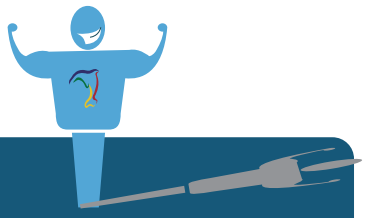
autentia

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...

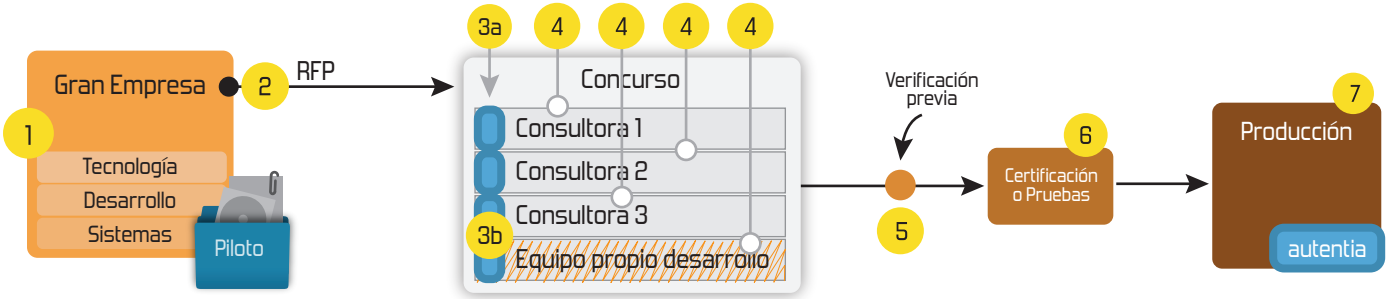
1. Desarrollo de componentes y proyectos a medida



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

Alejandro Pérez García

Alejandro es socio fundador de Autentia y nuestro experto en J2EE, Linux y optimización de aplicaciones empresariales.

Ingeniero en Informática

Si te gusta lo que ves, puedes contratarle para impartir **cursos presenciales** en tu empresa o para **ayudarte en proyectos** (Madrid). Puedes encontrarme en Autentia

Fecha de creación del tutorial: 2008-05-14

Spring + Hibernate + Anotaciones = Desarrollo Rápido en Java

Creación: 09-05-2008

Índice de contenidos

1. Introducción

Uno de los grandes problemas que tiene hoy en día el desarrollo de aplicaciones Web en Java es que el ciclo de desarrollo es, en muchas ocasiones, demasiado largo (o por lo menos más largo de lo que nos gustaría ;).

Debido a este problema han surgido alternativas del estilo de Ruby on Rails (http://www.rubyonrails.org/) o incluso Google App Engine (http://code.google.com/appengine/) una alternativa que propone Google, basada en el lenguaje Python.

Todas estas alternativas pueden resultar muy interesantes, pero suelen estar basadas en lenguajes con chequeo de tipos débil, o trasladando el chequeo de tipos a tiempo de ejecución (como Python), lo que provoca que puedan ser muy útiles para hacer rápidamente pequeñas aplicaciones o prototipos, pero que se pueden convertir en un gran problema cuando queremos construir aplicaciones medianas o grandes donde intervienen varias personas o incluso equipos en el proceso de desarrollo. Para este caso de aplicaciones medianas o grandes y grupos de desarrollo colaborativos, se hace necesario un lenguaje fuertemente tipado, donde podamos definir jerarquías de tipos (clases o interfaces) en las que el resto del equipo se pueda apoyar para desarrollar sin riesgos.

En este tutorial veremos como gracias a Spring + Hibernate + Anotaciones podemos conseguir un desarrollo tan rápido como el que podemos conseguir con las alternativas antes mencionadas.

Ya hemos visto en otros tutoriales el uso de Spring o Hibernate, pero en este tutorial vamos a intentar sacar todo el partido a las Anotaciones de Java 5 para, basándonos en el concepto de "convención frente a configuración", centrarnos en el código, olvidarnos de la base de de datos y de esos tediosos ficheros de configuración en XML.

Con esto no quiero decir que debamos olvidarnos por completo de esos ficheros XML, sino que debemos centrarnos a resolver el problema que nos ocupa, de forma rápida y con un buen diseño, consiguiendo un código legible y mantenible. Si luego queremos hacer ciertos refinamientos, o virguerías, los XML siempre estarán esperándonos para poder sobreescribir el comportamiento establecido con las anotaciones.

