Ingeniería de Telecomunicación

Escuela de Ingenieros
Universidad de Sevilla

PROYECTO FIN DE CARRERA

Implementación de las APIs de consulta y publicación de la especificación UDDI

David Reales Ríos

Tutora: Isabel Román Martínez

Sevilla, Mayo de 2005

1. Introducción

1.1. Motivación y Objetivos

La importancia y amplia difusión que está adquiriendo la tecnología de los servicios web hace necesaria la creación de un sistema para que las empresas proveedoras puedan publicitar sus servicios, de manera que las compañías puedan encontrarse unas a otras y realizar transacciones en la web. El proyecto de la especificación "Universal Description, Discovery and Integration", que en lo sucesivo denominaremos UDDI, está diseñado para hacer frente a este reto, habilitando una plataforma para publicar, descubrir y enlazar unos servicios web con otros.

El presente proyecto tiene como objetivo el desarrollo y la implementación de un registro privado para servicios web, basado en la especificación UDDI, de modo que un usuario pueda publicar y hacer una búsqueda de servicios de una manera rápida y sencilla. Además, se desea que la aplicación desarrollada sea independiente de la plataforma donde se ejecute y que utilice herramientas de libre distribución.

Diseñar un registro UDDI completo es una tarea que conlleva una inversión muy importante de tiempo y esfuerzo de un grupo de programadores, en lugar de ello, se implementarán las APIs de consulta y de publicación según la especificación UDDI API versión 2.04, de modo que se pueda hacer un uso programático del registro. Aunque la especificación UDDI se basa en mensajes XML, no se implementará la capa SOAP correspondiente que podría atender este tipo de peticiones. La razón es que no se usarán mensajes SOAP, sino formularios y servlets que accederán directamente a los métodos de la API, por lo que la realización de dicha capa queda fuera del alcance de este proyecto.

Para facilitar el uso de las APIs, se diseñará una interfaz basada en web para que se pueda hacer un uso básico de los métodos de las APIs. También se implementará una utilidad para configurar fácilmente algunos parámetros del registro, como el archivo de autenticación de usuarios, y un sistema de limpieza automática del repositorio de datos.

1.2. Material y Métodos

Uno de los objetivos del presente proyecto es la independencia de plataforma, de esta manera se consigue que el producto desarrollado pueda ser usado en una gran variedad de sistemas sin necesidad de ningún tipo de adaptación. Por ello, se ha optado por elegir JAVA como el lenguaje de programación para la implementación de la aplicación.

Además, se pretende que la aplicación no dependa de tecnologías propietarias, que podrían coartar la distribución de la misma. Para sortear este obstáculo usaremos herramientas de libre uso y distribución, tanto en fase de desarrollo como de depuración y ejecución.

Las herramientas utilizadas durante la creación del proyecto han sido las que se detallan a continuación:

1.2.1. Hardware

Se han utilizado dos máquinas, una para todo el proceso de diseño, programación y depuración, y donde también se ejecutará nuestra aplicación, que denominaremos "servidor", y otra desde la que accederemos a la aplicación desde una ubicación remota, para comprobar el correcto funcionamiento de la misma y que llamaremos "cliente".

Las características de ambos equipos son las siguientes:

- Servidor:
  Portátil Acer Aspire 2003 WLMi - Centrino 1.6 Ghz – 512 MB RAM

- Cliente:
  PC Clónico - AMD 64 3.4 Ghz – 512 MB RAM

1.2.2. Software

El sistema operativo empleado es el Microsoft Windows XP Home Edition en el "servidor" y el Professional en el "cliente", en ambos casos con el Service Pack 2 instalado.

En cuanto a los programas y utilidades empleadas, se detallan a continuación:

1.2.2.1. Java Development Kit

El primer paso para desarrollar nuestra aplicación consistirá en elegir un kit de desarrollo Java adecuado. Usaremos el J2EE de Sun, que entre otras características, permite el desarrollo de servicios web, manejo de documentos XML y creación de servlets y páginas JSP, requisitos indispensables para la aplicación que se implementará.

El JDK puede obtenerse en la siguiente ubicación:

http://java.sun.com/j2ee/1.4/download.html#sdk

1.2.2.2. Netbeans

Necesitaremos además un entorno de desarrollo en el que programar nuestra aplicación. Teniendo en mente el uso de programas de libre distribución hemos elegido el IDE Netbeans que nos proporciona Sun. Este entorno de desarrollo nos permite utilizar todas las características del J2EE, enlazar bases de datos, crear servicios web, registrarlos y probarlos, depurar el código y muchas otras características interesantes. Además, dispone de su propio contenedor de servlets, en el que haremos pruebas en tiempo de desarrollo.

Esta IDE puede descargarse en la siguiente dirección:

http://www.netbeans.org/community/releases/41/index.html

1.2.2.3. MySQL

MySQL es la base de datos de código abierto más extendida en el mundo, es fácil de usar y tiene un excelente rendimiento. Además, está disponible para más de 20 sistemas operativos, incluyendo distribuciones de Linux, Mac OS X, UNIX y Microsoft Windows, lo que la hace especialmente interesante en un proyecto que pretende independencia de plataforma.

El sistema de base de datos se puede descargar de:

http://dev.mysql.com/downloads/mysql/4.1.html

Además, necesitaremos el conector MySQL para JAVA, que podemos descargar de la siguiente ubicación:

http://www.mysql.com/products/connector/j/

Aunque no es estrictamente necesario, tambien hemos usado una herramienta disponible en la misma web de MySQL para administrar y monitorizar de manera gráfica nuestra base de datos. Dicha aplicación es MySQL Administrator, y puede descargarse en la siguiente dirección:

http://dev.mysql.com/downloads/administrator/index.html

1.2.2.4. Tomcat

Nuestro proyecto consistirá en la creación de una aplicación web, por lo que necesitaremos un contenedor de aplicaciones. Aunque como hemos comentado, el IDE ya dispone de su propio contenedor, utilizaremos también uno autónomo para comprobar que la aplicación se despliega y funciona correctamente. Usaremos el contenedor de servlets Tomcat del proyecto Apache, que resulta muy robusto y además es de libre distribución.

La dirección para descargarlo es la siguiente:

http://jakarta.apache.org/site/downloads/downloads_tomcat-5.cgi

1.2.2.5. GanttProject

Es una herramienta de código abierto para hacer diagramas de Gant como el que aparece en esta memoria. Puede descargarse de la siguiente dirección:

http://ganttproject.sourceforge.net/

1.3. Etapas de desarrollo y tiempos

Detallaremos a continuación las etapas por las que ha pasado el desarrollo del presente proyecto. Para ello hacemos uso del diagrama de Gant de la figura 1.1, que indica la extensión en el tiempo y el solapamiento de las diferentes etapas.

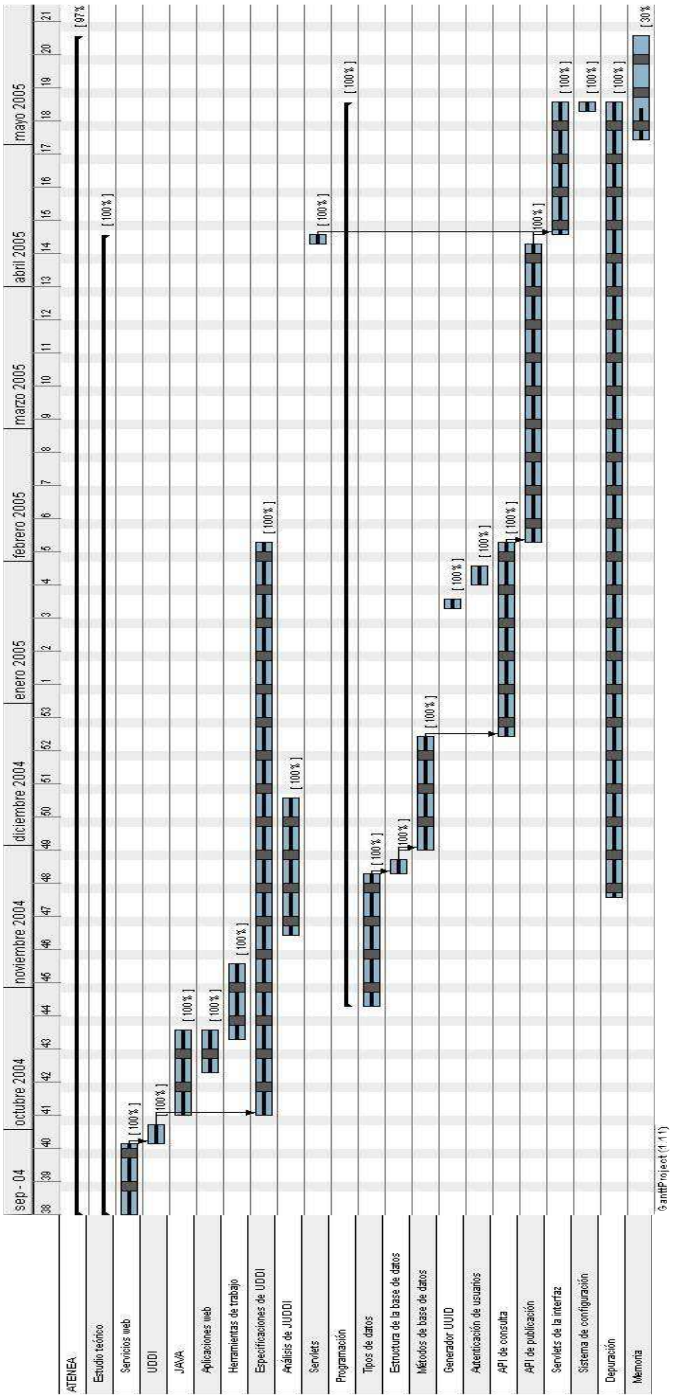

Figura 1.1 Diagrama de Gant

1.3.1. Estudio de la plataforma de servicios web

Se asimilan los conceptos relativos a los servicios web, qué son, para qué sirven y cómo se diseñan e implementan.

1.3.2. Estudio de UDDI

Los servicios web son componentes de software que "viven" en la red, para que tengan sentido debe haber una forma de publicitarlos, encontrarlos y obtener la información necesaria para poder utilizarlos. UDDI es la pieza clave en la plataforma de servicios web que permite dar una solución a estos problemas. En esta etapa estudiamos qué es UDDI y cómo funciona.

1.3.3. Estudio de JAVA

Dado que vamos a utilizar este lenguaje de programación buscando la independencia de plataforma, en esta etapa procedemos a repasar los conceptos básicos y a estudiar el acceso a base de datos, el manejo de documentos XML y otras características que nos harán falta para la labor de programación.

1.3.4. Estudio de desarrollo de aplicaciones web

En esta etapa nos documentamos acerca de la estructura de las aplicaciones web, cómo se crean y despliegan, y del uso de contenedores de aplicaciones.

1.3.5. Aprendizaje de uso de las herramientas de trabajo

Es momento de aprender a manejar las herramientas que vamos a emplear a lo largo del proyecto. Se realizan tutoriales acerca del uso del contenedor de aplicaciones Tomcat, del entorno de desarrollo Netbeans y del gestor de bases de datos MySQL.

1.3.6. Especificaciones de UDDI

Obtenemos las especificaciones de UDDI versión 2 desde la página web de la plataforma y pasamos a analizarlas para diseñar la aplicación. A lo largo de la etapa de programación será una guía para determinar el comportamiento de cada método de las APIs que se implementarán, así como para cumplir con las interfaces que se detallan en el documento.

Además, se utilizará otro documento oficial, la "UDDI Data Structure Reference" para determinar los tipos de datos que manejará el registro.

1.3.7. Análisis de JUDDI

Se intenta instalar la aplicación y hacerla funcionar, con malos resultados, como se verá en el apartado 2.3.5. Se pasa a analizar el código, que nos permite obtener una manera de afrontar el diseño de nuestra aplicación.

1.3.8. Estudio de Servlets

En esta etapa se estudia la programación de servlets, de cara a usarlos en la interfaz de la aplicación. Se revisan numerosos tutoriales y se realizan pruebas previas. Se aprende a usar variables de sesión para mantener a un usuario identificado en el sistema.

1.3.9. Programación de tipos de datos

Se utiliza el documento "Data Structure Reference" para programar cada uno de los tipos de datos del registro UDDI, creando métodos de acceso para los componentes de cada uno de ellos, y organizándolos en paquetes lógicos.

1.3.10. Estructura de la base de datos

Una vez conocidos los tipos de datos del registro, cómo se relacionan y encapsulan unos dentro de otros, y las relaciones de multiplicidad de cada uno, pasamos a diseñar la estructura de la base de datos que tendrá la responsabilidad de albergar toda la información del registro.

1.3.11. Métodos de base de datos

En esta etapa se programa el acceso a la base de datos desde el registro y la manipulación de tablas, la inserción, extracción y borrado de datos.

1.3.12. Generador UUID

Los tipos de datos en UDDI, a menudo están caracterizados por un identificador único universal denominado UUID (Universal Unique IDentifier). En esta etapa se programa el módulo que permite generar este identificador.

1.3.13. Autenticación de usuarios

Cuando un usuario quiere registrar información, debe estar autorizado para ello. Creamos un documento XML donde se guardarán las claves de acceso de cada usuario, y que le darán permiso para operar sobre sus propios datos. Además, creamos la estructura que soportará la identificación de usuarios mediante tokens de autenticación con tiempo de vida limitado.

1.3.14. Programación de las APIs de UDDI

Ésta es la etapa más larga del proyecto, y es la dedicada a implementar cada uno de los métodos de la API de consulta y de la API de publicación de la especificación UDDI. Se recogen los datos de la petición, se comprueba que cumplen con los requisitos establecidos y se enlaza con el módulo de métodos de base de datos para realizar la operación requerida y obtener los resultados. En el caso de que sea una operación de publicación, en primer lugar se comprueba que el token de autenticación sea válido y que disponga del permiso correspondiente.

1.3.15. Servlets de la interfaz de usuario

Una vez programadas las APIs, se crearán servlets para los métodos del registro que servirán de interfaz de usuario para poder manipular los datos contenidos en el mismo. La interfaz que se presenta al usuario no es completa, en el sentido de que no abarca todos los métodos ni con todas las opciones que se han habilitado al programar las APIs, pero permite manipular el registro adecuadamente.

1.3.16. Sistema de configuración

Hay ciertos parámetros en un registro UDDI que pueden configurarse para tener el comportamiento deseado, así como datos que han de actualizarse según la instalación que estemos haciendo, como la URL de despliegue del servicio. En esta etapa hemos programado un servlet que posibilita un amigable sistema de configuración para el usuario final.

1.3.17. Depuración de la aplicación

A lo largo de esta etapa se monitoriza el funcionamiento de la aplicación en todos sus niveles, se comprueba que funciona como es de esperar y se hacen cambios si es necesario.

1.3.18. Redacción de la memoria

Finalmente se redacta la presente memoria del proyecto.

1.4. Estructura del documento

La presente memoria se estructura en capítulos bien diferenciados. El nombre de cada capítulo y su contenido se detallan a continuación:

- **Capítulo 1: Introducción**

  Se explican las motivaciones y objetivos de este Proyecto, se detalla el material empleado y las etapas de desarrollo.

- **Capítulo 2: Estado del arte**

  Se hace un repaso de las tecnologías que envuelven este Proyecto, la plataforma de servicios web, el protocolo SOAP, la especificación UDDI y la aplicación de software de código abierto jUDDI.

- **Capítulo 3: Desarrollo del Proyecto**

  Se explican los pasos dados para crear la aplicación objetivo de este Proyecto, tanto del motor que procesa las consultas a la API UDDI como de la interfaz de usuario.

- **Capítulo 4: Conclusiones y líneas de continuación**

  Se hace una evaluación del trabajo hecho y se dan unos apuntes sobre las posibles mejores y la forma de afrontarlas.

- **Capítulo 5: Manual de instalación**

  Se escribe el manual de instalación para el usuario final.

- **Capítulo 6: Bibliografía**

  Por último se detallan las fuentes de información consultadas en la realización del Proyecto.

No se ha reservado un capítulo para el código dado que su volumen es excesivo para un documento escrito. En su lugar lo hemos incluído en el CD adjunto.

2. Estado del Arte

2.1. Servicios Web

2.1.1. Introducción

Los Servicios Web, lejos de ser una moda pasajera, representan un cambio fundamental en la industria, son el siguiente paso en la evolución de la World Wide Web y en la tecnología orientada a objetos, al alejarse de la arquitectura típica "cliente-servidor" y presentar arquitecturas distribuidas del tipo igual a igual o "peer to peer". Los Servicios Web permiten que las aplicaciones compartan información y que además invoquen funciones de otras aplicaciones independientemente de cómo se hayan creado las aplicaciones, sea cuál sea el sistema operativo o la plataforma en que se ejecutan y cuáles los dispositivos utilizados para obtener acceso a ellas. Aunque los Servicios Web son independientes entre sí, pueden vincularse y formar un grupo de colaboración para realizar una tarea determinada.

Podríamos pensar en un servicio web como si de una rutina en Internet se tratase. En programación estructurada, un conjunto de rutinas y un objetivo central constituyen un programa. El servicio web cumple la misma función, pero está disponible en la red para ser utilizado por otros servicios.

Por lo tanto podemos ver el conjunto de servicios web que pueblan Internet como una World Wide Web paralela, de carácter no humano, sino cibernético, con sus proveedores de información, sus usuarios y sus sistemas de búsqueda.

Una limitación importante en el uso de los servicios web es que sólo pueden funcionar si se dispone de conexión a Internet en el ordenador en el que se ejecuta. Sin embargo, un servicio web tiene ventajas que de alguna manera quitan importancia a este inconveniente. Las rutinas de los servicios web se actualizan de forma transparente para el programador y para el encargado de mantenimiento de la aplicación. Además, gracias a los servicios web un programa puede implementar funciones imposibles de crear usando rutinas de librerías, como por ejemplo, disponer de un buscador de páginas web estilo Google.

Por otro lado, el uso de servicios web supone un ahorro de carga de CPU muy importante con respecto a la implementación de la misma función por una rutina de librería. De hecho, el uso de CPU al usar un servicio web se reduce a la mínima expresión. En realidad, el trabajo se reparte por Internet, sobre el servidor del servicio web. Es por tanto un mecanismo de Computación Distribuida.

De todas formas, los servicios web no pueden sustituir a las rutinas de librería, ya que no son una versión mejorada de éstas, sino una herramienta con distinto ámbito de aplicación. Por ejemplo, no sería eficiente utilizar un servicio web para implementar una utilidad que necesitase procesar una gran cantidad de datos locales, ya que el ancho de banda necesario para transmitir toda la información sería excesivo.

En cambio, hay otras ocasiones en las que sí que es útil, cuando no necesario, un servicio web en lugar de una rutina de librería. Por ejemplo si nuestra aplicación

requiere hacer una búsqueda de páginas web, disponer de la cotización en tiempo real de acciones de Bolsa, consultar el nombre de un libro a partir de su ISBN, saber a quién pertenece cierto dominio en Internet, o mostrar al usuario previsiones meteorológicas.

En cualquier caso, los servicios web no van a revolucionar el mundo que ven los usuarios de Internet, es más bien una herramienta para programadores que les permite reutilizar componentes distribuidos por Internet en vez de hacer un módulo local. Es posible que esta nueva filosofía provoque que haya más elementos comunes en distintas páginas web, es decir, que el usuario advierta que se usan los mismos módulos en distintas web, pero poco más.

2.1.2. ¿Qué es un servicio web?

Antes de profundizar más, definamos el concepto de "servicio web". Un servicio web es una interfaz de red con la que podemos acceder a la funcionalidad de una determinada aplicación, y que está construida usando tecnologías estándar de Internet. Todo esto está ilustrado en la figura 2.1

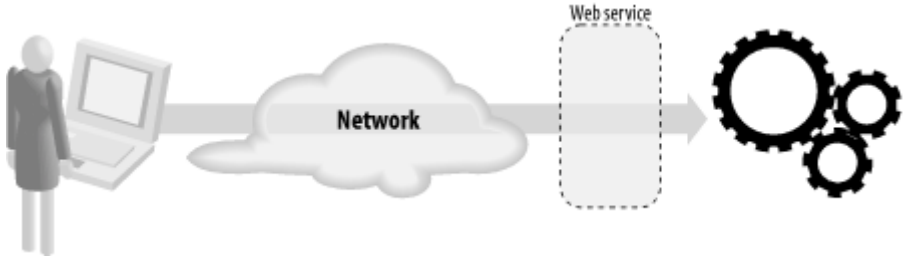

Figura 2.1 Servicio web como interfaz de aplicación

En otras palabras, si una aplicación puede ser utilizada a través de una red usando una cierta combinación de protocolos como HTTP, XML, SMTP o Jabber, entonces es un servicio web.

2.1.3. Fundamentos de los servicios web

Como aparece en las figuras 2.1 y 2.2, un servicio web es una interfaz colocada entre el código de la aplicación y el usuario de dicha aplicación. Funciona como una capa de abstracción, separando los detalles específicos de la plataforma donde se ejecuta y del lenguaje de programación usado, del modo en el que se invoca al código de la aplicación.

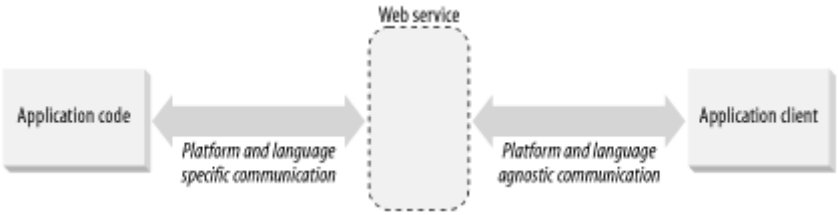

Figura 2.2 Servicio web como capa de abstracción

Los servicios web que vemos actualmente en Internet son páginas web HTML. En dichas web, los servicios ofrecidos por la aplicación (mecanismos para gestionar, publicar, buscar y obtener información) son invocados usando protocolos y formatos de datos estándar: HTTP y HTML. Las aplicaciones cliente que entienden estos estándares (navegadores web) pueden interactuar con los servicios de la aplicación para realizar tareas como hacer pedidos, enviar tarjetas de felicitación, leer noticias, etc…

Gracias a la capa de abstracción proporcionada por una interfaz basada en estándares, no importa si los servicios están escritos en Java y el navegador en C++, o los servicios desplegados en un entorno Unix y el navegador lo está en Windows. Los servicios web permiten la interoperabilidad entre plataformas heterogéneas, lo cuál es uno de los beneficios clave que se obtienen al usarlos.

Muchos e importantes desarrolladores de aplicaciones como IBM y Microsoft han adoptado completamente la tecnología de servicios web. IBM por ejemplo, está integrando el soporte para servicios web en sus productos WebSphere, Tivoli, Lotus y DB2. Igualmente, la nueva plataforma de desarrollo .NET de Microsoft está diseñada y construida en torno a los servicios web.

2.1.4. ¿Cómo son los servicios web?

Los servicios web son una interfaz basada en mensajes. El único requisito de un servicio web es que debe ser capaz de enviar y recibir mensajes usando alguna combinación de protocolos estándar de Internet. La forma más común de un servicio web es la de llamar a procedimientos ejecutados en un servidor, para lo cuál se envía un mensaje que codifica "llama a esta rutina con estos argumentos" y se recibe otro que codifica "aquí están los resultados de la llamada a rutina".

La figura 2.3 ilustra las partes de un servicio web. El código de aplicación alberga toda la lógica de funcionamiento y el código para realizar las tareas. El "listener" interactúa con los protocolos de transporte (HTTP, SOAP, Jabber, …) y recibe las peticiones. El proxy decodifica estas peticiones y las transforma en llamadas al código de aplicación. Luego el proxy puede volver a codificar la respuesta y mandarla al "listener" pero no es un paso obligatorio.

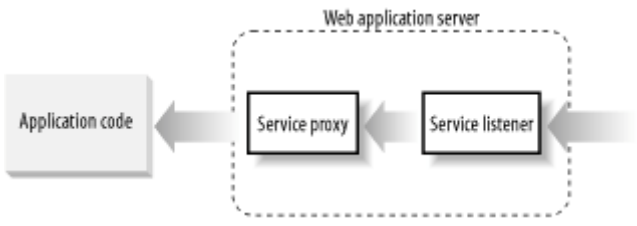

Figura 2.3 Partes de un servicio web

Los componentes proxy y "listener" pueden ser aplicaciones aisladas, como por ejemplo demonios servidor de TCP o HTTP, o bien pertenecer al contexto de cualquier otro tipo de servidor de aplicaciones. Por ejemplo, el Servidor de Aplicaciones WebSphere de IBM incluye soporte nativo para la recepción de mensajes SOAP sobre

HTTP y usarlos para invocar aplicaciones Java que hayan sido previamente desplegadas. El popular servidor web Apache dispone asimismo de un módulo que implementa SOAP. Hay incluso implementaciones de SOAP para los sistemas operativos Pocket PC y Palm de las PDAs.

Hay que tener en cuenta, no obstante, que un servicio web no requiere de un sistema de servidores para existir. Un servicio web puede ser desplegado en cualquier plataforma donde puedan usarse las tecnologías estándar de Internet. Esto significa que un servicio web puede ser albergado o utilizado tanto por un proveedor de servicios como por una PDA.

Por otra parte, los servicios web no requieren que las aplicaciones cumplan con el tradicional modelo cliente-servidor (donde el servidor alberga los datos y realiza todo el procesamiento) o con el modelo por capas (donde el alojamiento de datos, el procesamiento y la interfaz están en estratos diferentes), aunque desde luego se han desplegado en gran medida en estos tipos de arquitecturas. Los servicios web pueden tomar cualquier forma, pueden ser usados en cualquier lugar y pueden servir a variados propósitos. Por ejemplo, hay una importante sinergia entre los sistemas peer-to-peer (igual a igual) y los servicios web, donde los iguales usan protocolos estándar de Internet para comunicarse entre sí y proporcionarse servicios mutuamente.

2.1.5. Integración en tiempo real

Una vez comprendidos los fundamentos de los servicios web pasamos a detallar uno de los aspectos que hacen a éstos tan interesantes: la integración dinámica de servicios. Esto es, la integración en tiempo de ejecución de servicios para conseguir el objetivo de la aplicación final, y sin tener en cuenta los detalles específicos de la implementación en una tecnología determinada de cada uno de los servicios web.

Esta integración en tiempo real eleva el modelo de desarrollo de aplicaciones de Internet a un nuevo nivel, denominado arquitectura de servicios web.

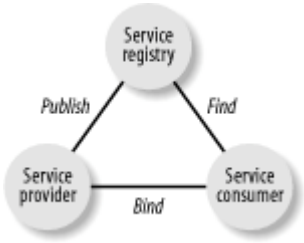

Figura 2.4 Arquitectura de servicios web

En la arquitectura de servicios web, el Proveedor de Servicios publica una descripción del servicio que ofrece usando el Registro de Servicios. El Consumidor del Servicio busca en el Registro un servicio que cumpla con sus necesidades. Dicho Consumidor podría ser tanto una persona como un programa.

La rama "Bind" indica que un Consumidor está utilizando los servicios ofrecidos por un Proveedor. La clave de la integración en tiempo real es que este enlace puede

ocurrir en cualquier momento, en particular en tiempo de ejecución. Es decir, un cliente podría no saber qué procedimientos llamará hasta que se esté ejecutando, busque en el registro y encuentre un candidato.

2.1.6. El modelo de capas de los servicios web

La arquitectura de los servicios web se implementa en torno a cinco tipos de tecnologías, organizadas en capas una sobre otra, tal y como aparece en la figura 2.5

Figura 2.5 Modelo de capas

No es sorprendente que este modelo sea muy similar al modelo de red TCP/IP con el que se describe la arquitectura de las aplicaciones basadas en Internet.

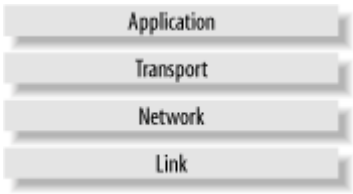

Figura 2.6 Modelo de red TCP/IP

En el modelo de servicios web aparecen tres capas adicionales: empaquetado, descripción y descubrimiento, que son las esenciales para proporcionar la capacidad de integración en tiempo real, así como para obtener la independencia de plataforma.

Dado que cada capa del modelo trata con un determinado problema, sólo hay que implementar aquellas que sean necesarias para nuestra aplicación. Cuando haga falta una nueva capa, no hay que reescribir de nuevo toda la infraestructura para dar soporte a una nueva forma de intercambio de información o a un nuevo sistema para autenticar usuarios.

El objetivo es la total modularización del entorno de programación distribuida, en oposición a las soluciones monolíticas de otras plataformas distribuidas más tradicionales como CORBA y COM. Este aspecto es especialmente importante en los servicios web dada la rápida evolución de los estándares

2.1.6.1. Más allá del modelo de capas

Las capas descritas no solucionan todos los problemas que pueden darse. Por ejemplo, no se ocupan de la seguridad, identificación y de otros aspectos tan importantes en un mercado global electrónico. Para solucionar estos aspectos se han desarrollado varios estándares:

- **XML Protocol**

  Es el estándar que está desarrollando el W3C para reemplazar a SOAP como el estándar de facto de los servicios web. Puede obtenerse más información en la web del grupo de trabajo:

  http://www.w3.org/2000/xp/Group/

- **XKMS**

  La XML Key Management Specification es un conjunto de servicios de seguridad y confiabilidad que añade la infraestructura de clave privada (PKI) a los servicios web. Puede obtenerse más información en la siguiente web:

  http://www.w3.org/TR/xkms/

- **SAML**

  Security Assertion Markup Language. Se trata de una gramática XML para expresar la ocurrencia de eventos de seguridad, como por ejemplo un evento de autenticación. Usada en la arquitectura de servicios web proporciona un sistema de autenticación flexible y estándar.

  http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security

- **XML-Dsig**

  XML Digital Signatures. Permite que cualquier documento XML pueda ser firmado digitalmente. Puede encontrarse información adicional en la siguiente web:

  http://www.w3.org/TR/xmldsig-core/

- **XML-Enc**

  XML Encryption. Esta especificación permite que los documentos XML puedan ser encriptados y expresar datos encriptados como XML. La página web de la especificación es la siguiente:

  http://www.w3.org/TR/xmlenc-core/

- **XSD**

XML Schemas. Son una aplicación de XML utilizada para expresar la estructura de documentos XML.

http://www.w3.org/XML/Schema

- **P3P**

Platform for Privacy Preferences. Es una gramática XML desarrollada por el W3C para expresar políticas de privacidad de datos. Puede encontrarse más información en la siguiente web:

http://www.w3.org/P3P/

- **WSFL**

Web Service Flow Language. Es una extensión a WSDL que permite expresar work flows en la arquitectura de servicios web. La especificación 1.0 de dicho lenguaje puede leerse en el siguiente documento PDF:

http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

- **Jabber**

Jabber es un nuevo protocolo de transporte ligero y asíncrono usado en aplicaciones peer-to-peer. El programa del mismo nombre, Jabber, es conocido por ser el "Linux de la mensajería instantánea" y su web es la siguiente:

http://www.jabber.org/

El IETF (Internet Engineering Task Force) ha formalizado los protocolos de flujos de datos de Jabber y los ha aglutinado bajo una nueva tecnología de mensajería instantánea y presencial llamada XMPP (eXtensible Messaging and Presence Protocol), de la que puede obtenerse más información en la siguiente dirección:

http://www.xmpp.org/

- **ebXML**

Es un conjunto de especificaciones basadas en XML para dar soporte a un emergente mercado electrónico. Usando SOAP, ofrece una solución

para implementar la integración de servicios business-to-business. Su web es la siguiente:

http://www.ebxml.org/

2.1.6.2. La capa de Descubrimiento

La capa "Discovery" proporciona mecanismos para que los clientes obtengan información descriptiva acerca de los proveedores. Uno de los mecanismos de descubrimiento más ampliamente reconocidos es el proyecto UDDI (Universal Description, Discovery and Integration) y es el objeto del presente proyecto.

IBM y Microsoft han planteado una alternativa a UDDI, el Web Service Inspection Language (WS-Inspection). Puede obtenerse más información en su página web:

http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html

2.1.6.3. La capa de Descripción

Cuando se implementa un servicio web, se debe decidir qué protocolos de red, transporte, etc… se usarán. La descripción de dicho servicio permite que un consumidor pueda contactar y usar el servicio.

El estándar de facto para proporcionar estas descripciones es el Web Service Description Language (WSDL). La página web de la especificación es la siguiente:

WSDL: http://www.w3.org/TR/wsdl

2.1.6.4. La capa de Empaquetado

Para que los datos de la aplicación puedan ser trasladados sobre la red por la capa de transporte, éstos deben ser "empaquetados" en un formato que entiendan todos los agentes de la comunicación. Otras formas de referirse a este proceso son "marshalling" y "serialización".

HTML es un tipo de formato de empaquetado, pero no es muy conveniente ya que atiende más a aspectos de presentación de la información que al significado. XML, en cambio, es la base para multitud de formatos de empaquetado de servicios web porque puede ser usado para representar el significado de la información transferida, y porque actualmente abundan los "parsers" XML.

SOAP es un formato de empaquetado muy común, construido sobre una base XML. De hecho, aunque hay otros protocolos de empaquetado basados en XML, como XML-RPC, el formato SOAP es el más utilizado actualmente.

2.1.6.5. La capa de Transporte

La capa de transporte se compone de todas las tecnologías que hacen posible una comunicación directa entre aplicaciones, justo encima de la capa de Red. Estas tecnologías incluyen protocolos como TCP, HTTP, SMTP, etc… El objetivo primordial de la capa de transporte es la de trasladar datos entre dos o más localizaciones de la red. Los servicios web se pueden implementar sobre casi cualquier protocolo de transporte.

La elección del protocolo de transporte estará basada en las necesidades de comunicación que tenga el servicio web que se desee implementar. Así por ejemplo, HTTP proporciona el protocolo más adecuado para evitar las restricciones de cortafuegos de muchos sistemas, al utilizar el puerto 80, que debe estar abierto para la navegación web, pero sin embargo no disponemos de comunicación asíncrona. En el caso contrario, XMPP (Jabber) proporciona una excelente comunicación asíncrona, aunque puede tener más problemas de utilización.

2.1.6.6. La capa de Red

La capa de red de la arquitectura de servicios web es exactamente la misma que la del modelo TCP/IP. Proporciona los mecanismos básicos de comunicación, direccionamiento y enrutado.

2.1.7. El modelo de servicios igual a igual

El modelo de servicios igual a igual es una visión alternativa y complementaria de la arquitectura de servicios web. Basada en la arquitectura peer-to-peer (P2P), cada miembro de un grupo de iguales comparte una colección común de recursos y servicios. Cuando nos referimos a un igual, éste puede ser una persona, un dispositivo, una aplicación, u otro grupo de iguales actuando como uno solo.

En este modelo, y aunque no sea del todo evidente, existen los mismos componentes básicos que en el modelo anterior. Existen proveedores y consumidores de servicios, e incluso hay registros de servicios. De todas formas, en este caso la distinción entre proveedor y consumidor no es tan evidente. Dependiendo del tipo de recurso o servicio, un mismo par (o igual) puede actuar como proveedor o como consumidor. Esto hace del modelo de servicios igual a igual un sistema más flexible y dinámico que el anterior.

La implementación más extendida del modelo de servicios web igual a igual es el de la mensajería instantánea. En este sistema, cada persona que utiliza la mensajería instantánea es un igual. Cuando se manda una invitación para iniciar una conversación, se está actuando como consumidor de servicios. Cuando alguien se registra en el servidor del sistema de mensajería, se están utilizando los recursos de un registro de servicios web, es decir, el servidor actúa como un registro al anotar donde están los pares y cuáles son sus capacidades de mensajería. Cuando se acepta una invitación para iniciar una conversación se está actuando como proveedor de servicios. La figura 2.7 ilustra este modelo:

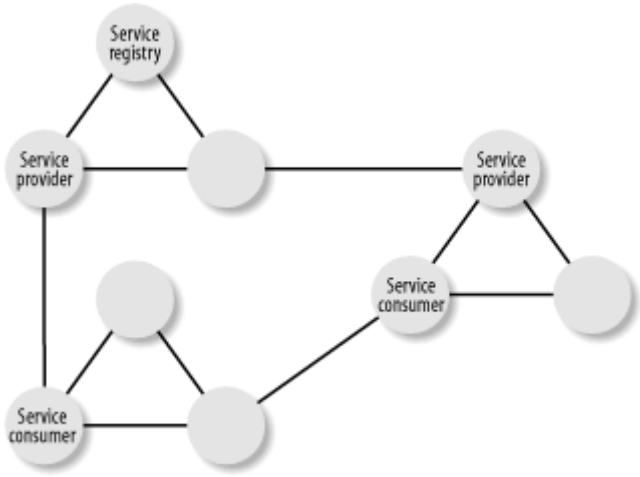

Figura 2.7 Red peer-to-peer de servicios web

Los servicios peer-to-peer y los servicios web emergieron y evolucionaron cada uno por su lado, haciendo uso de protocolos y tecnologías diferentes para implementar sus respectivos modelos. El modelo de servicios web peer-to-peer une ambos mundos unificando las tecnologías, los protocolos y los modelos en una visión conjunta global.

2.2. SOAP

2.2.1. Introducción

Como vimos antes el modelo de servicios web se compone de una variedad de capas. SOAP es un protocolo estandarizado de empaquetado para los mensajes compartidos entre aplicaciones. La especificación sólo define un envoltorio (envelope) basado en XML para la información que se transfiere, así como un conjunto de reglas para trasladar tipos de datos específicos de la aplicación y de la plataforma a una representación XML. El diseño de SOAP hace que sea muy indicado en una gran variedad de aplicaciones, lo que ha permitido su gran popularidad.

2.2.2. SOAP y XML

SOAP es una aplicación de la especificación XML. Para su definición y funcionamiento se apoya en estándares XML como XML Schema y XML Namespaces.

Gracias a XML Messaging (mensajería XML) las aplicaciones pueden intercambiar información usando documentos XML, como puede verse en la figura 2.8. Proporciona una manera flexible de comunicación y conforma la base de SOAP. Un mensaje puede ser cualquier cosa: una petición para conocer el tiempo meteorológico, una orden de compra, una consulta a una base de datos, un listado de asientos libres de un cine, o cualquier otro tipo de información relevante para una determinada aplicación.

Figura 2.8 Mensajería XML

Dado que XML no está vinculado a ninguna aplicación, sistema operativo o lenguaje de programación particular, la mensajería XML puede usarse en cualquier tipo de entorno. Por ejemplo, un programa escrito en Perl bajo un entorno Linux podría comunicarse mediante un mensaje XML con otro escrito en Java en una plataforma Windows y afectar de dicha forma a su comportamiento. SOAP proporciona una manera estándar de estructurar dichos mensajes.

La mensajería XML, y por tanto SOAP, tiene dos aplicaciones relacionadas: RPC y EDI.

RPC (Remote Procedure Call) es la base para la computación distribuida, la manera en que una aplicación hace una llamada a procedimiento, método o función de otra aplicación remota, pasando los argumentos y recogiendo los resultados.

EDI (Electronic Data Interchange) es la base para las transacciones automatizadas de negocios. Define una manera estándar para transmitir documentos financieros y comerciales, así como los mensajes relacionados.

Si se usa SOAP para EDI, lo que se conoce como "document-style" en SOAP, el mensaje será una orden de compra o similar. Si se usa para RPC, se conoce como "RPC-style" y el mensaje será una llamada con parámetros o una respuesta con los valores de retorno.

2.2.3. SOAP Messages

Un mensaje SOAP consiste en un envoltorio (envelope) que contiene una cabecera opcional y un cuerpo del mensaje, como puede verse en la figura 2.9.

Figura 2.9 La estructura de mensajes de SOAP

La cabecera es el mecanismo mediante el cuál SOAP puede ser ampliado para incluir nuevas características como la seguridad o la gestión de transacciones. Además puede ser usado para incluir parámetros de calidad del servicio asociado con el cuerpo del mensaje. Esta cabecera (SOAP header) es el primer elemento hijo de SOAP envelope. Los hijos de SOAP header se interpretarán como Header Blocks.

