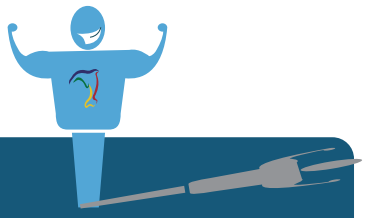
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

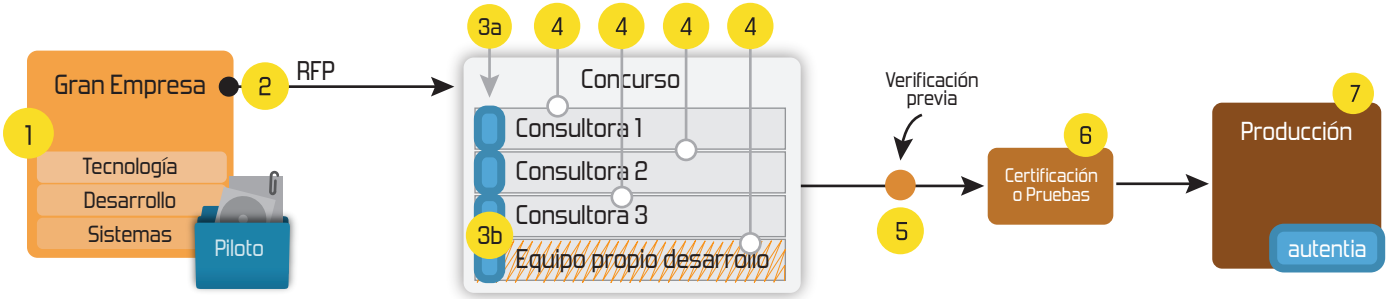
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
Gestor de contenidos (Alfresco)
Aplicaciones híbridas

Control de autenticación y acceso (Spring Security)
UDDI
Web Services
Rest Services
Social SSO
SSO (Cas)

JPA-Hibernate, MyBatis
Motor de búsqueda empresarial (Solr)
ETL (Talend)

Dirección de Proyectos Informáticos.
Metodologías ágiles
Patrones de diseño
TDD

Tareas programadas (Quartz)
Gestor documental (Alfresco)
Inversión de control (Spring)

BPM (jBPM o Bonita)
Generación de informes (JasperReport)
ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

Lanzado **TNTConcept versión 0.4.1** **(04/06/2007)**

Desde Autentia ponemos a vuestra disposición el software que hemos construido (100% gratuito y sin restricciones funcionales) para nuestra gestión interna, llamado TNTConcept (au**T**e**NT**ia).

Construida con las últimas tecnologías de desarrollo Java/J2EE (Spring, JSF, Acegi, Hibernate, Maven, Subversion, etc.) y disponible en licencia GPL, seguro que a muchos profesionales independientes y PYMES os ayudará a organizar mejor vuestra operativa.

Las cosas grandes empiezan siendo algo pequeño ..... Saber más en: http://tntconcept.sourceforge.net/

Fecha de creación del tutorial: 2007-06-27

Hibernate y el mapeo de la herencia

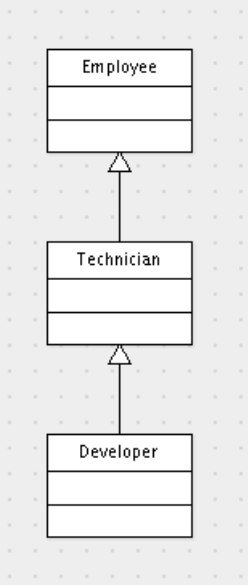
Creación: 20-06-2007

Índice de contenidos

1. Introducción

Siguiendo esta serie sobre Hibernate (http://www.hibernate.org/) vamos a ver en este tutorial como implementar las relaciones de herencia con las anotaciones de JPA.

Las clases que vamos a usar como ejemplo son:



Tanto en Hibernate como en JPA se definen tres estrategias para mapear esta relación de clases a tablas de nuestra base de datos:

- Una sola tabla para guardar toda la jerarquía de clases. Tiene la ventaja de ser la opción que mejor rendimiento da, ya que sólo es necesario acceder a una tabla (está totalmente desnormalizada). Tiene como inconveniente que todos los campos de las clases hijas tienen que admitir nulos, ya que cuando guardemos un tipo, los campos correspondientes a los otros tipos de la jerarquía no tendrán valor.

- Una tabla para el padre de la jerarquía, con las cosas comunes, y otra tabla para cada clase hija con las cosas concretas. Es la opción más normalizada, y por lo tanto la más flexible (puede ser interesante si tenemos un modelo de clases muy cambiante), ya que para añadir nuevos tipos basta con añadir nuevas tablas y si queremos añadir nuevos atributos sólo hay que modificar la tabla correspondiente al tipo donde se está añadiendo el atributo. Tiene la desventaja de que para recuperar la información de una clase, hay que ir haciendo join con las tablas de las clases padre.

- Una tabla independiente para cada tipo. En este caso cada tabla es independiente, pero los atributos del padre (atributos comunes en los hijos), tienen que estar repetidos en cada tabla. En principio puede tener serios problemas de rendimiento, si estamos trabajando con polimorfismo, por los SQL UNIOS que tiene que hacer para recuperar la información. Sería la opción menos aconsejable. Tanto es así que en la versión 3.0 de EJBs, aunque está recogida en la especificación, no es obligatoria su implementación.

Aquí tenéis un tar.gz con todo el código de los ejemplos.

2. Entorno

El tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil Asus G1 (Core 2 Duo a 2.1 GHz, 2048 MB RAM, 120 GB HD).
- Sistema Operativo: GNU / Linux, Debian (unstable), Kernel 2.6.21, KDE 3.5
- Máquina Virtual Java: JDK 1.6.0-b105 de Sun Microsystems
- Eclipse 3.2.2
- Hibernate 3.2.2.ga
- Hibernate Tools 3.2.0 Beta9
- MySql 5.0.41-2
- JUnit 4.3.1
- Maven 2.0.7

3. Una sola tabla para toda la jerarquía de clases

Recordemos que en esta ocasión usaremos una sola tabla para toda la jerarquía de clases. Es, posiblemente, la implementación más sencilla.

Veamos como creamos la tabla:

```
CREATE TABLE  `Employee` (
  `id` bigint unsigned NOT NULL auto_increment,
```

```
  `nif` varchar(10) default NULL,
  `name` varchar(50) NOT NULL,
  `phone` varchar(20) default NULL,
  `email` varchar(50) default NULL,
  `dtype` varchar(31) default NULL,

  `experienceYears` int unsigned default 0,

  `expertLanguajes` varchar(50) default NULL,

  PRIMARY KEY  (`id`)
) engine=innodb DEFAULT CHARSET=utf8 COLLATE=utf8_spanish_ci;
```

Podemos ver como en esta tabla están todos los atributos, tanto de la clase padre como de las derivadas (`experienceYears` es de la clase `Technician` y `expertLanguajes` es de la clase `Developer`). Nótese también como la tabla tiene el campo "**dtype**", este campo será el que se use como discriminador para saber a que tipo concreto se refiere el registro. Por defecto en este campo se guardará el nombre de la clase (sin el nombre del paquete).

Vemos ahora el código de las clases:

```
@Entity
@Table(name = "Employee", catalog = "curso")
//@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Employee implements java.io.Serializable {

    Long id;

    private String nif;

    private String name;

    private String phone;

    private String email;

    public Employee() {
    }

    public Employee(String name) {
        this.name = name;
    }

    public Employee(String nif, String name, String phone, String email) {
        this.nif = nif;
        this.name = name;
        this.phone = phone;
        this.email = email;
    }

    @Id
    @GeneratedValue
    @Column(name = "id", unique = true, nullable = false)
    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Column(name = "nif", length = 10)
    public String getNif() {
        return this.nif;
    }

    public void setNif(String nif) {
        this.nif = nif;
    }

    @Column(name = "name", nullable = false, length = 50)
    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Column(name = "phone", length = 20)
    public String getPhone() {
        return this.phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }

    @Column(name = "email", length = 50)
    public String getEmail() {
        return this.email;
    }

    public void setEmail(String email) {
        this.email = email;
```

```
        }
}
```