Últimos tutoriales

2008-05-14
Spring + Hibernate + Anotaciones = Desarrollo Rápido en Java

2008-05-06
J2ME. Internacionalización de aplicaciones para móviles

2008-05-05
Prototype.js: la sombra que se esconde detrás de todo

2008-05-05
Creación de una aplicación web con SpringMVC desde 0

2008-05-05
Cómo integrar Eastwood en nuestras aplicaciones web

2008-04-28
Cómo lanzar aplicaciones web desde Maven con Jetty

2008-04-28
Solución al problema de la exportación a HTML de informes JasperReports

2008-04-21
Proyecto Sakai: Una plataforma de e-learning libre (II)

2008-04-21
Proyecto Sakai: Una plataforma de e-learning libre (I)

2008-04-19
Ampliacion de la comparativa de antivirus freeware y opensource

Últimas ofertas de empleo

2008-04-29
Otras - Mecánica - ASTURIAS.

2008-04-28
T. Información - Administrador Sistemas UNIX / NT - CIUDAD REAL.

2008-04-23

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).
- Nvidia GEFORCE GO 7700
- Sistema Operativo: GNU / Linux, Debian (unstable), Kernel 2.6.24, KDE 3.5
- Java Sun 1.6.0_06
- Spring 2.5.4
- Hibernate 3.2.6
- JSF (RI 1.2) + Facelets 1.1.14 + ICEfaces 1.7

3. La aplicación

Vamos a hacer una pequeña aplicación donde se muestre un listado de productos. Podría quedar algo como:

Y la pantalla de edición de productos:

4. La capa de persistencia

Vamos a empezar "de abajo a arriba", es decir, partiremos definiendo nuestras entidades persistentes con Hibernate, e iremos "subiendo" hasta la capa de presentación y control con JSF, pasando antes por el negocio (el modelo) con Spring.

4.1. Las entidades

En nuestro ejemplo sólo tenemos la entidad *producto*, con los atributos *nombre*, *descripción* y *precio*.

Veamos como nos quedaría la clase:

view plain print ?

```
01. @Entity
02. public class Product {
03.     @Id
04.     @GeneratedValue
05.     private Integer id;
06.
07.     private String name;
08.
09.     private String description;
10.
11.     private float price;
12.
13.     Product() {
14.         // Sólo el manager puede constuir nuevas instancias
15.     }
16.
17.     public String getName() {
18.         return name;
19.     }
20.
21.     public void setName(String name) {
22.         this.name = name;
23.     }
24.
25.     public String getDescription() {
26.         return description;
27.     }
28.
29.     public void setDescription(String description) {
30.         this.description = description;
31.     }
32.
33.     public float getPrice() {
34.         return price;
35.     }
36.
37.     public void setPrice(float price) {
38.         this.price = price;
39.     }
40.
41. }
```

Podemos ver como se trata de una clase totalmente normal, donde en la línea 1 anotamos que se trata de una entidad, y en las líneas 3 y 4 indicamos cual es el id de la entidad y que este id será generado por la base de datos.

Cabe destacar dos cosas:

- Todas las notaciones usadas pertenecen al estándar de JPA por lo que son válidas tanto para Hibernate como para EJB3.0.
- Hemos anotado un atributo privado que no se usa en ningún sitio, ni siquiera tenemos getter o setter. Esto lo hacemos a posta ya que es algo que gestionará internamente Hibernate, y queremos condicionar lo menos posible nuestro diseño (nuestro negocio).

4.2. El DAO

El DAO es el Data Access Object, es decir, será la clase donde resida la lógica de manejo de Hibernate (o JDO o JDB o JPA o ...). De esta forma conseguimos que nuestra lógica de negocio no sepa nada de Hibernate, y siempre que quiera acceder a los datos lo hara usando esta clase.