La información incluida en el header es utilizada en algún lugar a lo largo del camino que lleva el mensaje del origen al destino y en general, no se va a emplear en el procesador SOAP de destino, aunque no obstante puede ser utilizado. Puede incluir atributos como el tipo de codificación URI, nombre de usuario y contraseña para autenticación, un identificador para mensajería fiable o incluso una firma digital para seguridad adicional. Los header incluyen un atributo llamado "mustUnderstand". Con dicho atributo el que envía el mensaje exige al receptor que comprenda todos los campos de la cabecera. Si el receptor no comprende alguno debe desechar el mensaje SOAP completo y devolver una SOAP Fault con el código de error apropiado.

El cuerpo contendrá el mensaje que deberá enviarse y procesarse. Cualquier cosa que pueda expresarse en sintaxis XML puede ir en el cuerpo del mensaje SOAP. El SOAP body debe ser hijo directo del SOAP envelope y estar a continuación del SOAP header. La semántica del contenido del body estará en un Esquema SOAP asociado.

La sintaxis XML que se sigue para expresar un mensaje SOAP está basado en el namespace soap-envelope, que puede obtenerse en la siguiente dirección:

http://www.w3.org/2001/06/soap-envelope

2.2.4. SOAP Faults

Un SOAP Fault es un tipo de mensaje SOAP específicamente creado para expresar errores que hayan podido producirse durante el procesamiento de otro mensaje SOAP.

Un SOAP Fault tiene la estructura que se muestra en el siguiente ejemplo:

```
<s:Envelope xmlns:s="http://www.w3.org/2001/06/soap-envelope">
   <s:Body>
      <s:Fault>
        <faultcode>Cliente.Autenticacion</faultcode>
        <faultstring>
          Password incorrecto
        </faultstring>
        <faultactor>http://www.test.com</faultactor>
        <details>
        <!—otros detalles especificos de aplicacion -->
        </details>
      </s:Fault>
   </s:Body>
</s:Envelope>
```

Podemos identificar cuatro bloques en el cuerpo de un SOAP Fault:

- **faultcode**

  Se trata de un código generado de forma algorítmica y que identifica de forma unívoca el error generado. El valor debe ser un "Qualified Name", es decir, que el nombre del código sólo tiene significado dentro de un cierto espacio de nombrado XML.

- **faultstring**

  Es la versión del bloque anterior, pero dispuesto para que pueda ser fácilmente asimilado por un lector humano.

- **faultactor**

  Identificador único del nodo de procesamiento de mensajes SOAP donde se generó el error.

- **details**

Se usa para escribir detalles específicos de la aplicación acerca del error que se ha generado. Debe existir si el error está directamente relacionado con algún problema con el cuerpo del mensaje SOAP transmitido. No debe ser usado para indicar un error producido por cualquier otra causa del procesamiento del mensaje.

2.2.5. SOAP y RPC

RPC es la aplicación SOAP más extendida en el momento actual. Veamos cómo se crea un mensaje para invocar un método remoto y cómo obtenemos los resultados.

2.2.5.1. Invocando métodos

Las reglas para empaquetar una llamada a procedimiento remoto en un envoltorio SOAP son las siguientes:

- La llamada a procedimiento remoto se representa mediante una estructura en la que cada parámetro de entrada o de entrada/salida se modela como un campo en dicha estructura.

- Los nombres y el orden en el que se colocan los parámetros en la estructura deben ser igual a como aparece en la definición del método invocado.

Así, por ejemplo, un método Java definido de la siguiente manera:

```
String compruebaCredenciales(String login, String password);
```

Que podría invocarse como se indica:

```
Resultado = compruebaCredenciales(“pepe”,”hd62dj”);
```

Tiene una representación SOAP:

```
<s:Envelope xmlns:s=" http://www.w3.org/2001/06/soap-envelope">
  <s:Body>
    <compruebaCredenciales xmlns="..."
     s:encodingStyle="http://www.w3.org/2001/06/soap-encoding">
       <login xsi:type="string">pepe</login>
       <password xsi:type="string">hd62dj</password>
    </compruebaCredenciales>
  </s:Body>
</s:Envelope>
```

Las convenciones SOAP RPC no requieren el uso del estilo de codificación soap-encoding, ni del tipado explícito de datos vía xsi:type, pero es ampliamente utilizado.

2.2.5.2. Recogiendo los resultados

La respuesta de los métodos es similar a las llamadas en el aspecto de que la respuesta se modela como una estructura con un campo por cada parámetro de salida o entrada/salida que aparece en la definición del método.

Si la llamada al método anterior hubiera devuelto un String con un valor "correcto", el mensaje SOAP de respuesta podría el siguiente:

```
<s:Envelope xmlns:s="...">
  <s:Body>
    <compruebaCredencialesResponse
    s:encodingStyle="http://www.w3.org/2001/06/soap-encoding">
      <return xsi:type="xsd:string">correcto</return>
    </compruebaCredencialesResponse>
  </SOAP:Body>
</SOAP:Envelope>
```

El nombre de la estructura (compruebaCredencialesResponse) no es importante, pero por convención se pone un "Response" tras el nombre de la estructura de invocación del método. Igualmente, el nombre del elemento "return" es arbitrario, podría llamarse de cualquier otra manera, ya que se asume que el valor de retorno estará contenido en el primer elemento de la estructura de respuesta.

2.2.5.3 Indicando errores

La convención SOAP RPC hace uso de los mensajes SOAP Fault como el método estándar para devolver respuestas de error a los clientes RPC. El mensaje SOAP Fault se usa para indicar la naturaleza exacta del error producido, y puede ser extendido para expresar detalles adicionales, usando el elemento "details".

2.2.6. Transporte

Como se ha mencionado antes, SOAP es un protocolo de empaquetado que reside en la arquitectura de servicios web encima de las capas de transporte y de red. Dado que es un protocolo de nivel superior, no tiene que considerar qué tipo de protocolo de transporte se está usando por debajo. De hecho, la implementación SOAP::Lite, basada en Perl, permite enviar mensajes SOAP sobre una gran variedad de protocolos de transporte: HTTP, FTP, TCP plano, SMTP, POP3, MQSeries y Jabber. Esto hace de SOAP un protocolo muy flexible y popular.

2.2.6.1. SOAP sobre HTTP

Dado su uso en Internet, HTTP es, de lejos, el protocolo de transporte más extendido para intercambiar mensajes SOAP. La especificación SOAP, de hecho, da un tratamiento especial a este protocolo, definiendo un mapa para traducir la semántica de los mensajes SOAP a HTTP.

SOAP sobre HTTP es una manera natural de implementar las convenciones SOAP RPC dado que HTTP es también un protocolo de petición y respuesta. El mensaje de petición SOAP se traduce en un HTTP POST al servidor HTTP, mientras que el servidor devuelve la respuesta SOAP en la respuesta HTTP. Puede verse más claro en la figura 2.10

Figura 2.10 SOAP sobre HTTP

Un ejemplo de petición HTTP conteniendo un mensaje SOAP puede ser el siguiente:

```
POST /Registry HTTP/1.1
Content-Type: text/xml
Content-Length: nnnn
SOAPAction: "urn:Registry#GetBusinessDetail"

<s:Envelope xmlns:s="http://www.w3.org/2001/06/soap-envelope">
  ...
</s:Envelope>
```

Con lo que se obtendría una respuesta HTTP conteniendo otro mensaje SOAP:

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: nnnn

<s:Envelope xmlns:s="http://www.w3.org/2001/06/soap-envelope">
  ...
</s:Envelope>
```

La cabecera HTTP “SOAPAction” está definida por la especificación SOAP e indica el propósito de la petición HTTP. Su valor es absolutamente arbitrario, pero su objetivo es indicar al servidor HTTP qué es lo que pretende hacer el mensaje SOAP embebido antes de que se decodifique el contenido del mismo. De este modo, el servidor HTTP puede rechazar rápidamente peticiones inaceptables.

Aunque HTTP sea el protocolo de transporte más popular para transmitir mensajes SOAP, no está exento de problemas, ya que no fue diseñado para transportar documentos XML. De hecho, la plataforma .NET de Microsoft usa SOAP sobre mensajería instantánea, lo que puede poner en aprietos la tradicional supremacía de HTTP en este campo.

2.3. UDDI

2.3.1. Introducción

Una vez diseñado y construido un servicio web, hay que establecer un mecanismo para que pueda ser descubierto por los potenciales clientes. Y éste es precisamente el objetivo del registro UDDI (Universal Description, Discovery and Integration), establecido por un consorcio industrial, como veremos en el apartado 2.3.2, para crear un directorio de servicios web. Un registro UDDI acepta información descriptiva acerca de una entidad, así como de los servicios web que ofrece, y permite a las partes interesadas realizar búsquedas online y extraer la información necesaria.

Para acudir a un establecimiento y adquirir algo, se necesita algo de información acerca de dicho establecimiento: su dirección postal, su teléfono, su página web o incluso su servicio web. Podemos obtener esta información directamente de un representante, o mediante una tarjeta comercial, folleto, e-mail o similar. También se puede obtener esta información mirando en un listín telefónico.

De forma similar, la información que un programa, ejecutándose de forma local, necesita para comunicarse con otro programa remoto al que se pueda acceder a través de la web, debe publicarse en algún lugar. UDDI actúa como unas páginas blancas y amarillas para los servicios web. Además, permite a los desarrolladores de aplicaciones interactuar con el registro tanto en tiempo de diseño como de ejecución.

La figura 2.11 ilustra el servicio UDDI público, gestionado por IBM, Microsoft, SAP, HP, y otras posibles instituciones. Cada institución ofrece una base de datos públicamente accesible con la información de todos los proveedores previamente registrados. Esta información se registra mediante peticiones SOAP contra uno de los centros de datos de estas instituciones, que luego se ve replicada en los demás.

Cada uno de los centros de datos previamente mencionados se hacen llamar “Operator sites”, y los programadores que tratan con UDDI usan las APIs de su especificación para comunicarse con uno o más de estos “sites”.

2.3.2. La organización de UDDI

UDDI comenzó como una colaboración entre IBM, Microsoft y Ariba para promocionar la adopción y el uso general de los estándares de los servicios web. Estas compañías fundaron UDDI.org e invitaron a otras a participar: algunas con derechos como fundadoras y otras con labores consultivas. Las compañías invitadas como fundadoras establecieron las reglas básicas, así como las especificaciones iniciales y requisitos.

Los tres fundadores originales fueron también los operadores originales del repositorio UDDI público inicial. Más tarde, Hewlett-Packard se unió al proyecto,

sustituyendo a Ariba como uno de los operadores, y SAP se unió como un nuevo operador.

Más de 300 compañías se han unido a UDDI.org apoyando el esfuerzo de establecer un listado de servicios web públicamente accesibles. Quince de estas compañías conforman el Grupo de Trabajo, responsable de establecer la hoja de ruta del proyecto, resolver conflictos y tomar decisiones. Las demás son parte del órgano consultivo, lo que les permite opinar acerca de un borrador antes de que se convierta en una especificación en firme.

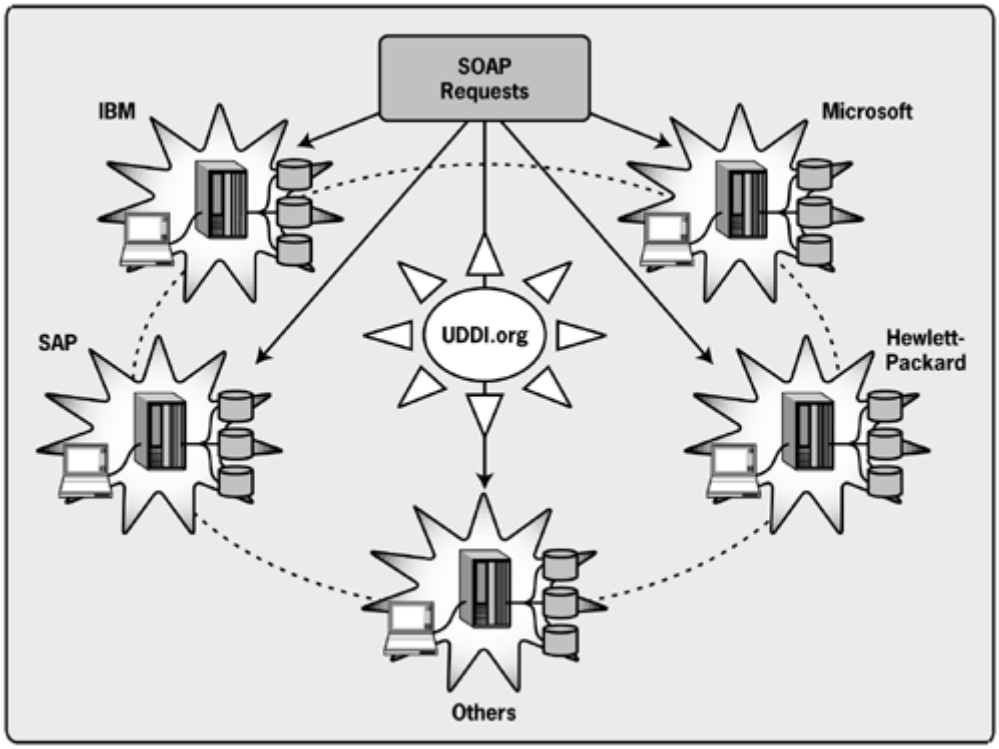

Figura 2.11 Servicio UDDI público

Los operadores, además del registro definitivo, o de producción, disponen de otro registro de prueba, permitiendo que los proveedores de servicios web prueben sus clientes UDDI antes de remitir la información al registro de producción. Desde luego, para publicar información real en el registro de producción, la entidad que pretende hacerlo debe estar autorizada a ello, y los datos deben ser validados. Para evitar problemas, la comunicación con UDDI requiere en este caso de mensajes encriptados mediante el uso de HTTPS. Uno de los aspectos en los que se trabaja en la última versión de UDDI, la v3, es el de implementar un mejor sistema de seguridad.

2.2.3. Conceptos esenciales

El registro UDDI público funciona un poco como el servicio de nombres de dominio de Internet (DNS). Las entidades pueden registrarse en el centro de datos de cualquier Operador (IBM, Microsoft, HP, …). A intervalos regulares, normalmente de

noche, la información contenida en uno de los registros se replica en los demás, asegurando la consistencia. De esta forma, un usuario que solicite la información registrada acerca de una determinada entidad, obtendrá el mismo resultado sea cual sea el Operador al que haga la petición. En cualquier caso, cuando se desee actualizar la información registrada, el agente que la hizo debe cursar la operación de actualización contra el Operador original.

Por razones obvias, la seguridad es un objetivo principal para los miembros de UDDI.org. Las entidades están autorizadas a actualizar su información por uno de los Operadores y por tanto, deben volver a dicho Operador cada vez que necesiten actualizar o borrar sus datos. Las entidades que quieran registrarse en UDDI deben obtener primero una autorización para hacerlo, y debe estar aprobada por al menos uno de los Operadores. Dicha aprobación implica el envío de un token de autorización a la entidad, lo cuál le permite entrar en el registro y guardar, actualizar o borrar sus datos. La autorización para acceder al registro es gestionada por cada Operador de manera independiente, es decir, los datos de autenticación en un Operador no tienen por qué servir para entrar al registro por otro. A efectos prácticos, una entidad que quiere registrarse en UDDI, debe elegir un Operador y permanecer en él.

Otro de los objetivos primordiales es el de la calidad, o fiabilidad, de los datos almacenados. Alguien debe asegurar que la entidad que va a ser registrada es una entidad real, que el nombre, su identificador, categoría, números de teléfono, página web y dirección postal, entre otros datos identificativos, son correctos. La versión 2 de UDDI permite utilizar a un tercero para validar todos los datos insertados.

Hoy en día, aunque UDDI ya está en estado de producción, la calidad de los datos no cumple con lo que era de esperar. La falta de terceros que trabajen con UDDI en el proceso de validación hace que la organización UDDI.org en sí sólo proporcione un mecanismo básico en el que se comprueban los identificadores de categoría y poco más. En otras palabras, no hay asegurada la autenticidad de la información almacenada en el registro.

UDDI tiene dos partes principales: publicación y descubrimiento. La parte de publicación significa que una entidad puede registrar su información para que otras entidades puedan buscar y descubrir sus servicios, que es la otra parte. Las entidades o individuos pueden acceder al registro e interactuar con él, bien mediante los mensajes SOAP definidos en las APIs, o mediante algún tipo de interfaz de usuario proporcionada por un Operador. Los mismos Operadores registran accesos a los ficheros WSDL que definen sus servicios web de publicación y consulta. UDDI proporciona dos ficheros WSDL diferentes para los servicios de publicación y consulta, usando un formato de documento XML propio.

Algunos métodos definidos en las APIs SOAP de UDDI se usan para registrar información, y otros para buscarla, navegar por ella y recogerla. Las APIs SOAP de UDDI requieren un subconjunto específico de SOAP, esto es, UDDI no usa el formato de serialización opcional de SOAP y otras características definidas en la especificación SOAP.

2.2.4. Funcionamiento de UDDI

La información almacenada en un registro UDDI a menudo se describe organizándola en tres principales categorías:

- **Páginas Blancas**

  La información es muy similar a la encontrada en un directorio telefónico, que incluye nombre, teléfono y dirección.
  Se guardan datos como el nombre de la entidad (Business Name), descripciones textuales multilingüe, dirección postal, información de contacto, nombre y dirección del sitio web, identificadores según alguna taxonomía, como DUNS (Data Universal Numbering System), etc.

- **Páginas Amarillas**

  Similar al equivalente telefónico, incluye categorías de catalogación industrial (como NAICS), de productos (como UN/SPSC), de ubicación geográfica (ISO 3166). Mediante el uso de claves y códigos predeterminados, las entidades pueden registrar su información de forma que la búsqueda sea muy fácil para otros servicios.

- **Páginas Verdes**

  La sección verde contiene la información técnica acerca de los servicios ofrecidos, como la forma en la que se debe interactuar con ellos, definiciones del proceso que se realiza, etc. También debe incluirse, si existe, un puntero al fichero de definición del servicio web ofrecido WSDL. La información incluida en esta categoría incluye todo lo relativo a las características y funcionalidades del servicio, así como un identificador único del mismo.

La estructura de datos en UDDI se organiza usando tipos complejos de esquemas XML. Estos esquemas permiten una gran flexibilidad en los datos almacenados, así como facilidad para extenderlos. De hecho, proporciona tantas opciones y extensiones que es casi imposible predecir el nivel de detalle con el que se registra una entidad. Si UDDI quiere tener un futuro, los datos registrados tendrán que ser regulados y normalizados de una mejor manera a como se hace ahora. Aunque UDDI.org está enfrentando este problema mediante la publicación de documentos de buenas prácticas y proporcionando asistentes para el proceso de registro, el problema aún existe.

La información de clasificación e identificación es útil para localizar fácilmente los negocios que cumplan con los requisitos del cliente. Una entidad puede insertar cualquier número de clasificaciones en su registro con el propósito de facilitar las búsquedas. Esta información de clasificación e identificación incluye cosas como códigos industriales IRS, números Dun and BrandStreet (DUNS), códigos de productos, geográficos, etc.

Algunas de las taxonomías de clasificación para información categorizada son:

- **North American Industry Classification System (NAICS)**

  http://www.census.gov/epcd/www/naics.html

- **Universal Standard Products and Services Classification (UNSPSC)**

  http://www.unspsc.org/

- **International Organization for Standardization (ISO) 3166**

  http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html

  Los Operadores suelen validar la información categorizada mediante estos códigos comprobando que los datos introducidos sean válidos, confrontando los códigos industriales via NAICS, los de productos y servicios via UNSPSC y los de información geográfica via ISO 3166. De todas formas la inclusión de cualquiera de estos códigos es opcional, así como su comprobación durante el proceso de registro. Se pueden usar otras taxonomías de clasificación, pero no serán validadas.

2.2.4.1. Modelo de datos

La información contenida en un registro UDDI está incluida en alguno de los siguientes tipos de estructuras de datos:

- **businessEntity**

  La estructura de más alto nivel. Describe el negocio o entidad que va a insertar datos en el registro. Las demás estructuras están relacionadas con ésta mediante referencias.

- **businessService**

  Incluye el nombre y descripción del servicio ofrecido por la entidad de nivel superior.

- **bindingTemplate**

  Información acerca del servicio, incluyendo la dirección de un punto de entrada para el acceso al mismo por los clientes.

- **tModel**

Una especie de huella dactilar, colección de información que identifica de forma unívoca las especificaciones del servicio. Pueden hacerse búsquedas primarias consultando por un tipo específico de tModel.

- **publisherAssertion**

Una estructura de relación, que asocia dos o más estructuras businessEntity atendiendo a algún tipo de dependencia, como podría ser el de una compañía con su filial, o el de un departamento dentro de una organización más grande.

Cuando los datos se envían por primera vez, UDDI asigna un identificador único a cada una de estas estructuras. Estos identificadores toman la forma de un UUID (Universally Unique Identifier), algunas veces llamado también GUID (Globally Unique Identifier). El formato de los UUID viene descrito en la norma ISO/IEC 11578:1996 Information Technology –Open Systems Interconnection- Remote Procedure Call, que puede obtenerse en la siguiente dirección:

http://www.iso.ch/cate/d2229.html

Un generador de UUID utiliza un algoritmo complejo, que toma como base la fecha y hora actual, así como la dirección de red del ordenador en el que se ejecuta el algoritmo, para obtener un número hexadecimal del que se garantiza la unicidad global. Si no se dispone de una dirección de red, existen otras opciones disponibles en la norma para obtener la unicidad. La dificultad para obtener un número igual con otro ordenador es tan elevada que puede considerarse imposible en la práctica. Un ejemplo de identificador UUID es CBC66F90-C6DA-11D9-AF90-B16C7375F46E.

La figura 2.12 ilustra el modelo de datos de UDDI, en el cuál toda la información derivada de una businessEntity puede obtenerse haciendo las consultas adecuadas. Las relaciones entre estructuras se realizan mediante las llamadas “key references”.

En la figura 2.12 los campos subrayados, como la businessKey, son elementos obligatorios. Es decir, una petición para guardar datos en una de estas estructuras será rechazada si alguno de estos elementos no aparece.

El modelo de datos es una jerarquía donde cada estructura está contenida en otra de nivel superior. La estructura businessEntity es la raíz, o de más alto nivel, es decir, es la que “contiene” a todas las demás. La estructura publisherAssertion fue añadida en la versión 2 de UDDI para permitir que múltiples entradas businessEntity pudieran estar relacionadas de alguna forma, por ejemplo para acomodar a grandes compañías que quisieran registrar sus divisiones o subsidiarias.

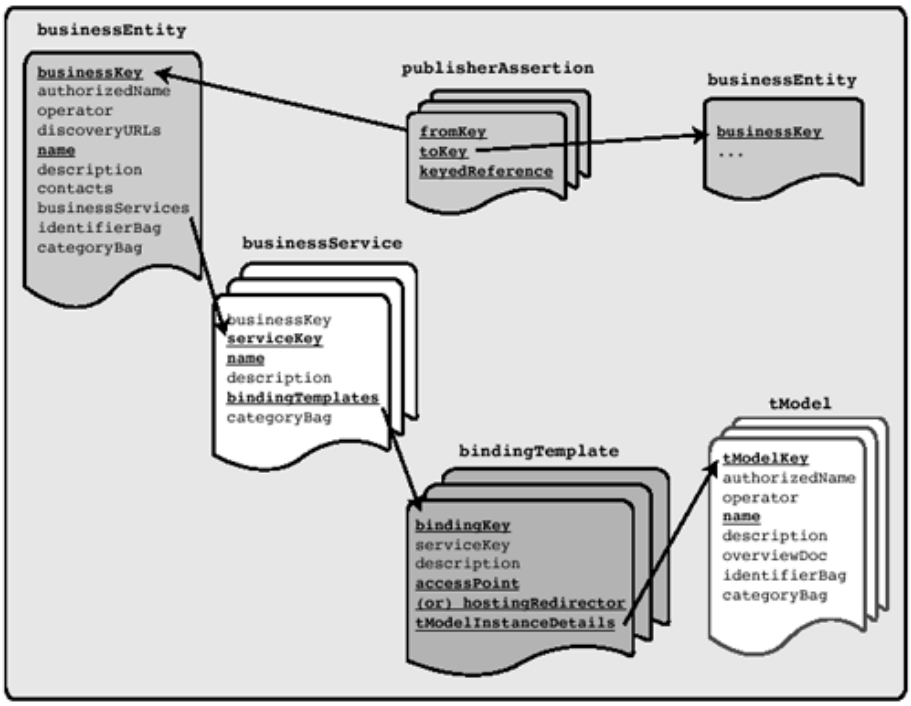

Figura 2.12 Modelo de datos de UDDI

Las flechas en la figura 2.12 indican los elementos usados para hacer las relaciones entre estructuras. La entrada businessServices en la estructura businessEntity es opcional. Si existe contendrá uno o más elementos serviceKey, con los que se indexan las estructuras businessService.

De forma similar, la entrada bindingTemplates en la estructura businessService contendrá uno o más elementos bindingKey, con los que se indexan las estructuras bindingTemplate. Y la entrada tModelnstanceDetails de la estructura bindingTemplate referencia las estructuras tModel.

La información contenida tanto en las estructuras bindingTemplate como en las tModel puede ser combinada para obtener una interfaz de servicio web completa.

2.2.4.2. Datos genéricos

UDDI está diseñado para aceptar cualquier tipo de descripción de servicios web, incluyendo lenguajes de descripción propietarios. WSDL proporciona un lenguaje de propósito general para describir interfaces, protocolos y detalles de despliegue y es un complemento de UDDI, pero no es obligatorio su uso. En otras palabras, las estructuras de datos definidas por UDDI no sólo se usan para almacenar referencias a documentos WSDL, sino que están diseñadas para ser completamente extensibles para contener y referenciar cualquier tipo de documento descriptivo del tipo de servicio que una entidad pueda prestar a otra.

Los Esquemas XML para cada una de estas estructuras no indican ningún tipo de formato especial para los datos guardados en el registro. Esto es, la forma en la que los datos son enviados al registro, mediante estas estructuras XML, pueden diferir de la forma en que se guardan. Esto no importa siempre que las estructuras XML puedan regenerarse a partir de los datos guardados.

Como se muestra en la figura 2.13, las estructuras de datos UDDI proporcionan varios tipos de información que pueden ser guardados y asociados a una businessEntity en particular. Estos datos incluyen identificadores, categorías e información de contacto.

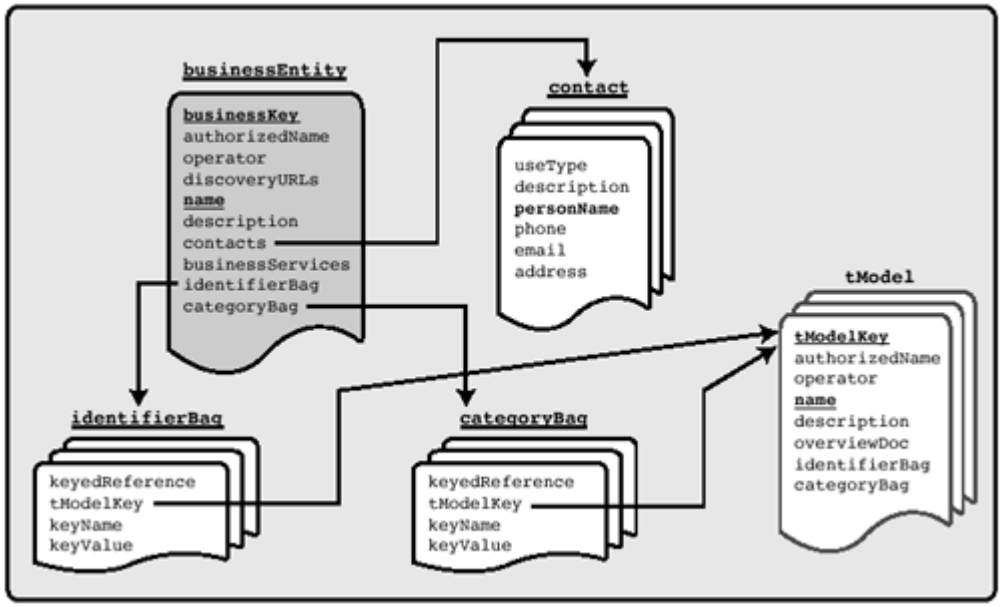

Figura 2.13 Información descriptiva en UDDI

Esta información se agrupa a su vez en otras estructuras que son a su vez secuencias de elementos. La información descriptiva es básicamente la que puede ser buscada en una consulta al registro y a partir de la cuál puede obtenerse información del servicio ofrecido vía referencia a un determinado tModel.

2.2.4.3. La estructura businessEntity

La estructura de más alto nivel, businessEntity, es desde la que parten la mayoría de las consultas al registro. Dependiendo del tipo de información que se desee encontrar a menudo se incluyen claves de identificadores o categorías en la búsqueda.

El siguiente código ilustra la definición XML de una estructura businessEntity. El Esquema XML define un elemento complejo del tipo “sequence”, con los elementos de los nombres indicados y además restringidos por atributos de multiplicidad. El atributo businessKey es obligatorio, y guardará un valor UUID que es generado por el registro cuando se crea la estructura. Por lo tanto, lo mínimo necesario para crear una estructura businessEntity es el nombre de la misma, y es precisamente ese elemento por el que se harán la mayoría de las búsquedas.

```
<element name = "businessEntity">
  <complexType>
    <sequence>
      <element ref = "discoveryURLs" minOccurs = "0"/>
      <element ref = "name" maxOccurs = "unbounded"/>
      <element ref = "description" minOccurs = "0"
                   maxOccurs = "unbounded"/>
      <element ref = "contacts" minOccurs = "0"/>
      <element ref = "businessServices" minOccurs = "0"/>
      <element ref = "identifierBag" minOccurs = "0"/>
      <element ref = "categoryBag" minOccurs = "0"/>
    </sequence>
    <attribute ref = "businessKey" use = "required"/>
    <attribute ref = "operator"/>
    <attribute ref = "authorizedName"/>
  </complexType>
</element>
```

2.2.4.4. La estructura bindingTemplate

Una estructura bindingTemplate proporciona la información necesaria para acceder físicamente al servicio web u otro tipo de servicio registrado en UDDI. Una entidad puede registrar múltiples bindingTemplates para un mismo servicio, y así indicar varios puntos de acceso, lo cuál puede ser bastante útil.

Los tipos de punto de acceso que pueden encontrarse en un bindingTemplate incluyen los siguientes tipos URL:

- mailto
- http
- https
- ftp
- fax
- phone
- other

La información que acompaña al tipo debe formatearse de manera consecuente. Es decir, junto con el tipo mailto debe aparecer una dirección de correo electrónico válida, junto al tipo http una url válida, etc. De esta manera una entidad puede proporcionar el tipo de acceso adecuado para varios mecanismos de contacto. No se realiza ningún tipo de validación para asegurar que el punto de acceso realmente exista.

2.2.4.5. La estructura tModel

En la nomenclatura UDDI, un tModel es el mecanismo para intercambiar metadatos acerca de un servicio web, como su descripción, o un puntero a un fichero WSDL. UDDI es lo bastante general como para albergar cualquier servicio accesible a través de Internet, y es por ello que no está limitado al uso de WSDL. Si WSDL se vuelve popular y se acepta de manera global, la mayoría de los tModel incluirán referencias a este tipo de documentos. Sin embargo por ahora se usan muchos otros tipos de documentos para describir un servicio y por tanto, se aceptan como parte de un tModel.

Un tModel es el equivalente en UDDI a un fichero WSDL, pero con un uso más variado, pudiendo incluir punteros y referencias a servicios usando direcciones distintas de URL y protocolos de transporte distintos de SOAP. A cada tModel se le asigna un identificador UUID en el momento de su almacenamiento por un Operador.

Los tModels están diseñados para identificar de forma unívoca especificaciones de interfaz de servicios web, en el amplio sentido en el que están definidos en UDDI, y para conseguir que sean fácilmente localizados. Proporciona puntos de entrada alternativos en la estructura de datos, de modo que a través de ellos se pueden descubrir servicios y enlazarlos con las entidades que los proveen. Puede darse el caso de que varias entidades expongan el mismo servicio, y es que una vez establecidas firmemente las características de un servicio, nada impide que puedan ofrecerlo distintas entidades.

UDDI define sus propios tModels para definir sus servicios, como los correspondientes a las APIs de consulta y publicación. Además, existen tModels para cada una de las taxonomías utilizadas, como NAICS, UNSPSC e ISO 3166. De esta forma vemos que también podemos emplear los tModels como un espacio de nombrado.

2.2.5. APIs de UDDI

Las APIs de UDDI están divididas entre las que sirven para registrar información y las que se usan para buscarla y navegar por ella. La API de publicación es usada por los denominados “publishers”, esto es, entidades y/o agentes de entidades que cursan peticiones para guardar información en el registro. La API de consulta es usada por los “consumers”, para encontrar y obtener información detallada acerca de una entidad o servicio que cumpla con sus requisitos de búsqueda.

En general, la API de publicación contiene métodos para guardar, actualizar y eliminar información. La API de consulta contiene métodos para obtener información resumida acerca de un conjunto de entradas o información detallada acerca de una entrada del registro en particular.

Cualquiera interesado en utilizar la API de publicación debe primero obtener un token de autenticación a través de un Operador. Cada Operador distribuye tokens de autenticación usando mecanismos propios, de esta manera, en la práctica, un “publisher” sólo interactuará con un único Operador. Es obligatorio usar la API de publicación sobre HTTPS (SSL v3.0) para obtener una encriptación de datos adecuada.

Las versiones de las APIs usadas en cada momento se indican mediante espacios de nombrado. Por ejemplo, el siguiente atributo se utiliza para indicar el uso de la versión 2:

```
xmlns="urn:uddi-org:api_v2"
```

Las APIs SOAP, tanto de consulta como de publicación, sólo pueden usarse mediante HTTP POST. Además, soportan Unicode, lo que requiere el uso de codificación UTF-8. Los "publishers" pueden así registrar entidades y descripciones varias usando múltiples lenguajes.

2.2.5.1. API de consulta

Usando uno o más criterios de búsqueda, la API de consulta permite navegar a través de la información contenida en el registro y obtener información específica acerca de una determinada entidad o servicio una vez conseguido su identificador UUID.

Se pueden usar varios criterios de búsqueda para obtener uno o más resultados. Una manera de proceder típica es la de hacer una consulta genérica que devolverá una lista de entidades (business) que se ajusten a cierta categoría o identificador. Con una entidad ya seleccionada, se hace otra consulta para obtener información de contacto y/o los servicios ofrecidos por ésta. Las consultas pueden hacerse, en general, buscando por un nombre determinado o usando información de categoría o identificadores varios, como códigos industriales o geográficos.

Los métodos definidos en la API de consulta son los siguientes:

- **find_binding**

  Se usa para localizar información específica de enlace (binding) para un determinado businessService. Devuelve un mensaje del tipo "bindingDetail", que incluye puntos de acceso al servicio clasificados por tipo de URL.

- **find_business**

  Es por el que se suele empezar. Se usa para encontrar información acerca de una o más entidades (businesses). Devuelve un mensaje del tipo businessList.

- **find_relatedBusinesses**

  Se usa para encontrar información acerca de entidades relacionadas con otra entidad en particular cuyo identificador UUID se proporciona durante la consulta. Devuelve un mensaje del tipo relatedBusinessList.

- **find_service**

Permite encontrar y obtener información acerca de los servicios ofrecidos por una entidad en particular. Devuelve un mensaje del tipo serviceList.

- **find_tModel**

  Se usa para buscar estructuras tModel, proporcionando una manera de encontrar servicios en el registro que cumplan con las especificaciones de dicho tModel. Devuelve un mensaje del tipo tModelList.

- **get_bindingDetail**

  Busca y devuelve información detallada acerca de un bindingTemplate en particular, permitiendo obtener toda la información necesaria para invocar un servicio ofrecido por una entidad en particular. Devuelve un mensaje del tipo bindingDetail.

- **get_businessDetail**

  Se usa para obtener información detallada acerca de una entidad (business) en particular. Devuelve un mensaje del tipo businessDetail. Normalmente es la segunda consulta que se hace al registro, una vez obtenida una lista de entidades resultado de una búsqueda anterior más general.

- **get_businessDetailExt**

  Se usa para obtener información extendida acerca de una entidad (business) en particular. Devuelve un mensaje del tipo businessDetailExt. Es decir, es lo mismo que get_businessDetail pero proporcionando información adicional, y que dependerá de la implementación que haga de UDDI el Operador.

- **get_serviceDetail**

  Busca y devuelve información detallada acerca de un servicio en particular, permitiendo obtener los identificadores de los bindingTemplate correspondientes. Devuelve un mensaje del tipo serviceList.

- **get_tModelDetail**

  Busca y devuelve información detallada acerca de un tModel en particular. Devuelve un mensaje del tipo tModelDetail.

Los métodos de la API de consulta no devuelven nada si no hay coincidencias para alguno de los criterios de búsqueda. La API incluye una opción para establecer el máximo número de resultados a devolver, así como un indicador, o bandera, para expresar que existen más resultados de los devueltos. Además, es posible utilizar caracteres especiales en el parámetro nombre de find_business, para sustituir cualquier carácter y en cualquier cantidad.

El método find_business, el principal, permite buscar por nombre, identificadores, categorías, tModels e incluso por URLs. En una misma búsqueda pueden introducirse hasta 5 nombres distintos.

El método find_service permite buscar servicios a partir de la entidad padre, indicando su UUID, o mediante una búsqueda por nombre, categorías o tModels. Una vez obtenido el UUID del servicio, se puede utilizar el método get_serviceDetail para obtener información detallada sobre el mismo.

Mediante los métodos find_binding y get_bindingDetail se obtiene toda la información necesaria para invocar un servicio. Se recomienda guardar en caché esta información para no tener que consultarla en cada invocación, y refrescarla cuando sea necesario, por ejemplo cuando falle la invocación.

El método get_businessDetailExt se incluye para que el registro UDDI pueda hacer frente a posibles extensiones. Por norma general, devolverá la misma información que get_businessDetail, salvo que una implementación determinada decida ofrecer una mayor funcionalidad.

En ocasiones el resultado de una consulta provee los datos para realizar la siguiente. Por ejemplo, una consulta get_bindingDetail podría devolver un tModelKey que identifique de forma unívoca un tipo de servicio. Más tarde se podría utilizar este tModelKey para buscar servicios con las mismas especificaciones ofrecidos por otras entidades.

Por último decir que para el uso de la API de consulta no es necesario ningún tipo de autenticación. Igualmente, el canal de comunicaciones no tiene porqué estar encriptado.

2.2.5.2. API de publicación

Los métodos de la API de publicación se usan para guardar, para actualizar y para borrar u ocultar información en el registro. Como en todas las APIs que guardan datos, es necesario que el agente que la utilice guarde la información apropiada para que en consultas posteriores los resultados sean significativos.

Si se pasa una clave UUID en blanco, se indica al registro que la información es nueva y va a guardarse por vez primera. Si se indica una UUID, la información proporcionada sustituirá a la que tenía dicho identificador. Por supuesto, para hacerlo, el agente debe tener derechos sobre la información que va a actualizar. Cuando se registra una businessEntity, el registro crea una URL a partir de la cuál puede obtenerse información detallada acerca de dicha entidad, y a la que puede accederse mediante HTTP POST.

La API de publicación permite que la entidad que vaya a ser guardada esté calificada por cualquier cantidad de códigos de clasificación. Esta clasificación incluye códigos de categorías y códigos geográficos.

Los métodos de la API de publicación son los siguientes:

- **add_publisherAssertions**

  Se usa para añadir asentimientos de relación a un conjunto existente.

- **delete_binding**

  Elimina un bindingTemplate de la estructura bindingTemplates perteneciente a un businessService.

- **delete_business**

  Elimina una entrada businessEntity del registro.

- **delete_publisherAssertions**

  Elimina asentimientos de relación de la colección controlada por una cuenta de "publisher". Una relación sólo es válida cuando está asentida por ambas partes, la eliminación del asentimiento por parte de cualquiera de ellas provoca su invalidación.

- **delete_service**

  Elimina un businessService de la colección businessServices que forma parte de una businessEntity en particular.

- **delete_tModel**

  Se usa para ocultar la información registrada acerca de un tModel. Cualquier tModel oculto de esta manera puede aún referenciarse en el registro de otras estructuras y puede ser consultado mediante un método get_tModelDetail, pero no será mostrado como resultado de una búsqueda mediante find_tModel. No hay manera de borrar totalmente un tModel, salvo mediante petición administrativa.

- **discard_authToken**

  Se usa para informar al Operador de que un token de autenticación previamente concedido ya no es válido y debe denegarse su uso una vez se reciba este mensaje.

- **get_assertionStatusReport**

  Se usa para obtener un informe del estado de las relaciones que involucren cualquiera de las entidades gestionadas por la cuenta de "publisher" indicada. De esta manera, un "publisher" puede obtener el

estado de las relaciones que ha hecho, así como el que otros han hecho con entidades bajo su control.

- **get_authToken**

  Se usa para solicitar un token de autenticación a un Operador. Esos tokens son necesarios para usar cualquiera de los otros métodos de la API de publicación. Es equivalente a una petición de login en el sistema.

- **get_publisherAssertions**

  Se usa para obtener una lista de todas las "publisher assertions" asociadas a una cuenta "publisher" individual.

- **get_registeredInfo**

  Proporciona un resumen con toda la información registrada a nombre de un usuario específico, y basado en el token de autenticación suministrado.

- **save_binding**

  Guarda un nuevo elemento bindingTemplate o actualiza uno ya existente.

- **save_business**

  Guarda un nuevo elemento businessEntity o actualiza uno ya existente. Este método es el que tiene un efecto de más amplio rango de todos los métodos save_xx. La versión 2 de UDDI introduce una nueva funcionalidad, mediante este método es posible referenciar servicios que en origen pertenecen a otra businessEntity. Este tipo de servicios se llaman servicios proyectados, y no se gestionan como parte de la entidad referenciadora, es decir, que si por ejemplo dicha entidad se elimina, el servicio proyectado no se ve afectado mientras no lo haga la businessEntity a la que en realidad pertenece.

- **save_service**

  Guarda un nuevo elemento businessService o actualiza uno ya existente.

- **save_tModel**

  Guarda un nuevo tModel o actualiza uno ya existente.

- **set_publisherAssertions**

  Se usa para establecer todas las "publisher assertion" asociadas a una cuenta "publisher" individual.

Los Operadores pueden establecer límites de espacio para evitar que los agentes guarden demasiada información. Estos límites pueden negociarse con el Operador que se quiera utilizar, pero las especificaciones garantizan los siguientes límites por cuenta de usuario:

- businessEntity por cuenta: 1
- businessService por cuenta: 4
- bindingTemplates: 2
- tModels: 100
- publisherAssertions o business relationships: 10
- tamaño máximo del mensaje: 2 MB

Cada uno de los bindingTemplate registrados debe contener un serviceKey válido que corresponda a un businessService controlado por la misma cuenta de usuario, es decir, con la misma clave de autenticación.

Si un bindingTemplate es asignado a más de un businessService, la relación se determina por orden de ejecución. La última orden es la que prevalece. En cambio, un businessService puede pertenecer a cualquier businessEntity controlada por la misma cuenta de usuario.

2.2.6. Escenario de uso

Imaginemos que la entidad "SkateBoots Company" quiere registrar en UDDI su información de contacto, las descripciones de servicios, y la información de acceso online a los mismos. Los pasos a seguir serían los siguientes:

1. Elegir un Operador con el que trabajar. Cada Operador tiene sus términos y condiciones para autorizar el uso de su réplica del Registro, y puede proporcionar servicios de valor añadido. Hay que tener en cuenta que para actualizar, modificar o borrar la información almacenada hay que volver a usar el Operador en el que se hizo el registro.

2. Implementar un cliente o utilizar alguno proporcionado por el Operador para acceder al registro.

3. Obtener un token de autenticación de parte del Operador.

4. Registrar la información acerca de la entidad. Incluir toda la información necesaria para que pueda encontrarse fácilmente en una consulta posterior.

5. Descartar el token de autenticación.

6. Usar la API de consulta para comprobar que los datos han sido bien introducidos, incluida la información de binding templates, para asegurarse de que

cualquiera que la obtenga pueda comunicarse e interactuar de forma adecuada con los servicios ofrecidos.

7. Actualizar la información cada vez que sea necesario para reflejar cambios en los datos de contacto y para incluir nueva información de servicios, obteniendo y descartando un token de autenticación del Operador en cada caso.

Los ejemplos siguientes muestran cómo la entidad "SkateBoots Company" podría registrar su información y cómo un distribuidor interesado en comerciar con la línea de productos de la compañía podría encontrar información para contactar con la misma y hacer un pedido, utilizando los servicios web de SkateBoots.com.

2.2.6.1. Actualizando el registro

Una vez obtenido un token de autenticación de uno de los Operadores, por ejemplo Microsoft, los responsables en SkateBoots.com deciden qué información publicar y usan una de las herramientas proporcionadas por Microsoft o bien construyen un programa en java, c# o cualquier otro lenguaje para generar los mensajes SOAP adecuados. El siguiente sería un ejemplo:

