Hibernate

Motivación de la charla

- ¿ Por qué esta charla ?
  - Diferentes audiencias, mismos problemas.

Introducción

- El acceso a base de datos aparece prácticamente en todos los proyectos, por pequeños que sean.
- El acceso a base de datos es mecánico y repetitivo.
- ¿Por qué no utilizar un sistema que nos facilite esta tarea?
- Hibernate puede ser la solución
  http://www.hibernate.org/
- Otras alternativas: EJB 3.0, JDO
  - Wikipedia: object relational mapping
  - http://www.javaskyline.com/database.html

Comprendiendo los principios del O/R Mapping

- Las principales diferencias entre el mundo relacional y el mundo orientado a objetos:

ORIENTACIÓN A OBJETOS	BASES DE DATOS RELACIONALES
Clases y Objetos	Tablas y Registros
Atributos	Columnas
Indentidad	Clave Primaria
Relación o referencia a objetos	Clave Extranjera
Herencia / Polimorfismo	No Soportado
Métodos	Procedimientos almacenados
Código portable	No necesariamente (depende del fabricante)

Comprendiendo los principios del O/R Mapping

- El O/R Mapping consiste en representar cada tabla de la base de datos como una clase o un conjunto de clases.
- Gracias al O/R Mapping podemos seguir trabajando con Clases Java sin preocuparnos de la base de datos subyacente.
  - Nos podemos centrar en la lógica de negocio
  - Aseguramos la separación entre capas no pasando objetos propios de una capa a la capa contigua (Connection, Statement, ResultSet, ...).
- Cuando entre dos tablas existe integridad referencial, tendremos un clase que hace referencia a otra.

Diseñando el modelo

- Gracias a Hibernate (o cualquier otro O/R mapping similar) nos centraremos en nuestro modelo de clases, y NO en el modelo de tablas.
- De esta manera nuestro negocio no queda condicionado por la capa de persistencia.
- O mínimamente condicionado. Recomendaciones de Hibernate:
  - Cada entidad debe tener un identificador.
  - El identificador debe ser “nulable” (no un tipo básico).
  - Cuidado con los métodos `equals()` y `hashCode()`.

¿ Cómo definir el modelo ?

- Para convertir nuestras clases de negocio en entidades persistentes tenemos dos opciones:
  - Uso de anotaciones
    - Muy recomendable su uso, por su sencillez.
    - Apuesta por la idea de: "**Convención frente a configuración**".
    - Requiere Java 5 (soporte de anotaciones).
    - Son las mismas anotaciones que JPA (Java Persistence API).
  - Uso de descriptores en ficheros XML
    - Ficheros XML externos donde se especifica todos los mapeos entre la clase y la tabla correspondiente.
    - Se necesita más tiempo para desarrollar (puesto que tenemos que escribir dos ficheros en vez de uno)
    - Mas complejo de mantener, y hay mas información redundante.

Declarar entidades

- Nuestras entidades no son más que un conjunto de atributos privados con sus correspondientes “getters” y “setters” (también llamado POJO – Plain Old Java Object).
  - Definir la entidad:
    - @Entity
  - Definir la tabla de una entidad:
    - @Table(name=”Tabla”)
    - Si no se especifica, se usa el nombre de la entidad.

http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=hibernateAnnotations

http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html

Mapear entidades

- ¿ Donde mapear ?
  - Atributos: field access
  - Métodos: property access.
- Debemos seleccionar uno de los dos mecanismos.
- Hibernate usará uno u otro en función de la posición de las anotaciones @Id o @EmbeddedId
- No debemos mezclar.
- Se puede modificar con la anotación
  - @AccessType (Hibernate)

Claves primarias

- Hibernate recomienda que cada entidad tenga al menos un atributo que la identifique unívocamente.
  - Para especificar la clave primaria: @Id
- Generación de las claves primarias:
  - @GeneratedValue(strategy = GenerationType.AUTO)
    - Se selecciona automáticamente la forma de generar las claves primarias en función de la base de datos que estemos usando.
  - @GeneratedValue(strategy = GenerationType.IDENTITY)
  - @GeneratedValue(strategy = GenerationType.SEQUENCE)
    - Claves autogeneradas por una secuencia.
    - Con @SequenceGenerator especificamos los datos de la secuencia.
  - @GeneratedValue(strategy = GenerationType.TABLE)
    - Claves autogeneradas por una tabla.
    - Con @TableGenerator especificamos los datos de la tabla.

