Trabajando con JPA

Operaciones de lectura

Índice del capítulo

- Tipos de consultas.
- Trabajando con JPQL.
- La API Criteria.
- Trabajando en modo lazy.

Tipos de consultas

- Introducción.
- La interface Query.
- La interface TypedQuery.
- La interface NamedQuery.
- La interface NativeQuery.

Introducción

- Hay tres tipos básicos de consultas JPA:
  - **Query**, escrita en la sintaxis del lenguaje de consulta de persistencia de Java (**JPQL**).
  - **NativeQuery**, escrito en sintaxis **SQL simple**.
  - **Criteria API Query**, construida programáticamente a través de diferentes **métodos**.

La interface Query

- Introducción.
- TypedQuery.
- NamedQuery.

La interface Query

- Query es **similar en sintaxis a SQL**, y generalmente se usa para realizar operaciones CRUD:

```
public UserEntity getUserByIdWithPlainQuery(Long id) {
   Query jpqlQuery = getEntityManager().createQuery("SELECT u FROM UserEntity u WHERE u.id=:id");
   jpqlQuery.setParameter("id", id);
   return (UserEntity) jpqlQuery.getSingleResult();
}
```

- La consulta que debemos construir trabaja con la sintaxis JPQL.

La interface Query

- Esta consulta recupera el registro coincidente de la tabla de usuarios y también lo asigna al objeto **UserEntity**.
- Hay dos **subtipos de consulta** adicionales:
  - TypedQuery.
  - NamedQuery.

La interface TypedQuery

- Necesitamos prestar atención a la **operación return** en nuestro ejemplo anterior.
- JPA no puede saber cuál será el **tipo de resultado de la consulta** y, como resultado, tenemos que hacer un **casting** o conversión forzosa de tipo.
- Pero, JPA proporciona un **subtipo de consulta especial** conocido como **TypedQuery**.
- Siempre se prefiere esto si conocemos nuestro tipo de resultado de consulta de antemano.
- Además, hace que nuestro código sea mucho más robusto y fácil de probar.

La interface TypedQuery

- Se modifica el ejemplo anterior:

```
public UserEntity getUserByIdWithTypedQuery(Long id){
   TypedQuery<UserEntity> typedQuery = getEntityManager().createQuery("SELECT u FROM UserEntity u WHERE u.id=:id",
     UserEntity.class);
   typedQuery.setParameter("id", id);
   return typedQuery.getSingleResult();
}
```

- De esta manera, obtenemos código **más fuerte** de forma gratuita, **evitando posibles excepciones** de conversión en el futuro.

La interface NamedQuery

- Si bien podemos definir **dinámicamente** una consulta en métodos específicos, pueden convertirse en una base de código difícil de mantener.
- ¿Qué pasaría si pudiéramos mantener las consultas de uso general **en un lugar centralizado** y fácil de leer?
- JPA también nos ofrece esta característica gracias a **otro subtipo de consulta** conocido como **NamedQuery**. Podemos definir NamedQueries en **orm.xml** o en un archivo de propiedades.
- Además, podemos **definir NamedQuery en la propia clase de Entidad**, proporcionando una forma centralizada, rápida y fácil de leer y encontrar las consultas relacionadas con una Entidad.
- Todas las NamedQueries deben tener un **nombre único**.

La interface NamedQuery

- Veamos cómo podemos agregar un **NamedQuery** a nuestra clase UserEntity:

```
@Table(name = "users")
@Entity
@NamedQuery(name = "UserEntity.findByUserId", query = "SELECT u FROM UserEntity u WHERE u.id=:userId")
public class UserEntity {
   @Id
   private Long id;
   private String name;
   //Standard constructor, getters and setters.
}
```

- La **anotación @NamedQuery** debe agruparse dentro de una anotación **@NamedQueries** si usamos Java antes de la versión 8. A partir de Java 8 en adelante, simplemente podemos **repetir la anotación @NamedQuery** en nuestra clase Entity.

La interface NamedQuery

- Ejemplo:

```
public UserEntity getUserByIdWithNamedQuery(Long id) {
    Query namedQuery = getEntityManager().createNamedQuery("UserEntity.findByUserId");
    namedQuery.setParameter("userId", id);
    return (UserEntity) namedQuery.getSingleResult();
}
```

La interface NativeQuery

- Una NativeQuery es simplemente **una consulta SQL**. Esto nos permite liberar todo el poder de nuestra base de datos, ya que podemos usar funciones específicas que **no están disponibles en la sintaxis restringida de JPQL**.

- Esto tiene un coste. Perdemos la **portabilidad de la base de datos** de nuestra aplicación con NativeQuery porque nuestro proveedor de JPA ya no puede abstraer detalles específicos de la implementación o el proveedor de la base de datos.

- Ejemplo:

```
public UserEntity getUserByIdWithNativeQuery(Long id) {
    Query nativeQuery = getEntityManager().createNativeQuery(
      "SELECT * FROM users WHERE id=:userId", UserEntity.class);
    nativeQuery.setParameter("userId", id);
    return (UserEntity) nativeQuery.getSingleResult();
}
```

La interface NativeQuery

- Siempre debemos considerar si una NativeQuery **es la única opción**.
- La mayoría de las veces, una buena consulta JPQL puede satisfacer nuestras necesidades y, lo que es más importante, **mantener un nivel de abstracción** de la implementación real de la base de datos.
- Usar NativeQuery **no significa necesariamente** encerrarse en un proveedor de base de datos específico. Después de todo, si nuestras consultas no usan comandos SQL específicos y solo usan una sintaxis SQL estándar, cambiar de proveedor no debería ser un problema.

Trabajando con JPQL

- Introducción.
- Consultas básicas.
- Filtrando los resultados.
- Parámetros de las consultas.
- Proyectando los resultados.
- Joins entre entidades.
- Otros conceptos.
- Paginación de resultados.
- Subconsultas.
- JPQL vs HQL.

Introducción

- JPQL es el **lenguaje** en el que se construyen las **queries en JPA**. Aunque en apariencia es muy similar a SQL, en la realidad operan en **mundos totalmente distintos**.
- En JPQL las preguntas se construyen sobre **clases y entidades**, mientras que SQL opera sobre **tablas, columnas y filas** en la base de datos.
- Una consulta JPQL puede contener los siguientes elementos:

```
SELECT c
FROM Category c
WHERE c.categoryName LIKE :categoryName
ORDER BY c.categoryId
```

Introducción

- Una cláusula **SELECT** que especifica **el tipo de entidades** o **valores** que se recuperan.
- Una cláusula **FROM** que especifica una declaración de **la entidad que es usada** por otras cláusulas.
- Una cláusula opcional **WHERE** para **filtrar** los resultados devueltos por la query.
- Una cláusula opcional **ORDER BY** para **ordenar** los resultados devueltos por la query.
- Una cláusula opcional **GROUP BY** para realizar agregación.
- Una cláusula opcional **HAVING** para realizar un filtrado en conjunción con la agregación.

