Spring

Sesión 6: Acceso remoto. Pruebas

Indice

- **Spring para acceso remoto**
  - **Spring vs. EJB**
  - **Comparación de los protocolos disponibles**
- Acceso remoto con HttpInvoker
- Pruebas

Acceso remoto en Spring

- Hacer accesibles nuestros métodos y clases de negocio desde otras máquinas (no solo a través de la capa web)
- ¿Por qué acceso remoto?
  - Clientes ricos (Swing, etc)
  - Servicios web
  - Aplicaciones distribuidas
- Spring tiene ciertas limitaciones, relativas sobre todo a **transacciones distribuídas que impliquen objetos remotos**. Para eso necesitaremos EJBs

Opciones en Spring para acceso remoto

- RMI
  - Clientes Java, eficiente, problemas con firewalls
- Hessian y Burlap
  - Clientes en varios lenguajes, eficiencia media, *firewall friendly*
- HTTP invoker
  - Clientes Spring, eficiencia media, *firewall friendly*
- Servicios web SOAP
  - Estandarización, portabilidad máxima en cuanto a clientes, poca eficiencia, *firewall friendly*
- *Servicios web REST:*
  - *Superan el coste computacional de los SOAP pero a cambio requieren más trabajo manual*

REST vs Otros “protocolos”

- En Spring, RMI, HttpInvoker, Hessian y Burlap comparten una filosofía subyacente común
  - Tenemos un objeto remoto que nos da servicios (métodos) y accedemos a ellos como si fuera un objeto local
  - Misma filosofía que en EJBs, como ya veréis en la parte *enterprise*

- REST es distinto
  - Realizamos operaciones prefijadas con las entidades (Create/Read/Update/Delete)

Indice

- Spring para acceso remoto
  - Spring vs. EJB
  - Comparación de los protocolos disponibles
- **Acceso remoto con HttpInvoker**
- Pruebas

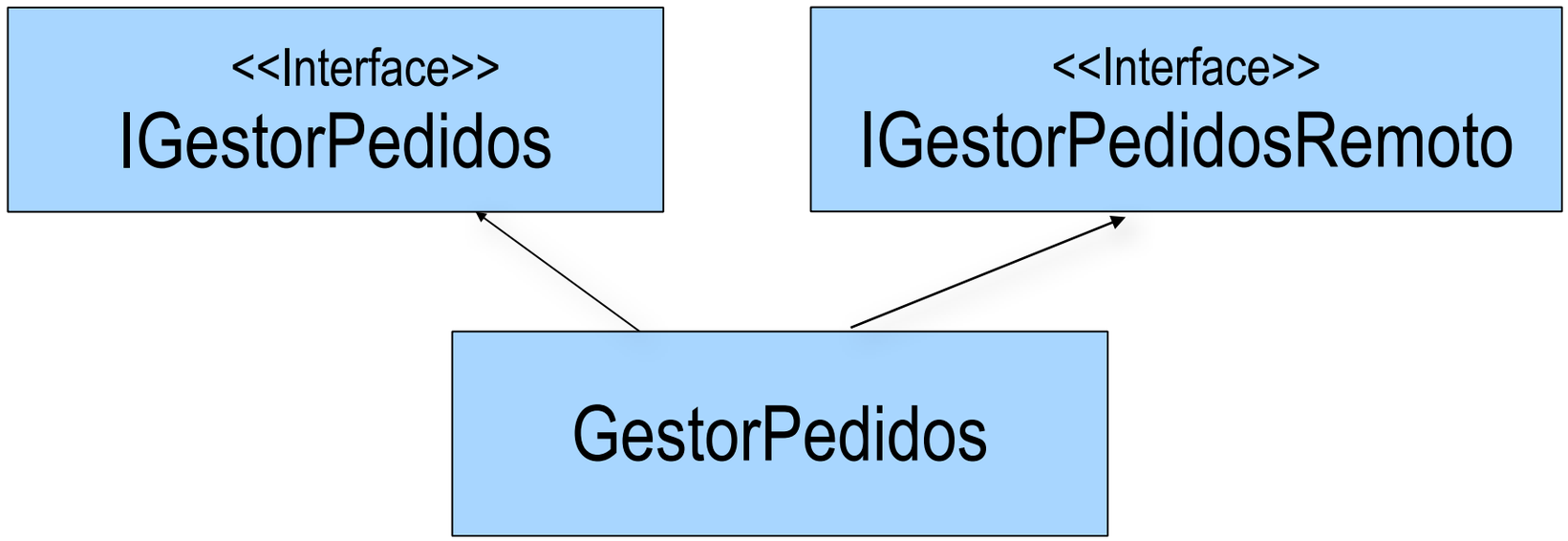
Conceptos básicos

- **Exporter**: en el **servidor**. Es un bean de Spring que nos permite exportar el servicio
  - Lo asociará a una URL (un “endpoint”)

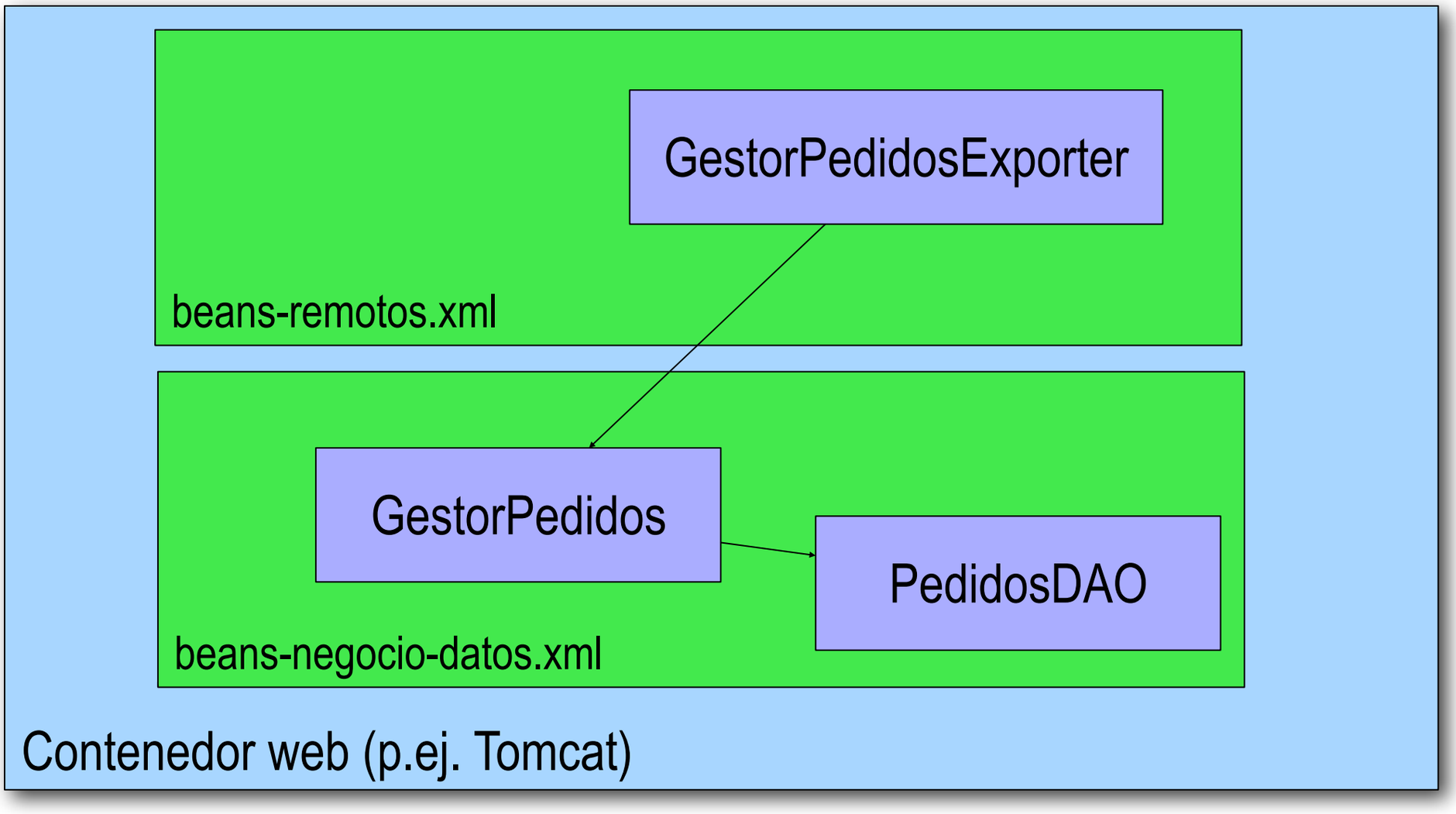
- **ProxyFactoryBean**: en el **cliente**. Es un bean de Spring que implementa el mismo interfaz al que queremos acceder y que actúa de proxy
  - Llamamos a un objeto local y él delega esa llamada en un objeto remoto, escondiendo los detalles de la comunicación

Muy importante

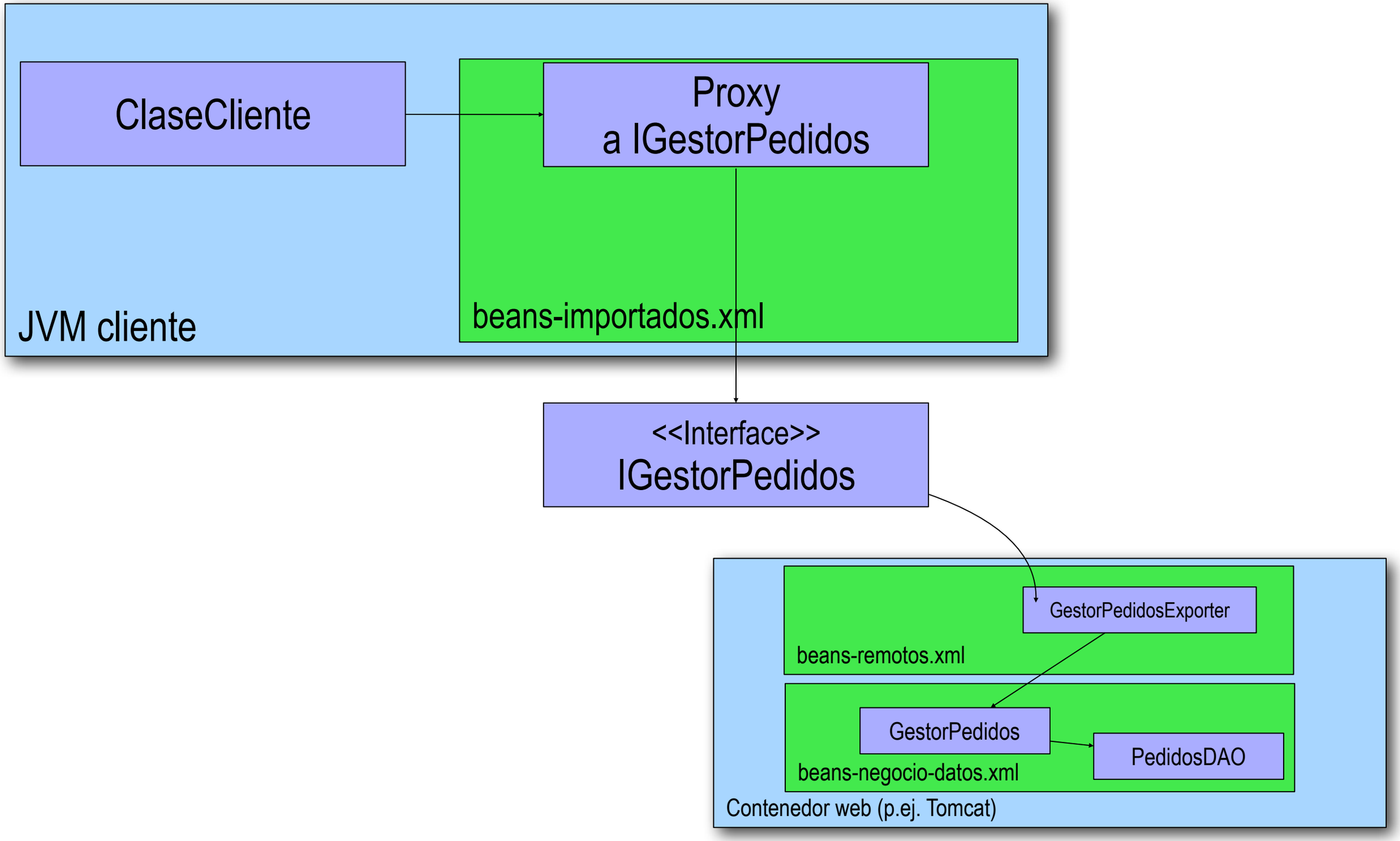
- El bean al que queremos acceder de forma remota **debe** implementar un interface
- Lo que “veremos” desde el cliente es el interface, **no** la implementación
  - Esto nos permitirá tener un interface local y uno remoto (con menos operaciones, posiblemente)



Esquema en el servidor



Esquema en el cliente



Ejemplo

- Queremos acceder remotamente a este bean

```
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", "ola k ase"};

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

HTTP invoker en el servidor

- Inconveniente: el cliente debe ser Java y además requiere las librerías de Spring
- La comunicación con el cliente se hace a través de un servlet
  - Así se puede acceder al servicio remoto por HTTP
  - La implementación del servlet ya está hecha en Spring (clase DispatcherServlet, misma que usábamos en MVC)
- Se asocian ciertas URL con el servlet en el web.xml

HTTP invoker en el servidor (II)

- El fichero de definición de beans exportados se debe llamar por defecto igual que el servlet seguido de "-servlet.xml" (y estar en WEB-INF)
  - En nuestro ejemplo debe ser "remoting-servlet.xml"
- En él se define/n el/los **exporter**, bean/s de la clase HttpInvokerServiceExporter

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

- El *endpoint* será http://localhost:8080/*MI_APLICACION*/remoting/saludadorHTTP

HTTP invoker en el cliente

- Llamar al bean remoto desde Java

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

-

```
<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>
```

Indice

- Spring para acceso remoto
  - Spring vs. EJB
  - Comparación de los protocolos disponibles
- Acceso remoto “clásico”
  - HttpInvoker
- **Pruebas**

Soporte de Spring para pruebas

- Para pruebas unitarias/de integración
  - Uso de un fichero de configuración/perfil distinto al de “producción”
  - Resolución de dependencias
- Para pruebas de objetos de acceso a datos
  - Soporte de bases de datos embebidas
  - Transaccionalidad automática
- Para pruebas de la capa web
  - Pruebas fuera del contenedor web

Pruebas unitarias

- Clase a probar: UsuariosDAOJDBC, que implementa la interfaz:

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

- Clase de prueba

```
@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());
  }
}
```

Fichero de configuración para pruebas

- Base de datos embebida (por defecto HSQLDB)
  - Almacena los datos en memoria, pruebas más rápidas
- Por ahora, el resto igual que “en producción”

```
<!-- 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"/>
```

Transaccionalidad en pruebas

- Deshace automáticamente los efectos de las pruebas ejecutadas
  - Util para dejar la BD en un estado conocido
- Debemos declarar un gestor de transacciones en el fichero de configuración de pruebas

```
<jdbc:embedded-database id="miDataSource">
        <jdbc:script location="classpath:db.sql"/>
        <jdbc:script location="classpath:testdata.sql"/>
</jdbc:embedded-database>

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

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

Pruebas de integración

- Spring resuelve las dependencias, como “en producción”

```
public interface IUsuariosBO {
  //Este método debe comprobar que el password coincide con lo que devuelve el DAO
  //si no lo hace, devolverá null
  public Usuario login(String login, String password);
  //Estos métodos delegan el trabajo en el DAO
  public List<Usuario> listar();
  public void alta(Usuario u);
}
```

```
@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"));
  }
}
```

Pruebas de integración con mocks

- Mockito

```
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"));
```

Pruebas de integración con mocks (II)

- Sustituir UsuariosDAOJDBC que implementa IUsuariosDAO por el *mock*
  - En el fichero de configuración de beans para la capa DAO

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

- Especificar el comportamiento del *mock*

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

  @Autowired
  IUsuariosDAO udao;

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

  @Test
  public void testLogin() {
    assertNotNull(ubo.login("test", "test"));
    assertNull(ubo.login("experto", "experto"));
  }
}
```

Pruebas de la capa web

- Funcionalidad añadida en la última versión de Spring (3.2). Antes estaba en un proyecto aparte
- Podemos probar los controller sin necesidad de desplegar la aplicación en un contenedor web
  - Podemos comprobar el código de estado devuelto por el servidor, el contenido de la respuesta (para AJAX/REST), el nombre de la vista a la que se intenta saltar (para JSP y similares), el contenido del modelo...
  - Lo único que no se puede probar son los JSP, requieren de un contenedor web
- Ejemplo (luego veremos la sintaxis con más detalle)

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

```
@Test
public void testHola() throws Exception  {
  this.mockMvc.perform(get("/hola"))
    .andExpect(status().isOk())
    .andExpect(content().string("Hola Spring"));
}
```

Clase de pruebas para la capa web

- Necesitamos un `MockMVC`, que se construye a partir de un `WebApplicationContext`
- Necesitaremos referenciar también un fichero de configuración de la capa web (puede ser el mismo de producción)

```
@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...
  ...
}
```

Ejemplos de pruebas de controllers

- Hacer la petición

```
//hacer peticiones HTTP y especificar el tipo MIME
mockMvc.perform(get("/usuarios/").accept(MediaType.APPLICATION_JSON));
```

```
//enviar parámetros HTTP (/verUsuario?login=experto)
mockMvc.perform(get("/verUsuario").param("login", "experto"));
```

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

- Comprobar la respuesta

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

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

```
//Comprobar el modelo
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");
```

¿Preguntas...?