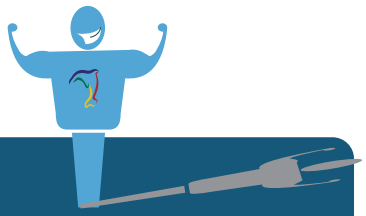
autentia

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...

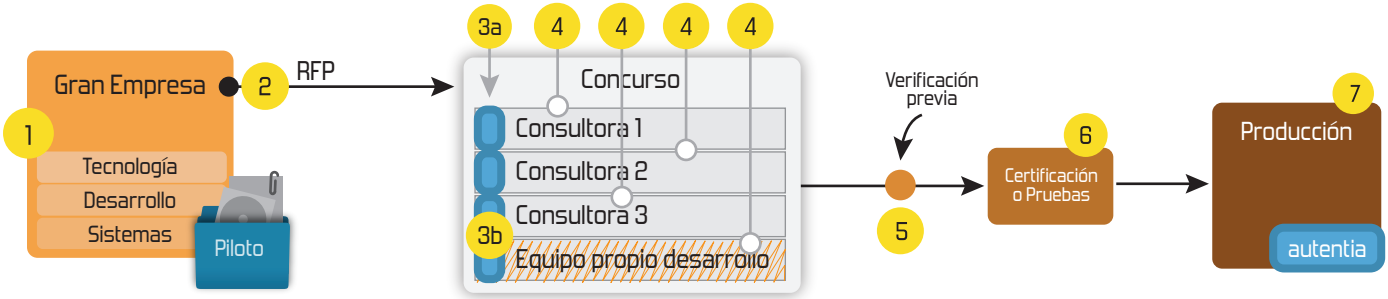
1. Desarrollo de componentes y proyectos a medida



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

Daniel Diaz Suarez

Desarrollador Web en Autentia

Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/JEE

Ver todos los tutoriales del autor

Fecha de publicación del tutorial: 2013-07-25 Tutorial visitado 1 veces Descargar en PDF

Spring Container e Inyección de Dependencias

0. Índice de contenidos.

Este es el segundo tutorial de una serie que nos ayudará a dar los primeros pasos con el framework Spring 3

1. Hola Mundo con Spring 3

1. Entorno

Este tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil MacBook Pro 17' (2.8 GHz Intel Core Duo, 8GB DDR3 SDRAM)
- Sistema Operativo: Mac OS X Lion 10.7.5
- Eclipse Kepler

2. Introducción

El framework Spring se compone de varios módulos, todos ellos giran entorno al Spring Core y más concretamente al Spring Container, el cual hace uso intensivo del patrón de inyección de Dependencias o de Inversión de Control, por lo tanto, para trabajar con bien con Spring es imprescindible conocer cómo funciona el contenedor de Spring y en qué consiste la Inyección de Dependencias, este tutorial nos servirá como una introducción al Contenedor de Dependencias de Spring y el patrón DI, también veremos un uso práctico de como hacer un test unitario usando Spring para ver el potencial de usar ambas.

3. Spring Container

El contenedor de Spring es uno de los puntos centrales de Spring, se encarga de crear los objetos, conectarlos entre si, configurarlos y además controla los ciclos de vida de cada objeto mediante el patrón de Inyección de Dependencias (Dependency Injection ó DI).

Podemos personalizar el contenedor de Spring mediante configuración XML o programáticamente, a lo largo del tutorial, veremos como hacerlo de ambas maneras.

Algunas de las cosas que tendremos que configurar a la hora de crear un Contexto de Aplicación de Spring son:

- Crear Beans* individualmente o a través de un "escaneador" de ficheros
- Configurar los servicios que usará

Los beans son la manera que tiene de denominar Spring a los objetos Java de los que se encarga, es decir aquellos que se encuentren en el contenedor de Spring

Los Bean se pueden declarar mediante anotaciones en POJO's (Plain Old Java Object , objetos normales de Java) o mediante XML, el siguiente ejemplo muestra como declarar un bean mediante configuración XML:

```
1 <bean id="petStore" class="org.springframework.samples.jpetstore.services.PetStoreService
2     <property name="accountDao" ref="accountDao"></property>
3     <property name="itemDao" ref="itemDao"></property>
4 </bean>
```

En este ejemplo estamos creado un Bean con id petStore, y se le indica donde se encuentra la clase, y las propiedades accountDao e itemDao están haciendo referencia a dos Beans con respectivas id's.