Veamos un ejemplo sencillo: Primero definimos una interfaz, así podemos intercambiar la implementación fácilmente si algún día nos cansamos de Hibernate (no lo creo ;):

view plain print ?

```
01. public interface Dao {
02.
03.     public void persist(Object entity);
04.
05.     public void persist(Object[] entities);
06.
07.     public <T> List<T> find(Class<T> entityClass);
08.
09.     public <T> T load(Class<T> entityClass, Serializable id);
10.
11.     public <T> List<T> find(String hql);
12.
13. }
```

Para el ejemplo sólo hemos definido algunas operaciones simples. Ahora veamos una posible implementación usando las facilidades que nos proporciona Spring + Hibernate:

view plain print ?

```
01. @Repository
02. public class SpringHibernateDao extends HibernateDaoSupport implements Dao {
03.
04.     @Autowired
05.     public SpringHibernateDao(SessionFactory sessionFactory) {
06.         super.setSessionFactory(sessionFactory);
07.     }
08.
09.     @Transactional
10.     public void persist(Object entity) {
11.         getHibernateTemplate().saveOrUpdate(entity);
12.     }
13.
14.     @Transactional
15.     public void persist(Object[] entities) {
16.         for (int i = 0; i < entities.length; i++) {
17.             persist(entities[i]);
18.         }
19.     }
20.
21.     @Transactional(readOnly = true)
22.     public <T> List<T> find(Class<T> entityClass) {
23.         final List<T> entities = getHibernateTemplate().loadAll(entityClass);
24.         return entities;
25.     }
26.
27.     @Transactional(readOnly = true)
28.     public <T> T load(Class<T> entityClass, Serializable id) {
29.         final T entity = (T)getHibernateTemplate().load(entityClass, id);
30.         return entity;
31.     }
32.
33.     @Transactional(readOnly = true)
34.     public <T> List<T> find(String hql) {
35.         final List<T> entities = getHibernateTemplate().find(hql);
36.         return entities;
37.     }
38.
39. }
```

No es el ámbito de este tutorial estudiar la implementación de los métodos, para más información sugiero al lector repasar otros tutoriales relacionados o acudir a la documentación de Spring e Hibernate y la documentación sobre Generics de Java.

Aunque el lector en un principio no entienda la implementación lo que creo que queda claro es que es sencilla, puesto que se limita a unas pocas líneas (de nuevo sugiero repasar la documentación).

Donde si vamos a hacer especial hincapié es en las nuevas anotaciones que nos han aparecido:

- En la línea 1 nos encontramos con **`@Repository`**. Esta es una anotación de Spring. Estamos indicando que esta es una clase relacionada con la capa de persistencia, y que debe ser un Singleton (sólo habrá una instancia de la clase `HibernateDaoSupport`, y todos los Threads de la aplicación la compartirán).
- En la línea 4 nos encontramos con **`@Autowired`**. Esta es una anotación de Spring. Sirve para indicarle a Spring que cuando vaya a crear la instancia de HibernateDaoSupport debe "inyectarle" (pasarle) en el constructor una referencia al SessionFactory (el SessionFactory sí lo configuraremos mediante XML, lo veremos más adelante).
- Por último, en la línea 9, 14, 20, ... nos encontramos con la anotación **`@Transactional`**. Esta es una anotación de Spring. Estamos indicando que el método en cuestión es transaccional. Lo que hará Spring es comprobar si ya existe una transacción abierta, si existe se unirá a ella, y si no existe, abrirá una nueva transacción (este comportamiento es configurable). De esta forma nos aseguramos que toda operación de la base de datos se realiza dentro de una transacción. Además si durante la ejecución del método se produce alguna excepción de Runtime, se hará automáticamente rollback de la transacción (este comportamiento también es configurable).

Ya hemos terminado con la capa de persistencia. Rápido ¿verdad?. En ningún momento hemos visto sentencias SQL, ni siquiera para crear las tablas de la base de datos. Más adelante veremos como configuramos Hibernate para que se encargue de crearnos las tablas automáticamente (Los ficheros de configuración los veremos todos al final, por ahora sigamos con el código Java).

5. La capa de negocio

En esta aplicación el negocio no es gran cosa, poco más que obtener los productos o guardarlos, así que la clase nos va a quedar muy sencillita:

view plain print ?

```
01. @Service
02. public class ProductMgr {
03. 
04.     @Resource
05.     private Dao dao;
06. 
07.     public Product newProduct() {
08.         return new Product();
09.     }
10. 
11.     public void persist(Product product) {
12.         dao.persist(product);
13.     }
14. 
15.     public List<Product> getProducts() {
16.         final List<Product> list = dao.find(Product.class);
17.         return list;
18.     }
19. }
```

Haciendo una clase tan sencilla y que lo único que hace es delegar en el DAO, hay quien me podría acusar de estar cayendo en el antipatrón "Poltergeist", ya que desde control podríamos usar directamente el DAO para recuperar o guardar los productos, y quitarnos esta clase de enmedio. Pero no creo que este sea el caso ya que prima el MVC y el bajo acoplamiento.

