Spring

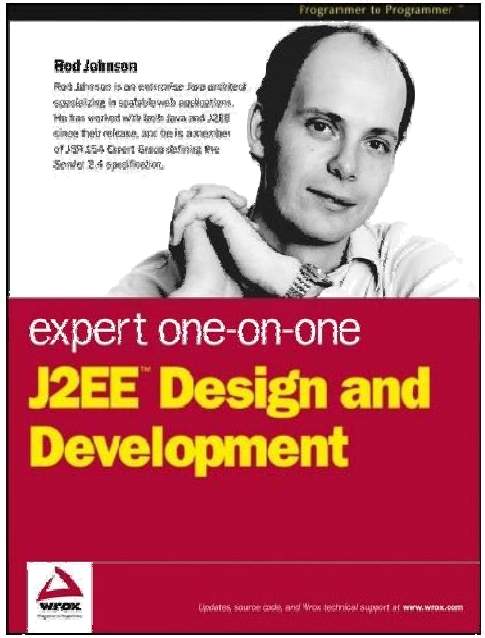
Sesión 1: El contenedor de *beans*

Indice

- Introducción a Spring. Spring vs. EJB 3
- El contenedor de beans (Spring core)
- Trabajo con beans
  - Definir beans
  - Instanciar beans
  - Ámbitos
- 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 2.x**
- Actualmente es un ***framework*** **completo** que cubre todas las capas de la aplicación, con decenas de desarrolladores y miles de descargas de SourceForge al día
  - MVC
  - Negocio (donde empezó originalmente)
  - Acceso a datos



EJBs

- La tecnología estándar para aplicaciones distribuidas en JavaEE
- Un EJB es un objeto que actúa como "componente de negocio". El servidor de aplicaciones proporciona una serie de servicios
  - Puede estar en cualquier máquina física y no obstante se puede usar de manera (casi) transparente
  - Transaccionalidad declarativa
  - Seguridad declarativa en el acceso a métodos
- Para EJBs necesitamos un servidor de aplicaciones (no vale Tomcat, sí p.ej. JBoss)

Generaciones de EJBs

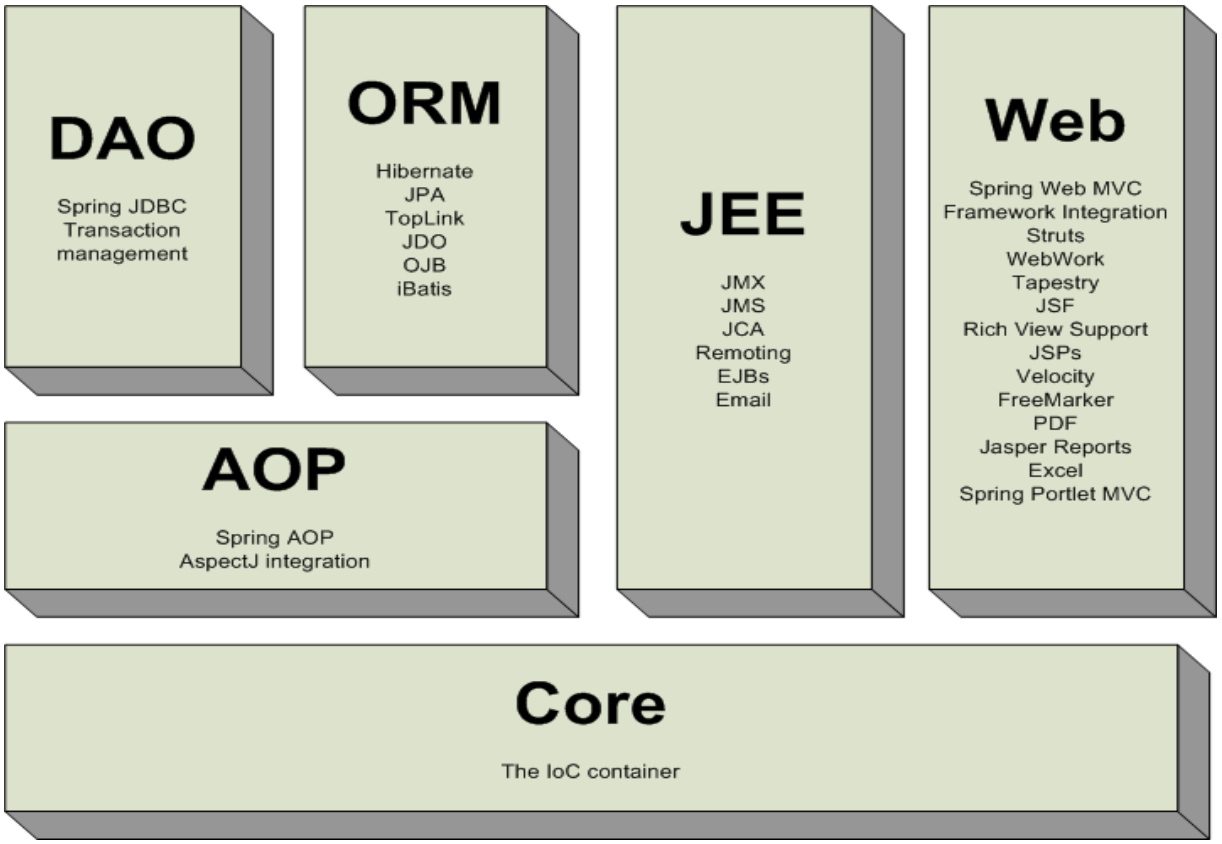
- La generación 2.X de EJB era demasiado pesada para las máquinas de la época y además compleja y difícil de usar
  - En esta época fue cuando surgió Spring
- La generación actual de EJBs (3.X) es mucho más sencilla de usar
  - JPA es parte de la especificación EJB 3
  - Inspirada en la forma de trabajar de Spring y otros *frameworks*
  - No obstante, Spring ha seguido evolucionando

