado de clases asociación

```
public class ProjectAssingmentId implements Serializable {
       private int employee;
       private int project;
……
}
```


Mapeado objeto-relacional
Mapeado de clases asociación

```
@Entity
@IdClass(ProjectAssingmentId.class)
public class ProjectAssignment {

       @Id
       @ManyToOne
       private Employee employee;

       @Id
       @ManyToOne
       private Project project;

       @Temporal(TemporalType.DATE)
       private Date startDate;
……
}
```

Mapeado objeto-relacional
Herencia

- Clases mapeadas
  - Son clases que no pueden participar en relaciones ni ser entidades
  - Sirven para representar atributos comunes a subclases
  - Son la versión JPA de las clases abstractas, aunque no es obligatorio que sean clases abstractas
  - Se definen a través de la anotación `@MappedSuperclass`

Mapeado objeto-relacional
Herencia

- Clases transitorias
  - Son clases no anotadas, por lo que no son persistidas
  - Si una entidad hereda de ellas, tiene su estado, pero no lo persiste

Mapeado objeto-relacional
Herencia

- Modelos de herencia
  - En JPA la herencia se puede representar de distintas formas
  - Estrategia de tabla única
    - Considera una única tabla con todas las posibles columnas de las subclases
    - El identificador debe ser del mismo tipo para toda las clases de la jerarquía
    - Se consigue etiquetando la clase base de la jerarquía con la anotación `@Inheritance(strategy=InheritanceType.SINGLE_TABLE)`

Mapeado objeto-relacional
Herencia

  - Estrategia de join
    - Aquí cada entidad tiene su tabla
    - Para juntar todos los atributos de una clase, hay que hacer join por todas las superclases
    - El identificador debe ser del mismo tipo para toda las clases de la jerarquía
    - Se consigue etiquetando la clase base de la jerarquía con la anotación `@Inheritance(strategy=InheritanceType.JOINED)`

Mapeado objeto-relacional
Herencia

- Estrategia de tabla por clase
  - Replica todos los atributos heredados por cada clase en cada tabla
  - Se consigue etiquetando la clase base de la jerarquía con la anotación `@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)`
- A día de hoy, la especificación no soporta la mezcla de estrategias de herencia

Mapeado objeto-relacional
Mapeado de colecciones

- JPA soporta el uso de otros contenedores de datos, además de `Collection`:
  - `Set`
  - `List`
  - `Map`

Mapeado objeto-relacional
Mapeado de colecciones

```
@Entity
public class Department {
        ……
        @OneToMany(mappedBy="department")
        @OrderBy("name ASC")
        private List<Employee> employees;
        ……….
}

//ordenados por claves de empleados
@Entity
public class Department {
        ……
        @OneToMany(mappedBy="department")
        @OrderBy
        private List<Employee> employees;
        ……….
}
```


Entity manager
Contextos de persistencia

- Ya sabemos que una unidad de persistencia es una configuración con nombre de clases entidad
- Un *contexto de persistencia* es un conjunto gestionado de instancias de entidad
- Cada contexto de persistencia está asociado con una unidad de persistencia, restringiendo las instancias a las clases de la unidad

Entity manager
Contextos de persistencia

- Los entity managers gestionan contextos de persistencia, y por ende, las instancias allí contenidas
- Su comportamiento puede variar si están definidos en el contexto de un servidor de aplicaciones o no
- Nosotros los usaremos fuera de contenedores de aplicaciones

Entity manager
Contextos de persistencia

- Los contextos de persistencia gestionados por aplicaciones se crean con factorías
- Son los encargados de gestionar tanto entidades, como las transacciones en las que se ven involucradas

Entity manager
Contextos de persistencia

```
public class Test {
public static void main(String[] args) {
  EntityManagerFactory emf =
Persistence.createEntityManagerFactory("EmployeeService");
  EntityManager em = emf.createEntityManager();
  em.getTransaction().begin();
    Employee e= em.find(Employee.class, 25);
    e.setSalary(2000);
  em.getTransaction().commit();
  em.close();
  emf.close();
 }
}
```