Nótese que la línea donde está definido el tipo de mapeo para la herencia (justo encima de la declaración de la clase) está comentada. Es decir, no es necesaria ya que `SINGLE_TABLE` es el valor por defecto. En el ejemplo se a puesto comentada la línea para que el lector vea como sería la definición de la estrategia que se va a utilizar para mapear la herencia de estas entidades.

Veamos las otras clases:

```
@Entity
public class Technician extends Employee {

	private int experienceYears = 0;


	public int getExperienceYears() {
		return experienceYears;
	}

	public void setExperienceYears(int experienceYears) {
		this.experienceYears = experienceYears;
	}
}



@Entity
public class Developer extends Technician {

	private String expertLanguajes = null;

	public String getExpertLanguajes() {
		return expertLanguajes;
	}

	public void setExpertLanguajes(String expertLanguajes) {
		this.expertLanguajes = expertLanguajes;
	}
}
```

Se puede ver como las clases derivadas son extremadamente sencillas, y no especifican ninguna anotación especial. Tampoco ninguna de ellas tiene un atributo que especifique el id (la PK) del objeto. Esto no es necesario ya que esta propiedad se hereda del padre `Employee`.

4. Una tabla para cada subclase ("join" de tablas)

Recordamos que en esta estrategia tendremos una tabla para los atributos comunes (los atributos del padre), y una tabla por cada subclase (con exclusivamente los atributos nuevos que aporta la subclase). Para conseguir un objeto se hará join con las tablas necesarias.

```
CREATE TABLE  `Employee` (
   `id` bigint unsigned NOT NULL auto_increment,
   `nif` varchar(10) default NULL,
   `name` varchar(50) NOT NULL,
   `phone` varchar(20) default NULL,
   `email` varchar(50) default NULL,

   PRIMARY KEY  (`id`),
   constraint fk_bossId foreign key (bossId) references Employee (id)
) engine=innodb DEFAULT CHARSET=utf8 COLLATE=utf8_spanish_ci;

CREATE TABLE  `Technician` (
   `id` bigint unsigned NOT NULL auto_increment,
   `employeeId` bigint unsigned NOT NULL,
   `experienceYears` int unsigned default 0,

   PRIMARY KEY  (`id`),
   constraint fk_employeeId foreign key (employeeId) references Employee (id)
) engine=innodb DEFAULT CHARSET=utf8 COLLATE=utf8_spanish_ci;

CREATE TABLE  `Developer` (
   `id` bigint unsigned NOT NULL auto_increment,
   `employeeId` bigint unsigned NOT NULL,
   `expertLanguajes` varchar(50) default NULL,

   PRIMARY KEY  (`id`),
   constraint fk_technicianId foreign key (employeeId) references Employee (id)
) engine=innodb DEFAULT CHARSET=utf8 COLLATE=utf8_spanish_ci;
```

Cabría destacar como la tabla `Developer`, a pesar de almacenar los datos de la clase `Developer`, que es hija de `Technician`, tiene una clave ajena con `Employee`, que es el padre de la jerarquía.

Veamos como quedan las clases:

```
@Entity
@Table(name = "Employee", catalog = "curso")
@Inheritance(strategy=InheritanceType.JOINED)
```

```
public class Employee implements java.io.Serializable {

    Long id;

    private String nif;

    private String name;

    private String phone;

    private String email;

    public Employee() {
    }

    public Employee(String name) {
        this.name = name;
    }

    public Employee(String nif, String name, String phone, String email) {
        this.nif = nif;
        this.name = name;
        this.phone = phone;
        this.email = email;
    }

    @Id
    @GeneratedValue
    @Column(name = "id", unique = true, nullable = false)
    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Column(name = "nif", length = 10)
    public String getNif() {
        return this.nif;
    }

    public void setNif(String nif) {
        this.nif = nif;
    }

    @Column(name = "name", nullable = false, length = 50)
    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Column(name = "phone", length = 20)
    public String getPhone() {
        return this.phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }

    @Column(name = "email", length = 50)
    public String getEmail() {
        return this.email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

Esta clase, el padre de la jerarquía, es igual que en el ejemplo anterior, salvo que está vez si es obligatorio definir la anotación para indicar la estrategia a seguir para mapear la herencia (`InheritanceType.JOINED`).

Veamos las clases hijas:

```
@Entity
@PrimaryKeyJoinColumn(name="employeeId")
public class Technician extends Employee {

    private int experienceYears = 0;


    public int getExperienceYears() {
        return experienceYears;
    }

    public void setExperienceYears(int experienceYears) {
        this.experienceYears = experienceYears;
    }
}
```

```
@Entity
@PrimaryKeyJoinColumn(name="employeeId")
public class Developer extends Technician {

	private String expertLanguajes = null;

	public String getExpertLanguajes() {
		return expertLanguajes;
	    }

	public void setExpertLanguajes(String expertLanguajes) {
		this.expertLanguajes = expertLanguajes;
	    }
}
```

Comparando con el ejemplo anterior, ambas clases lo único que hacen es indicar el nombre del campo que contiene la clave ajena (la FK) para hacer el join con el padre de la jerarquía. Volvemos a insistir aquí que `Developer` no hace referencia a `Technician`, sino a `Employee`.

5. Una tabla para cada clase ("union" de tablas)

Recordemos que esta estrategia consiste en tener una tabla para cada clase, estando los campos correspondientes a los atributos comunes (los atributos del padre) repetidos en cada una de estas tablas. Recordemos también que, aunque está recogida en la especificación de EJB 3.0, no es obligatoria su implementación. Y que puede tener problemas de rendimiento especialmente si trabajamos con polimorfismo.

Crearemos las tablas de la siguiente manera:

```
--
-- Create all the tables and insert some example data.
--

CREATE TABLE  `IdsGenerator` (
   `id` varchar(10) NOT NULL,
   `employeeIds` bigint unsigned NOT NULL,

   PRIMARY KEY  (`id`)
) engine=innodb DEFAULT CHARSET=utf8 COLLATE=utf8_spanish_ci;

CREATE TABLE  `Employee` (
   `id` bigint unsigned NOT NULL,
   `nif` varchar(10) default NULL,
   `name` varchar(50) NOT NULL,
   `phone` varchar(20) default NULL,
   `email` varchar(50) default NULL,

   PRIMARY KEY  (`id`)
) engine=innodb DEFAULT CHARSET=utf8 COLLATE=utf8_spanish_ci;

CREATE TABLE  `Technician` (
   `id` bigint unsigned NOT NULL,
   `nif` varchar(10) default NULL,
   `name` varchar(50) NOT NULL,
   `phone` varchar(20) default NULL,
   `email` varchar(50) default NULL,
   `experienceYears` int unsigned default 0,

   PRIMARY KEY  (`id`)
) engine=innodb DEFAULT CHARSET=utf8 COLLATE=utf8_spanish_ci;

CREATE TABLE  `Developer` (
   `id` bigint unsigned NOT NULL,
   `nif` varchar(10) default NULL,
   `name` varchar(50) NOT NULL,
   `phone` varchar(20) default NULL,
   `email` varchar(50) default NULL,
   `experienceYears` int unsigned default 0,
   `expertLanguajes` varchar(50) default NULL,

   PRIMARY KEY  (`id`)
) engine=innodb DEFAULT CHARSET=utf8 COLLATE=utf8_spanish_ci;

--
-- Se dan de alta algunos valores iniciales
--
INSERT INTO IdsGenerator VALUES('Employee', 1);
```

Se puede ver como cada tabla hija tiene todas las columnas de su padre, más las suyas propias. También es destacable que los ids (las PK) no son definidos como auto incrementales. Esto es una limitación de esta estrategia: los ids deben ser únicos para las tres tablas, así que no podemos marcar los ids como auto incrementales en cada tabla.

Es recomendable que la base de datos sea la encargada de generar las claves, y no la aplicación. Si queremos que siga siendo la base de datos la encargada de generar los ids, habrá que usar una secuencia (por ejemplo en Oracle), u otra tabla (en nuestro ejemplo la tabla `IdsGenerator`) que se encargue de guardar los ids únicos para las tres tablas.

La clase padre de la jerarquía quedará de la siguiente manera:

```
@Entity
@Table(name = "Employee", catalog = "curso")
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Employee implements java.io.Serializable {

    Long id;

    private String nif;

    private String name;

    private String phone;

    private String email;

    public Employee() {
    }

    public Employee(String name) {
        this.name = name;
    }

    public Employee(String nif, String name, String phone, String email) {
        this.nif = nif;
        this.name = name;
        this.phone = phone;
        this.email = email;
    }

    @Id
    @GeneratedValue(strategy=GenerationType.TABLE, generator="idsGenerator")
    @TableGenerator(name="idsGenerator", table="IdsGenerator",
        pkColumnName="id", pkColumnValue="Employee", valueColumnName="employeeIds")
    @Column(name = "id", unique = true, nullable = false)
    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Column(name = "nif", length = 10)
    public String getNif() {
        return this.nif;
    }

    public void setNif(String nif) {
        this.nif = nif;
    }

    @Column(name = "name", nullable = false, length = 50)
    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Column(name = "phone", length = 20)
    public String getPhone() {
        return this.phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }

    @Column(name = "email", length = 50)
    public String getEmail() {
        return this.email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

Esta vez en la anotación `@Inheritance` hemos definido la estrategia `TABLE_PER_CLASS`. También se puede ver como se esta usando otra tabla de la base de datos para generar los identificadores. Aunque no es necesario para la herencia (podríamos haber asignado los identificadores de forma manual) explicamos brevemente la anotación `@TableGenerator`:

- `name`: nombre único para identificar a ese `TableGenerator`. Se usa en la anotación `@GeneratedValue` para hacer referencia a ese generador de claves.
- `table`: nombre de la tabla donde se van a guardar las claves. Se puede usar una misma tabla para guardar los ids de distintas entidades. Cada entidad tendrá su propio registro dentro de esa tabla.
- `pkColumnName`: nombre de la columna que tiene la PK en la tabla especificada con el atributo "`table`".
- `pkColumnValue`: el valor que identifica ha esta entidad. Es decir, dentro del campo especificado por "`pkColumnName`" se buscará esté valor para determinar el registro que guarda los ids para esta entidad. Gracias a esto es como se consigue tener en una única

tabla los ids para más de una entidad.

Veamos las clases derivadas:

```
@Entity
public class Technician extends Employee {

	private int experienceYears = 0;


	public int getExperienceYears() {
		return experienceYears;
	    }

	public void setExperienceYears(int experienceYears) {
		this.experienceYears = experienceYears;
	    }
}



@Entity
public class Developer extends Technician {

	private String expertLanguajes = null;

	public String getExpertLanguajes() {
		return expertLanguajes;
	    }

	public void setExpertLanguajes(String expertLanguajes) {
		this.expertLanguajes = expertLanguajes;
	    }
}
```

Podemos ver como se mantienen sencillas y no hay que añadir ninguna anotación especial (son iguales que cuando usábamos una única tabla para toda la jerarquía).

6. Ejemplo de uso

A continuación presentamos un ejemplo de uso que vale para cualquiera de las tres estrategias. Con esto se demuestra que podemos cambiar la configuración, pero que la lógica de negocio sigue intacta.

```
@Test
public void addEmployee() {
	log.info("\n\n*** addEmployee ***\n");

	try {
		Employee employee = new Employee("yo mismo");

		session.persist(employee);

		log.debug("La clave del nuevo objeto es: " + employee.getId());


		Technician technician = new Technician();
		technician.setName("Yo soy el técnico !!!");
			technician.setExperienceYears(24);
		session.save(technician);
		log.debug("La clave del nuevo objeto es: " + technician.getId());

		Developer developer = new Developer();
		developer.setName("Yo soy el desarrollador !!!");
			developer.setExperienceYears(14);
		developer.setExpertLanguajes("Java");
		session.save(developer);
		log.debug("La clave del nuevo objeto es: " + developer.getId());

	} catch (Exception e) {
		log.error(e);
	    }
}
```

7. Conclusiones

Como veis no hay ninguna solución que valga para todas las situaciones. Según nuestras condiciones tendremos que escoger la estrategia que más nos interesa.

Podríamos destacar la estrategia "SINGLE_TABLE" como la más óptima en cuenta a rendimiento, y la estrategia "JOINED" como la más flexible. En cuanto a al estrategia "TABLE_PER_CLASS" habrá que tener especial cuidado al elegirla por sus problemas de rendimiento trabajando con polimorfismo y por que nos podemos encontrar con proveedores de JPA que no la implementen.

8. Sobre el autor

Alejandro Pérez García, Ingeniero en Informática (especialidad de Ingeniería del Software)

Socio fundador de Autentia (Formación, Consultoría, Desarrollo de sistemas transaccionales)

mailto:alejandropg@autentia.com

Autentia Real Business Solutions S.L. - "Soporte a Desarrollo"

http://www.autentia.com

Recuerda

que el personal de Autentia te regala la mayoría del conocimiento aquí compartido (Ver todos los tutoriales)

¿Nos vas a tener en cuenta cuando necesites consultoría o formación en tu empresa?

¿Vas a ser tan generoso con nosotros como lo tratamos de ser con vosotros?

info@autentia.com

Somos pocos, somos buenos, estamos motivados y nos gusta lo que hacemos .....
Autentia = Soporte a Desarrollo & Formación

Autentia S.L. Somos expertos en:
J2EE, Struts, JSF, C++, OOP, UML, UP, Patrones de diseño ..
y muchas otras cosas

Nuevo servicio de notificaciones

Si deseas que te enviemos un correo electrónico cuando introduzcamos nuevos tutoriales, inserta tu dirección de correo en el siguiente formulario.

Subscribirse a Novedades	
e-mail	
	Enviar

Otros Tutoriales Recomendados (También ver todos)

Nombre Corto	Descripción
Hibernate Tools y la generación de código	En este tutorial vamos a ver como usar estas herramientas para hacer el esqueleto de una pequeña aplicación, de manera muy sencilla, generando código a partir de las tablas creadas en la base de datos.
Hibernate y las anotaciones de EJB 3.0	En este tutorial Alejandro Pérez nos muestra las ventajas que nos aporta Hibernate y las anotaciones de EJB 3.0
Creación automática de recursos Hibernate con Middlegen	En este tutorial aprendereis como utilizar la herramienta middlegen para generar distintas capas de persistencia (CMP 2.0, JDO, Hibernate, Torque), a partir de un modelo físico de datos, de un modo automático, mediante el uso de la herramienta middlegen
Hibernate 3 y los tipos de datos para cadenas largas	En este tutorial se contará la experiencia que hemos tenido a la hora de manejar los diferentes tipos de datos existentes para grandes cadenas de texto, tales como el tipo Clob de Oracle, o los tipos TEXT de MySQL y SQLServer, utilizando la última versión
Manejar dos bases de datos distintas con Hibernate	Alejandro Pérez nos enseña como manejar dos bases de datos distintas con Hibernate

Introducción a Hibernate	Cesar Crespo nos enseña como utilizar unos de los sistemas más extendidos de mapeo de objetos a estructuras relacionales (tablas de base de datos)
Hibernate y Joins con la clase Criteria	En este tutorial vamos a ver como hacer "joins" entre entidades relacionadas, y las implicaciones que esto puede tener, usando Hibernate
Hibernate 3.1, Colecciones, Fetch y Lazy	En este tutorial vamos a ver cómo se comportan ciertas relaciones, y cómo podemos optimizar las consultas a la base de datos con Hibernate