La filosofía de Spring

- Los objetos de negocio deberían ser POJOs
- Inyección como medio de resolver dependencias
- 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 convencional (ej. Tomcat)
- Cuando ya hay algo que funciona, incorpóralo a tu solución, no “reinventes la rueda”
  - JPA para persistencia de datos
  - Hessian para acceso remoto

Módulos de Spring

- Excepto el **Core** todos son opcionales, no es “todo o nada” como muchos servidores de aplicaciones



Versiones de Spring

- La generación actual es la 2.x
  - De la 1 a la 2, se **potenció** bastante el *framework* y se **simplificó** mucho su **configuración**
- La versión que veremos aquí es la 2.5
  - Permite el uso de anotaciones, pero (evidentemente) requiere Java 5
- Spring tiene una excelente documentación on-line (increible, eh!!)

Spring Core

- Es un contenedor que gestiona el ciclo de vida de los objetos de nuestra aplicación
  - En realidad no es más que un conjunto de librerías que se puede ejecutar en cualquier servidor web java
  - Ofrece servicios a nuestros objetos, como inyección de dependencias
- Juntándolo con otros módulos, más servicios
  - (+ AOP): Transaccionalidad declarativa
  - (+ Spring Remoting): Acceso remoto
  - (+ Spring Security): Seguridad declarativa

Colaboración entre objetos

- Normalmente un objeto necesitará de otros para hacer su trabajo
- Si simplemente los instancia con new() perdemos flexibilidad
  - Necesitamos un mecanismo para poder resolver las dependencias en tiempo de ejecución
  - Hay dos opciones, ambas necesitan de la colaboración del “contenedor”

    Búsqueda(*lookup*): le pedimos al contenedor el objeto necesario, usando un nombre abstracto, para flexibilizar

    Inyección de dependencias: el contenedor llama a nuestro código, inicializando el objeto

Inyección de dependencias

- Hay que confiar en que Spring llamará al setGestorAlmacen al instanciar un objeto de la clase GestorPedidos

```
public class GestorPedidos {
        private GestorAlmacen gestorAlmacen;

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

        public void realizarPedido(Pedido p) {
            if (gestorAlmacen.hayStock(p))
                ...
        }
    }
```

Definir los *beans*

- Un bean en Spring es un componente de negocio "gestionado"
  - Normalmente relacionado con otros componentes (dependencias)
- Hay dos opciones para la configuración
  - **XML**: la clásica. Tediosa, pero al ser independiente del código fuente nos da más flexibilidad y elimina la recompilación
  - **Anotaciones**: mucho más *cool* y más sencilla de usar

Ficheros de definición de beans

- La manera “clásica” de definir beans es con XML
- **Siempre** tiene que haber al menos un fichero XML
  - Ciertas cosas hay que hacerlas en XML
  - O al menos, con XML decirle a Spring que usaremos anotaciones para hacer cierta tarea

```
<?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-2.5.xsd">

  <bean id="miGestor"
            class="es.ua.jtech.spring.negocio.GestorUsuarios">
  </bean>


</beans>
```

Definir un bean con anotaciones

- Spring ofrece:
  - **@Service**: componente de negocio
  - **@Repository**: DAO
  - **@Component**

```
package es.ua.jtech.spring.negocio;

@Service("miGestor")
public class GestorUsuarios {
   public UsuarioTO login(String login, String password) {
      ...
   }
}
```

¿Y en el fichero XML...?