Siempre debemos intentar que la capa de control y presentación sean lo más tontas posibles. Pensar por un momento que no usamos esta clase "manager" y que usamos el DAO desde las clases de control de JSF (los managed-beans), si ahora quisiéramos montar un web service para aprovechar esta aplicación desde otras aplicaciones ¿cuanto código que ya habríamos escrito en el managed-bean tendríamos que repetir en el web service?

Pero vamos al lío, que hemos venido a hablar de las anotaciones ;)

- En la línea 1 nos encontramos con **`@Service`**. Esta es una anotación de Spring, similar a `@Repository` que ya habíamos visto antes. Estamos indicando que esta es una clase relacionada con la capa de servicio (clases de negocio), y que debe ser un Singleton.
- En la línea 4 nos encontramos con **`@Resoruce`**. Esta anotación es del estándar, por lo que es válida tanto con Spring como con EJB3.0. Esta indicando que al crear la instancia de esta clase se debe "inyectar" (inicializar) en este atributo una referencia a la instancia del Dao (es la instancia que habíamos declarado anteriormente con `@Repository`).

Se acabo Â¡Â¡Â¡ Ya hemos terminado con negocio !!!

6. La capa de control

Vamos a implementar el control con los managed-beans de JSF. Como tenemos dos pantallas podemos hacer dos managed-bean.

El de la pantalla con el listado de productos nos podría quedar algo como:

view plain print ?

```
01. @Controller
02. @Scope("session")
03. public class ListProduct {
04. 
05.     @Resource
06.     private ProductMgr productMgr;
07. 
08.     @Resource
09.     private EditProduct editProduct;
10. 
11.     private UIData productsDataTable;
12. 
13.     public String editProduct() {
14.         Product product;
15.         try {
16.             product = (Product)productsDataTable.getRowData();
17. 
18.         } catch (IllegalArgumentException e) {
19.             // No se ha seleccionado ninguna fila; se está añadiendo un nuevo elemento.
20.             product = productMgr.newProduct();
21.         }
22.         editProduct.setProduct(product);
23. 
24.         return "editProduct";
25.     }
26. 
27.     public List<Product> getProducts() {
28.         return productMgr.getProducts();
29.     }
30. 
31.     public UIData getProductsDataTable() {
32.         return productsDataTable;
33.     }
34. 
35.     public void setProductsDataTable(UIData productsDataTable) {
36.         this.productsDataTable = productsDataTable;
37.     }
38. }
```

La clase para la edición de los productos:

```
view plain print ?
01. @Controller
02. @Scope("session")
03. public class EditProduct {
04.
05.     @Resource
06.     private ProductMgr productMgr;
07.
08.     private Product product;
09.
10.     public String save() {
11.         productMgr.persist(product);
12.         return "home";
13.     }
14.
15.     public Product getProduct() {
16.         return product;
17.     }
18.
19.     public void setProduct(Product product) {
20.         this.product = product;
21.     }
22. }
```

Fijándonos en la clase de listado, las nuevas anotaciones que aparecen son:

- En la línea 1 nos encontramos con **`@Controller`**. Esta es una anotación de Spring, similar a `@Repository` o `@Service` que ya habíamos visto antes. Estamos indicando que esta es una clase relacionada con la capa de control.
- En la línea 2 nos encontramos con **`@Scope("session")`**. Esta es una anotación de Spring. Con ella estamos sobreescribiendo el comportamiento por defecto de Spring, que es hacer Singletons, y le estamos diciendo que nos cree una instancia diferente de esta clase por cada sesión Http. Es decir, cada usuario tendrá su propio managed-bean.
- También cabe destacar desde la línea 5 hasta la 9. La anotación @Resource ya la hemos comentado antes, pero quiero recalcar como se está "inyectando" la referencia al manager (la clase de negocio) y la referencia a otro managed-bean de la capa de control de JSF, es decir, Spring es capaz de gestionar las dependencias entre los diferentes managed-beans de JSF.

7. La capa de presentación

Está implementada con JSF + Facelets + ICEfaces, pero no tiene nada de especial. Es decir la construiremos como habitualmente se trabaja con estas tecnologías.

Cuando queramos acceder a los managed-beans desde el *Expression Language* simplemente lo haremos. Por ejemplo:

- ```
  view plain print ?
  01. <ice:commandButton value="Añadir producto" action="#{listProduct.editProduct}" />
  ```
- ```
  view plain print ?
  01. <tnt:inputText entity="#{editProduct.product}" field="name" required="true" />
  ```
- ```
  view plain print ?
  01. <ice:commandButton value="Guardar" action="#{editProduct.save}" />
  ```

8. Los ficheros de configuración

8.1. hibernate.cfg.xml (configuración de Hibernate)

```
view plain print ?
01. <?xml version="1.0" encoding="UTF-8"?>
02.
03. <!DOCTYPE hibernate-configuration PUBLIC
04.     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
05.     "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
06.
07. <hibernate-configuration>
08.     <session-factory>
09.         <property name="hibernate.hbm2ddl.auto">create</property>
10.         <property name="hibernate.show_sql">true</property>
11.
12.         <mapping class="com.autentia.store.product.Product"/>
13.     </session-factory>
14. </hibernate-configuration>
```

En la línea 9 es donde le estamos diciendo a Hiberante que queremos que nos cree las tablas al arrancar la aplicación. Ojo porque si las tablas ya existen las borra primero, es decir, esto puede ser muy conveniente para desarrollo o pruebas, pero no para producción !!! Lo que podemos hacer es, una vez están creadas, hacer un "export" de la base de datos para obtener los scripts de creación que podemos retocar para dejarlos listos para producción (pero nos ahorramos lo gordo)

8.2. applicationContext.xml (configuración de Spring)
```

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/XM
    xmlns:util="http://www.springframework.org/schema/util" xmlns:context="http://www.springfra
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/util http://www.springframework.org/schema/util/s
        http://www.springframework.org/schema/context http://www.springframework.org/schema/con
        http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spr
        http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/sprin

    <context:annotation-config />
    <context:component-scan base-package="com.autentia.store" />

    <aop:aspectj-autoproxy />

    <tx:annotation-driven transaction-manager="transacctionManager"/>

    <bean id="propertyConfigurer" class="org.springframework.beans.factory.config.PropertyPlace
        <property name="locations">
            <list>
                <value>classpath:jdbc.properties</value>
            </list>
        </property>
    </bean>

    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="${jdbc.driverClassName}" />
        <property name="url" value="${jdbc.url}" />
        <property name="username" value="${jdbc.username}" />
        <property name="password" value="${jdbc.password}" />
    </bean>

    <bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean
        <property name="dataSource" ref="dataSource" />
        <property name="configurationClass" value="org.hibernate.cfg.AnnotationConfiguration"/>
        <property name="configLocation">
            <value>classpath:${hibernate.cfg.file}</value>
        </property>
    </bean>

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

Puede parecer que hay mucho pero en realidad sólo hay 4 cosas: configuración de Spring para que haga caso a las anotaciones, definir el datasource (de hibernate, del servidor por jndi, ...), definir el sessionFactory de Hibernate, y definir el transactionManager (el de Hibernate, JTA, ...)

Si os fijáis no hay ni una sola definición de bean de clases que hayamos escrito nosotros, de forma que este fichero se mantendrá constante con independencia de los beans que tenga nuestra aplicación.

8.3. faces-config.xml (configuración de JSF)

view plain print ?

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE faces-config PUBLIC "-//Sun Microsystems, Inc.//DTD
  JavaServer Faces Config 1.1//EN" "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">

<faces-config xmlns="http://java.sun.com/JSF/Configuration">
    <application>
        <locale-config>
            <default-locale>es</default-locale>
            <supported-locale>en</supported-locale>
        </locale-config>

        <view-handler>com.icesoft.faces.facelets.D2DFaceletViewHandler</view-handler>
        <variable-resolver>org.springframework.web.jsf.DelegatingVariableResolver</variable-res
    </application>

    <!--
        | ================= Navigation rules =================
    -->
    <navigation-rule>
        <navigation-case>
            <from-outcome>home</from-outcome>
            <to-view-id>/listProduct.jspx</to-view-id>
        </navigation-case>
    </navigation-rule>

    <navigation-rule>
        <from-view-id>/listProduct.jspx</from-view-id>
        <navigation-case>
            <from-outcome>editProduct</from-outcome>
            <to-view-id>/editProduct.jspx</to-view-id>
        </navigation-case>
    </navigation-rule>

</faces-config>
```