En el contenedor Spring se suelen crear y almacenar objetos de servicio, DAO's, y objetos que nos permitan

conectarnos con otras partes del sistema como Bases de Datos, Sistemas de Colas de Mensaje, etc... No se suelen configurar los objetos de dominio de nuestra aplicación para que se encargue el contenedor de Spring, ese sería el trabajo de los DAO's o Repositorios.

4. Inyección de dependencias

El patrón de Inyección de Dependencias, también conocido como de Inversión de Control es un patrón que tiene como finalidad conseguir un código mas desacoplado, que nos facilitará las cosas a la hora de hacer Tests y además nos permite cambiar partes del sistema más fácilmente en caso de que fuese necesario.

Tener el código desacoplado nos permite cambiar las dependencias en tiempo de ejecución basándonos en cualquier factor que considerásemos, para ello necesitaríamos un Inyector o Contenedor que sería el encargado de inyectar las dependencias correctas en el momento necesario.

Siguiendo el patrón de Inyección de Dependencias (DI, Dependency Injection) los componentes declaran sus dependencias, pero no se encargan de conseguirlas, ahí es donde entra el Contenedor de Spring, que en nuestras aplicaciones de Spring será el encargado de conseguir e inyectar las dependencias a los objetos.

El siguiente código muestra un ejemplo de clase que no usa el patrón de Inyección de Dependencia, además de estar fuertemente acopladas las dependencias, es la propia clase la que se encarga de crear una instancia de la dependencia:

```
1  public class GeneradorPlaylist {
2
3      private BuscadorCanciones buscadorCanciones;
4
5      public GeneradorPlaylist(){
6          this.buscadorCanciones = new BuscadorCanciones();
7      }
8
9      //Resto de métodos de la clase
10
11 }
```

La clase `GeneradorPlaylist` necesita una instancia de la clase `BuscadorCanciones` para funcionar, a lo largo del tutorial vamos a ver como mejorar el diseño de esta clase usando la inyección de dependencias y Spring.

4.1 Inyección de dependencias mediante constructor

En el siguiente ejemplo podemos ver como el objeto declara sus dependencias en el constructor, podemos observar que no hay código que se encargue de buscar esa dependencia o crearla, simplemente la declara, esto nos ayuda a tener clases Java mucho más limpias a la vez que nos ayuda a facilitar el Testing, ya que en un entorno de Tests podríamos intercambiar ese objeto por un Mock sin cambiar el código (mediante la configuración de Spring).

```
1  public class GeneradorPlaylist {
2
3      private BuscadorCanciones buscadorCanciones;
4
5      public GeneradorPlaylist(BuscadorCanciones buscadorCanciones){
6          this.buscadorCanciones = buscadorCanciones;
7      }
8
9      //Resto de métodos de la clase
10
11 }
```

Para informar a Spring de cual es la dependencia que tiene que inyectar en `GeneradorPlaylist` podemos hacerlo mediante XML o anotaciones, en el siguiente ejemplo vamos a ver como se configuraría mediante XML:

```
1      <!-- Le decimos a Spring que cree el Bean que luego inyectaremos  -->
2      <bean id="buscadorCanciones" class="com.autentia.BuscadorCanciones">
3
4      <!-- Creamos el Bean y le inyectamos el buscadorCanciones que hemos creado arriba -->
5      <bean id="generadorPlaylist" class="com.autentia.GeneradorPlaylist">
6          <constructor-arg type="com.autentia.BuscadorCanciones" ref="buscadorCanciones">
7      </constructor-arg></bean>
8
9  </bean>
```

4.2 Inyección de dependencias mediante "Setter"

Spring también permite inyectar la dependencia mediante los Setter (métodos set*()), cada forma de inyectar las dependencias tiene sus ventajas y sus desventajas, aunque la mayoría de los desarrolladores prefieren inyectar las dependencias mediante los métodos Set.