Consultas básicas

- La consulta más sencilla en JPQL es la que selecciona todas las instancias de un único tipo de entidad:

```
SELECT e FROM Empleado e
```

- La sintaxis de JPQL es **similar** a la de SQL. De esta forma los desarrolladores con experiencia en SQL pueden comenzar a utilizarlo rápidamente.

- La **diferencia principal** es que en SQL se seleccionan **filas de una tabla** mientras que en JPQL se seleccionan **instancias de un tipo de entidad** del modelo del dominio.

- La cláusula SELECT es ligeramente distinta a la de SQL listando **solo el alias e del Empleado**. Este tipo indica el tipo de datos que va a devolver la consulta, **una lista de cero o más instancias de tipo Empleado**.

Consultas básicas

- A partir del **alias**, podemos obtener **sus atributos** y recorrer **las relaciones** en las que participa utilizando el operador punto (.).
- Por ejemplo, si sólo queremos los **nombres de los empleados** podemos definir la siguiente consulta:

```
SELECT e.nombre FROM Empleado e
```

- Ya que la entidad Empleado tiene un campo persistente llamado **nombre** de tipo String, esta consulta devolverá **una lista de cero o más objetos de tipo String**.

Consultas básicas

- Podemos también seleccionar una **entidad cuyo tipo no listamos en la cláusula SELECT**, utilizando los atributos relación.
- Por ejemplo:

```
SELECT e.departamento FROM Empleado e
```

- Debido a la **relación muchos-a-uno** entre Empleado y Departamento la consulta devolverá **una lista de instancias de tipo Departamento**.

Filtrando los resultados

- Igual que SQL, JPQL contiene una **cláusula WHERE** para especificar condiciones que deben cumplir los datos que se devuelven.
- La mayoría de los operadores disponibles en SQL están en JPQL, incluyendo las operaciones básicas de comparación: **IN, LIKE y BETWEEN**, funciones como **SUBSTRING o LENGTH** y subqueries. Igual que antes, la diferencia fundamental es que se utilizan expresiones sobre entidades y no sobre columnas.
- Ejemplo:

```
SELECT e
  FROM Empleado e
  WHERE e.departamento.name = 'NA42' AND
      e.direccion.provincia IN ('ALC','VAL')
```

Parámetros de las consultas

- JPQL soporta dos tipos de sintaxis para la ligadura (binding) de parámetros: **posicional y por nombre**. Vemos un ejemplo del primer caso:

```
SELECT e
  FROM Empleado e
  WHERE e.departamento = ?1 AND e.salario > ?2
```

- Veremos en el siguiente apartado como ligar parámetros determinados a esas posiciones. La segunda forma de definir parámetros es por **nombre**:

```
SELECT  e
  FROM Empleado e
  WHERE e.departamento = :dept AND e.salario > :sal
```

Proyectando los resultados

- Es posible recuperar sólo alguno de los atributos de las instancias. Esto es útil cuando tenemos una gran cantidad de atributos (columnas) y sólo vamos a necesitar listar algunos. Por ejemplo, la siguiente consulta selecciona sólo el nombre y el salario de los empleados:

```
SELECT e.nombre, e.salario FROM Empleado e
```

- El resultado será **una colección de cero o más instancias de arrays** de tipo Object. Cada array contiene dos elementos, el primero un String y el segundo un Double:

```
List result = em.createQuery("SELECT e.nombre, e.salario FROM Empleado e WHERE e.salario > 30000 " +
  "ORDER BY e.nombre").getResultList();
Iterator empleados = result.iterator();
while (empleados.hasNext()) {
  Object[] tupla = (Object[]) empleados.next();
  String nombre = (String) tupla[0];
  int salario = ((Integer) tupla[1]).intValue();
```

Joins entre entidades

- Al igual que en SQL, es posible **definir consultas** que realicen una selección en el resultado de unir (join) entidades entre las que se ha establecido una relación.

- Por ejemplo, el siguiente código muestra un join entre las entidades Empleado y CuentaCorreo para recuperar todos los correos electrónicos de un departamento específico:

```
SELECT c.correo
  FROM Empleado e, CuentaCorreo c
  WHERE e = c.empleado AND
      e.departamento.nombre = 'NA42'
```

Joins entre entidades

- En JPQL también **es posible especificar joins** en la cláusula FROM **utilizando el operador JOIN**.
- Una ventaja de este operador es que el join puede especificarse **en términos de la propia asociación** y que el motor de consultas proporcionará automáticamente el criterio de join necesario cuando genere el SQL.
- El siguiente código muestra la misma consulta reescrita para utilizar este operador:

```
SELECT c.correo FROM Empleado e JOIN e.cuentasCorreo c WHERE e.departamento.nombre = 'NA42'
```

- JPQL soporta **múltiples tipos de joins**, incluyendo **inner y outer** joins, **left joins** y una técnica denominada **fetch joins** para cargar datos asociados a las entidades que no se devuelven directamente.
- Vamos a ver algunos ejemplos más, para tener una idea de la potencia de la sintaxis.

Joins entre entidades

- Selecciona todos los departamentos distintos asociados a empleados:

```
SELECT DISTINCT e.departamento FROM Empleado e
```

- Otra forma de hacer la misma consulta utilizando el operador JOIN:

```
SELECT DISTINCT d FROM Empleado e JOIN e.departamento d
```

- Selecciona los departamentos distintos que trabajan en **Alicante** y que participan en el proyecto 'BlueBook':

```
SELECT DISTINCT e.departamento
  FROM Proyecto p JOIN p.empleados e
  WHERE p.nombre = 'BlueBook' AND
      e.direccion.localidad = 'ALC'
```

Joins entre entidades

- Selecciona los proyectos distintos que pertenecen a empleados de un departamento:

```
SELECT DISTINCT p FROM Departamento d JOIN d.empleados JOIN e.proyectos p
```

- Selecciona **todos los empleados y recupera** (para evitar el lazy loading) la entidad Direccion con la que está relacionado cada uno:

```
SELECT e
  FROM Empleado e JOIN FETCH e.direccion
```

Otros conceptos

- Paginación de resultados.
- Subconsultas.
- Valores nulos y colecciones vacías.
- Funciones.
- Patrones de coincidencia.
- Agrupaciones.
- Ordenación.

