SQL nativo con JPA e Hibernate

[30/06/2018](https://danielme.com/2018/06/30/sql-nativo-con-jpa-e-hibernate/)~ [DANIELME.COM](https://danielme.com/author/danielmedotcom/)

Última actualización: 27/06/2022



A pesar de la existencia de herramientas tan poderosas como [Hibernate](http://hibernate.org/orm/" \t "_blank), producto compatible con el estándar [JPA](https://danielme.com/2021/08/03/curso-jakarta-ee-jpa-con-hibernate-introduccion/" \t "_blank), existen escenarios en los que necesitamos trabajar directamente con el lenguaje SQL de nuestra base de datos.

El caso más típico es la ejecución de consultas complejas que requieren de características avanzadas de las bases de datos tales como tablas derivadas o funciones analíticas, no disponibles en el lenguaje [JPQL\HQL](https://danielme.com/2022/01/19/curso-jakarta-ee-jpa-con-hibernate-el-lenguaje-jpql-hql-consultas-basicas-y-modificaciones/), ofrecido por Hibernate y en el que trabajamos con entidades y no con las tablas. O, simplemente, la consulta es más sencilla y eficiente escribirla con SQL.

Siempre podemos recurrir al uso directo de JDBC o, en el caso de Spring, apoyarnos en [Spring JDBC Template](https://danielme.com/2017/10/02/spring-jdbc-template-simplificando-el-uso-de-sql/). Sin embargo, si ya estamos utilizando JPA en nuestros proyectos, es muy recomendable aprovechar su magnífico soporte para la ejecución de consultas SQL, denominadas por JPA como «nativas». Esto nos permite beneficiarnos de algunas de las funcionalidades de JPA. Es perfectamente lícito usar JPA en lo que es realmente bueno (la sincronización automática de las entidades con la base de datos) y dejar para SQL la explotación de los datos.

En este tutorial veremos las principales opciones que tenemos para trabajar con SQL en JPA 2.2 e Hibernate 5. Asumo que el lector ya posee unos conocimientos básicos de JPA.

**IMPORTANTE**. El contenido de este tutorial, con ciertas adaptaciones, también se encuentra disponible en mi curso integral de [Jakarta EE](https://danielme.com/curso-jakarta-ee-indice/).

[Curso Jakarta EE 9 (38). JPA con Hibernate (21): trabajando con SQL.](https://danielme.com/2022/06/22/curso-jakarta-ee-jpa-con-hibernate-trabajando-con-sql/)  
[Curso Jakarta EE 9 (39). JPA con Hibernate (22): procedimientos almacenados SQL.](https://danielme.com/2022/07/13/curso-jakarta-ee-jpa-con-hibernate-procedimientos-almacenados-sql/)

Proyecto de ejemplo

Exploraremos las posibilidades de ejecución de SQL en JPA «jugando» con una aplicación Maven para Java 17 con Hibernate 5 que utiliza la base de datos embebida en memoria [HyperSQL (HSQLDB)](http://hsqldb.org/). Tanto esta última como las consultas SQL son irrelevantes para el propósito del tutorial.

Tenemos una única entidad.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @Entity  @Table(name = "user")  public class User {        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)      private Long id;        @Column(nullable = false)      private String name;        @Column(nullable = false, unique = true)      private String email; |

Todas las operaciones de acceso a la base de datos, realizadas a con el [gestor de entidades](https://danielme.com/2021/09/09/curso-jakarta-ee-jpa-con-hibernate-contexto-persistencia-entitymanager/) de JPA implementado por Hibernate, se irán escribiendo en la clase *UserDAO*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class UserDao {        private final EntityManager em;        public UserDao(EntityManager em) {          this.em = em;      }  ... |

La configuración de Hibernate y la base de datos se establece en el fichero */src/main/resources/META-INF/persistence.xml*. El modo *update* de Hibernate creará de manera automática la tabla *users* si no existe.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | <persistence xmlns="<http://java.sun.com/xml/ns/persistence>"               xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"               xsi:schemaLocation="<http://java.sun.com/xml/ns/persistence> <http://java.sun.com/xml/ns/persistence/persistence_2_1.xsd>"               version="2.1">      <persistence-unit name="sqlDemoPersistence" transaction-type="RESOURCE\_LOCAL">            <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>            <class>com.danielme.blog.nativesql.entities.User</class>            <properties>              <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver"/>              <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:mem:testdb"/>              <property name="javax.persistence.jdbc.user" value="sa"/>              <property name="javax.persistence.jdbc.password" value=""/>              <property name="hibernate.hbm2ddl.auto" value="update"/>              <property name="hibernate.format\_sql" value="true"/>              <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>          </properties>        </persistence-unit>  </persistence> |