Entity manager
Contextos de persistencia

- Cuando una transacción es revocada, todos los cambios hechos en la base de datos son abandonados
  - Sin embargo, en memoria no es posible simular este comportamiento
  - Al hacer rollback en negocio
    - La transacción contra la base de datos hace rollback
    - El contexto de persistencia es liberado, y las instancias de entidad desvinculadas de la base de datos

Entity manager
Contextos de persistencia

- Así, tenemos un conjunto de entidades desvinculadas, en un estado distinto del de la base de datos
- Si iniciamos una nueva transacción y fusionamos las entidades en un nuevo contexto de persistencia:
  - Alguna entidad podría tener una clave que ha sido asignada en el transcurso a otra entidad
  - Si hay versionado para el tratamiento optimista de la concurrencia, podría estar desactualizado

Entity manager
Contextos de persistencia

- Si es necesario salvar parte del estado de las entidades desvinculadas podría ser más razonable volver a leerlas y modificar su estado

- Veremos ahora las operaciones fundamentales que proporcionan los entity managers

Entity manager
Operaciones

- `persist()`
  - Acepta una nueva instancia de entidad y la gestiona, si no lo estaba
  - No conlleva ninguna acción hasta que una transacción del entity manager haga commit (con independencia si el `persist()` ha sido invocado dentro o fuera de la transacción)
  - Está pensado para instancias nuevas que no existen en la base de datos

Entity manager
Operaciones

- Es importante darse cuenta que cuando las entidades participan en relaciones, éstas solo se ven afectadas si la entidad modificada es la poseedora de la relación

Entity manager Operaciones

```
Department dept= em.find(Department.class, 30);
Employee emp= new Employee();
emp.setId(53);
emp.setName("Fausto");
/* si se comenta, el empleado no se vincula al
departamento, a pesar de incluir al empleado en la lista
del departamento
emp.setDepartment(dept);
*/
dept.getEmployees().add(emp);
em.persist(emp);
```


Entity manager Operaciones

- `find()`
  - Localiza instancias de entidades en base a su clave primaria
  - Devuelve:
    - Una instancia gestionada, si se invoca en el contexto de una transacción
    - Una instancia desvinculada, si se invoca fuera de una transacción

Entity manager Operaciones

```
em.getTransaction().begin();
Project pp= em2.find(Project.class, 1);
Collection<Employee> employees= pp.getEmployees();
Iterator<Employee> it= employees.iterator();
while (it.hasNext())
{
System.out.println(it.next());
}
em.getTransaction().commit();
```


Entity manager Operaciones

- `remove()`
  - Elimina una instancia de entidad de la base de datos, una vez hecho commit
  - Evidentemente, respeta las restricciones de integridad
  - Se puede invocar fuera de una transacción, pero no surte efecto hasta que una transacción del entity manager se compromete

Entity manager
Operaciones

```
Employee emp= em.find(Employee.class, 25);
ParkingSpace ps= emp.getParkingSpace();
emp.setParkingSpace(null);
em.remove(ps);
```


Entity manager
Operaciones

- `clear()` y `detach()`
  - La primera limpia el entity manager, dejando a las entidades que gestionaba desvinculadas de la base de datos
  - La segunda desvincula selectivamente una entidad del entity manager
  - Cuando se trabaja con una instancia de entidad desvinculada, hay que tener cuidado con sus relaciones, ya que en función de la implementación, podrían ser accesibles los objetos referenciados o no

Entity manager
Operaciones

- `flush()`
  - Actualiza la base de datos con las entidades creadas, modificadas o eliminadas en el entity manager
  - Es como un commit, pero permitiendo hacer un rollback si fuera necesario

Entity manager
Operaciones

- `merge()`
  - Fusiona una instancia de entidad desvinculada con su entity manager
  - La desvinculación ha podido producirse por:
    - Cierre del contexto de persistencia
    - Invocación del método `clear()` o `detach()`
    - Invocación de un `rollback()`
    - Serialización de la instancia de entidad

Entity manager
Operaciones

- `refresh()`
  - Refresca una entidad gestionada, actualizando el estado en memoría con el actual en la base de datos
  - Sólo tiene sentido en entornos concurrentes

Entity manager
Operaciones

- A veces es deseable que las operaciones realizadas sobre una entidad afecten a las demás *en cascada*
  - Para ello, hay que utilizar el atributo `cascade` en las definiciones de las relaciones

Entity manager
Operaciones

```
@Entity
public class Employee {
……
       @OneToOne(cascade={CascadeType.PERSIST,
CascadeType.REMOVE, CascadeType.DETACH, CascadeType.REFRESH})
       ParkingSpace parkingSpace;
       @OneToMany(cascade={CascadeType.ALL})
       Collection<Phone> phones;
…… }
```


Concurrencia
Introducción

- Una entidad gestionada pertenece a un único contexto de persistencia
- Por lo tanto, no debería ser gestionada por más de un contexto de persistencia de manera simultánea
- Esto es responsabilidad de la aplicación
- Fusionar la misma entidad en dos contextos de persistencia diferentes abiertos podría producir resultados indefinidos

Concurrencia
Introducción

- Los entity managers y los contextos de persistencia que gestionan no están diseñados para ser accedidos por más de una hebra de ejecución simultáneamente, es decir sus métodos no son `synchronized`
- Por lo tanto es responsabilidad de la aplicación mantenerlos dentro de la hebra que lo obtuvo

Concurrencia
Introducción

- Por defecto JPA funciona con un nivel de aislamiento read commited
- Así, los cambios comprometidos por una transacción pueden ser vistos durante la ejecución de una transacción paralela

Concurrencia
Bloqueos

- JPA implementa fácilmente el bloqueo optimista:
  - Basta con incluir un campo de versión con la anotación `@Version` y bloquear:
    - `em.lock(object, LockModeType.OPTIMISTIC)`
      `em.lock(object, LockModeType.OPTIMISTIC_FORCE_INCREMENT)`
  - Si otra hebra modifica la entidad, al hacer commit salta una `OptimisticLockException`

Concurrencia
Bloqueos

  - Ojo, al hacer pruebas, que la implementación puede estar optimizada y si los cambios en el objeto no modifican su estado en la BD, no los hace y no aumenta el número de versión
  - Hay dos tipos de bloqueos optimistas:
    - Optimista: sólo aumenta el número de versión si hay modificaciones en el objeto
    - De incremento forzado: aumento el número de versión al hacer la lectura, es decir, sin necesidad de modificar el objeto. Tiene sentido cuando se modifica el lado N de una relación, y se quiere “bloquear” el lado 1 (leyéndolo)

Concurrencia
Bloqueos

```
@Entity
Public class Habitación {
       @Id private int id;
       @Version private int version;
…….
}
```


Concurrencia
Bloqueos

```
try {
       em.getTransaction().begin();
       Habitacion h= em.find(Habitacion.class, id);
       em.lock(h, LockModeType.OPTIMISTIC);
       //de manera atómica:
       //Employee emp= em.find(Employee.class, id,
       //LockModeType.OPTIMISTIC);
       If (h.estaVacia()) h.reservar(datosReserva);
       em.getTransaction().commit();
     } catch (OptimisticLockException e) { …… }
```


Concurrencia
Bloqueos

```
public class EmployeeASImp {
public void addUniform(int id, Uniform uniform)
{ …..
       em.getTransaction().begin();
       Employee emp= em.find(Employee.class, id);
       em.lock(emp, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
       emp.addUniform(uniform);
       uniform.setEmployee(emp);
       em.getTransaction().commit();
 …….. }
```


Concurrencia
Bloqueos

```
Public void calculateCleaningCost(int id, uCost)
{…..
       em.getTransaction().begin();
       Employee emp= em.find(Employee.class, id);
       Float cost = emp.getUniforms().size()*uCost;
       emp.setCost(emp.getCost()+cost);
       em.getTransaction().commit();
….. }
}
```

Concurrencia
Bloqueos

- El bloqueo pesimista se implementa también de manera similar:
  - `em.lock(object, LockModeType.PESSIMISTIC_WRITE)`
  - `em.lock(object, LockModeType.PESSIMISTIC_READ)`
  - `em.lock(object, LockModeType.PESSIMISTIC_FORCE_INCREMENT)`
  - Hay tres tipos de bloqueos:
    - De escritura: bloquea incluso las lecturas
    - De lectura: sólo bloquea las escrituras (la implementación puede implementarlo como de lectura)
    - Incremento forzado: bloqueo write que incrementa el número de versión

Concurrencia
Bloqueos

```
@Entity
Public class Habitación {
       @Id private int id;
       @Version private int version;
…….
}
……
       em.getTransaction().begin();
       Habitacion h= em.find(Habitacion.class, id,
              LockModeType.PESSIMISTIC_WRITE);
       If (h.estaVacia()) h.reservar(datosReserva);
       em.getTransaction().commit();
……..
```


JP QL

- Java Persistence Query Language (JP QL) es un lenguaje que permite representar preguntas complejas para extraer instancias de entidad
- Las ventajas frente a SQL son:
  - Acceso en negocio a datos persistentes
  - Portabilidad
- El principal inconveniente es tener que aprender otro lenguaje de query

JP QL

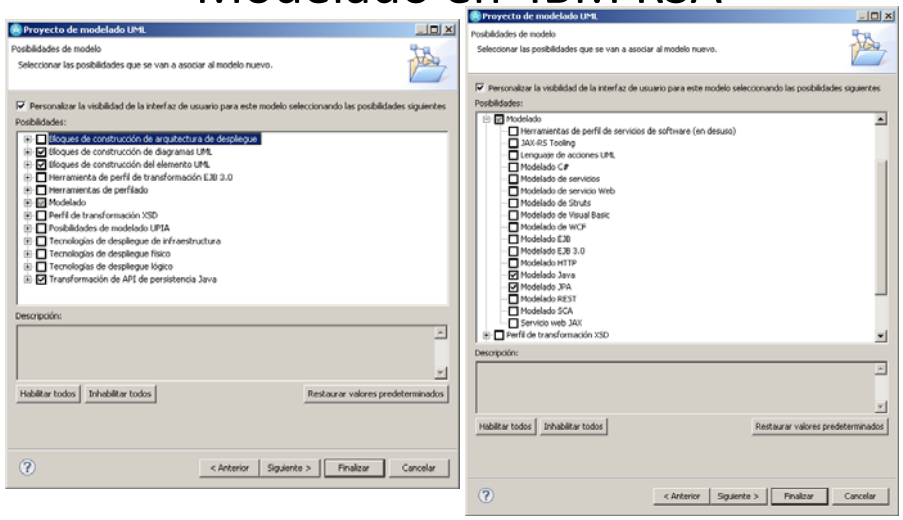
- Es la implementación JP QL de las querys del almacén del dominio
- Ejemplo:

```
……
String query="
SELECT e
FROM Employee e JOIN e.deparment d
WHERE e.salary < 1000
ORDER BY deptName, bonus DESC"
……
List result= em.createQuery(query).getREsultList();
……
```

Modelado en IBM RSA

- IBM RSA está basado en Eclipse
- Por tanto puede soportar la definición de entidades JPA utilizando una notación visual
- No obstante, para modelar aplicaciones JPA es más útil usar la “Transformación de API de persistencia Java”, dentro de las posibilidades de modelado UML, y utilizar después transformaciones UML a JPA

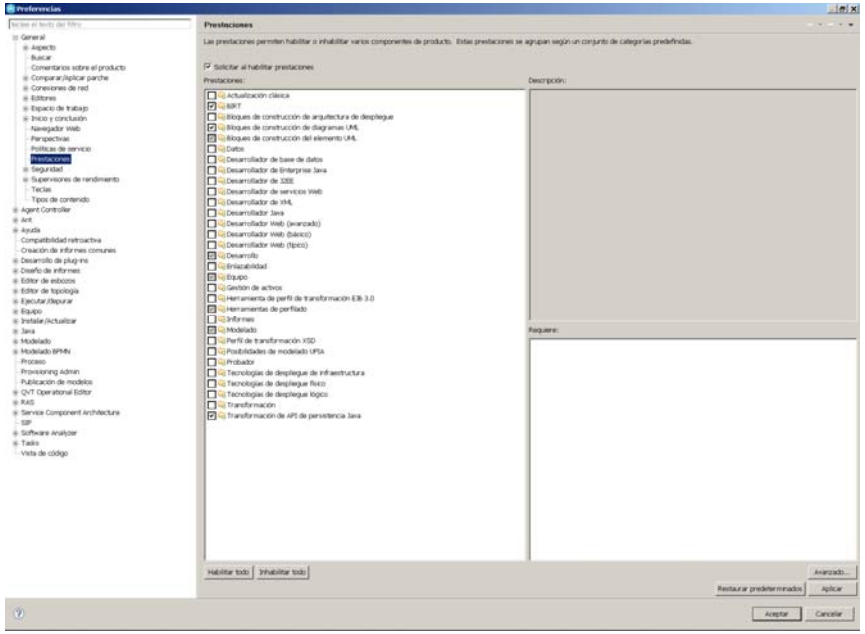
Modelado en IBM RSA



Activación del modelado JPA en IBM RSA

Modelado en IBM RSA

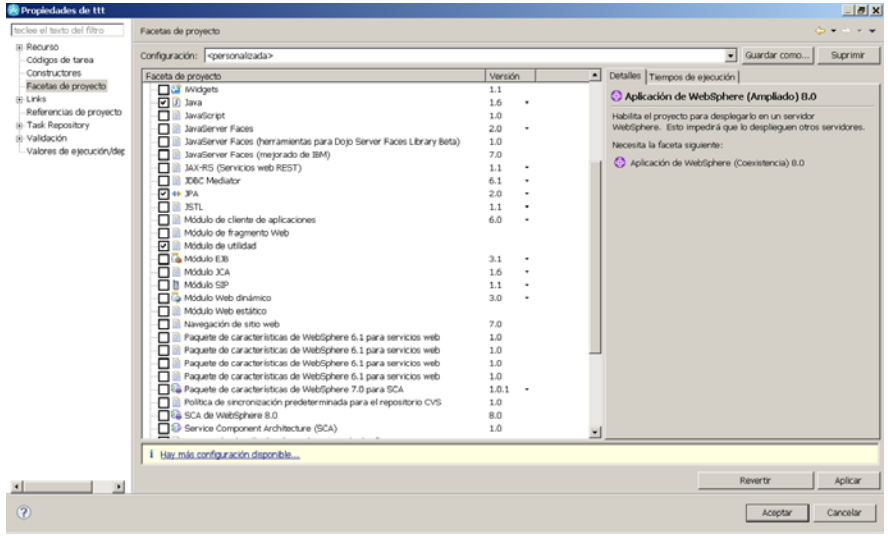
- Deben estar activadas estas opciones, en el apartado General/Prestaciones (capabilities)



Activación del modelado JPA en IBM RSA

Modelado en IBM RSA

- Si además, queremos utilizar las clases JPA en diagramas de secuencia, tenemos que activar la faceta JPA (y la faceta Modulo de utilidad), además de la faceta Java
- Para ello deberemos tener instalada alguna implementación de JPA, por ejemplo ElipseLink (debe ser la versión 2.1.x, que es la compatible con IBM RSA 8.0.3 y 8.5)



Activación del faceta JPA (y Módulo de utilidad)

Conclusiones

- JPA: marco implementación del almacén del dominio
- Gestiona:
  - Persistencia de objetos
  - Carga dinámica de relaciones
  - Transacciones
  - Concurrencia