Claves primarias compuestas

- Dos técnicas:
  - Usando @EmbeddedId
    - @EmbeddedId para marcar el atributo de la clave primaria en la entidad correspondiente
    - Anotar la clase de ese atributo cómo @Embeddable
  - Usar a nivel de clase la anotación @IdClass (.....class) y anotar los atributos en la clase indicada como @Id

Mapeo de atributos

- Tipos Básicos. Tipos primitivos, “wrappers” y valores enumerados.
  - @Basic
  - Valor por defecto.
- Ajuste fino de tipos básicos
  - Valores enumerados: @Enumerated
    - Almacenar el ordinal: @Enumerated(EnumType.ORDINAL) (por defecto)
    - Almacenar el texto: @Enumerated(EnumType.STRING)
  - Objetos largos: @Lob (BLOB y CLOB)
  - Tiempo:
    - @Temporal(TemporalType.DATE) java.sql.Date
    - @Temporal(TemporalType.TIME) java.sql.Time
    - @Temporal(TemporalType.TIMESTAMP) java.sql.Timestamp
  - Atributo no persistente: @Transient

Mapeo de relaciones

- Especificar las relaciones entre entidades suele ser la parte más complicada de cualquier O/R Mapping.
- Con las anotaciones se simplifica bastante el problema.
  - Principales anotaciones (la izquierda se refiere a "this" y la derecha a la otra entidad):
    - @OneToOne
    - @ManyToOne
    - @OneToMany
    - @ManyToMany
  - Para refinar como se refleja la relación de las entidades en la base de datos:
    - @JoinColumn (ManyToOne)
    - @JoinTable (ManyToMany)

Mapeo de relaciones

- Relación one-to-one: Se pueden usar diferentes técnicas
  - Compartiendo PK

```
@Entity
public class Body {
    @Id
    public Long getId() { return id; }

    @OneToOne(cascade = CascadeType.ALL)
    @PrimaryKeyJoinColumn
    public Heart getHeart() {
        return heart;
    }
    ...
}
```

```
@Entity
public class Heart {
    @Id
    public Long getId() { ...}
}
```

  - Usando @Embedded y @Embeddable

Mapeo de relaciones

- Relación one-to-one: Se pueden usar diferentes técnicas (cont.)
  - A través de FK

```
@Entity
public class Customer implements Serializable {
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name="passport_fk")
    public Passport getPassport() {
        ...
    }

@Entity
public class Passport implements Serializable {
    @OneToOne(mappedBy = "passport")
    public Customer getOwner() {
    ...
}
```

  - A través de tabla intermedia.

```
@Entity
public class Customer implements Serializable {
    @OneToOne(cascade = CascadeType.ALL)
    @JoinTable(name = "CustomerPassports",
        joinColumns = @JoinColumn(name="customer_fk"),
        inverseJoinColumns = @JoinColumn(name="passport_fk")
    )
    public Passport getPassport() {
        ...
    }

@Entity
public class Passport implements Serializable {
    @OneToOne(mappedBy = "passport")
    public Customer getOwner() {
    ...
}
```

Mapeo de relaciones

- Relación many-to-one: Dos técnicas:
  - Usando FK.

```
@Entity()
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, Cas
    @JoinColumn(name="COMP_ID")
    public Company getCompany() {
        return company;
    }
    ...
}
```

  - Usando tabla intermedia.

```
@Entity()
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeTyp
    @JoinTable(name="Flight_Company",
        joinColumns = @JoinColumn(name="FLIGHT_ID"),
        inverseJoinColumns = @JoinColumn(name="COMP_ID")
    )
    public Company getCompany() {
        return company;
    }
    ...
}
```

Mapeo de relaciones

- Relación one-to-many.
  - A la hora de mapear este tipo de relaciones es necesario identificar si es bidireccional o no.
  - En relaciones bidireccionales, dos técnicas:
    - FK

```
@Entity
public class Troop {
    @OneToMany(mappedBy="troop")
    public Set<Soldier> getSoldiers() {
    ...
}

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk")
    public Troop getTroop() {
    ...
}
```

```
@Entity
public class Troop {
    @OneToMany
    @JoinColumn(name="troop_fk") //we need to duplicate the physical
    public Set<Soldier> getSoldiers() {
    ...
}

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk", insertable=false, updatable=false)
    public Troop getTroop() {
    ...
}
```

    - Tabla Intermedia (ver anterior)

Mapeo de relaciones

- Relación one-to-many (cont.)

