Spring

Sesión 1: Spring core

Puntos a tratar

- Introducción. Spring vs. JavaEE estándar
- El contenedor de beans (Spring core)
- Trabajo con beans
  - Definir beans
  - Instanciar beans
  - Ámbitos
- Acceder a recursos externos con beans
- Definir beans en XML y Java

¿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 JavaEE basada en EJBs**
- Actualmente es un ***framework*** **completo** que cubre todas las capas de la aplicación, con decenas de desarrolladores y miles de descargas al día
  - MVC
  - Negocio (donde empezó originalmente)
  - Acceso a datos

spring source

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
  - Inspirada en la forma de trabajar de Spring y otros *frameworks*
  - Irónicamente, el principal problema de Spring pueden ser sus buenas ideas

La filosofía de Spring

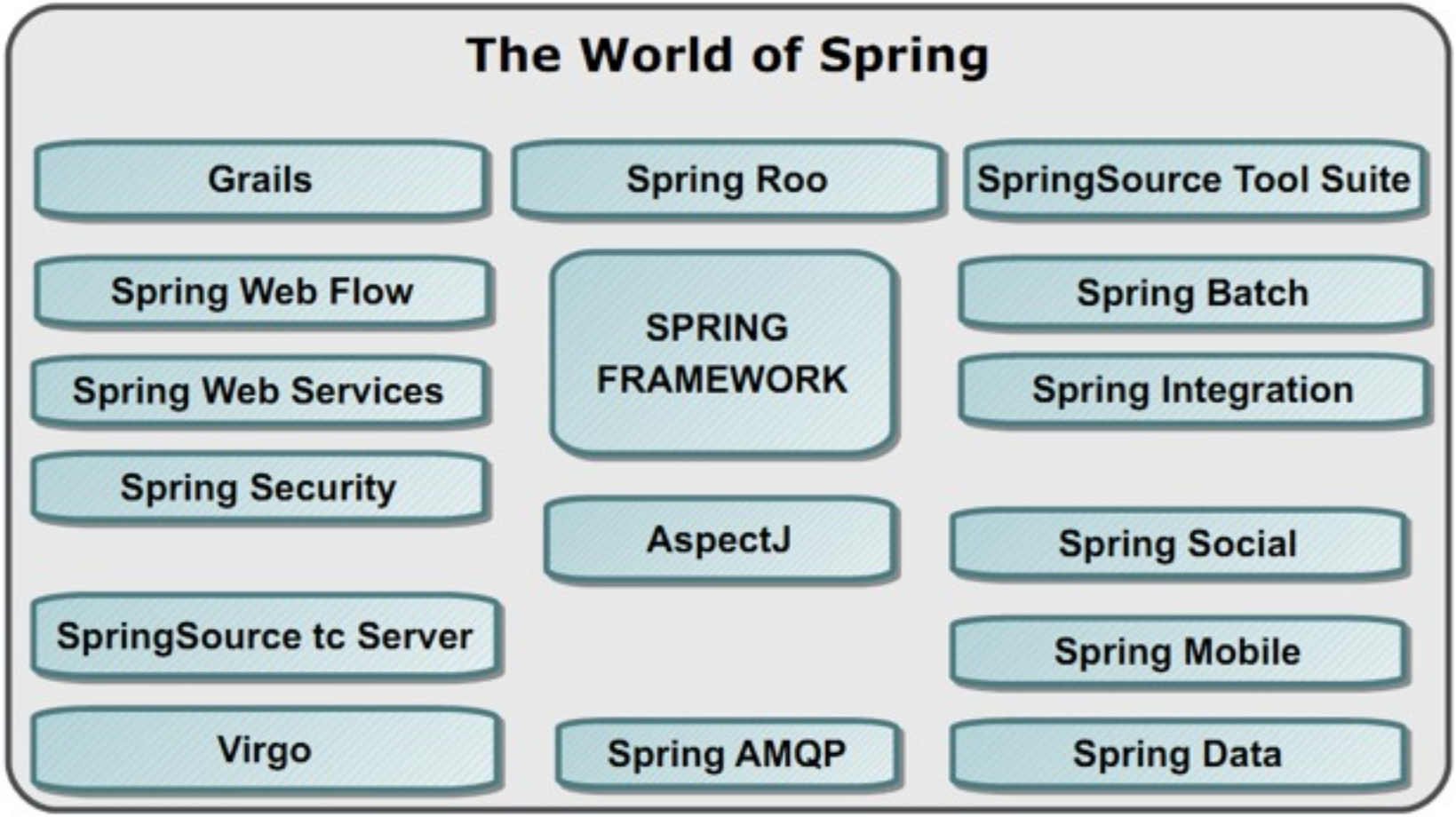
- Los objetos de negocio deberían ser POJOs
- Inyección como medio de resolver dependencias
- Se pueden conseguir servicios equivalentes a los que nos dan los EJB usando AOP de manera casi transparente al programador
- El contenedor debe poder ser un servidor web convencional (ej. Tomcat)
- Cuando ya hay algo que funciona, incorpóralo a tu solución, no “reinventes la rueda”
  - JPA , Hibernate para persistencia de datos
  - AspectJ para AOP
  - Hessian para acceso remoto
  - ...

Qué nos aporta Spring

+

Blueprints

El "ecosistema" de Spring



Versiones de Spring

- La generación actual es la 3
  - En la 2.X, se **potenció** bastante el *framework* y se **simplificó** mucho su **configuración**
  - Cada vez más énfasis en el **uso de anotaciones**
  - Las novedades de la versión 3 están sobre todo en la capa web (validación, REST)
- Spring tiene una excelente documentación
  - Hay mucha bibliografía adicional disponible y también una amplia comunidad de desarrollo

Spring vs. JavaEE estándar

- Hace unos años Spring tenía una ventaja tecnológica muy considerable
- Actualmente los dos "contendientes" son muy similares
- Ventajas de Spring:
  - Despliegue en servidores convencionales (Tomcat)
  - "Portabilidad"
- Ventajas del estándar
  - Es estándar ;-)

JavaOne 2011: Migrating Spring Applications to Java EE 6

Spring migration analyzer (JavaEE -> Spring)

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

Spring vs. factorías

- Normalmente un objeto de negocio necesita la colaboración de otros objetos de negocio, DAOs, etc.
- En código java “puro” la forma estándar de instanciar un objeto es la factoría (para independizarnos de la implementación)

```
public class JPATest {
  private EntityManagerFactory emf;

  public void test1() {
    emf = Persistence.createEntityManagerFactory("simplejpa");
    EntityManager em = emf.createEntityManager();
    …
  }
}
```

- Spring core es una alternativa a las factorías basada en la **inyección de dependencias**

Inyección de dependencias

- "Confiamos" en que "alguien" instancie y nos pase el objeto que necesitamos para trabajar
  - En nuestro caso ese "alguien" es Spring Core

```
public class JPATest {
  private EntityManagerFactory emf;

  public void test1() {
    emf = Persistence.createEntityManagerFactory("simplejpa");
    EntityManager em = emf.createEntityManager();
  }

  public void setEmf(EntityManagerFactory emf) {
   this.emf = emf;
  }
}
```

- Si se configura adecuadamente, Spring nos asegura que el setter se llamará justo después de crear el JPATest
  - Pero para eso el JPATest también lo tiene que crear Spring

Definir los *beans*

- Un bean en Spring es un componente cuyo ciclo de vida está "gestionado" por Spring
  - No se crea con new, sino que lo crea Spring Core y lo obtenemos con inyección de dependencias
  - Normalmente relacionado con otros componentes (dependencias)
- Hay tres 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
  - **Java**: podemos usar el chequeo de código de nuestro IDE para validar la configuración

Ficheros de definición de beans

- La manera “clásica” de definir beans es 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.xsd">

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

</beans>
```

- Aunque no la usemos, **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

Definir un bean con anotaciones

- Spring ofrece varios estereotipos:
- O la estándar de JSR330: @Named
  - **@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 (otro objeto gestionado)**: inyección de dependencias. También podríamos usar el @Inject/@Named estándar

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

@Service
public class UsuariosBO implements IUsuariosBO{
   @Autowired
   private IUsuariosDAO iudao;

   ...
}
```

"Programar contra interfaces, no implementaciones"

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

@Repository
public class UsuariosDAO implements IUsuariosDAO{
   public Usuario getUsuario(String id) {
   ...
}
```

Spring soporta las anotaciones JSR330

- Preferible si queremos mayor portabilidad
- Pero no soporta las del estándar CDI completo, JSR299

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

@Named
public class UsuariosBO implements IUsuariosBO{
   @Inject
   private IUsuariosDAO iudao;

   ...
}
```

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

@Named
public class UsuariosDAO implements IUsuariosDAO{
   public Usuario getUsuario(String id) {
   ...
}
```

¿Y para acceder al bean desde mi código? (II)

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

```
//En un servlet o JSP, sin Spring MVC
ServletContext sc = getServletContext();
WebApplicationContext wac =
        WebApplicationContextUtils.getWebApplicationContext(sc);
IUsuariosBO ubo =  wac.getBean(IUsuariosBO.class);
```

Inyección en setters o constructores

- En lugar de la variable miembro también se puede anotar un setter, un constructor o un método cualquiera

```
@Service
public class UsuariosBOSimple implements IUsuariosBO {
  IUsuariosDAO udao;

  @Autowired
  public setUsuariosDAO(UsuariosDAO udao) {
    this.udao = udao;
  }
```

- Ventaja: podemos resolver las dependencias “manualmente” (útil p.ej. para tests)

```
IUsuariosBO ubo = new UsuariosBOSimple();
ubo.setUsuariosDAO(new UsuariosDAOJPA());
```

Referencia “por nombre”

- @Autowired se puede usar si solo hay un bean definido de un tipo
- Si hubiera por ejemplo dos implementaciones distintas de IUsuariosDAO (UsuariosDAOJDBC, UsuariosDAOJPA), se puede inyectar por nombre del 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(“miUBO”)
public class UsuariosBO {
   @Resource(name=”miUDAO”)
   private IUsuariosDAO iudao;

   ...
}
```

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

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

Otra forma de resolver la ambigüedad

- usando @Autowired + @Qualifier

```
@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;
}
```

Ámbito de los beans

- Por defecto los beans son *singleton*
  - Apropiado por ejemplo 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 {
   ...
}
```

Acceso a recursos externos con beans

- En JavaEE, el estándar para acceder a recursos externos a una aplicación es JNDI
  - Conexión con bases de datos, colas de mensajes, servidores de correo,...
- JNDI no usa inyección de dependencias. A cada recurso se le asigna un nombre lógico y tenemos un API que nos permite buscar el recurso por nombre
- Etiqueta **jndi-lookup**: "convierte" un recurso JNDI en un bean de Spring

```
<jee:jndi-lookup id="miBean" jndi-name="jdbc/MiDataSource"
        resource-ref="true"/>
```

- Ahora ya podemos inyectarlo

```
@Repository
public class UsuariosDAOJDBC implements IUsuariosDAO {
  @Resource(name="miBean")
  DataSource ds;
  ...
}
```

Mejorando el ejemplo de JNDI

- Externalizamos el nombre JNDI del XML

```
<jee:jndi-lookup id="miBean" jndi-name="${datasource}"
        resource-ref="true"/>
<context:property-placeholder
        location="classpath:datasource.properties"/>
```

- Spring es lo suficientemente “listo” para saber que el recurso JNDI anterior es un DataSource, por tanto podemos inyectar con @Autowired

```
@Repository
public class UsuariosDAOJDBC implements IUsuariosDAO {
  @Autowired
  DataSource ds;
  ...
}
```

Inicialización de los beans

- Por defecto la inicialización es cuando se arranca el contenedor de Spring, es decir, cuando se arranca la aplicación
  - Normalmente veremos que nada más arrancar se crea una instancia de cada bean (recordemos, ámbito por defecto)
- Podemos hacer que la inicialización se haga cuando se intente acceder al bean

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 ="miUsuariosBO"
               class="es.ua.jtech.spring.negocio.UsuariosBO"
               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 o en Java

```
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>
```

Configuración Java

- Una clase anotada con @configuration es como un fich. de config., no es código normal
- Cada bean se define con un método anotado con @Bean.
  - El nombre del método es el nombre que Spring le da al bean
  - Dentro del método debemos construir y devolver el bean

```
@Configuration
public class SampleConfig {
   @Bean
   public IUsuariosDAO udao() {
     return new UsuariosDAOJPA();
   }

   @Bean
   public IUsuariosBO ubo() {
     IUsuariosBO ubo = new UsuariosBOSimple();
     ubo.setCredito(100);
     ubo.setIUsuariosDAO(udao());
     return ubo;
   }
}
```

¿Preguntas...?