```
POST /save_business HTTP/1.1
Host: www.skateboots.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "save_business"

<?xml version="1.0" encoding="UTF-8" ?>

<Envelope xmlns="http://schemas/xmlsoap.org/soap/envelope/">
  <Body>
    <save_business generic="2.0" xmlns="urn:uddi-org:api_v2">
      <businessKey="">
      </businessKey>
      <name>
      Skateboots, Inc.
      </name>
      <description>
      You know, like the ones in Batman and Robin . . .
      </description>
      <identifierBag> ... </identifierBag>
      ...
    </save_business>
  </Body>
</Envelope>
```

Este ejemplo ilustra un mensaje SOAP solicitando el registro en UDDI de una businessEntity para la compañía SkateBoots.com. El elemento businessKey se deja en blanco de forma intencionada, ya que el Operador generará automáticamente la clave UUID para esta estructura de datos. La mayoría de los campos que puede contener esta estructura se omiten con el propósito de mostrar un ejemplo sencillo.

SkateBoots siempre puede ejecutar otra operación save_business para añadir información adicional a la básica que se acaba de insertar. En particular, sería

conveniente incluir identificadores e información de categorías, y enlazar sus servicios web. Supondremos que SkateBoots tienes dos servicios web: Preseason Orders (pedidos de pretemporada) y In-season Orders (pedidos de temporada).

2.2.6.2. Consultando la información

Una vez que SkateBoots.com ha actualizado su entrada en el registro UDDI con toda la información necesaria, las compañías que quieran convertirse en distribuidores de sus productos pueden buscar información de contacto y obtener descripciones de servicios y puntos de acceso para los dos servicios web que SkateBoots publica para realizar pedidos online: preseason e in-season.

```
POST /get_businessDetail HTTP/1.1
Host: www.skateboots.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "get_businessDetail"

<?xml version="1.0" encoding="UTF-8" ?>

<Envelope xmlns="http://schemas/xmlsoap.org/soap/envelope/">
  <Body>
    <get_businessDetail generic="2.0" xmlns="urn:uddi-org:api_v2">
      <businessKey="C90D731D-777D-4130-9DE3-5303371170C2">
      </businessKey>
    </get_businessDetail>
  </Body>
</Envelope>
```

Este ejemplo ilustra una sencilla consulta SOAP para obtener información detallada acerca de la entidad SkateBoots Company. Para ello se utiliza la businessKey, la clave UUID que identifica de manera unívoca dicha entidad, y un mensaje del tipo get_businessDetail. Previamente podríamos haber utilizado una consulta find_business indicando el nombre total o parcial, y usando si fuera necesario caracteres comodín, para obtener una lista de identificadores UUID, que utilizaríamos en esta segunda consulta para asegurarnos de que la entidad obtenida es la que buscábamos.

Por ejemplo, el siguiente podría ser un mensaje businessList generado por el Operador a raíz de una consulta find_business. Los atributos serviceKey de los elementos serviceInfo pueden utilizarse para obtener información de enlace (bindingTemplate) con el que invocar los servicios ofrecidos por SkateBoots.com.

```
<businessList generic="2.0" operator="uddi.sourceOperator"
  truncated="false" xmlns:="urn:uddi-org:api_v2">
  <businessInfos>
    <businessInfo businessKey="C90D731D-777D-4130-9DE3-
     5303371170C2">
      <name>Skateboots Inc.</name>
      <serviceInfos>
        <serviceInfo serviceKey="D90D731D-777D-4130-9DE3-...">
          <name>PreSeason Orders</name>
        </serviceInfo>
```

```
          <serviceInfo serviceKey="E90D731D-777D-4130-9DE3-...">
            <name>In-season Orders</name>
          </serviceInfo>
        </serviceInfos>
      </businessInfo>
      <businessInfo> ... [more Skateboot manufacturers]
    </businessInfos>
  </businessList>
```

Otra aproximación al problema sería la de buscar directamente la definición mediante el o los tModel del servicio, usando identificadores y códigos de categorías. Pero en este caso el distribuidor debería asegurarse de que sólo una compañía (SkateBoots) fabrica los productos que quiere vender, de modo que sólo existe un servicio web para realizar pedidos online a la fábrica.

La figura 2.14 ilustra de forma simplificada la información almacenada en UDDI para la compañía SkateBoots.com una vez que ha hecho el registro. Una visión más completa podría hacerse incluyendo identificadores e información taxonómica de categorías, estructuras tModel, etc.

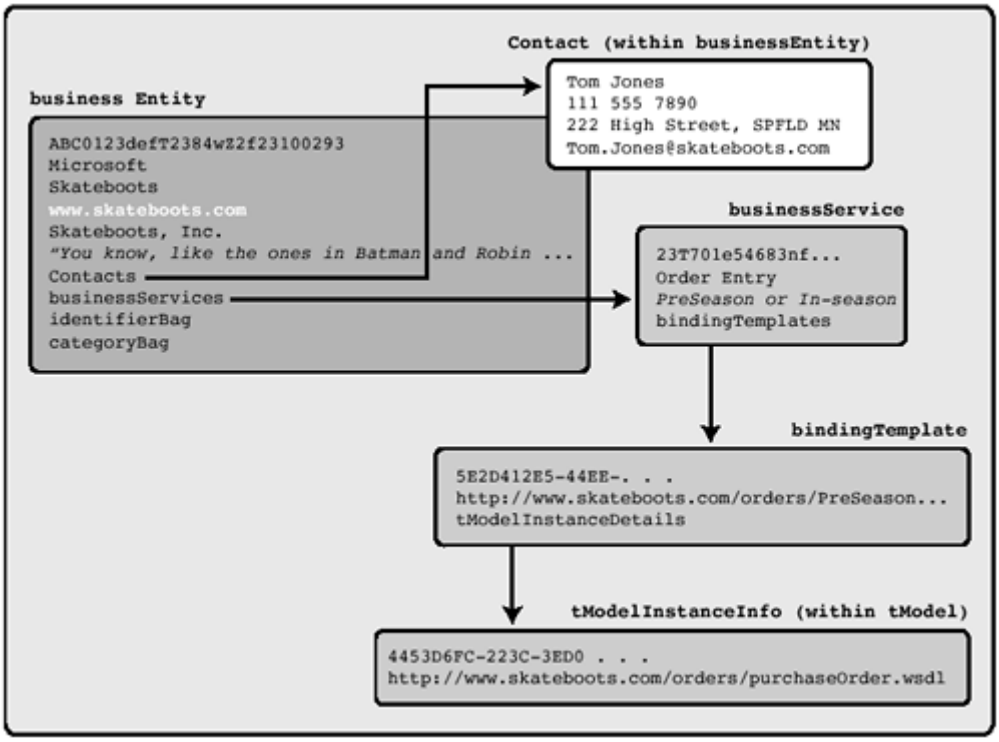

Figura 2.14 Información simplificada de SkateBoots, Inc.

La figura 2.14 muestra un camino sobre la estructura de datos de UDDI. Se ha comenzado obteniendo la estructura businessEntity de SkateBoots, Inc, a partir de la cuál se ha conseguido información de contacto (almacenada dentro de la estructura) y encontrado las descripciones de sus servicios mediante el bindingTemplate adecuado.

2.2.7. Usando WSDL en UDDI

El modelo de datos de UDDI define una estructura genérica para almacenar información acerca de entidades (business) y los servicios que proveen. Este modelo es completamente extensible, disponiendo de múltiples secuencias de estructuras de información. Aunque fue diseñado antes que WSDL, UDDI se anticipó a la necesidad de incluir algún documento del estilo de WSDL. De todas formas, UDDI está diseñado y se presume que pueda trabajar con cualquier otro lenguaje de descripción de servicios.

WSDL se representa en UDDI mediante una combinación de businessService, bindingTemplate y tModel. Como con cualquier servicio registrado en UDDI, la información genérica se guarda en la estructura businessService, y la información específica de cómo y dónde acceder al servicio se guarda en una o más estructuras bindingTemplate asociadas. Cada estructura bindingTemplate incluye un elemento que contiene la dirección de red del servicio y tiene asociado uno o más estructuras tModel que describen e identifican unívocamente el servicio.

Es posible almacenar la información de WSDL de varias maneras en UDDI, dada su flexibilidad y extensibilidad. En los manuales de buenas prácticas de WSDL se recomienda separar la información común y la información específica para un determinado servicio. Aunque WSDL no requiere ser usado con UDDI, el toolkit de IBM ha tenido en cuenta la utilidad de registrar WSDL en UDDI y para ello lo separa para que se adapte mejor a las estructuras de UDDI: interfaz de servicio e implementación de servicio. La figura 2.15 muestra cómo se divide un fichero WSDL en estas dos partes. Esta es la separación recomendada, de modo que la información común a un tipo de negocio o entidad, como el formato de los mensajes, los tipos de puerto y protocolos estén en la parte de interfaz de servicio, mientras que la información acerca de un punto de acceso particular al servicio estén en la parte de implementación de servicio.

WSDL está compuesto de un conjunto de documentos XML aislados, también llamados elementos, que pueden agruparse para formar un documento WSDL completo. La parte de interfaz de servicio se suelen registrar como tModels UDDI; el campo overviewDoc de la estructura tModel se utiliza para guardar un puntero al documento WSDL. De este modo, el cliente del servicio web puede trabajar con UDDI, mirando en el campo overviewDoc, para obtener la dirección donde encontrar el fichero WSDL.

Cuando UDDI se utiliza para guardar información WSDL, o para apuntar al documento, el tModel debe ser marcado con el tipo “wsdlSpec”, indicando que el campo overviewDoc apunta a la parte de interfaz de servicio de un fichero WSDL.

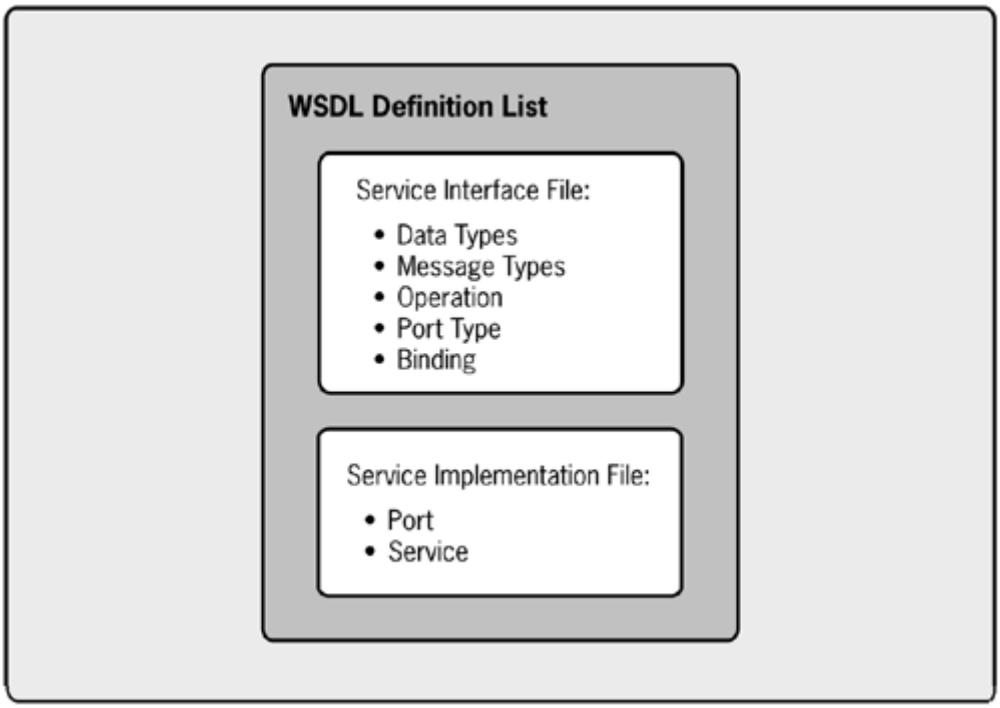

Figura 2.15 Separación recomendada de WSDL para UDDI

2.2.8. UDDI para uso privado

UDDI puede usarse en una intranet como un directorio de servicios web internos. Las aplicaciones pueden ser registradas en el UDDI interno, y se pueden hacer extensiones al modelo de datos para que se adapte a las necesidades del escenario de uso en el que nos encontremos. Cuando se usa la tecnología de servicios web para la integración de aplicaciones internas, el tenerlas almacenadas y catalogadas en el registro hace que sea muy fácil para otros departamentos descubrir aplicaciones listas para ser integradas de la misma manera que se hace en internet.

La figura 2.16 muestra el papel de un registro UDDI interno como almacén de metadatos acerca de servicios en proyectos de integración corporativos. Dada la facilidad de ampliación de la estructura de datos de UDDI y su capacidad para guardar virtualmente cualquier tipo de información identificativa, es sencillo para una entidad adaptar el registro a su uso interno.

Un registro UDDI local, o privado, se puede usar para guardar metadatos acerca de los puntos de acceso de una aplicación de la organización que exponga sus interfaces a otras aplicaciones con el objetivo de integrarse con ellas. Por ejemplo, un banco puede tener información de un cliente determinado en la aplicación de cuentas corrientes, en la de créditos y en la de gestión de valores y fondos de inversión. Un sistema de gestión global del cliente necesitaría consolidar los datos de estas tres aplicaciones. Si se han usado servicios web u otro tipo de arquitectura orientada a servicios para realizar la integración de estas aplicaciones, los metadatos correspondientes a las interfaces de estas aplicaciones podrían guardarse en un registro UDDI para uso privado del banco. El sistema de gestión global podría encontrar estas interfaces en el registro y utilizarlas para completar su tarea de manera automática.

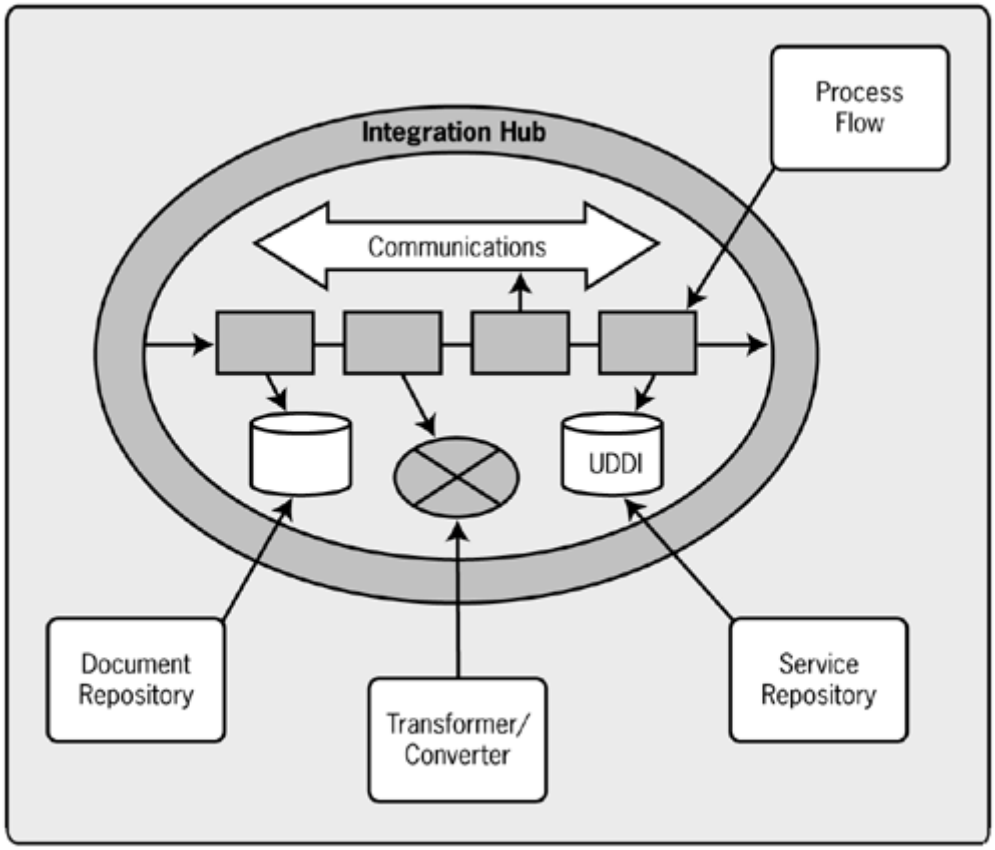

Figura 2.16 UDDI para uso privado

2.2.9. Requisitos de SOAP y Unicode

Veamos ahora cuáles son los requisitos para el uso de SOAP con UDDI, incluyendo el uso que UDDI hace del SOAP Action Header y prohibe el uso de SOAP headers y el opcional SOAP encoding. También veremos los requisitos de UDDI para el uso de Unicode.

2.2.9.1. SOAP

La versión 1 de UDDI requería una cadena vacía en la cabecera http SOAP Action. En cambio, la versión 2 admite la inclusión en este campo del nombre del método de la API referenciado en el cuerpo del mensaje SOAP. Por ejemplo:

```
SOAPAction: "find_business"
```

UDDI no tiene en cuenta las cabeceras SOAP. No obstante, se permite que las implementaciones de UDDI ignoren cualquier cabecera en el mensaje SOAP mientras no incluyan el atributo “mustUnderstand”, el cuál requiere un procesador SOAP para entender la cabecera en la que aparece dicho atributo. Si un Operador UDDI recibe un mensaje SOAP que incluye una cabecera requerida, el Operador rechazará el mensaje y devolverá un SOAP Fault “mustUnderstand”. Consecuentemente, el atributo “actor” de una cabecera SOAP tampoco está soportado.

UDDI no soporta el SOAP encoding opcional. Cualquier mensaje SOAP que contenga el atributo de espacio de nombrado de codificación (SOAP encoding namespace) fallará y se devolverá un SOAP Fault, un código de error UDDI del tipo E_Unsupported y un texto errInfo que describa el error.

Los documentos SOAP de petición y respuesta, enviados y recibidos de un Operador UDDI, usan el espacio de nombrado SOAP por defecto, sin prefijo XML, por ejemplo:

```
xmlns="http://schemas/xmlsoap.org/soap/envelope/"
```

2.2.9.2. Unicode

Los Operadores UDDI originales decidieron utilizar codificación UTF-8 en todas las peticiones, por lo tanto, se requiere que todos los Operadores soporten todos los caracteres definidos en el estándar Unicode para dar soporte multilenguaje.

Así pues, las entidades que necesiten registrar su información en más de un idioma pueden mandar los mensajes al registro conteniendo textos descriptivos en una gran variedad de lenguajes.

2.2.10. Conclusiones

UDDI proporciona un mecanismo flexible, potente y extensible para registrar y descubrir servicios en Internet. UDDI además es bastante útil en un entorno de intranet para uso privado. El registro UDDI público está gestionado por compañías privadas, y sus especificaciones están controladas por dichas compañías en el consorcio UDDI.org hasta que se traspase a una entidad independiente para su mantenimiento.

UDDI representa sus datos usando Esquemas XML y utiliza APIs SOAP para almacenar y devolver información. Los Operadores suelen gestionar tanto un registro público como otro de pruebas. La información almacenada en el repositorio UDDI no es consistente, y tal vez no lo sea nunca, en tanto en cuanto no se disponga de un estándar global de categorización. No obstante, UDDI proporciona un modelo firme y ampliamente aceptado y un conjunto de servicios para la gestión de metadatos de servicios web.

2.3. jUDDI

2.3.1. Introducción

jUDDI es una implementación en Java de código abierto de la especificación Universal Description, Discovery and Integration para servicios web.

Ya hemos comentado las bondades de un registro UDDI para que los proveedores registren sus servicios y puedan ser localizados y consumidos por los potenciales clientes. Pero ¿qué ocurre si queremos instalar un registro UDDI para pruebas locales?. Existen soluciones comerciales, y por tanto de pago, como Systinet UDDI, GLUE, o WebSphere UDDI Server. Todas estas aplicaciones son buenas soluciones, pero hay que pagar una licencia. Para evitar este pequeño inconveniente, hay aplicaciones de código abierto que nos pueden ayudar, como por ejemplo soapuddi, que aún no soporta tModels, o el más avanzado jUDDI, con la versión 2 de UDDI implementada.

Sin embargo hay que recalcar que jUDDI aún está sin terminar y en continuo desarrollo, por lo que hay que estar pendiente y bajarse y compilar las últimas fuentes disponibles en el repositorio CVS. Comenzó siendo desarrollado por un equipo de programadores en Sourceforge como una alternativa gratuita a las implementaciones comerciales de UDDI que ya existían en el mercado, y la importancia que ha ido adquiriendo al contar con cada vez más programadores y su buen hacer, ha hecho que sea trasladado al proyecto Apache, en su vertiente de servicios web. Actualmente, la web del proyecto es la siguiente:

http://ws.apache.org/juddi/

2.3.2. Características

Sus características principales son las siguientes:

- Código abierto
- Independiente de plataforma
- Funciona con JDK 1.3.1 y posteriores.
- Implementa la especificación versión 2 de UDDI
- Funciona con varias bases de datos relacionales que soporten el estándar ANSI SQL (MySQL, DB2, SyBase, JDataStore, etc.)
- Desplegable en cualquier servidor de aplicaciones Java que cumpla con la especificación Servlet 2.3 (Jakarta Tomcat, JOnAS, WebSphere, WebLogic, Borland Enterprise Server, jun, etc.)

2.3.3. Requisitos

jUDDI es una aplicación web totalmente escrita en Java y por lo tanto, puede ser desplegada en cualquier contenedor de servlets siempre que soporte la versión 2.1 o posterior de la especificación, como por ejemplo Jakarta Tomcat. Además, es necesario tener instalado Java 1.3 o posterior. Como en cualquier contenedor de aplicaciones, los detalles de despliegue variarán de un producto a otro.

Además, se necesita un almacén de datos externo en el que guardar los datos que gestiona el registro. Normalmente se utiliza un sistema gestor de base de datos relacional como MySQL, Oracle, DB2 u otros. En el paquete de distribución se incluyen instrucciones para hacerlo funcionar con varios gestores comerciales y de código abierto.

2.3.4. Configuración

jUDDI consiste en un núcleo central que funciona como procesador de peticiones. Estas peticiones, o consultas UDDI, son mensajes XML que son convertidos nada más llegar a objetos Java. Este proceso se denomina "unmarshalling" en la nomenclatura anglosajona y lo podemos traducir como deserialización. A continuación se invoca al método correspondiente que realizará todo el procesamiento y obtendremos otro objeto Java como resultado. Por último este objeto Java se convierte a un documento XML en el proceso de serialización y se devuelve la respuesta UDDI.

Para cumplir con su cometido, jUDDI utiliza los servicios de tres módulos: el almacén de datos, el autenticador y el generador de UUID. jUDDI está preconfigurado para utilizar las implementaciones por defecto de cada uno de estos módulos, pero puede modificarse su comportamiento modificando unos ficheros de configuración.

2.3.4.1. Persistencia

Como hemos comentado más arriba, jUDDI utiliza tres módulos que dan soporte al núcleo central de procesamiento. Uno de ellos, el más importante, es el almacén de datos. Para ello, jUDDI está preconfigurado para usar JDBC para acceder al sistema gestor de base de datos que elijamos.

Para configurar este módulo en primer lugar hay que crear una base de datos "jUDDI" en el sistema elegido, establecer los permisos de uso y crear las tablas adecuadas. Para ello se pueden usar los scripts suministrados, que en el momento de escribir esta memoria están disponibles para MySQL, DB2, HSQLdb, SyBase, PostgreSQL, Oracle, TotalXML y JDataStore.

Para terminar hay que configurar un DataSource JNDI con el nombre "jdbc/juddiDB" en el servidor de aplicaciones que vayamos a usar. El proceso varía de un producto a otro, por lo que hay que revisar la documentación que corresponda. En el caso de Jakarta Tomcat, pueden seguirse las instrucciones de la siguiente web:

http://jakarta.apache.org/tomcat/tomcat-5.0-doc/jndi-datasource-examples-howto.html

2.3.4.2. Autenticación

Este módulo permite que sólo los usuarios autorizados accedan a las partes críticas de la aplicación. La autenticación de un "publisher" de jUDDI es un proceso de dos pasos.

En el primer paso, el usuario envía su nombre de usuario y contraseña en un mensaje get_authToken, y se comprueba que es válido. El módulo por defecto acepta cualquier intento de autenticación que se haga. Es de esperar que se desarrollen nuevos módulos creados por los propios usuarios de jUDDI.

Durante el segundo paso se comprueba que el "publisher" ha sido definido en jUDDI. Se dice que un "publisher" está definido en jUDDI cuando existe una fila con su nombre en la tabla "PUBLISHER" del repositorio de datos. Para definir nuevos "publishers" hay que entrar al gestor y hacer la petición SQL correspondiente. Por ejemplo, con la siguiente sentencia definiríamos un "publisher" de nombre "John Doe":

```
INSERT INTO PUBLISHER (PUBLISHER_ID,PUBLISHER_NAME,ADMIN,ENABLED)
VALUES ('jdoe','John Doe','false','true');
```

2.3.4.3. Identificación

En este módulo se genera el identificador único que distingue cada uno de las estructuras de datos de UDDI, es decir, el UUID. La generación de dicho identificador normalmente requiere acceder al hardware de la máquina, en particular, a la dirección MAC de la tarjeta de red del ordenador que actúa de host. Lamentablemente, no es fácil acceder a dicha información usando Java. La especificación UDDI indica un método alternativo cuando no se puede acceder a esta información, y es el que utiliza jUDDI.

2.3.5. Problemas y carencias

En el momento de empezar a desarrollar nuestro proyecto, la aplicación jUDDI aún presentaba muchos problemas, ya que estaba en un estado muy inicial. En primer lugar, el proceso de instalación descrito en las instrucciones era insuficiente, dejando muchos aspectos de configuración en el aire, lo que a la postre provocaba que no hubiera conexión JDBC con el repositorio. Hay que decir que en la lista de distribución del proyecto jUDDI las preguntas acerca de este tema eran las más comunes, y lo que funcionaba para unos usuarios no funcionaba para otros. Se intentaron múltiples soluciones, instalando distintas librerías y probando distintas configuraciones JNDI en Tomcat, pero no funcionó ninguna. Por lo tanto, esta aplicación no se ha tenido operativa en ningún momento durante la realización del presente proyecto, aunque su estructura interna sí que ha servido como modelo a seguir. Además se ha analizado el código para ver dónde fallaba y se ha corregido para cumplir con las especificaciones. Por ejemplo, en jUDDI, al menos en el estado inicial de desarrollo, no se hacían comprobaciones de seguridad, como la existencia de claves por las que se hace una consulta, pudiendo saltar una excepción SQL cuando debería generarse un código de error UDDI específico (E_invalidKeyPassed). En los métodos de extracción de detalles de una estructura de datos determinada no se extraían todos los campos. Igualmente, se manejaban objetos que podían ser null y por lo tanto podría generarse una excepción. De

este modo, podemos decir que jUDDI ha sido una guía para entender la estructura que debía adoptar nuestro proyecto, pero que en el momento de iniciarse no era una aplicación fiable en absoluto.

3. Desarrollo del Proyecto

3.1. Servidor

En este apartado pasamos a describir el proceso de diseño de la aplicación principal del proyecto, que hemos denominado "Atenea", y que consiste en un núcleo en java donde se implementan todas las funciones de la API UDDI versión 2, tanto las de consulta como las de publicación.

Implementaremos un servicio web, en el sentido de que ofrecerá sus servicios a cualquier cliente para que pueda ser accedido a través de Internet, pero como se ha indicado en la introducción, no se creará la capa SOAP para tratar con mensajería XML. En lugar de ello, se ha creado una interfaz web para que los métodos de la API sean fácilmente accesibles para un usuario humano.

3.1.1. Preparación de las herramientas de trabajo

Para implementar nuestra aplicación necesitaremos varias herramientas: un kit de desarrollo Java, un entorno de desarrollo para programar, un contenedor de aplicaciones y un sistema gestor de base de datos.

3.1.1.1. Java SDK

Este es el Kit de Desarrollo de Software de Java. Es necesario para poder compilar el código que escribamos para la aplicación. El paquete puede obtenerse en la siguiente dirección:

http://java.sun.com/j2ee/1.4/download.html#sdk

Una vez bajado, basta con ejecutar el instalador y seguir las instrucciones en pantalla. Como vamos a usarlo desde el entorno de desarrollo no hace falta configurar variables de entorno ni nada más.

3.1.1.2. IDE NetBeans

Pasamos ahora a bajar e instalar el entorno de desarrollo, herramienta con la que escribiremos y compilaremos el código de la aplicación. Además, y dado que tiene su propio contenedor de aplicaciones, podremos probar la aplicación, así como depurarla para corregir los errores que se produzcan en tiempo de ejecución.

El producto puede obtenerse en la siguiente dirección:

http://www.netbeans.org/community/releases/41/index.html

Una vez bajado el instalador, lo ejecutamos y seguimos las instrucciones en pantalla. Tras elegir el directorio donde queramos instalarlo, hay que elegir el SDK Java

que vamos a usar con el entorno. Como puede verse en la figura 3.1., el instalador nos muestra los que hay ya instalados en el sistema y sólo tenemos que seleccionarlo.

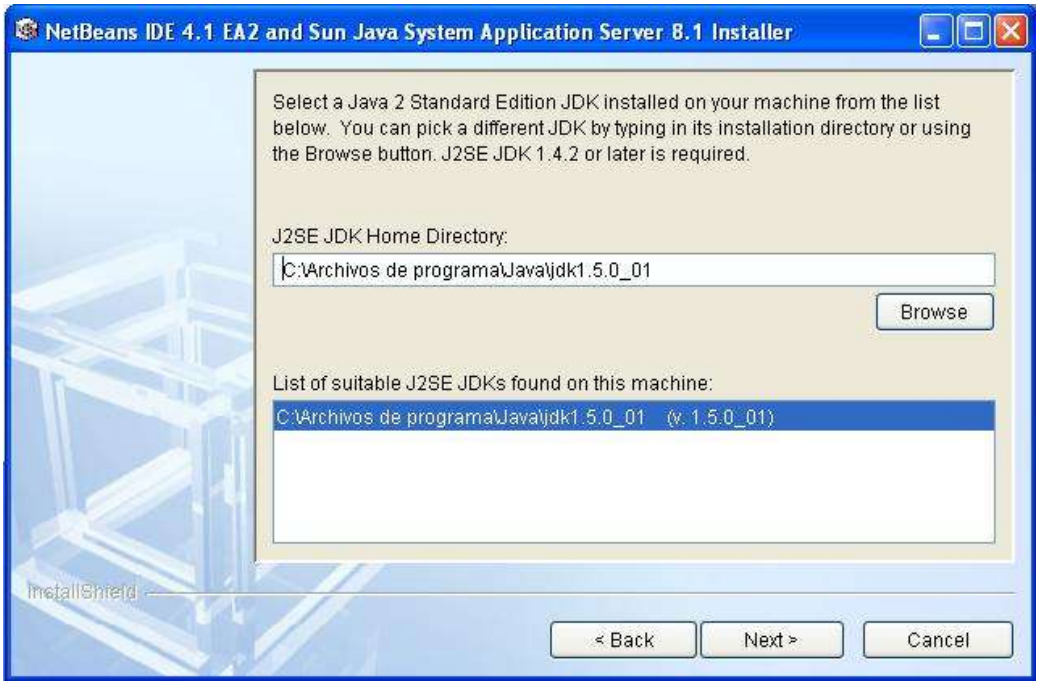

Figura 3.1. Selección de Java SDK en NetBeans

Ahora sólo basta aceptar para que se empiecen a copiar los archivos al sistema y termine la instalación. Una vez ejecutado, podemos pinchar en la pestaña Runtime para comprobar que tenemos varios servidores de aplicaciones ya instalados. En este proyecto utilizaremos el Tomcat integrado, que aparece remarcado en la figura 3.2.

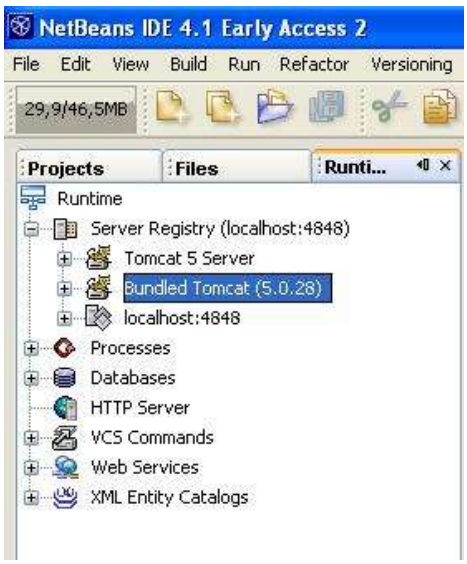

Figura 3.2. Tomcat integrado en NetBeans

También podemos ver la línea Databases, que configuraremos más tarde para poder acceder a los datos de nuestro repositorio sin tener que abandonar el IDE.

3.1.1.3. Jakarta Tomcat

Aunque hemos visto que el IDE NetBeans ya dispone de servidores de aplicaciones para pruebas en tiempo de codificación, es conveniente disponer de uno externo en el que desplegar la aplicación creada y comprobar que todo funciona correctamente. Este servidor es gratuito y está muy extendido, por lo que es ideal para nuestras necesidades. Podemos obtenerlo de la siguiente dirección:

http://jakarta.apache.org/site/downloads/downloads_tomcat-5.cgi

Una vez bajado el instalador, lo ejecutamos y seguimos las instrucciones. Como se ve en la figura 3.3., seleccionamos una instalación normal.

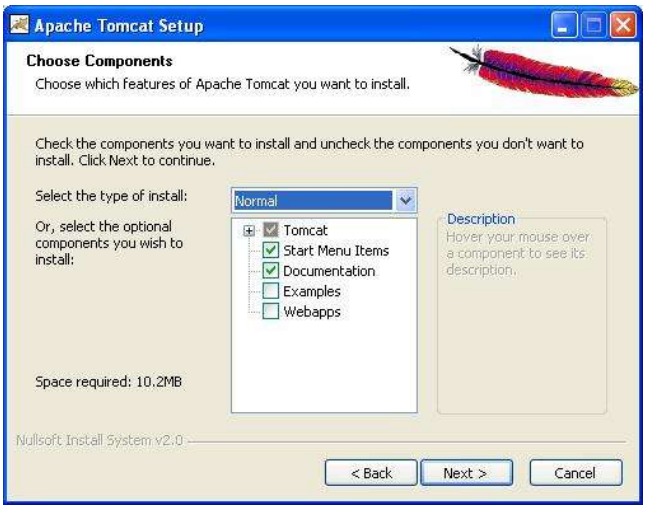

Figura 3.3. Instalación de Tomcat

Seleccionamos el directorio de destino y los ficheros comenzarán a copiarse al sistema. Una vez terminado el proceso, comprobamos que funciona correctamente. Para ello, ejecutamos el configurador de Tomcat y pulsamos "Start" para arrancarlo:

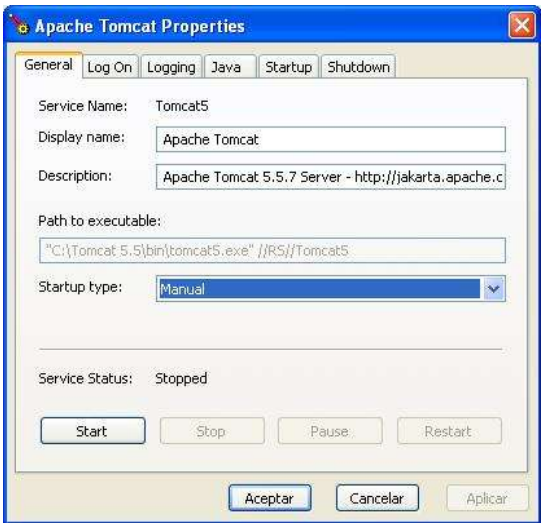

Figura 3.4. Arrancando Tomcat

Si todo se ha realizado correctamente, veremos la siguiente página de bienvenida en el navegador:

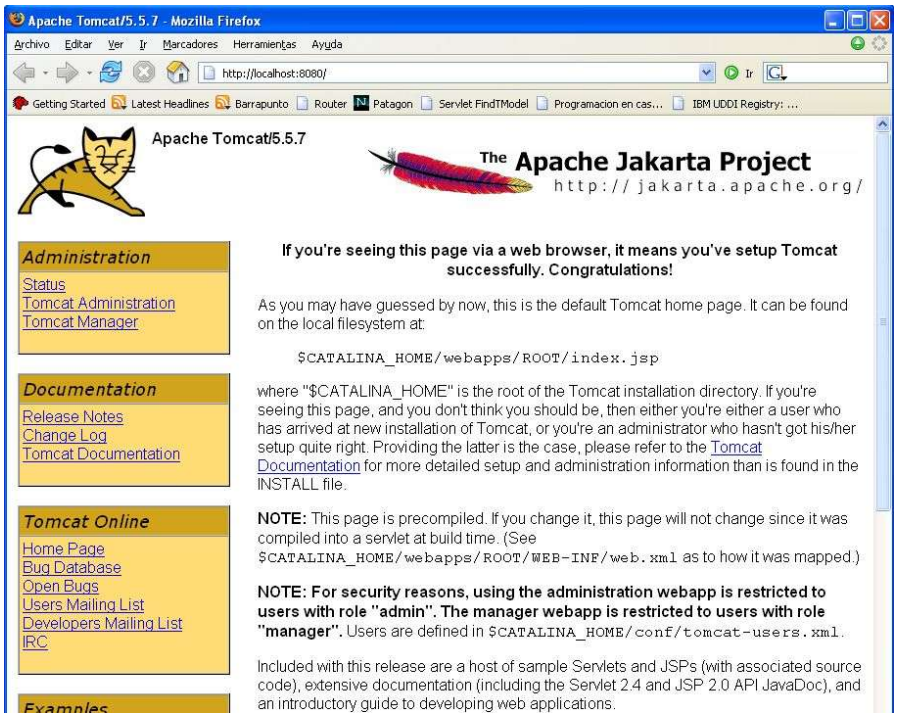

Figura 3.5. Página de bienvenida de Tomcat

A continuación, pararemos el servicio y crearemos un usuario para poder acceder a las áreas de gestión y administración de Tomcat y así configurarlo y desplegar aplicaciones.

Para ello editamos el fichero $CATALINA_HOME/conf/tomcat-users.xml, donde $CATALINA_HOME es el directorio de instalación, y añadimos un usuario con los papeles de administrador y gestor:

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="tomcat"/>
  <role rolename="manager"/>
  <role rolename="admin"/>
  <user username="tomcat" password="tomcat" roles="tomcat"/>
  <user username="admin" password="p0ba1r" roles="admin,manager"/>
</tomcat-users>
```

3.1.1.4. MySQL

Utilizaremos este sistema gestor de base de datos por ser de libre distribución, estar muy extendido y ser muy potente y estable, existiendo además versiones para distintos sistemas operativos. Este proyecto se ha creado en una plataforma Windows,

de modo que nos bajaremos la versión para este sistema que puede encontrarse en la siguiente ubicación:

http://dev.mysql.com/downloads/mysql/4.1.html

Una vez bajado, descomprimimos el archivo y ejecutamos el instalador:

Figura 3.6. Instalación de MySQL

Elegimos el directorio de instalación y se realizará todo el proceso de manera automática. Una vez instalado, se nos da opción a usar un asistente para configurar la aplicación. Con dicho asistente podemos configurar las cuentas de usuario y hacer que MySQL se ejecute como un servicio de Windows, de modo que se inicie en cada arranque del sistema operativo y esté disponible de manera inmediata.

Figura 3.7. Asistente de configuración de MySQL

Elegimos configuración detallada y a continuación se nos ofrecen tres alternativas: “máquina de desarrollo” (poco uso de memoria), “servidor”, y “servidor dedicado” (usa toda la memoria que esté disponible). En nuestro caso elegiremos

"máquina de desarrollo", pero en una instalación que se haga para albergar nuestra aplicación y quizás otras aplicaciones web más, sería conveniente utilizar la opción "servidor". Si el equipo donde se va a instalar se va a usar exclusivamente para albegar el servidor MySQL, sin ningún otro tipo de servicio, se elegiría "servidor dedicado".

Ahora se nos ofrecen tres tipos de uso: "base de datos multifuncional", "sólo base de datos transaccional", "sólo base de datos no transaccional". Elegimos la multifuncional.

Dejamos el directorio para albergar el fichero de datos de InnoDB tal cuál aparece y pulsamos continuar.

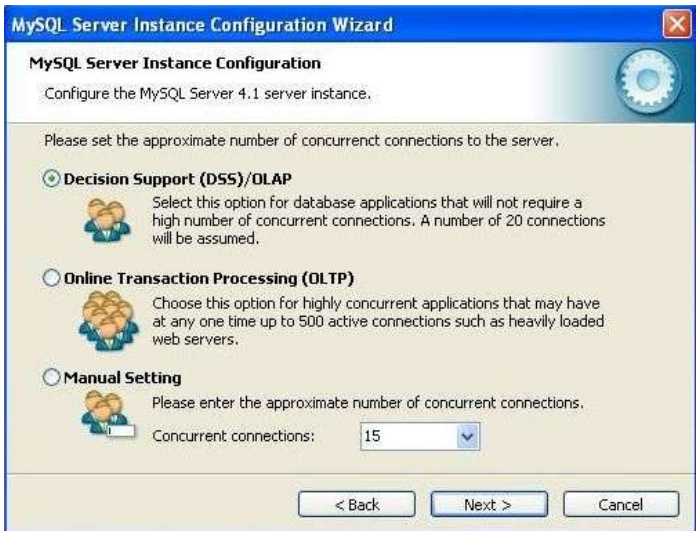

Figura 3.8. Configuración de la carga de conexiones

Pasamos ahora a configurar la carga de conexiones de nuestro servidor. En tiempo de desarrollo lo hemos dejado en "Decisión Support (DSS)/OLAP", lo que permite hasta 20 conexiones simultáneas, más que suficiente para un servidor de pruebas, pero en una instalación de producción debería cambiarse a "Online Transaction Procesing(OLTP)" para que pueda soportar un mayor número de conexiones simultáneas.

En el siguiente paso, habilitamos la comunicación TCP/IP para que pueda admitir conexiones externas, aunque no sería estrictamente necesario, ya que la aplicación que va a comunicarse con nuestra base de datos va a residir en el mismo equipo. Elegimos tambien el puerto, que dejamos en el que está por defecto: 3306

Elegimos soporte multilenguaje ya que con Atenea podremos guardar textos en múltiples idiomas. Seleccionamos la casilla que permite ejecutar MySQL como un servicio de Windows e incluimos el directorio bin de MySQL en el path del sistema.

A continuación configuramos la cuenta de root, eligiendo una contraseña adecuada e indicando que sólo queremos aceptar conexiones como root desde el equipo local. No crearemos ninguna cuenta anónima.

Figura 3.9. Configuración de la cuenta de administrador

Con esto terminamos de configurar el servidor MySQL. Si queremos detener el servicio, podemos ejecutar en una consola MSDOS:

```
NET STOP MySQL
```

Para volver a arrancarlo escribiríamos:

```
NET START MySQL
```

En el caso de que queramos desinstalar MySQL como servicio de Windows, el comando sería:

```
$MYSQL_DIR\bin\mysqld --remove
```

Y para volverlo a instalar, tal y como hicimos con el asistente:

```
$MYSQL_DIR\bin\mysqld --install
```

donde en ambos casos $MYSQL_DIR es el directorio de instalación de MySQL.

El siguiente paso consistirá en crear una cuenta de usuario exclusiva para nuestra aplicación, cuyo nombre será atenea y con clave atenea. Sólo podrá utilizarse desde el equipo local. Para ello entramos al servidor con privilegios de root escribiendo en una consola MSDOS:

```
mysql –u root –p<password>
```

Donde <password> deberá sustituirse por la contraseña que configuramos con el asistente. Una vez dentro ejecutamos los siguientes comandos SQL:

```
use mysql;
insert into user (user, host) values ('atenea', 'localhost');
set password for 'atenea'@'localhost' = PASSWORD('atenea'));
```

De este modo ya tenemos un usuario 'atenea', que sólo podrá conectarse desde el equipo local y con la clave encriptada. Más adelante le daremos los permisos necesarios para acceder a nuestra base de datos.

También es buena idea instalar un administrador gráfico para MySQL, que además de permitirnos modificar algunos parámetros de funcionamiento, sirve para monitorizar la carga de trabajo que soporta. La aplicación puede descargarse desde la siguiente ubicación:

http://www.mysql.com/products/administrator/

Figura 3.10. Instalador de MySQL Administrador

Una vez bajado el instalador, lo ejecutamos y seguimos unos sencillos pasos. Elegimos todos los paquetes y el directorio de destino y el proceso se completará automáticamente.

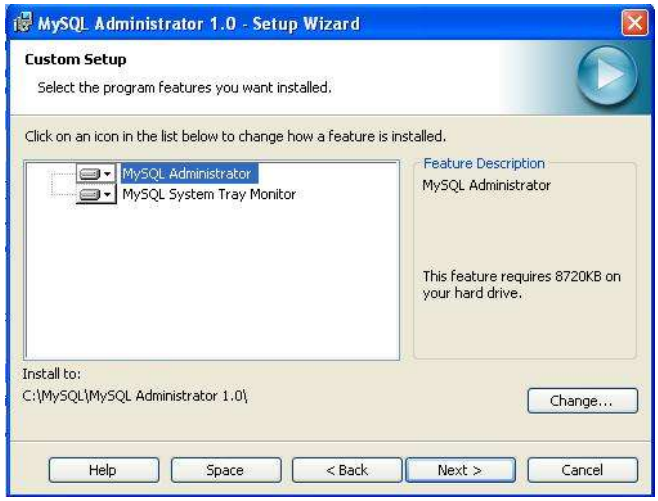

Figura 3.11. Selección de paquetes de MySQL Administrator

Con esto concluye la instalación de MySQL Administrator. Al ejecutarlo nos aparece una ventana de inserción de datos pidiéndonos nombre de usuario y clave, como

aparece en la figura 3.12. Introducimos la autenticación de root e indicamos que nos conectamos desde el equipo local.

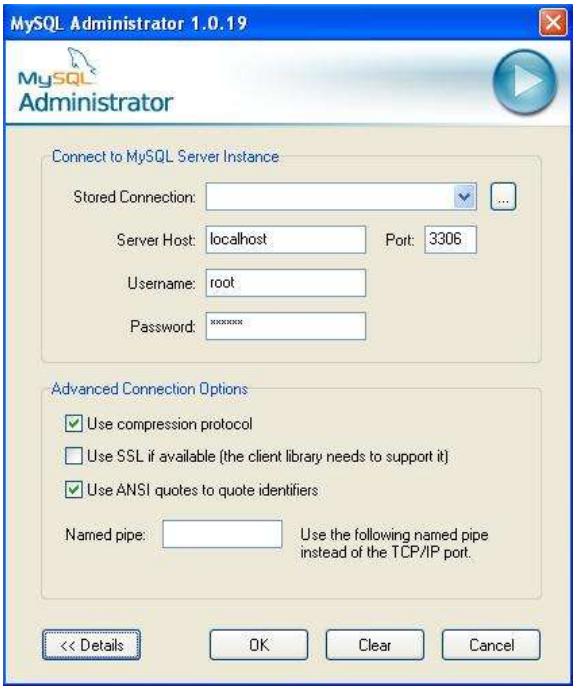

Figura 3.12. Acceso a MySQL Administrator

Con esto ya tendremos acceso total al servidor y podremos monitorizar su funcionamiento.

3.1.1.5. Librerías adicionales

Para terminar de preparar nuestro entorno de trabajo sólo quedan por bajar unas cuántas librerías. En primer lugar necesitamos un conector Java para MySQL, de modo que nuestra aplicación pueda comunicarse con el repositorio. Podemos bajarlo de la siguiente dirección web:

http://www.mysql.com/products/connector/j/

Descomprimimos el paquete en una carpeta y vemos cómo dentro hay un archivo llamado mysql-connector-java-<versión>-bin.jar. Dicho archivo es el que nos interesa, luego lo añadiremos al proyecto utilizando NetBeans.

Por último nos harán falta tres librerías más, que indicamos a continuación, junto con la dirección desde la que se pueden descargar:

- **Commons-pool**
  http://jakarta.apache.org/site/downloads/downloads_commons-pool.cgi

- **Commons-collections**
  http://jakarta.apache.org/site/downloads/downloads_commons-collections.cgi

- **Commons-dbcp**
  http://jakarta.apache.org/site/downloads/downloads_commons-dbcp.cgi

3.1.2. Creando el proyecto en NetBeans

Ya disponemos de las herramientas necesarias, es hora de crear un proyecto nuevo. Ejecutamos NetBeans y elegimos File→New Project. Aparece un asistente como el de la figura 3.13.

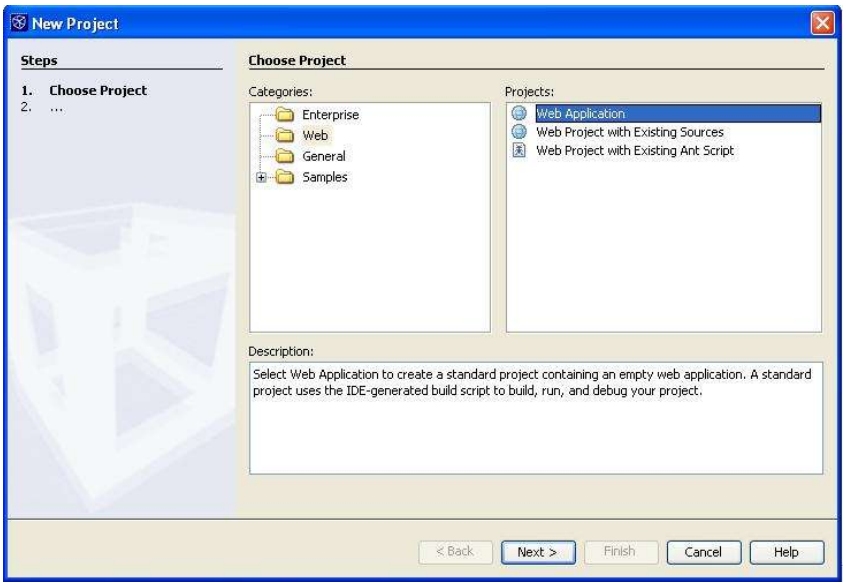

Figura 3.13. Creación de nuevo proyecto en NetBeans

Dado que vamos a crear un aplicación web, elegimos la categoría "Web" y dentro de ella, el tipo "Web Application". Pulsamos Next y aparece el formulario de la figura 3.14.

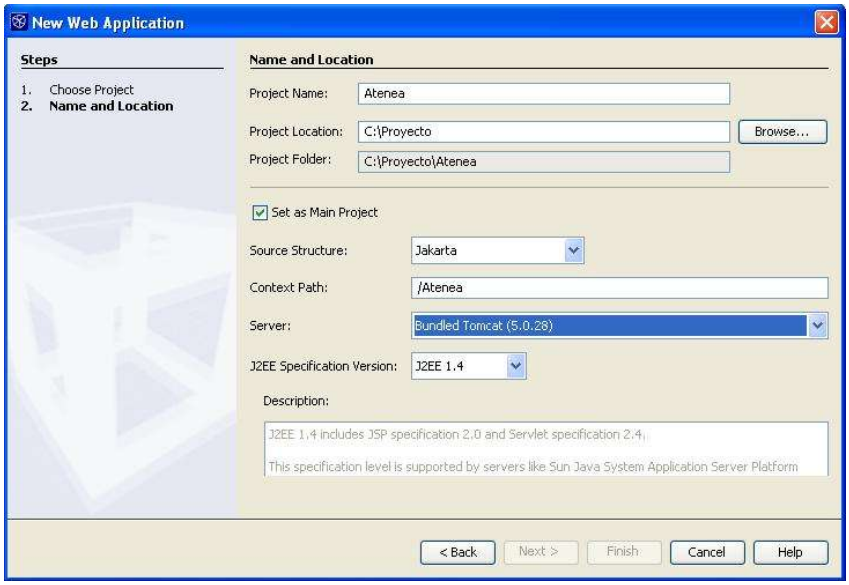

Figura 3.14 Características del nuevo proyecto

Elegimos de nombre clave "Atenea", e indicamos el directorio de trabajo donde se creará la estructura de directorios de nuestro proyecto. Dicha estructura será del tipo Jakarta y se utilizará como contenedor de aplicaciones de prueba el Tomcat integrado en Netbeans. Por último pulsamos Finish.

Una vez hecho esto, en la pestaña "Projects" estará listado nuestro proyecto, con sus componentes ordenados en forma de árbol.

El siguiente paso es editar las propiedades del proyecto. Para ello seleccionamos File → "Atenea" Properties. En el cuadro de diálogo que aparece, seleccionamos la entrada "Compiling Sources" y añadimos las 4 librerías que indicamos en el apartado 3.1.1.5.

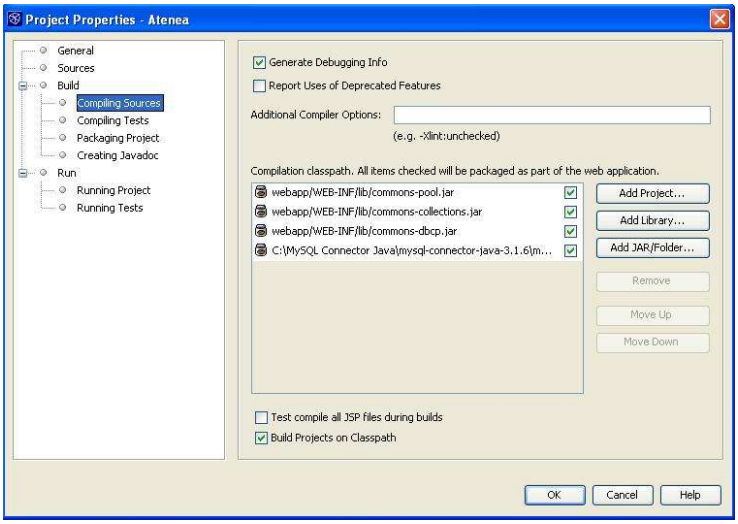

Figura 3.15. Inclusión de librerías en el proyecto

Una vez hecho esto, ya podemos empezar a escribir el código fuente de nuestra aplicación. Tan sólo hay que hacer clic con el botón derecho del ratón sobre "Source Packages" y seleccionar Java Class (si se va a escribir una clase Java), servlet, jsp o lo que queramos. En todos los casos, un pequeño asistente nos pedirá el nombre y unos pocos datos para crear un esqueleto del tipo de archivo que vayamos a crear.

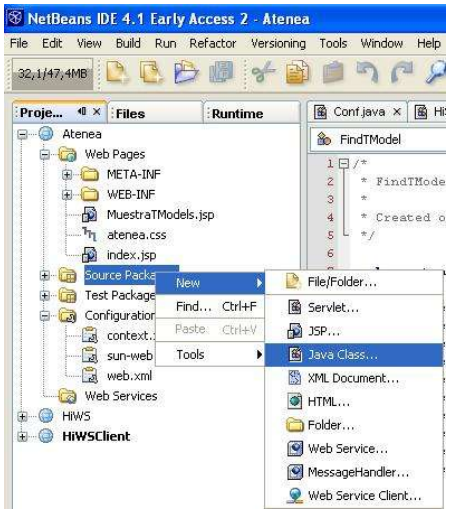

Figura 3.16. Selección del tipo de archivo a crear

3.1.3. Esquema de trabajo

Vamos a emplear una metodología de dentro hacia fuera. Esto es, crearemos primero el núcleo de la aplicación, para luego ir añadiendo capas y nuevas funcionalidades.

En primer lugar lo que vamos a hacer es modelar en Java los tipos de datos que vamos a emplear, teniendo en cuenta las relaciones existentes entre ellos tal y como se define en la norma.

Posteriormente crearemos una base de datos que permita la persistencia de estos datos. Para ello crearemos cada una de las tablas, junto con sus claves primarias que las identifican, y las claves foráneas que enlazan unas con otras.

Una vez hecho esto, crearemos la infraestructura de acceso a bases de datos en la aplicación "atenea". Utilizaremos reserva, o "pool", de conexiones para un acceso más eficiente al gestor de base de datos. Además haremos uso de transacciones para asegurarnos de que las operaciones se lleven a cabo de manera completa o que se vuelva al estado anterior a la transacción.

Con dicha infraestructura y la base de datos ya diseñada pasaremos a crear las clases que accederán a cada una de las tablas, para insertar, borrar, eliminar o actualizar cada uno de los campos de las mismas.

A continuación crearemos unos módulos indispensables para la API de publicación: el autenticador y el generador de UUID. El autenticador permitirá saber si los datos de autenticación suministrados por un usuario son válidos consultando un fichero de configuración, para así devolver un token de autenticación temporal. El generador de UUID es el módulo que crea identificadores UUID únicos mediante un mecanismo bien detallado en las especificaciones, y que se usa para identificar de manera unívoca los tipos principales de datos de UDDI.

Por último crearemos los métodos de la API propiamente dichos. Cada uno de estos métodos llamará a una función con el mismo nombre que se encargará de hacer todas las comprobaciones de seguridad, incluyendo la autenticación y la generación de UUIDs, y luego accederá a los métodos de la base de datos para consultar o hacer modificaciones en el repositorio.

3.1.4. Estructura de datos

Antes de empezar a programar las APIs, tenemos que saber cuáles son los tipos de datos que vamos a manejar. Para ello, acudimos a la web del proyecto UDDI y descargamos el documento "UDDI Versión 2.03 Data Structure Reference" de la siguiente ubicación:

http://uddi.org/pubs/DataStructure-V2.03-Published-20020719.pdf

En dicho documento se hace referencia a los 5 tipos de datos principales de UDDI, aunque más bien podríamos decir que son estructuras, ya que contienen múltiples componentes con multiplicidad variable. Dichos tipos son los ya descritos businessEntity, businessService, bindingTemplate, tModel y publisherAssertion.

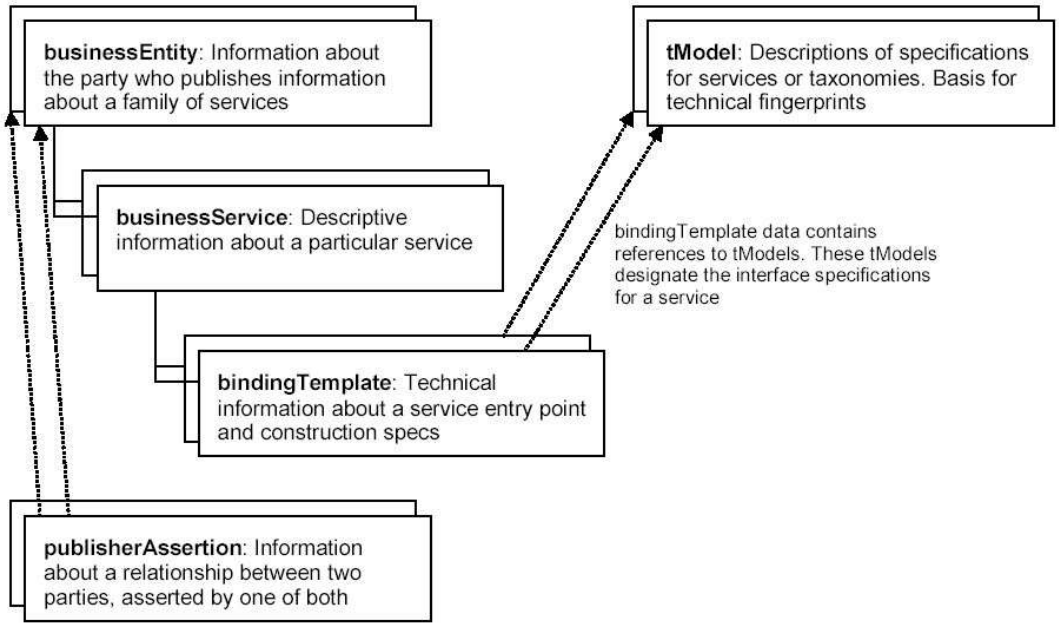

Figura 3.17. Los cinco tipos de datos de UDDI

Esta división por tipo de información proporciona una partición simple que permite una visión de conjunto fácilmente asimilable de los datos que maneja el registro. En estos cinco tipos de datos reside toda la información que puede suministrarse a un registro UDDI. Cada uno de ellos es una estructura XML que contiene múltiples elementos o atributos y cuyo significado se explica en el documento que acabamos de descargar. Además, se encuentran definidas en el "UDDI Versión 2.0 API Schema" que podemos encontrar en la dirección:

http://uddi.org/schema/uddi_v2.xsd

Este Esquema define aproximadamente 25 peticiones y 15 respuestas, cada una de las cuáles contiene alguna de estas estructuras, una versión reducida o una simple referencia a ellas. Es por esto que antes de ponernos a programar las APIs, necesitamos modelar las estructuras de datos como objetos Java.

En el Esquema XML se muestra que los datos gestionados por un registro UDDI están organizados en relaciones padre/hijo. Esta relación también puede advertirse en la figura 3.17 para los tipos principales.

La estructura businessEntity contiene una o más businessService. Cada businessService contiene una o más bindingTemplate, que a su vez contienen punteros, o referencias, a estructuras tModel. Es importante recalcar que ninguna estructura del núcleo principal está contenida en más de una estructura padre. Esto significa que sólo una businessEntity (identificada unívocamente por su UUID) contendrá, o será usada para expresar información, acerca de una estructura businessService determinada

(también identificada por su UUID). Por otro lado, las referencias funcionan de un modo diferente. En la misma figura 3.17 vemos que la estructura bindingTemplate contiene referencias a estructuras tModel. Al contrario de lo que ocurre con las relaciones padre/hijo, las referencias a una misma estructura pueden ser múltiples. Esto quiere decir que un mismo tModel puede ser referenciado por más de una estructura.

Por lo tanto, hay que distinguir entre lo que es una clave, o "key", y una referencia. Hay cinco tipos principales, cada instancia de los cuáles viene identificada por una clave única. El elemento "businessKey" contenido en la estructura businessEntity es una clave, no una referencia. De igual modo, los elementos "serviceKey" y "bindingKey", contenidos en la estructura businessService, son claves. Lo mismo ocurre para el elemento "tModelKey" de un tModel. La clave de una estructura publisherAssertion es la agrupación de las claves de las estructuras que relaciona.

Por otro lado, las referencias pueden aparecer en diferentes lugares, especialmente las referencias a tModel. Como hemos visto en la estructura bindingTemplate, cuando un tModel es referenciado ocurre dentro de otra estructura en forma de lista (tModelInstanceDetails) diseñada para albergar estas referencias a tModel. Dicha lista no es uno de los cinco tipos principales, por lo que no dispone de una clave que la identifique. Sin embargo, su propia identidad queda resuelta mediante la estructura que la contiene (bindingTemplate). Esto es, cualquier clave contenida en una estructura que no pertenezca a las cinco principales es una referencia. Ejemplos son los tModelKey encontrados en listas dentro de una bindingTemplate, o las que aparecen en esquemas de categorización o identificación, en cuyo contexto el tModel representa un espacio de nombrado.

En esta fase del proyecto vamos a codificar cada una de las estructuras de datos presentes en el documento de referencia. Para ello crearemos un paquete llamado "atenea.tipos" donde guardaremos las clases Java correspondientes.

Analizando los componentes de las estructuras descritas vemos que hay varios tipos de elementos. Los elementos de tipo String o UUID que sean de multiplicidad uno o cero los modelaremos como un objeto String. Los elementos del tipo estructura los modelaremos como un objeto nuevo. Por último, los elementos que puedan estar repetidos los modelaremos mediante un vector del objeto que corresponda.

Por ejemplo, el campo "description", que aparece en multitud de estructuras, es un elemento del tipo String pero con una particularidad, tiene un código de lenguaje asociado. Lo modelaremos como un objeto con dos atributos, "description" y "languageCode", ambos de tipo String, junto con los métodos constructores y accesores. A continuación se muestra el código escrito para la clase Description:

Fichero: Description.java

```
/*
 * ATENEA: Implementación en Java de la especificación UDDI v2.0
 * Autor: David Reales Ríos
 * Fecha: 2004
 */
```

```
package atenea.tipos;

import java.util.Vector;

/**
 * Description: Clase para almacenar algún tipo de información textual
 *              junto con un código de lenguaje opcional.
 */
public class Description implements RegistryObject
{

    String description;
    String languageCode;

    /**
     * Constructor básico de la clase
     */
    public void Description()
    {
    }

    /**
     * Constructor especificando la información textual
     * @param texto La descripción a añadir
     */
    public void Description(String texto)
    {
        setDescription(texto);
    }

    /**
     * Constructor especificando la información textual y el código
     * @param texto La descripción a añadir
     * @param codigo El código de lenguaje a usar
     */
    public void Description(String texto, String codigo)
    {
        setDescription(texto);
        setLanguageCode(codigo);
    }

    /**
     * Establece la información textual de este objeto Description
     * @param texto El texto a usar como description
     */
    public void setDescription(String texto)
    {
        this.description = texto;
    }


    /**
     * Devuelve la información textual de este objeto Description
     * @return El texto contenido en description
     */
    public String getDescription()
    {
        return this.description;
    }

    /**
     * Establece el código de lenguaje de este objeto Description
     * @param codigo El código a usar (debe ser un código ISO)
     */
    public void setLanguageCode(String codigo)
    {
        this.languageCode = codigo;
    }







    /**
     * Devuelve el código de lenguaje de este objeto Description
     * @return El código usado, o null si no hay ninguno asignado
     */
    public String getLanguageCode()
    {
```

```
        return this.languageCode;
    }

}
```

En el siguiente ejemplo vemos el tipo de dato “address”, utilizado para guardar la información de dirección de un contacto. Dicho tipo se compone de tres atributos y de un conjunto variable de elementos “addressLine”, que modelaremos mediante un Vector.

Fichero: Address.java

```
/*
 * ATENEA: Implementación en Java de la especificación UDDI v2.0
 * Autor: David Reales Ríos
 * Fecha: 2004
 */

package atenea.tipos;

import java.util.Vector;

/**
 * Address: La estructura address es una sencilla lista de elementos addressLine.
 *          Cada elemento addressLine es una cadena de texto. Los registros UDDI
 *          son responsables de mantener el orden de los addressLine que se van
 *          añadiendo. Además, las estructuras Address tienen tres atributos
 *          opcionales. El atributo useType describe el tipo de dirección en
 *          un formato de texto libre. Los valores sortCode no tienen relevancia
 *          dentro del registro UDDI, pero pueden ser usados por interfaces de
 *          usuario que presentan la información ordenada de alguna manera.
 */
 public class Address implements RegistryObject
 {
       String useType;
       String sortCode;
       String tModelKey;
       Vector addressLineVector;

       /**
         * Constructor básico de la clase, sin addressLines ni atributos
         */
       public Address()
       {
       }


      /**
       * Constructor de la clase, sin addressLine, pero con los atributos
       * useType y sortCode suministrados.
       * @param tipo El parámetro useType de la nueva Address
       * @param codigo El parámetro sortCode la de nueva Address
       */
       public Address( String tipo, String codigo)
       {
              this.useType = tipo;
              this.sortCode = codigo;
       }









       /**
        * Establece el parámetro useType de esta Address a la cadena de texto
        * suministrada.
        * @param tipo El nuevo parámetro useType de esta Address, o null
        *              si se desea eliminar el parámetro useType.
        */
       public void setUseType(String tipo)
       {
              this.useType = tipo;
       }
```

```
/**
 * Devuelve el parámetro useType de esta Address, o null si no tiene
 * ninguno definido.
 * @return El parámetro useType
 */
public String getUseType()
{
	return this.useType;
}

/**
 * Establece el parámetro sortCode de esta Address a la cadena de texto
 * suministrada.
 * @param codigo El nuevo parámetro sortCode, o null si se desea
 *                    que address deje de tener dicho parámetro.
 */
public void setSortCode(String codigo)
{
	this.sortCode = codigo;
}

/**
 * Devuelve el parámetro sortCode de esta address.
 * @return El parámetro sortCode de esta address, o null si no tiene.
 */
public String getSortCode()
{
	return this.sortCode;
}

/**
 * Establece la key del tModel de esta address a la key suministrada.
 * @param clave La nueva key del tModel de esta address
 */
public void setTModelKey(String clave)
{
	this.tModelKey = clave;
}

/**
 * Devuelve la key del tModel.
 * @return La tModelKey.
 */
public String getTModelKey()
{
	return this.tModelKey;
}


/**
 * Añade una nueva AddressLine a esta Address, al final del
 * conjunto de AddressLines ya existentes.
 * @param linea La nueva AddressLine a añadir.
 */

public void addAddressLine(AddressLine linea)
{
	if (this.addressLineVector == null)
		this.addressLineVector = new Vector();
	this.addressLineVector.add(linea);
}




/**
 * Establece el conjunto de AddressLines de esta Address.
 * @param lineas El conjunto de AddressLines a poner.
 */
public void setAddressLineVector(Vector lineas)
{
	if (this.addressLineVector == null)
		this.addressLineVector = new Vector();
	this.addressLineVector = lineas;
}

/**
 * Devuelve el conjunto de AddressLines de esta Address.
 * @return El vector de AddressLines de esta Address. Si no
 *          existen AddressLines se devuelve una enumeración
 *          vacía.
```

```
         */
        public Vector getAddressLineVector()
        {
                return this.addressLineVector;
        }
}
```

Tras analizar los tipos de datos, su estructura y la dependencia entre ellos, los hemos agrupado en paquetes lógicos. Además de los tipos de datos genéricos o que son usados por varias clases de estructuras, y que irán en el paquete "atenea.tipos", hemos creado ocho subpaquetes. La descripción y las clases incluidas en cada uno se detallan a continuación.

- **atenea.tipos**

  Aquí incluimos las clases genéricas o que no encajan bien en los restantes paquetes. Las clases incluidas son las siguientes:

  - **Address**: Guarda la dirección de un contacto. Es una estructura contenedora de objetos AddressLine.
  - **AddressLine**: Almacena líneas de texto, categorizadas con un par nombre-valor según el espacio de nombrado definido en el tModel de la estructura Address padre.
  - **BindingKey**: Guarda la clave de un bindingTemplate.
  - **BusinessKey**: Guarda la clave de un businessEntity.
  - **CategoryBag**: Agrupa varios objetos keyedReference y se usa para categorizar.
  - **Description**: Almacena alguna clase de información textual, junto con un código de idioma opcional.
  - **Direction**: Clase usada en SharedRelationships, indica la dirección de algún tipo de relación mediante dos claves: fromKey y toKey.
  - **DiscoveryURL**: Almacena una URL y un atributo useType asociado.
  - **DiscoveryURLs**: Elemento de businessEntity que almacena URLs. Es una estructura contenedora de elementos DiscoveryURL.
  - **EMail**: Guarda una dirección de correo electrónico, que puede estar acompañada opcionalmente por un parámetro useType. Con él se puede expresar en formato libre el tipo de email guardado, por ejemplo: "soporte técnico".
  - **IdentifierBag**: Agrupa varios objetos keyedReference y se usa para categorizar.

- **KeyedReference**: Estructura de datos que contiene una pareja nombre/valor más una referencia a tModel.

- **Name**: Estructura usada en BusinessService, BusinessEntity y en Tmodel. Almacena una cadena de texto junto con un atributo calificador de lenguaje.

- **OverviewDoc**: Estructura opcional de un elemento InstanceDetails usado para almacenar información acerca del uso de un determinado TModel dentro de cierto BindingTemplate.

- **OverviewURL**: Almacena la URL de un elemento OverviewDoc

- **Parametros**: No es una clase que derive que ningún modelo de datos de UDDI sino que es propia de "atenea". Guarda los parámetros de configuración del registro.

- **PersonName**: Elemento obligatorio de la estructura Contact, que guarda el nombre de un contacto.

- **Phone**: Elemento de la estructura Contact, que guarda un número de teléfono, acompañado de forma opcional por un atributo useType que indique el tipo de uso.

- **RegistryObject**: Interfaz que extiende la clase Serializable. Todas las clases basadas en tipos de datos de "atenea" implementan esta interfaz para que en una ampliación puedan ser convertidas a un documento XML.

- **ServiceKey**: Guarda la clave de un businessService.

- **SharedRelationships**: Parte del mensaje de respuesta relatedBusinessList. Incluye elementos keyedReference y un elemento Direction que expresa el sentido de la relación.

- **TModelBag**: Estructura que sirve de contenedor de elementos tModel.

- **TModelKey**: Guarda la clave de un tModel.

❖ atenea.tipos.assertion

En este paquete se incluye una única clase, la que guarda las relaciones entre estructuras businessEntity.

- **PublisherAssertion**: Almacena una relación entre estructuras businessEntity.

❖ atenea.tipos.binding

Se incluyen las clases relacionadas con la estructura bindingTemplate.

- **AccessPoint**: Almacena un puntero a un punto de entrada a un servicio, adornado por un parámetro urlType que indica el tipo (mail, http, ftp, etc.)

- **BindingTemplate**: Este objeto almacena la información técnica acerca de un servicio. Además de los elementos descritos en la especificación 2 de UDDI para la estructura bindingTemplate, se ha añadido un elemento categoryBag, que se incluirá en la versión 3, para obtener una mayor funcionalidad en la búsqueda.

- **BindingTemplates**: Este objeto modela la estructura del mismo nombre que actúa como contenedora de estructuras BindingTemplate.

- **HostingRedirector**: Elemento de estructura BindingTemplate que apunta a otra BindingTemplate donde se encuentra la información.

- **InstanceDetails**: Almacena información específica acerca de un servicio requerida bien para obtener los detalles de implementación del servicio relativos a una referencia específica tModelKey, bien para proporcionar parámetros adicionales y ayuda en la configuración del servicio.

- **InstanceParms**: Elemento opcional de instanceDetail. Se usa para contener parámetros de configuración o una referencia a una url donde encontrar un fichero que contenga dichos parámetros.

- **TModelInstanceDetails**: Es una estructura contenedora de elementos tModelInstanceInfo.

- **TModelInstanceInfo**: Almacena los detalles específicos de un BindingTemplate mediante referencia a un tModel.

❖ **atenea.tipos.business**

Se incluyen las clases relacionadas con la estructura businessEntity.

- **BusinessEntity**: La estructura businessEntity almacena toda la información conocida sobre un negocio o entidad, así como de los servicios que ofrece. Es la estructura de más alto nivel, de la que dependen todas las demás.

- **BusinessEntityExt**: Extensión de BusinessEntity. Puede incluir información adicional a la presente en BusinessEntity a discreción del Operador. En nuestro caso lo que se hará será incluir una copia exacta de la BusinessEntity normal.

- **Contact**: Clase para almacenar toda la información de un contacto humano. La clase businessEntity tiene un atributo llamado Contacts que es un conjunto de objetos de la clase Contact.

- **Contacts**: Elemento de BusinessEntity que actúa como contenedora de elementos de tipo Contact.

❖ **atenea.tipos.publisher**

Se incluyen las clases que tratan con el Publisher, es decir, con el usuario autorizado para modificar datos en el registro.

- **Publisher**: Guarda la información relativa a un Publisher. Es decir, la de un usuario autorizado a hacer cambios en la información del registro. Incluye su clave, nombre y apellidos, teléfonos de trabajo y móvil, email, así como información acerca de está autorizado o tiene privilegios de administrador. En este proyecto estos privilegios no se usan, pero pueden ser útiles en una futura ampliación.

- **PublisherID**: Guarda la clave que identifica a cierto Publisher.

❖ **atenea.tipos.request**

En este paquete incluimos la versión java de cada uno de los mensajes XML que se pueden enviar a un registro UDDI. De este modo empaquetamos en un único objeto todos los elementos que pueden tener estos mensajes.

Todos estos mensajes tienen en común dos atributos, la versión del mensaje, llamado en UDDI "generic" y que valdrá "2", y el espacio de nombrado al que pertenece, de valor "urn:uddi-org:api_v2".

Además se incluyen todos los elementos relacionados con estos mensajes y que requieren su codificación como objetos java independientes.

- **AddPublisherAssertions**: Se usa para añadir relaciones a un conjunto ya existente de relaciones entre entidades. Incluye un token de autenticación y un vector de PublisherAssertion.

- **AuthInfo**: Elemento común en múltiples mensajes de la API de publicación y que almacena un token de autenticación, obtenido gracias a una llamada a la API get_authToken.

- **DeleteBinding**: Mensaje para eliminar estructuras BindingTemplate del registro. Incluye un token y un vector de BindingKey.

- **DeleteBusiness**: Se utiliza para borrar estructuras BusinessEntity del registro. Incluye un token y un vector de BusinessKey.

- **DeletePublisher**: Se usa para borrar información acerca del Publisher en el Registro. No pertenece a la API UDDI, sino que es exclusiva de

"atenea". La función correspondiente no está implementada, pero se incluye para que en una futura ampliación de la aplicación, un usuario con privilegios de administrador pueda borrar Publisher del Registro. Incluye un token de autenticación y un vector de elementos PublisherID.

- **DeletePublisherAssertions**: Se utiliza para borrar estructuras PublisherAssertion del registro. Incluye un token y un vector de elementos PublisherAssertion.

- **DeleteService**: Se usa para borrar uno o varios BusinessService del registro. Incluye un token y un vector de BusinessKey.

- **DeleteTModel**: Este mensaje se utiliza para borrar un TModel del registro. En realidad el TModel se marca como borrado en lugar de ser físicamente eliminado. Se incluye un token y un vector de TModelKey.

- **DiscardAuthToken**: Se usa para informar a un Operador de que un token de autenticación previamente proporcionado ha dejado de ser válido. Incluye un token de autenticación.

- **FindBinding**: Este mensaje se utiliza para buscar estructuras BindingTemplate. Incluye una ServiceKey, un CategoryBag, un TModelBag y una lista de calificadores FindQualifiers. Además se añade un parámetro maxRows para limitar el número de resultados.

- **FindBusiness**: Se usa para encontrar información acerca de uno o más estructuras BusinessEntity. Incluye un vector de elementos Name, un IdentifierBag, un CategoryBag y un TModelBag, así como un elemento DiscoveryURLs y una lista de calificadores FindQualifiers. Además se añade un parámetro maxRows para limitar el número de resultados.

- **FindPublisher**: Se usa para encontrar información acerca de uno o más Publisher. No es un método de la API y no está implementada la funcionalidad, pero se ha incluido para facilitar una posible ampliación. Incluye un elemento Name y una lista FindQualifiers, así como el parámetro maxRows.

- **FindQualifier**: Elemento común en múltiples mensajes de la API de consulta y que sirve para modificar los parámetros de la búsqueda. Tan sólo contiene un objeto String con el valor de alguna constante definida por la norma.

- **FindQualifiers**: Elemento opcional de varios métodos de la API de consulta y que sirve para modificar su comportamiento. Es una estructura contenedora de elementos FindQualifier.

- **FindRelatedBusiness**: Este mensaje se utiliza para encontrar qué otras BusinessEntity están relacionadas con la que tiene la clave que se pasa como parámetro. Incluye una BusinessKey, una lista FindQualifiers y un parámetro maxRows.

- **FindService**: Mensaje utilizado para encontrar los servicios ofrecidos por alguna BusinessEntity. Incluye una BusinessKey, un vector de elementos Name, un CategoryBag, un TModelBag, una lista FindQualifiers y un parámetro maxRows.

- **FindTModel**: Mensaje utilizado para encontrar estructuras Tmodel en el registro. Incluye un elemento Name, un CategoryBag, un IdentifierBag, una lista FinQualifiers y un parámetro maxRows.

- **GetAssertionStatusReport**: Este mensaje proporciona información acerca de las aserciones que involucran alguna de la BusinessEntity gestionadas por un Publisher en particular. Incluye un token de autenticación y un parámetro del tipo CompletionStatus.

- **GetAuthToken**: Este mensaje se usa para obtener un token de autenticación. Incluye un par de parámetros de tipo String, con el login y la contraseña.

- **GetBindingDetail**: Mensaje utilizado para obtener información detallada acerca de uno o varios BindingTemplate. Incluye un vector de elementos BindingKey.

- **GetBusinessDetail**: Mensaje utilizado para obtener información detallada acerca de uno o varios BusinessEntity. Incluye un vector de elementos BusinessKey.

- **GetBusinessDetailExt**: Mensaje utilizado para obtener información extendida acerca de uno o varios BusinessEntity. Incluye un vector de elementos BusinessKey.

- **GetPublisherAssertions**: Se usa para obtener el conjunto completo de PublisherAssertion asociadas a un Publisher en particular. Incluye un token de autenticación

- **GetPublisherDetail**: No pertenece a la API UDDI, pero este mensaje puede utilizarse para obtener toda la información almacenada en el registro acerca de uno o varios Publisher. Incluye un vector de PublisherID.

- **GetRegisteredInfo**: Con este mensaje se solicita al registro una lista abreviada con las BusinessEntity y TModel controlados por la cuenta asociada al token de autenticación suministrado. Incluye un token de autenticación.

- **GetServiceDetail**: Mensaje utilizado para obtener información detallada acerca de uno o varios BusinessService. Incluye un vector de elementos ServiceKey.

- **GetTModelDetail**: Mensaje utilizado para obtener información detallada acerca de uno o varios TModel. Incluye un vector de elementos TModelKey.

- **SaveBinding**: Este mensaje se usa para guardar o actualizar información acerca de una o más estructuras BindingTemplate, junto con la relación que las enlaza con su estructura BusinessService padre. Incluye un token de autenticación y un vector de elementos BindingTemplate.

- **SaveBusiness**: Este mensaje se usa para guardar o actualizar información acerca de una o más estructuras BusinessEntity. Incluye un token de autenticación y un vector de elementos BusinessEntity.

- **SavePublisher**: No pertenece a la API UDDI, pero este mensaje puede utilizarse para guardar la información acerca de uno o varios Publisher. Incluye un token de autenticación y un vector de Publisher.

- **SaveService**: Este mensaje se usa para guardar o actualizar información acerca de una o más estructuras BindingService, junto con la relación que las enlaza con su estructura BusinessEntity padre. Incluye un token de autenticación y un vector de elementos BusinessService.

- **SaveTModel**: Este mensaje se usa para guardar o actualizar información acerca de una o más estructuras TModel. Incluye un token de autenticación y un vector de elementos TModel.

- **SetPublisherAssertions**: Mensaje utilizado para establecer el conjunto de relaciones asentidas por un Publisher dado. Incluye un token de autenticación y un vector de elementos PublisherAssertion.

- **ValidateValues**: Este mensaje sirve para mandar estructuras de un registro UDDI a un tercero para que las valide. Es dicha institución la que debe implementar el método de validación correspondiente. Incluye un vector de BusinessEntity, otro de BusinessService y otro de TModel.

❖ atenea.tipos.response

En este paquete incluimos la versión java de cada uno de los mensajes XML que un registro UDDI puede devolver a un usuario. De este modo empaquetamos en un único objeto todos los elementos que pueden tener estos mensajes.

Todos estos mensajes tienen en común dos atributos, la versión del mensaje, llamado en UDDI "generic" y que valdrá "2", y el espacio de nombrado al que pertenece, de valor "urn:uddi-org:api_v2".

Además se incluyen todos los elementos relacionados con estos mensajes y que requieren su codificación como objetos java independientes.

- **AssertionStatusItem**: Elemento de AssertionStatusReport. Guarda una relación entre entidades, sus propiedades y su estado de asentimiento. Incluye un elemento CompletionStatus, un KeysOwned, dos String con las claves de las entidades que relaciona y una KeyedReference.

- **AssertionStatusReport**: Mensaje de respuesta a la petición GetAssertionStatusReport. Devuelve todas las aserciones completas e incompletas relacionadas con una cuenta Publisher en particular.

- **AuthToken**: Mensaje de respuesta a GetAuthToken. Contiene la información de autenticación necesaria para poder hacer llamadas a la API de publicación. Incluye un elemento AuthInfo.

- **BindingDetail**: Este mensaje devuelve información detallada acerca de una o varias estructuras BindingTemplate es respuesta a un mensaje GetBindingDetail o FindBinding. Incluye un vector de elementos BindingTemplate.

- **BusinessDetail**: Este mensaje devuelve una o más estructuras completas BusinessEntity en respuesta a un mensaje GetBusinessDetail. Incluye un vector de elementos BusinessEntity.

- **BusinessDetailExt**: Este mensaje devuelve una o más estructuras completas BusinessEntityExt en respuesta a un mensaje del tipo GetBusinessDetailExt. Incluye un vector de BusinessEntityExt.

- **BusinessInfo**: Contiene una versión abreviada de los datos presentes en una estructura BusinessEntity y se emplea como elemento en un mensaje BusinessList. Incluye una clave BusinessKey, un vector de elementos Name, otro de elementos Description y un elemento ServiceInfos.

- **BusinessInfos**: Elemento del mensaje BusinessList contenedor de elementos BusinessInfo. Incluye un vector de BusinessIinfo.

- **BusinessList**: Este mensaje devuelve una o más estructuras BusinessInfo en respuesta a un mensaje FindBusiness. Incluye un elemento del tipo BusinessInfos.
- **CompletionStatus**: Parámetro utilizado en varios mensajes relacionados con las PublisherAssertion y que indica el estado de la misma. Incluye un valor String con el valor de la constante apropiada según la norma.

- **DispositionReport**: Esta estructura se utiliza para expresar el resultado genérico de una acción en el registro, así como para indicar mensajes de error. Incluye un vector de elementos Result.

- **ErrInfo**: Elemento de la estructura Result que sirve para contener un error. Incluye un par de objetos String, con el código de error y su mensaje asociado.

- **KeysOwned**: Elemento de AssertionStatusItem y que sirve para referenciar las estructuras relacionadas con una aserción determinada. Incluye un par de objetos String con la clave de dichas estructuras.

- **PublisherAssertions**: Este mensaje devuelve cero o más estructuras PublisherAsertion en respuesta a un mensaje SetPublisherAssertions o GetPublisherAssertions. Incluye un parámetro authorizedName y un vector de elementos PublisherAssertion.

- **PublisherDetail**: Este mensaje contiene información detallada acerca de una o más estructuras Publisher. No está en la API UDDI, pero lo incluimos para facilitar la ampliación de "atenea". Se devuelve en respuesta a un mensaje GetPublisherDetail o SavePublisher. Incluye un vector de elementos Publisher.

- **PublisherInfo**: Elemento de PublisherInfos, con la información más relevante acerca de una estructura Publisher. Incluye un par de objetos String con el identificador y el nombre.

- **PublisherInfos**: Elemento del mensaje PublisherList contenedor de elementos PublisherInfo. Incluye un vector de PublisherInfo.

- **PublisherList**: Este mensaje devuelve una o más estructuras de tipo PublisherInfo en respuesta a un mensaje FindPublisher. Incluye un elemento del tipo BusinessInfos.

- **RegisteredInfo**: Este mensaje devuelve información suficiente para identificar todas las businessEntity y datos de tModel publicados por el que hace la petición. Incluye un elemento de tipo BusinessInfos y otro TModelInfos.

- **RelatedBusinessInfo**: Elemento de RelatedBusinessInfos, con información de cada una de las BusinessEntity relacionadas con la referenciada por la BusinessKey del mensaje FindRelatedBusiness que provocó la consulta. Incluye la BusinessKey original, un vector de nombres, otro de descripciones y un elemento SharedRelationships.
- **RelatedBusinessInfos**: Elemento del mensaje RelatedBusinessList contenedor de elementos RelatedBusinessInfo. Incluye un vector de RelatedBusinessInfo.

- **RelatedBusinessList**: Este mensaje devuelve una o más estructuras tipo RelatedBusinessInfo en respuesta a un mensaje FindRelatedBusines. Incluye una BusinessKey y un elemento del tipo RelatedBusinessInfos.

- **Result**: Elemento de DispositionReport y que aparece en clases como las relacionadas con las excepciones. Almacena códigos de error y los mensajes asociados. Incluye un entero errno con el número de error y un vector de elementos ErrInfo.

- **ServiceDetail**: Este mensaje contiene información detallada acerca de una o más estructuras BusinessService. Se genera como respuesta a una consulta GetServiceDetail . Incluye un vector de BusinessService.

- **ServiceInfo**: Elemento de ServiceInfos con información acerca de una BusinessService. Incluye un par de String con las claves BusinessKey y ServiceKey y un vector de elementos Name.

- **ServiceInfos**: Elemento de ServiceList, contenedor de elementos ServiceInfo. Incluye un vector de elementos ServiceInfo.

- **ServiceList**: Este mensaje devuelve cero o más estructuras del tipo ServiceInfo en respuesta a un mensaje FindService. Incluye un elemento ServiceInfos.

- **TModelDetail**: Este mensaje devuelve cero o más estructuras completas TModel en respuesta a un mensaje del tipo GetTModelDetail. Incluye un vector de elementos TModel.

- **TModelInfo**: Elemento de TModelList con información acerca de un TModel. Incluye una clave TModelKey y un elemento Name.

- **TModelInfos**: Elemento de TModelList, contenedor de elementos TmodelInfo. Incluye un vector de elementos TModelInfo.

- **TModelList**: Este mensaje devuelve cero o más estructuras TModelInfo en respuesta a un mensaje FindTModel. Incluye un elemento TModelInfos.

❖ **atenea.tipos.service**

En este paquete incluimos las clases relacionadas con una de las estructuras principales del registro: BusinessService.

- **BusinessService**: Clase de modela la estructura BusinessService. Incluye un par de objetos String con las claves BusinessKey y ServiceKey, un vector de elementos Name, otro de elementos Description, un elemento BindingTemplates y un CategoryBag.

- **BusinessServices**: Elemento que agrupa varias estructuras del tipo BusinessService. Incluye un vector de elementos BusinessService.

❖ **atenea.tipos.tmodel**

En este paquete incluimos la clase que se encarga de modelar los TModel.

- **TModel**: Esta clase modela una de las estructuras principales de un registro UDDI: el TModel. Guarda las contantes con los identificadores

UUID de los TModel que constituyen el núcleo de UDDI. Incluye objetos String para modelar los elementos TModelKey, authorizedName, operator y name, un vector de elementos Description, un OverviewDoc, un CategoryBag y un IdentifierBag.

3.1.5. Diseño de la base de datos

Ahora lo que necesitamos es diseñar el repositorio de datos donde guardar toda la información del registro de modo que no se pierda al cerrar la aplicación. Como hemos comentado anteriormente utilizaremos el gestor de base de datos MySQL para este propósito.

Analizando los tipos de datos en UDDI hemos creado una estructura de tablas respetando las siguientes directrices:

- Por cada estructura del núcleo básico de UDDI creamos una tabla, con una clave primaria que coincidirá con su clave UUID.

- Cada una de estas tablas contendrá como elementos los componentes de los tipos de datos en los que están basados, siempre que sean de multiplicidad no superior a uno.

- Si algún componente puede tener multiplicidad superior a uno, se sacará como una nueva tabla. Se utilizará la clave primaria de la tabla de la que depende como clave foránea y un índice que permitirá diferenciar cada uno de estos elementos. Como clave primaria se utilizará la combinación de este índice junto con la clave foránea.

- Si algún componente de las estructuras es obligatorio se pondrá el atributo NOT NULL. En caso contrario se pondrá NULL.

Además de las tablas para salvaguardar la información de los tipos de datos de UDDI, hemos creado tres nuevas más:

- Tabla PUBLISHER

  En esta tabla se guardará información acerca de los Publisher, su identificador, su nombre completo, información de contacto como teléfonos y direcciones de correo electrónico, así como los permisos de los que dispone. Estos datos se modificarían con llamadas a "atenea" usando métodos administrativos que originalmente no están presentes en la especificación UDDI.

- Tabla AUTH_TOKEN

  En esta tabla se guardan todos y cada uno de los tokens de autenticación usados en el Registro, junto con el identificador y el nombre del

Publisher con el que están relacionados. Se incluye la fecha de creación y del último uso, datos que se utilizarán posteriormente para eliminar tokens caducados. A efectos estadísticos se incluye un campo para contabilizar el número de usos. Y por último se añade un campo llamado ESTADO con el que codificar si el token está disponible (1) para su uso o si debe considerarse caducado (0).

- Tabla CONFIGURACION

  Esta tabla se ha creado para guardar los parámetros configurables del Registro. Incluye el nombre del Operador, la URL donde se desplegará la aplicación "atenea", el path hacia el fichero que contendrá los datos de autenticación de usuarios, el máximo número de elementos Name por los que se puede hacer una búsqueda y el tiempo de caducidad de los tokens en milisegundos.

Para crear la base de datos hemos escrito unos archivos de instalación sql, que pueden cargarse identificándose en MySQL como administrador ejecutando la siguiente sentencia en una consola:

```
mysql –u root –p<password> < <nombre_fichero>
```

Donde <nombre_fichero> debe sustituirse por el de los que se muestran a continuación. El primero crea la base de datos y asigna todos los permisos para explotarla al usuario que antes creamos para la aplicación.

Fichero: crea_bd.sql

```
CREATE DATABASE atenea;

GRANT ALL ON atenea.* TO atenea@"localhost" IDENTIFIED BY "atenea";
```

El segundo fichero es el que crea las tablas, según las directrices que explicamos anteriormente.

Fichero: crea_tablas.sql

```
USE atenea;

CREATE TABLE BUSINESS_ENTITY
(
  BUSINESS_KEY        VARCHAR(41)    NOT NULL,
  AUTHORIZED_NAME     VARCHAR(255)   NOT NULL,
  PUBLISHER_ID        VARCHAR(20)    NULL,
  OPERATOR            VARCHAR(255)   NOT NULL,
  DATE                TIMESTAMP      NOT NULL,
  PRIMARY KEY (BUSINESS_KEY)
);

CREATE TABLE DISCOVERY_URL
(
```

```
  BUSINESS_KEY                VARCHAR(41)    NOT NULL,
  ID                  INT             NOT NULL,
  USE_TYPE            VARCHAR(255)    NOT NULL,
  URL_VALUE           VARCHAR(255)    NOT NULL,
  PRIMARY KEY (BUSINESS_KEY, ID),
  FOREIGN KEY (BUSINESS_KEY) REFERENCES BUSINESS_ENTITY(BUSINESS_KEY)
);

CREATE TABLE BUSINESS_ENTITY_NAME
(
  BUSINESS_KEY                VARCHAR(41)    NOT NULL,
  ID                  INT             NOT NULL,
  LANG_CODE           VARCHAR(2)      NULL,
  NAME_VALUE          VARCHAR(255)    NOT NULL,
  PRIMARY KEY (BUSINESS_KEY, ID),
  FOREIGN KEY (BUSINESS_KEY) REFERENCES BUSINESS_ENTITY(BUSINESS_KEY)
);

CREATE TABLE BUSINESS_ENTITY_DESCRIPTION
(
  BUSINESS_KEY        VARCHAR(41)    NOT NULL,
  ID                  INT             NOT NULL,
  LANG_CODE           VARCHAR(2)      NULL,
  DESCRIPTION         VARCHAR(255)    NOT NULL,
  PRIMARY KEY (BUSINESS_KEY, ID),
  FOREIGN KEY (BUSINESS_KEY) REFERENCES BUSINESS_ENTITY(BUSINESS_KEY)
);

CREATE TABLE CONTACT
(
  BUSINESS_KEY        VARCHAR(41)    NOT NULL,
  ID                  INT             NOT NULL,
  USE_TYPE            VARCHAR(255)    NULL,
  PERSON_NAME         VARCHAR(255)    NOT NULL,
  PRIMARY KEY (BUSINESS_KEY, ID),
  FOREIGN KEY (BUSINESS_KEY) REFERENCES BUSINESS_ENTITY(BUSINESS_KEY)
);

CREATE TABLE BUSINESS_SERVICE
(
  BUSINESS_KEY        VARCHAR(41)    NOT NULL,
  SERVICE_KEY         VARCHAR(41)    NOT NULL,
  DATE                TIMESTAMP      NOT NULL,
  PRIMARY KEY (SERVICE_KEY),
  FOREIGN KEY (BUSINESS_KEY) REFERENCES BUSINESS_ENTITY(BUSINESS_KEY)
);









CREATE TABLE BUSINESS_ENTITY_IDENTIFIER
(
  BUSINESS_KEY        VARCHAR(41)    NOT NULL,
  ID                  INT             NOT NULL,
  TMODEL_KEY          VARCHAR(41)    NULL,
  KEY_NAME            VARCHAR(255)    NULL,
  KEY_VALUE           VARCHAR(255)    NOT NULL,
  PRIMARY KEY (BUSINESS_KEY, ID),
  FOREIGN KEY (BUSINESS_KEY) REFERENCES BUSINESS_ENTITY(BUSINESS_KEY)
);

CREATE TABLE BUSINESS_ENTITY_CATEGORY
(
  BUSINESS_KEY        VARCHAR(41)    NOT NULL,
  ID                  INT             NOT NULL,
  TMODEL_KEY          VARCHAR(41)    NULL,
  KEY_NAME            VARCHAR(255)    NULL,
  KEY_VALUE           VARCHAR(255)    NOT NULL,
  PRIMARY KEY (BUSINESS_KEY, ID),
  FOREIGN KEY (BUSINESS_KEY) REFERENCES BUSINESS_ENTITY(BUSINESS_KEY)
);

CREATE TABLE CONTACT_DESCRIPTION
(
  BUSINESS_KEY        VARCHAR(41)    NOT NULL,
  CONTACT_ID          INT             NOT NULL,
```

```
  ID                  INT            NOT NULL,
  LANG_CODE           VARCHAR(2)     NULL,
  DESCRIPTION         VARCHAR(255)   NOT NULL,
  PRIMARY KEY (BUSINESS_KEY, CONTACT_ID, ID),
  FOREIGN KEY (BUSINESS_KEY, CONTACT_ID) REFERENCES CONTACT (BUSINESS_KEY, ID)
);

CREATE TABLE PHONE
(
  BUSINESS_KEY        VARCHAR(41)    NOT NULL,
  CONTACT_ID          INT            NOT NULL,
  ID                  INT            NOT NULL,
  USE_TYPE            VARCHAR(255)   NULL,
  PHONE_NUMBER        VARCHAR(50)    NOT NULL,
  PRIMARY KEY (BUSINESS_KEY, CONTACT_ID, ID),
  FOREIGN KEY (BUSINESS_KEY, CONTACT_ID) REFERENCES CONTACT (BUSINESS_KEY, ID)
);

CREATE TABLE EMAIL
(
  BUSINESS_KEY        VARCHAR(41)    NOT NULL,
  CONTACT_ID          INT            NOT NULL,
  ID                  INT            NOT NULL,
  USE_TYPE            VARCHAR(255)   NULL,
  EMAIL_ADDRESS              VARCHAR(255)   NOT NULL,
  PRIMARY KEY (BUSINESS_KEY, CONTACT_ID, ID),
  FOREIGN KEY (BUSINESS_KEY, CONTACT_ID) REFERENCES CONTACT (BUSINESS_KEY, ID)
);

CREATE TABLE ADDRESS
(
  BUSINESS_KEY        VARCHAR(41)    NOT NULL,
  CONTACT_ID          INT            NOT NULL,
  ID                  INT            NOT NULL,
  USE_TYPE            VARCHAR(255)   NULL,
  SORT_CODE           VARCHAR(10)    NULL,
  TMODEL_KEY          VARCHAR(41)    NULL,
  PRIMARY KEY (BUSINESS_KEY, CONTACT_ID, ID),
  FOREIGN KEY (BUSINESS_KEY, CONTACT_ID) REFERENCES CONTACT (BUSINESS_KEY, ID)
);









CREATE TABLE ADDRESS_LINE
(
  BUSINESS_KEY        VARCHAR(41)    NOT NULL,
  CONTACT_ID          INT            NOT NULL,
  ADDRESS_ID          INT            NOT NULL,
  ID                  INT            NOT NULL,
  LINE_VALUE          VARCHAR(80)    NOT NULL,
  KEY_NAME            VARCHAR(255)   NULL,
  KEY_VALUE           VARCHAR(255)   NULL,
  PRIMARY KEY (BUSINESS_KEY, CONTACT_ID, ADDRESS_ID, ID),
  FOREIGN KEY (BUSINESS_KEY, CONTACT_ID, ADDRESS_ID) REFERENCES
    ADDRESS (BUSINESS_KEY, CONTACT_ID, ID)
);

CREATE TABLE BUSINESS_SERVICE_NAME
(
  SERVICE_KEY         VARCHAR(41)    NOT NULL,
  ID                  INT            NOT NULL,
  LANG_CODE           VARCHAR(2)     NULL,
  NAME_VALUE          VARCHAR(255)   NOT NULL,
  PRIMARY KEY (SERVICE_KEY, ID),
  FOREIGN KEY (SERVICE_KEY) REFERENCES BUSINESS_SERVICE (SERVICE_KEY)
);

CREATE TABLE BUSINESS_SERVICE_DESCRIPTION
(
  SERVICE_KEY         VARCHAR(41)    NOT NULL,
  ID                  INT            NOT NULL,
  LANG_CODE           VARCHAR(2)     NULL,
  DESCRIPTION         VARCHAR(255)   NOT NULL,
  PRIMARY KEY (SERVICE_KEY, ID),
```

```
  FOREIGN KEY (SERVICE_KEY) REFERENCES BUSINESS_SERVICE (SERVICE_KEY)
);

CREATE TABLE BINDING_TEMPLATE
(
  SERVICE_KEY         VARCHAR(41)    NOT NULL,
  BINDING_KEY         VARCHAR(41)    NOT NULL,
  ACCESS_POINT_TYPE   VARCHAR(20)    NULL,
  ACCESS_POINT_VALUE  VARCHAR(255)   NULL,
  HOSTING_REDIRECTOR  VARCHAR(255)   NULL,
  DATE                TIMESTAMP      NOT NULL,
  PRIMARY KEY (BINDING_KEY),
  FOREIGN KEY (SERVICE_KEY) REFERENCES BUSINESS_SERVICE (SERVICE_KEY)
);

CREATE TABLE BUSINESS_SERVICE_CATEGORY
(
  SERVICE_KEY         VARCHAR(41)    NOT NULL,
  ID                  INT            NOT NULL,
  TMODEL_KEY          VARCHAR(41)    NULL,
  KEY_NAME            VARCHAR(255)   NULL,
  KEY_VALUE           VARCHAR(255)   NOT NULL,
  PRIMARY KEY (SERVICE_KEY, ID),
  FOREIGN KEY (SERVICE_KEY) REFERENCES BUSINESS_SERVICE (SERVICE_KEY)
);



CREATE TABLE BINDING_CATEGORY
(

  BINDING_KEY         VARCHAR(41)    NOT NULL,
  ID                  INT            NOT NULL,
  TMODEL_KEY_REF      VARCHAR(41)    NULL,
  KEY_NAME            VARCHAR(255)   NULL,
  KEY_VALUE           VARCHAR(255)   NOT NULL,
  PRIMARY KEY (BINDING_KEY,ID),
  FOREIGN KEY (BINDING_KEY)
 REFERENCES BINDING_TEMPLATE (BINDING_KEY)
);




CREATE TABLE BINDING_TEMPLATE_DESCRIPTION
(
  BINDING_KEY         VARCHAR(41)    NOT NULL,
  ID                  INT            NOT NULL,
  LANG_CODE           VARCHAR(2)     NULL,
  DESCRIPTION         VARCHAR(255)   NOT NULL,
  PRIMARY KEY (BINDING_KEY, ID),
  FOREIGN KEY (BINDING_KEY) REFERENCES BINDING_TEMPLATE (BINDING_KEY)
);

CREATE TABLE TMODEL_INSTANCE_INFO
(
  BINDING_KEY         VARCHAR(41)    NOT NULL,
  ID                  INT            NOT NULL,
  TMODEL_KEY          VARCHAR(41)    NOT NULL,
  INSTANCE_PARMS      VARCHAR(255)   NULL,
  OVERVIEW_URL        VARCHAR(255)   NULL,
  PRIMARY KEY (BINDING_KEY, ID),
  FOREIGN KEY (BINDING_KEY) REFERENCES BINDING_TEMPLATE (BINDING_KEY)
);

CREATE TABLE TMODEL_INSTANCE_INFO_DESCRIPTION
(
  BINDING_KEY         VARCHAR(41)    NOT NULL,
  TMODEL_INSTANCE_INFO_ID     INT    NOT NULL,
  ID                  INT            NOT NULL,
  LANG_CODE           VARCHAR(2)     NULL,
  DESCRIPTION         VARCHAR(255)   NOT NULL,
  PRIMARY KEY (BINDING_KEY, TMODEL_INSTANCE_INFO_ID, ID),
  FOREIGN KEY (BINDING_KEY, TMODEL_INSTANCE_INFO_ID) REFERENCES
    TMODEL_INSTANCE_INFO (BINDING_KEY, ID)
);

CREATE TABLE INSTANCE_DETAILS_DESCRIPTION
(
  BINDING_KEY         VARCHAR(41)    NOT NULL,
```

```
  TMODEL_INSTANCE_INFO_ID     INT     NOT NULL,
  ID                  INT             NOT NULL,
  LANG_CODE           VARCHAR(2)      NULL,
  DESCRIPTION         VARCHAR(255)    NOT NULL,
  PRIMARY KEY (BINDING_KEY, TMODEL_INSTANCE_INFO_ID, ID),
  FOREIGN KEY (BINDING_KEY, TMODEL_INSTANCE_INFO_ID) REFERENCES
    TMODEL_INSTANCE_INFO (BINDING_KEY, ID)
);

CREATE TABLE INSTANCE_DETAILS_OVERVIEWDOC_DESCRIPTION
(
  BINDING_KEY         VARCHAR(41)     NOT NULL,
  TMODEL_INSTANCE_INFO_ID     INT     NOT NULL,
  ID                  INT             NOT NULL,
  LANG_CODE           VARCHAR(2)      NULL,
  DESCRIPTION         VARCHAR(255)    NOT NULL,
  PRIMARY KEY (BINDING_KEY, TMODEL_INSTANCE_INFO_ID, ID),
  FOREIGN KEY (BINDING_KEY, TMODEL_INSTANCE_INFO_ID) REFERENCES
    TMODEL_INSTANCE_INFO (BINDING_KEY, ID)
);

CREATE TABLE TMODEL
(
  TMODEL_KEY          VARCHAR(41)     NOT NULL,
  AUTHORIZED_NAME     VARCHAR(255)    NOT NULL,
  PUBLISHER_ID        VARCHAR(20)     NOT NULL,
  OPERATOR            VARCHAR(255)    NOT NULL,
  NAME                VARCHAR(255)    NOT NULL,
  OVERVIEW_URL        VARCHAR(255)    NULL,
  DELETED             VARCHAR(5)      NULL,
  DATE                TIMESTAMP       NOT NULL,
  PRIMARY KEY (TMODEL_KEY)
);









CREATE TABLE TMODEL_DESCRIPTION
(
  TMODEL_KEY          VARCHAR(41)     NOT NULL,
  ID                  INT             NOT NULL,
  LANG_CODE           VARCHAR(2)      NULL,
  DESCRIPTION         VARCHAR(255)    NOT NULL,
  PRIMARY KEY (TMODEL_KEY, ID),
  FOREIGN KEY (TMODEL_KEY) REFERENCES TMODEL (TMODEL_KEY)
);

CREATE TABLE TMODEL_IDENTIFIER
(
  TMODEL_KEY          VARCHAR(41)     NOT NULL,
  ID                  INT             NOT NULL,
  TMODEL_KEY_REF      VARCHAR(255)    NULL,
  KEY_NAME            VARCHAR(255)    NULL,
  KEY_VALUE           VARCHAR(255)    NOT NULL,
  PRIMARY KEY (TMODEL_KEY, ID),
  FOREIGN KEY (TMODEL_KEY) REFERENCES TMODEL (TMODEL_KEY)
);

CREATE TABLE TMODEL_CATEGORY
(
  TMODEL_KEY          VARCHAR(41)     NOT NULL,
  ID                  INT             NOT NULL,
  TMODEL_KEY_REF      VARCHAR(255)    NULL,
  KEY_NAME            VARCHAR(255)    NULL,
  KEY_VALUE           VARCHAR(255)    NOT NULL,
  PRIMARY KEY (TMODEL_KEY, ID),
  FOREIGN KEY (TMODEL_KEY) REFERENCES TMODEL (TMODEL_KEY)
);

CREATE TABLE TMODEL_OVERVIEWDOC_DESCRIPTION
(
  TMODEL_KEY          VARCHAR(41)     NOT NULL,
  ID                  INT             NOT NULL,
  LANG_CODE           VARCHAR(2)      NULL,
  DESCRIPTION         VARCHAR(255)    NOT NULL,
  PRIMARY KEY (TMODEL_KEY, ID),
  FOREIGN KEY (TMODEL_KEY) REFERENCES TMODEL (TMODEL_KEY)
```

```
);

CREATE TABLE PUBLISHER_ASSERTION
(
  FROM_KEY            VARCHAR(41)    NOT NULL,
  TO_KEY              VARCHAR(41)    NOT NULL,
  TMODEL_KEY          VARCHAR(41)    NOT NULL,
  KEY_NAME            VARCHAR(255)   NOT NULL,
  KEY_VALUE           VARCHAR(255)   NOT NULL,
  ASSERTION           VARCHAR(4)     NOT NULL,
  FOREIGN KEY (FROM_KEY) REFERENCES BUSINESS_ENTITY (BUSINESS_KEY),
  FOREIGN KEY (TO_KEY) REFERENCES BUSINESS_ENTITY (BUSINESS_KEY)
);

CREATE TABLE PUBLISHER
(
  PUB_ID              VARCHAR(20)    NOT NULL,
  PUB_NAME            VARCHAR(255)   NOT NULL,
  LAST_NAME           VARCHAR(100)   NULL,
  FIRST_NAME          VARCHAR(100)   NULL,
  MIDDLE_INIT         VARCHAR(5)     NULL,
  WORK_PHONE          VARCHAR(50)    NULL,
  MOBILE_PHONE        VARCHAR(50)    NULL,
  EMAIL               VARCHAR(255)   NULL,
  ADMIN               VARCHAR(5)     NULL,
  AUTHORIZED          VARCHAR(5)     NULL,
  PRIMARY KEY (PUB_ID)
);




CREATE TABLE AUTH_TOKEN
(
  AUTH_TOKEN          VARCHAR(51)    NOT NULL,
  PUB_ID              VARCHAR(20)    NOT NULL,
  PUB_NAME            VARCHAR(255)   NOT NULL,
  CREADO              TIMESTAMP      NOT NULL,
  ULTIMO_USO          TIMESTAMP      NOT NULL,
  NUMERO_USOS         INT            NOT NULL,
  ESTADO              INT            NOT NULL,
  PRIMARY KEY (AUTH_TOKEN)
);

CREATE TABLE CONFIGURACION
(
  OPERATOR            VARCHAR(255)   NOT NULL,
  OPERATOR_URL        VARCHAR(255)   NOT NULL,
  USUARIOS_PATH       VARCHAR(255)   NOT NULL,
  MAX_NAME_ELEMENTS   INT            NOT NULL,
  TIME_OUT            INT            NOT NULL
);
```

El tercer fichero nos servirá para inicializar la base de datos con los TModels del núcleo de UDDI, utilizados en la categorización de servicios. Además, se incluyen los datos de un Publisher, el que utilizará posteriormente el cliente de la aplicación para guardar y actualizar información en el Registro.

Fichero: inicializa_tablas.sql

```
USE atenea;

INSERT INTO TMODEL (TMODEL_KEY, AUTHORIZED_NAME, OPERATOR, NAME, OVERVIEW_URL, DATE)
VALUES ('uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4','Administrator','Atenea','uddi-
org:types','http://www.uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm#UDDItypes',NULL);

INSERT INTO TMODEL_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4',0,'en','UDDI Type Taxonomy');

INSERT INTO TMODEL_OVERVIEWDOC_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4',0,'en','Taxonomy used to categorize
Service Descriptions.');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
```

```
VALUES ('uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4',0,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','categorization');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4',1,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','checked');



INSERT INTO TMODEL (TMODEL_KEY, AUTHORIZED_NAME, OPERATOR, NAME, OVERVIEW_URL, DATE)
VALUES ('uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2','Administrator','Atenea','ntis-
gov:naics:1997','http://www.uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm#NAICS',NULL);

INSERT INTO TMODEL_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2',0,'en','Business Taxonomy: NAICS
(1997 Release)');

INSERT INTO TMODEL_OVERVIEWDOC_DESCRIPTION (TMODEL_KEY, ID,LANG_CODE, DESCRIPTION)
VALUES ('uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2',0,'en','This tModel defines the
NAICS industry taxonomy.');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2',0,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','categorization');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2',1,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','checked');



INSERT INTO TMODEL (TMODEL_KEY, AUTHORIZED_NAME, OPERATOR, NAME, OVERVIEW_URL, DATE)
VALUES ('uuid:CD153257-086A-4237-B336-6BDCBDCC6634','Administrator','Atenea','unspsc-
org:unspsc','http://www.uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm#UNSPSC',NULL);

INSERT INTO TMODEL_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:CD153257-086A-4237-B336-6BDCBDCC6634',0,'en','Product Taxonomy: UNSPSC
(Version 7.3)');

INSERT INTO TMODEL_OVERVIEWDOC_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:CD153257-086A-4237-B336-6BDCBDCC6634',0,'en','This tModel defines Version
7.3 of the UNSPSC product taxonomy.');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:CD153257-086A-4237-B336-6BDCBDCC6634',0,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','categorization');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:CD153257-086A-4237-B336-6BDCBDCC6634',1,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','Checked');



INSERT INTO TMODEL (TMODEL_KEY, AUTHORIZED_NAME, OPERATOR, NAME, OVERVIEW_URL, DATE)
VALUES ('uuid:DB77450D-9FA8-45D4-A7BC-04411D14E384','Administrator','Atenea','unspsc-
org:unspsc:3-
1','http://www.uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm#UNSPSC31',NULL);

INSERT INTO TMODEL_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:DB77450D-9FA8-45D4-A7BC-04411D14E384',0,'en','Product Taxonomy: UNSPSC
(Version 3.1)');

INSERT INTO TMODEL_OVERVIEWDOC_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:DB77450D-9FA8-45D4-A7BC-04411D14E384',0,'en','This tModel defines the
UNSPSC product taxonomy.');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:DB77450D-9FA8-45D4-A7BC-04411D14E384',0,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','categorization');



INSERT INTO TMODEL (TMODEL_KEY, AUTHORIZED_NAME, OPERATOR, NAME, OVERVIEW_URL, DATE)
VALUES ('uuid:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88','Administrator','Atenea','uddi-
org:iso-ch:3166-
1999','http://www.uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm#ISO3166',NULL);

INSERT INTO TMODEL_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88',0,'en','ISO 3166-1:1997 and 3166-
2:1998. Codes for names of countries and their subdivisions. Part 1: Country codes. Part
2:Country subdivision codes.');
```

```
INSERT INTO TMODEL_OVERVIEWDOC_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88',0,'en','Taxonomy used to categorize
entries by geographic location.');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88',0,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','categorization');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88',1,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','checked');



INSERT INTO TMODEL (TMODEL_KEY, AUTHORIZED_NAME, OPERATOR, NAME, OVERVIEW_URL, DATE)
VALUES ('uuid:A035A07C-F362-44DD-8F95-E2B134BF43B4','Administrator','Atenea','uddi-
org:general_keywords','http://www.uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm#GenKW',N
ULL);

INSERT INTO TMODEL_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:A035A07C-F362-44DD-8F95-E2B134BF43B4',0,'en','Special taxonomy consisting
of namespace identifiers and the keywords associated with the namespaces');

INSERT INTO TMODEL_OVERVIEWDOC_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:A035A07C-F362-44DD-8F95-E2B134BF43B4',0,'en','This tModel defines an
unidentified taxonomy.');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:A035A07C-F362-44DD-8F95-E2B134BF43B4',0,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','categorization');



INSERT INTO TMODEL (TMODEL_KEY, AUTHORIZED_NAME, OPERATOR, NAME, OVERVIEW_URL, DATE)
VALUES ('uuid:4064C064-6D14-4F35-8953-9652106476A9','Administrator','Atenea','uddi-
org:owningBusiness','http://www.uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm#owningBusi
ness',NULL);

INSERT INTO TMODEL_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:4064C064-6D14-4F35-8953-9652106476A9',0,'en','A pointer to a
businessEntity that owns the tagged data.');

INSERT INTO TMODEL_OVERVIEWDOC_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:4064C064-6D14-4F35-8953-9652106476A9',0,'en','This tModel indicates the
businessEntity that published or owns the tagged tModel. Used with tModels to establish
an "owned" relationship with a registered businessEntity.');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:4064C064-6D14-4F35-8953-9652106476A9',0,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','categorization');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:4064C064-6D14-4F35-8953-9652106476A9',1,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','checked');



INSERT INTO TMODEL (TMODEL_KEY, AUTHORIZED_NAME, OPERATOR, NAME, OVERVIEW_URL, DATE)
VALUES ('uuid:807A2C6A-EE22-470D-ADC7-E0424A337C03','Administrator','Atenea','uddi-
org:relationships','http://www.uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm#Relationshi
ps',NULL);

INSERT INTO TMODEL_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:807A2C6A-EE22-470D-ADC7-E0424A337C03',0,'en','Starter set classifications
of businessEntity relationships');

INSERT INTO TMODEL_OVERVIEWDOC_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:807A2C6A-EE22-470D-ADC7-E0424A337C03',0,'en','This tModel is used to
describe business relationships. Used in the publisher assertion messages.');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:807A2C6A-EE22-470D-ADC7-E0424A337C03',0,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','relationship');



INSERT INTO TMODEL (TMODEL_KEY, AUTHORIZED_NAME, OPERATOR, NAME, OVERVIEW_URL, DATE)
VALUES ('uuid:327A56F0-3299-4461-BC23-5CD513E95C55','Administrator','Atenea','uddi-
org:operators','http://www.uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm#Operators',NULL
);

INSERT INTO TMODEL_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
```

```
VALUES ('uuid:327A56F0-3299-4461-BC23-5CD513E95C55',0,'en','Taxonomy for categorizing
the businessEntity of an operator of a registry.');

INSERT INTO TMODEL_OVERVIEWDOC_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:327A56F0-3299-4461-BC23-5CD513E95C55',0,'en','This checked value set is
used to identify UDDI operators.');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:327A56F0-3299-4461-BC23-5CD513E95C55',0,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','categorization');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:327A56F0-3299-4461-BC23-5CD513E95C55',1,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','checked');



INSERT INTO TMODEL (TMODEL_KEY, AUTHORIZED_NAME, OPERATOR, NAME, OVERVIEW_URL, DATE)
VALUES ('uuid:E59AE320-77A5-11D5-B898-0004AC49CC1E','Administrator','Atenea','uddi-
org:isReplacedBy','http://www.uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm#IsReplacedBy
',NULL);

INSERT INTO TMODEL_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:E59AE320-77A5-11D5-B898-0004AC49CC1E',0,'en','An identifier system used to
point (using UDDI keys) to the tModel (or businessEntity) that is the logical
replacement for the one in which isReplacedBy is used');

INSERT INTO TMODEL_OVERVIEWDOC_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:E59AE320-77A5-11D5-B898-0004AC49CC1E',0,'en','This is a checked value
set.');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:E59AE320-77A5-11D5-B898-0004AC49CC1E',0,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','identifier');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:E59AE320-77A5-11D5-B898-0004AC49CC1E',1,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','checked');



INSERT INTO TMODEL (TMODEL_KEY, AUTHORIZED_NAME, OPERATOR, NAME, OVERVIEW_URL, DATE)
VALUES ('uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823','Administrator','Atenea','dnb-com:D-
U-N-S','http://www.uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm#D-U-N-S',NULL);

INSERT INTO TMODEL_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823',0,'en','Dun&Bradstreet D-U-N-S®
Number');

INSERT INTO TMODEL_OVERVIEWDOC_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823',0,'en','This tModel is used for the
Dun&Bradstreet D-U-N-S® Number identifier.');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823',0,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','identifier');



INSERT INTO TMODEL (TMODEL_KEY, AUTHORIZED_NAME, OPERATOR, NAME, OVERVIEW_URL, DATE)
VALUES ('uuid:B1B1BAF5-2329-43E6-AE13-
BA8E97195039','Administrator','Atenea','thomasregister-
com:supplierID','http://www.uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm#Thomas',NULL);

INSERT INTO TMODEL_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:B1B1BAF5-2329-43E6-AE13-BA8E97195039',0,'en','Thomas Registry Suppliers');

INSERT INTO TMODEL_OVERVIEWDOC_DESCRIPTION (TMODEL_KEY, ID, LANG_CODE, DESCRIPTION)
VALUES ('uuid:B1B1BAF5-2329-43E6-AE13-BA8E97195039',0,'en','This tModel is used for the
Thomas Register supplier identifier codes.');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:B1B1BAF5-2329-43E6-AE13-BA8E97195039',0,'uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4','types','identifier');

INSERT INTO TMODEL_CATEGORY (TMODEL_KEY, ID, TMODEL_KEY_REF, KEY_NAME, KEY_VALUE)
VALUES ('uuid:B1B1BAF5-2329-43E6-AE13-BA8E97195039',1,'uuid:A035A07C-F362-44DD-8F95-
E2B134BF43B4','color de ojos','verdes');


INSERT INTO PUBLISHER (PUB_ID, PUB_NAME, LAST_NAME, FIRST_NAME, ADMIN, AUTHORIZED)
VALUES ('Atenea', 'Editorial Atenea', 'Reales Rios', 'David', 'TRUE', 'TRUE');
```

Por último hemos creado un fichero para introducir los parámetros de configuración del Registro que detallamos al describir la tabla CONFIGURACION.

Fichero: configura.sql

```
USE ATENEA;

INSERT INTO CONFIGURACION (OPERATOR, OPERATOR_URL, USUARIOS_PATH, MAX_NAME_ELEMENTS,
TIME_OUT)
VALUES ('Atenea','http://localhost:8084/Atenea','C:\\Proyecto\\usuarios.xml',5,3600000);
```

Como podemos ver, estos datos serían válidos en la instalación actual de desarrollo del proyecto. Cuando se vaya a instalar la aplicación, habrá que modificar estos valores. Para ello se creará una interfaz de usuario como se verá en la parte del cliente y en el manual del usuario.

3.1.6. Infraestructura de acceso a base de datos

En esta parte se crearán las clases que gestionarán el acceso a nuestra base de datos. Crearemos un nuevo paquete, "atenea.basedatos", y escribiremos las clases para crear una conexión al repositorio y para gestionar transacciones.

- **atenea.basedatos**

  - **GestorConexiones**: En esta clase se gestionan los aspectos de conexión a nuestra base de datos "atenea". Se utilizan constantes con el nombre del driver, la url de la base de datos, el nombre de usuario, la contraseña y el número máximo de conexiones activas e inactivas para proporcionar un pool de conexiones al repositorio. Se utiliza un pool para optimizar el acceso, ya que una nueva consulta puede utilizar una conexión del pool en lugar de esperar a que se cree una nueva. Una línea de mejora del proyecto podría ser la creación de una utilidad para el usuario final que modificase las constantes utilizadas.

  - **Transaccion**: Esta clase se encarga de la importante labor de gestionar las transacciones en "Atenea". Incluye los métodos "begin", "commit" y "rollback" necesarios para asegurar la integridad de la base de datos. En cada llamada al método "begin", una nueva conexión se introduce en la transacción, quedando los cambios introducidos por esta conexión en un estado provisional. Cuando se ha terminado con todo el trabajo que constituye la transacción se llama al método "commit" para que los

cambios se hagan definitivos. Si se produce algún tipo de error durante la transacción se debe llamar al método "rollback" para que la base de datos vuelva al estado inicial anterior a la llamada a "begin" por la primera de las conexiones de la transacción.

3.1.7. Métodos de base de datos

Una vez tenemos la estructura de la base de datos diseñada y disponemos de los métodos que nos van a permitir trabajar programáticamente con ella, es el momento de crear las clases que se encargarán de acceder a cada una de las tablas.

Cada una de ellas dispone de métodos para insertar, actualizar, extraer y borrar cada uno de los campos de la tabla que gestiona. Las pondremos en el paquete "atenea.basedatos". Por ejemplo, la siguiente es la clase que gestiona la tabla BusinessEntity:

Clase TablaBusinessEntity

```
/*
 * TablaBusinessEntity.java
 *
 * Created on 25 de febrero de 2005, 19:47
 */

package atenea.basedatos;

import atenea.tipos.business.BusinessEntity;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Timestamp;
import java.util.Vector;

/**
 * Clase que gestiona la Tabla BusinessEntity
 */
class TablaBusinessEntity {

    static String selectSQL = "SELECT AUTHORIZED_NAME, OPERATOR, DATE " +
            "FROM BUSINESS_ENTITY WHERE BUSINESS_KEY = ? ORDER BY DATE";

    static String comprobarPermisoSQL = "SELECT * FROM BUSINESS_ENTITY " +
            "WHERE BUSINESS_KEY = ? AND PUBLISHER_ID = ?";

    static String deleteSQL = "DELETE FROM BUSINESS_ENTITY WHERE BUSINESS_KEY = ?";

    static String insertSQL = "INSERT INTO BUSINESS_ENTITY " +
            "(BUSINESS_KEY, OPERATOR, PUBLISHER_ID, AUTHORIZED_NAME, DATE) "+
            "VALUES (?,?,?,?,?)";

    static String selectPubIDSQL = "SELECT PUBLISHER_ID FROM BUSINESS_ENTITY " +
            "WHERE BUSINESS_KEY = ?";

    static String selectByPubIDSQL = "SELECT BUSINESS_KEY " +
            "FROM BUSINESS_ENTITY WHERE PUBLISHER_ID = ?";

    /**
     *  Recoge una fila de la tabla business_entity.
     *  @param clave businessKey del businessEntity.
     *  @param conexion Conexión al repositorio de datos.
     *  @return BusinessEntity Objeto businessEntity con los campos
     *          authorized_name y operator
     *  @throws SQLException
     */
```

```
    public static BusinessEntity select(String clave, Connection conexion)
    throws SQLException {

        BusinessEntity business = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        try {
            pstmt = conexion.prepareStatement(selectSQL);
            pstmt.setString(1,clave);
            rs = pstmt.executeQuery();
            if (rs.next()) {
                business = new BusinessEntity();
                business.setBusinessKey(clave);
                business.setAuthorizedName(rs.getString(1)); // Authorized_Name
                business.setOperator(rs.getString(2)); // Operator
            }
            return business;
        } finally {
            try {
                rs.close();
                pstmt.close();
            } catch (Exception e) {System.out.println("Error al cerrar consulta en
TablaBusinessEntity\n");}
        }
    }


    /**
     *  Comprobamos que el publisher referenciado tiene permisos para modificar un
     *  businessEntity.
     *  @param clave businessKey del businessEntity.
     *  @param pubID Identificador del publisher
     *  @param conexion Conexión al repositorio de datos.
     *  @return boolean Devuelve true si tiene permisos.
     *  @throws SQLException
     */
    public static boolean comprobarPermiso(String clave, String pubID, Connection
conexion)
    throws SQLException {

        boolean autorizado = false;
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        if ((clave == null) || (pubID == null))
            return false;

        try {
            pstmt = conexion.prepareStatement(comprobarPermisoSQL);
            pstmt.setString(1,clave);
            pstmt.setString(2,pubID);
            rs = pstmt.executeQuery();
            if (rs.next())
                autorizado = true;
            return autorizado;
        } finally {
            try {
                rs.close();
                pstmt.close();
            } catch (Exception e) {System.out.println("Error al cerrar consulta en
TablaBusinessEntity\n");}
        }
    }

    /**
     *  Borra filas de la tabla BUSINESS_ENTITY relacionadas con una businessEntity.
     *  @param businessKey Clave de la businessEntity.
     *  @param conexion Conexión con el repositorio de datos.
     *  @throws SQLException
     */
    public static void delete(String businessKey, Connection conexion)
    throws SQLException {
        PreparedStatement pstmt = null;

        try {
            if ((businessKey != null) && (conexion != null)) {
                pstmt = conexion.prepareStatement(deleteSQL);
                pstmt.setString(1,businessKey);
                pstmt.executeUpdate();
            }
        } finally {
```

```
            try {pstmt.close();} catch (Exception e) {
                System.out.println("Error al cerrar consulta en
TablaBusinessEntity\n");}
        }
    }


    /**
     *  Inserta una fila en la tabla BUSINESS_ENTITY.
     *  @param business businessEntity.
     *  @param pubID Identificador del publisher
     *  @param conexion Conexión al repositorio de datos.
     *  @return void
     *  @throws SQLException
     */
    public static void insert(BusinessEntity business, String pubID, Connection
conexion)
    throws SQLException{

        PreparedStatement pstmt = null;
        Timestamp timeStamp = new Timestamp(System.currentTimeMillis());

        try {
            pstmt = conexion.prepareStatement(insertSQL);
            pstmt.setString(1, business.getBusinessKey()); // BUSINESS_KEY
            pstmt.setString(2, business.getOperator()); // OPERATOR
            pstmt.setString(3, pubID); // PUBLISHER_ID
            pstmt.setString(4, business.getAuthorizedName()); // AUTHORIZED_NAME
            pstmt.setTimestamp(5, timeStamp); // DATE

            pstmt.executeUpdate();
        } finally {
            try {
                pstmt.close();
            } catch (Exception e) {
                System.out.println("Error al cerrar actualización en
TablaBusinessEntity\n");
            }
        }
    }


    /**
     *  Recoge el publisherID con permisos sobre la businessEntity especificada.
     *  Si la businessKey es nula, no existe o corresponde a una businessEntity sin
     *  publisher asociado se devolverá null.
     *  @param clave businessKey del businessEntity.
     *  @param conexion Conexión al repositorio de datos.
     *  @return String pubID Identificador del publisher asociado
     *  @throws SQLException
     */
    public static String selectPubID(String clave, Connection conexion)
    throws SQLException {

        String pubID = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        if (clave == null)
            return null;

        try {
            pstmt = conexion.prepareStatement(selectPubIDSQL);
            pstmt.setString(1,clave);
            rs = pstmt.executeQuery();
            if (rs.next()) {
                pubID = rs.getString(1); // PUBLISHER_ID
            }
            return pubID;
        } finally {
            try {
                rs.close();
                pstmt.close();
            } catch (Exception e) {System.out.println("Error al cerrar consulta en
TablaBusinessEntity\n");}
        }
    }

    /**
     *  Recoge las businessKey de las businessEntity bajo responsabilidad
     *  del publisher identificado por el parámetro pubIB
     *  @param pubID Identificador del publisher.
```

```
     *  @param conexion Conexión al repositorio de datos.
     *  @return Vector Conjunto de businessKey
     *  @throws SQLException
     */
    public static Vector selectByPubID(String pubID, Connection conexion)
    throws SQLException {

        Vector keyVector = new Vector();
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        try {
            pstmt = conexion.prepareStatement(selectByPubIDSQL);
            pstmt.setString(1,pubID);
            rs = pstmt.executeQuery();
            while (rs.next()) {
                keyVector.add(rs.getString(1)); // BUSINESS_KEY
            }
            return keyVector;
        } finally {
            try {
                rs.close();
                pstmt.close();
            } catch (Exception e) {System.out.println("Error al cerrar consulta en
TablaBusinessEntity\n");}
        }
    }


}
```

3.1.8. Autenticador y Generador de UUID

En esta parte del desarrollo del proyecto vamos a crear los mecanismos para comprobar la identidad de los usuarios y para generar los identificadores únicos para las estructuras de UDDI. Las clases que veamos aquí se ubicarán en el paquete "atenea.seguridad".

En primer lugar veremos el Autenticador. Como sabemos, un registro UDDI se compone de dos partes bien diferenciadas, una de consulta y otra de publicación. Si bien la parte de consulta es pública, puede ser utilizada por cualquiera, no ocurre lo mismo con la de publicación. Para utilizar la API de publicación es necesario tener un token de autenticación válido, y para obtener dicho token es necesario disponer de un nombre de usuario previamente autorizado por el Operador, junto con su contraseña asociada.

En el caso de "Atenea", hemos optado por crear un fichero de usuarios en formato XML donde se guardarán los datos de autenticación: login, contraseña y Publisher asociado para cada uno de los usuarios autorizados. Dicho fichero no está encriptado y es un posible agujero de seguridad, pero se confía en el buen hacer del administrador del sistema donde se instale "Atenea" para ubicarlo en un lugar difícilmente accesible por los intrusos. Por esta misma razón, el path que apunta a dicho fichero es un parámetro configurable en nuestra aplicación.

La estructura de dicho fichero es muy sencilla. Partimos de un elemento raiz llamado "usuarios" del que cuelgan elementos del tipo "usuario". Cada uno de ellos dispone de tres atributos obligatorios, "login", "password" y "publisher".

Un ejemplo de fichero de usuarios podría ser el siguiente:

Fichero: usuarios.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
    Document   : usuarios.xml
    Created on : 14 de abril de 2005, 23:33
    Author     : David
    Description:
        Lista de usuarios autorizados para publicar en Atenea.
-->

<usuarios>
    <usuario login="david" password="p1ba2r" publisher="Atenea"/>
    <usuario login="pepe" password="pepe" publisher="pepe"/>
    <usuario login="antonio" password="antonio" publisher="Atenea"/>
</usuarios>
```

La clase Autenticador accede a dicho fichero XML, lo parsea y mete los datos de autenticación en memoria en una tabla Hash. Cuando se instancia una clase Autenticador se crea dicha tabla Hash, que queda disponible para consultas posteriores. Dispone de un método getPublisher, que recibe como parámetros de entrada el login y contraseña del usuario y que devuelve el Publisher asociado, el cuál posteriormente se utilizará para generar el token de autenticación usando el método de la API getAuthToken. Si alguno de los parámetros de entrada introducidos no es válido se lanzará la excepción correspondiente "UnknownUserException" con una explicación clara de la causa del error.

La otra parte de este apartado es el Generador de identificadores UUID que programaremos en la clase GeneradorUUID del paquete "atenea.seguridad". Como sabemos las estructuras principales de UDDI disponen de un campo en el que se introduce un valor que las identifica de manera unívoca y global.

Para obtener dicho valor se debe utilizar el algoritmo descrito por el Open Group en la siguiente dirección web:

http://www.opengroup.org/onlinepubs/009629399/apdxa.htm

Dicho algoritmo es lo suficientemente preciso como para asegurarnos de que no habrá dos identificadores UUID iguales en el mundo. Para ello utiliza la hora actual y la dirección MAC de la tarjeta de red del ordenador en el que se esté utilizando el algoritmo. Con estos datos es imposible obtener dos UUID iguales. Sin embargo, en Java no es fácil acceder a la dirección MAC de la tarjeta, por lo que en estos casos se utiliza un método alternativo, también descrito en la norma, para asegurarnos la unicidad. Este método es muy complejo, de modo que para evitar cualquier tipo de error hemos usado uno ya hecho por Maarten Coene.

Hay que decir que aunque el UUID generado es único, al ser de un tamaño finito, puede volver a repetirse pasado un tiempo. Según la norma este suceso se producirá en torno al año 3400 de nuestra era, lo que nos da un margen de uso bastante holgado.

3.1.9. Métodos de la API

El último paso en la creación del servidor, o núcleo, de la aplicación "Atenea" consiste en la programación de los métodos de la API propiamente dichos.

Para ello hemos programado la clase Registro, en el paquete raíz "atenea", donde se definirán las constantes y los atributos configurables del Registro. Inicialmente, los atributos configurables, como la url de despliegue de la aplicación, tienen un valor por defecto, pero al instanciar la clase se llama a un método de configuración que cargará los valores establecidos por el usuario, como se verá en 3.1.10. En el mismo constructor de la clase también se llama al método que elimina tokens caducados, como se verá en 3.1.11.

Esta clase implementa los métodos de la API UDDI. Cada uno de ellos recibe los elementos del mensaje XML correspondiente de forma separada y devuelve el objeto que modela el mensaje XML de respuesta que indican las especificaciones. El siguiente código muestra el método find_binding a modo de ejemplo:

Método find_binding de la clase atenea.Registro:

```
public BindingDetail find_binding(String serviceKey, TModelBag tModelBag,
            CategoryBag categoryBag,FindQualifiers findQualifiers,
            int maxRows)
            throws UDDIException {
         FindBinding consulta = new FindBinding();
         consulta.setGeneric(GENERIC_V2);
         consulta.setServiceKey(serviceKey);
         consulta.setTModelBag(tModelBag);
         consulta.setCategoryBag(categoryBag);
         consulta.setFindQualifiers(findQualifiers);
         consulta.setMaxRows(maxRows);

         return (BindingDetail)ejecutar(consulta);
    }
```

Como podemos ver, el método recoge los componentes de un mensaje de consulta find_binding de forma separada: la clave serviceKey, el tModelBag, el categoryBag, los calificadores de búsqueda findQualifiers y el parámetro que limita el número de resultados maxRows.

A continuación instancia un objeto de la clase atenea.tipos.request.FindBinding en el que empaqueta los componentes anteriores. Dicha clase se describió en el apartado 3.1.4. y modela un mensaje find_binding de la API de consulta.

Por último llama al método "ejecutar", que veremos a continuación y que está definido en la misma clase atenea.Registro. En este caso se llama a "ejecutar" con un objeto FindBinding y devolverá un objeto de tipo atenea.tipos.RegistryObject que convertiremos en un atenea.tipos.response.BindingDetail, objeto que modela el mensaje de respuesta correspondiente para este método de la API.

Todos los métodos de la clase atenea.Registro que modelan métodos de la API funcionan de la misma manera, recogen los componentes del mensaje por separado, los agrupan en un objeto que modela el mensaje de consulta, llaman al método "ejecutar" y devuelven el objeto que modela el mensaje de respuesta.

El método "ejecutar" del que hablamos se muestra a continuación:

Método ejecutar de la clase atenea.Registro:

```
public RegistryObject ejecutar(RegistryObject peticion)
    throws UDDIException {
```

```
        String nombreClase = peticion.getClass().getName();

        Funcion funcion = (Funcion)reserva.busca(nombreClase);
        if (funcion == null)
            throw new UnsupportedException(nombreClase);

        RegistryObject respuesta = funcion.ejecutar(peticion);

        return respuesta;
    }
```

Antes de explicar este método hay que hablar de las clases Funcion_xxx. Las clases Funcion_xxx son las que se encargan de realizar todo el trabajo de que hay detrás de cada uno de los métodos de la API. Se ha creado un paquete "atenea.funciones" donde organizar todas estas clases. A cada método de la API le corresponde una clase Funcion_xxx. Por ejemplo, al método find_binding anterior le correspondería la clase "atenea.funciones.FuncionFindBinding". Cada clase Funcion_xxx dispone de un método "ejecutar" que es el encargado de realizar el trabajo.

Cuando se llama al método "iniciar" de la clase atenea.Registro se instancia una clase que dispone de un HashMap con todas las clases Funcion y que las relaciona con el mensaje del método de la API correspondiente. Dicha instancia recibe el nombre de "reserva".

Si volvemos al código del método "ejecutar" de atenea.Registro vemos que lo primero que se hace es obtener el nombre completo de la clase del objeto que se ha utilizado para llamarlo. Con dicho nombre se llama al método "busca" de "reserva", que nos devolverá una instancia de la clase Funcion_xxx que corresponda. Por último se llama al método "ejecutar" de dicha clase para obtener el objeto de respuesta. En el caso de una llamada con un objeto FindBinding devolverá un objeto BindingDetail.

Veamos ahora el método "ejecutar" de las clases Funcion_xxx. Lo primero que hace es volver a desempaquetar los componentes que venían agrupados en el objeto mensaje con el que se le llamó. Se comprueba que estén todos los componentes obligatorios del mensaje que corresponda y si no es así se actúa como indica la norma en cada caso, devolviendo un mensaje vacío o algún tipo de error. Se realiza un análisis previo de los componentes del mensaje para comprobar la consistencia de los datos y si es posible se corrigen, en caso contrario se lanzará la excepción correspondiente. A continuación se obtiene una conexión con el repositorio instanciando la clase BaseDatos que veremos a continuación. Con dicha conexión podemos realizar comprobaciones más estrictas, como por ejemplo asegurarnos de que las claves suministradas existen en la base de datos. Una vez hechas todas las comprobaciones de seguridad puede llamarse al método correspondiente de la clase BaseDatos para que haga las modificaciones o consultas en el repositorio y devuelva los resultados si procede.

La clase "atenea.basedatos.BaseDatos" es la que agrupa toda la operativa de conexión con la base de datos, incluye una conexión y gestiona todas las operaciones mediante transacciones. Incluye un método para cada uno de los métodos de la API que realiza las operaciones en la base de datos que correspondan. Para ello llama de manera recursiva a los métodos necesarios de las clases que gestionan las tablas en la base de datos, y que describimos en el apartado 3.1.7.

3.1.10. Configuración del Registro

Para modificar los parámetros configurables del Registro se ha optado por crear un módulo de forma que el administrador pueda realizar esta labor sin tener que volver a compilar la aplicación.

Para ello se ha optado por construir una nueva tabla en la base de datos, de nombre CONFIGURACION, tal y como detallamos en el apartado 3.1.5. En dicha tabla se guardarán los parámetros siguientes:

- **OPERATOR**: Guarda el nombre del Operador. Por defecto se ha utilizado el nombre "Atenea", pero puede usarse cualquiera que se estime oportuno.

- **OPERATOR_URL**: Dirección públicamente accesible en la que residirá la aplicación. Se utiliza para crear automáticamente elementos DiscoveryURL para las estructuras BusinessEntity registradas en Atenea.

- **USUARIOS_PATH**: Ruta al fichero de configuración de usuarios en el que residen los datos de autenticación. Se recomienda que dicho fichero no se ubique en lugares fácilmente accesibles desde el exterior como el directorio raíz de la aplicación.

- **MAX_NAME_ELEMENTS**: Máximo número de elementos Name que pueden usarse en una búsqueda. Por defecto el valor es 5.

- **TIME_OUT**: Tiempo de caducidad, en milisegundos, de los tokens de autenticación. Por defecto este valor es de 3600000, es decir, de una hora.

Cuando se llama al método constructor de la clase principal de la aplicación, es decir, de "atenea.Registro", a su vez éste llama al método "cargaParametros".

Método cargaParametros de la clase atenea.Registro:

```
void cargaParametros() throws UDDIException {

        BaseDatos bd = null;
        Parametros param = null;
        try {
            // Conseguimos un manejador para el repositorio de datos
            bd = new BaseDatos();
            bd.empezarTransaccion();
            // Extraemos los parámetros
            param = bd.extraeParamReg();
            // Y los establecemos en la clase Registro
            estableceParam(param);
            bd.commit();
        } catch(SQLException e) {
            throw new UDDIException(e.getMessage());
        } finally {
            if (bd!= null)
                try {bd.release();} catch (SQLException sql_e) {
                    throw new UDDIException(sql_e.getMessage());}
        }
    }
```

Dicho método instancia la clase "atenea.basedatos.BaseDatos" para obtener una conexión con el repositorio y extraer los parámetros desde la base de datos. A continuación se llama al método "estableceParam" el cuál copia los valores obtenidos en los parámetros configurables de la clase "atenea.Registro" de modo que estén disponibles para el resto de clases.

3.1.11. Limpieza automática de tokens

La especificación UDDI no indica en ningún momento qué debe hacerse con los tokens caducados o desechados. Si bien desde el punto de vista teórico esto no es un problema, sí que lo es cuando se realiza una implementación.

Si no hiciéramos nada los tokens caducados simplemente se quedarían en la base de datos indefinidamente, ocupando un espacio que podría utilizarse para datos que realmente hicieran falta y seguramente haciendo el acceso a la misma más lento.

Para solucionar este pequeño inconveniente se ha optado por dotar a los tokens de un tiempo de caducidad, pasado el cuál el token deja de ser válido y debe ser borrado.

Al igual que ocurría con el módulo de configuración, cuando se llama al constructor de la clase "atenea.Registro" éste a su vez llama al método "limpiarTokens".

Método limpiarTokens de la clase atenea.Registro:

```
void limpiarTokens() throws UDDIException {

        BaseDatos bd = null;
        Vector tokenVector = null;
        AuthToken token = null;

        try {
            // Conseguimos un manejador para el repositorio de datos
            bd = new BaseDatos();
            bd.empezarTransaccion();
            bd.limpiarTokens();
            bd.commit();
        } catch(SQLException e) {
            throw new UDDIException(e.getMessage());
        } finally {
            if (bd!= null)
                try {bd.release();} catch (SQLException sql_e) {
                    throw new UDDIException(sql_e.getMessage());}
        }
    }
```

Como vemos, este método lo único que hace es instanciar la clase "atenea.basedatos.BaseDatos" y llamar a su método "limpiarTokens".

Método limpiarTokens de la clase atenea.basedatos.BaseDatos:

```
public void limpiarTokens() throws UDDIException {

        Vector keyVector = null;
        String token = null;
        long ultimoUso, timeOut, tiempoActual;

        try {
            // Extraemos todos los tokens del repositorio de datos
            keyVector = TablaAuthToken.extraeAuthTokens(conexion);

            // Analizamos uno a uno...
```

```
                if (keyVector!=null) {
                    if (keyVector.size()>0) {
                        for (int i=0; i<keyVector.size(); i++) {
                            token = (String)keyVector.elementAt(i);
                            // Extraemos el momento en el que se usó por última vez
                            ultimoUso=TablaAuthToken.selectUltimoUso(token,conexion);
                            if (ultimoUso > 0) {
                                timeOut = Registro.TIME_OUT;
                                tiempoActual = System.currentTimeMillis();
                                // Si está caducado, se borra
                                if ((tiempoActual-ultimoUso) >= timeOut)
                                    TablaAuthToken.delete(token,conexion);
                            }
                        }
                    }
                }

            } catch(java.sql.SQLException sql_ex) {throw new UDDIException(sql_ex);}
        }
```

Este método utiliza la clase de la tabla AuthToken para, en primer lugar, extraer los tokens y luego obtener la fecha de último uso de cada uno y comprobar si desde entonces ya ha transcurrido un tiempo mayor que el establecido de caducidad, en cuyo caso se vuelve a llamar a un método de esa clase para borrarlo.

Con este mecanismo nos aseguramos de que cada vez que se instancie la clase "atenea.Registro" se va a realizar una limpieza automática de los tokens del Registro.

3.2. Interfaz de usuario

Dado que no vamos a implementar la capa SOAP, es decir, la parte que procesaría los mensajes XML que llegasen al contenedor de aplicaciones y que los mapearía a un método Java, necesitamos un método alternativo para hacer uso de las APIs que hemos programado.

La solución que hemos adoptado ha sido la de desarrollar una interfaz de usuario básica que permita al usuario final registrar información y hacer consultas de una manera sencilla. Dicha interfaz estará compuesta por un conjunto de servlets que se ejecutarán en el mismo contenedor de aplicaciones de Atenea y que accederán a los métodos del Registro de una forma directa.

Como hemos comentado, para utilizar toda la potencia de las APIs programadas sería necesario implementar un procesador de mensajes XML que los mapee a los métodos de la API y viceversa, que transforme los objetos de respuesta de la API en mensajes XML y los envíe de vuelta al usuario. El objetivo principal de este Proyecto era el de implementar la funcionalidad de cada uno de los métodos de las APIs UDDI, cosa que ya hemos hecho al programar el Servidor; en este apartado crearemos una interfaz básica para que el usuario final pueda utilizar parte de lo que se ha hecho. Esta interfaz no pretende ser completa, ya que no utiliza todos los métodos definidos en las APIs, pero sí que permite acceder a los más importantes. Igualmente, tampoco utiliza toda la flexibilidad de que dispone el Registro para hacer consultas, pero es lo suficientemente práctica.

Para ello en primer lugar crearemos un fichero “index.jsp”, que colocaremos en el directorio WEB-INF y que hará las veces de página de inicio para la aplicación.

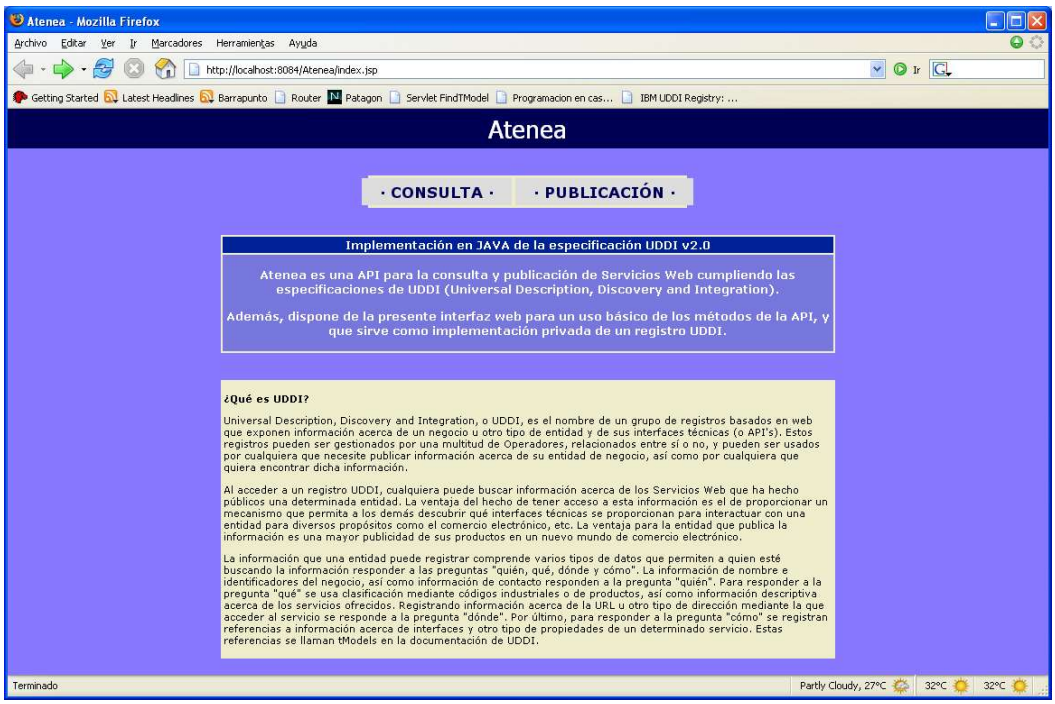

Figura 3.18. Página de inicio de la interfaz de usuario

Como podemos ver, esta página es muy sencilla y dispone de dos enlaces a sendos servlets. El botón "CONSULTA" lleva al servlet "Find", que manejará los métodos de la API de consulta. El botón "PUBLICACIÓN" lleva al servlet "Publish" que manejará los métodos de la API de publicación.

Dado que los servlets que vamos a programar tienen muchos métodos comunes, haremos que los hereden de una clase padre llamada "atenea.Interfaz", cuyo código se muestra a continuación:

Clase atenea.Interfaz:

```
/*
 * Interfaz.java
 *
 * Created on 22 de abril de 2005, 9:54
 */

package atenea;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Vector;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/** Interfaz: Clase que implementa los métodos más comúnmente usados por los servlets de
 *            la interfaz: FindTModel, FindService, SaveBusiness, etc...
 */
public abstract class Interfaz extends HttpServlet {

    /** Creates a new instance of Interfaz */
    public Interfaz() {
    }

    /** Genera la cabecera de un documento HTML con el título especificado
     *  @param out PrintWriter
     *  @param titulo String con el titulo del documento
     */
    void cabecera(PrintWriter out, String titulo) {
        out.println("<html>");
        out.println("<head>");
        out.println("<title>"+titulo+"</title>");
        out.println("<link rel=\"stylesheet\" href=\"atenea.css\">");
        out.println("</head>");
        out.println("<body>");
    }

    /** Cierra el documento HTML
     */
    void pie(PrintWriter out) {
        out.println("</body>");
        out.println("</html>");
    }

    /** Método que comprueba si un determinado campo existe, primero mirando
     *  que realmente el objeto no sea un null y luego comprobando que no es
     *  una cadena vacía, que podría devolver el formulario aunque el usuario
     *  no haya introducido datos.
     *  @param campo El valor del campo como objeto String
     *  @return resultado Un valor booleano, que es true si el campo existe.
     */
    boolean existe(String campo) {
        boolean resultado = false;
        if (campo!=null) {
            if (!campo.equals(""))
                resultado = true;
        }
        return resultado;
    }

    /** Muestra un mensaje de error en pantalla
     *  @param out PrintWriter
     *  @param texto String Texto a mostrar
     */
```

```
    void error(PrintWriter out, String texto) {
        out.println("<table align=\"center\"><tr id=\"error\"><td
                    align=\"center\">ERROR</td></tr>");
        out.println("<tr id=\"fields\"><td
                    align=\"center\"><b>"+texto+"</b></td></tr>");
        out.println("<tr><td align=\"center\"><a href=\"/Atenea\">Volver al
                    índice</a></td></tr>");
        out.println("</table>");
    }

    /** Muestra un mensaje en pantalla
     *  @param out PrintWriter
     *  @param texto String Texto a mostrar
     */
    void mensaje(PrintWriter out, String texto) {
        out.println("<table align=\"center\"><tr id=\"header\"><td
                    align=\"center\">ATENEA</td></tr>");
        out.println("<tr id=\"fields\"><td
                    align=\"center\"><b>"+texto+"</b></td></tr>");
        out.println("<tr><td align=\"center\"><a href=\"/Atenea\">Volver al
                    índice</a></td></tr>");
        out.println("</table>");
    }

    /** Devuelve true si hay indicios de una posible inyección de código malicioso
     *  en la consulta.
     *  @param Vector campos Conjunto de elementos String con el valor de cada campo
     *  @return boolean true si hay indicios de inyección SQL
     */
    boolean compruebaSQLInjection(Vector campos) {
        boolean inyeccion = false;
        if (campos != null) {
            int tamano = campos.size();
            for (int i=0; i<tamano; i++) {
                String campo = (String)campos.elementAt(i);
                if (campo != null)
                    if (campo.contains("\"")) inyeccion = true;
            }
        }
        return inyeccion;
    }

    /** Clase abstracta, se definirá según el servlet
     */
    protected void processRequest(HttpServletRequest request, HttpServletResponse
                                  response) throws ServletException, IOException {
    }

    /** Handles the HTTP <code>GET</code> method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
    }

    /** Handles the HTTP <code>POST</code> method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
    }

    /** Returns a short description of the servlet.
     */
    public String getServletInfo() {
        return "Short description";
    }

}
```

Esta clase define los métodos para generar la cabecera y el pie del código HTML de la página web que va a construir cada uno de los servlets. Además define unos métodos para mostrar mensajes de información y de error en pantalla. El método "existe" es ampliamente utilizado en los servlets para comprobar la existencia de

valores válidos en los parámetros que se le pasan. Por último, se ha implementado un método muy sencillo para detectar posibles ataques del tipo SQLInjection. Este tipo de ataques a la seguridad de una aplicación que utilice base de datos consiste en inyectar código en un formulario de modo que cuando la aplicación haga la consulta se produzca un fallo de seguridad, como el volcado de la estructura de la tabla u otro tipo de consecuencias realmente graves para la integridad y confidencialidad de los datos almacenados.

Con esta clase ya escrita, podemos crear los dos servlets principales de la interfaz: Find y Publish. Find sólo consiste en una lista de enlaces a cada uno de los servlets que manejan sendos métodos de la API de consulta. En cambio el servlet Publish requiere algo más de trabajo.

Dado que el acceso a la API de publicación está restringido, tendremos que implementar un mecanismo de autenticación en web. Para ello utilizamos una variable de sesión para guardar el token de autenticación. El servlet "atenea.Publish" tiene tres modos de funcionamiento en función de cómo se invoque. Si lo llamamos por primera vez, es decir, sin ningún tipo de información previa, mostrará un formulario para que el usuario introduzca su login y password. Al pulsar en el botón "OK" del formulario, se vuelve a llamar al mismo servlet pero pasando dichos valores mediante HTTP GET. Cuando el servlet es invocado con estos valores instancia la clase "atenea.Registro" y la utiliza para obtener un token válido. Si dicho token es un objeto null se vuelve a mostrar el formulario para que el usuario vuelva a autenticarse. Si por el contrario el token es válido se accede a la lista de enlaces para utilizar los métodos de la API de publicación.

Si por ejemplo abandonamos la página y volvemos a ella un momento después, la variable de sesión aún guardará nuestro token, por lo que no tendremos que volver a identificarnos y accederemos a la lista de métodos directamente. El tiempo de caducidad de esta variable de sesión se ha establecido en 15 minutos, por lo que si tardamos más habrá que repetir el proceso.

Figura 3.19. Autenticación de usuarios

3.2.1. Consulta de datos: Servlet Find

El servlet "atenea.Find", como ya dijimos, muestra en pantalla una lista de enlaces a cada uno de los servlets que se encargan de los métodos principales de la API de consulta. Su código es el siguiente:

Clase atenea.Find:

```
/*
 * Find.java
 *
 * Created on 19 de abril de 2005, 19:28
 */

package atenea;

import java.io.*;
import java.net.*;

import javax.servlet.*;
import javax.servlet.http.*;

/**
 *  Servlet que se encarga de presentar los enlaces a los servlets que se
 *  encargan de los métodos de la API de consulta.
 */
public class Find extends Interfaz {

    /** Processes requests for both HTTP <code>GET</code> and <code>POST</code>
     *  methods.
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest(HttpServletRequest request, HttpServletResponse
                                  response)throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        cabecera(out, "ATENEA: Bienvenido al servicio de Consulta");
        muestraFunciones(out);
        pie(out);

        out.close();
    }


    void muestraFunciones(PrintWriter out) {
        out.println("<center>");
        out.println("<h1>ATENEA: Bienvenido al servicio de Consulta</h1>");
        out.println("<h3>Pulse sobre el tipo de dato que quiere buscar o sobre el
                     que quiere obtener detalles.</h3>");
        out.println("<table>");
        out.println("<tr><td><h2 id=\"header\">· Buscar ·</h2></td></tr>");
        // FindTModel
        out.println("<tr><td><a href=/Atenea/FindTModel><h2
                     id=\"centro\">Technical Model</h2></a></td></tr>");
        // FindBinding
        out.println("<tr><td><a href=/Atenea/FindBinding><h2
                     id=\"centro\">Binding Template</h2></a></td></tr>");
        // FindService
        out.println("<tr><td><a href=/Atenea/FindService><h2
                     id=\"centro\">Business Service</h2></a></td></tr>");
        // FindBusiness
        out.println("<tr><td><a href=/Atenea/FindBusiness><h2
                     id=\"centro\">Business Entity</h2></a></td></tr>");
        out.println("</table><table>");
        out.println("<tr><td><h2 id=\"header\">· Información detallada
                     ·</h2></td></tr>");
        // GetTModelDetail
        out.println("<tr><td><a href=/Atenea/GetTModelDetail><h2
                     id=\"centro\">Technical Model</h2></a></td></tr>");

// GetBindingDetail
        out.println("<tr><td><a href=/Atenea/GetBindingDetail><h2
                     id=\"centro\">Binding Template</h2></a></td></tr>");
        // GetServiceDetail
        out.println("<tr><td><a href=/Atenea/GetServiceDetail><h2
                     id=\"centro\">Business Service</h2></a></td></tr>");
        // GetBusinessDetail
        out.println("<tr><td><a href=/Atenea/GetBusinessDetail><h2
```

```
                        id=\"centro\">Business Entity</h2></a></td></tr>");
            out.println("</table>");

            out.println("</center>");
        }

    }
```

El resultado puede verse en la figura 3.20.

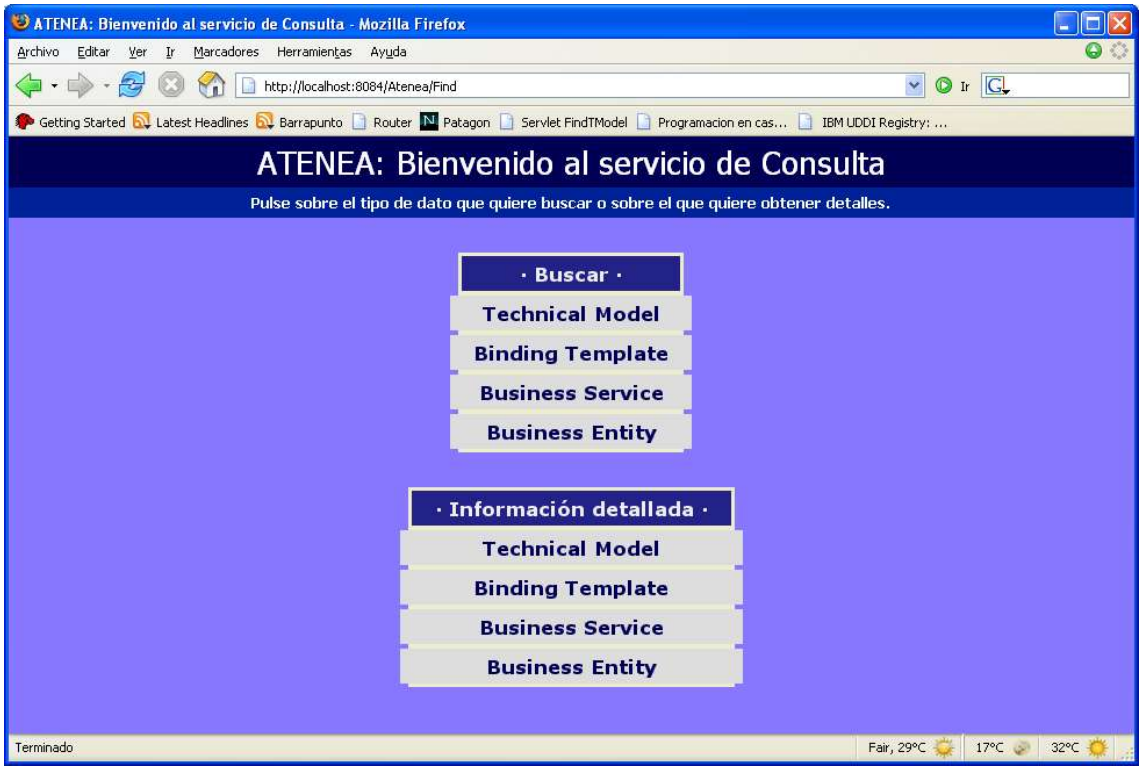

Figura 3.20. Servlet Find

Si pulsamos por ejemplo en el botón "BusinessEntity" de la sección "Buscar" accederemos al servlet "atenea.FindBusiness", cuyo código se reproduce a continuación.

Clase atenea.FindBusiness:

```
/*
 * FindBusiness.java
 *
 * Created on 23 de abril de 2005, 18:06
 */

package atenea;

import atenea.error.UDDIException;
import atenea.tipos.CategoryBag;
import atenea.tipos.DiscoveryURL;
import atenea.tipos.DiscoveryURLs;
import atenea.tipos.IdentifierBag;
import atenea.tipos.KeyedReference;
import atenea.tipos.Name;
import atenea.tipos.TModelBag;
import atenea.tipos.request.FindQualifier;
import atenea.tipos.request.FindQualifiers;
import atenea.tipos.response.BusinessInfo;
import atenea.tipos.response.BusinessInfos;
import atenea.tipos.response.BusinessList;
import atenea.tipos.tmodel.TModel;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Vector;
import javax.servlet.ServletException;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


/**
 * Servlet encargado de gestionar llamadas al Registro para realizar
 * consultas FindBusiness.
 */
public class FindBusiness extends Interfaz {

       private String name, discoveryURL, discoveryURLUseType, tModelKey, categoria,
identificador;
    private String catName, catValue, idValue;
    private String exactMatch, caseSensitive, sortName, sortDate;
    private Registro reg;
    private static final int MAX_ROWS = 100;

    /** Processes requests for both HTTP <code>GET</code> and <code>POST</code> methods.
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest(HttpServletRequest request, HttpServletResponse
                                  response)throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        cabecera(out,"Servlet FindBusiness");

        extraerParametros(request);
        if (!existe(name) && !existe(discoveryURL) && !existe(tModelKey) &&
                (!existe(categoria) || categoria.equals("None")) &&
                (!existe(identificador) || identificador.equals("None")))
            mostrarFormulario(out);
        else
            procesarBusqueda(out);

        pie(out);

        out.close();
    }

    void extraerParametros(HttpServletRequest request) {
        name = request.getParameter("name");
        discoveryURL = request.getParameter("discoveryURL");
        discoveryURLUseType = request.getParameter("discoveryURLUseType");
        tModelKey = request.getParameter("tModelKey");
        categoria = request.getParameter("categoria");
        catName = request.getParameter("catName");
        catValue = request.getParameter("catValue");
        identificador = request.getParameter("identificador");
        idValue = request.getParameter("idValue");
        exactMatch = request.getParameter("exactMatch");
        caseSensitive = request.getParameter("caseSensitive");
        sortName = request.getParameter("sortName");
        sortDate = request.getParameter("sortDate");
    }

    void mostrarFormulario(PrintWriter out) {
        out.println("<h1>ATENEA: Formulario para buscar Business Entity</h1>");
        out.println("<h3>Introduzca valores por los que buscar usando uno o más de los
                     criterios de búsqueda ");
        out.println("abajo indicados, y luego pulse el botón Buscar para comenzar la
                     búsqueda.</h3>");
        out.println("<h3>Puede usar el caracter '%' como comodín.</h3>");

        out.println("<table width=\"80%\" align=\"center\">");
        out.println("<form method=\"POST\" action=\"/Atenea/FindBusiness\">");

        out.println("<tr id=\"header\"><td colspan=\"2\">");
        out.println("<formSectionTitle>Identificación de la Business
                     Entity</formSectionTitle></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">Nombre de la
                     businessEntity</td>");
        out.println("<td><input type=\"text\" name=\"name\" size=\"50\"
                     maxlength=\"255\"></td></tr>");
        out.println("<tr><td align=\"right\">Exact Match?</td><td><input type=\"radio\"
                     name=\"exactMatch\" value=\"yes\"> Sí  ");
        out.println("<input type=\"radio\" name=\"exactMatch\" value=\"no\" checked>
                     No</td></tr>");
        out.println("<tr><td align=\"right\">Case sensitive?</td><td><input
                     type=\"radio\" name=\"caseSensitive\" value=\"yes\"> Sí  ");
```

```
        out.println("<input type=\"radio\" name=\"caseSensitive\" value=\"no\" checked>
                    No</td></tr>");

        out.println("<tr id=\"header\"><td colspan=\"2\">");
        out.println("<formSectionTitle>URL del Business
                    Entity</formSectionTitle></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">discoveryURL</td>");
        out.println("<td><input type=\"text\" name=\"discoveryURL\" size=\"50\"
                    maxlength=\"255\"></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">useType</td>");
        out.println("<td><input type=\"text\" name=\"discoveryURLUseType\" size=\"50\"
                    maxlength=\"255\"></td></tr>");

        out.println("<tr id=\"header\"><td colspan=\"2\">");
        out.println("<formSectionTitle>Huella técnica del Business
                    Entity</formSectionTitle></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">tModelKey</td>");
        out.println("<td><input type=\"text\" name=\"tModelKey\" size=\"41\"
                    maxlength=\"41\"></td></tr>");

        out.println("<tr id=\"header\"><td colspan=\"2\">");
        out.println("<formSectionTitle>Categoría</formSectionTitle></td></tr>");
        out.println("<tr><td align=\"right\">Categoría</td>");
        out.println("<td><select name=\"categoria\">");
        out.println("<option value=\"None\">Elija una");
        out.println("<option value=\"UDDITypes\">UDDI Types Taxonomy");
        out.println("<option value=\"GeneralKeywords\">General Keywords");
        out.println("<option value=\"NAICS\">NAICS (1997)");
        out.println("<option value=\"UNSPSC\">UNSPSC");
        out.println("<option value=\"UNSPSC31\">UNSPSC (Versión 3.1)");
        out.println("<option value=\"ISO3166\">ISO 3166 (GEO)");
        out.println("<option value=\"OwningBusiness\">OwningBusiness");
        out.println("<option value=\"Operators\">Operators");
        out.println("</select></td>");
        out.println("<tr><td width=\"20%\" align=\"right\">Nombre </td>");
        out.println("<td><input type=\"text\" name=\"catName\" size=\"50\"
                    maxlength=\"255\"></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">Valor </td>");
        out.println("<td><input type=\"text\" name=\"catValue\" size=\"50\"
                    maxlength=\"255\"></td></tr>");

        out.println("<tr id=\"header\"><td colspan=\"2\">");
        out.println("<formSectionTitle>Identificador</formSectionTitle></td></tr>");
        out.println("<tr><td align=\"right\">Identificador</td>");
        out.println("<td><select name=\"identificador\">");
        out.println("<option value=\"None\">Elija uno");
        out.println("<option value=\"isReplacedBy\">isReplacedBy");
        out.println("<option value=\"DUNS\">D-U-N-S");
        out.println("<option value=\"Thomas\">Thomas Register");
        out.println("</select></td>");
        out.println("<tr><td width=\"20%\" align=\"right\">Valor </td>");
        out.println("<td><input type=\"text\" name=\"idValue\" size=\"50\"
                    maxlength=\"255\"></td></tr>");

        out.println("<tr id=\"header\"><td colspan=\"2\">");
        out.println("<formSectionTitle>FindQualifiers</formSectionTitle></td></tr>");
        out.println("<tr><td align=\"right\">Ordenar por Nombre</td><td><input
                 type=\"radio\" name=\"sortName\" value=\"asc\" checked> Ascendente  ");
        out.println("<input type=\"radio\" name=\"sortName\" value=\"desc\">
                    Descendente</td></tr>");
        out.println("<tr><td align=\"right\">Ordenar por Fecha</td><td><input
               type=\"radio\" name=\"sortDate\" value=\"asc\" checked> Ascendente  ");
        out.println("<input type=\"radio\" name=\"sortDate\" value=\"desc\">
                    Descendente</td></tr>");
        out.println("<tr><td colspan=\"2\"><input type=\"submit\"
                    value=\"Buscar\"></td></tr>");
        out.println("</table>");
        out.println("</form>");
    }

    void procesarBusqueda(PrintWriter out){

        out.println("<h1>ATENEA: Resultado de la Búsqueda de Business Entity</h1>");

        // Si existe un name creamos un vector de objetos Name
        Vector nameVector = null;
        if (existe(name)) {
            nameVector = new Vector();
            Name nombre = new Name(name);
            nameVector.add(nombre);
        }
```

```
// Creamos el objeto DiscoveryURLs
DiscoveryURLs discURLs = null;
if (existe(discoveryURL)) {
    discURLs = new DiscoveryURLs();
    DiscoveryURL discURL = new DiscoveryURL();
    discURL.setUrlValue(discoveryURL);
    if (existe(discoveryURLUseType))
        discURL.setUseType(discoveryURLUseType);
    discURLs.addDiscoveryURL(discURL);
}

// Generamos el objeto identifierBag
IdentifierBag iBag = null;
if (existe(identificador) && !identificador.equals("None")) {
    iBag = new IdentifierBag();
    KeyedReference keyedRef = new KeyedReference();
    keyedRef.setKeyValue(idValue);
    if (identificador.equals("isReplacedBy"))
        keyedRef.setTModelKey(TModel.ISREPLACEDBY_TMODEL_KEY);
    if (identificador.equals("DUNS"))
        keyedRef.setTModelKey(TModel.DUNS_TMODEL_KEY);
    if (identificador.equals("Thomas"))
        keyedRef.setTModelKey(TModel.THOMAS_REGISTER_TMODEL_KEY);
    iBag.addKeyedReference(keyedRef);
}


// Generamos el objeto categoryBag
CategoryBag cBag = null;
if (existe(categoria) && !categoria.equals("None")) {
    cBag = new CategoryBag();
    KeyedReference keyedRef = new KeyedReference();
    if (categoria.equals("UDDITypes"))
        keyedRef.setTModelKey(TModel.TYPES_TMODEL_KEY);
    if (categoria.equals("GeneralKeywords"))
        keyedRef.setTModelKey(TModel.GENERAL_KEYWORDS_TMODEL_KEY);
    if (categoria.equals("NAICS"))
        keyedRef.setTModelKey(TModel.NAICS_TMODEL_KEY);
    if (categoria.equals("UNSPSC"))
        keyedRef.setTModelKey(TModel.UNSPSC_TMODEL_KEY);
    if (categoria.equals("UNSPSC31"))
        keyedRef.setTModelKey(TModel.UNSPSC_31_TMODEL_KEY);
    if (categoria.equals("ISO3166"))
        keyedRef.setTModelKey(TModel.ISO_CH_TMODEL_KEY);
    if (categoria.equals("OwningBusiness"))
        keyedRef.setTModelKey(TModel.OWNING_BUSINESS_TMODEL_KEY);
    if (categoria.equals("Operators"))
        keyedRef.setTModelKey(TModel.OPERATORS_TMODEL_KEY);
    if (existe(catName))
        keyedRef.setKeyName(catName);
    if (existe(catValue))
        keyedRef.setKeyValue(catValue);
    cBag.addKeyedReference(keyedRef);
}


// Creamos el objeto TModelBag
TModelBag tBag = null;
if (existe(tModelKey)) {
    tBag = new TModelBag();
    tBag.addTModelKey(tModelKey);
}

// Generamos el objeto FindQualifiers
FindQualifiers qual = new FindQualifiers();
if (!existe(sortName))
    qual.addFindQualifier(new FindQualifier(FindQualifier.SORT_BY_NAME_ASC));
else {
    if (sortName.equals("desc"))
      qual.addFindQualifier(new FindQualifier(FindQualifier.SORT_BY_NAME_DESC));
    else
      qual.addFindQualifier(new FindQualifier(FindQualifier.SORT_BY_NAME_ASC));
}
if (!existe(sortDate))
    qual.addFindQualifier(new FindQualifier(FindQualifier.SORT_BY_DATE_ASC));
else {
    if (sortDate.equals("desc"))
      qual.addFindQualifier(new FindQualifier(FindQualifier.SORT_BY_DATE_DESC));
    else
      qual.addFindQualifier(new FindQualifier(FindQualifier.SORT_BY_DATE_ASC));
}
if (existe(exactMatch))
```

```
            if (exactMatch.equals("yes"))
              qual.addFindQualifier(new FindQualifier(FindQualifier.EXACT_NAME_MATCH));
        if (existe(caseSensitive))
            if (caseSensitive.equals("yes"))
           qual.addFindQualifier(new FindQualifier(FindQualifier.CASE_SENSITIVE_MATCH));

        // Mecanismo de seguridad anti-SQLInjection
        // Devuelve false si hay algún tipo de amenaza
        if (!comprobacionSeguridad()) {
            error(out,"Por favor no intente inyectar SQL");
            return;
        }


        // Hacemos la consulta
        try {
            reg = new Registro();
            reg.iniciar();
            if (reg.disponible()){
                BusinessList businessList = reg.find_business(nameVector, discURLs,
                                                iBag, cBag, tBag, qual, MAX_ROWS);
                if (businessList != null){
                    BusinessInfos infos = businessList.getBusinessInfos();
                    if (infos!=null) {
                        Vector infoVector = infos.getBusinessInfoVector();
                        if (infoVector!=null && infoVector.size()>0)
                            dibujaTabla(infoVector, out);
                        else error(out, "No hay resultados que mostrar");
                    }
                } else error(out, "No hay resultados");
            } else error(out, "Registro no disponible");
            reg.apagar();
        } catch (UDDIException e){ error(out, "Excepción: "+ e.getMessage());}

    }

    void dibujaTabla(Vector infoVector, PrintWriter out) {

        boolean esPar = false;
        boolean mostrar = false;

        out.println("<table><tr id=\"header\"><td colspan=\"2\">Business
                     Entity</td></tr>");
        out.println("<tr id=\"fields\"><td>Name</td><td>businessKey</td></tr>");

        for (int i=0; i<infoVector.size(); i++){
            BusinessInfo info = (BusinessInfo)infoVector.elementAt(i);
            Vector nameVector = info.getNameVector();
            String businessKey = info.getBusinessKey();
            Vector descVector = info.getDescriptionVector();

            if (esPar)
                out.println("<tr id=\"par\">");
            else
                out.println("<tr id=\"impar\">");
            esPar = !esPar;

            if (nameVector!=null){
                if (nameVector.size()>0) {
                    out.println("<td>");
                    for (int k=0; k<nameVector.size(); k++) {

                        // Implementamos aqui el caseSensitive, que no funciona en
                        // Windows
                        Name nombre = (Name)nameVector.elementAt(k);
                        if ( (caseSensitive.equals("yes") &&
                              nombre.getNameValue().startsWith(name)) ||
                              caseSensitive.equals("no") ) {
                            mostrar = true;
                            out.println(nombre.getNameValue()+"<br>");
                        }
                    }
                    out.println("</td>");
                } else out.println("<td></td>");
            }

            if (mostrar) {
                out.println("<td><a
              href=\"GetBusinessDetail?key="+businessKey+"\">"+businessKey+"</a></td>");
            } else out.println("<td></td>");
            out.println("</tr>");
        }
```

```
        out.println("</table>");
    }

    /** Mecanismo básico que parsea cada campo del formulario para que no se
     *  inyecte código malicioso al servidor
     */
    boolean comprobacionSeguridad() {
        boolean todoOK = true;
        Vector campos = new Vector();
        campos.add(name);
        campos.add(discoveryURL);
        campos.add(discoveryURLUseType);
        campos.add(tModelKey);
        campos.add(categoria);
        campos.add(catName);
        campos.add(catValue);
        campos.add(identificador);
        campos.add(idValue);
        campos.add(exactMatch);
        campos.add(caseSensitive);
        campos.add(sortName);
        campos.add(sortDate);
        todoOK = !compruebaSQLInjection(campos);
        return todoOK;
    }
}
```

Este servlet funciona de la siguiente manera: cuando se invoca por primera vez, esto es, sin parámetros, muestra un formulario de entrada de datos como el de la figura 3.21. El usuario utiliza los campos de entrada de texto y los selectores para configurar los parámetros de búsqueda de la Business Entity y a continuación pulsa el botón "Buscar". En ese momento el formulario envía los datos usando GET al mismo servlet FindBusiness. Cuando el servlet recibe estos datos comprueba que existen los mínimos necesarios para hacer una búsqueda, como puede verse en el siguiente extracto de código. Si está todo correcto se procesa la búsqueda, en caso contrario se vuelve a mostrar el formulario para que el usuario introduzca unos nuevos criterios.

Extracto del código de atenea.FindBusiness:

```
if (!existe(name) && !existe(discoveryURL) && !existe(tModelKey) &&
   (!existe(categoria) || categoria.equals("None")) &&
   (!existe(identificador) || identificador.equals("None")))
            mostrarFormulario(out);
        else
            procesarBusqueda(out);
```

En el método que procesa la búsqueda, y en función de los parámetros de búsqueda que hayamos usado en el formulario, se crearán los elementos del mensaje de consulta FindBusiness. Antes de llamar a la base de datos se llamará al método que identifica posibles intentos de inyección SQL y si es así se generará un mensaje de error advirtiendo de ello al usuario. Por último se instancia la clase "atenea.Registro" y se llama a su método "find_business" con los elementos recién creados.

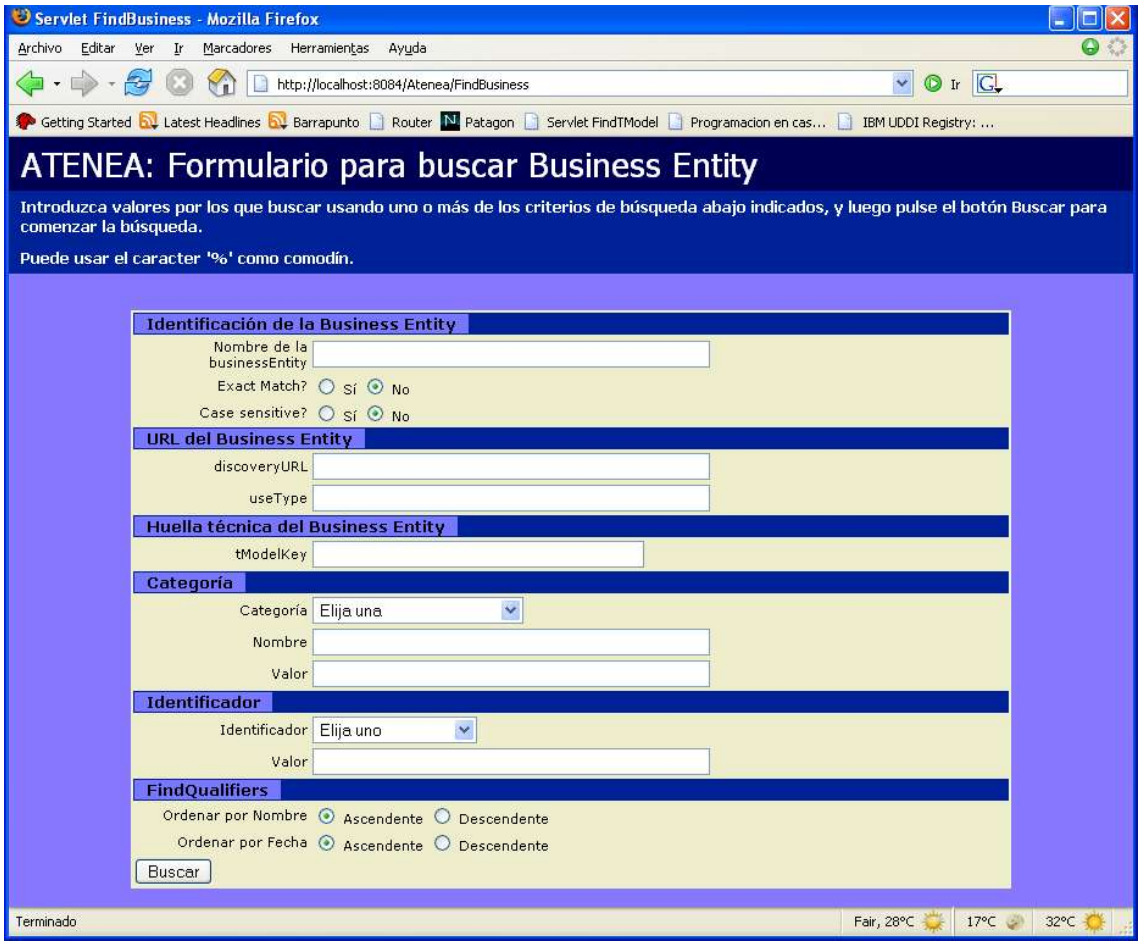

Figura 3.21. Formulario de FindBusiness

Dicho método devolverá un objeto BusinessDetail con el resultado de la búsqueda. Si hay resultados que mostrar se llamará al método "dibujaTabla" para que los muestre en pantalla, en caso contrario se mostrará un mensaje indicando que no hubo resultados en la búsqueda.

En la figura 3.22 se muestra el posible resultado de una búsqueda.

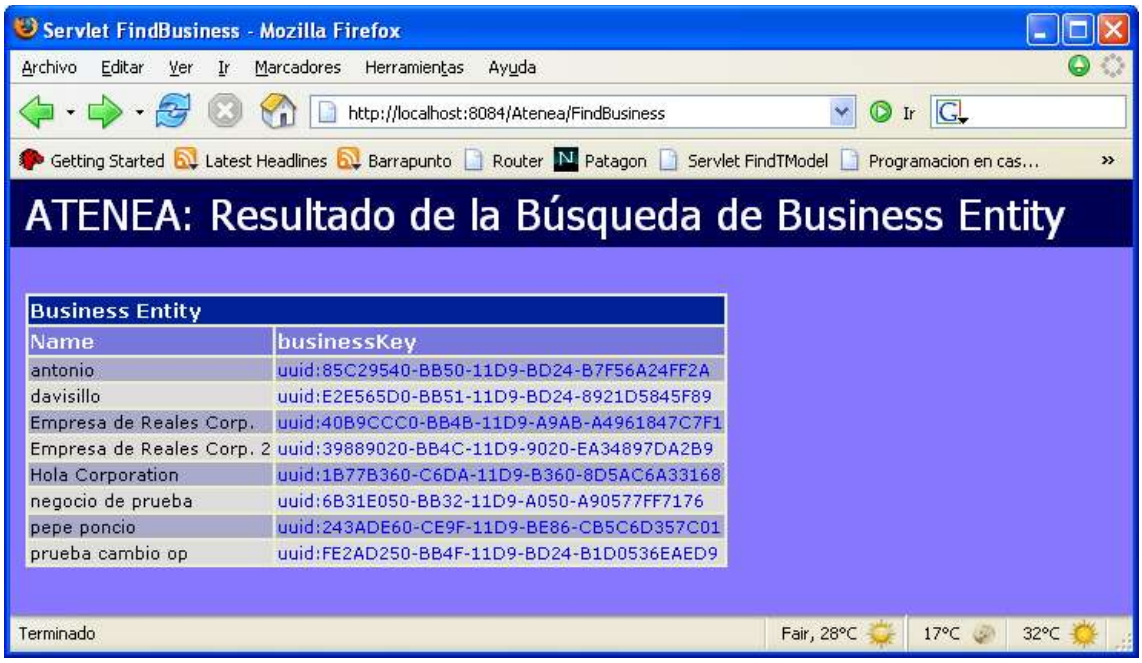

Figura 3.22. Resultado de una búsqueda FindBusiness

Como podemos ver, se muestra el elemento Name y la BusinessKey de cada una de las BusinessEntity que han cumplido con los requisitos de la búsqueda. Si pulsamos sobre cualquiera de estas BusinessKey llamaremos al servlet GetBindingDetail con el valor de la clave usando un método http GET. Dicho servlet funciona de manera análoga al que acabamos de describir, y muestra información detallada sobre la entidad, como se ve en la figura 3.23.

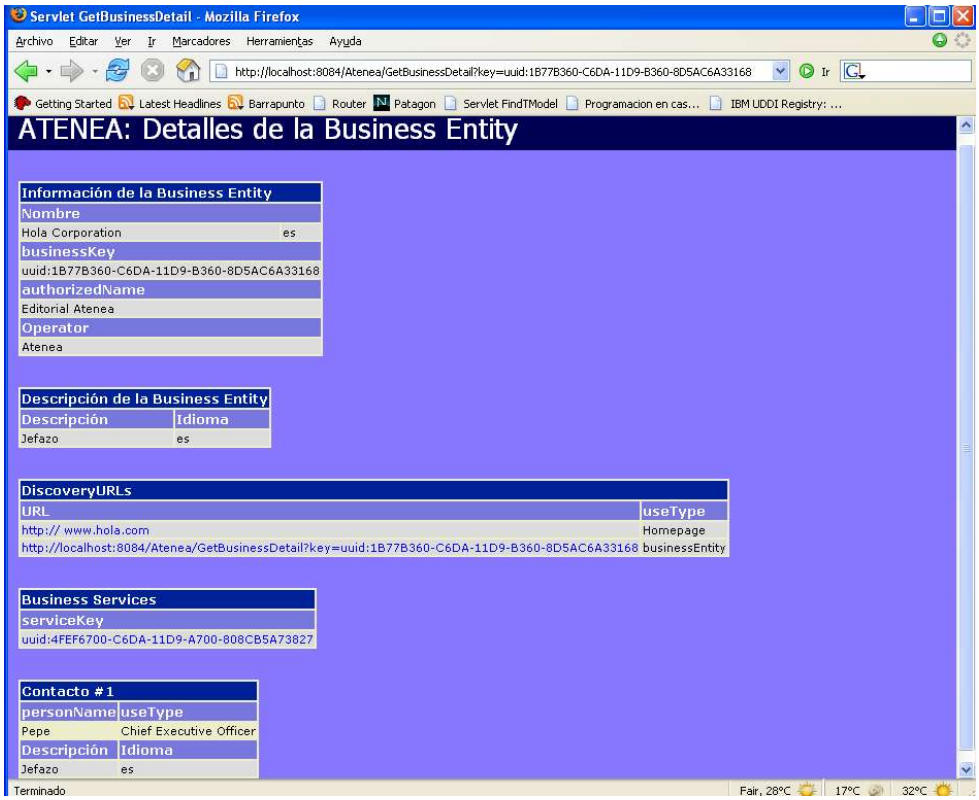

Figura 3.23. Información detallada de una BusinessEntity

En la figura 3.23 se advierten dos elementos DiscoveryURL, el primero de ellos introducido por el agente humano encargado de registrar la información y el segundo insertado de forma automática por el Registro y que apunta precisamente a esta página de información. El Registro utiliza el parámetro de configuración OPERATOR_URL para crear esta dirección.

Además podemos ver que la BusinessEntity tiene un Servicio. Hemos creado un enlace que apunta al servlet GetServiceDetail y que le pasa como parámetro la ServiceKey usando un método http GET. Este servlet funciona de forma parecida a GetBusinessDetail, el resultado de la consulta sería el de la figura 3.24.

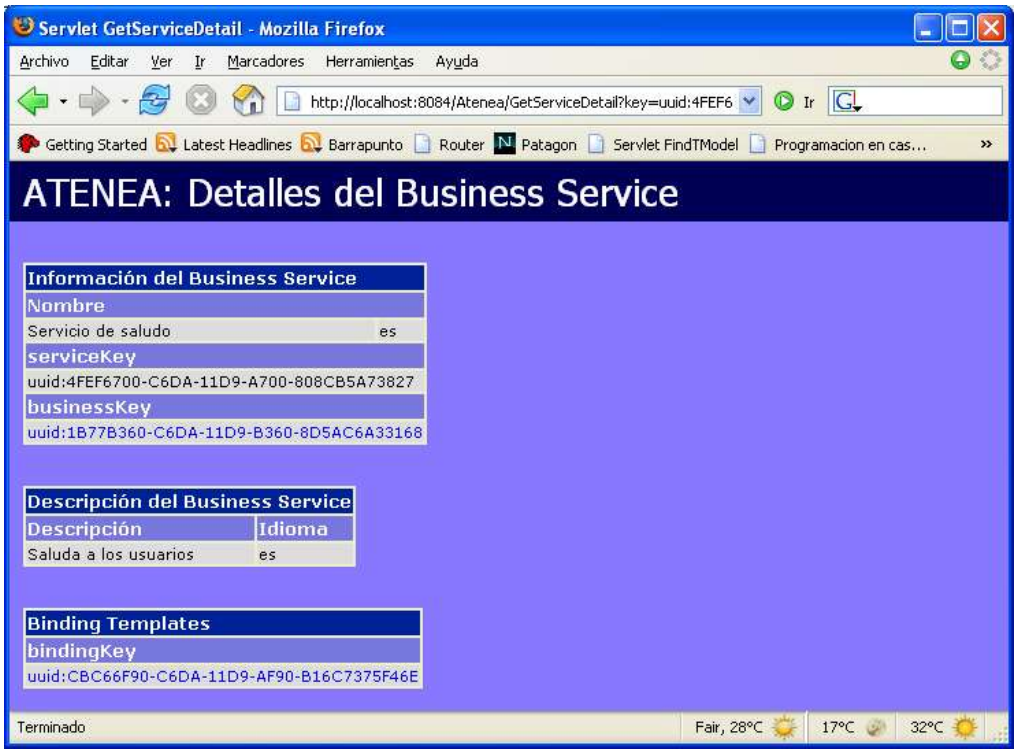

Figura 3.24. Información detallada de un BusinessService

En total, la parte de la interfaz dedicada a la API de consulta dispone de los siguientes servlets para buscar y mostrar información:

- FindBinding
- FindBusiness
- FindService
- FindTModel
- GetBindingDetail
- GetBusinessDetail
- GetServiceDetail
- GetTModelDetail

3.2.2. Publicación y modificación de datos: Servlet Publish

El servlet "atenea.Publish" muestra en pantalla una lista de enlaces a cada uno de los servlets que se encargan de los métodos principales de la API de publicación. Su código es el siguiente:

Clase atenea.Publish:

```
/*
 * Publish.java
 *
 * Created on 14 de abril de 2005, 19:19
 */

package atenea;

import atenea.error.UDDIException;
import atenea.tipos.response.AuthToken;
import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;

/**
 *  Servlet que se encarga de presentar los enlaces a los servlets que se
 *  encargan de los métodos de la API de publicación.
 */
public class Publish extends Interfaz {

    private String userID, cred;
    private AuthToken token = null;
    private Registro reg;

    /** Processes requests for both HTTP <code>GET</code> and <code>POST</code> methods.
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest(HttpServletRequest request, HttpServletResponse
                                  response) throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        HttpSession session = request.getSession(true);

        // Expiracion de la sesión en 15 minutos
        session.setMaxInactiveInterval(60*15);

        cabecera(out, "ATENEA: Bienvenido al servicio de Publicación");
        extraeParametros(request);

        // Si nos pasan parámetros de autenticación, intentamos obtener un token
        if ((userID!=null) && (cred!=null)) {
            token = obtieneToken(userID, cred, out);
            session.setAttribute("token", token);
        }

        // Si tenemos un token en la variable de sesión podemos continuar
        // Si no, pedimos al usuario que se identifique
        if (session.getAttribute("token") != null)
            muestraFunciones(out);
        else
            mostrarFormulario(out);

        pie(out);
        out.close();
    }

    void muestraFunciones(PrintWriter out) {
        out.println("<center>");
        out.println("<h1>ATENEA: Bienvenido al servicio de Publicación</h1>");
        out.println("<h3>Pulse sobre el tipo de dato que desea publicar o
                     eliminar</h3>");
        out.println("<table>");
        out.println("<tr><td><h2 id=\"header\">· Guardar ·</h2></td></tr>");
        // Save findTModel
        out.println("<tr><td><a href=/Atenea/SaveTModel><h2 id=\"centro\">Technical
```

```
                    Model</h2></a></td></tr>");
        // Save bindingTemplate
        out.println("<tr><td><a href=/Atenea/SaveBinding><h2 id=\"centro\">Binding
                    Template</h2></a></td></tr>");
        // Save businessService
        out.println("<tr><td><a href=/Atenea/SaveService><h2 id=\"centro\">Business
                    Service</h2></a></td></tr>");
        // Save businessEntity
        out.println("<tr><td><a href=/Atenea/SaveBusiness><h2 id=\"centro\">Business
                    Entity</h2></a></td></tr>");
        out.println("</table><table>");
        out.println("<tr><td><h2 id=\"header\">· Borrar ·</h2></td></tr>");
        // Delete tModel
        out.println("<tr><td><a href=/Atenea/DeleteTModel><h2 id=\"centro\">Technical
                    Model</h2></a></td></tr>");
        // Delete bindingTemplate
        out.println("<tr><td><a href=/Atenea/DeleteBinding><h2 id=\"centro\">Binding
                    Template</h2></a></td></tr>");
        // Delete businessService
        out.println("<tr><td><a href=/Atenea/DeleteService><h2 id=\"centro\">Business
                    Service</h2></a></td></tr>");
        // Delete businessEntity
        out.println("<tr><td><a href=/Atenea/DeleteBusiness><h2 id=\"centro\">Business
                    Entity</h2></a></td></tr>");
        out.println("</table></center>");
    }

    void extraeParametros(HttpServletRequest request) {
        userID = request.getParameter("userID");
        cred = request.getParameter("cred");
    }

    void mostrarFormulario(PrintWriter out) {
        out.println("<center>");
        out.println("<h1>ATENEA: Introduzca su identificación de usuario</h1>");
        out.println("<table width=\"80%\" align=\"center\">");
        out.println("<form method=\"POST\" action=\"Publish\">");
        out.println("<tr id=\"header\"><td colspan=\"2\">");

        out.println("<formSectionTitle>Datos de
                    autenticación</formSectionTitle></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">userID </td>");
        out.println("<td><input type=\"text\" name=\"userID\" size=\"20\"
                    maxlength=\"20\"></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">password </td>");
        out.println("<td><input type=\"password\" name=\"cred\" size=\"20\"
                    maxlength=\"20\"></td></tr>");

        out.println("<tr><td></td><td colspan=\"2\"><input type=\"submit\"
                    value=\"OK\"></td></tr>");
        out.println("</form></table>");
        out.println("</center>");
    }

    AuthToken obtieneToken(String userID, String cred, PrintWriter out) {

        AuthToken token = null;

        try {
            reg = new Registro();
            reg.iniciar();
            if (reg.disponible()){
                token = reg.get_authToken(userID, cred);
            }
            reg.apagar();
        } catch (UDDIException e){}

        return token;
    }

}
```

Como comentamos anteriormente este servlet se encarga de gestionar el acceso a los métodos de la API de publicación presentando un formulario para la autenticación de usuarios. Una vez autenticado muestra la lista de enlaces a los servlets que se encargan de los métodos principales de esta API, como puede verse en la figura 3.25.

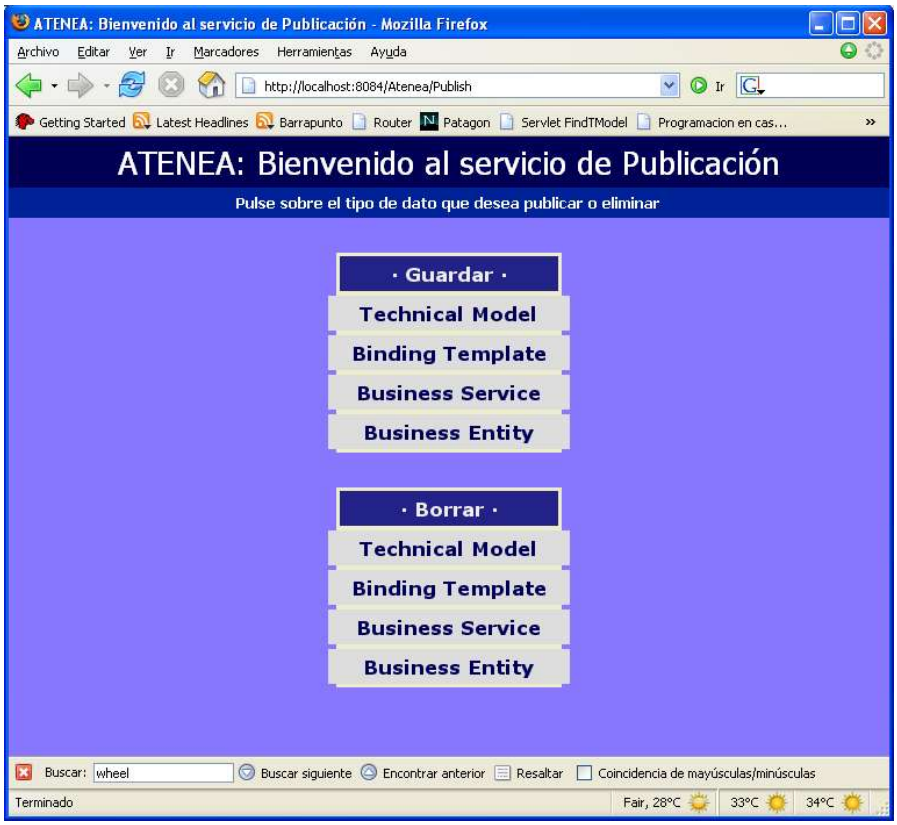

Figura 3.25. Servlet Publish

Si pulsamos por ejemplo en el botón “BusinessService” de la sección “Guardar” accederemos al servlet “atenea.SaveService”, cuyo código se reproduce a continuación:

Clase atenea.SaveService:

```
/*
 * SaveService.java
 *
 * Created on 19 de abril de 2005, 13:05
 */

package atenea;

import atenea.error.UDDIException;
import atenea.tipos.Description;
import atenea.tipos.KeyedReference;
import atenea.tipos.Name;
import atenea.tipos.request.AuthInfo;
import atenea.tipos.response.AuthToken;
import atenea.tipos.service.BusinessService;
import atenea.tipos.tmodel.TModel;
import java.io.*;
import java.util.Vector;

import javax.servlet.*;
import javax.servlet.http.*;


/**
 * Servlet encargado de gestionar llamadas al Registro para realizar
 * peticiones SaveService.
 */
public class SaveService extends Interfaz {

    private String businessKey, serviceKey, nameValue, nameLang;
    private String description, descriptionLang;
    private String category, catName, catValue;
    private BusinessService service;
    private AuthInfo authInfo;
```

```
    private Registro reg;

    /** Processes requests for both HTTP <code>GET</code> and <code>POST</code> methods.
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest(HttpServletRequest request, HttpServletResponse
                                  response) throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        HttpSession session = request.getSession(true);
        AuthToken token = (AuthToken)session.getAttribute("token");

        cabecera(out,"Servlet Save Business Service");

        // Comprobamos que tenemos un token válido
        if (token == null) {
            out.println("<h1>ATENEA: Save BusinessService");
            error(out,"No se ha identificado o su identificación ha caducado");
            pie(out);
            out.close();
            return;
        } else {
            authInfo = token.getAuthInfo();
        }

        extraerParametros(request);

        if (!existe(businessKey))
            mostrarFormulario(out);
        else
            procesar(out);

        pie(out);
        out.close();
    }


    void extraerParametros(HttpServletRequest request) {
        businessKey = request.getParameter("businessKey");
        serviceKey = request.getParameter("serviceKey");
        nameValue = request.getParameter("nameValue");
        nameLang = request.getParameter("nameLang");
        description = request.getParameter("description");
        descriptionLang = request.getParameter("descriptionLang");
        category = request.getParameter("category");
        catName = request.getParameter("catName");
        catValue = request.getParameter("catValue");
    }

    void mostrarFormulario(PrintWriter out) {
        out.println("<h1>ATENEA: Formulario para guardar un businessService");
        out.println("<h3>Introduzca los valores del businessService que desea
                     guardar.</h3>");
        out.println("<h3>Si es un businessService nuevo deje el campo serviceKey en
                     blanco.</h3>");
        out.println("<h3>Los campos marcados con un (*) son obligatorios.</h3>");

        out.println("<table width=\"80%\" align=\"center\">");
        out.println("<form method=\"POST\" action=/Atenea/SaveService>");

        // Identificación del businessService
        out.println("<tr id=\"header\"><td colspan=\"2\">");
        out.println("<formSectionTitle>Identificación del
                     BusinessService</formSectionTitle></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">businessKey(*)</td>");
        out.println("<td><input type=\"text\" name=\"businessKey\" size=\"41\"
                     maxlength=\"41\"></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">serviceKey</td>");
        out.println("<td><input type=\"text\" name=\"serviceKey\" size=\"41\"
                     maxlength=\"41\"></td></tr>");

        // Nombre y descripción
        out.println("<tr id=\"header\"><td colspan=\"2\">");
        out.println("<formSectionTitle>Nombre y
                     descripción</formSectionTitle></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">Nombre del
                     businessService</td>");
        out.println("<td><input type=\"text\" name=\"nameValue\" size=\"50\"
                     maxlength=\"255\"></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">Idioma (código ISO 3166)
                     </td>");
```

```
        out.println("<td><input type=\"text\" name=\"nameLang\" size=\"2\"
                    maxlength=\"2\"></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">Texto de la descripción
                    </td>");
        out.println("<td><input type=\"text\" name=\"description\" size=\"50\"
                    maxlength=\"255\"></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">Idioma (código ISO 3166)
                    </td>");
        out.println("<td><input type=\"text\" name=\"descriptionLang\" size=\"2\"
                    maxlength=\"2\"></td></tr>");

        // Categoria
        out.println("<tr id=\"header\"><td colspan=\"2\">");
        out.println("<formSectionTitle>Categoría</formSectionTitle></td></tr>");
        out.println("<tr><td align=\"right\">Categoría</td>");
        out.println("<td><select name=\"category\">");
        out.println("<option value=\"None\">Elija una");
        out.println("<option value=\"UDDITypes\">UDDI Types Taxonomy");
        out.println("<option value=\"GeneralKeywords\">General Keywords");
        out.println("<option value=\"NAICS\">NAICS (1997)");
        out.println("<option value=\"UNSPSC\">UNSPSC");
        out.println("<option value=\"UNSPSC31\">UNSPSC (Versión 3.1)");
        out.println("<option value=\"ISO3166\">ISO 3166 (GEO)");
        out.println("<option value=\"OwningBusiness\">OwningBusiness");
        out.println("<option value=\"Operators\">Operators");
        out.println("</select></td>");
        out.println("<tr><td width=\"20%\" align=\"right\">Nombre </td>");
        out.println("<td><input type=\"text\" name=\"catName\" size=\"50\"
                    maxlength=\"255\"></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">Valor </td>");
        out.println("<td><input type=\"text\" name=\"catValue\" size=\"50\"
                    maxlength=\"255\"></td></tr>");

        // Botón de asentimiento
        out.println("<tr><td colspan=\"2\"><input type=\"submit\"
                    value=\"OK\"></td></tr>");
        out.println("</form></table>");
    }

    void procesar(PrintWriter out) {

        out.println("<h1>ATENEA: Save Business Service</h1>");

        // Mecanismo de seguridad anti-SQLInjection
        // Devuelve false si hay algún tipo de amenaza
        if (!comprobacionSeguridad()) {
            error(out,"Por favor no intente inyectar SQL");
            return;
        }

        // Creamos un nuevo objeto businessService  y los objetos auxiliares
        service = new BusinessService();
        service.setBusinessKey(businessKey);

        // Guardamos la serviceKey, si existe
        if (existe(serviceKey))
            service.setServiceKey(serviceKey);

        // Guardamos el elemento Name
        if (existe(nameValue)) {
            Name name = new Name();
            name.setNameValue(nameValue);
            if (existe(nameLang))
                name.setLangCode(nameLang);
            else
                name.setLangCode("es"); // En español por defecto
            service.addName(name);
        }

        // Guardamos la descripción
        if (existe(description)) {
            Description desc = new Description();
            desc.setDescription(description);
            if (existe(descriptionLang))
                desc.setLanguageCode(descriptionLang);
            else
                desc.setLanguageCode("es"); // En español por defecto
            service.addDescription(desc);
        }

        // Guardamos la categoria
        if (existe(category) && !category.equals("None")) {
```

```
            KeyedReference keyedRef = new KeyedReference();
            if (category.equals("UDDITypes"))
                keyedRef.setTModelKey(TModel.TYPES_TMODEL_KEY);
            if (category.equals("GeneralKeywords"))
                keyedRef.setTModelKey(TModel.GENERAL_KEYWORDS_TMODEL_KEY);
            if (category.equals("NAICS"))
                keyedRef.setTModelKey(TModel.NAICS_TMODEL_KEY);
            if (category.equals("UNSPSC"))
                keyedRef.setTModelKey(TModel.UNSPSC_TMODEL_KEY);
            if (category.equals("UNSPSC31"))
                keyedRef.setTModelKey(TModel.UNSPSC_31_TMODEL_KEY);
            if (category.equals("ISO3166"))
                keyedRef.setTModelKey(TModel.ISO_CH_TMODEL_KEY);
            if (category.equals("OwningBusiness"))
                keyedRef.setTModelKey(TModel.OWNING_BUSINESS_TMODEL_KEY);
            if (category.equals("Operators"))
                keyedRef.setTModelKey(TModel.OPERATORS_TMODEL_KEY);
            if (existe(catName))
                keyedRef.setKeyName(catName);
            if (existe(catValue))
                keyedRef.setKeyValue(catValue);
            service.addCategory(keyedRef);
        }


        // Una vez creado el objeto con todos los parámetros introducidos por el
        // usuario podemos llamar al registro para que haga la inserción
        try {
            reg = new Registro();
            reg.iniciar();
            if (reg.disponible()){
                Vector serviceVector = new Vector(1);
                serviceVector.add(service);
                reg.save_service(authInfo, serviceVector);
                mensaje(out,"Operación concluida con éxito");
            }
            reg.apagar();
        } catch (UDDIException e){error(out,"Excepción: "+ e.getMessage());}
    }


    /** Mecanismo básico que parsea cada campo del formulario para que no se
     *  inyecte código malicioso al servidor
     */

    boolean comprobacionSeguridad() {
        boolean todoOK = true;
        Vector campos = new Vector();
        campos.add(businessKey);
        campos.add(serviceKey);
        campos.add(nameValue);
        campos.add(nameLang);
        campos.add(description);
        campos.add(descriptionLang);
        campos.add(category);
        campos.add(catName);
        campos.add(catValue);
        todoOK = !compruebaSQLInjection(campos);
        return todoOK;
    }

}
```

Lo primero que hace este servlet es extraer la variable de sesión con el token, para asegurarse de que aún es válido. Si no existe o ha caducado se muestra un mensaje de error. En caso contrario el servlet continúa con su funcionamiento normal.

Este servlet funciona de manera análoga a los anteriores: cuando se invoca por primera vez, esto es, sin parámetros, muestra un formulario de entrada de datos como el de la figura 3.26. El usuario utiliza los campos de entrada de texto y los selectores para establecer los datos del nuevo BusinessService y a continuación pulsa el botón “OK”. En ese momento el formulario envía los datos usando GET al mismo servlet SaveService. Cuando el servlet recibe estos datos comprueba que existen los mínimos necesarios para realizar la petición de guardado de datos al Registro. Si está todo

correcto se procesan los datos, en caso contrario se vuelve a mostrar el formulario para que el usuario introduzca unos nuevos criterios.

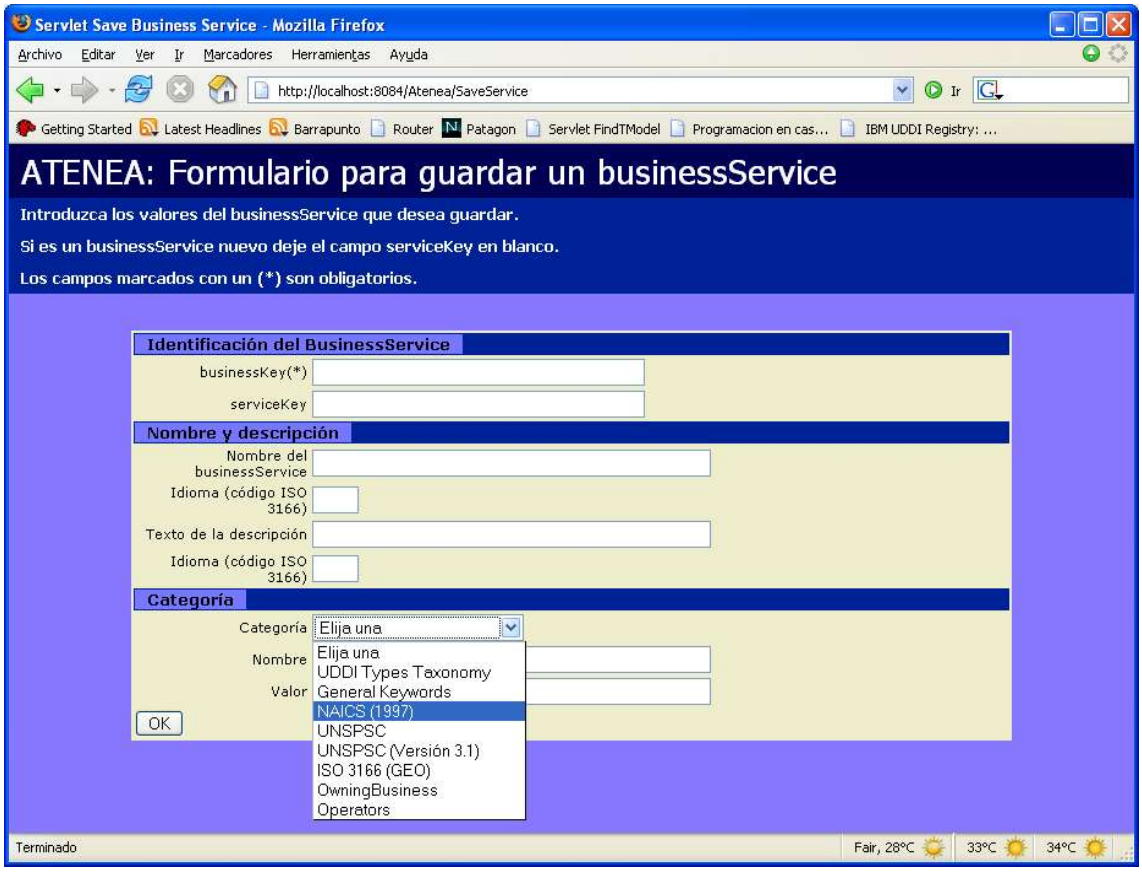

Figura 3.26. Formulario de SaveService

En el método que procesa los datos, y en función de los campos que hayamos rellenado en el formulario, se crearán los elementos del mensaje SaveService. Antes de llamar a la base de datos se llamará al método que identifica posibles intentos de inyección SQL y si es así se generará un mensaje de error advirtiendo de ello al usuario. Por último se instancia la clase "atenea.Registro" y se llama a su método "save_service" con los elementos recién creados. Si se produce algún tipo de excepción se mostrará un mensaje de error al usuario, en caso contrario se le indicará que la operación ha concluido con éxito.

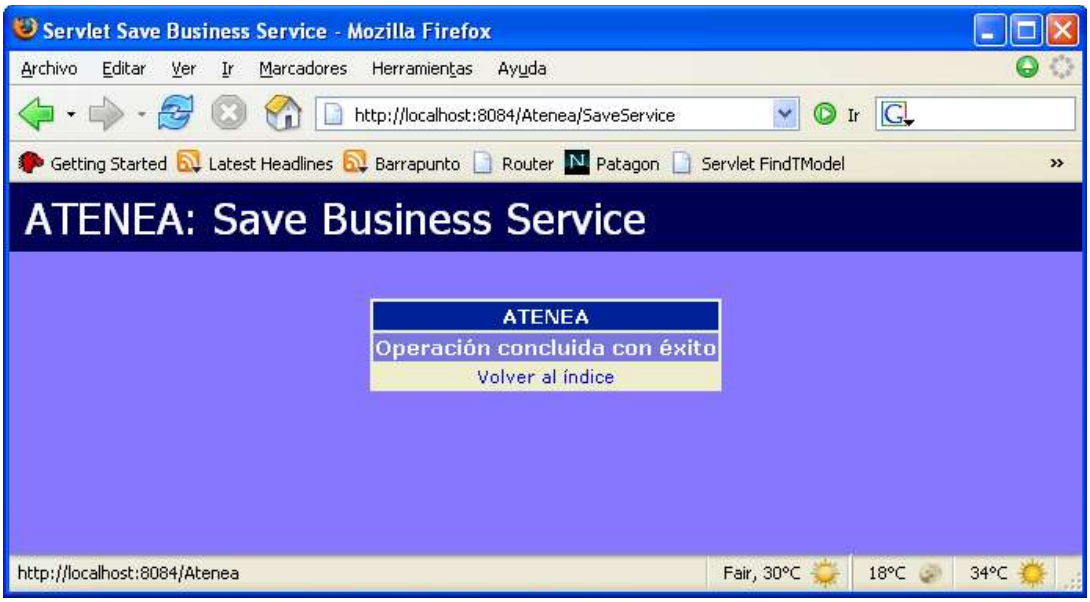

Figura 3.27. Operación concluida con éxito

Por ejemplo, si hubiéramos intentado registrar un BusinessService referenciando la UUID de una BusinessEntity inexistente, el Registro nos habría dado el error de la figura 3.28.

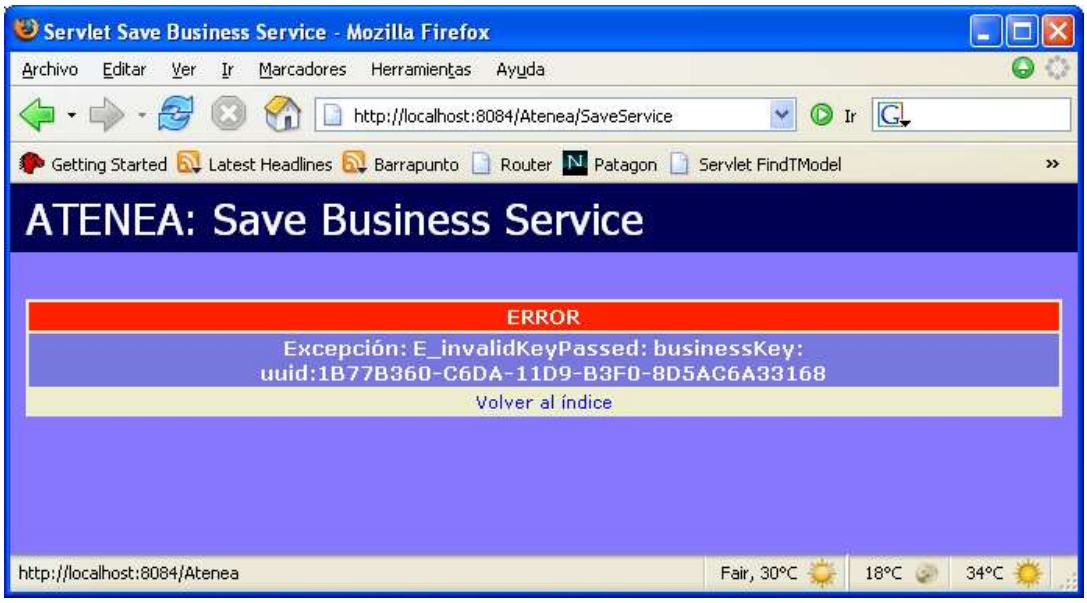

Figura 3.28. Mensaje de error

3.2.3. Interfaz de configuración del Registro

Como comentamos en el apartado 3.1.10 se ha creído conveniente dotar a la aplicación de un mecanismo sencillo para establecer los parámetros de configuración del mismo sin tener que volver a compilar el código fuente. En dicho apartado se describieron las clases y el mecanismo interno que permite que esto sea así. Básicamente lo que se hace es leer estos parámetros desde una tabla de la base de datos. En este apartado crearemos una interfaz para que el usuario final pueda establecer estos valores en dicha tabla de la base de datos de una manera cómoda y sencilla.

Para ello se ha escrito la clase "atenea.Conf", un servlet cuyo código se muestra a continuación:

Clase atenea.Conf:

```
/*
 * Conf.java
 *
 * Created on 2 de mayo de 2005, 21:42
 */

package atenea;

import atenea.basedatos.BaseDatos;
import atenea.error.UDDIException;
import atenea.tipos.Parametros;
import java.io.*;
import java.sql.SQLException;

import javax.servlet.*;
import javax.servlet.http.*;

/**
 * Servlet para establecer los parámetros de configuración del Registro
 * en la tabla CONFIGURACION de la base de datos.
 */
public class Conf extends Interfaz {

    String operator, operator_url, usuarios_path, max_name_elements, time_out;

    /** Processes requests for both HTTP <code>GET</code> and <code>POST</code> methods.
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest(HttpServletRequest request, HttpServletResponse
                                  response) throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        cabecera(out, "ATENEA: Configuración del Registro");

        extraerParametros(request);
        if (!existe(operator) ||
                !existe(operator_url) ||
                !existe(usuarios_path) ||
                !existe(max_name_elements) ||
                !existe(time_out))
            mostrarFormulario(out);
        else
            procesar(out);

        pie(out);
        out.close();
    }



    void extraerParametros(HttpServletRequest request) {
        operator = request.getParameter("operator");
        operator_url = request.getParameter("operator_url");
```

```
        usuarios_path = request.getParameter("usuarios_path");
        max_name_elements = request.getParameter("max_name_elements");
        time_out = request.getParameter("time_out");
    }

    void mostrarFormulario(PrintWriter out) {
        out.println("<h1>ATENEA: Configuración del Registro</h1>");
        out.println("<h3>Introduzca los parámetros de configuración</h3>");

        out.println("<table width=\"80%\" align=\"center\">");
        out.println("<form method=\"POST\" action=\"/Atenea/Conf\">");

        out.println("<tr id=\"header\"><td colspan=\"2\">");
        out.println("<formSectionTitle>Nombre y URL</formSectionTitle></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">Nombre del Operador</td>");
        out.println("<td><input type=\"text\" name=\"operator\" size=\"50\"
                    maxlength=\"255\" value=\"Atenea\"></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">URL</td>");
        out.println("<td><input type=\"text\" name=\"operator_url\" size=\"50\"
                  maxlength=\"255\" value=\"http://localhost:8084/Atenea\"></td></tr>");

        out.println("<tr id=\"header\"><td colspan=\"2\">");
        out.println("<formSectionTitle>Fichero de Autenticación de
                    Usuarios</formSectionTitle></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">Path completo</td>");
        out.println("<td><input type=\"text\" name=\"usuarios_path\" size=\"50\"
                  maxlength=\"255\" value=\"c:\\proyecto\\usuarios.xml\"></td></tr>");

        out.println("<tr id=\"header\"><td colspan=\"2\">");
        out.println("<formSectionTitle>Parámetros
                     adicionales</formSectionTitle></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">Máxima cantidad de elementos
                    Name en búsquedas</td>");
        out.println("<td><input type=\"text\" name=\"max_name_elements\" size=\"2\"
                    maxlength=\"2\" value=\"5\"></td></tr>");
        out.println("<tr><td width=\"20%\" align=\"right\">Caducidad de los tokens de
                    autenticación (segundos)</td>");
        out.println("<td><input type=\"text\" name=\"time_out\" size=\"4\"
                    maxlength=\"4\" value=\"3600\"></td></tr>");


        out.println("<tr><td colspan=\"2\"><input type=\"submit\"
                    value=\"OK\"></td></tr>");
        out.println("</form></table>");
    }

    void procesar(PrintWriter out){

        BaseDatos bd = null;
        int max_name_elements_int = Integer.parseInt(max_name_elements);
        int time_out_int = Integer.parseInt(time_out);
        Parametros param = new Parametros(operator, operator_url, usuarios_path,
                                          max_name_elements_int, time_out_int);

        try {
            bd = new BaseDatos();
            bd.guardaParamReg(param);
            mensaje(out,"Operación concluida con éxito");
        } catch (SQLException e) { error(out,"Excepción: "+e.getMessage());
        } catch (UDDIException e) { error(out,"Excepción: "+e.getMessage());}
    }

}
```

La primera vez que se llama a este servlet, muestra el formulario de la figura 3.29 para que el administrador de la aplicación establezca los valores que considere oportunos. Cuando el usuario pulsa el botón “OK” el servlet se llama a sí mismo con estos parámetros en un método http GET. Si están todos los parámetros necesarios se procesan, en caso contrario se vuelve a mostrar el formulario.

Figura 3.29. Interfaz de configuración

El proceso de los parámetros simplemente consiste en empaquetarlos en un objeto "atenea.tipos.Parametros", instanciar la clase "atenea.basedatos.BaseDatos" e invocar su método "guardaParamReg" para que almacene estos datos en la tabla CONFIGURACION de la base de datos. Si se produce algún tipo de excepción se mostrará un mensaje de error al usuario, en caso contrario se le indicará que la operación ha concluido con éxito.

Como se indicará en el manual de instalación, se recomienda eliminar este servlet una vez utilizado por el administrador para evitar un uso no autorizado.

4. Conclusiones y líneas de continuación

4.1. Conclusiones

Una vez realizado todo el trabajo se constata que se han cumplido todos los objetivos planteados en la introducción: se han implementado las APIs de la especificación UDDI, tanto la de consulta, de acceso libre, como la de publicación, de acceso restringido.

Para ello se ha diseñado una base de datos para obtener respaldo de los datos almacenados en el Registro UDDI y se han programado en Java las clases correspondientes que implementan cada uno de los métodos de las APIs. Dado que no se iba a dar soporte para mensajería XML, se ha creado una interfaz web para que el usuario final pueda utilizar el Registro de una manera cómoda y sencilla.

Además se ha creado un sistema de configuración para que el usuario pueda modificar ciertos parámetros del Registro y se ha implementado un sistema de limpieza automática de los datos obsoletos que van quedando en la base de datos.

4.2. Líneas de continuación

La amplicación más obvia para este Proyecto es el de implementar un procesador de mensajes SOAP. Dicho procesador recibiría peticiones en forma de documentos XML, los convertiría a objetos Java como los utilizados en este proyecto y llamaría al método de la API correspondiente. Dicho método devolvería un objeto al procesador SOAP, que lo convertiría de nuevo en un documento XML para mandarlo de vuelta al usuario.

Otras líneas de continuación menos importantes podrían ser las siguientes:

- Usar los mensajes ya definidos SavePublisher, GetPublisherDetail, etc. e implementar los métodos correspondientes que permitan a un Administrador gestionar las cuentas Publisher desde dentro de Atenea, en lugar de modificar a mano la base de datos.

- Hacer configurable el login y password de la base de datos Atenea. En este Proyecto se ha utilizado el par “atenea/atenea” pero sería deseable que pudiera usarse el que se quisiera modificando algún fichero de configuración.

- Encriptar el fichero de autenticación de usuarios de modo que no fuera obvio su contenido si algún intruso logra dar con él. Otra opción sería meter estos datos directamente en alguna tabla de la base de datos.

5. Manual de instalación

La presente sección pretende ser una guía para el usuario final de forma que pueda poner en funcionamiento la aplicación desarrollada en el Proyecto.

5.1. Preparación del entorno

Para que nuestra aplicación pueda funcionar necesita que las siguientes plataformas y programas estén instalados en el sistema:

- **Java Runtime Environment**

  Este es el entorno de ejecución de Java (JRE) que nos permitirá ejecutar aplicaciones Java como la aplicación que hemos desarrollado. Vamos a la siguiente web y descargamos el instalador:

  http://java.com/es/

  Ejecutamos el programa que hemos bajado, elegimos una instalación típica y en pocos minutos tendremos la máquina virtual Java instalada en nuestro sistema.

- **Jakarta Tomcat**

  Necesitamos un contenedor de servlets/jsp en el que desplegar nuestra aplicación. Podríamos haber elegido otro pero usamos éste por ser de código abierto tal y como se requería en los objetivos del Proyecto. Vamos a la siguiente web y descargamos el instalador:

  http://jakarta.apache.org/site/downloads/downloads_tomcat-5.cgi

  Para instalarlo seguimos las instrucciones ya descritas en el apartado 3.1.1.3.

- **MySQL**

  Por último necesitamos un gestor de base de datos de código abierto y lo suficientemente estable. Hemos elegido MySQL ya que cumple con todos los requisitos. Podemos bajar el instalador en la siguiente dirección web:

  http://dev.mysql.com/downloads/index.html

  Para instalarlo seguimos las instrucciones ya descritas en el apartado 3.1.1.4.

5.2. Estructura de la distribución

Junto con la presente memoria se adjunta un CD con la aplicación, el código fuente y todos los ficheros de configuración necesarios. El directorio base es PFC-DRR, que contiene la estructura que se muestra en la figura 5.1.

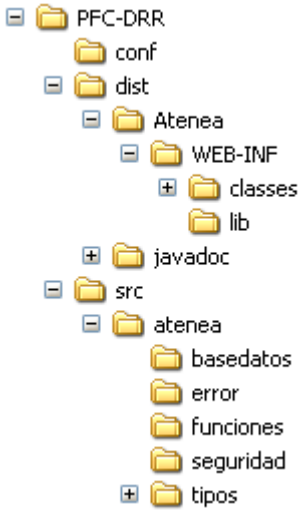

Figura 5.1. Estructura de directorios de la distribución

- **conf**: Directorio con el fichero de autenticación de usuarios y con los scripts sql para generar la base de datos.

- **dist**: Directorio con los archivos de distribución de la aplicación. Contiene el directorio "Atenea" con la aplicación web y el directorio "javadoc" con la documentación electrónica.

- **src**: Directorio con el código fuente del Proyecto

5.3. Configuración de la base de datos

Durante la instalación de MySQL creamos un usuario llamado "atenea" que podía entrar usando la contraseña "atenea". Es éste el usuario que utilizará nuestra aplicación para comunicarse con la base de datos.

Vamos ahora a crear la base de datos "atenea" y a darle todos los permisos a dicho usuario. Para ello copiamos los archivos *.sql del directorio "conf" en el directorio que queramos, abrimos una consola, nos posicionamos en dicho directorio y escribimos la siguiente sentencia:

```
mysql –u root –p<password> < crea_bd.sql
```

Donde <password> debe sustituirse por la contraseña de administrador que hayamos elegido al instalar MySQL.

A continuación crearemos las tablas:

```
mysql –u root –p<password> < crea_tablas.sql
```

Las inicializamos introduciendo los valores de los tModels del núcleo de UDDI:

```
mysql –u root –p<password> < inicializa_tablas.sql
```

Y por último introducimos los parámetros de configuración por defecto.

```
mysql –u root –p<password> < configura.sql
```

5.4. Fichero de autenticación de usuarios

En el directorio "conf" también hay otro fichero, llamado "usuarios.xml" y que lista los usuarios autorizados para publicar con nuestra aplicación. Es el momento de editarlo para establecer dicha lista.

Su estructura es muy sencilla, se compone de un elemento raíz "usuarios" del que cuelgan tantos elementos "usuario como se quiera. Cada uno de ellos tiene tres atributos: login, password y publisher.

Fichero usuarios.xml de ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
    Document   : usuarios.xml
    Created on : 14 de abril de 2005, 23:33
    Author     : David
    Description:
        Lista de usuarios autorizados para publicar en Atenea.
-->

<usuarios>
    <usuario login="david" password="p1ba2r" publisher="Atenea"/>
    <usuario login="pepe" password="pepe" publisher="pepe"/>
    <usuario login="antonio" password="antonio" publisher="Atenea"/>
</usuarios>
```

El atributo "login" se corresponde con "userID" en la nomenclatura UDDI, "password" sería el "cred" y "publisher" el identificador del Publisher. Como vemos, distintos uaurios pueden utilizar el mismo Publisher.

Si se indica un "Publisher" que no exista en la base de datos de Atenea, dicho usuario quedará invalidado. Podemos crear nuevos Publisher insertando los datos adecuados en la Tabla Publisher. Por ejemplo, para crear el Publisher "pepe", de nombre "Publisher Pepe", cuyo contacto fuera "Pepe Gonzalez" y sin privilegios de ninguna clase, habría que entrar al gestor de base de datos y escribir lo siguiente:

```
USE ATENEA;

INSERT INTO PUBLISHER
(PUB_ID, PUB_NAME, LAST_NAME, FIRST_NAME, ADMIN, AUTHORIZED)
VALUES ('pepe', 'Publisher Pepe', 'Gonzalez', 'Pepe', 'FALSE', 'FALSE');
```

Una vez creados los Publishers y editado el fichero de configuración de usuarios, se deberá ubicar en algún directorio fuera del alcance de ataques externos. Por ejemplo, sería muy mala idea dejarlo en el directorio raiz de la aplicación.

5.5. Despliegue de la aplicación

Este paso consiste únicamente en copiar el directorio "Atenea" del directorio "dist" en el directorio "webapps" de Tomcat. Si el servidor de aplicaciones estaba en funcionamiento, habrá que apagarlo y volver a encenderlo para que se de cuenta de que hemos desplegado una nueva aplicación.

5.6. Configuración del Registro

En este último paso daremos los últimos retoques a la instalación. Tan sólo habrá que abrir un navegador de Internet y escribir lo siguiente en la barra de direcciones:

http://localhost:8080/Atenea/Conf

Donde se ha supuesto que el servidor de aplicaciones Tomcat está utilizando el puerto 8080, que es lo más habitual.

Aparecerá el formulario de configuración del Registro, tal y como se describió en el apartado 3.2.3. Los campos más importantes a rellenar en este apartado son el de la URL y el del path del fichero de autenticación.

En la URL escribiremos dónde hemos instalado la aplicación. Si queremos que los elementos DiscoveryURL generados por la aplicación sean útiles para máquinas distintas a la que está ejecutando la aplicación, esta URL debe ser una dirección de red válida en el entorno en el que estemos trabajando. Es decir, si ponemos como URL la dirección por defecto:

http://localhost:8080/Atenea

esta dirección sólo será válida para nuestra máquina. Es aconsejable escribir una dirección de red, si se dispone de ella, que pueda utilizarse desde máquinas externas.

El otro campo importante es el de la ubicación del fichero de autenticación. Antes comentábamos que debía colocarse en alguna carpeta fuera del alcance de intrusos. Es en este paso en el que indicamos al Registro dónde buscar dicho fichero.

Una vez terminado de configurar el Registro, es conveniente borrar o cambiar el nombre de la clase "Conf.class" del directorio "webapps\Atenea\WEB-INF\classes\atenea" de Tomcat. De esta manera evitamos un uso no autorizado del mismo.

5.7. Ejecución de la aplicación

Una vez instalada y configurada la aplicación, basta con abrir un navegador de Internet y escribir lo siguiente en la barra de direcciones:

http://localhost:8080/Atenea

Para acceder desde una ubicación remota bastaría con sustuir “localhost” por la dirección de red del ordenador que esté sirviendo de host para la aplicación. Aparecerá la página de inicio de la figura 3.18 y podremos comenzar a buscar y publicar datos en el Registro.

6. Bibliografía

6.1. Libros

- "**Programación con XML**". Ricardo Eito Brun. Anaya Multimedia.
- "**Web Services Essentials**". Ethan Cerami. O'Reilly
- "**Programming Web Services with SOAP**". Pavel Kulchenko, James Snell, Doug Todwell. O'Reilly.
- "**Understanding Web Services: XML, WSDL, SOAP, and UDDI".** Eric Newcomer. Addison Wesley.
- "**Java Servlet Programming**". Jason Hunter. O'Reilly.
- **"Managing and Using MySQL, 2nd Edition**" Tim king, George Reese, Randy Jay Yarger.

6.2. Apuntes

- Asignatura "Bases de Datos".

6.3. Recursos de Internet

- Definición de servicio web

  http://es.wikipedia.org/wiki/Servicios_Web

- Introducción a la plataforma de servicios web

  http://web-services.bankhacker.com/

- Introducción a UDDI

  http://www.fisica.uson.mx/carlos/WebServices/WS_UDDI.htm

- Especificaciones de UDDI y documentos relacionados

  http://www.uddi.org/specification.html

- Tutorial de Servlets, variables de sesión y JSP

  http://www.programacion.com/java/tutorial/servlets_jsp/10/

- APIs de Java para XML

  http://www.programacion.com/java/tutorial/apis_xml/2/

- Acceso a bases de datos desde Java (JDBC)

http://www.programacion.com/java/tutorial/jdbc/

- Documentación de Tomcat

  http://jakarta.apache.org/tomcat/tomcat-5.5-doc/index.html

- Documentación de MySQL

  http://dev.mysql.com/doc/