Usaremos la siguiente clase para hacer pruebas gracias a [JUnit 4](https://danielme.com/2016/09/07/tutorial-testing-con-junit-4/). Aunque no los muestre en el tutorial, contendrá tests para los distintos métodos que vayamos viendo.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | package com.danielme.blog.nativesql.dao;    import com.danielme.blog.nativesql.DbUtils;  import org.junit.BeforeClass;      import javax.persistence.EntityManager;  import javax.persistence.EntityManagerFactory;  import javax.persistence.Persistence;    public class UserDaoTest {        private static UserDao userDao;        @BeforeClass      public static void setup() throws Exception {          EntityManagerFactory entityManagerFactory = Persistence                  .createEntityManagerFactory("sqlDemoPersistence");          EntityManager entityManager = entityManagerFactory.createEntityManager();          userDao = new UserDao(entityManagerFactory.createEntityManager());          DbUtils.loadScript(entityManager, "users.sql");      } |

Al inicio, *UserDaoTest* crea una instancia de *UserDAO* que será el objetivo de las pruebas. Ello requiere la creación de un gestor de entidades o *EntityManager* para la unidad de persistencia que acabamos de definir en el fichero *persistence.xml*. Además, se ejecuta el script */src/test/resources/users.sql* para poner en la BD los datos de prueba. He realizado esta tarea de forma «artesanal» para mantener el ejemplo lo más simple posible; recomiendo [Database Rider](https://github.com/database-rider/database-rider) para estos menesteres. Para más información sobre JUnit, consultar el tutorial [Testing con JUnit 4](https://danielme.com/2016/09/07/tutorial-testing-con-junit-4/).

Por último, he incluido una configuración básica de [Log4j 2](https://logging.apache.org/log4j/2.x/) que nos permitirá ver en la salida estándar de Java las trazas de Hibernate.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | <?xml version="1.0" encoding="UTF-8"?>  <Configuration>        <Appenders>          <Console name="Console" target="SYSTEM\_OUT">              <PatternLayout pattern="%c{1}:%L - %m%n"/>          </Console>      </Appenders>        <Loggers>          <Logger name="org.hibernate" level="info" additivity="false">              <AppenderRef ref="Console"/>          </Logger>          <Logger name="org.hibernate.SQL" level="debug" additivity="false">              <AppenderRef ref="Console"/>          </Logger>          <Logger name="org.hibernate.type" level="debug" additivity="false">              <AppenderRef ref="Console"/>          </Logger>            <Root level="info">              <AppenderRef ref="Console"/>          </Root>      </Loggers>    </Configuration> |

Ejecución de SQL con JPA

Para poder ejecutar SQL, lo primero será recurrir a las sobrecargas del método [createNativeQuery](https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html" \l "createNativeQuery-java.lang.String-" \t "_blank) del gestor de entidades las cuales reciben una cadena con la sentencia SQL. Más adelante veremos que también podemos utilizar el método [createNamedQuery](https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html" \l "createNamedQuery-java.lang.String-" \t "_blank) para definir las consultas de manera estática con anotaciones o en un fichero XML.

|  |  |
| --- | --- |
| 1 | Query nativeQuery = em.createNativeQuery("SELECT id, name, email FROM user ORDER BY email DESC"); |

Vemos que la llamada a *createNativeQuery* retorna una instancia de [Query](https://jakarta.ee/specifications/persistence/2.2/apidocs/javax/persistence/query) que usaremos de igual manera que cuando escribimos las consultas con JPQL\HQL y la API Criteria (métodos *createQuery*). Así pues, en el momento que tengamos un objeto *Query* (o uno de sus subtipos) la forma de trabajar es común a todas las estrategias de escritura de consultas aceptadas por JPA.

Una vez que tengamos la consulta definida en un objeto Query, se ejecuta con el método [getResultList()](https://jakarta.ee/specifications/persistence/2.2/apidocs/javax/persistence/query" \l "getResultList()) el cual devuelve una lista en la que cada elemento consiste en un array de Object (Object[]). Cada posición del array se corresponde con uno de los elementos retornados por la claúsula SELECT, respetándose el orden en el que aparecen. Para el ejemplo, tendríamos algo así.

LISTA OBJECT[]

(0) ---> [id] [name] [email]

(1) ---> [id] [name] [email]

Esta estructura de datos no es nada práctica, así que lo mejor será recorrer la lista y recoger los datos en la clase adecuada. En nuestro caso, nos sirve *User*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public List<User> findAll() {      Query nativeQuery = em.createNativeQuery("SELECT id, name, email FROM user ORDER BY email DESC");        List<Object[]> results = nativeQuery.getResultList();        return results              .stream()              .map(result -> new User(((BigInteger) result[0]).longValue(), (String) result[1], (String) result[2]))              .collect(Collectors.toList());  } |

Cuando el resultado sea exactamente uno y siempre exista, aunque el valor retornado por la base de datos sea NULL, usaremos el método [query#getSingleResult()](https://jakarta.ee/specifications/persistence/2.2/apidocs/javax/persistence/query" \l "getSingleResult()). Lo veremos en varios ejemplos del tutorial.

Parámetros de entrada

El sistema de gestión de parámetros o variables de entrada a reemplazar en tiempo de ejecución que define JPA para las consultas nativas es el mismo de JDBC. Se indican en la consulta con el caracter ‘?’ y sus valores se proporcionan respetando el orden en el que aparecen con los métodos *setParameter* de *Query*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public List<User> findByText(String text) {      Query nativeQuery = em.createNativeQuery("SELECT \* FROM user " +                  "WHERE name LIKE LOWER(?) OR email LIKE LOWER(?)", User.class);      String textSearch = text == null ? "%" : "%" + text.toLowerCase() + "%";      nativeQuery.setParameter(1, textSearch);      nativeQuery.setParameter(2, textSearch);      return nativeQuery.getResultList();  } |

Es un sistema confuso y propenso a errores. En el ejemplo, hay dos parámetros que en realidad son el mismo, pero eso no queda patente al ver la cadena con la consulta. Y cuanto más parámetros aparezcan, menos legible será el código. Si hay muchos, hay que andar contándolos para no equivocarnos al darles valor. Creánme cuando digo que he perdido mucho tiempo con detalles de este tipo ;).

Por fortuna, Hibernate admite en SQL el sistema de parámetros de JPQL, más práctico. Consiste en asignar a cada parámetro distinto un nombre único precedido por dos puntos. Al darles valor, se usa el nombre (sin los dos puntos) para indicar el parámetro. Solo se le da valor a cada parámetro una vez.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public List<User> findByText(String text) {      Query nativeQuery = em.createNativeQuery("SELECT \* FROM user " +                  "WHERE name LIKE LOWER(:text) OR email LIKE LOWER(:text)", User.class);      String textSearch = text == null ? "%" : "%" + text.toLowerCase() + "%";      nativeQuery.setParameter("text", textSearch);      return nativeQuery.getResultList();  } |

En vez de nombres, se pueden usar números precedidos por «?». No tienen por qué seguir un orden secuencial, aunque es lo más lógico.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public List<User> findByText(String text) {      Query nativeQuery = em.createNativeQuery("SELECT \* FROM user " +                  "WHERE name LIKE LOWER(?1) OR email LIKE LOWER(?1)", User.class);      String textSearch = text == null ? "%" : "%" + text.toLowerCase() + "%";      nativeQuery.setParameter(1, textSearch);      return nativeQuery.getResultList();  } |

Lo que jamás debemos hacer es «pegar» los valores a la cadena con la consulta porque nos expone a problemas de seguridad de tipo [SQL injection](https://es.wikipedia.org/wiki/Inyecci%C3%B3n_SQL). Además, el código queda más claro si usamos parámetros en lugar de ir pegando cadenas.

Procesamiento de los resultados

Hemos visto que los registros que obtenemos como respuesta a una consulta SQL consisten en Object[]. Sin embargo, en función de lo que devuelva la consulta contamos con otras alternativas.

Escalar

Si el resultado es un valor único convertible en un tipo básico de Java (un número, una cadena, etcétera), es suficiente con aplicarle el casting adecuado. En el siguiente ejemplo, para la función de conteo (COUNT) el controlador JDBC de HSQLDB retorna un BigInteger.

|  |  |
| --- | --- |
| 1  2  3 | public BigInteger count() {      return (BigInteger) em.createNativeQuery("SELECT count(\*) FROM user").getSingleResult();  } |

Entidad

Si el resultado coincide exactamente con una entidad (todos sus atributos, ni más ni menos), la conversión a la misma puede ser automática. Basta con indicar la clase de la entidad cuando construyamos el objeto *Query*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public List<User> findAll() {      Query nativeQuery = em.createNativeQuery("SELECT id, name, email FROM user ORDER BY email DESC", User.class);      return nativeQuery.getResultList();  }    public User findById(Long id) {      Query nativeQuery = em.createNativeQuery("SELECT id, name, email FROM user WHERE id = ?", User.class);      nativeQuery.setParameter(1, id);      return (User) nativeQuery.getSingleResult();  } |

Mucho cuidado con el código anterior si tenemos en mente la manera en la que se crean consultas con JPQL. Aunque indiquemos una entidad, *createNativeQuery* sigue devolviendo un objeto de tipo Query en lugar de una query tipada ([TypedQuery](https://jakarta.ee/specifications/persistence/2.2/apidocs/javax/persistence/typedquery)) tal y como hacen los métodos *createQuery* (JPQL y Criteria API).

Si tenemos dos entidades, porque, por ejemplo, queremos recuperar una y conseguir que ya incluya una relación de tipo [Lazy](https://danielme.com/2021/11/18/curso-jakarta-ee-jpa-con-hibernate-carga-de-relaciones-lazy-loading-bytecode-enhancement/), lo explico [aquí](https://danielme.com/2022/06/22/curso-jakarta-ee-jpa-con-hibernate-trabajando-con-sql/).

El resultado no es una entidad

**Tuplas**

El uso de la interfaz [Tuple](https://docs.jboss.org/hibernate/jpa/2.1/api/javax/persistence/Tuple.html) nos abstrae del array Object[] y permite acceder a los resultados a través del alias de la columna -o su posición empezando por cero- en la proyección de la SELECT.

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7 | public List<User> findAllWithTuples() {      Query nativeQuery = em.createNativeQuery("SELECT id, name, email FROM user ORDER BY email DESC", Tuple.class);      List<Tuple> tuples = nativeQuery.getResultList();      return tuples.stream()              .map(t -> new User(t.get("id", BigInteger.class).longValue(), t.get("name", String.class), t.get("email", String.class)))              .collect(Collectors.toList());  } | |
| 1  2  3  4  5  6  7 | public List<User> findAllWithTuples() {      Query nativeQuery = em.createNativeQuery("SELECT id, name, email FROM user ORDER BY email DESC", Tuple.class);      List<Tuple> tuples = nativeQuery.getResultList();      return tuples.stream()              .map(t -> new User(t.get(0, BigInteger.class).longValue(), t.get(1, String.class), t.get(2, String.class)))              .collect(Collectors.toList());  } |

**SqlResultSetMapping**

Una de las capacidades de JPQL que más me gustan por su practicidad es el volcado directo de la consulta en una clase o *Record* mediante su constructor.

|  |  |
| --- | --- |
| 1  2 | SELECT NEW com.danielme.jakartaee.jpa.dto.ExpenseSummaryDTO(e.concept, e.amount, e.date)  FROM Expense e |

Lo anterior no es posible en una consulta nativa, pues la base de datos no sabe qué es eso de NEW. Pero JPA permite configurar la transformación entre el resultado y la clase en la que queremos recogerlo mediante un constructor. Supongamos que queremos ejecutar la siguiente consulta.

|  |  |
| --- | --- |
| 1 | select id, CONCAT(name, '-', email) as concat From user |

Y obtener un listado de esta clase.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class UserDetail {        private Long id;      private String details;            public UserDetail() {          super();      }        public UserDetail(Long id, String details) {          super();          this.id = id;          this.details = details;      } |

Podemos conseguir que el resultado de la consulta se «mapee» de forma automática en instancias de esta clase *UserDetail* configurando la relación entre las columnas obtenidas en el SQL y los parámetros del constructor. De este modo, estamos instruyendo a Hibernate cómo llamar al constructor con las columnas del resultado. Esta configuración se creará con anotaciones en una clase que sea una entidad JPA.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @SqlResultSetMapping(          name = "DetailMapping",          classes = @ConstructorResult(                  targetClass = UserDetail.class,                  columns = {                      @ColumnResult(name = "id", type=Long.class),                      @ColumnResult(name = "concat")                      }))  @Entity  @Table(name = "user")  public class User { |

La otra opción es hacerlo en el fichero */META-INF/orm.xml*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <entity-mappings          xmlns="<http://xmlns.jcp.org/xml/ns/persistence/orm>"          version="2.2">        <sql-result-set-mapping name="DetailMapping">          <constructor-result target-class="com.danielme.blog.nativesql.UserDetail">              <column name="id" class="java.lang.Long"/>              <column name="concat"/>          </constructor-result>      </sql-result-set-mapping>    </entity-mappings> |

Proporcionaremos el nombre del *SqlResultSetMapping* al construir el objeto *Query* y ya lo tenemos.

|  |  |
| --- | --- |
| 1  2  3  4 | public List<UserDetail> findAllDetail() {      Query nativeQuery = em.createNativeQuery("SELECT id, CONCAT(name, '-', email) AS concat FROM user", "DetailMapping");      return nativeQuery.getResultList();  } |

ResultTransformer (Hibernate)

Hibernate ofrece numerosas funcionalidades no recogidas en el estándar JPA. Una de ellas son los transformadores de resultados que permiten personalizar el procesamiento de los resultados y pueden aplicarse a cualquier consulta, ya sea JPQL, Criteria API o SQL.

Un transformador es una implementación de los dos métodos de la interfaz [ResultTransformer](https://docs.jboss.org/hibernate/orm/current/javadocs/org/hibernate/transform/ResultTransformer.html).

* transformTuple: se *invoca* por cada registro del ResultSet, y recibimos el Object[] con las columnas del registro así como los nombres de las columnas que se devuelven. Lo usaremos para crear y devolver el objeto con los datos del registro.
* transformList: se invoca tras el procesamiento completo del ResultSet y, por tanto, después de todas las llamadas a *transformTuple*. Aquí recibimos la lista con todos los registros que hemos mapeado con el método *transformTuple* para hacer cualquier tipo de tratamiento final.

Implementemos de nuevo el método *findAllDetail* utilizando un *ResultTransformer*. Tenemos que acceder a la clase de Hibernate que implementa *Query* para poder indicarle el transformador que queremos aplicar con el método [setResultTransformer](https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/query/Query.html" \l "setResultTransformer-org.hibernate.transform.ResultTransformer-).

Nota. setResultTransformer fue marcado obsoleto en Hibernate 5.2, pero no se ofrece una alternativa hasta Hibernate 6.0, versión en la que sigue disponible esta funcionalidad tal y como la explico en el presente artículo.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public List<UserDetail> findAllDetailTransformer() {      NativeQuery nativeQuery = ((Session) this.em.getDelegate()).createSQLQuery("SELECT id, CONCAT(name, '-', email) AS concat FROM user");      nativeQuery.setResultTransformer(new DetailTransformer());      return nativeQuery.list();  }    private static class DetailTransformer implements ResultTransformer {        private static final long serialVersionUID = 1L;        @Override      public Object transformTuple(Object[] tuple, String[] aliases) {          return new UserDetail(((BigInteger) tuple[0]).longValue(), (String) tuple[1]);      }        @Override      public List transformList(List collection) {          return collection;      }    } |

Obsérvese que hemos vuelto a realizar el mismo procesamiento de la respuesta que en la versión previa del método. En apariencia, la única ventaja que obtenemos al utilizar el *ResultTransformer* en nuestro ejemplo es la estandarización del tratamiento del resultado.

Lo interesante es que Hibernate nos regala varias implementaciones de *ResultTransformer*. Una de ellas realiza de forma automática la transformación basándose en la existencia de setters en la clase compatibles con los nombres y tipos de los elementos del resultado. Se trata de [AliasToBeanResultTransformer](https://docs.jboss.org/hibernate/orm/5.6/javadocs/org/hibernate/transform/AliasToBeanResultTransformer.html" \t "_blank) y podemos usarla de la siguiente forma haciendo que los nombres de los alias del SQL coincidan con los atributos con setters del bean y con sus tipos.

|  |  |
| --- | --- |
| 1  2  3  4  5 | public List<UserDetail> findAllAliasToBean() {      Query query = em.createNativeQuery("SELECT id as \"id\", CONCAT(name, '-', email) AS \"details\" FROM user");      query.unwrap(org.hibernate.query.Query.class).setResultTransformer(new AliasToBeanResultTransformer(UserDetail.class));      return query.getResultList();  } |

NOTA: los alias de las columnas están con dobles comillas para evitar que HSQLDB los convierta a mayúsculas. Es una particularidad de esta base de datos que no aplica a MySQL o PostgreSQL.

Puesto que para el identificador Hibernate devuelve el valor encapsulado en un objeto *BigInteger*, vamos a cambiar el tipo de este atributo en UserDetail

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | package com.danielme.blog.nativesql;    import java.math.BigInteger;    public class UserDetail {        private BigInteger id;      private String details;        public UserDetail() {          super();      }        public UserDetail(BigInteger id, String details) {          super();          this.id = id;          this.details = details;      }        public BigInteger getId() {          return id;      }        public void setId(BigInteger id) {          this.id = id;      }        public String getDetails() {          return details;      }        public void setDetails(String details) {          this.details = details;      }    } |

[AliasToBeanConstructorResultTransformer](https://docs.jboss.org/hibernate/orm/5.6/javadocs/org/hibernate/transform/AliasToBeanConstructorResultTransformer.html) es otro de los transformadores que encontramos en el paquete org.hibernate.transform. Permite usar un constructor sin necesidad de configurar un *ConstructorResult*. Si bien nos obliga a declararlo con la clase [Constructor](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/reflect/Constructor.html), su ventaja sobre *AliasToBeanResultTransformer* es que no se requieren los métodos setters. En consecuencia, la clase para el resultado puede ser inmutable (atributos finales), algo deseable en este tipo de clases, e imprescindible para poder usar los *Records* de Java 17. Veamos precisamente un ejemplo con *Record*.

|  |  |  |
| --- | --- | --- |
| 1  2 | public record UserDetailRecord(BigInteger id, String details) {  } | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | | public List<UserDetailRecord> findAllConstructorTransformer() {      Constructor<UserDetailRecord> constructor;      try {          constructor = UserDetailRecord.class.getConstructor(BigInteger.class, String.class);      } catch (NoSuchMethodException ee) {          throw new RuntimeException("constructor for " + UserDetailRecord.class + "not found !!!");      }      Query query = em.createNativeQuery("SELECT id as \"id\", CONCAT(name, '-', email) AS \"details\" FROM user");      query.unwrap(org.hibernate.query.Query.class)              .setResultTransformer(new AliasToBeanConstructorResultTransformer(constructor));        return query.getResultList();  } | |

Consultas nombradas

Al igual que sucede con JPQL, las consultas SQL pueden definirse de forma estática (no modificables en tiempo de ejecución) como consultas nombradas (Named Queries) de JPA utilizando tanto anotaciones como XML (orm.xml). Por ejemplo, vamos a hacer nuevas versiones de los métodos *findById* y *findAllDetail*.

En primer lugar, definimos las consultas en la entidad *User* con la anotación [@NamedNativeQuery](https://jakarta.ee/specifications/persistence/2.2/apidocs/javax/persistence/namednativequeries). Además de la consulta con el atributo *query*, opcionalmente indicamos la entidad en la que se debe mapear el resultado o el bien un *resultSetMapping* si lo necesitamos. Hasta este momento, estos valores los habíamos usado como argumentos del método *createNativeQuery*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @NamedNativeQueries({      @NamedNativeQuery(name = "selectFindById", query = "SELECT id, name, email FROM user WHERE id = ?", resultClass = User.class),      @NamedNativeQuery(name = "selectFindAllDetail", query = "SELECT id, CONCAT(name, '-', email) AS concat FROM user", resultSetMapping = "DetailMapping")  })  @Entity  @Table(name = "user")  public class User { |