  En relaciones unidireccionales, dos técnicas:

  - Usando FK (no recomendado por Hibernate)

```
@Entity
public class Customer implements Serializable {
    @OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
    @JoinColumn(name="CUST_ID")
    public Set<Ticket> getTickets() {
    ...
}

@Entity
public class Ticket implements Serializable {
    ... //no bidir
}
```

  - Usando Tabla Intermedia.

```
@Entity
public class Trainer {
    @OneToMany
    @JoinTable(
            name="TrainedMonkeys",
            joinColumns = @JoinColumn( name="trainer_id"),
            inverseJoinColumns = @JoinColumn( name="monkey_id")
    )
    public Set<Monkey> getTrainedMonkeys() {
    ...
}

@Entity
public class Monkey {
    ... //no bidir
}
```

Mapeo de las relaciones

- Many-to-Many.
  - Sólo una posibilidad. A través de tabla intermedia.

```
@Entity
public class Employer implements Serializable {
    @ManyToMany(
        targetEntity=org.hibernate.test.metadata.manytomany.Employee.class,
        cascade={CascadeType.PERSIST, CascadeType.MERGE}
    )
    @JoinTable(
        name="EMPLOYER_EMPLOYEE",
        joinColumns=@JoinColumn(name="EMPER_ID"),
        inverseJoinColumns=@JoinColumn(name="EMPEE_ID")
    )
    public Collection getEmployees() {
        return employees;
    }
    ...
}
```

No simetría de las relaciones

- En relaciones bidireccionales (mapeadas en ambos lados) uno de los lados es el encargado de manejar la relación. El otro lado no es más que un “espejo”.
- Esto puede producir errores “inesperados” si no se tiene claro este concepto.

No simetría de las relaciones

- Para evitar problemas podemos programar de manera “defensiva” creando métodos que se encarguen de manejar correctamente la relación (suponemos una relación many-to-many entre Persona y Evento):

```
...
// Clase Persona
...

protected List  getEventos() {
  return eventos;
}
protected void setEventos(List eventos) {
  this.eventos = eventos;
}

publc void sumameAlEvento(Evento evento) {
    this.getEvents.add(evento);
    evento.getPersonas().add(this);
}

public void borrameDelEvento(Evento evento) {
   this.getEvents.remove(evento);
    evento.getPersonas().remove(this);
}
```

Otras consideraciones: Eager o Lazy

- Con Hibernate se puede configurar cuando (lazy) y como (fetch) se recuperan las entidades relacionadas.
- En JPA (Java Persistence API) sólo se refiere a cuando se recuperan los elementos:
  - FetchType.LAZY: recuperación tardía
    - Valor por defecto para:
      - OneToMany
      - ManyToMany
  - FetchType.EAGER: recuperación temprana
    - Valor por defecto para:
      - OneToOne
      - ManyToOne
- `session.get()` hace recuperación temprana.
- `session.load()` hace recuperación tardía

http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=hibernateJoin

Lazy en atributos básicos.

www.autentia.com

- Usando la anotación @Basic(fetch=...LAZY) no se suficiente.
- Es necesario instrumentalizar nuestras clases en tiempo de compilación.

```
<build>
   <plugins>
   <plugin>
       <artifactId>maven-antrun-plugin</artifactId>
       <executions>
         <execution>
           <phase>process-classes</phase>
           <goals>
             <goal>run</goal>
           </goals>
         </execution>
       </executions>
       <configuration>
         <tasks>
           <taskdef name="instrument" classname="org.hibernate.tool.instrument.javassist.InstrumentTask">
             <classpath>
               <path refid="maven.runtime.classpath" />
               <path refid="maven.plugin.classpath" />
             </classpath>
           </taskdef>
           <instrument verbose="false">
             <fileset dir="${project.build.outputDirectory}">
               <include name="**/persistence/entity/**/*.class" />
             </fileset>
           </instrument>
         </tasks>
       </configuration>
     </plugin>
   </plugins>
</build>
```