Se puede apreciar como sólo hay configuración general de JSF y reglas de navegación. Pero no declaramos ningún managed-bean. Esto funciona gracias a la línea 13 donde se le indica a JSF que debe delegar en Spring para buscar los managed-beans. Es decir, JSF los buscara entre los que declaremos en el fichero (si es que declaramos alguno, que no es nuestro caso), y si no lo encuentra, lo buscará en Spring.

Como se puede comprobar, también nos ahorramos escribir cantidad de código en ese XML.

9. Diferencia entre las anotaciones `@Repository`, `@Service`, `@Controller`

La diferencia es básicamente semántica, es decir, cada una denota perfectamente a que "capa" corresponde la clase anotada. Pero todas se comportan de igual manera (por ejemplo en todas nuestras clases podríamos haber usado la anotación `@Service` y hubiera funcionado igual).

Esto se consigue porque las tres anotaciones extienden la anotación `@Component`.

El hecho de usar anotaciones diferentes puede ser muy interesante si luego queremos aplicar aspectos (AOP = Aspect Oriented Programming) a todas las clases de una misma capa. Es decir, por ejemplo, puedo hacer una regla para aplicar cierto *advice* a todas las clases con la anotación `@Controller`.

10. Conclusiones

Gracias a Spring + Hibernate + Anotaciones podemos conseguir reducir los tiempos de desarrollo con Java.

Ya existen otro tipo de frameworks similares como EJB3.0 o Seam (de JBoss) que también se basan en anotaciones, pero podemos ver algunas ventajas de usar Spring + Hibernate:

- Con Spring + Hibernate podemos usar contenedores ligeros como Tomcat, mientras que con EJB3.0 o Seam estamos condenados a usar un servidor de aplicaciones como JBoss. El tener que usar por obligación un servidor de aplicaciones aumenta los tiempos de desarrollo ya que se tarda más en desplegar en un servidor de aplicaciones que en un Tomcat. Además necesitaremos más recursos.
- Con Spring + Hibernate podemos hacer todo el desarrollo en un Tomcat, aunque finalmente acabemos instalando en producción en un servidor de aplicaciones.
- Con Spring + Hibernate podemos escribir aplicaciones normales de escritorio o línea de comandos, mientras que con EJB3.0 o Seam no, es decir con Spring + Hibernate, no es que nos valga con un contenedor ligero, es que no tenemos porque usar un contenedor en absoluto.
- Spring da cantidad de facilidades, como Seam, y posiblemente más que EJB3.0 (ya que se integra con gran cantidad de otros frameworks). Aunque no hay que olvidar que EJB3.0 permite transacciones distribuidas (Spring + Hibernate no lo permiten, aunque se pueden unir a una transacción JTA gestionada por un servidor de aplicaciones), y además los EJBs son por si mismos objetos distribuidos (muy fáciles de localizar y usar desde cualquier punto de nuestra red).

Al final ni todo es absolutamente bueno ni todo es absolutamente malo. Por eso debemos conocer opciones, evitar el "Golden Hammer", y quedarnos con lo mejor de cada casa ;)

11. 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

Tutoriales recomendados

Nombre	Resumen	Fecha	Visitas	pdf
Proyecto con JSF Java Server Faces Myfaces, Maven y Eclipse: Hibernate (segunda parte)	En este artículo se va a continuar con el desarrollo de la aplicación Myfaces JSF con Maven multimódulo que comenzamos en un tutorial anterior. Además también se tratará de la integración de Hibernate con las aplicaciones.	2007-07-31	3786	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	2922	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	5115	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	52900	pdf
Creación de una aplicación web con SpringMVC desde 0	Este tutorial te resultará muy útil para aprender a usar el patrón modelo-vista-controlador (MVC) con Spring a nuestros desarrollos web	2008-05-05	341	pdf
Creación de una aplicación con Spring e Hibernate desde 0	Este tutorial vamos a explicar paso a paso cómo crear una pequeña aplicación usando Spring e Hibernate con anotaciones partiendo desde 0	2008-02-15	3801	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	7837	pdf

Nota: