Spring

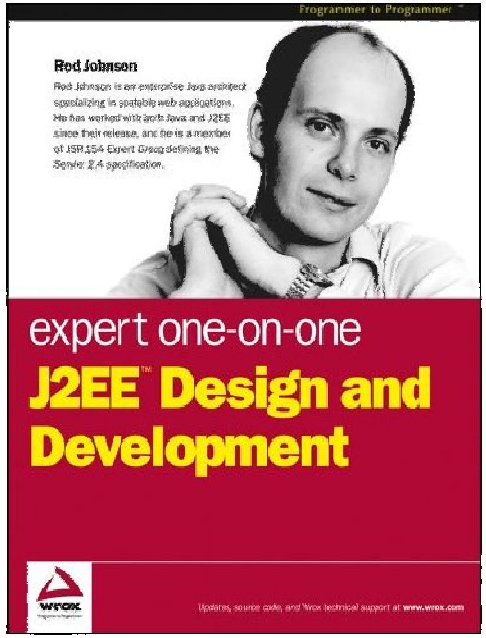
Sesión 1: El contenedor de *beans*

Indice

- Introducción a Spring. Filosofía
- Dependency injection vs. dependency lookup
- Beans de negocio
  - Definir: propiedades y relaciones entre ellos
  - Instanciar beans
- Acceder a recursos JNDI con beans

¿Qué es Spring?

- Inicialmente, un ejemplo hecho para el libro “J2EE design and development” de Rod Johnson, que defendía **alternativas** a la “visión oficial” de **aplicación J2EE basada en EJBs**
- Actualmente es un *framework* completo que cubre todas las capas del desarrollo
  - MVC
  - Negocio (donde empezó originalmente)
  - Acceso a datos



La filosofía “ortodoxa-J2EE-clásica”

- Los objetos de negocio deben ser EJBs, porque eso nos da una serie de servicios “sin necesidad de programar”
  - Seguridad, transaccionalidad, concurrencia, persistencia,…
- Los EJBs necesitan soporte del servidor de aplicaciones
  - Ya no vale Tomcat
- La realidad es que programar EJBs es complicado y los servidores de aplicaciones son costosos en dinero y recursos

La filosofía de Spring

- Los objetos de negocio deberían ser POJOs
- Se pueden conseguir servicios equivalentes a los que nos dan los EJB usando AOP de manera casi transparente al programador
- El contenedor debe poder ser un servidor web normal (ej. Tomcat)
- Programar contra interfaces es mejor que contra clases concretas
- Dependency injection es mejor que dependency lookup
- Cuando ya hay algo que funciona, incorpóralo a tu solución, no “reinventes la rueda”
  - Hibernate para persistencia de datos
  - Hessian para acceso remoto)

Inversion of Control

- El principio de Hollywood: **“gracias, ya le llamaremos...”**
- Los objetos son llamados por el contenedor (*callback*)
- No es nada raro, ocurre en EJBs, eventos, entornos de ventanas, etc. Resulta “raro” comparado con la programación “tradicional” de consola

```
public class GetPedidoAccion extends Action {
        public ActionForward execute(ActionMapping am, ActionForm af,

HttpServletRequest request,

                                HttpServletResponse response) {
                ...
        }
}
```

Lookup vs. injection

- **Dependency lookup:** el componente es responsable de localizar a sus “colaboradores”
- En la visión clásica,esto se hace con JNDI

```
public class GestorPedidos implements SessionBean {
        private GestorAlmacen almacen;

        public void ejbCreate() {
          Context ctx = new InitialContext();
          Object o = ctx.lookup("java:comp/env/GestorAlmacen");
          GestorAlmacenHome home = (GestorAlmacenHome)
PortableRemoteObject.narrow(o,

                    GestorAlmacenHome.class);
          almacen = home.create();
    }
}
```

Lookup vs. injection

- **Dependency injection:** el contenedor le pasa al componente, justo después de instanciarlo, la referencia a los colaboradores
- En Spring se sigue la convención JavaBean porque:
  - Es simple
  - No depende de APIs externos (*goodbye* JNDI)

```
public class GestorPedidos {
        private GestorAlmacen gestorAlmacen;

        public void setGestorAlmacen(GestorAlmacen ga) {
                gestorAlmacen = ga;
        }
    }
```

