

Manual de uso del Servidor y del Driver de Persistencia

1. Introducción
2. Arquitectura del Servidor de Persistencia
3. Instalación y ejecución del Servidor de Persistencia
 - a. Instalación
 - b. Ejecución
4. Instalación y uso del Driver de Persistencia
 - a. Instalación en un proyecto Java
 - b. Uso del Driver
5. Resolución de problemas y contacto

1. Introducción

En el siguiente documento se describe el servicio de persistencia que se usará en el caso práctico de la asignatura. Los entregables se organizan en dos ficheros:

- ServidorPersistencia.jar
- DriverPersistencia.jar

Además se proporciona la documentación en formato javadoc (javadoc.zip) y este documento que explica el uso del servidor y su driver de acceso.

El archivo ServidorPersistencia.zip contiene una aplicación principal ejecutable (ServidorPersistencia.jar) que implementa el servidor de persistencia, una carpeta /lib con las librerías requeridas para el servidor y por último, un readme.txt (con indicaciones breves para la ejecución del servidor).

El archivo DriverPersistencia.jar proporciona el API para acceder al servidor desde aplicaciones Java.

El archivo javadoc.zip contiene la documentación javadoc del driver para su uso. Se debe descomprimir el contenido y acceder a la documentación a través del index.html (se visualiza en un navegador web).

2. Arquitectura del servidor de persistencia

El servidor de persistencia es una aplicación que será lanzada desde un terminal del sistema operativo y que se ejecutará en segundo plano a la espera de recibir peticiones por parte de los clientes que harán uso del driver proporcionado para la persistencia.

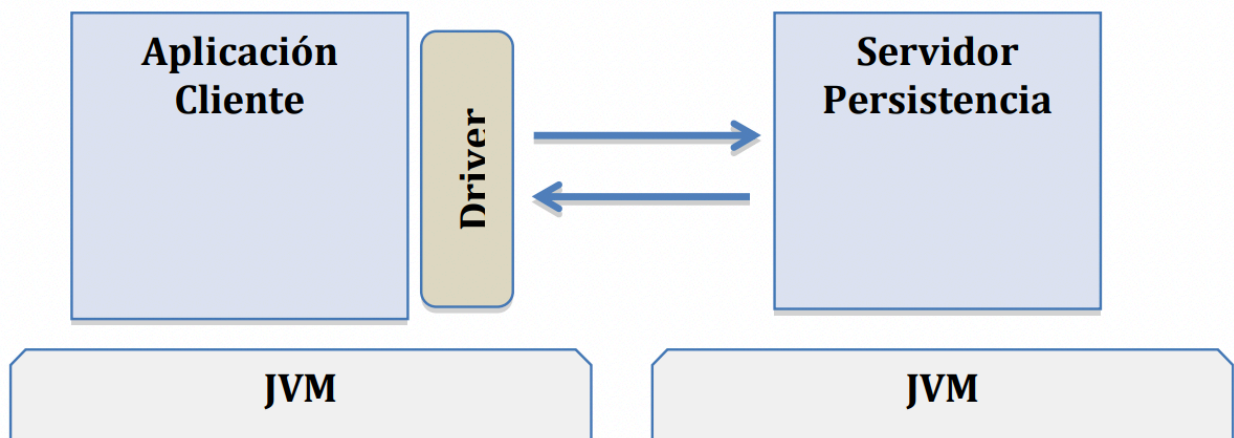
El servidor está implementado haciendo uso de tecnologías Java para la persistencia (JPA) y para la distribución (RMI). No obstante, esto es totalmente transparente al programador puesto que en el desarrollo de las aplicaciones cliente del servidor de persistencia se usará un driver de acceso que les oculta los mencionados requerimientos de persistencia y distribución. Además, el servidor de persistencia se distribuye en dos versiones:

- A. H2:** incluye un motor de base de datos (en concreto H2) que permite usar una base de datos embebida y que por tanto no requiere gestor de base de datos externo ni configuraciones externas. Por lo tanto, el servidor se lanzará de manera autónoma y sin ningún requerimiento ni dependencia de ejecución. Por ello es la opción recomendada para la práctica.
- B. MySQL:** versión del servidor que requiere una base de datos MySQL. Por tanto, es necesario tener instalado este gestor y además, configurar una instancia con los siguientes datos:

```
"Esquema/Catálogo BBDD": PersistenceServer
"Host/Puerto de la instancia de BBDD" : localhost:3306
"Usuario" : root
"Password" : <<sin password>>
```

Por otro lado, el driver de acceso al servidor tampoco tendrá ningún requerimiento de uso de ninguna tecnología específica. Solo requiere usar los métodos definidos en su API de acceso y con los datos que el API requiere en forma de parámetros y valores de retorno.

Vamos a ver a continuación el escenario de uso del servidor y su driver:



Como se puede comprobar, la aplicación cliente que el alumno desarrolle para el caso práctico de la asignatura usará el driver para acceder al servidor de persistencia.

El principal propósito del servidor de persistencia es ofrecer un mecanismo distribuido para gestionar la persistencia de nuestra aplicación, permitiendo registrar entidades y hacer persistentes los objetos registrados (para su posterior consulta o recuperación). El servidor también puede actuar como infraestructura para las comunicaciones asíncronas.

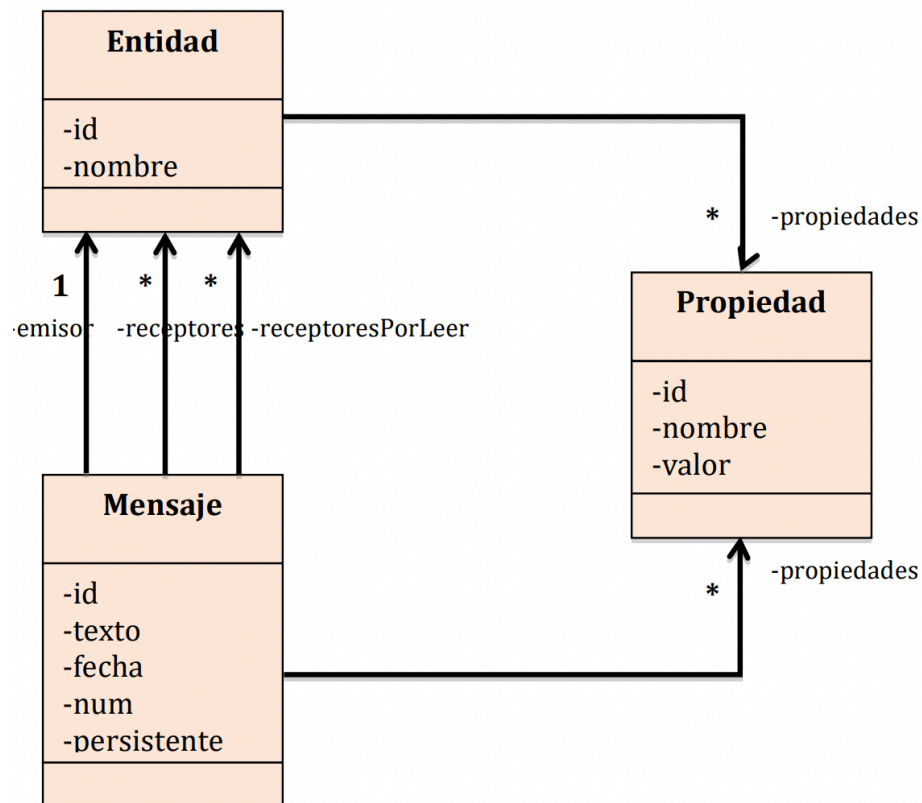
Antes de adentrarnos en la exploración del API que proporciona el driver y que es necesario conocer bien para usarlo en aplicaciones cliente, vamos a describir el modelo de datos en que se basa el servidor de persistencia y que necesitaremos conocer para utilizar el driver.

Para gestionar la persistencia en el servidor se define un modelo básico pero flexible para ser adaptado a diferentes dominios de problemas. Básicamente se definen tres clases: Entidad, Propiedad y Mensaje. Las entidades tan solo tendrán un identificador y un nombre (que habitualmente se usa para definir el tipo de la entidad: Usuario, Venta,

LineaVenta, etc.). Los mensajes gestionan un texto (como contenido principal del mensaje), una fecha, un número y un atributo para indicar si el mensaje es persistente o no. Además, contendrán información sobre el emisor del mensaje (entidad), los destinatarios (entidades) y los destinatarios que aún están pendientes de recibir el mensaje. Por último, tanto entidades como mensajes permiten tener una lista de propiedades (con su valor correspondiente) que les permite establecer los atributos que requiere una entidad o mensaje de manera abierta (e incluso en tiempo de ejecución).

Abajo se muestra el diagrama de clases del modelo de datos. Este modelo viene implementado en el driver mediante tres clases java beans que ofrecen las propiedades indicadas en cada clase, con sus correspondientes métodos get y set (tanto para los atributos de la clase como para los atributos que gestionan las relaciones entre las entidades). Hay que tener en cuenta que las relaciones bidireccionales entre dos entidades A y B, se representarán como dos propiedades (referencias), una propiedad en A cuyo valor/es serán de tipo B y en B una propiedad cuyo valor/es serán de tipo A. Más adelante explicaremos en detalle cómo se registra una referencia de una entidad a otra.

Vamos a ver en los siguientes apartados como usar el servidor de persistencia y su driver de acceso, dentro de un proyecto Java con Eclipse.



3. Instalación y ejecución el Servidor de Persistencia

1. Instalación

Para instalar el servidor basta con descomprimir el archivo `ServidorPersistencia.zip` en una carpeta de nuestro disco duro (evitar nombres de directorios largos y con espacios en blanco al estilo "Archivos de Programa" o "Documents and Settings"). Una vez descomprimido el archivo encontraremos que contiene lo siguiente:

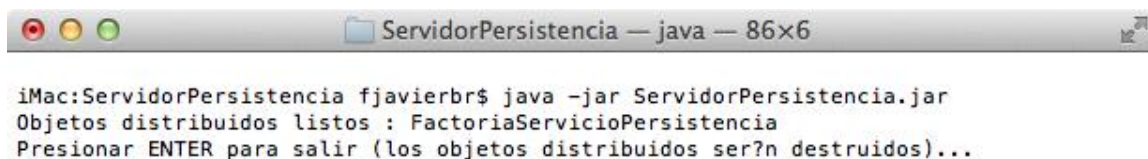
ServidorPersistencia.jar: la implementación del servidor empaquetada en un .jar ejecutable.

/lib : contiene las librerías con las tecnologías que requiere el servidor para la persistencia y el acceso a base de datos.

Readme.txt: con instrucciones breves para la ejecución del servidor de persistencia. Debemos conservar la estructura generada durante la descompresión (no cambiar ubicación de carpetas y/o archivos).

3.2. Ejecución

Abrimos un terminal del sistema operativo, accedemos a la carpeta donde se encuentra ubicado el servidor de mensajes anteriormente descomprimido e invocamos al interprete de java con el siguiente comando: **java -jar ServidorPersistencia.jar**. Se ejecutará el servidor y si todo va bien veremos los siguientes mensajes:



```
iMac:ServidorPersistencia fjavierbr$ java -jar ServidorPersistencia.jar
Objetos distribuidos listos : FactoriaServicioPersistencia
Presionar ENTER para salir (los objetos distribuidos ser?n destruidos)...
```

Para finalizar la ejecución del servidor, tal y como pone en el mensaje, basta con pulsar ENTER. Se finalizará el servidor y se cerrarán los canales de comunicación, aunque no se perderán los datos registrados puesto que se hacen persistentes en una base de datos.

4. Instalación y uso del driver de acceso al servidor

1. Instalación en un proyecto Java

Para instalar el driver de acceso al servidor que tendremos ejecutándose desde la línea de comandos basta con añadirlo como librería a nuestro proyecto Java. Para ello, crearemos una carpeta /lib en el proyecto e importaremos sobre la carpeta la librería del driver (seleccionamos la carpeta, botón derecho y elegimos “import”, para luego indicar “General->File system” e indicar el DriverPersistencia.jar; también es posible arrastrar y soltar el fichero desde la carpeta a /lib). Debemos importar también las librerías .jar de eclipselink y h2 que están en la carpeta /lib de la distribución del servidor: eclipselink.jar javax.persistence_xxx.jar y h2-1.3.160.jar. Una vez importadas las cuatro librerías .jar, seleccionamos nuestro proyecto java, y con el botón derecho abrimos el menú emergente para elegir “BuildPath->Configure BuildPath”. En la ventana que se nos abre, elegimos la pestaña “Libraries” y hacemos click en el botón “Add jar” para añadir los cuatro .jar que acabamos de importar en la carpeta /lib de nuestro proyecto java.

4.2. Uso del driver

Para usar el driver necesitamos manejar las clases:

beans.Entidad, beans.Propiedad y beans.Mensaje (si se manejan mensajes en un servicio de mensajería): se trata de java beans con las propiedades indicadas en el apartado anterior, y con los métodos get/set correspondientes (ver javadoc).

tds.driver.FactoriaServicioPersistencia: factoría para obtener acceso al servidor de persistencia mediante un objeto del tipo ServicioPersistencia.

Y la interfaz principal de acceso al servidor de persistencia:

tds.driver.ServicioPersistencia: API de acceso y comunicación con el servidor de persistencia.

Nota: Cuando importamos las clases del driver, puede suceder que Eclipse marque en rojo un error (haciendo referencia a que no se encuentran ciertas clases). No se debe hacer caso a este error, puesto que realmente las clases que busca Eclipse sí que están disponibles, insertadas como .jar dentro del propio DriverPersistencia.jar. De hecho, el error que marca no se notifica en la vista de problemas y se puede ejecutar la aplicación normalmente.

4.3 Obtener una referencia al servidor

Para obtener una referencia al API de Servicio de Persistencia necesitamos obtener una implementación de dicho acceso a través de la factoría:

```
ServicioPersistencia servPersistencia =  
    FactoriaServicioPersistencia.getInstance().getServicioPers  
    istencia();
```

Una vez hecho esto, podemos usar el objeto del tipo ServicioPersistencia para la comunicación con el servidor de persistencia.

Algunos ejemplos son:

i) recuperar todas las entidades registradas en el servidor:

```
servPersistencia.recuperarEntidades();
```

ii) registrar una entidad Usuario con una propiedad 'nif' con valor '12345678A':

```
Entidad usuario= new Entidad();  
  
usuario.setNombre("Usuario");  
  
usuario.setPropiedades(new  
    ArrayList<Propiedad>(Arrays.asList(  
        new Propiedad("nif", "12345678A"))));  
  
servPersistencia.registrarEntidad(usuario);
```

4.4 Registrar Entidades (ver PDF transparencias ACTUALIZADO)

Para el registro de entidades es necesario crear un objeto Entidad, establecer el nombre (tipo) de la entidad (venta, producto, lineaventa, etc.) y establecer las propiedades del objeto según dicha entidad (tipo). Por ejemplo, un objeto del negocio prod1 de tipo Producto se registraría del siguiente modo, suponiendo que tiene los campos nombre, descripción y precio:

```
//Registrar objeto prod1 de tipo Producto
```

```

Entidad eProducto = new Entidad();
eProducto.setNombre("Producto");
eProducto.setPropiedades(new ArrayList<Propiedad>(Arrays.asList(
    new Propiedad("nombre", prod1.getNombre()),
    new Propiedad("descripcion", prod1.getDescripcion()),
    new Propiedad("precio", prod1.getPrecio()))));
eProducto = servPersistencia.registrarEntidad(eProducto);

//el id de la Entidad se autogenera
idProducto = eProducto.getId();
//se asigna al objeto prod1
prod1.setCodigo(idProducto)

```

El último comentario indica que las clases del modelo (Entidad, Propiedad, y Mensaje) tienen un atributo interno id que contiene el identificador único del objeto. Este valor es gestionado por el motor de persistencia y por lo tanto se autogenera. El modo de trabajar con este atributo es el que se muestra en el código de arriba: (1) cuando se crea el objeto Entidad, no se debe asociarle un valor con setId() sino dejar que lo genere la propia base de datos, (2) se puede recuperar el identificador (id) con el método getId() una vez registrada la entidad y generado su identificador. Este identificador se puede usar en nuestra propia aplicación para clases que necesiten que sus objetos tengan un identificador único como se muestra en el código anterior.

4.5 Recuperar Entidades (ver PDF transparencias ACTUALIZADO)

Dado un identificador de una entidad se puede recuperar de la base de datos con el siguiente código. El siguiente código recupera una entidad Producto cuyo id está registrado en la variable idProducto.

```

Entidad eProducto =
servPersistencia.recuperarEntidad(idProducto);
double precio =

Double.parseDouble(servPersistencia.recuperarPropiedadEntidad(eProducto, "precio"));
String nombre =
servPersistencia.recuperarPropiedadEntidad(eProducto, "nombre");
String descripcion =
servPersistencia.recuperarPropiedadEntidad(eProducto, "descripcion");
Producto producto = new Producto(precio,nombre,descripcion);

```

4.6. Modificar Entidades

Para modificar una entidad que ya existe en base de datos basta con recuperar la entidad y modificar sus propiedades. Se puede utilizar el método del API para modificar propiedades. El siguiente código actualiza un objeto Producto ya almacenado, producto1 registra el objeto con los datos a almacenar.

```

Entidad eProducto;
eProducto =
servPersistencia.recuperarEntidad(producto1.getCodigo());

for (Propiedad prop :
    eProducto.getPropiedades()) { if
    (prop.getNombre().equals("codigo")) {
        prop.setValor(String.valueOf(producto1.getCodigo()));
    }
    else if (prop.getNombre().equals("precio"))
        { prop.setValor(String.valueOf(producto1.get
        Precio()));
    }
    else
        if(prop.getNombre().equals("nomb
        re"))
        { prop.setValor(producto1.getNom
        bre());
    }
    else
        if(prop.getNombre().equals("descripci
        on"))
        { prop.setValor(producto1.getDescrip
        tion());
    }
    servPersistencia.modificarPropiedad(prop);
}

```

4.7 Borrar Entidades

Para borrar un objeto Entidad simplemente usamos el método borrarEntidad():

```

Entidad eProducto;
eProducto =
    servPersistencia.recuperarEntidad(idProducto);
servPersistencia.borrarEntidad(eProducto);

```

4.8 Referencias unidireccionales entre entidades

Para relacionar una entidad A con una entidad B mediante una asociación unidireccional con cardinalidad 1 hay que añadir a la entidad A una propiedad que almacenará el valor del id de la entidad B referenciada. Por ejemplo, si tenemos una entidad LineaVenta

que referencia a una entidad Producto, entonces creamos una propiedad en LineaVenta, que podemos llamar 'producto', que contendrá el valor del id de la entidad Producto referenciada:

```
Entidad eLineaVenta = new Entidad();
eLineaVenta.setNombre("LineaVenta");
eLineaVenta.setPropiedades(new ArrayList<Propiedad>(Arrays.asList(
    new Propiedad("codigo", String.valueOf(lineaVenta.getCodigo())),
    new Propiedad("unidades", String.valueOf(lineaVenta.getUnidades())),
    new Propiedad("producto", String.valueOf(lineaVenta.getProducto().
                                                    getCodigo()))
)));
eLineaVenta = servPersistencia.registrarEntidad(eLineaVenta);
```

Si la relación de asociación entre entidades fuera con cardinalidad múltiple, por ejemplo, entre Venta y LineaVenta, entonces la propiedad que representa la relación debería almacenar un string con la secuencia de ids de los objetos referenciados, en este caso las líneas de venta:

```
String líneas = obtenerCodigosLineaVenta(venta.getLineasVenta());

Entidad eVenta = new Entidad();
eVenta.setNombre("Venta");
eVenta.setPropiedades(
    new ArrayList<Propiedad>(Arrays.asList(
        new Propiedad("codigo", String.valueOf(venta.getCodigo())),
        new Propiedad("fecha", dateFormat.format(venta.getFecha())),
        new Propiedad("lineasventa", líneas)
    )));
eVenta = servPersistencia.registrarEntidad(eVenta);
```

El siguiente método convertiría una lista de objetos LineaVenta en un string con sus ids.

```
private String obtenerCodigosLineaVenta(List<LineaVenta> lineasVenta)
{
    String líneas = "";
    for (LineaVenta lineaVenta: lineasVenta) {
        líneas += lineaVenta.getCodigo() + " ";
    }
    return líneas.substring(0, líneas.length()-1);
}
```

Del mismo modo en la recuperación de objetos desde la base de datos sería preciso un método que convirtiese un string con una lista de ids en objetos. El siguiente método convertiría un string de esa naturaleza en líneas de venta.

```
private List<LineaVenta> obtenerLineasVentaCodigos(String líneas)
{
    List<LineaVenta> lineasVenta = new LinkedList<LineaVenta>();
    StringTokenizer strTok = new StringTokenizer(líneas, " ");
    while (strTok.hasMoreTokens()) {lineasVenta.add(
        adaptadorLV.recuperarLineaVenta(
            Integer.valueOf((String)strTok.nextElement()));
    )}
    return lineasVenta;
}
```


4.9 Referencias bidireccionales entre entidades

Habitualmente encontraremos modelos del dominio donde se establecen asociaciones bidireccionales entre clases, por ejemplo entre una clase Usuario y una clase Venta que representa las ventas realizadas por un usuario: Usuario tendría una lista de ventas realizadas y una Venta referencería al usuario que la ha realizado. Esto puede generar un problema a la hora de recuperar las entidades puesto que si, por ejemplo, tratamos de recuperar una entidad Usuario, entonces al tratar de recuperar su lista de Ventas realizada se produciría un bucle infinito dado que cada Venta intentaría recuperar a su usuario. Para que no se produzca este bucle infinito es necesario que al recuperar una venta se compruebe si el usuario ya se ha recuperado y no se intente recuperar de nuevo. O al revés si el orden es el inverso, primero se recupera la venta y luego sus usuarios.

Una posible estrategia para solucionar este problema se basaría en el manejo de un pool de objetos por parte de los adaptadores DAO. Como muestra el código de la clase de abajo, este pool estará implementado como una tabla cuyo clave es el id del objeto y sería una clase singleton con métodos para añadir y recuperar objetos de la tabla.

```
public class PoolDAO {
    private static PoolDAO instance = null;
    private HashMap<Integer, Object> pool;

    private PoolDAO(){
        pool = new Hashtable<Integer, Object>();
    }
    public static PoolDAO getInstance(){
        if(instance == null) instance = new
        PoolDAO(); return instance;
    }
    public void addObject(int id, Object
        object) { pool.put(id, object);
    }
    public Object getObject(int id) {
        return pool.get(id);
    }
    public boolean contains(int id){
        return pool.containsKey(id);
    }
}
```

Antes de recuperar un nuevo objeto se comprueba si ya existe en el pool y sólo se recupera cuando no existe. Por ejemplo, el método de un DAO que se encarga de recuperar un objeto Usuario de la base de datos comprobaría si el usuario está en el pool, si no lo está lo cargaría y si lo está retornaría el objeto del pool. Supongamos que Usuario tiene atributos nombre y dni de tipo String. En la clase DAO para Usuario tendríamos:

```
public Usuario recuperar(int id) {
    if ( !PoolObjetos.getInstance().contains(id)){
        Entidad eUsuario =
            servPersistencia.recuperarEntidad(id); return
            entidadAUsuario(eUsuario);
    }
    else
        return (Usuario)
            PoolObjetos.getInstance().getObject(id);
}
```

Los métodos que convierten una entidad en un objeto (por ejemplo entidadAUsuario en este caso), primero recuperarán los campos que corresponden a atributos (y asociaciones unidireccionales si las hubiese), en segundo lugar crearán una instancia que almacenarán en el pool (en nuestro ejemplo un Usuario), y en tercer lugar recuperarán el/los objeto/s de la asociación bidireccional (Venta en este caso) y los asignarán al objeto creado que será finalmente retornado.

```
private Usuario entidadAUsuario(Entidad eUsuario) {
    int id = eUsuario.getId();
    String nombre = servPersistencia.recuperarPropiedadEntidad(eUsuario,
        "nombre");
    String dni = servPersistencia.recuperarPropiedadEntidad(eUsuario, "dni");

    Usuario usuario = new Usuario(nombre, dni);
    usuario.setId(id);
    PoolObjetos.getInstance().addObject(usuario.getId(),usuario);

    String ventasCodigos =
        servPersistencia.recuperarPropiedadEntidad(eUsuario, "ventas");
    if (!ventasCodigos.equals("") && ventasCodigos != null)
    { List<Venta> ventas =
        obtenerVentasCodigos(ventasCodigos);
        usuario.setVentas(ventas);
    }
    return usuario;
}
```

El método obtenerVentasCódigo() invocaría al método que recupera una venta, como hemos explicado en el apartado 4.8, pero ahora al recuperar el usuario que ha realizado la venta se encontraría que ya está en el pool y no es necesario recuperarlo de la base de datos, con lo que se evita el bucle infinito.

5. Resolución de problemas y contacto

En caso de problemas con el funcionamiento del servidor o del driver, remitir error a la dirección de correo: fjavier@um.es , indicando en el mail:

- o **Situación que generó el error (indicar código del cliente de la aplicación programada).**
- o **Traza de la excepción: copiar y pegar la traza completa de la excepción**
- o **Cualquier comentario que se quiera hacer notar.**

Por favor, indicar al menos estos datos para que se pueda atender el problema. También se puede mandar un mail para cualquier sugerencia.

Anexo A: Modelo de Clases del Servidor

```
public class Entidad implements Serializable { private int
    id;

    private String nombre;
    private List<Propiedad> propiedades;

    public Entidad() {
        super();
    }
    public int getId() {
        return this.id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getNombre() {
        return this.nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public List<Propiedad> getPropiedades() { return
        propiedades;
    }
    public void setPropiedades(List<Propiedad> propiedades)
        { this.propiedades = propiedades;
    }
    public final boolean equals(final Object o) { return
        ((Entidad)o).getId() == this.getId();
    }
}
```

```
public class Propiedad implements Serializable { private int
    id;
    private String nombre;
    private String valor;

    public Propiedad() {
        super();
    }
    public Propiedad(String nombre, String valor) { super();

        this.nombre = nombre;
        this.valor = valor;
    }
    public String getNombre() {
        return this.nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getValor() {
        return this.valor;
    }
    public void setValor(String valor) {
        this.valor = valor;
    }
}
```

```
    public int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
    public final Boolean equals(final Object o) { return  
        ((Propiedad)o).getId() == this.getId();  
    }  
}
```