Otras consideraciones: equals() y hashCode()

- No se debe usar el Id de la entidad.
- Precauciones a tener en cuenta si hay recuperación tardía:
  - Ojo con comparar la clase con `equals()`:
    - `getClass().equals(other.getClass())`
  - Ojo con acceder directamente a los atributos del objeto:
    - Si la carga del objeto se hizo con “session.load()” los atributos pueden no estar inicializados
  - Ojo con comparar listas con `equals()`:
    - `getPhones().equals(other.getPhones())`

http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=hibernateEquals

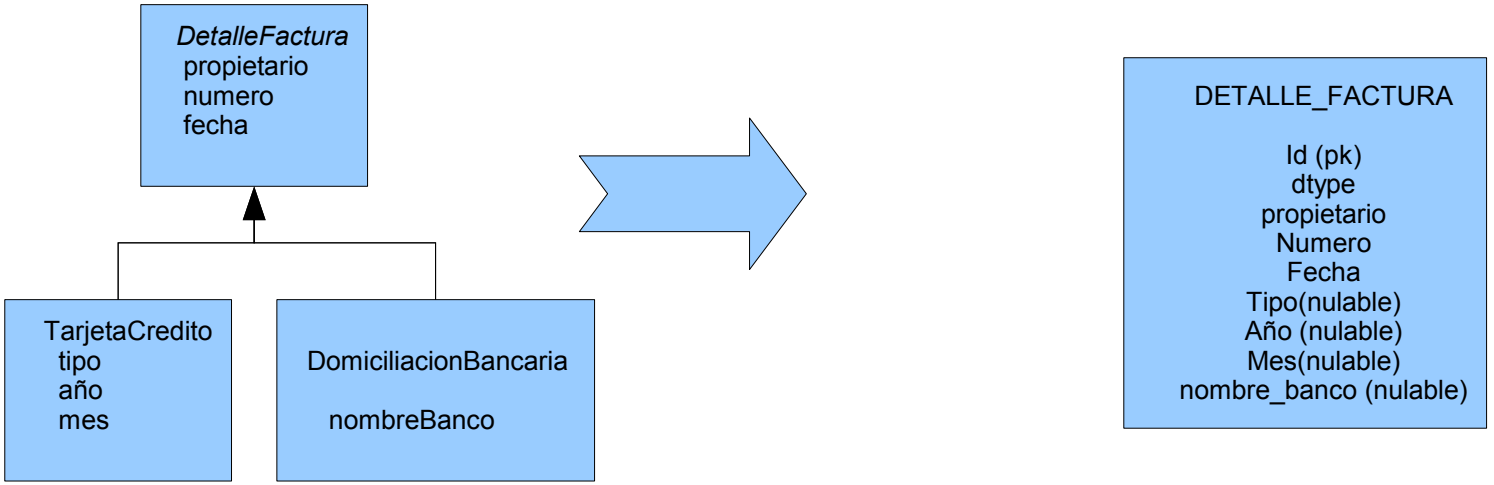
Semántica de las colecciones

- Hibernate no trata todas las colecciones igual:
  - Bag Semantic: List o Collection
  - List Semantic: List + @IndexColumn
  - Set Semantic: Set

http://www.jroller.com/eyallupu/entry/hibernate_exception_simultaneously_fetch_multiple

