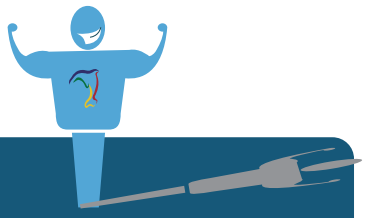
autentia

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...

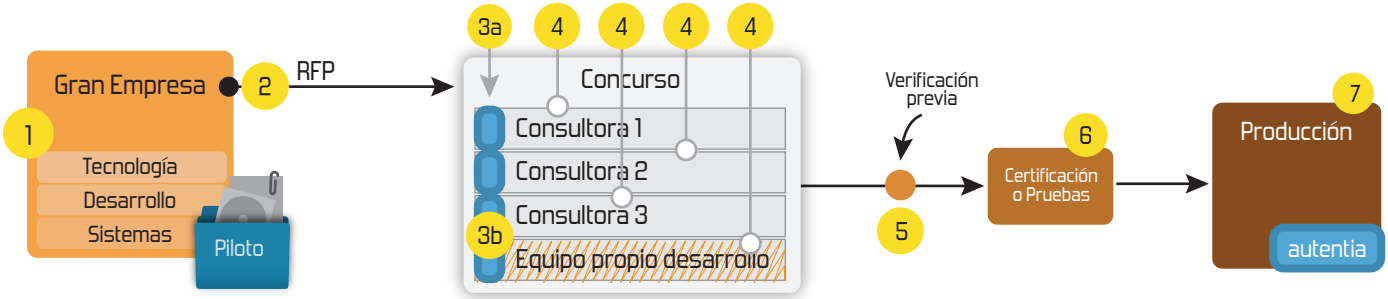
## 1. Desarrollo de componentes y proyectos a medida



## 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**

**Tutorial desarrollado por**

**Raúl Expósito Díaz**

Consultor tecnológico de desarrollo de proyectos informáticos.

Ingeniero Técnico en Informática de Gestión por la Universidad de Alcalá e Ingeniero en Informática por la Universidad Carlos III de Madrid. Perfil XING

Puedes encontrarme en Autentia

Somos expertos en Java/J2EE



Descargar este documento en formato PDF: springHibernateAnotaciones.pdf

**Fecha de creación del tutorial: 2008-02-15**

# Creación de una aplicación con Spring e Hibernate desde 0

## 1. Introducción

Como el título reza, en este tutorial vamos a crear una pequeña aplicación usando Spring e Hibernate partiendo desde 0. Este es un tutorial con un enfoque claramente práctico, con lo cual voy a intentar no entrar en conceptos teóricos ni técnicos. Simplemente crearemos unas entidades, unos DAO, una clase con un método main() que ejecute la aplicación y, como no podia ser de otro modo, juntaremos todas las piezas con Spring haciendo uso de su soporte para Hibernate.

Para no complicar mucho el ejemplo voy a crear la típica aplicación que accede a una base de datos con información sobre empresas y personal, y para ello usaremos Hibernate con anotaciones, no con mapeos en ficheros `.hbm.xml`. La aplicación no tiene apenas funcionalidad y se utilizará por consola, pero servirá para ver cómo podemos integrar Hibernate con anotaciones en un proyecto de Spring. Además veremos que gracias al uso de anotaciones es muy sencillo sustituir Hibernate por cualquier otro motor de persistencia que pueda usarse a través de JPA.

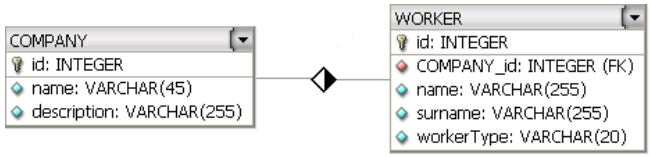
En adictosaltrabajo ya vemos publicado otros tutoriales sobre Spring e Hibernate que seguramente os sirvan de referencia o como complemento a éste. Si quieres descargar el código fuente de este tutorial no tienes más que pinchar aqui.

## 2. Entorno

- Debian GNU/Linux 4.1 (Lenny)
- JDK 6 Update 1
- Eclipse 3.3 (Europa)
- MySQL 5.0.51
- Spring 2.5
- Hibernate 3.2.5

## 3. Modelo de datos

El modelo de datos, como se puede observar, es muy sencillo. Tan solo una relación `N:M` entre `COMPANY` y `WORKER` :






El script de creación para `MySQL` es el siguiente:

```
CREATE DATABASE autentiaSpringHb;
USE autentiaSpringHb;

CREATE TABLE COMPANY (
id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
name VARCHAR(45) NOT NULL,
description VARCHAR(255) NOT NULL,
PRIMARY KEY(id)
) engine=innodb default charset=utf8 collate=utf8_spanish_ci;

CREATE TABLE WORKER (
id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
COMPANY_id INTEGER UNSIGNED NOT NULL,
name VARCHAR(255) NOT NULL,
surname VARCHAR(255) NOT NULL,
workerType VARCHAR(20) NOT NULL,
PRIMARY KEY(id),
CONSTRAINT fk_person_company FOREIGN KEY(COMPANY_id) REFERENCES COMPANY(id)
) engine=innodb default charset=utf8 collate=utf8_spanish_ci;
```

## 4. Entidades

Para definir las entidades vamos a utilizar las anotaciones de Hibernate, y gracias a ellas nos ahorraremos los `.hbm.xml`. Como podreis observar viendo los ejemplos es muy sencillo mapear los atributos de las clases y las propias clases, aparte que hace que sea más fácil mantener las clases ya que no tenemos que mantener, aparte de la clase, un fichero xml.

A continuación pego el código del fichero `Company.java`

view plain print ?

```
01. package com.autentia.springhb.entity;
02.
03. import javax.persistence.Column;
04. import javax.persistence.Entity;
05. import javax.persistence.GeneratedValue;
06. import javax.persistence.GenerationType;
07. import javax.persistence.Id;
08. import javax.persistence.Table;
09.
10. @Entity
11. @Table(name="COMPANY")
12. public class Company extends AbstractEntity {
13.
14.     private static final long serialVersionUID = 14606693617179884691;
15.
16.     private String name;
17.     private String description;
18.
19.     @Id @GeneratedValue(strategy = GenerationType.AUTO)
20.     @Override
21.     public Long getId() {
22.         return id;
23.     }
24.
25.     @Column(length=45, nullable=false)
26.     public String getName() {
27.         return name;
28.     }
29.
30.     public void setName(String name) {
31.         this.name = name;
32.     }
33.
34.     @Column(length=255, nullable=true)
35.     public String getDescription() {
36.         return description;
37.     }
38.
39.     public void setDescription(String notes) {
40.         this.description = notes;
41.     }
42. }
43.
```

Tras ello, el del fichero `Worker.java`

view plain print ?

```
01. package com.autentia.springhb.entity;
02.
03. import javax.persistence.Column;
04. import javax.persistence.Entity;
05. import javax.persistence.EnumType;
06. import javax.persistence.Enumerated;
07. import javax.persistence.GeneratedValue;
08. import javax.persistence.GenerationType;
09. import javax.persistence.Id;
10. import javax.persistence.JoinColumn;
11. import javax.persistence.ManyToOne;
12. import javax.persistence.Table;
13.
14. @Entity
15. @Table(name="WORKER")
16. public class Worker extends AbstractEntity {
17.
18.     private static final long serialVersionUID = 6430623703467941209L;
19.
20.     private WorkerType workerType = WorkerType.DESARROLLADOR;
21.     private String name;
22.     private String surname;
23.     private Company company;
24.
25.     @Id @GeneratedValue(strategy = GenerationType.AUTO)
26.     @Override
27.     public Long getId() {
28.         return id;
29.     }
30.
31.     @Column(length = 20, nullable = false)
32.     @Enumerated(EnumType.STRING)
33.     public WorkerType getWorkerType() {
34.         return workerType;
35.     }
36.
37.     public void setWorkerType(WorkerType workerType) {
38.         this.workerType = workerType;
39.     }
40.
41.     @Column(length=255, nullable=false)
42.     public String getName() {
43.         return name;
44.     }
45.
46.     public void setName(String name) {
47.         this.name = name;
48.     }
49.
50.     @Column(length=255, nullable=false)
51.     public String getSurname() {
52.         return surname;
53.     }
54.
55.     public void setSurname(String surname) {
56.         this.surname = surname;
57.     }
58.
59.     @ManyToOne(targetEntity = Company.class)
60.     @JoinColumn(name = "COMPANY_id")
61.     public Company getCompany() {
62.         return company;
63.     }
64.
65.     public void setCompany(Company company) {
66.         this.company = company;
67.     }
68. }
69.
```

