JPA 2.1

...

AVANZADO

francisco.philip@gmail.com

Temario

Creación de contextos
- Standalone
- JEE

EntityManager

Mapping
- Tablas, identidades
- Relaciones
- Bloqueo y concurrencia
- Casos complejos
  - Herencia
  - Embebidos
  - Colecciones

Transaccionabilidad
- Programática
- Declarativa JEE
  - EJB
  - CDI

JQPL
- Como aplicar al patrón DAO
- JPQL Dinámicas vs no dinámicas

Criteria
- CriteriaBuilder
- CriteriaQuery
- Where
- SubQuery
- Parameters
- Functions
- Operations
- Metamodel
- Tuple Queries
- Best Practices

Mejoras 2.1
- CriteriaUpdate
- CirteriaDelete
- Store Procedures
- ConstructorResult
- Runtime/Progammatic named queries
- Contextos desincronizaoos
- EntityListener injectables
- Entity Graphs
- Converters
- Generadores DLL

Patron DAO
- Porque si, porque no.
- Buenas prácticas

No SQL
- Ejemplos MongoDB

Auditoría
- Uso y limitaciones de @Audit

Logging
- Niveles de logging con sl4j e Hibernate

Buenas Prácticas
- Despejar Dudas

Testing básico
- Que testear
- Buenas prácticas
- JUnit
- TestNG
- Arquillian / OpenEJB

JPA

Introducción

Motivación

Introducción

Las bases de datos relacionales almacenan la información mediante tablas, filas, y columnas, de manera que para almacenar un objeto hay que realizar una correlación entre el sistema orientado a objetos de Java y el sistema relacional de nuestra base de datos. *JPA* (Java Persistence API) es una abstracción sobre *JDBC* que nos permite realizar dicha correlación de forma sencilla, realizando por nosotros toda la conversión entre nuestros objetos y las tablas de una base de datos. Esta conversión se llama *ORM* (Object Relational Mapping), y puede configurarse a través de metadatos (mediante xml o anotaciones). Por supuesto, JPA también nos permite seguir el sentido inverso, creando objetos a partir de las tablas de una base de datos, y también de forma transparente. A estos objetos los llamaremos desde ahora *entidades* (entities).

Motivación

JPA establece una interface común que es implementada por un proveedor de persistencia, de manera que podemos elegir en cualquier momento el proveedor que más se adecue a nuestras necesidades.

Así, es el proveedor quién realiza el trabajo, pero siempre funcionando bajo la API de JPA.

- EclipseLink (Eclipse)
- Hibernate (RedHat)
- Open JPA (Apache)
- DataNucleus
- Ebean (SourceForge)
- TopLink Essentials (Glassfish)
- TopLink (Oracle)
- Kodo (Oracle)

NOVEDADES

Mejoras y novedades del API Java de Persistencia promovidas en la versión 2.1

MEJORAS y NOVEDADES JPA 2.1

Attribute Converter.

Named Stored Procedure Query.

Stored Procedure Query.

Criteria API Bulk Operations.

Constructor Result Mapping.

Unsynchronized Persistence Context.

Named Entity Graph.

Entity Graph.

Generación de esquemas.

Programmatic Named Queries.

Soporte CDI en Entity Listener.

Mejoras JPQL.

Features

Versión 2.1

Persistence Units

Standalone
JEE

Las unidades de persistencia no solo agrupa entidades y sus relaciones, sino también comportamiento y configuración.

Persistence Units

Las unidades de persistencia en JPA son creadas mediante la clase *Persistence* que nos permite crear factorías de gestores de entidades.

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("jpa21:persistence-unit");
```

y son definidas en el fichero de configuración *persistence.xml*

```
<persistence-unit name="jpa21:persistence-unit">
      <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
      <class>houseware.learn.jpa21.persistenceUnit.League</class>
      <class>houseware.learn.jpa21.persistenceUnit.Club</class>
      <class>houseware.learn.jpa21.persistenceUnit.Player</class>
</persistence-unit>
```

Configuraciones

Las unidades de persistencia poseen un con conjunto de propiedades que customizan el comportamiento y su configuración.

```
Map<String, String> map = new HashMap<>();
map.put("hibernate.dialect", "org.hibernate.dialect.HSQLDialect");
map.put("hibernate.hbm2ddl.auto", "create");
map.put("hibernate.show", "true");
map.put("javax.persistence.jdbc.url","jdbc:hsqldb:mem:jpa21:persistence-unit");

EntityManagerFactory factory= Persistence.createEntityManagerFactory("jpa21.persistence-unit",map)
```

Entity Manager : Creación

La creación de Entity Managers se realiza mediante la factoría de

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("jpa21:persistence-unit");

EntityManager entityManager = factory.createEntityManager();
```

Ejemplos

Creación Standalone (SE) Persistence Unit

Creación EE de Persistence Units

Mapping

Describiremos los metadatos que definen las entidades, su identidad, atributos y relaciones.

Entidad

@Entity

Determina que la clase es una entidad.

@Table

Asigna el nombre de la tabla para la entidad.

@SecondaryTable

Nos permite asociar tablas a una entidad.

@Embeddable

Permite a una clase ser embebida en otra.

@Embedded

Atributos

@Column

@Embedded

@Transient

@Version

@Enumerated

@Temporal

@Basic

@GeneratedValue

Identidad y Secuencias

@Id

@IdClass

@EmbeddedId

@GeneratedValue

recomendación

Relaciones

@OneToOne

@ManyToOne

@OneToMany

@AssociationOverride

@ElementCollection

@CollectionTable

@JoinColumn / @JoinColumns

@JoinTable

@PrimaryKeyJoinColumn

@MapKeyJoinColumn / @MapKeyJoinColumns

@ForeignKey

Ciclo de vida

@PostLoad

@PrePersist

@PostPersist

@PreUpdate

@PostUpdate

@PreRemove

@PostRemove

@EntityListeners

Herencia

@MappedSuperclass

@Inheritance

SINGLE_TABLE, TABLE_PER_CLASS, JOINED

Sobreescritura de atributos del modelo

@AttributeOverride

@AttributeOverrides

@AssociationOverride

@AssociationOverrides

Equals & Hashcode

Porque
Casos de uso

Es nuestra responsabilidad definir la igualdad entre objetos.

Para que?

Los objeto Java heredan los métodos *equals* y *hashCode* , sin embargo, son útiles sólo para los objetos de valor, por ser de ninguna utilidad para el comportamiento orientado a objetos sin estado .

Al comparar las referencias utilizando el operador "== ", determinar la igualdad de este tipo de objetos es un poco más complicado.

Clave e igualdad de Negocio

La entidades, requieren de un mecanismo de igualación o comparación orientada al negocio que representa y para ello se requiere que se defina lo que es único e igual dentro del contexto del negocio.

Estas funcionalidades son requisito cuando:

- se suman entidades a las colecciones.
- se re adjuntan entidades a un nuevo contexto.

Las claves de negocio, son ajenas a la tecnología.

Entity Manager

Flush Mode
Transactional
Merge vs Persist

Comportamiento y optimizaciones de la instancia del EntityManager.

FlushMode

La especificación JPA define 2 tipos de Flush Mode

`FlushModeType.AUTO` (default)

> En caso de ejecutarse una consulta dentro de una transacción, se ejecutará el flush de manera automática.

`FlushModeType.COMMIT`

> Solo se sincronizará al realizarse el commit.

Recuerde que fuera de una transacción, la implementación no ejecutará ningún flush().

HibernateFlushOperationsTest

Flush con Hibernate

Según la documentación de de hibernate, las operaciones SQL se agrupan siguiendo este orden.

- inserciones
- actualizaciones
- eliminación de colecciones.
- inserción de colecciones
- eliminaciones

HibernateFlushOperationsTest

LockType

to-do

```
Employee employee = em.find(Employee.class, 1, LockModeType.PESSIMISTIC_WRITE);

em.lock(employee , LockModeType.PESSIMISTIC_WRITE);

em.refresh(employee , LockModeType.PESSIMISTIC_WRITE);
```

LockTypeTest

LockType Optimista

Bloqueo optimista

Es completamente automático y activado por defecto en JPA, sin tener en cuenta si un campo de versión (que es requerido por algunos proveedores de ORM APP) se define en la clase de entidad o no.

`LockModeType.OPTIMISTIC_FORCE_INCREMENT / LockModeType.WRITE`

to-do

`LockModeType.OPTIMISTIC / LockModeType.READ`

to-do

LockTypeTest

LockType Pesimista

Bloqueo pesimista

to-do

`LockModeType.PESSIMISTIC_FORCE_INCREMENT`

to-do

`LockModeType.PESSIMISTIC_READ`

to-do

`LockModeType.PESSIMISTIC_WRITE`

to-do

LockTypeTest

Entity Listeners

JPQL

Sentencias JPQL

Criteria API

Construcción de consultas mediante objetos.

Java Persistence Criteria API se utiliza para definir consultas a través de construir las mismas mediante objetos, por sobre el uso de las consultas basadas en literales como JPQL

Criteria API

El API JPA Criteria es utilizado para definir dinámicamente consultas orientadas a objetos, por sobre las orientadas a cadena de caracteres como es JPQL.

La misma posee dos modos de uso, la *type-restrcited* o type-safe y la *non-typed*. La primera utiliza un conjunto de clases o metamodelos generados o construidos que definen los atributos de las consulta. Las *non-typed*, utilizan cadenas de caracteres para referenciar atributos de la clase.

Criteria Builder

Criteria Builder

CriteriaBuilder es la interfaz principal dentro del API Criteria.

Es obtenido mediante EntityManager o EntityManagerFactory usando el método getCriteriaBuilder()

CriteriaBuilder es utilizado como constructor de objetos CriteriaQuery y expresiones

Criteria Builder

CriteriaBuilder define un API para crear objetos *CriteriaQuery*:

- *createQuery() -* Crea una instancia de *CriteriaQuery*.
- *createQuery(Class) -* Crea una instancia de *CriteriaQuery* utilizando *generics* evitando el *cast* del resultado.
- *createTupleQuery() -* Crea una instancia de *CriteriaQuery* que devuelve objetos *Tuple* por sobre Object[].
- *createCriteriaDelete(Class) -* Crea una instancia de *CriteriaDelete* para eliminar por lotes objetos sobre la base de datos (JPA 2.1).
- *createCriteriaUpdate(Class) -* Crea una instancia de *CriteriaUpdate* para actualizar por lotes objetos sobre la base de datos (JPA 2.1).

CriteriaBuilder también define todas las comparaciones y funciones usadas en las consultas.

Where

Se utiliza para definir las condiciones (predicados) que filtran los resultados de las consultas.

Los predicados como *isNotNull*, *isNull*, *like*, se construyen mediante *CriteriaBuilder*.

Comparaciones (I)

equal, notEqual	igual a, no igual	`criteriaBuilder.equal(employee.get("firstName"), "Bob")`
lessThan, lt	menor que	`criteriaBuilder.lessThan(employee.get("salary"), 100000)`
greaterThan, gt	mayor que	`criteriaBuilder.greaterThan(employee.get("salary"), criteriaBuilder.parameter(Integer.class, "sal"))`
lessThanOrEqualTo le	menor o igual a	`criteriaBuilder.lessThanOrEqualTo(employee.get("salary"), 100000)`
reaterThanOrEqual To,ge	mayor o igual a	`criteriaBuilder.greaterThanOrEqualTo(employee.get("salary"), criteriaBuilder.parameter(v.class, "sal"))`

Comparaciones (II)

like, notLike	evalúa el match de dos cadenas de caracteres, '%' y '_' son metacaracteres válidos así como el escapado de los mismos.	`criteriaBuilder.like(employee.get("firstName"), "A%")`
		`criteriaBuilder.notLike(employee.get("firstName"), "%._%", '.')`
between	evalúa un valor entre otros 2.	`criteriaBuilder.between(employee.<String>get("firstName"), "A", "C")`
isNull	compara el valor con nulo	`criteriaBuilder.isNull(employee.get("endDate"))`
		`criteriaBuilder.get("endDate").isNull()`

Comparaciones (III)

in	evalua si el valor esta contenido en la lista	`criteriaBuilder.in(employee.get("firstName")).value("Bob").value("Fred").value("Joe")`
		`employee.get("firstName").in("Bob", "Fred", "Joe")`
		`employee.get("firstName").in(criteriaBuilder.parameter(List.class, "names")`

Operaciones Lógicas (I)

and	*and* dos o más predicados juntos	`criteriaBuilder.and(criteriaBuilder.equal(employee.get("firstName"), "Bob"), criteriaBuilder.equal(employee.get("lastName"), "Smith"))`
or	*or* os o más predicados juntos	`criteriaBuilder.or(criteriaBuilder.equal(employee.get("firstName"), "Bob"), criteriaBuilder.equal(employee.get("firstName"), "Bobby"))`
not	negamos el predicado	`criteriaBuilder.not(criteriaBuilder.or(criteriaBuilder.equal(employee.get("firstName"), "Bob"), criteriaBuilder.equal(employee.get("firstName"), "Bobby")))`
		`criteriaBuilder.or(criteriaBuilder.equal(employee.get("firstName"), "Bob"), criteriaBuilder.equal(employee.get("firstName"), "Bobby")).not()`

Operaciones Lógicas (II)

conjunction	predicado para true	`Predicate where = criteriaBuilder.conjunction();`
		`if (name != null) {`
		`  where = criteriaBuilder.and(where, criteriaBuilder.equal(employee.get(Employee_.firstName, name));`
		`}`

disjunction	predicado para false	`Predicate where = criteriaBuilder.disjunction();`
		`if (name != null) {`
		`  where = criteriaBuilder.or(where, criteriaBuilder.equal(employee.get(Employee_.firstName), name));`
		`}`

Funciones (I)

diff	resta	`criteriaBuilder.diff(employee.<Number>get(Employee_.salary), 1000)`
sum	suma	`criteriaBuilder.sum(employee.<Number>get(Employee_.salary), 1000)`
prod	multiplaca	`criteriaBuilder.prod(employee.<Number>get(Employee_.salary), 2)`
quot	divide	`criteriaBuilder.quot(employee.<Number>get(Employee_.salary), 2)`
abs	valor absoluto	`criteriaBuilder.abs(criteriaBuilder.diff(employee.<Number>get(Employee_.salary), employee.get(Employee_.manager).<Number>get(Employee_.salary))`

Funciones (II)

selectCase	crea una sentencia “select case”	`criteriaBuilder.selectCase(employee.get(Employee_.status)).`
		`  when(0, Employee_.active).`
		`  when(1, Employee_.consultant).`
		`  otherwise("unknown")`

		`criteriaBuilder.selectCase().`
		`  when(criteriaBuilder.equal(employee.get(Employee_.status), 0), "active").`
		`  when(criteriaBuilder.equal(employee.get(Employee_.status), 1), "consultant").`
		`  otherwise("unknown")`

coalesce	evalúa el primer argumento no nulo.	`criteriaBuilder.coalesce(criteriaBuilder.concat(employee.<Number>get(Employee_.salary), 0)`

Funciones (III)

concat	concatena dos o más strings.	`criteriaBuilder.concat( criteriaBuilder.concat(employee.<String>get ("firstName"), " "), employee.<String>get("lastName"))`
currentDate	actual fecha de la base de datos	`criteriaBuilder.currentDate()`
currentTime	actual hora de la base de datos	`criteriaBuilder.currentTime()`
currentTimestamp	actual timestamp de la base de datos	`criteriaBuilder.currentTimestamp()`
length	largo de la cadena de caracteres o valor binario	`criteriaBuilder.length(employee.<String>get("lastName"))`
locate	the index of the string within the string, optionally starting at a start index	`criteriaBuilder.locate("-", employee.<String>get ("lastName"))`

Funciones (IV)

lower	convierte el valor del string a minúsculas	`criteriaBuilder.lower(employee.<String>get("lastName"))`
mod	computes the remainder of dividing the first integer by the second	`criteriaBuilder.mod(employee.<Integer>get("hoursWorked"), 8)`
nullif	returns null if the first argument to equal to the second argument, otherwise returns the first argument	`criteriaBuilder.nullif(employee.<Number>get("salary"), 0)`
sqrt	calcula la raiz cuadrada del numero	`criteriaBuilder.sqrt(employee.<Number>get("salary"))`
substring	the substring from the string, starting at the index, optionally with the substring size	`criteriaBuilder.substring(employee.<String>get("lastName"), 0, 2)`

Funciones (V)

substring	the substring from the string, starting at the index, optionally with the substring size	`criteriaBuilder.substring(employee.<String>get("lastName"), 0, 2)`
trim	trims leading, trailing, or both spaces or optional trim character from the string	`criteriaBuilder.trim(TrimSpec.TRAILING, employee.<String>get("lastName"))`
		`criteriaBuilder.trim(employee.<String>get("lastName"))`
		`criteriaBuilder.trim(TrimSpec.LEADING, '-', employee.<String>get("lastName"))`

Operaciones Especiales (I)

`index`		
`key, value`		
`size`		
`isEmpty,` `isNotEmpty`		
`isMember,` `isNotMember`		
`type`		
`as`		
`function`		

Consultas Tuple

A Tuple se utiliza para resolver las consultas multi selección.

Normalmente, JPA retorna un array de objetos en las consultas multi-selección, el cual no es muy útil.

Una Tuple es una estructura del tipo mapa, donde los resultados pueden ser accedidos mediante su nombre o posición.

Attribute Converter

JPA 2.1 trae varias mejoras. Uno de ellos es el Atributo Converter. Permite al desarrollador especificar métodos para convertir entre la base de datos y la representación de Java de un atributo.

Que podemos convertir

El *Attribute Converter* soporta la conversión de tipo de todos los atributos básicos definidos por las clases de entidad, superclases asignadas, o clases integrables. Las únicas excepciones son los atributos de identificación, los atributos de la versión, los atributos de relación y atributos anotado como temporal o enumerado.

Cómo implementar un convertidor?

Un convertidor debe implementar la interfaz *javax.persistence.AttributeConverter<X,Y>*, donde X es la clase de la representación entidad y Y la clase de la representación base de datos del atributo. Además, un convertidor tiene que ser anotado con la anotación javax.persistence.Converter.

Store Procedures

Desde JPA 2.1 podemos crear y mapear procedimientos almacenados sin utilizar llamadas nativas.

Procedimientos almacenados en JPA 2.1

JPA 2.1 incorpora las llamadas a procedimientos almacenados de base de datos mediante *StoredProcedureQuery*, la anotación *@NamedStoredProcedureQuery* y la etiqueta XML *<named-stored-procedure-query>*.

JPA nos permite hacer las llamadas de los procedimientos definidos como metadatos mediante *EntityManager.createNamedStoredProcedureQuery()*, y crear llamadas de manera dinámica mediante *EntityManager.createStoredProcedureQuery()*.

StoredProcedureQuery es una Consulta JPA que provee de configuración para los parámetros de entrada y salida. Al igual que las consultas nativas, *StoredProcedureQuery* puede devolvernos entidades (objetos) o datos nativos. Para ello podemos utilizar ResultSetMapping para mapear los columnas en entidades.

```
Query query = session.createSQLQuery(
    "CALL GetStocks(:stockCode)")
    .addEntity(Stock.class)
    .setParameter("stockCode", "7277");
```

Parámetros de la llamada

Existen 4 tipos de parámetros para la ejecución de los Procedimientos almacenados de base de datos:

IN: para los parámetros de entrada a la llamada

OUT: parámetro de salida de la llamada.output parameters,

INOUT: se definen los parámetros de entrada y salida d la llamada

REF_CURSOR: nos retorna un cursor del resultado .

Named Store Procedure Query

Store Procedure Query

Criteria API Bulk Operations

CriteriaUpdate / CriteriaDelete

La actualización y eliminación masiva en el API Criteria, se encuentran entre las nuevas mejoras de la especificación.

CriteriaUpdate

La interfaz *CriteriaUpdate* nos permite realizar operaciones de actualización por “bulto” o de manera masiva.

Esta opción nos permite utilizar el API *Criteria* de JPA para la actualización de entidades, evitando así la construcción de *NamedQueries* o *nativas* complejas.

CriteriaUpdate

```
CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
CriteriaUpdate<Price> criteriaUpdate = criteriaBuilder.createCriteriaUpdate(Price.class);
criteriaUpdate.from(Price.class);

Root<Price> root = criteriaUpdate.getRoot();
Path<Number> path = root.get("value");
Expression<Number> value = root.get("value");
Expression<Number> quot = criteriaBuilder.quot(value, factor);
criteriaUpdate.set(path, criteriaBuilder.sum(value, quot));

criteriaUpdate.where(entityManager.getCriteriaBuilder().equal(root.get("country"), country));
return entityManager.createQuery(criteriaUpdate).executeUpdate();
```

CriteriaDelete

La interfaz *CriteriaDelete* nos permite realizar operaciones de *eliminación* por “bulto” o de manera masiva.

Esta opción nos permite utilizar el API *Criteria* de JPA para la actualización de entidades, evitando así la construcción de *NamedQueries* o *nativas* complejas.

CriteriaDelete

```
CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
CriteriaDelete<Price> criteriaUpdate = criteriaBuilder.createCriteriaDelete(Price.class);
criteriaUpdate.from(Price.class);

Root<Price> root = criteriaUpdate.getRoot();

criteriaUpdate.where(entityManager.getCriteriaBuilder().equal(root.get("country"), country));
return entityManager.createQuery(criteriaUpdate).executeUpdate();
```

Constructor Mapping Result

Mapeo Value Objects

JPA 2.1 introduce la anotación @ConstructorResult que nos permite construir Value Objects no mapeados como entidades en el resultado de las @SqlResultSetMapping.

Porque?

Muy a menudo JPQL es insuficiente para la construcción de consultas óptimas, para esos casos, JPA nos permite usar consultas nativas.

El único problema es que estas consultas devuelven una lista de objetos *List<Object[]>* y no entidades mapeadas.

Para ello, JPA 2.1 nos permite mapear estos resultados contra *Value Objects*.

Queries Nativas

```
@NamedNativeQueries({
	@NamedNativeQuery(name = "BookAuthor", resultSetMapping = "BookAuthorMapping",
		query = "SELECT b.id, b.title, b.author_id, b.version, a.id as authorId, a.firstName, a.lastName, a.version as
	authorVersion " +"FROM Book b JOIN Author a ON b.author_id = a.id"),
	@NamedNativeQuery(name = "AuthorBookCount",
		query = "SELECT a.id, a.firstName, a.lastName, a.version, count(b.id) as bookCount " +"FROM Book b JOIN
	Author a ON b.author_id = a.id GROUP BY a.id, a.firstName, a.lastName, a.version"),
	@NamedNativeQuery(name = "AuthorBookCountXml", resultSetMapping = "AuthorBookCountMappingXml",
		query = "SELECT a.id, a.firstName, a.lastName, a.version, count(b.id) as bookCount " +"FROM Book b JOIN
	Author a ON b.author_id = a.id GROUP BY a.id, a.firstName, a.lastName, a.version"),
	@NamedNativeQuery(name = "TotalBookXml", resultSetMapping = "TotalBookMappingXml",
		query = "SELECT a.id, a.firstName, a.lastName, a.version, count(b.id) as bookCount " +"FROM Book b JOIN
	Author a ON b.author_id = a.id GROUP BY a.id, a.firstName, a.lastName, a.version"),
	@NamedNativeQuery(name = "TotalBook", resultSetMapping = "TotalBookMapping",
		query = "SELECT a.id, a.firstName, a.lastName, a.version, count(b.id) as bookCount FROM Book b JOIN Author a
	ON b.author_id = a.id GROUP BY a.id, a.firstName, a.lastName, a.version")})
```

Sql Result Mapping

```
@SqlResultSetMappings({
        @SqlResultSetMapping(name = "BookAuthorMapping", entities = {
                @EntityResult(entityClass = Book.class, fields = {
                        @FieldResult(name = "id", column = "id"),
                        @FieldResult(name = "title", column = "title"),
                        @FieldResult(name = "author", column = "author_id"),
                        @FieldResult(name = "version", column = "version")}),
                @EntityResult(entityClass = Author.class,  fields = {
                        @FieldResult(name = "id", column = "authorId"),
                        @FieldResult(name = "firstName", column = "firstName"),
                        @FieldResult(name = "lastName", column = "lastName"),
                        @FieldResult(name = "version", column = "authorVersion")})}),
        @SqlResultSetMapping(name = "TotalBookMapping",  classes =
                @ConstructorResult(targetClass = TotalBook.class, columns = {
                        @ColumnResult(name = "id", type = Long.class),
                        @ColumnResult(name = "firstName"),
                        @ColumnResult(name = "lastName"),
                        @ColumnResult(name = "bookCount", type = Long.class)}))
})
```

Antes

```
List<Object[]> results = this.em.createNativeQuery("SELECT b.id, b.title, b.author_id, b.version, a.id as authorId, a.firstName, a.lastName, a.version as authorVersion  FROM Book b JOIN Author a ON b.author_id = a.id").getResultList();

results.stream().forEach((record) -> {
        System.out.println("Author: ID [" + record[3] + "] firstName [" + record[4] + "] lastName [" + record[5] + "] Book: ID [" + record[0] + "] Title [" + record[1] + "]");
});
```

Ahora

```
List<Object[]> results = this.em.createNamedQuery("AuthorBookCountXml").getResultList();
results.stream().forEach((record) -> {{
        Author author = (Author) record[0];
        Long bookCount = (Long) record[1];
        System.out.println("Author: ID [" + author.getId() + "] firstName [" + author.getFirstName() + "] lastName [" + author.getLastName() + "] number of books [" + bookCount + "]");
});

List<TotalBook> results = this.em.createNamedQuery("TotalBook").getResultList();
results.stream().forEach((record) -> {{
        System.out.println("Author: ID [" + record.getId() + "] firstName [" + record.getFirstName() + "] lastName [" + record.getLastName() + "] number of books [" + record.getTotal() + "]");
});
```

Unsynchronized Persistence Context

Disabling automatic join of transactions in Stateful services

La nueva versión JPA 2.1 nos brinda un mecanismo de sincronización entre diferentes contextos.

Ejemplo

```
@Stateful
public class ShoppingCart {
	@PersistenceContext(type =EXTENDED, synchronization =UNSYNCHRONIZED)
	EntityManager em;
	Customer customer;
	Order order;

	public void startToShop(Integer custId) {
		customer = em.find(Customer.class,custId);
		order = new Order();
	}
	public void addToCart(Book book) {
		Item item = new Item(book);
		order.addItem(item);
	}
	public void confirmOrder() {
		em.joinTransaction();
		customer.addOrder(order);
	}
}
```

SynchronizationType.UNSYNCHRONIZED

Named Entity Graph

Resolving LazyLoadingExcetpion.
Performance enhance

Optimizando el Lazy Loading

To-Do

Lazy Loading

FetchType es un atributo de de relación JPA, donde se puede definir EAGER en el caso que se necesite cargar la relación, o el valor LAZY en el caso que no la cual siempre es ventajosa en performance, sin embargo debemos tener en cuenta que a esta relación no se puede acceder si no se encuentra dentro del contexto transaccional.

LazyLoadingException, es la excepción que lanza Hibernate cuando se quiere acceder fuera del contexto o con la sesión cerrada.

Entity Graph

El modelado mediante *Fetch Type* tiene sus inconvenientes ya que no son dinámicos. En estos casos es altamente recomendable el uso del nuevo desarrollo *EnittyGraph*

Generación de Esquemas

Programmatic Named Queries

Soporte CDI en Entity Listeners

Mejoras JPQL

No SQL

Hibernate OGM

http://blog.eisele.net/2015/01/nosql-with-hibernate-ogm-part-one.html

To-Do