- Le “decimos” a Spring que examine automáticamente **todos los subpaquetes** de algún paquete en busca de beans

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:component-scan
              base-package="es.ua.jtech.spring"/>
</beans>
```

¿Y para acceder al bean desde mi código?

- **Caso 1: Desde otro bean (u otro objeto gestionado)**: inyección de dependencias con @Resource
  - Se pone el nombre dado al definir el componente

```
package es.ua.jtech.spring.negocio;

@Service("miGestor")
public class GestorUsuarios {
   @Resource("miUDAO")
   private UsuariosDAO udao;

   ...
}
```

```
package es.ua.jtech.spring.datos;

@Repository("miUDAO")
public class UsuariosDAO {
   public UsuarioTO getUsuario(String id) {
   ...
}
```

¿Y para acceder al bean desde mi código?

- **Caso 2: desde un objeto no gestionado**: escribir código que busque el objeto
  - El API cambia si es una aplicación web o de escritorio
  - Si no usamos el módulo Spring MVC, **servlets y JSP no son gestionados** por Spring

En un servlet o JSP, sin Spring MVC

```
ServletContext sc = getServletContext();
WebApplicationContext wac =
        WebApplicationContextUtils.getWebApplicationContext(sc);
GestorUsuarios gu = (GestorUsuarios) wac.getBean("miGestor");
```

Referencia “por tipo”

- Referenciar un bean por nombre es propenso a errores. Si solo hay uno definido de un tipo (un *singleton*), no hay ambigüedad. En Spring esto se conoce como “Autowiring”
  - **@Autowired**. se puede colocar delante de la propiedad o bien delante del *setter* (o en un parámetro de un método cualquiera)

```
package es.ua.jtech.spring.negocio;

@Service(“miGestor”)
public class GestorUsuarios {
   @Autowired
   private UsuariosDAO udao;

   ...
}
```

```
package es.ua.jtech.spring.datos;

@Repository(“miUDAO”)
public class UsuariosDAO {
   public UsuarioTO getUsuario(String id) {
   ...
}
```

Ambigüedad por tipo

- Se puede resolver usando @Autowired + @Qualifier
- Supongamos un interface IUsuariosDAO y varias implementaciones de él

```
@Repository("JDBC")
public class UsuariosDAOJDBC implements IUsuariosDAO {
    public UsuarioTO login(String login, String password) {
        //Aquí vendría la implementación JDBC...
    }
}
```

```
@Service("miGestor")
public class GestorUsuarios {
   @Autowired
   @Qualifier("JDBC")
   private IUsuariosDAO udao;
}
```

Todo esto también se puede hacer en XML...

- ...pero es más “doloroso”. Por ejemplo el *autowiring*

```
<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-2.5.xsd">

   <bean id="miUsuariosDAO"
               class="es.ua.jtech.spring.datos.UsuariosDAO">
   </bean>

   <bean id ="miGestorUsuarios"
               class="es.ua.jtech.spring.negocio.GestorUsuarios"
               autowire="byType">
   </bean>
</beans>
```

- *A cambio:*
  - *Independiente del código fuente*
  - *Podemos definir varios beans de la misma clase*

Spring puede inicializar las propiedades del bean

- Solo tiene sentido en XML

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"...
  <bean id="misPrefs" class="springbeans.PrefsBusqueda">
    <property name="maxResults" value="100"/>
    <property name="idioma" value="es"/>
    <property name="ascendente"  value="true"/>
  </bean>
</beans>
```

Ámbito de los beans

- Por defecto los beans son *singleton*
  - Apropiado para equivalentes a EJB stateless y para DAOs, que no suelen guardar estado
- Pero hay otros ámbitos:
  - prototype: cuando se inyecta o busca un bean, siempre es nuevo
  - en aplicaciones web: session, request

```
package es.ua.jtech.spring.negocio;

@Service("miGestor")
@Scope("prototype")
public class GestorUsuarios {
   ...
}
```

Dependencias entre beans de distinto ámbito

- Supongamos un *singleton* que contiene una referencia a un *session*. ¿Qué pasa cuando se borra la sesión?
  - La inyección se hace al instanciar el bean dependiente (el que contiene la referencia), luego la referencia deja de ser válida cuando “muere” el bean de session

```
package es.ua.jtech.spring.negocio;

@Service(“miGestor”)
public class GestorUsuarios {
   @Autowired
   private Preferencias prefs;
}
```

```
package es.ua.jtech.spring.negocio;

@Service
@Scope(“session”)
public class PrefsBusqueda {
    private int maxResults;
    ...
}
```

Dependencias entre beans de distinto ámbito (II)

- **Solución**: inyectar un objeto *proxy*. El proxy tiene el mismo interfaz que éste, pero no “sabe hacer nada”, delega las llamadas en el objeto. Lo que sí sabe hacer es **gestionar el ciclo de vida** del objeto
- Usamos el XML para decirle a Spring que use *proxies* AOP

```
...
<context:component-scan base-package="es.ua.jtech.spring"
    scoped-proxy="targetClass"/>
...
```

Enlazar JNDI con beans

- **Etiqueta jndi-lookup**. Si estamos en un contenedor, hay que poner resource-ref=true (añade el prefijo java:comp/env)

```
<?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"...>

  <context:component-scan base-package="es.ua.jtech.spring"/>
  <jee:jndi-lookup id=”mids” jndi-name="jdbc/MiDataSource"
                           resource-ref=”true”/>
</beans>
```

```
package es.ua.jtech.spring.datos;
@Repository
public class UsuariosDAO {
  @Autowired
  DataSource ds;
}
```

¿Preguntas...?