En el fichero orm.xml se haría del siguiente modo.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <named-native-query name="selectFindById" result-class="com.danielme.blog.nativesql.entities.User">      <query>SELECT id, name, email FROM user WHERE id = ?</query>  </named-native-query>    <named-native-query name="selectFindAllDetail" result-set-mapping="DetailMapping">      <query>SELECT id, CONCAT(name, '-', email) AS concat FROM user</query>  </named-native-query> |

Ahora ejecutamos estas consultas como cualquier *NamedQuery* sin ninguna particularidad especial por el hecho de estar escritas en SQL.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public User findByIdNamedQuery(Long id) {      Query nativeQuery = em.createNamedQuery("selectFindById");      nativeQuery.setParameter(1, id);      return (User) nativeQuery.getSingleResult();  }    public List<UserDetail> findAllDetailNamedQuery() {      Query nativeQuery = em.createNamedQuery("selectFindAllDetail");      return nativeQuery.getResultList();  } |

Un beneficio del uso de consultas nombradas es la posibilidad de tipar los resultados con [TypedQuery](https://jakarta.ee/specifications/persistence/2.2/apidocs/javax/persistence/typedquery). El requisito es que devuelva un escalar o bien hayamos definido el resultado con *resultSetMapping* \ *resultClass*. Como es nuestro caso, es posible hacer lo siguiente.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public User findByIdNamedQuery(Long id) {      TypedQuery<User> nativeQuery = em.createNamedQuery("selectFindById", User.class);      nativeQuery.setParameter(1, id);      return nativeQuery.getSingleResult();  }    public List<UserDetail> findAllDetailNamedQuery() {      TypedQuery<UserDetail> nativeQuery = em.createNamedQuery("selectFindAllDetail", UserDetail.class);      return nativeQuery.getResultList();  } |

Paginación

La recuperación de datos de forma paginada, esto es, en grupos o páginas, puede realizarse en la consulta SQL.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public List<User> findAll(int first, int max) {      Query nativeQuery = em.createNativeQuery("SELECT \* FROM user ORDER BY id LIMIT :first OFFSET :max", User.class);      nativeQuery.setParameter("first", first);      nativeQuery.setParameter("max", max);      return nativeQuery.getResultList();  } |

Pero es más cómodo usar los métodos [setFirstResult](https://jakarta.ee/specifications/persistence/2.2/apidocs/javax/persistence/query" \l "setFirstResult(int)) y [setMaxResults](https://jakarta.ee/specifications/persistence/2.2/apidocs/javax/persistence/query" \l "setMaxResults(int)) de *Query*.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public List<User> findAll(int first, int max) {      Query nativeQuery = em.createNativeQuery("SELECT \* FROM user ORDER BY id", User.class);      nativeQuery.setFirstResult(first);      nativeQuery.setMaxResults(max);      return nativeQuery.getResultList();  } |

Sea como fuere, aprovecho para destacar que cuando paginamos hay que aplicar un criterio de ordenación para garantizar la correcta «navegación» por las páginas de datos.

Modificaciones

También podemos realizar operaciones de modificación (INSERT, UPDATE, DELETE) ejecutando nuestro SQL con el método [executeUpdate](https://jakarta.ee/specifications/persistence/2.2/apidocs/javax/persistence/query" \l "executeUpdate()" \t "_blank). En este caso la operación debe ejecutarse dentro de una transacción. Por simplicidad, se ha hecho una gestión «chapucera» de la misma; lo habitual en una aplicación web es usar el sistema de transacciones de Spring o del servidor de aplicaciones (p.ej. WildFly) si trabajamos con Jakarta EE.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public void insert(String name, String email) {      em.getTransaction().begin();      Query query = em.createNativeQuery("INSERT INTO user (name, email) VALUES(:name,:email)");      query.setParameter("name", name);      query.setParameter("email", email);      query.executeUpdate();      em.getTransaction().commit();  } |

Debemos tener en cuenta que los cambios realizados directamente con SQL en la base de datos, aunque sea mediante el gestor de entidades, no se reflejan en las entidades que ya existieran en los contextos de persistencia en uso. Tampoco se ejecutarán sus eventos (@PrePersist, @PreUpdate, etcétera)