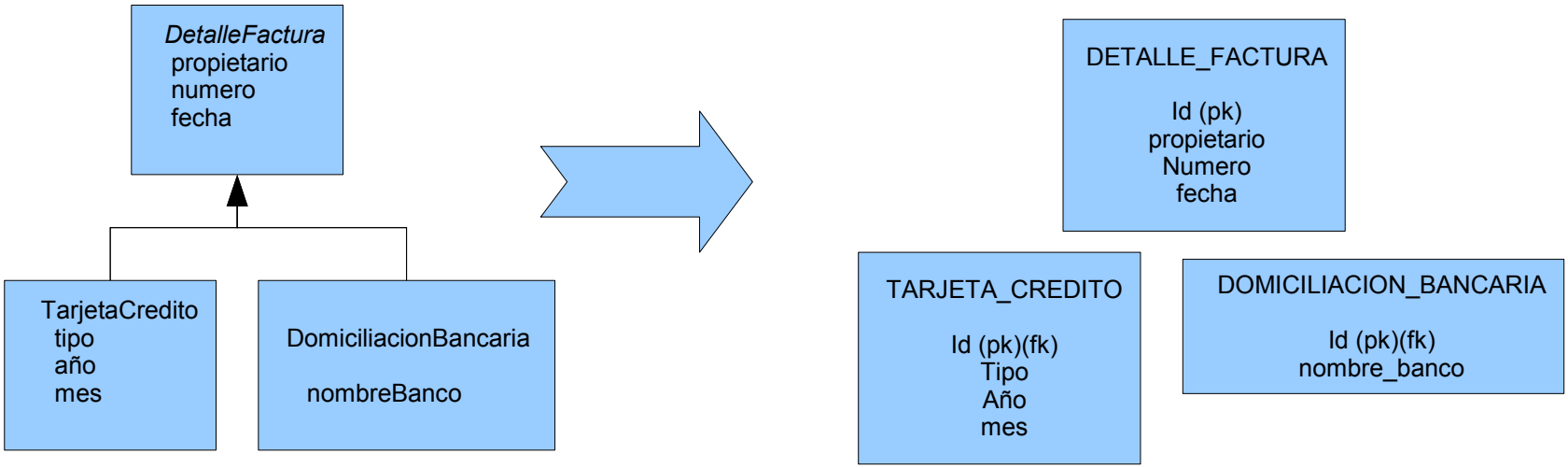
Mapeo de la herencia

- Tres aproximaciones
  - Single table (valor por defecto) (Table per class hierarchy)
    - @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
    - No se anota nada en las clases hijas.
    - Para toda la jerarquía de herencia se usa una sola tabla.
    - No está normalizada, pero es la que da mejor rendimiento.
    - El campo dtype guarda el nombre de la clase sin paquete.



Mapeo de la herencia

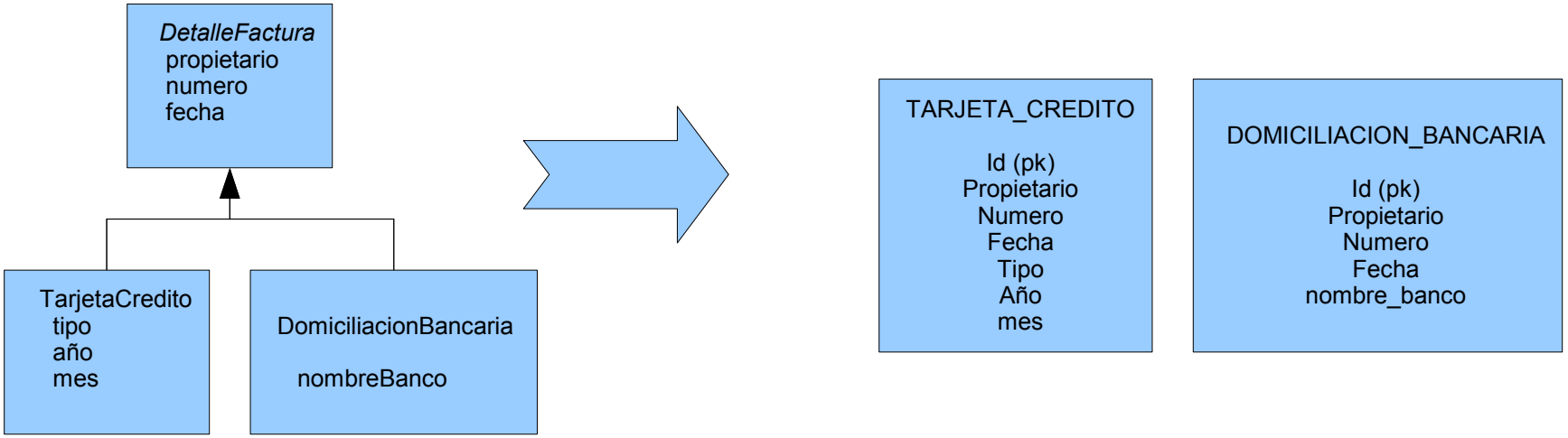
- Joined table (Table per subclass)
  - @Inheritance(strategy=InheritanceType.JOINED)
  - En las clases hijas @PrimaryKeyJoinColumn(name="claveAjena") para indicar el campo que guarda la clave ajena al padre de la jerarquía.
  - Una tabla con los campos comunes (del padre) y una tabla por cada subclase (con exclusivamente los atributos nuevos que aporta).
  - Normalizada pero menor rendimiento.