Como habreis podido observar ambas heredan de `AbstractEntity`, donde se infieren cosas comunes a las clases mencionadas anteriormente:

view plain print ?

```
01. package com.autentia.springhb.entity;
02.
03. import java.io.Serializable;
04.
05. public abstract class AbstractEntity implements Serializable {
06.
07.     protected Long id;
08.
09.     // Este metodo es abstracto porque las anotaciones no son heredables
10.     public abstract Long getId();
11.
12.     // Este metodo es protegido para evitar que un prrgramador pueda poner un
13.     // idenficador en la instancia, ya que los identitificadores deben ser
14.     // gestionados por la capa de persistencia
15.     protected void setId(final Long id) {
16.         this.id = id;
17.     }
18. }
19.
```

Y para rematar, los trabajadores poseen un tipo de trabajador representado con la siguiente clase enumerada:

```
view plain print ?
01. package com.autentia.springhb.entity;
02.
03. public enum WorkerType {
04.     JEFE, CONSULTOR, DESARROLLADOR, BECARIO
05. }
06.
```

**IMPORTANTE:** Daros cuenta de que las anotaciones NO POSEEN ninguna dependencia con Hibernate (todas proceden de `javax.persistence`). Gracias al uso de estas anotaciones, que forman parte de JPA, podremos cambiar hibernate por cualquier otro ORM que funcione bajo JPA de una manera muy sencilla en los ficheros de configuración de Spring.

## 5. DAO

Otra de las partes que vamos a necesitar en nuestra aplicación son los DAO. Para ello vamos a definir unas interfaces con las operaciones que se puede realizar con ellos y tras eso implementaremos esas interfaces usando el soporte que brinda Spring para Hibernate. Las interfaces son muy simples:

Interfaz `CompanyDAO`:

```
view plain print ?
01. package com.autentia.springhb.dao;
02.
03. import java.util.List;
04.
05. import com.autentia.springhb.entity.Company;
06.
07. public interface CompanyDao extends GenericDao {
08.
09.     /**
10.       * Recupera todas las empresas
11.       */
12.     public List<Company> findAll();
13.
14.     /**
15.       * Recupera una empresa buscandola por su id
16.       */
17.     public Company findById(final Long id);
18. }
19.
```

Interfaz `WorkerDAO`:

```
view plain print ?
01. package com.autentia.springhb.dao;
02.
03. import java.util.List;
04.
05. import com.autentia.springhb.entity.Worker;
06.
07. public interface WorkerDao extends GenericDao {
08.
09.     /**
10.       * Recupera todos los trabajadores de todas las empresas
11.       */
12.     public List<Worker> findAll();
13.
14.     /**
15.       * Recupera un trabajador buscandolo por su id
16.       */
17.     public Worker findById(final Long id);
18. }
19.
```

Interfaz `GenericDAO`:

```
view plain print ?
01. package com.autentia.springhb.dao;
02.
03. import com.autentia.springhb.entity.AbstractEntity;
04.
05. public interface GenericDao {
06.
07.     /**
08.       * Almacena un objeto en base de datos
09.       */
10.     public void save(final AbstractEntity object);
11.
12.     /**
13.       * Elimina un objecto de la base de datos
14.       */
15.     public void delete(final AbstractEntity object);
16.
17. }
18.
```

Como podeis ver hay una pequeña herencia entre interfaces, ya que tanto `WorkerDAO` como `CompanyDAO` heredan de `GenericDAO` ya que, al fin y al cabo, los métodos `save()` y `delete()` borran cualquier tipo de entidad que pueda ser persistida en base de datos.