Paginación de resultados

- Es posible realizar un **paginado de los resultados**, definiendo un **número máximo** de instancias a devolver en la consulta y un número de instancia a partir del que se construye la lista:

```
Query q = em.createQuery("SELECT e FROM Empleado e");
q.setFirstResult(20);
q.setMaxResults(10);
List empleados = q.getResultList();
```

Subconsultas

- Es posible **anidar** múltiples queries.
- Podemos utilizar el **operador IN** para obtener las instancias que se encuentran en el resultado de la subquery.
- Por ejemplo, la siguiente expresión devuelve los empleados que participan en proyectos de tipo 'A'.

```
SELECT e FROM Empleado e
WHERE e.proyecto IN (SELECT p
  FROM Proyecto p
  WHERE p.tipo = 'A')
```

Valores nulos y colecciones vacías

- Es posible utilizar los operadores IS NULL o IS NOT NULL para comprobar si un atributo es o no nulo:

```
WHERE e.departamento IS NOT NULL
```

- En el caso de colecciones hay que utilizar los operadores IS EMPTY o IS NOT EMPTY:

```
WHERE e.proyectos IS EMPTY
```

Funciones

- Es posible utilizar llamadas a **funciones predefinidas** en JPQL. Veamos unos cuantos ejemplos:
  - **CONCAT** concatena dos cadenas: CONCAT(string1, string2):
    - WHERE CONCAT(e.nombre, e.apellido) LIKE ‘Ju%cia’
  - **SUBSTRING** extrae una subcadena SUBSTRING(string, position, length).
  - **LOWER** devuelve los caracteres de una cadena convertidos en minúsculas LOWER(string).
  - **UPPER** devuelve los caracteres de una cadena convertidos en mayúsculas UPPER(string).
  - **LENGTH** devuelve la longitud de una cadena LENGTH(string).
  - **SIZE** devuelve el tamaño de una colección WHERE SIZE (e.proyectos) > 4.
  - **CURRENT_TIME**, CURRENT_DATE, CURRENT_TIMESTAMP devuelven el tiempo del momento actual.

Patrones de coincidencia

- Veamos el conjunto de **funciones y operadores** que permiten introducir **pattern matching** sobre campos de texto en las búsquedas.
- **LIKE**: permite buscar un **patrón en un texto** y comprueba si una **cadena especificada** cumple un patrón específico. Si se precede con el operador NOT devuelve aquellos valores que no cumplen el patrón. El patrón puede incluir cualquier carácter y los siguientes caracteres libres:
  - El **carácter tanto por ciento** (%) empareja con 0 o más caracteres cualesquiera.
  - El **carácter subrayado** (_) empareja un único carácter cualquiera.
- El operando **izquierdo** es siempre la **cadena que se comprueba** y el **derecho el patrón**. Ejemplos:
  - c.name LIKE '_r%' es TRUE para 'Brasil' and FALSE for ‘Dinamarca'.
  - c.name LIKE '%' es siempre TRUE.
  - c.name NOT LIKE '%' es siempre FALSE.

Patrones de coincidencia

- Para emparejar un **carácter subrayado** o **tanto por ciento** tiene que ir precedido por un carácter de escape. Por ejemplo:
  - '100%' LIKE '%\%' ESCAPE '\' se evalua a TRUE.
  - '100' LIKE '%\%' ESCAPE '\' se evalua a FALSE.
- En las expresiones de arriba sólo el primer carácter tanto por ciento (%) representa un carácter libre. El segundo (que aparece tras el carácter de escape) representa un carácter % real %.

Patrones de coincidencia

- **LOCATE**: El **operador LOCATE**(str, substr [, start]) busca una subcadena y devuelve su posición (comenzando a contar por 1).
- El **tercer argumento**, si está presente, especifica la posición a partir de la que comienza la búsqueda.
- Se devuelve 0 si no se encuentra la subcadena. Por ejemplo:
  - LOCATE('India', 'a') devuelve 5
  - LOCATE('Alicante', 'a', 3) devuelve 5
  - LOCATE('Mejico', 'a') devuelve 0

Patrones de coincidencia

- **TRIM**: El operador TRIM([[LEADING|TRAILING|BOTH] [char] FROM] str) devuelve una cadena después de haber eliminado los caracteres del comienzo (LEADING), del final (TRAILING) o ambos (BOTH).
- Si no se especifica el carácter, se considera el espacio en blanco. Ejemplos:
  - TRIM(' UK ') devuelve 'UK'
  - TRIM(LEADING FROM ' UK ') devuelve ‘UK'
  - TRIM(TRAILING FROM ' UK ') devuelve ' UK'
  - TRIM(BOTH FROM ' UK ') devuelve 'UK'
  - TRIM('A' FROM 'ARGENTINA') devuelve 'RGENTIN'

Agrupaciones

- JPQL proporciona las siguientes **funciones de agrupación**:
  - AVG,COUNT,MAX,MIN,SUM.
- Pueden ser aplicadas a un número de valores. El tipo devuelto por las funciones es Long o Double, dependiendo de los valores implicados en la operación.
- Por ejemplo, para obtener el mayor sueldo de un conjunto de empleados:

```
SELECT MAX(e.sueldo) FROM Empleado e
```

- Para obtener el número de elementos que cumplen una condición:

```
SELECT COUNT(e) FROM Empleado e WHERE ...
```

Agrupaciones

- Podemos realizar consultas más complicadas utilizando **GROUP BY y HAVING**.
- Por ejemplo, podemos obtener los empleados y el número de proyectos en los que participan de la siguiente forma:

```
SELECT p.empleado, COUNT(p) FROM Proyecto p GROUP BY p.empleado
```

- El código para recorrer la lista resultante sería el siguiente:

```
List result = em.createQuery("SELECT p.empleado, COUNT(p) FROM Proyecto p GROUP BY p.empleado").getResultList();
Iterator res = result.iterator();
while (res.hasNext()) {
  Object[] tupla = (Object[]) res.next();
  Empleado emp = (Empleado) tupla[0];
  long count = ((Long) tupla[1]).longValue();
  ...
}
```

Agrupaciones

- La siguiente consulta combina todos los elementos. Primero se **filtrarían** los resultados con la cláusula **WHERE**, después los resultados son **agrupados** y por último se comprueba la cláusula **HAVING**.

- Devuelve los empleados que participan en más de 5 proyectos creados entre dos fechas dadas:

```
SELECT p.empleado, COUNT(p)
FROM Proyecto p
WHERE p.fechaCreacion is BETWEEN :date1 and :date2
GROUP BY p.empleado
HAVING COUNT(p) > 5
```

Ordenación

- Es posible ordenar la lista resultante de la consulta por alguno de los campos usando la directiva **ORDER BY**.
- Un ejemplo de su uso:

```
SELECT e FROM Proyecto p JOIN p.empleados e
 WHERE p.nombre = 'Pepe'
 ORDER BY e.name
```

JPQL vs HQL

- JPQL es una estandarización de HQL (Hibernate Query Language, Lenguaje de consultas de Hibernate).
- Las consultas escritas con ambos se ejecutan sin distinción alguna en Hibernate con el gestor de entidades de JPA y sus clases asociadas.
- Si bien, **todas las sentencias JPQL** son válidas en **HQL**, lo contrario **no siempre es cierto** porque HQL posee **características no recogidas en el estándar**.