Definir beans

- Los beans de Spring son los objetos de la capa de negocio
- Se configuran con XML

```
<?xml version="1.0"?>
<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/schema/beans/spring-beans-
2.0.xsd">

  <bean id="miBean" class="mipaquete.MiClase">
      <!-- propiedades del bean y referencias a otros beans -->
      ...
  </bean>
  <bean id="..." class="...">
      ...
  </bean>
  ...
</beans>
```

Instanciar beans en una aplicación web

- Vale lo anterior, pero ¿cuándo se crea el objeto "miBean"?
- Si usamos Spring MVC, es automático
- Si no usamos Spring MVC

```
<!-- en el web.xml se ha definido dónde está el XML con la configuración,
      mirar en los apuntes cómo se hace-->
<%
 ServletContext sc = getServletContext();
 WebApplicationContext wac =

          WebApplicationContextUtils.getWebApplicationContext(sc);
 MiClase ejemplo = (MiClase) wac.getBean("miBean");
%>
```

Eh!! aquí hay algo raro

- ¿Esto no es *dependency lookup*?

```
MiClase ejemplo = (MiClase) wac.getBean("miBean");
```

- Mmm..sí
- No obstante en Spring se puede configurar todo con *dependency injection* usando el módulo MVC, que veremos en la siguiente sesión

Spring puede inicializar las propiedades del bean

Código Java

```
package springbeans;
public class PrefsBusqueda {
        private int maxResults;
        private boolean ascendente;
        private String idioma;

        //...aquí vienen los getters y
setters
}
```

XML de config.

```
<?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/schema/beans/spring-beans-2.0.xsd">
  <bean id="misPrefs" class="springbeans.PrefsBusqueda">
        <property name="maxResults"> <value>100</value> </property>
        <property name="idioma"> <value>es</value> </property>
        <property name="ascendente"> <value>true</value> </property>
  </bean>
</beans>
```

Relacionar beans

- Unos componentes necesitan a otros
- Supongamos que un **Buscador** necesita unas **PrefsBusqueda**

XML

```
<bean id="misPrefs"
    class="springbeans.PrefsBusqueda">
  <!--propiedades de este bean -->
  ...
</bean>
```

```
<bean id="miBuscador"
    class="springbeans.Buscador">
  <property name="prefs">
    <ref bean="misPrefs"/>
  </property>
<bean>
```

Java

```
package springbeans;

public class Buscador {
  private PrefsBusqueda prefs;

  public PrefsBusqueda getPrefs() {
    return prefs;
  }

  public void setPrefs(PrefsBusqueda misPrefs) {
    this.prefs = misPrefs;
  }
}
```

Relacionar beans automáticamente (*autowiring*)

- Usando *reflection*, se puede ver que **Buscador** tiene un **getPrefs/setPrefs** de tipo ***PrefsBusqueda*** , luego
- Automáticamente se puede instanciar un bean de tipo ***PrefsBusqueda y llamar a Buscador.setPrefs (autowire "por tipo")***

XML

```
<bean id="misPrefs"
    class="springbeans.PrefsBusqueda">
  <!--propiedades de este bean -->
  ...
</bean>
```

```
<bean id="miBuscador"
    class="springbeans.Buscador"
    autowire="byType">
</bean>
```

Acceso a recursos JNDI con beans

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
       xsi:schemaLocation="http://www.springframework.org/schema/bean
s      http://www.springframework.org/schema/beans/spring-beans-
2.0.xsd          http://www.springframework.org/schema/jee
       http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">
  <jee:jndi-lookup id="miFD" jndi-name="jdbc/MiDataSource"/>
  <bean id="miBuscador" class="springbeans.Buscador">
    <property name="fuenteDatos">
      <ref bean="miFD"/>
    </property>
  </bean>
</beans>
```

Java

```
public class Buscador {
  private DataSource fuenteDatos;
  ...
```

El Servlet controlador

- Ya está hecho en Struts
- Debemos configurar la aplicación para que todas las peticiones vayan a parar al mismo servlet (en el web.xml)

```
<servlet>
  <servlet-name>controlador</servlet-name>
  <servlet-class>
        org.apache.struts.action.ActionServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>controlador</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

¿Preguntas...?