Para indicarle a Spring que queremos que inyecte la dependencia, podemos hacerlo mediante anotaciones o XML, vamos a ver como sería mediante anotaciones.

```
1  public class GeneradorPlaylist {
2
3      @autowired
4      private BuscadorCanciones buscadorCanciones;
5
6      public setBuscadorCanciones(BuscadorCanciones buscadorCanciones){
7          this.buscadorCanciones = buscadorCanciones;
8      }
9
10     //Resto de métodos de la clase
11 }
```

Mediante la anotación `@autowired` le indicamos a Spring que se tiene que encargar de buscar un Bean que cumpla los requisitos para ser inyectado, en este caso el único requisito es que sea del tipo BuscadorCanciones, en caso de que hubiese mas de un Bean que cumpliese esos requisitos tendríamos que decirle a Spring cuál es el Bean correcto.

4. Ejemplo Práctico: Testing usando la inyección de dependencias con el Spring Container

Ahora que sabemos como inyectar las dependencias a través de Spring, vamos a ver como aprovecharnos de una de las ventajas principales de usar Spring, el testing.

Usando el ejemplo anterior, vamos a ver como podríamos inyectar un objeto "Mock" (falso) que nos permitan hacer un Test Unitario de la clase que queremos probar, en este caso GeneradorPlaylist.

Hay distintas maneras de conseguir esto, Spring nos permite definir unos perfiles en nuestro archivo de configuración

para crear unas Beans u otras basándose en el perfil que queremos ejecutar la aplicación, este método lo veremos en futuros tutoriales.

De momento vamos a ver como ejecutar la aplicación con un Contexto de Aplicación de Testing, lo cual nos permite definir un contexto de aplicación para cada Test (en el cual podemos configurar solo la parte que nos interesa) , para ello vamos a usar la clase `SpringJUnit4ClassRunner` que nos provee Spring.

En el ejemplo siguiente vamos a hacer un Test Unitario para la clase `GeneradorPlaylist`:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"/TestApplicationContext.xml"})
public class GeneradorPlaylistTest{
    private BuscadorCanciones buscadorCanciones;
    private GeneradorPlaylist generadorPlaylist;

    @Before
    public void setUpTest(){
        generadorPlaylist = new generadorPlaylist();

        //Creamos un mock del buscadorCanciones
        buscadorCanciones = mock(BuscadorCanciones.class);

        generadorPlaylist.setBuscadorCanciones(buscadorCanciones);
    }

    @Test
    public void primerTest(){
        // Logica del Test
    }

    //Resto de Tests..
}
```

Como podemos ver no estamos haciendo nada del otro mundo para hacer un test que pruebe la clase GeneradorPlaylist, lo interesante se que lo hemos podido hacer sin necesidad de cambiar nada en la clase, ni de escribir código para poder probar un Bean del contenedor de Spring. Además el hecho de poder elegir que partes del sistema cargar para cada Test, aligera mucho el proceso de pasar los test, ya que no necesita levantar una Aplicación entera para probar cada pequeña parte de la Aplicación, lo cual también nos ayuda a aislar los test.

5. Conclusiones

Como hemos podido ver, el patrón de Inyección de Dependencias nos permite crear aplicaciones menos acopladas, lo cual facilitará la tarea de crear Test Unitarios, además nos libra de mucho código "pegamento" para unir las distintas dependencias con las clases que las necesitan, lo que nos ayudará código más limpio y reusable.

Spring Container es el aliado perfecto al usar este patrón, ya que nos permite configurar las dependencias a inyectar fácilmente mediante anotaciones o código XML, de manera que podamos librar a nuestros POJO's de tener código innecesario.