Mapeo de la herencia

www.autentia.com

- Table per concrete class
  - @Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
  - No se anota nada en las clases hijas.
  - Cada clase tiene su propia tabla con todos los campos (los del padre y los suyos propios).
  - Está en la especificación de EJB 3.0, pero NO es obligatoria su implementación !!!
  - No normalizada y problemas de rendimiento.



Atributos de sólo lectura

- En algunas ocasiones, es necesario mapear atributos pero únicamente en modo lectura:

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumns( { @JoinColumn(name = "id_pais", referencedColumnName = "id_pais"),
        @JoinColumn(name = "id_provincia", referencedColumnName = "id_provincia"),
        @JoinColumn(name = "id_municipio", referencedColumnName = "id_municipio") })
public ComMunicipios getComMunicipios() {
    return this.comMunicipios;
}

public void setComMunicipios(ComMunicipios comMunicipios) {
    this.comMunicipios = comMunicipios;
}
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumns( {
        @JoinColumn(name = "id_pais",
                referencedColumnName = "id_pais", insertable = false, updatable = false),
        @JoinColumn(name = "id_provincia",
                referencedColumnName = "id_provincia", insertable = false, updatable = false) })
public ComProvincias getComProvincias() {
    return this.comProvincias;
}
public void setComProvincias(ComProvincias comProvincias) {
    this.comProvincias = comProvincias;
}
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "id_pais",
        insertable = false,
        updatable = false)
public ComPaises getComPaises() {
    return this.comPaises;
}
```

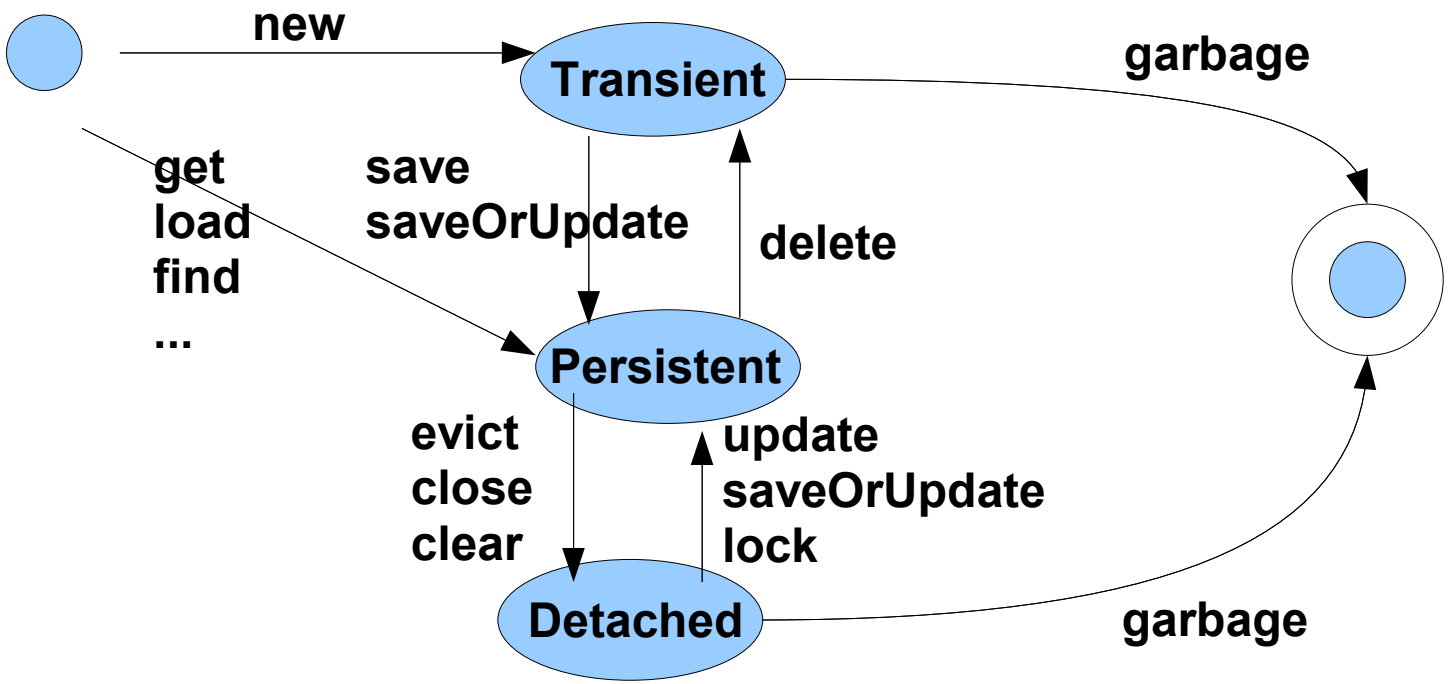
Principios de funcionamiento API

- Prácticamente todo el trabajo lo realizaremos a través de la interfaz **org.hibernate.Session** (podríamos comparar esta sesión con la conexión java.sql.Connection de JDBC).
- La sesión la obtenemos a través de una factoría: **org.hibernate.SessionFactory** (comparable con java.sql.DriverManager).
- **org.hibernate.Query** y **org.hibernate.Criteria** son dos clases que nos permiten, al igual que el **HQL** (Hibernate Query Language), personalizar las consultas a la base de datos (comparable con java.sql.Statement).
- **org.hibernate.Transaction.** Nos permite marcar los ámbitos de una transacción.

Principios de funcionamiento API

- Clase SessionFactory:
  - getCurrentSession()
  - openSession()
- Clase Session:
  - beginTransaction()
  - load() y get()
  - save(), update(), saveOrUpdate()
  - delete()
  - flush()
  - createCriteria()
  - createQuery(), createSQLQuery()
  - getNamedQuery()
  - close()

Principios de funcionamiento
Ciclo de vida



Preparando "el DAO"

- A la hora de implementar el DAO tenemos dos opciones:
  - Hacer una clase DAO por cada POJO:
    - Más fácil de implementar si el acceso a datos lo hacemos nosotros.
    - Implica más clases, más código, y por lo tanto más complejidad y tiempo para desarrollar y mantener.
    - No se pueden aprovechar entre proyectos.
  - Hacer un único DAO para todos los POJOs:
    - Bastante más difícil de hacer si el acceso a datos lo implementamos nosotros.
    - Muy sencillo de implementar si usamos Hibernate.
    - Una única clase para todos los POJOs
    - Se puede reutilizar en muchos proyectos.

Preparando el DAO.

- Además, a la hora de implementar nuestro DAO con hibernate, es muy recomendable apoyarse en las templates de Spring.
- Cómo ejemplo os recomendamos la implementación del DAO del framework wuija de autentia.

Gestión de la Transaccionalidad

- Hibernate permite trabajar con la transaccionalidad independientemente de si el entorno es gestionado (sea JTA o Spring) o no gestionado.
- Para ello, Hibernate define el interfaz Transaction
- Nuestro problema será únicamente de configuración para decirle a hibernate que tipo de Gestor de Transacciones vamos a utilizar: JDBC o JTA

```
<!-- Transaction integration -->
    <attribute name="TransactionStrategy">
        org.hibernate.transaction.JTATransactionFactory</attribute>
    <attribute name="TransactionManagerLookupStrategy">
        org.hibernate.transaction.JBossTransactionManagerLookup</attribute>
```

Gestión de la Transaccionalidad

- Sea el entorno gestionado o no, si vamos a gestionar programáticamente la transaccionalidad (JDBC o Bean BMT) la lógica será siempre la misma:

```
try {
    factory.getCurrentSession().beginTransaction();
    // do some work
    ...
    factory.getCurrentSession().getTransaction().commit();

} catch (RuntimeException e) {
    factory.getCurrentSession().getTransaction().rollback();
    throw e; // or display error message
}
```

- Adicionalmente, una vez se ha producido una excepción, se ha de descartar la sesión.

Gestión declarativa de la Transaccionalidad

- Cuando queremos usar transaccionalidad declarativa:
  - En un entorno con Beans CMT
    - Únicamente será necesario configurar la TransactionStrategy a: org.hibernate.transaction.CMTTransactionFactory
- Si queremos usar Spring para la gestión declarativa de la transaccionalidad:

```
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionM
anager">
<property name="sessionFactory" ref="sessionFactory" />
</bean>
```

Gestión de la sesión

- En el uso de la sesión, no usar los antipatrónes:
  - *session-per-operation*.
  - *session-per-user-session*
  - *session-per-app*lication
    - (no es thread-safe, ante un error hay que descartar la sesión, problemas de memoria)
- Ámbito ideal para el uso de la sesión:
  - *Session-per-request*:
    - ¿ Dónde ? En entorno no JTA.
    - ¿ Por qué ?
      - Aprovecho caché de primer nivel
      - Evito excepciones de lazy inizialization en presentación.
    - Spring lo implementa por tí.

Bloqueo Pesimista / Bloqueo Optimista.

- Por defecto Hibernate trabaja con bloqueo optimista.
- Esto quiere decir que no se hacen bloqueos, pensando que es raro que dos usuarios trabajen exactamente sobre el mismo registro en el mismo instante de tiempo.
- El bloque optimista garantiza la escalabilidad de la aplicación.
- Hibernate proporciona bloqueo optimista comprobando la versión del registro o el timestamp.
  - @Version
  - La propiedad tiene que ser de tipo: int, Integer, short, Short, long, Long, o Timestamp.

Bloqueo Pesimista / Bloqueo Optimista.

- Para el bloqueo pesimista: siempre se usa el bloqueo de la base de datos subyacente.
- La clase `LockMode` define los distintos niveles de bloqueo disponibles en Hibernate:
  - **LockMode.WRITE** se adquiere al insertar o actualizar una fila.
  - **LockMode.UPGRADE** se puede conseguir explícitamente usando SELECT ... FOR UPDATE, en las bases de datos que soportan esta sintaxis.
  - **LockMode.UPGRADE_NOWAIT** se puede conseguir en Oracle usando SELECT ... FOR UPDATE NOWAIT.
  - **LockMode.READ** se adquiere al realizar lecturas sobre un nivel de aislamiento: *Repeatable Read* o *Serializable*. Tambien se puede adquirir por petición expresa del usuario.
  - **LockMode.NONE** todos los objetos se cambian a este modo de bloqueo una vez termina la transacción.
- Para definir el LockMode se hará una llamada a:
  - Session.load()
  - Session.lock()
  - Query.setLockMode()

¿Y cuando el HQL no es suficiente?

- Cuando HQL es insuficiente:
  - Podemos usar queries nativas y ResultTransformer.

```
final SQLQuery query = session.createSQLQuery(queryString);
query.setResultTransformer
       (Transformers.aliasToBean(transformerClass));
```

En procesos batch

- La sesión de hibernate no es infinita, y en procesos batch podemos llegar a comprobarlo de manera penosa.
- Para evitar este problema:

```
Session session = null;
Transaction tx = null;
try {
        session = getSessionFactory().openSession();
        tx = session.beginTransaction();
        session.setFlushMode(FlushMode.MANUAL);
        session.setCacheMode(CacheMode.IGNORE);
        Query query = session.getNamedQuery(XXX);
        ScrollableResults results =
        query.setFetchSize(batchSize).scroll(ScrollMode.FORWARD_ONLY);
        while (results.next()) {
                // DO SOMETHING
                if (index % batchSize == 0) {
                        session.clear();
                }
                index++;
        }
```

¿ Preguntas ?

www.autentia.com