JPQL vs HQL

- JPQL (Java Persistence Query Language) y HQL (Hibernate Query Language) son lenguajes de consulta utilizados en el contexto de mapeo objeto-relacional (ORM) para acceder a bases de datos mediante frameworks como Java Persistence API (JPA) y Hibernate, respectivamente.

- Aunque comparten **similitudes**, hay algunas **diferencias** en la sintaxis entre JPQL y HQL.

JPQL vs HQL

- Aquí hay algunas de las **diferencias** más destacadas:
  - Directiva **SELECT**:
    - En JPQL, la cláusula SELECT es obligatoria y se utiliza para especificar las propiedades o entidades que se deben recuperar.
    - En HQL, la cláusula SELECT es opcional. Si no se proporciona, se seleccionan todos los campos de la entidad.
  - **Identificación de Parámetros**:
    - JPQL utiliza el **símbolo ?** para identificar parámetros posicionales. Por ejemplo, SELECT e FROM Employee e WHERE e.salary > ?1.
    - HQL utiliza el **símbolo :** para identificar parámetros con nombre. Por ejemplo, FROM Employee WHERE salary > :minSalary.

JPQL vs HQL

- (cont.):
  - **Funciones y Expresiones**:
    - Ambos lenguajes admiten funciones y expresiones, pero puede haber algunas diferencias en la sintaxis exacta de las funciones específicas.
    - Por ejemplo, para concatenar cadenas en JPQL, se usa CONCAT (SELECT CONCAT(e.firstName, ' ', e.lastName) FROM Employee e) mientras que en HQL se utiliza el operador || (SELECT e.firstName || ' ' || e.lastName FROM Employee e).
  - **Tratamiento de NULL**:
    - En JPQL, se utiliza IS NULL o IS NOT NULL para verificar nulidad.
    - En HQL, se usa IS NULL o IS NOT NULL de manera similar a JPQL.

JPQL vs HQL

- (cont.):
  - Sintaxis **JOIN**:
    - La sintaxis para realizar **JOINs** puede variar ligeramente entre JPQL y HQL.
    - Ambos soportan **INNER JOIN**, **LEFT JOIN**, y **RIGHT JOIN**, pero la sintaxis específica puede diferir.
  - **Operadores de Conjunto**:
    - JPQL utiliza **MEMBER OF** y **NOT MEMBER OF** para operaciones de conjuntos.
    - HQL utiliza **IN** y **NOT IN** para operaciones de conjuntos.

JPQL vs HQL

- (cont.):
  - Tratamiento de Fechas:
    - La sintaxis para trabajar con fechas puede variar en detalles, como la forma de comparar fechas.
    - Por ejemplo, en JPQL, podrías tener algo como:

```
SELECT e FROM Entity e WHERE e.fecha > :fecha
```

    - y en HQL algo como.

```
FROM Entity WHERE fecha > :fecha
```

La API Criteria

- Introducción.
- Creando una consulta.
- Estructura.
- Conceptos fundamentales.
- Cláusula SELECT.
- Cláusula FROM.
- Cláusula WHERE.
- Cláusula ORDER BY.
- Cláusula GROUP BY.
- Otras sentencias.

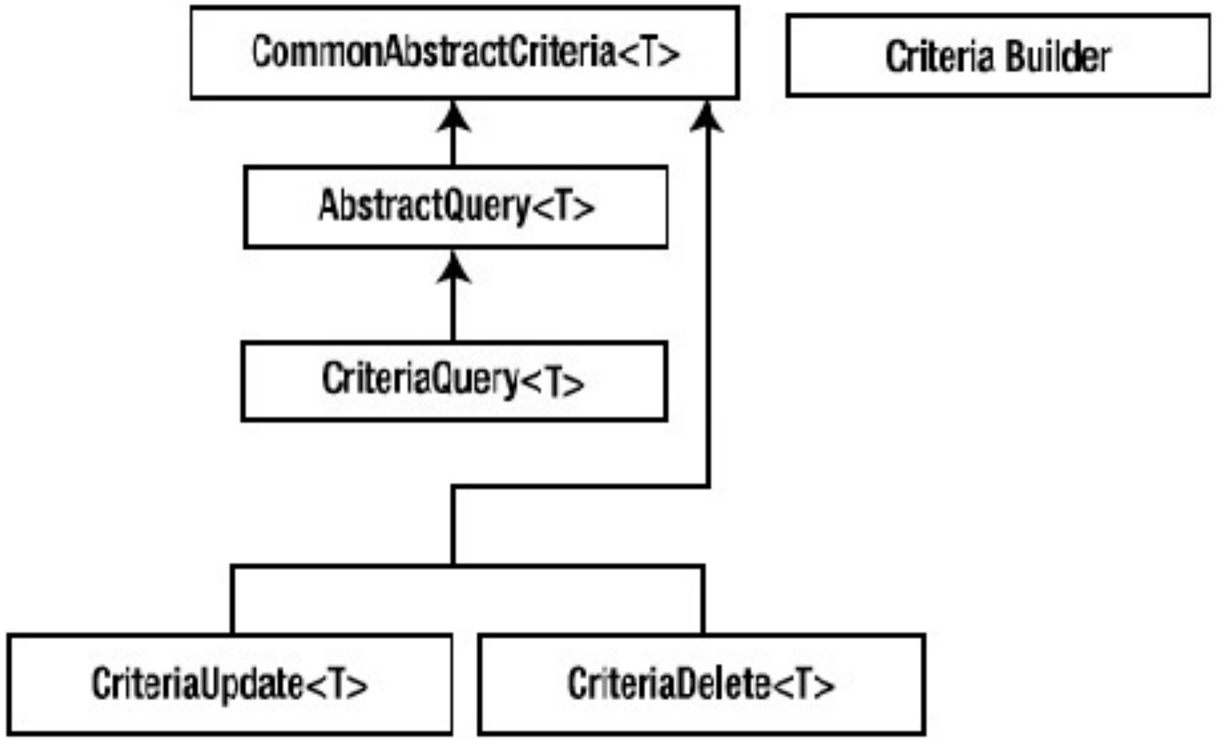
Introducción

- Antes de que **los lenguajes como JPQL** llegarán a estandarizarse, el **método más común** para la construcción de consultas era a través de **una API de programación**.
- Con la llegada de JPQL **las API de construcción de consultas** se siguen manteniendo debido a que dan acceso a características adicionales que no proporciona todavía JPQL.
- Criteria nos permite **construir consultas que estandarizan** muchas de las características que existen en aplicaciones con persistencia de datos.
- Criteria **no es simplemente una traducción de JPQL** al lenguaje de programación Java.
- Criteria adopta las **mejores prácticas** como por ejemplo encadenamiento de métodos y hace un uso completo de las características del lenguaje de programación Java.

Creando una consulta

- El corazón del Api Criteria es el **interfaz CriteriaBuilder** que se puede obtener desde el interfaz **EntityManager** a través del método getCriteriaBuilder().

- La jerarquía de interfaces se muestra a continuación:



Creando una consulta

- **CriteriaBuilder** es una factoría que nos permite crear instancias de la **interfaz CriteriaQuery** definiciones de consulta).
- Las definiciones de consultas son la **base** para la construcción de nuevas consultas.
- Una instancia de **CriteriaQuery** es básicamente un caparazón vacío en el que únicamente definimos el tipo de retorno de la consulta, posteriormente veremos como ir definiendo las distintas cláusulas que ya conocemos debe tener una consulta.

Creando una consulta

- Existen 3 métodos para la creación de instancias CriteriaQuery:
  - **createQuery**(Class<T>): Método más común con un parámetro que especifica la clase correspondiente al resultado de la consulta.
  - **createQuery**(): Método sin parámetros que corresponde con una consulta con un resultado de tipo Object.
  - **createTupleQuery**(): Método que corresponde con una consulta que devuelve múltiples elementos (la cláusula SELECT tiene varias expresiones) . Es equivalente a createQuery(Tuple). El interfaz Tuple se puede utilizar siempre que se deseen combinar varios elementos en un único tipo de objeto.

Creando una consulta

- El siguiente paso será usar el método del interfaz EntityManager para construir la consulta a partir de este.
  - createQuery(CriteriaQuery<T> criteriaQuery):
    - Método que devuelve TypedQuery a partir de CriteriaQuery.
  - createQuery(java.lang.String qlString) :
    - Método que devuelve Query a partir de String que representa una consulta JPQL.
- **Consejo**:
  - Para entender las diferencias entre definición de consultas con Api Criteria vs JPQL podemos pensar que cada instancia de CriteriaQuery es similar a la representación interna de una consulta JPQL. Cada proveedor de persistencia usa la representación interna después de que el String JPQL haya sido parseado.

Estructura

- La estructura de las consultas está formada por las mismas cláusulas que en JPQL.
- Existe una correspondencia entre JPQL y Api Criteria, en la siguiente tabla podemos identificar cada cláusula con **el método e interfaz del Api Criteria** que necesitaremos para definirlo:

JP QL Clause	Criteria API Interface	Method
SELECT	CriteriaQuery	`select()`
	Subquery	`select()`
FROM	AbstractQuery	`from()`
WHERE	AbstractQuery	`where()`
ORDER BY	CriteriaQuery	`orderBy()`
GROUP BY	AbstractQuery	`groupBy()`
HAVING	AbstractQuery	`having()`

Conceptos fundamentales

- Origen de consulta (Query Roots).
- Expresiones de ruta (Path expressions).

Origen de consulta

- Un origen de consulta en criteria corresponde con **una variable de identificación** usada en la cláusula **FROM** de las consultas JPQL.
- El **interfaz AbstractQuery** proporciona el **método from()** para definir las entidades que formarán parte de la base de nuestra consulta.
- El método admite una **entidad como parámetro** y añade un nuevo origen a la consulta:

```
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);
Root<Employee> emp = c.from(Employee.class);
```

- El **método from()** devuelve una instancia de Root correspondiente al tipo de entidad definido.
- Cada **llamada al método from() es aditiva**, es decir, cada llamada añade otro origen a la consulta que produce un producto cartesiano si hay varios origen definidos y no se añade ningún filtro en la cláusula WHERE

Origen de consulta

- En el siguiente ejemplo vemos la correspondencia entre consulta JPQL entre varias entidades:

```
SELECT DISTINCT d
FROM Department d, Employee e
WHERE d = e.department
```

```
CriteriaQuery<Department> c = cb.createQuery(Department.class);
Root<Department> dept = c.from(Department.class);
Root<Employee> emp = c.from(Employee.class);
c.select(dept).distinct(true).where(cb.equal(dept, emp.get("department")));
```

Expresiones de ruta

- Son una pieza clave debido a la potencia y flexibilidad tanto en JPQL como en Api Criteria.
- Podemos ver la correspondencia entre JPQL y las expresiones de ruta en Api Criteria:

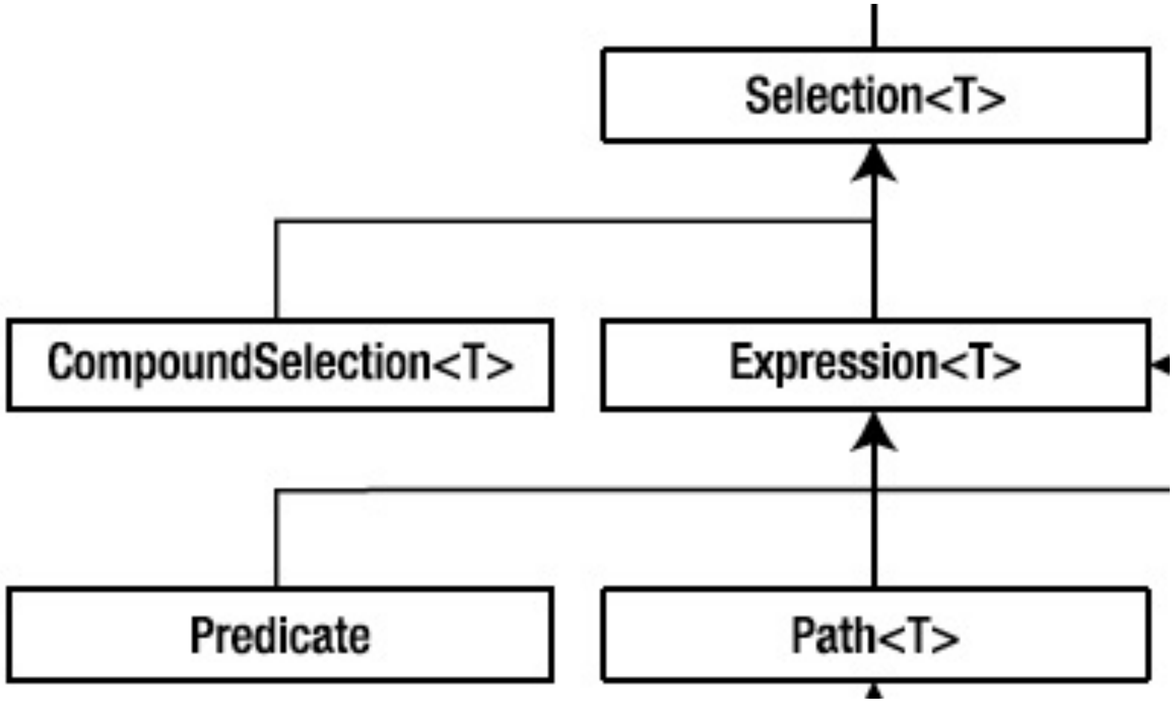
```
SELECT e
FROM Employee e
WHERE e.address.city = 'New York'
```

```
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);
Root<Employee> emp = c.from(Employee.class);
c.select(emp).where(cb.equal(emp.get("address").get("city"), "New York"));
```

- En Api Criteria es necesario almacenar la instancia Root en una variable local para posteriormente utilizarlo para formar expresiones de ruta en las cláusulas que sea necesario. En el ejemplo anterior se utiliza la expresión de ruta para filtrar los resultados en la cláusula WHERE

Cláusula SELECT

- Expresiones únicas:
  - Se utiliza el método **select(Selection<? extends T> selection)** del interfaz CriteriaQuery para construir la cláusula SELECT de una consulta con Criteria.
  - La jerarquía de interfaces se muestra a continuación:



Cláusula SELECT

- Expresiones únicas:
  - Podemos pasar un origen de consulta como parámetro:

```
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);
Root<Employee> emp = c.from(Employee.class);
c.select(emp);
```

  - También podemos pasar al metodo select() una expresión que seleccione un atributo de una entidad o una expresión compatible con el tipo requerido:

```
CriteriaQuery<String> c = cb.createQuery(String.class);
Root<Employee> emp = c.from(Employee.class);
c.select(emp.<String>get("name"));
```

Cláusula SELECT

- Expresiones únicas:
  - Notas:
  - El tipo de la expresión proporcionada al método select() debe ser compatible con el tipo resultado usado para instanciar el objeto CriteriaQuery.
  - El método get() devuelve un objeto Path<Object> porque el compilador no puede inferir el tipo basado en la propiedad name, así que, es necesario indicar el tipo de retorno como hemos realizado en el ejemplo anterior.

Cláusula SELECT

- Expresiones múltiples:
  - Cuando definimos una **cláusula SELECT** con **múltiples expresiones**, debemos hacerlo compatible con el tipo resultado usado para instanciar el objeto CriteriaQuery (definición de consulta).
  - Si definimos el tipo de resultado Tuple el método select() debe recibir un parámetro de tipo CompoundSelection<Tuple>.
  - Una opción es utilizar el método del interfaz CriteriaBuilder
    - CompoundSelection<Tuple> tuple(Selection<?>... selections)

Cláusula SELECT

- Ejemplo:

```
CriteriaQuery<Tuple> c = cb.createTupleQuery();
Root<Employee> emp = c.from(Employee.class);
c.select(cb.tuple(emp.get("id"), emp.get("name")));
```

Cláusula SELECT

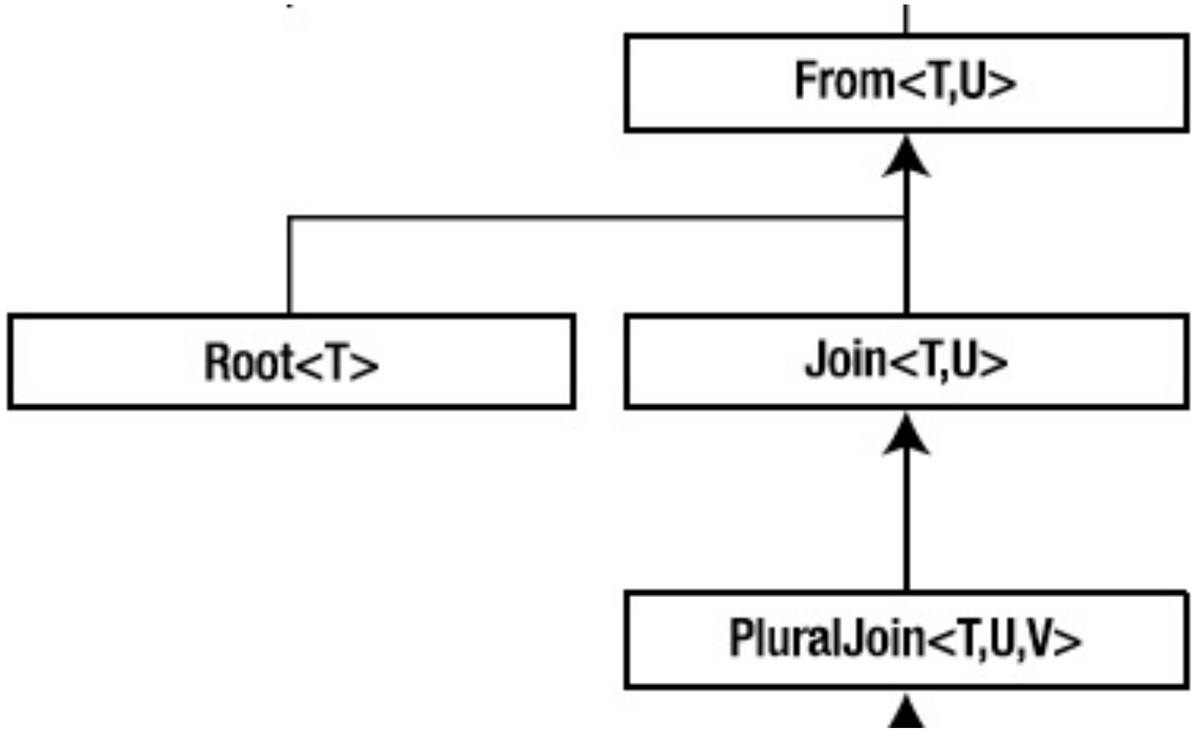
- Otra opción es utilizar el método del interfaz CriteriaQuery:
  - CriteriaQuery<T> multiselect(Selection<?>... selections)
- Ejemplo:

```
CriteriaQuery<Tuple> c = cb.createTupleQuery();
Root<Employee> emp = c.from(Employee.class);
c.multiselect(emp.get("id"), emp.get("name"));
```

Cláusula FROM

- Los Join son creados en Criteria utilizando el **método join()** de la interfaz From. Cualquier origen de consulta (query root) puede ser un join y los join() pueden encadenarse con otro join().

- La jerarquía de interfaces se muestra a continuación:



Cláusula FROM

- El método join está sobrecargado, y por tanto hay numerosas formas de generar el Join, os vamos a detallar una de las más sencillas de manejar.
  - <X,Y> Join<X,Y>join(java.lang.String attributeName, JoinType jt)
- El método join() tiene dos tipos parametrizados que corresponden con la entidad origen y destino de la unión. Esto nos permite mayor claridad para entender la unión que se está realizando.
- Además incluye un **argumento** que indica el **tipo de unión** (inner, left o right):

```
Join<Employee,Project> project = emp.join("projects", JoinType.LEFT);
```

Cláusula WHERE

- La cláusula WHERE se establece utilizando el **método where()** de la interfaz **AbstractQuery**.
- El método acepta argumentos con ninguno o varios Predicate o simplemente una Expresion<Booelean>.
- La clave para construir expresiones en Criteria es el interfaz CriteriaBuilder, que contiene métodos para todos predicados, expresiones y funciones soportadas en JPQL y características adicionales.

Cláusula WHERE

- El **uso de parámetros en Criteria** es diferente de JPQL.
- Es necesario **crear explícitamente** una instancia del tipo correcto del **ParameterExpresion** que será usada posteriormente en alguna expresión condicional.
- El método parameter() necesita un tipo de clase y un nombre para que el parámetro sea usado posteriormente:

```
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);
Root<Employee> emp = c.from(Employee.class);
c.select(emp);
ParameterExpression<String> deptName =cb.parameter(String.class, "deptName");
c.where(cb.equal(emp.get("dept").get("name"), deptName));
```

Cláusula WHERE

- A continuación vemos un ejemplo de construcción de un predicado (Predicate) que filtra por varios parámetros:

```
Predicate criteria = cb.conjunction();
if (name != null) {
  ParameterExpression<String> p = cb.parameter(String.class, "name");
  criteria = cb.and(criteria, cb.equal(emp.get("name"), p));
}
if (deptName != null) {
  ParameterExpression<String> p = cb.parameter(String.class, "dept");
  criteria = cb.and(criteria, cb.equal(emp.get("dept").get("name"), p));
}
```

Cláusula ORDER BY

- El **método orderBy()** de la interfaz CriteriaQuery establece **el orden para una definición** de consulta criteria. El método acepta **uno o varios objetos de tipo Order**.
- Estos objetos son creadas gracias a los **métodos asc() y desc()** de la interfaz CriteriaBuilder.
- El siguiente ejemplo ordena los resultados por **nombre de departamento** ascendente y, en segundo lugar, por **nombre de empleado** descendente.

```
CriteriaQuery<Tuple> c = cb.createQuery(Tuple.class);
Root<Employee> emp = c.from(Employee.class);
Join<Employee,Department> dept = emp.join("dept");
c.multiselect(dept.get("name"), emp.get("name"));
c.orderBy(cb.desc(dept.get(“name")), cb.asc(emp.get("name")));
```

Cláusula GROUP BY

- Los **métodos groupBy() y having()** de la interfaz AbstractQuery son equivalentes a JPQL, ambos métodos admiten uno o más expresiones que se usan para agrupar y filtrar datos.

- A continuación tenemos la relación entre JPQL y Criteria de una consulta que obtiene los empleados y el número de proyectos en los que está asignado siempre que cumpla que está asignado a más de un proyecto:

```
SELECT e, COUNT(p)
FROM Employee e JOIN e.projects p
GROUP BY e
HAVING COUNT(p) >= 2
```

```
CriteriaQuery<Object[]> c = cb.createQuery(Object[].class);
Root<Employee> emp = c.from(Employee.class);
Join<Employee,Project> project = emp.join("projects");
c.multiselect(emp, cb.count(project)).groupBy(emp).having(cb.ge(cb.count(project),2));
```

Sentencias update y delete

- Las operaciones UPDATE son creadas utilizando el método createCriteriaUpdate() de la interfaz CriteriaBuilder.
- Utilizan los mismos métodos para construir las cláusulas FROM y WHERE que las sentencias SELECT.
- Los métodos específicos se encuentran encapsulados en el interfaz CriteriaUpdate.

Sentencias update y delete

- A continuación vemos la relación entre JPQL y Criteria con un ejemplo:

```
UPDATE Employee e
SET e.salary = e.salary + 5000
```

```
CriteriaUpdate<Employee> q = cb.createCriteriaUpdate(Employee.class);
Root<Employee> emp = q.from(Employee.class);
q.set(emp.get("salary"), cb.sum(emp.get("salary"), 5000));
```

- El **método set()** se utiliza para actualizar el valor de la propiedad especificada.

Sentencias update y delete

- Las **operaciones de tipo DELETE** son creadas utilizando el **método createCriteriaDelete()** de la interfaz CriteriaBuilder.
- Utilizan los mismos métodos para construir las cláusulas FROM y WHERE que las sentencias SELECT. Los métodos específicos se encuentran encapsulados en el interfaz CriteriaDelete.
- A continuación vemos la relación entre JPQL y Criteria con un ejemplo:

```
DELETE FROM Employee e
WHERE e.department IS NULL
```

```
CriteriaDelete q = cb.createCriteriaDelete(Employee.class);
Root emp = c.from(Employee.class);
q.where(cb.isNull(emp.get("dept")));
```

Trabajando con lazy

- Introducción.
- Llamar a un método sobre una relación mapeada.
- Fetch Join en JPQL.
- Fetch Join en la API Criteria.
- Named Entity Graph.
- Dynamic Entity Graph.
- Conclusión.

Introducción

- La **carga diferida/lazy** de asociaciones entre entidades es una **buena práctica** bien establecida en JPA.
- Su **objetivo principal** es recuperar **sólo las entidades solicitadas** de la base de datos y cargar las entidades relacionadas **sólo si es necesario**.
- Ese es un buen enfoque si solo necesita las entidades solicitadas. Pero crea trabajo adicional y puede ser la causa de problemas de rendimiento si también necesita algunas de las entidades relacionadas.
- Echemos un vistazo a las diferentes formas de activar la inicialización y sus ventajas y desventajas específicas.

Llamar a un método sobre una relación mapeada

- Comencemos con el enfoque más obvio y, lamentablemente, también el más ineficiente. Utiliza el **método find** en EntityManager y llama a un método en la relación.

```
Order order = this.em.find(Order.class, orderId);
order.getItems().size();
```

- Este código funciona **bien**, es **fácil de leer** y se usa con **frecuencia**. Entonces, ¿cuál es el problema?

- Este código realiza una **consulta adicional** para **inicializar** la relación. Eso no parece un problema real, pero calculemos la cantidad de consultas ejecutadas en un escenario más real. Digamos que tienes una entidad con 5 asociaciones que necesitas inicializar. **Entonces obtendrás 1 + 5 = 6 consultas**.

- Bien, esas son 5 consultas adicionales. **Eso todavía no parece un gran problema**. Pero nuestra aplicación será utilizada por **más de un usuario en paralelo**.

Llamar a un método sobre una relación mapeada

- Digamos que su sistema tiene que atender a **100 usuarios paralelos**. Entonces obtendrás **100 + 5*100 = 600 consultas**.
- Esto se denomina **el problema de los n+1 select** y debería ser obvio que no es un buen enfoque.
- Tarde o temprano, la **cantidad de consultas adicionales realizadas ralentizará su aplicación**.
- Por lo tanto, **deberías intentar evitar este enfoque y echar un vistazo a otras opciones**.

Fetch Join en JPQL

- Una **mejor opción** para inicializar asociaciones diferidas es utilizar una **consulta JPQL** con un FETCH JOIN.

```
Query q = this.em.createQuery("SELECT o FROM Order o JOIN FETCH o.items i WHERE o.id = :id");
q.setParameter("id", orderId);
newOrder = (Order) q.getSingleResult();
```

- Eso le dice al administrador de entidades que cargue **la entidad seleccionada y la relación** dentro de la misma consulta.

Fetch Join en JPQL

- Las **ventajas** y **desventajas** de este enfoque son:
  - La ventaja es que JPA recupera **todo dentro de una consulta**. Desde el punto de vista del **rendimiento**, esto es mucho **mejor** que el **primer enfoque**.
  - Y la **principal desventaja** es que necesita **escribir código adicional** que ejecute la consulta. Pero se pone aún peor si la **entidad tiene múltiples asociaciones** y necesita inicializar diferentes asociaciones para diferentes casos de uso. En este caso, debe escribir una query para cada combinación requerida de asociaciones que desee inicializar. **Eso puede resultar bastante complicado**.
- El uso de FETCH JOIN en declaraciones JPQL puede requerir una **gran cantidad de consultas**, lo que **dificultará el mantenimiento** del código base.
- Antes de comenzar a escribir muchas consultas, debe pensar en la **cantidad de combinaciones de fetch join diferentes que podría necesitar**. Si el número es **bajo**, este es un buen enfoque para limitar el número de consultas realizadas.

Fetch Join en la API Criteria

- Este enfoque es **básicamente el mismo que el anterior**. Pero esta vez estás utilizando la API de Criteria en lugar de la consulta JPQL:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery q = cb.createQuery(Order.class);
Root o = q.from(Order.class);
o.fetch("items", JoinType.INNER);
q.select(o);
q.where(cb.equal(o.get("id"), orderId));

Order order = (Order)this.em.createQuery(q).getSingleResult();
```

Fetch Join en la API Criteria

- Las ventajas y desventajas son las **mismas** que para la consulta JPQL **con fetch join**.
- JPA recupera la entidad y la relación con una consulta de la base de datos y necesita un **código** específico para cada combinación de **asociaciones**.
- Pero a menudo ya tiene muchos casos de uso de códigos de consulta específicos, si está utilizando la API de Criteria. Entonces **esto podría no ser un gran problema**. Si ya está utilizando la API de Criteria para crear la consulta, este es un buen enfoque para reducir la cantidad de consultas realizadas.

Named Entity Graph

- Es una nueva característica de JPA 2.1.
- Se puede utilizar para definir un gráfico de entidades que se consultarán desde la base de datos.
- La definición de un gráfico de entidad se realiza mediante anotaciones y es independiente de la consulta.

```
@Entity
@NamedEntityGraph(name = "graph.Order.items", attributeNodes = @NamedAttributeNode("items"))
public class Order implements Serializable {
  ...
}
```

Named Entity Graph

- El EntityManager puede usar Named entity graph para buscar:

```
EntityGraph graph = this.em.getEntityGraph("graph.Order.items");
Map hints = new HashMap();
hints.put("javax.persistence.fetchgraph", graph);
Order order = this.em.find(Order.class, orderId, hints);
```

- Esta es básicamente una versión mejorada de nuestro primer enfoque.
- El EntityManager **recuperará el gráfico definido de entidades** de la base de datos con una consulta.
- La única desventaja es que necesita anotar un gráfico de entidad con nombre para cada combinación de asociaciones que se recuperarán dentro de una consulta.
- Necesitará **menos anotaciones adicionales** como en nuestro segundo enfoque, pero aún así puede resultar bastante complicado. Por lo tanto, los gráficos de entidades con nombre son una gran solución, si solo necesita definir una cantidad limitada de ellos y reutilizarlos para diferentes casos de uso. De lo contrario, el código será difícil de mantener.

Dynamic Entity Graph

- El gráfico de entidad dinámica es **similar** al gráfico de entidad con nombre. La única diferencia es que el gráfico de entidad se define mediante una API de Java:

```
EntityGraph graph = this.em.createEntityGraph(Order.class);
Subgraph itemGraph = graph.addSubgraph("items");

Map hints = new HashMap();
hints.put("javax.persistence.loadgraph", graph);

Order order = this.em.find(Order.class, orderId, hints);
```

Dynamic Entity Graph

- La definición a través de una API puede ser **una ventaja y una desventaja**.
- Si necesita muchos **gráficos de entidades** específicos de casos de uso, podría ser mejor definir el gráfico de entidades dentro del código Java específico y no agregar una anotación adicional a la entidad. **Esto evita entidades con docenas de anotaciones**.
- Por otro lado, el **gráfico de entidades dinámicas** requiere **más código** y **un método adicional** para ser reutilizable.
- Por lo tanto, recomiendo utilizar **gráficos de entidades dinámicas** si necesita crear un gráfico de caso de uso específico, **que no reutilizará**. Si desea reutilizar el gráfico de entidad, es más fácil anotar un gráfico de entidad con nombre.

Conclusión

- Cada estrategia tiene sus ventajas y desventajas:
  - Inicializar una relación diferida llamando a un método en una relación asignada provoca una consulta adicional. **Esto debería evitarse por motivos de rendimiento**.
  - El uso de FETCH JOIN reduce la **cantidad de consultas a una**, pero es posible que necesite **muchas consultas diferentes**.
  - La API de Criteria también admite FETCH JOIN y necesita un código específico para cada combinación de asociaciones que se inicializarán.
  - Los **gráficos de entidades con nombre** son una **buena solución si reutiliza el gráfico** definido en nuestro código.
  - Los **gráficos de entidades dinámicas** pueden ser la **mejor solución** si necesita definir un gráfico de caso de uso específico.