Una vez comentada la herencia entre interfaces, vamos a por la implementación:

Clase `HbSpringCompanyDaoImpl`:

```
view plain print ?
01. package com.autentia.springhb.dao.impl.hbspring;
02.
03. import java.util.List;
04.
05. import com.autentia.springhb.dao.CompanyDao;
06. import com.autentia.springhb.entity.Company;
07.
08. @SuppressWarnings("unchecked")
09. public class HbSpringCompanyDaoImpl extends AbstractHbSpringGenericDaoImpl implements CompanyDao {
10.
11.     public List<Company> findAll() {
12.         System.out.println("usando el metodo 'findAll()' de la clase '" + this.getClass().getSimpleN
13.         return getHibernateTemplate().find("from Company");
14.     }
15.
16.     public Company findById(Long id) {
17.         System.out.println("usando el metodo 'findById()' de la clase '" + this.getClass().getSimple
18.         return (Company) getHibernateTemplate().get(Company.class, id);
19.     }
20. }
21.
```

Clase `HbSpringWorkerDaoImpl`:

```
view plain print ?
01. package com.autentia.springhb.dao.impl.hbspring;
02.
03. import java.util.List;
04.
05. import com.autentia.springhb.dao.WorkerDao;
06. import com.autentia.springhb.entity.Worker;
07.
08. @SuppressWarnings("unchecked")
09. public class HbSpringWorkerDaoImpl extends AbstractHbSpringGenericDaoImpl implements WorkerDao {
10.
11.     public List<Worker> findAll() {
12.         System.out.println("usando el metodo 'findAll()' de la clase '" + this.getClass().getSimpleN
13.         return getHibernateTemplate().find("from Worker");
14.     }
15.
16.     public Worker findById(Long id) {
17.         System.out.println("usando el metodo 'findById()' de la clase '" + this.getClass().getSimple
18.         return (Worker) getHibernateTemplate().get(Worker.class, id);
19.     }
20.
21. }
22.
```

Clase `AbstractHbSpringGenericDaoImpl`:

```
view plain print ?
01. package com.autentia.springhb.dao.impl.hbspring;
02.
03. import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
04.
05. import com.autentia.springhb.dao.GenericDao;
06. import com.autentia.springhb.entity.AbstractEntity;
07.
08. public abstract class AbstractHbSpringGenericDaoImpl extends HibernateDaoSupport implements GenericD
09.
10.     public void save(final AbstractEntity object) {
11.         System.out.println("usando el metodo 'save()' de la clase '" + this.getClass().getSimpleName
12.         getHibernateTemplate().saveOrUpdate(object);
13.     }
14.
15.     public void delete(final AbstractEntity object) {
16.         System.out.println("usando el metodo 'delete()' de la clase '" + this.getClass().getSimpleNa
17.         getHibernateTemplate().delete(object);
18.     }
19. }
20.
```

Bueno, como podeis ver tanto los DAO `HbSpringCompanyDaoImpl` como `HbSpringWorkerDaoImpl` heredan de `AbstractHbSpringGenericDaoImpl`, que a su vez hereda de `HibernateDaoSupport`. ¿Qué conseguimos con esto?. `HibernateDaoSupport` es una clase de Spring muy simple, y su función es sencillamente dejar más limpio el código de nuestros DAO implementando métodos como el `getHibernateTemplate()`, que nos permitirá lanzar consultas que usen Hibernate, y otros métodos que permitan la inyección de sesiones de Hibernate desde Spring (en concreto me refiero al método `setSessionFactory()`, que como ya lo heredamos, no tenemos que implementarlo)

## 6. Configuración de Spring

Tenemos base de datos, tenemos entidades, tenemos DAO, ahora sólo nos falta una cosa, juntarlo todo para poder trabajar con ello. Es aqui donde Spring entra en juego:

view plain print ?

```
xml version="1.0" encoding="UTF-8"?
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/s

    <!-- ------------------------- -->
    <!-- Configuracion del datasource -->
    <!-- ------------------------- -->

    <bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <property name="driverClassName">
            <value>com.mysql.jdbc.Driver</value>
        </property>
        <property name="url">
            <value>jdbc:mysql://localhost/autentiaSpringHb</value>
        </property>
        <property name="username">
            <value>autentia</value>
        </property>
        <property name="password">
            <value>autentia</value>
        </property>
    </bean>

    <!-- ------------------------ -->
    <!-- Configuracion de hibernate -->
    <!-- ------------------------ -->

    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
        <property name="dataSource">
            <ref bean="dataSource" />
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.MySQLDialect
                </prop>
            </props>
        </property>
        <property name="annotatedClasses">
            <list>
                <value>
                    com.autentia.springhb.entity.Company
                </value>
                <value>
                    com.autentia.springhb.entity.Worker
                </value>
            </list>
        </property>
    </bean>

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

    <!-- ------------------------------------------------- -->
    <!-- Definicion de los DAO a utilizar por la aplicacion     -->
    <!-- ------------------------------------------------- -->

    <bean id="companyDao"
        class="com.autentia.springhb.dao.impl.hbspring.HbSpringCompanyDaoImpl">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <bean id="workerDao"
        class="com.autentia.springhb.dao.impl.hbspring.HbSpringWorkerDaoImpl">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
</beans>

```

Para configurar el datasource indicamos el driver que vamos a usar, cual es la URL de conexión , y cual es el usuario y la clave con las que acceder a la base de datos.

Para configurar Hibernate indicamos el dialecto con el que debe trabajar, cuales son las clases anotadas que debe persistir, y cual es el datasource donde encontrar la conexión con base de datos. Tambien configuraremos el gestor de transacciones.

Finalmente definimos los DAO que vamos a utilizar en la aplicacion y les inyectaremos la factoria de sesiones de Hibernate. Esta factoria es facilitada por Spring, al igual que la clase `HibernateDAOSupport`.

Si en algún momento quisieramos quitar Hibernate y utilizar otro motor de persistencia deberiamos:

1. Crear una nueva implementación de los DAO para las interfaces indicadas anteriormente
2. Cambiar la definición de los DAO que hacemos en Spring (bastaria con cambiar el `class="..."`)
3. Quitar la configuración de Hibernate del fichero de configuración de Spring y configurar el otro motor de persistencia.

Puede parecer mucho, pero si se jerarquizan bien los DAO y se crean pruebas de integración el trabajo de migrar de un motor de persistencia a otro puede ser muy poco, ya que por un lado tendremos que cambiar poco código (si hemos jerarquizado bien) y sabremos que la aplicación funciona en unos momentos (si hemos creado pruebas de integración)

## 7. Ejecución del programa

Para lanzar la aplicación utilizaremos el clásico método `main()`

view plain print ?

```
01. package com.autentia.springhb;
02.
03. import org.springframework.context.ApplicationContext;
04. import org.springframework.context.support.ClassPathXmlApplicationContext;
05.
06. import com.autentia.springhb.dao.CompanyDao;
07. import com.autentia.springhb.dao.WorkerDao;
08. import com.autentia.springhb.entity.Company;
09. import com.autentia.springhb.entity.Worker;
10. import com.autentia.springhb.entity.WorkerType;
11.
12. public class AutentiaSpringHb {
13.
14.     private static String[] files = new String[] {"applicationContextHbSpring.xml"};
15.
16.     public static void main(String[] args) {
17.
18.         final ApplicationContext context = new ClassPathXmlApplicationContext(files);
19.
20.         final CompanyDao companyDao = (CompanyDao) context.getBean("companyDao");
21.         final WorkerDao workerDao = (WorkerDao) context.getBean("workerDao");
22.
23.         Company company = new Company();
24.         company.setName("Autentia Real Business Solutions");
25.         company.setDescription("somos pocos, somos buenos y nos gusta lo que hacemos");
26.         companyDao.save(company);
27.
28.         Worker worker_01 = new Worker();
29.         worker_01.setName("Roberto");
30.         worker_01.setSurname("Canales");
31.         worker_01.setWorkerType(WorkerType.JEFE);
32.         worker_01.setCompany(company);
33.         workerDao.save(worker_01);
34.
35.         Worker worker_02 = new Worker();
36.         worker_02.setName("Carlos");
37.         worker_02.setSurname("GarcíÃa");
38.         worker_02.setWorkerType(WorkerType.CONSULTOR);
39.         worker_02.setCompany(company);
40.         workerDao.save(worker_02);
41.
42.         Worker worker_03 = new Worker();
43.         worker_03.setName("Alfonso");
44.         worker_03.setSurname("Blanco");
45.         worker_03.setWorkerType(WorkerType.BECARIO);
46.         worker_03.setCompany(company);
47.         workerDao.save(worker_03);
48.
49.         System.out.println ("Resumen:");
50.         System.out.println ("· hay " + companyDao.findAll().size() + " empresa/s");
51.         System.out.println ("· hay " + workerDao.findAll().size() + " empleado/s");
52.     }
53. }
54.
```

Lo que hacemos paso a paso es:

1. Creamos un contexto a partir del fichero de configuración de Spring. Como podeis ver es una constante con un array de Strings, donde cada fichero de configuración es un String. Esto es, podeis crear tantos ficheros como querais para configurar vuestra aplicación con Spring, aunque yo sólo haya utilizado uno.
2. Recuperamos los DAO que vamos a utilizar.
3. Creamos la empresa Autentia y a 3 empleados de ella y los enlazamos.
4. Tras eso, buscamos en la base de datos y vemos que se han guardado. Ademas durante la ejecución veremos la salida de los System.out que hemos dejado en los DAO a modo de log.

## 8. Conclusiones

Si mirais otra vez el código vereis que hemos escrito poco código y que éste es muy fácil de escribir y mantener, pero que con él hemos conseguido las 4 operaciones básicas sobre base de datos (crear, leer, actualizar y borrar), aunque en el ejemplo solo creemos entidades.

Tambien hemos visto cómo utilizar Spring para juntar todos estos componentes, qué deberiamos hacer para cambiar de motor de persistencia, y cómo hacer para acceder al contexto de Spring desde una clase propia y, desde ella, recuperar los DAO y poder operar con los beans declarados en Spring.

Espero que os sea de utilidad.

## Tutoriales recomendados

| Nombre | Resumen | Fecha | Visitas | pdf |
|---|---|---|---|---|
| Spring: definición dinámica de Beans | Este tutorial habla sobre la modificación dinámica de los beans del contexto para simplificar la configuración de Spring | 2007-05-09 | 2783 | pdf |
| Hibernate y el mapeo de la herencia | En este tercer tutorial de la saga vamos a ver en este tutorial como implementar las relaciones de herencia con las anotaciones de JPA | 2007-06-27 | 2686 | pdf |
| Comparativa entre EJB3 y Spring | En este tutorial os mostramos una comparativa entre EJB3 y Spring esperando que os ayude a decidir qué tecnología utilizar. | 2007-10-17 | 1908 | pdf |
| SpringIDE, plugin de Spring para Eclipse | En adictosaltrabajo os hemos ido presentando diversos plugins para Eclipse. Esta vez le toca el turno a SpringIDE, un plugin que os ayudará a desarrollar aplicaciones que utilicen Spring. | 2008-01-19 | 737 | pdf |
| Anotaciones en EJB 3.0 | Este tutorial nos va a enseñar algunas características del API de EJB 3.0 y las mejoras introducidas en la nueva version 3.0 | 2007-05-25 | 6146 | pdf |
| Comparativa entre Hibernate y EJB3 en la Capa de Persistencia | El presente documento pretende dar algunas luces a la comparativa entre la opción de usar Hibernate y/ó EJB3 para la capa de persistencia | 2007-08-16 | 3646 | pdf |
| 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 | 2007-06-25 | 3769 | pdf |
| 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. | 2007-06-13 | 5641 | pdf |
| 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) | 2004-08-14 | 49099 | pdf |

### Nota: