Acceso remoto. Pruebas

Índice

1. Acceso remoto

Uno de los puntos que hacen atractivos a los EJB es que permiten simplificar la programación distribuida, proporcionando un acceso a objetos remotos "casi transparente" para el programador. Se puede localizar de manera sencilla un objeto remoto que reside en otra máquina cualquiera y llamar a sus métodos como si el objeto estuviera en la máquina local. En esta sección veremos alternativas para conseguir el mismo objetivo, que si bien no son tan sofisticadas, son más ligeras que la implementación de muchos contenedores de EJBs.

1.1. Evaluación de las alternativas

Spring no proporciona una única alternativa a EJBs para acceso remoto. Según los requerimientos de la aplicación y las características de la implementación será más apropiada una u otra alternativa. Veamos cuáles son:

- **RMI:** A pesar de que usar RMI directamente pueda parecer algo "primitivo", Spring implementa una serie de clases que proporcionan una capa de abstracción sobre el RMI "puro", de modo que por ejemplo no hay que gestionar directamente el servidor de nombres, ni ejecutar manualmente `rmic` y el cliente puede abstraerse totalmente de que el servicio es remoto. RMI será la alternativa adecuada cuando nos interese **buen rendimiento, clientes Java y sepamos que el servidor de nombres no es un problema (p.ej. con firewalls)**
- **HTTP invoker:** Es una opción muy similar a la de RMI, usando serialización de Java, pero a través del puerto HTTP estándar, con lo que eliminamos los posibles problemas de firewalls. Nótese que el cliente también debe ser Spring, ya que el protocolo está implementado en librerías propias del framework (un cliente RMI podría ser no-Spring). Será apropiado cuando nos interese **buen rendimiento, clientes Spring y tengamos posibles problemas con los puertos permitidos.** .
- **Protocolos Hessian y Burlap:** son protocolos que funcionan a través del puerto HTTP estándar. Hessian es binario y Burlap XML, por lo que el primero es más eficiente. Teóricamente pueden funcionar con clientes no-Java, aunque con ciertas limitaciones. No son protocolos originalmente diseñados en el seno de Spring, sino de una empresa llamada Caucho (aunque son también *open source*). Será interesante cuando queramos **buen rendimiento, clientes no-Java y tengamos posibles problemas con los puertos permitidos.**
- **Servicios web SOAP:** indicados para el acceso a componentes remotos en plataformas heterogéneas (con clientes escritos en casi cualquier lenguaje). Su punto débil es básicamente el rendimiento: la necesidad de transformar los mensajes que intercambian cliente servidor a formato neutro en XML hace que sean poco eficientes, pero es lo que al mismo tiempo los hace portables. Serán apropiados cuando **El rendimiento no sea crítico, y queramos la máxima portabilidad en cuanto a clientes.**

- **Servicios web REST:** Superan la "pesadez" de sus hermanos SOAP gracias al uso de protocolos ligeros como HTTP. Como contrapartida, al carecer en general de una especificación formal tendremos que programar manualmente el servicio, tanto en la parte cliente como servidor. Serán los indicados si queremos **máxima portabilidad en cuanto a clientes y el API del servicio es simple y fácil de programar manualmente.**

Limitaciones del acceso remoto en Spring

Nótese que en el acceso remoto a componentes los EJBs siguen teniendo ciertas ventajas sobre Spring, en particular la propagación remota de transacciones, que no es posible en Spring. Es el precio a pagar por poder usar un contenedor web java convencional como Tomcat en lugar de un servidor de aplicaciones completo.

Discutiremos a continuación con más detalle las características de estas alternativas y cómo usarlas y configurarlas dentro de Spring. Obviaremos los servicios web SOAP, que por su complejidad quedan fuera del ámbito posible para esta sesión.

En todos los casos vamos a usar el siguiente ejemplo, muy sencillo, de componente al que deseamos acceder de forma remota, con su interfaz:

```
package servicios;

public interface ServicioSaludo {
   public String getSaludo();
}
```

```
package servicios;

@Service("saludador")
public class ServicioSaludoImpl implements ServicioSaludo {
   String[] saludos = {"hola, ¿qué tal?", "me alegra verte", "yeeeeeey"};

   public String getSaludo() {
         int pos = (int)(Math.random() * saludos.length);
         return saludos[pos];
   }
}
```

1.2. RMI en Spring

Aunque el uso directo de RMI puede resultar tedioso, Spring ofrece una capa de abstracción sobre el RMI "puro" que permite acceder de forma sencilla y casi transparente a objetos remotos.

Usando la clase `RmiServiceExporter` podemos exponer la interfaz de nuestro servicio como un objeto RMI. Se puede acceder desde el cliente usando RMI "puro" o bien, de modo más sencillo con un `RmiProxyFactoryBean`.

La configuración en el lado del servidor quedará como sigue:

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <!-- el nombre del servicio es arbitrario.
    Sirve para referenciarlo desde el cliente -->
    <property name="serviceName" value="miSaludador"/>
    <!-- el servicio es el bean que hacemos accesible -->
    <property name="service" ref="saludador"/>
    <!-- el bean debe implementar un interface -->
    <property name="serviceInterface" value="servicios.ServicioSaludo"/>
    <!-- Cómo cambiar el puerto para RMI (por defecto es el 1099) -->
    <property name="registryPort" value="1199"/>
</bean>
```

En la configuración anterior se ha cambiado el puerto del servidor de nombres RMI para evitar posibles conflictos con el del servidor de aplicaciones. A partir de este momento, el objeto remoto es accesible a través de la URL `rmi://localhost:1199/miSaludador`.

La configuración en el cliente quedaría como sigue:

```
<bean id="saludadorRMI"
        class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="serviceUrl" value="rmi://localhost:1199/miSaludador"/>
    <property name="serviceInterface" value="servicios.ServicioSaludo"/>
</bean>
```

Nótese que el id del bean es arbitrario, aunque luego habrá que referenciarlo en el código del cliente. Los valores de las propiedades `serviceUrl` y `serviceInterface` evidentemente tienen que coincidir con los dados en la configuración en el servidor.

Para acceder al objeto de forma remota todo lo que necesitaríamos es tener en el cliente el código del interfaz `ServicioSaludo` y usar el siguiente código Java para conectar:

```
ClassPathXmlApplicationContext contexto =
         new ClassPathXmlApplicationContext("clienteRMI.xml");
ServicioSaludo ss =  contexto.getBean(ServicioSaludo.class);
System.out.println(ss.getSaludo());
```

Suponiendo que el fichero de configuración en el cliente lo hemos llamado `clienteRMI.xml`

1.3. Hessian y Burlap

Hessian y Burlap son dos protocolos diseñados originalmente por la empresa Caucho, desarrolladora de un servidor de aplicaciones J2EE de código abierto llamado `Resin`. Ambos son protocolos para acceso a servicios remotos usando conexiones HTTP estándar. La diferencia básica entre ambos es que Hessian es binario (y por tanto más eficiente que Burlap) y este es XML (y por tanto las comunicaciones son más sencillas de depurar). Para ambos también se han desarrollado implementaciones en distintos lenguajes de manera que el cliente de nuestra aplicación podría estar escrito en C++, Python, C#, PHP u otros.

1.3.1. Uso de los protocolos

Usaremos Hessian en el siguiente ejemplo, aunque la configuración de Burlap es prácticamente idéntica. Hessian se comunica mediante HTTP con un servlet . Por tanto el primer paso será crear dicho servlet en nuestro `web.xml`. Nos apoyaremos en la clase `DispatcherServlet` propia de Spring, ya que se integra de manera automática con el resto de elementos de nuestra configuración. A continuación se muestra el fragmento significativo del `web.xml`.

```
<servlet>
   <servlet-name>remoting</servlet-name>
   <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
   <servlet-name>remoting</servlet-name>
   <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>
```

Esto hace accesible el servlet a través de la URL `http://localhost:8080/remoting` (si por ejemplo usamos Tomcat cuyo puerto por defecto es el 8080). Recordemos del tema de MVC que en Spring, la configuración de un `DispatcherServlet` se debe guardar en un xml con nombre *nombreDelServlet*-servlet.xml (en nuestro caso `remoting-servlet.xml`). Aclarar que aunque podríamos hacer uso del mismo DispatcherServlet para gestionar tanto los controllers MVC como el acceso remoto, no es necesario, podemos usar una instancia distinta para cada cosa.

```
<bean name="/saludador"
class="org.springframework.remoting.caucho.HessianServiceExporter">
  <property name="service" ref="saludador"/>
  <property name="serviceInterface" value="servicios.ServicioSaludo"/>
</bean>
```

Aquí hacemos uso de la clase `HessianServiceExporter`, que nos permite exportar de forma sencilla un servicio Hessian. En nuestro caso estará accesible en la URL `http://localhost:8080/contexto-web/remoting/saludador`. Nos falta la configuración del cliente y el código que llama al servicio:

```
<bean id="miSaludador"
class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
        <property name="serviceUrl"
                value="http://localhost:8080/contexto-web/remoting/saludador"/>
        <property name="serviceInterface"
value="servicios.ServicioSaludo"/>
</bean>
```

Suponiendo que el código XML anterior se guarda en un fichero llamado `clienteHessian.xml`

```
ClassPathXmlApplicationContext contexto =
        new ClassPathXmlApplicationContext("clienteHessian.xml");
ServicioSaludo ss =  (ServicioSaludo) contexto.getBean("miSaludador");
System.out.println(ss.getSaludo());
```

En el ejemplo anterior bastaría con escribir `Burlap` allí donde aparece `Hessian` y todo debería funcionar igual, pero ahora usando este procolo basado en mensajes XML.

1.3.2. Autentificación HTTP con Hessian y Burlap

Al ser una conexión HTTP estándar podemos usar autentificación BASIC. De este modo podemos usar la seguridad declarativa del contenedor web también para controlar el acceso a componentes remotos. En la configuración del servidor podríamos añadir el siguiente código:

```
<!-- cuidado con el copiar/pegar,
el nombre de la clase está partido -->
<bean class="org.springframework.web.servlet.
        handler.BeanNameUrlHandlerMapping">
   <property name="interceptors">
      <list>
         <ref bean="authorizationInterceptor"/>
      </list>
   </property>
</bean>

<!-- cuidado con el copiar/pegar,
el nombre de la clase está partido -->
<bean id="authorizationInterceptor"
   class="org.springframework.web.servlet.
   handler.UserRoleAuthorizationInterceptor">
   <property name="authorizedRoles">
    <list>
      <value>admin</value>
      <value>subadmin</value>
    </list>
   </property>
</bean>
```

Usando AOP añadimos un interceptor que resulta ser de la clase `UserRoleAuthorizationInterceptor`. Dicho interceptor solo permite el acceso al bean si el usuario resulta estar en uno de los roles especificados en la propiedad `authorizedRoles`. El `BeanNameUrlHandlerMapping` es el objeto que "tras las bambalinas" se encarga de asociar los beans que comienzan con "/" con los servicios en la URL del mismo nombre (en nuestro caso el bean "/saludador").

1.4. HTTP invoker

Esta es una implementación propia de Spring, que utiliza la serialización estándar de Java para transmitir objetos a través de una conexión HTTP estándar. Será la opción a elegir cuando los objetos sean demasiado complejos para que funcionen los mecanismos de serialización de Hessian y Burlap.

La configuración es muy similar al apartado anterior, podemos usar el mismo `DispatcherServlet` pero ahora para el acceso al servicio se debe emplear la clase `HttpInvokerServiceExporter` en lugar de `HessianServiceExporter`

```
<bean name="/saludadorHTTP"
class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
  <property name="service" ref="saludador"/>
  <property name="serviceInterface" value="servicios.ServicioSaludo"/>
</bean>
```

En la parte del cliente la definición del bean sería:

```
<bean id="httpProxy"
class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  <property name="serviceUrl"
value="http://localhost:8080/MiAplicacion/remoting/saludadorHTTP"/>
  <property name="serviceInterface" value="servicios.ServicioSaludo"/>
</bean>
```

Por defecto en el cliente se usan las clases estándar de J2SE para abrir la conexión HTTP. Además Spring proporciona soporte para el `HttpClient` de Jakarta Commons. Bastaría con poner una propiedad adicional en el `HttpInvokerProxyFactoryBean`:

```
<bean id="httpProxy"
class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  ...
  <property name="httpInvokerRequestExecutor">
    <!-- cuidado con el copiar/pegar,
    el nombre de la clase está partido -->
    <bean class="org.springframework.remoting.httpinvoker.
             CommonsHttpInvokerRequestExecutor"/>
  </property>
  ...
</bean>
```

2. Pruebas

En cualquier metodología moderna de ingeniería de software las pruebas son un elemento fundamental. Reconociendo este hecho, Spring nos da soporte para poder implementar nuestras pruebas del modo más sencillo posible. Otro aspecto importante, sobre todo en prubeas de integración es trabajar en un entorno lo más parecido posible a cómo se

ejecutará el código en producción. En Spring podemos hacer que las dependencias entre objetos se satisfagan automáticamente en pruebas del mismo modo que en producción.

Vamos a usar aquí el soporte para JUnit 4 y superior, que es la recomendada en la versión actual de Spring. El soporte de Junit 3 está *deprecated* y se eliminará en futuras versiones.

2.1. Pruebas unitarias

Supongamos que en la capa de acceso a datos de nuestra aplicación tenemos un DAO que implementa este interfaz:

```
public interface IUsuariosDAO {
    public List<Usuario> listar();
    public Usuario getUsuario(String login);
    public void alta(Usuario u);
    ...
}
```

y que queremos escribir código de prueba para él. Lo primero que nos da Spring es la posibilidad de usar un fichero de configuración de beans para pruebas diferente al fichero de producción. El esqueleto de la clase que prueba el DAO podría ser algo como:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:config/daos-test.xml")
public class UsuariosDAOTest  {
  //Spring nos da la instancia del DAO a probar
  @Autowired
  IUsuariosDAO dao;

  //Esto ya no tiene nada de particular de Spring
  @Test
  public void testListar() {
    List<Usuario> lista = dao.listar();
    assertEquals(10, lista.size());
  }
}
```

La anotación `@RunWith` es la que nos "activa" el soporte de testing de Spring. Con `@ContextConfiguration` especificamos el fichero o ficheros de configuración de beans que vamos a usar para las pruebas. Aquí podemos configurar lo que queremos que sea distinto de producción, por ejemplo, que los dataSources conecten con bases de datos de prueba, que los objetos de los que dependemos sean *mocks*, etc.

Profiles

A partir de la versión 3.1 de Spring hay una forma alternativa de hacer esto usando lo que se denominan *profiles*. La idea es que podemos hacer que ciertos beans se definan dentro de un determinado perfil (p.ej. "test") y ciertos otros en uno distinto (p.ej. "produccion"). Usando diferentes mecanismos podemos especificar qué perfil queremos usar en un momento dado. Se recomienda consultar la documentación de Spring para más información.

Nótese que la referencia al DAO que queremos probar nos la da el propio Spring, lo que

es una forma cómoda de instanciar el objeto, y además ofrece la ventaja adicional de que si este dependiera de otros dichas dependencias se resolverían automáticamente, lo que nos facilita las pruebas de integración, como veremos después.

¿Qué tendría de especial nuestro fichero de configuración para pruebas?. Como ya hemos dicho, lo más habitual es definir la conexión con la base de datos de prueba en lugar de con la real. Yendo un paso más allá y para agilizar las pruebas, podríamos usar una base de datos embebida en lugar de usar simplemente otras tablas en el mismo servidor de bases de datos de producción:

```
<?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:context="http://www.springframework.org/schema/context"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xmlns:jdbc="http://www.springframework.org/schema/jdbc"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.2.xsd
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-3.2.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.2.xsd">

    <!-- conexión con la BD de pruebas -->
    <jdbc:embedded-database id="miDataSource">
        <jdbc:script location="classpath:db.sql"/>
        <jdbc:script location="classpath:testdata.sql"/>
    </jdbc:embedded-database>

    <!-- los DAOs están en este package -->
    <context:component-scan base-package="es.ua.jtech.spring.datos"/>

    <!-- soporte de transacciones en las pruebas -->
    <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
      <!-- importante: este "ref" coincide con el "id" del dataSource -->
      <property name="dataSource" ref="miDataSource"/>
    </bean>
</beans>
```

En el fichero anterior estaríamos definiendo un DataSource Spring que conectaría con una base de datos embebida, cuyos datos estarían en los dos scripts SQL a los que se hace referencia (supuestamente uno para crear las tablas y otro para insertar los datos). Spring usa HSQLDB como base de datos embebida por defecto, aunque podemos cambiar la configuración para usar otras. Por supuesto necesitaremos incluir la dependencia correspondiente en el pom.xml (suponiendo HSQLDB):

```
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <version>2.0.0</version>
  <scope>test</scope>
</dependency>
```

Por otro lado, vemos que en el fichero se define un "transaction manager". Este se usa como soporte para gestionar la transaccionalidad de las pruebas. Un problema muy típico de las pruebas de acceso a datos es que la ejecución de una prueba altera el contenido de la base de datos y quizás interfiere con las pruebas siguientes. Esto nos obliga a restablecer los datos iniciales tras cada prueba, por ejemplo ejecutando el script SQL que inserta los datos. Una alternativa es el **soporte de transaccionalidad** que nos da Spring: podemos hacer un *rollback* automático de todos los cambios generados durante la prueba, lo que facilita mucho el trabajo.

Para que funcione esta transaccionalidad de las pruebas necesitamos un transaction manager en el fichero de configuración de pruebas al igual que es necesario para que funcione la transaccionalidad declarativa en producción. Una vez definido el gestor de transacciones, debemos anotar la clase de pruebas con `@TransactionConfiguration`. El valor del atributo "transactionManager" debe coincidir con el id del transaction manager definido en el fichero de configuración XML. Finalmente anotaremos con `@Transactional` los métodos de prueba para los que queramos hacer un *rollback* automático, o bien anotaremos la propia clase si queremos hacerlo en todos:

```
//modificación del test anterior
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:config/daos-test.xml")
@TransactionConfiguration(transactionManager = "txManager",
                          defaultRollback = true)
@Transactional
public class UsuariosDAOTest  {
...
```

2.2. Pruebas de integración. Uso de objetos mock

Supongamos que tenemos una capa de negocio en la que hay un "gestor de usuarios" con el siguiente interfaz:

```
public interface IUsuariosBO {
  //Este método debe comprobar que el password coincide con lo que
devuelve el DAO
  public Usuario login(String login, String password);
  //Estos métodos delegan el trabajo en el DAO
  public List<Usuario> listar();
  public void alta(Usuario u);
}
```

Las implementaciones de dicho interfaz hacen uso del DAO de usuarios:

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

  @Override
  public Usuario login(String login, String password) {
    Usuario u = udao.getUsuario(login);
```

```
    if (u!=null  && u.getPassword().equals(password))
      return u;
    else
      return null;
  }
}
```

Además de probar por separado el BO y el DAO nos va a interesar hacer pruebas de integración de ambos. En Spring si hacemos test de la capa BO automáticamente estamos haciendo pruebas de integración ya que el framework resuelve e instancia la dependencia BO->DAO.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath:config/daos-test.xml",
   "classpath:config/bos-test.xml"})
public class UsuariosBOTest {
  @Autowired
  IUsuariosBO ubo;

  @Test
  public void testLogin() {
    //el usuario "experto" sí está en la BD
    assertNotNull(ubo.login("experto", "experto"));
    //pero no el usuario "dotnet" (¡jamás!)
    assertNull(ubo.login("dotnet", "dotnet"));
  }
}
```

Como antes, simplemente le pedimos a Spring con @Autowired que nos inyecte el objeto que deseamos probar. No necesitamos instanciar el DAO, lo hará Spring. La única diferencia con el ejemplo anterior es que usamos un fichero de configuración adicional para la capa BO (en un momento veremos por qué nos podría interesar tener la capa DAO y BO en ficheros de configuración distintos). Nótese que cuando se usan varios, hay que hacerlo con notación de inicialización de array (entre llaves), y explicitando el nombre del atributo: "location".

Pero ¿qué ocurre si queremos hacer pruebas unitarias de la capa BO, sin que "interfiera" el comportamiento de los DAO?. Necesitamos entonces sustituir estos últimos por **objetos *mock***. Podríamos escribirlos nosotros mismos, pero en Java hay varios *frameworks* que nos ayudarán a generar estos objetos. Vamos a usar aquí mockito. El objetivo no es ver aquí cómo se usa este *framework* en sí, sino ver una breve introducción a cómo usarlo dentro de Spring.

Supongamos que queremos construir un *mock* del interfaz `IUsuariosDAO` y hacer que tenga un determinado comportamiento. El API básico de mockito es bastante simple:

```
import static org.mockito.Mockito.*;
//creamos el mock
IUsuariosDAO udaoMock = mock(IUsuariosDAO.class);
//Creamos un usuario de prueba con login "hola" y password "mockito"
Usuario uTest = new Usuario("hola", "mockito");
//grabamos el comportamiento del mock
when(udaoMock.getUsuario("hola")).thenReturn(uTest);
//imprime el usuario de prueba
```

```
System.out.println(udaoMock.getUsuario("hola"));
```

Como se ve en el código anterior, primero creamos el *mock* con la llamada al método estático `mock` y luego especificamos para la entrada indicada (`when`) qué salida debe proporcionar (`thenReturn`). Nuestro *mock* será una especie de "tabla" con respuestas predefinidas ante determinadas entradas.

Ahora para integrar el mock dentro de Spring tenemos que resolver dos problemas: el primero es cómo sustituir nuestra implementación de IUsuariosDAO por el mock. Definiremos un bean de Spring que se construya llamando al método `mock` de mockito. Vamos a hacerlo en XML, aunque también lo podríamos hacer con configuración Java (pero no con la anotación @Repository, ya que la clase a anotar no es nuestra, sino generada por mockito):

```
<?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:context="http://www.springframework.org/schema/context"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xmlns:jdbc="http://www.springframework.org/schema/jdbc"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.2.xsd
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-3.2.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.2.xsd">

  <bean id="usuarioDAO" class="org.mockito.Mockito" factory-method="mock">
    <constructor-arg value="es.ua.jtech.dao.IUsuariosDAO" />
  </bean>
</beans>
```

Ahora podemos ver por qué nos interesaba tener en ficheros de configuración distintos la capa DAO y BO. De este modo podemos tener un XML para la capa DAO de "producción" (que busque anotaciones @Repository digamos en es.ua.jtech.dao) y de pruebas de integración y este otro para pruebas con mocks.

El segundo problema a resolver es cómo especificar el comportamiento del *mock*. Aprovecharemos el `@Autowired` de Spring para acceder al objeto y el `@Before` para especificar su comportamiento. Si el bean es un *singleton*, el BO tendrá acceso a la misma instancia que hemos controlado nosotros y por tanto su comportamiento será el deseado.

```
@RunWith(SpringJUnit4ClassRunner.class)
//daos-mock-test.xml es el XML de configuración anterior,
//con la definición del mock de IUsuariosDAO
@ContextConfiguration(locations={"classpath:config/daos-mock-test.xml",
   "classpath:config/bos-test.xml"})
public class UsuariosBOMockTest {
  @Autowired
  IUsuariosBO ubo;
```

```
  //Esto nos dará acceso al mock
  @Autowired
  IUsuariosDAO udao;

  @Before
  public void setup() {
    when(udao.getUsuario("test")).thenReturn(new Usuario("test","test"));
    when(udao.getUsuario("test2")).thenReturn(new
Usuario("test2","test2"));
  }

  @Test
  public void testLogin() {
    //este usuario está "grabado" en el mock
    assertNotNull(ubo.login("test", "test"));
    //pero este no, por tanto el BO debería devolver null
    assertNull(ubo.login("experto", "experto"));
  }
}
```

2.3. Pruebas de la capa web

Una de las novedades de la versión 3.2 de Spring (la última en el momento de escribir estos apuntes) es la extensión del soporte de pruebas a la capa web. Sin necesidad de desplegar la aplicación en un contenedor web podemos simular peticiones HTTP y comprobar que lo que devuelve el controller, la vista a la que se salta o los objetos que se añaden al modelo son los esperados.

Hay dos posibilidades para ejecutar pruebas en la capa web, una de ellas es usar la misma configuración que se usa en producción y otra es establecer manualmente una configuración simplificada. Vamos a ver aquí la primera de ellas, ya que nos permite hacer pruebas más completas y más próximas a como se ejecuta el código dentro del contenedor web.

Veamos primero un ejemplo muy sencillo. Supongamos que tenemos el siguiente controller, que lo único que hace es devolver un mensaje de texto:

```
@Controller
public class HolaSpringController {
  @RequestMapping("/hola")
  public @ResponseBody String hola() {
    return "Hola Spring";
  }
}
```

Para las pruebas necesitamos una instancia de `MockMVC`, que se construye a partir del `WebApplicationContext` (el contexto de aplicación generado a partir del fichero de configuración de la capa web). Podemos pedirle a Spring una instancia de este último con `@Autowired`, así que no es excesivamente complicado construir el `MockMVC`:

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration(locations={"classpath:config/web-test.xml",
   "classpath:config/daos-test.xml", "classpath:config/bos-test.xml"})
```

```
public class HolaSpringControllerTest {
  @Autowired
  private WebApplicationContext wac;

  private MockMvc mockMvc;

  @Before
  public void setup() {
        this.mockMvc =
MockMvcBuilders.webAppContextSetup(this.wac).build();
  }

  //ahora vienen los test, un poco de paciencia...
  ...
}
```

Nótese que además necesitamos anotar la clase del test con `@WebAppConfiguration`, para indicarle a Spring que estamos usando un contexto de aplicación web.

El fichero de configuración de la capa web para testing (en el ejemplo, web-test.xml) puede ser el mismo de producción o uno simplificado. Como mínimo, si usamos anotaciones, deberá tener el context:component-scan. Además también debería tener un viewresolver, así podremos comprobar el nombre físico de la página a la que saltamos tras el controller.

El API para hacer las llamadas de prueba es bastante intuitivo. Por desgracia al ser una funcionalidad muy recientemente añadida a Spring (antes estaba en un proyecto aparte) la documentación de referencia todavía no es muy detallada y para algunas cosas hay que acudir al propio Javadoc del API. Vamos a ver aquí algunos ejemplos sencillos. Por ejemplo, para comprobar que la respuesta da un código de estado OK y que es la que deseamos, haríamos

```
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
...
...
@Test
public void testHola() throws Exception  {
  this.mockMvc.perform(get("/hola"))
    .andExpect(status().isOk())
    .andExpect(content().string("Hola Spring"));
}
```

como se ve, el esquema básico es que se hace una petición con `perform` y luego se espera una serie de resultados.

Vamos a ver varios ejemplos de cómo hacer peticiones y comprobar el resultado de la ejecución del controller. Para más información, se recomienda consultar la sección correspondiente de la documentación de Spring

2.3.1. Simular la petición

Podemos especificar el método HTTP a usar y el tipo de datos que aceptamos (el valor de la cabecera Accept):

```
mockMvc.perform(get("/usuarios/").accept(MediaType.APPLICATION_JSON));
```

Podemos usar URI templates como en REST

```
this.mockMvc.perform(put("/usuarios/{id}", 42)
    .content("{'login':'experto', 'password':'experto'}"));
```

O añadir parámetros HTTP a la petición

```
mockMvc.perform(get("/verUsuario").param("login", "experto"));
```

2.3.2. Comprobar el resultado

Ya hemos visto cómo comprobar el código de estado y el contenido de la respuesta. Veamos otros ejemplos. Podemos comprobar la vista a la que se intenta saltar:

```
this.mockMvc.perform(post("/login").param("login","experto").param("password",
"123456"))
  .andExpect(view().name("home"));
```

También podemos verificar que el modelo generado por el controller es correcto. Por ejemplo, supongamos que el controller debería haber añadido al modelo un atributo "usuario" con el usuario a ver, cuyos datos se muestran a través de la vista "datos_usuario"

```
this.mockMvc.perform(get("/usuarios/experto"))
   .andExpect(model().size(1))
   .andExpect(model().attributeExists("usuario"))
   .andExpect(view().name("datos_usuario"));
```

Podemos verificar una parte del contenido de la respuesta. Si es JSON, podemos usar la sintaxis de JsonPath para especificar la parte que nos interesa. Este ejemplo buscaría una propiedad llamada "localidad" con valor "alicante".

```
this.mockMvc.perform(get("/usuarios/experto").accept("application/json;charset=UTF-8"))
  .andExpect(status().isOk())
  .andExpect(content().contentType("application/json"))
  .andExpect(jsonPath("$.localidad").value("Alicante");
```

En caso de que la respuesta sea XML podemos usar el estándar xpath.

Podemos comprobar otros aspectos del resultado. El framework nos ofrece distintos `ResultMatchers`, que nos sirven para esto. Se recomienda consultar directamente el JavaDoc del API de Spring, en concreto el package `org.springframework.test.web.servlet.result`