Universidad Carlos III de Madrid

Ingeniería en Telecomunicación

Proyecto Final de Carrera

Federación de bases de datos mediante ontologías: Antive

Autor: David Muñoz Díaz
Tutora: Damaris Fuentes Lorenzo

Resumen

Internet es la principal fuente de información actualmente. El crecimiento vertiginoso en el número de entidades que operan en ella ha dado lugar a una gran cantidad de información disponible. Sin embargo, al operar de forma independiente, la información se encuentra repartida en múltiples bases de datos sin cooperación ni cohesión. Incluso una entidad, al crecer, puede necesitar la integración de bases de datos con información relacionada, que operan de forma independiente.

Una forma de conseguir la integración es considerar el concepto de *recurso*: un conjunto de información relacionada con una entidad. Personas, documentos y otras cosas se pueden representar como recursos con nombres, títulos y otras propiedades.

La Web Semántica se basa en el concepto de recurso: una página web semántica incluye información de los recursos sobre los que versa el documento. Esta *meta-infomación* puede usarse para obtener relaciones entre diferentes recursos, de forma que, paralelamente a los documentos, exista una gran colección de recursos y relaciones que den semántica a los contenidos. Para determinar los tipos de recursos y las propiedades que poseen, la Web Semántica emplea *ontologías*: descripciones formales de los conceptos involucrados en un dominio de conocimiento.

En este proyecto se plantea el uso de ontologías y otras tecnologías de la Web Semántica como base para la federación de bases de datos. Se estudia la conveniencia de este enfoque y las ventajas que aporta respecto a las herramientas de federación de bases de datos existentes. Para probar la utilidad de estas tecnologías se ha realizado una aplicación, Antive, que permite llevar a cabo el objetivo del proyecto.

Abstract

Internet is the main information source nowadays. The great growth in the number of organizations operating on it has lead to a huge amount of information available. However, as the organizations operate with independence, all the information available is distributed in multiple databases with neither cooperation nor coherence. Even in a single organization, when growing, several databases operating independently with related information may need to be integrated.

One way to achieve this integration is to consider the concept of *resource*; that is, a collection of information related to a single entity. People, documents and other things can be represented as resources with names, titles and other properties.

The Semantic Web is based on the concept of resource: a semantic web page includes information about the resources the page is about. This *meta-information* can be used to determine relations between different resources, in a way that, parallelly to the documents, there exist a great collection of resources and relations that give semantic to the contents. In order to determine the types of resources and the properties they have, the Semantic Web uses a language to define *ontologies*: formal descriptions of the concepts involved in a domain of knowledge.

In this project, the use of ontologies and other technologies from the Semantic Web is considered as the basis for the database federation. The convenience of this approach and the advantages offered in contrast to existing database federation tools is studied. In order to test these technologies' convenience, an application, Antive, is developed to reach the objective of this project.

A mis padres

Índice general

Índice de figuras

Índice de listados

Lista de acrónimos

AJAX	Asynchronous Javascript And XML
ANSI	American National Standards Institute
API	Application Programming Interface
BBDD	Base(s) de datos
CGI	Common Gateway Interface
COC	Convention Over Configuration
CSS	Cascade Style Sheet
DCMI	Dublin Core Metadata Initiative
DISCO	Distributed Information Search COmponent
DOM	Document Object Model
DOME	Domain Ontology Management Environment
DRY	Don't Repeat Yourself
FastCGI	Fast Common Gateway Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
J2EE	Java 2 Enterprise Edition
JS	JavaScript
JSON	Javascript Simple Object Notation
JSP	Java Server Pages
KRAFT	Knowledge Reuse And Fusion/Transformation
LDAP	Lightweight Directory Access Protocol
MVC	Model-View-Controller
OBSERVER	Ontology Based System Enhanced with Relationship for Vocabulary hEterogeneity Resolution
ODL	Object Definition Language
ODMG	Object Database Management Group
OQL	Object Query Language
OWL	Ontology Web Language
PHP	Personal Home Page
RDF	Resource Description Framework
RDFS	RDF Schema
REST	REpresentational State Transfer

SGBD	Sistema de Gestión de Bases de Datos
SIMS	Single Interface for Multiple Sources
SIMS II	Single Interface for Multiple Sources of Incomplete Information
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
TCP	Transmission Control Protocol
TSIMMIS	The Stanford-Ibm Manager of Multiple Information Sources
UML	Unified Modelling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XHTML	XML HyperText Markup Language
XML	eXtensible Markup Language
XML-RPC	XML Remote Procedure Call

Capítulo 1

Introducción

En este capítulo se presentan al lector los factores que motivan la realización de este proyecto, ofreciendo una descripción general del problema que se plantea. Se realizará un análisis muy general del proyecto software que se plantea y en el que, a lo largo de los siguientes capítulos, se profundizará para acompañar al lector durante todo el ciclo de desarrollo del proyecto.

1.1. Descripción del problema

Internet es la principal fuente de información del mundo actual. El crecimiento que ha experimentado en los últimos años y el aumento de la presencia en él de empresas, universidades, organismos estatales y otras entidades, ha supuesto la incorporación de grandes cantidades de datos accesibles desde la red de redes.

Internet se fundamenta en la transparencia de las comunicaciones. Dos nodos conectados a la red pueden intercambiar información sin necesidad de considerar los sistemas que median en el flujo de datos. Esta *abstracción* de la realidad física de las comunicaciones es lo que ha llevado a Internet a ser la herramienta fundamental para el intercambio de conocimiento que es hoy en día.

Una vez que se ha conseguido la abstracción a nivel de la *transmisión* de la información, el siguiente paso consiste en la abstracción a nivel superior, que es el de la *representación* de la información. Actualmente no se puede afirmar que se haya alcanzado el éxito en este sentido. Esto se debe a que la mayor parte de las entidades que operan en Internet lo hacen desde el enfoque de los *servicios*: cada entidad almacena *su* información de forma autónoma y permite el acceso a la misma a través de *su* interfaz, ofreciendo las respuestas a las consultas en *su* formato. Aunque exista cierta conformidad con estándares para el acceso, la consulta y la representación de la información (como los lenguajes de consulta y formatos XML), no se puede decir que exista una *abstracción* porque el acceso a la información requiere conocer los detalles sintácticos en cada caso: el lenguaje de consulta, parámetros de conexión, etc.

Por ello, resulta deseable una *cooperación* entre los diferentes sistemas de almacenamiento de

información que permita realizar consultas a los mismos abstrayéndose de los esquemas individuales de cada uno de ellos y ofreciendo al usuario un único esquema global. De esta forma, las entidades que posean grandes bases de datos incoherentes e independientes contarían con un único sistema de consulta que contuviera toda la información disponible.

Sin embargo, la abstracción no se consigue únicamente con esta *federación de bases de datos*, ya que si bien se consigue eliminar la *física* del acceso a datos (máquinas, direcciones, parámetros de conexión, etc.), la información continúa siendo un conjunto de datos sin semántica.

La Web Semántica se presenta como una solución para lograr el nivel más alto de abstracción, añadiendo metadatos semánticos a los resultados provenientes de una consulta, de forma que puedan determinarse las relaciones entre diferentes *recursos*. Se entiende un recurso como algo *suficientemente importante como para ser referido como una cosa por sí misma* [Richardson and Ruby, 2007], o de forma más sencilla, *algo con identidad propia*.

Sin embargo, el avance de la Web Semántica está resultando muy lento. Tan solo una pequeña parte de las empresas y entidades que operan en Internet se ha adherido a esta nueva tecnología, en parte por el impacto que supone incorporar metadatos semánticos a las interfaces de consulta.

1.2. Objetivos

El objetivo de este proyecto es el análisis, diseño e implementación de un sistema que permita federar un conjunto de bases de datos para obtener un esquema global empleando las técnicas de la Web Semántica que ofrezca una interfaz web para realizar consultas. Las ventajas de contar con un sistema así serían múltiples.

En primer lugar, y situándose en el nivel más bajo de abstracción, este sistema se podría usar para federar el contenido de diversas bases de datos de una organización. Así, si una organización posee un conjunto de bases de datos heterogéneas y no cohesionadas, se puede emplear este sistema para localizar *recursos* cuya información se halle distribuida.

En segundo lugar, al emplear técnicas de Web Semántica, el esquema general se especifica en términos de *conceptos* en lugar de *tablas y columnas*. Al realizar una consulta, los resultados se abstraen en *recursos* en lugar de en *registros*, lo que ofrece una representación *semántica* de la información.

En tercer lugar, las consultas también se hacen en términos semánticos, trasladando las *columnas* y *claves ajenas* a *propiedades*.

En cuarto lugar, la federación de bases de datos no tiene que limitarse a las pertenecientes a una única empresa o entidad, sino que se puede generar un esquema que englobe los conceptos relativos a varias entidades y federar sus bases de datos, obteniendo así un conjunto de recursos cuya información sea mucho más completa.

Por último, este sistema fomentaría la adopción de la Web Semántica por parte de las entidades

que lo emplearan.

1.3. Enfoque y metodología

1.3.1. Ciclo de vida

En el desarrollo de este sistema se empleará una metodología de desarrollo muy común en los proyectos software que consiste en el empleo de un ciclo de vida en V. Este ciclo de vida fue propuesto por Alan Davis en [Davis, 1997] y considera diferentes fases distribuidas en tiempo y por niveles de abstracción como se muestra en la figura 1.1.

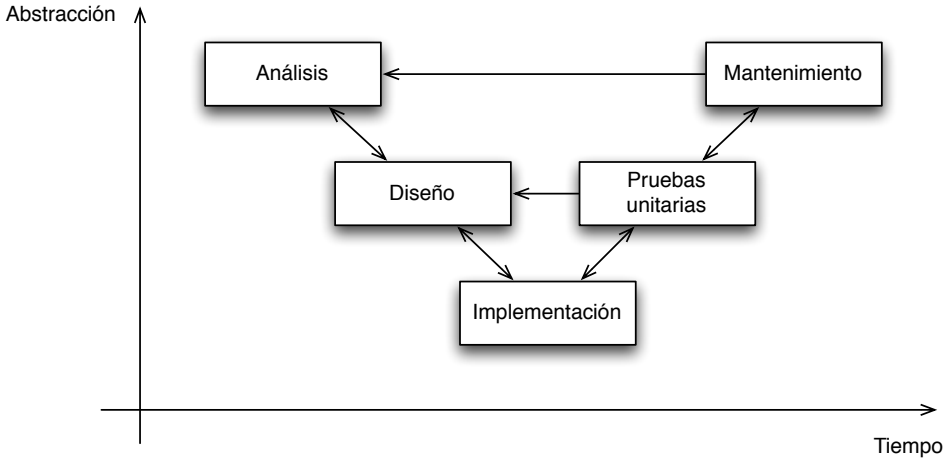

Figura 1.1: Ciclo de vida en V

Ya que los proyectos software son por lo general cambiantes y ofrecen una evolución continua, este ciclo de vida se enmarca dentro del contexto de las diferentes etapas de un *prototipado evolutivo*. Este término hace referencia a un modelo de ciclo de vida en el que el producto final se divide en diferentes prototipos que se realizan de forma secuencial y obteniendo una retroalimentación continua y temprana de los responsables del proyecto.

Por último, se dividirá el proyecto en componentes y se contemplará el desarrollo de cada uno de ellos como un subproyecto, aplicando el ciclo de vida a cada uno de ellos de forma independiente.

1.3.2. Paradigma de modelado

Para el diseño y la implementación del software se empleará principalmente el paradigma de la *programación orientada a objetos* en el que se definen componentes que encapsulan datos y funcionalidades relacionadas entre sí, creando estructuras llamadas *objetos* cuya interacción determina el comportamiento de la aplicación. Este paradigma refleja de cierta forma la percepción humana del mundo real y permite por ello realizar un modelado que resulte natural al programador.

En algunos casos puede resultar apropiado un enfoque *estructurado* en el que la funcionalidad del componente se divide en *métodos* que operan sobre unas estructuras de datos comunes. Este punto de vista puede ser apropiado para componentes en los que se realicen operaciones sobre una misma estructura de datos y en los que la diferenciación por *objetos* no aporte una ventaja significativa.

1.4. Estructura del documento

El resto de este documento se estructura de la siguiente manera:

- El capítulo 2 realiza un estudio del marco en el que se encuentra el proyecto, contemplando algunas soluciones de federación de bases de datos disponibles actualmente.
- El capítulo 3 realizará un análisis de la solución que se desea obtener exponiendo los requisitos que se precisan para alcanzar el éxito del proyecto.
- El capítulo 4 se realizará un diseño arquitectónico del sistema, contemplando los diferentes componentes y capas que se desarrollarán.
- El capítulo 5 versa sobre el desarrollo de la aplicación, los lenguajes y las herramientas empleadas. Debido a la complejidad de la herramienta y a la posibilidad de ser extendida en el futuro por otros desarrolladores, además de contar con un código ampliamente comentado, se ha optado por realizar un análisis del código de la aplicación, que se incluye en este capítulo.
- El capítulo 6 muestra unos ejemplos de uso de la aplicación que permiten conocer las posibilidades de la herramienta.
- El capítulo 7 resumirá las conclusiones que se obtienen de este proyecto y ofrecerá una serie de líneas de trabajo que pueden resultar de interés para futuros desarrollos.
- El apéndice A realiza una introducción al lenguaje Ruby que se ha empleado en la elaboración de gran parte del proyecto como apoyo al futuro desarrollador.
- Los apéndices B y C muestran ejemplos de código de Antive que pueden ser de utilidad para futuras referencias.
- El apéndice D explica el mecanismo para poner a punto el entorno de desarrollo empleado en la realización de este proyecto.

Capítulo 2

Estado del arte

En este capítulo se realizará una revisión del contexto en el que se desarrolla el proyecto, realizando una introducción a los conceptos y tecnologías con los que se relaciona. Se analizarán algunas soluciones existentes en este terreno contemplando sus principales ventajas y desventajas.

2.1. Sistemas federados

Cuando se dispone de un conjunto de bases de datos independientes se puede integrar toda la información disponible en una *federación* de esas distintas bases, formando una única *base de datos federada* o *virtual*. Los esquemas de las bases de datos que participan en la federación se denominan *esquemas locales*, mientras que la base de datos federada ofrece un único *esquema global* al usuario.

El caso más general de un sistema semejante es el de los sistemas federados *fuertemente acoplados* [Sheth and Larson, 1990] [Brisaboa et al., 2002b] en los que los usuarios pueden acceder a la información mediante un lenguaje de consulta que no contemple las características físicas de la federación. En este caso, el *sistema gestor de la base de datos federada* descompone la consulta en sub-consultas para cada una de las bases de datos componentes de la federación y, tras obtener los resultados parciales, los compone para generar la respuesta final.

Por otro lado, existen sistemas federados *débilmente acoplados* en los que no existe esta transparencia de cara a las bases de datos que componen la federación. Por ello se emplean lenguajes de consulta *multi-base de datos* en los que queda reflejada esta falta de transparencia [Grant et al., 1993] [Lakshmanan et al., 2001].

En [Busse et al., 1999] se propone una última distinción para los sistemas de información federados que consiste en los *basados en mediadores*, en los que, además de las bases de datos, se consideran otras fuentes de información que se consultan haciendo uso de *mediadores*, componentes software que adaptan las consultas entre el sistema global y los sistemas componentes. Este enfoque resulta muy interesante actualmente, ya que muchos servicios de Internet permiten realizar consultas a sus bases de datos mediante APIs XML, SOAP, etc.

En este sentido es preciso diferenciar entre *wrappers (traductores* o *envoltorios*), que son componentes software que traducen la interfaz de las diferentes fuentes a una interfaz común, y *mediadores* en sentido estricto, que son componentes que implementan cierta lógica para dividir y distribuir las consultas. Sin embargo, aunque un sistema basado en mediadores se plantea con el objetivo de que la federación final sea fuertemente acoplada, algunas limitaciones de este caso, como la incapacidad de realizar escritura en las fuentes de datos, promueven esta distinción.

2.2. Sistemas de integración

En esta sección se verán algunos proyectos relacionados con la federación de bases de datos cuyo estudio sirve de base para la realización de este proyecto.

Garlic: Garlic[1] es un proyecto desarrollado por IBM Research para el diseño e implementación de sistemas de información multimedia basados en bases de datos heterogéneas[Haas et al., 1997] [Carey et al., 1995]. Garlic ofrece un enfoque orientado a objetos, de forma que el usuario accede a la información disponible en los distintos repositorios empleando la abstracción de recursos. Se podría clasificar como un sistema basado en mediadores, ya que su funcionamiento se basa en *envolver* cada base de datos con mediadores controlados por un sistema central.

La definición de los recursos (*objetos* en la nomenclatura de la herramienta) se realiza mediante un lenguaje sencillo, semejante a la definición de interfaces Java. Sin embargo, además de esta definición, el usuario de la herramienta debe aportar una *implementación* de cada objeto que determine el comportamiento del mismo en términos de las consultas que se deben realizar para recuperar toda la información relacionada. Esta implementación se realiza en un lenguaje propio de Garlic semejante a SQL.

Este enfoque ofrece una gran flexibilidad, ya la implementación de cada tipo de objeto puede resultar tan compleja como el lenguaje lo permita. Sin embargo, para una federación basada en correspondencias entre los esquemas locales y el global, esta solución resulta demasiado compleja.

TSIMMIS: También desarrollado por IBM, TSMIMMIS [Garcia-molina et al., 1997] [Chawathe et al., 1994] se centra en el empleo de un conjunto de mediadores (en sentido estricto) distribuidos de forma jerárquica, de manera que la consulta se invoca sobre un mediador, que la distribuye a otro conjunto de mediadores de un nivel inferior, etc. La última capa de mediadores interactúa directamente con los envoltorios de las bases de datos.

De esta manera, TSIMMIS implementa la federación de las bases de datos mediante el funcionamiento *distribuido* de un conjunto de mediadores. De forma semejante a Garlic, TSIMMIS emplea varios lenguajes para definir los envoltorios de las fuentes de datos, para realizar las consultas a la federación y para definir el comportamiento de los mediadores.

[1] http://www.almaden.ibm.com/cs/garlic/

El principal inconveniente de TSIMMIS es que es necesario implementar el comportamiento de los mediadores, y esta implementación puede llegar a ser muy compleja.

DISCO: La arquitectura de DISCO [Tomasic et al., 1995] [Tomasic et al., 1997] es prácticamente idéntica a la de TSIMMIS, empleando una jerarquía de mediadores que cooperan para lograr la integración de las bases de datos. Por otro lado, de forma semejante a Garlic, DISCO emplea la orientación a objetos para modelar los parámetros de las consultas y los resultados obtenidos.

Como ventaja sobre ambos sistemas, DISCO se basa en el estándar ODMG tanto para la descripción de los tipos de objetos (empleado el lenguaje ODL) como para la realización de consultas (mediante el lenguaje OQL). Estos lenguajes resultan más sencillos de emplear que los utilizados en los dos proyectos anteriores.

Sin embargo, DISCO es un sistema de federación *débilmente acoplado* ya que en las consultas que se ejecutan se hace explícito el conjunto de fuentes de datos que se debe consultar, por lo que el sistema no aporta la transparencia al usuario que se pretende en este proyecto.

Actualmente, este proyecto parece detenido.

InfoMaster: De igual manera que los sistemas anteriores, InfoMaster [Genesereth et al., 1997] envuelve las diferentes fuentes de datos con *wrappers* pero no emplea un conjunto de mediadores como en el caso de DISCO o TSIMMIS, sino que se utiliza un único componente que centraliza el análisis de la consulta del usuario, la ejecución de las subconsultas y la federación de los resultados.

Para lograrlo, InfoMaster emplea un conjunto de reglas que relacionan los esquemas locales de las bases de datos componentes con el esquema global de la federación. Además, el lenguaje empleado para definir estas reglas permite realizar transformaciones en los datos de las fuentes para ser representados en el esquema global (por ejemplo, transformaciones en unidades, moneda, etc.).

InfoMaster se centra en la generación de *catálogos* a partir de las bases de datos componentes considerando que estas son independientes, por lo que no contempla la posibilidad de que la información de un único recurso se distribuya en diversas fuentes.

SIMS: [Arens et al., 1996] Es un sistema de consulta a fuentes de datos heterogéneas que aporta como novedad que en lugar de un *esquema global* considera un *modelo del dominio de la aplicación*, en el que se describen objetos, relaciones, etc. Tras ello, para cada fuente de datos, se describe su modelo de datos y la forma en que se relaciona con el dominio de la aplicación. Estas descripciones se construyen empleando una plataforma de representación de conocimiento llamada Loom.

Loom[2] es un sistema que permite modelar un dominio de conocimiento y, a partir de este modelo, proporcionar un soporte para lógica deductiva y un mecanismo de consulta. Así pues, la

[2]http://www.isi.edu/isd/LOOM/LOOM-HOME.html

novedad que aporta SIMS respecto al resto de proyectos consiste en el empleo de una *descripción del dominio de la aplicación* de carácter *semántico*.

En 2005 el desarrollo de Loom se detuvo para iniciarse una nueva versión de la plataforma llamada PowerLoom[3] cuyo desarrollo parece activo actualmente. Por otro lado, en 1998, SIMS dio paso a una revisión llamada SIMS II. Actualmente el proyecto parece descontinuado.

Los proyectos aquí expuestos son una muestra representativa de las aproximaciones existentes al terreno de la integración de fuentes de datos. El caso básico consiste en ofrecer un esquema global de tipo relacional, en el que existan tablas cuyo contenido se obtenga de las bases de datos componentes. Por otro lado, la orientación a objetos ofrece ya la abstracción de recursos y propiedades. Generalizando este concepto se llega a la necesidad de modelar el dominio de la aplicación, describiendo objetos, relaciones entre ellos, etc.

Este enfoque resulta muy interesante, y es el que se persigue en este proyecto. Para ello es necesario contar con un lenguaje que permita describir estos dominios de la aplicación. Actualmente se han realizado esfuerzos para crear lenguajes ricos y flexibles que permitan realizar esta tarea, estandarizándolos y creando herramientas que faciliten la creación, análisis y manipulación de estas descripciones. Estos esfuerzos se engloban dentro del marco de la Web Semántica y de la definición de ontologías, de los que se hablará en los siguientes apartados.

2.3. Web Semántica

La Web ha experimentado en los últimos años un crecimiento muy elevado. En este sentido, los buscadores suponen el principal punto de entrada a la Web, recopilando la información que consiguen localizar en el contenido de los documentos. Sin embargo, la información así disponible no es suficientemente rica como para poder localizar recursos y relaciones o como para poder contar con una capacidad de razonamiento.

La Web Semántica se enfrenta a este problema, incorporando información adicional para que la información disponible en la Web pueda ser analizada de forma automática. De esta manera se posibilita que los diferentes sistemas conectados a la Web puedan trabajar en cooperación.

2.3.1. Objetivos

Los objetivos que se persiguen con la Web Semántica son [Álvarez Espinar, 2007]:

1. Dar sentido a los contenidos, facilitando a las máquinas comprender lo que se desea representar en ellos.
2. Conectar los recursos de la Web, empleando relaciones semánticas en lugar de hiperenlaces.
3. Gestionar los recursos de forma inteligente, permitiendo localizar la información más eficientemente, extraer los contenidos de los documentos e integrando los recursos heterogéneos.

[3] http://www.isi.edu/isd/LOOM/PowerLoom/

4. Interoperabilidad semántica, permitiendo integrar metadatos de distintos vocabularios para obtener relaciones entre recursos y ofrecer mayor precisión en las búsquedas.

Actualmente estos objetivos no se pueden alcanzar debido a que no existe un contenido en las páginas web que permita a las máquinas extraer la información útil, analizarla, catalogarla y obtener relaciones entre recursos.

2.3.2. Características de la Web Semántica

En la Web, la información se almacena en documentos con formato HTML que, aparte del texto del propio documento, incluyen *marcas* o *etiquetas* que dan *formato* al contenido en términos de títulos, secciones, tipos de letra, etc. Algunas marcas representan *enlaces* a otros documentos, por lo que se podían obtener ciertas relaciones. Los buscadores emplean algoritmos que determinan la relevancia de las páginas en función de algunas palabras clave, el número de enlaces, etc.

La información que aportan estas marcas es *sintáctica*, es decir, a nivel de *representación*, por lo que no aportan información sobre el *contenido* del documento que pueda ser procesada por una máquina. Los buscadores se limitan a emplear algoritmos que analizan el contenido de las páginas buscando palabras clave.

La Web Semántica se enfrenta a esta carencia, empleando el concepto de *metainformación semántica*, es decir, información relativa al *contenido* del documento. En un documento, la metainformación determina los recursos sobre los que el documento versa, de forma que puede ser procesada por una máquina. Un buscador, en este caso, *conoce* el contenido del documento y puede determinar su nivel de relevancia en una búsqueda.

Cuando dos documentos versan sobre un mismo recurso, se obtiene que ambos documentos están relacionados. De esta forma se generan relaciones *semánticas* que permiten a las máquinas realizar deducciones. En este sentido, la Web Semántica supone un gran interés en ámbitos como el comercio electrónico, gestión del conocimiento, enseñanza, investigación, etc.

Para lograr este objetivo, el W3C[4] propone emplear el lenguaje RDF [5] para describir los diferentes recursos y sus propiedades. Este lenguaje se basa en la idea de que tanto los recursos (personas, documentos, entidades, etc.) como los nombres de las propiedades ("nombre", "edad", "título", "año de publicación", etc.) se identifican mediante una URI única.

Aunque no es una práctica que se haya adoptado por la totalidad de la Web, sí es común que las diferentes entidades que operan en Internet permitan acceder a sus recursos mediante URIs. Por otro lado, ya que los nombres de las propiedades deben ser URIs únicas, se hace necesario que algunas propiedades *comunes* (como "nombre" o "título") tengan nombres controlados por alguna entidad independiente. Con este objetivo existen algunos repositorios como el de la organización DCMI [6] que publica un conjunto de nombres de propiedades para describir documentos, tales como

[4]Página principal del World Wide Web Consortium, http://www.w3c.org
[5]http://www.w3.org/RDF/
[6]Página web de Dublin Core, http://dublincore.org/

título, autor, editor, tema, etc.

El empleo de RDF es el primer paso hacia la Web Semántica, y su empleo permite que una máquina pueda realizar consultas semánticas. Sin embargo, para alcanzar los objetivos planteados en la sección 2.3.1 y en concreto la capacidad de razonamiento y deducción, es necesario realizar una descripción formal de los términos empleados en las descripciones de los recursos. Con ese objetivo se emplean lenguajes de descripción de ontologías, que se verán en la siguiente sección.

2.4. Ontologías

Una de las definiciones más conocidas del término *ontología* es la dada por Gruber en 1993, que la define como *una especificación de una conceptualización*. Esta definición hace referencia a dos aspectos:

1. Para *entender* un dominio hay que emplear *conceptualizaciones*, representaciones simplificadas y abstractas de cada parte del dominio.
2. Para *definir* un dominio hay que *especificar* o *describir formalmente* las conceptualizaciones.

Se puede entender, por tanto, que una ontología es una especificación de los diferentes conceptos involucrados en un dominio a tratar. Los mecanismos que se emplean en una ontología son [Cámara, 2002]:

- *Clases*: Identifican conceptos del dominio del discurso, como *personas* o *libros*.
- *Subclases*: Representan conceptos derivados o refinados a partir de otros conceptos, como un *niño* es una subclase de *persona*. Se dice que una subclase *hereda* de su clase padre.
- *Jerarquía de clases*: Conjunto de clases y subclases que comparten relaciones de herencia, como *ser vivo*, *persona* y *niño*.
- *Instancias*: Objetos concretos de una clase.
- *Propiedades*: Características o atributos de la diferentes clases.
- *Hechos*: Definen los valores posibles de una propiedad.
- *Cardinalidad*: Define cuántos valores puede tener una propiedad.
- *Axiomas*: Teoremas que se declaran obre relaciones que deben cumplir los elementos de una ontología. Permiten inferir conocimiento que no está indicado explícitamente en la taxonomía de conceptos.

2.4.1. Lenguajes de ontologías para la Web Semántica

Uno de los ámbitos donde más se han estudiado las ontologías es la Inteligencia Artificial. La necesidad de expresar los conceptos de una ontología requiere un lenguaje lógico y formal. En este sentido, se distinguen tres tipos de lenguajes:

1. *Lógica de primer orden, o lógica de predicados*: Supone que existen *objetos*, entidades indi-

viduales con *propiedades* distintivas entre los que existen relaciones. Se definen *axiomas* que describen fragmentos de conocimiento y, mediante un conjunto de reglas de inferencia aplicadas a los axiomas, se deriva nuevo conocimiento. Un ejemplo de este tipo de lenguajes es KIF[7].

2. *Lógica basada en marcos*: Un marco es una representación de un concepto, con atributos y relaciones a otros conceptos. Asociado a él existe una definición *declarativa* y otra *procedimental*. La primera está basada en lógica de descripciones, que permite describir la semántica del concepto. La segunda permite realizar cálculos e inferencias sobre la primera. Comparados con los lenguajes del grupo anterior, son más intuitivos pero ofrecen menos capacidad de inducción. Un ejemplo de lenguaje basado en marcos es Ontolingua[8].
3. *Lógica de descripción (DL)*: Surge como evolución de la lógica basada en marcos y está orientada a ser más robusto en razonamiento. Los lenguajes DL permiten definir semántica formal basada en lógica, a diferencia de los otros tipos vistos anteriormente. Algunos ejemplos son Loom[9] o CLASSIC.

Estos tipos de lógicas han dado lugar a una serie de lenguajes empleados en la Web Semántica:

RDF: En la sección anterior se introdujo el concepto de RDF como un marco de trabajo que emplea un lenguaje para describir recursos que permitía una representación de los metadatos semánticos. Aunque es un lenguaje sencillo, no contempla los mecanismos suficientes como para considerarlo un lenguaje de definición de ontologías. Sin embargo, permite una descripción semántica de los recursos de forma sencilla.

RDF describe un recurso mediante un conjunto de tripletas $(sujeto, propiedad, objeto)$. El *sujeto* es el recurso del que se realiza la descripción; la *propiedad* es la característica del sujeto que se describe y el *objeto* es el valor asignado a la propiedad, que puede ser un tipo básico (cadena de caracteres, números, etc.) u otro recurso.

Por otro lado, existe un lenguaje basado en XML llamado RDF/XML[10] que permite realizar la misma tarea que con RDF empleando una sintaxis XML.

RDFS: RDF emplea un lenguaje para describir recursos, por lo que no se puede considerar un lenguaje ontológico ya que no incorpora los mecanismos para definir tipos de recursos, las propiedades que poseen, etc. Con este objetivo surge RDFS[11] que permite definir tipos de objetos, propiedades, establecer relaciones de herencia, etc.

[7] http://logic.stanford.edu/kif/kif.html
[8] http://www.ksl.stanford.edu/software/ontolingua/
[9] http://www.isi.edu/isd/LOOM/
[10] http://www.w3.org/TR/rdf-syntax-grammar/
[11] http://www.w3.org/TR/rdf-schema/

DAML+OIL Este lenguaje[12] surge como una cooperación entre dos extensiones de RDF, DAML[13] y OIL[14], que extienden RDF para añadir construcciones más complejas que enriquecen el lenguaje. Una ontología DAML+OIL posee una estructura que permite que una máquina pueda inferir una tripleta RDF a partir de otra mediante mecanismos de herencia, equivalencia, etc.

OWL OWL[15] se desarrolla partiendo de DAML+OIL como base y es actualmente el lenguaje de definición de ontologías más rico, incluyendo vocabulario para definir relaciones de disjunción, cardinalidad, igualdad, simetría entre propiedades, etc.

OWL ofrece tres sublenguajes diseñados para satisfacer las necesidades de los usuarios:

1. *OWL Lite*: Orientado a los usuarios que necesitan definir jerarquías de clase y condiciones sencillas.
2. *OWL DL*: Ofrece una máxima expresividad dentro de unos límites que garantizan *totalidad computacional* (todas las conclusiones son computables) y *decidibilidad* (todas las operaciones terminan en un tiempo finito).
3. *OWL Full*: Ofrece una máxima expresividad sin garantías computacionales.

Estos lenguajes se relacionan entre si de la forma:

$$Lite \subset DL \subset Full$$

Así:

- Una ontología válida *OWL Lite* es una ontología válida *OWL DL*.
- Una ontología válida *OWL DL* es una ontología válida *OWL Full*.

2.4.2. Ventajas e inconvenientes del uso de ontologías

Al disponer de una especificación de los conceptos empleados en un dominio de conocimiento, las ventajas que se consiguen con el empleo de una ontología son múltiples. En el marco de la Web Semántica, su empleo es fundamental para conseguir interoperatividad y capacidad de inferencia. Sin embargo, su empleo no está exento de inconvenientes: definir una ontología no es una tarea trivial, y el paso del tiempo puede requerir que un diseño inicial quede descartado. Es necesario que una ontología se defina con *claridad* (con definiciones objetivas, completas, con condiciones necesarias y suficientes), *coherencia* (las inferencias no deben contradecir las definiciones de los conceptos), y *extensibilidad* (la ontología debe ser fácilmente extensible).

Para enfrentarse a la definición de una ontología es conveniente seguir un guión como el siguiente [Noy and mcguinness, 2001]:

[12] http://www.w3.org/TR/daml+oil-reference
[13] http://www.daml.org
[14] http://www.ontoknowledge.org/oil/
[15] http://www.w3.org/TR/owl-ref/

1. *Determinar el dominio y ámbito de la ontología*: Se define el contexto de la ontología, para qué se va a emplear y por parte de quién.
2. *Reutilizar ontologías existentes*: Antes de crear una ontología nueva conviene comprobar si la comunidad cuenta con alguna que pueda ser reutilizada.
3. *Enumerar términos importantes*: Realizar una lista de los términos que se emplean en el dominio del discurso, agrupando sinónimos.
4. *Definir clases, jerarquía, propiedades y rango de las mismas*: En ocasiones se puede realizar primero una jerarquía de clases y después una definición de las propiedades.
5. *Crear instancias*: Para las clases que lo requieran se define una instancia de las mismas, completando los valores de sus propiedades.

2.4.3. Ontologías en federación de bases de datos

Una vez comprendido el concepto de ontología, se hace evidente su utilidad en términos de la federación de bases de datos: en lugar de emplear un esquema global de tipo relacional, el uso de una ontología aporta una gran cantidad de información en términos de clases y propiedades, restricciones y relaciones entre clases.

En este sentido, el título de este proyecto cobra sentido ya que lo que se pretende es obtener la capacidad de tomar un conjunto de bases de datos y *mapear*[16] las tablas y columnas a clases y propiedades de una ontología, de forma que los registros y las relaciones entre tablas se traduzcan a recursos y propiedades. Si se envuelven diferentes tipos de fuentes de datos con *wrappers* se pueden federar repositorios heterogéneos.

De esta forma, la conveniencia de emplear las tecnologías disponibles en la Web Semántica para la definición de recursos y ontologías se hace máxima. Las consultas a la federación de bases de datos se puede realizar accediendo directamente a las URLs de los recursos, de forma que toda la información disponible puede ser inspeccionada por un agente externo y, mediante la publicación de la ontología, formar parte de un sistema inteligente de búsqueda de información.

En la siguiente sección se enumeran algunos proyectos existentes relacionados con este enfoque.

2.5. Sistemas de integración basados en ontologías

En la sección 2.2 se han expuesto algunos sistemas de integración de fuentes de datos. A partir de su estudio se ha considerado la conveniencia de emplear ontologías para la definición de los esquemas globales. A continuación se enumeran otros sistemas existentes que emplean un enfoque similar, basado en ontologías, para lograr la federación de fuentes de datos.

[16]Se empleará la palabra *mapeado* en el sentido de una *correspondencia* entre campos de una fuente de datos y elementos de una ontología.

OBSERVER: Ofrece como novedad que emplea varias ontologías, cada una de las cuales representa una fuente de datos [Mena et al., 1996]. Ya que las ontologías pueden ser diferentes entre sí, se emplea un componente central para relacionar los conceptos de las ontologías disponibles.

Aunque este enfoque ofrece la ventaja de una mayor adaptabilidad frente a los cambios, emplear un elevado número de ontologías independientes y mantener sus correspondencias puede suponer una tarea muy compleja.

Para definir las ontologías, OBSERVER soporta los lenguajes Loom y CLASSIC, que no forman parte de la recomendación del W3C para la pila de formatos de la Web Semántica, por lo que sería necesario realizar un traductor para permitir a los sistemas externos acceder a una ontología especificada en OWL.

DOME: DOME [Cui and O'Brien, 2000] es, en cierto sentido, semejante a OBSERVER ya que emplea un conjunto de ontologías diferentes para definir el esquema federado . Los usuarios de DOME crean una ontología general que modele el dominio del discurso. Esta labor se denomina *análisis top-down*.

Tras ello, DOME hace uso de herramientas para la extracción automática de ontologías a partir de los esquemas de una fuente, creando conjuntos de clases y relaciones que pueden inferirse a partir de las definiciones de los esquemas locales. A este análisis se le denomina `bottom-up` y se basa en la capacidad de las herramientas de realizar una ingeniería inversa de las fuentes. Por último se definen mapeados entre las entidades de las ontologías resultantes.

De igual manera que OBSERVER, DOME emplea el lenguaje CLASSIC para la definición de las ontologías, lo que se aleja del objetivo planteado en este proyecto.

KRAFT: KRAFT [Gray et al., 1997] emplea un enfoque *mixto*, definiendo una ontología local para cada fuente de datos (como DOME y OBSERVER) y una ontología global para el dominio (como SIMS). También se crea un conjunto de correspondencias entre la ontología global y las locales. Este sistema presenta el inconveniente de que un cambio en una fuente de datos implica alterar tanto su ontología local como la ontología global.

Universidad de La Coruña: El Departamento de Computación de la Universidad de La Coruña ha desarrollado un sistema de federación de bases de datos basado en ontologías [Brisaboa et al., 2002b] [Brisaboa et al., 2001] [Brisaboa et al., 2002a]. Aunque su estudio se ha realizado fundamentalmente en el ámbito de las bases de datos documentales, el sistema se puede emplear para federar cualquier base de datos.

La arquitectura propuesta en este proyecto se basa en el empleo de envoltorios para las bases de datos y un mediador que distribuye las consultas en base a una ontología. En este sentido, el sistema ha evolucionado partiendo del empleo de varias ontologías (una común y varias específicas para cada base de datos) hasta alcanzar el empleo de una única ontología común, obviando las

ontologías específicas: cuando las consultas se reparten a los envoltorios de bases de datos, estos hacen uso de un *árbol de correspondencias* para relacionar los campos de las bases de datos y los conceptos de la ontología.

Uno de los puntos débiles de este sistema consiste en que la ontología general se especifica en un lenguaje propio de esta plataforma que contempla un subconjunto de las entidades representables en, por ejemplo, OWL, tales como cardinalidad, uniones e intersecciones de clases, etc. Por ello, durante la evolución del sistema, los autores han decidido adoptar el nombre de *árbol de conceptos* para lo que inicialmente identificaban como ontología. Aunque en [Brisaboa et al., 2002b] se hace un estudio de cómo adoptar una ontología para sustituir un *árbol de conceptos*, esta adopción no se ha llevado a cabo.

Si no se requiere el uso de un lenguaje rico como OWL para la definición de una ontología, y por tanto si los mecanismos del lenguaje de definición de árboles de conceptos de este sistema resulta suficiente para realizar el modelo de datos de la aplicación, este sistema puede suponer una buena solución sólo en el caso en que no se considere que la información de un mismo recurso pueda estar en varias bases de datos a la vez. Esto se debe a un criterio de diseño del propio sistema.

Las soluciones estudiadas en este capítulo muestran que, aunque existen soluciones para la integración de bases de datos empleando ontologías, se puede comprobar que no están alineadas con el objetivo planteado en el proyecto, o bien presentan carencias en la funcionalidad que se consideran importantes. Por ello se motiva la creación de una nueva herramienta, Antive, cuyo diseño e implementación se aborda en los siguientes capítulos.

Capítulo 3

Análisis

En este capítulo se hará una introducción a la herramienta Antive estudiando los problemas que se desean abordar y las soluciones que Antive debe ofrecer. Se ubicará la herramienta en un contexto apropiado para su uso y se analizarán los casos de uso fundamentales.

3.1. Problemas y soluciones generales

En las siguientes secciones se detallan las funcionalidades básicas que Antive debe presentar, enfocando la aplicación desde un nivel alto de abstracción para obtener posteriormente el conjunto de requisitos necesarios para el diseño que se aborda en los próximos capítulos.

3.1.1. Composición de repositorios

La tarea fundamental de la aplicación debe ser federar diferentes repositorios de datos que se suponen independientes y no coordinados, donde la información de los recursos no está almacenada de forma *coherente*, es decir, que diferentes fuentes de datos aporten diferente información respecto a un mismo concepto.

Bajo esta suposición se puede dar que la información relativa a un recurso se encuentre:

1. En su totalidad en una de las bases de datos.
2. Distribuida en varias bases de datos.
3. Replicada en varias bases de datos.
4. Distribuida de forma no coherente en varias bases de datos.

De esta manera, una de las principales características de la aplicación consistirá en no considerar ninguna cohesión entre las diferentes fuentes, de forma que al realizar una consulta para recuperar un recurso, el sistema debe ser capaz de consultar de forma independiente a cada una de las bases de datos y encontrar relaciones entre toda la información disponible para componer el recurso final.

De ahí que, a pesar de la falta de coherencia y la distribución de las diferentes bases, lo que

se busca es conseguir un sistema federado en el que se ofrezca una completa transparencia tanto en el esquema de las bases de datos individuales como en su localización u otras características particulares.

3.1.2. Heterogeneidad

Pese a que la aplicación debe centrarse en la federación de bases de datos, es muy común que el acceso a determinados repositorios no se realice directamente contra el SGBD sino que se emplee una interfaz para facilitar la consulta y añadir seguridad y control a la misma.

Por ello, es conveniente plantear que los repositorios de datos no sean únicamente bases de datos relacionales con interfaces SQL, sino que la aplicación deberá permitir incorporar otras fuentes de datos. Es especialmente interesante considerar accesos a servicios web basados en SOAP, aplicaciones de consulta vía XML-RPC o servicios de consulta con acceso a través de URLs. A medida que el enfoque semántico avanza por la Web, crece el número de sitios que permiten consultar recursos mediante URLs de tipo REST y de los que se obtiene la información en formato RDF o similar, por lo que es de esperar que el acceso a este tipo de repositorios de información deba ser abordado en un futuro próximo.

3.1.3. Uso de ontologías

Se ha visto en el capítulo 2 que para ofrecer un único esquema al usuario y abstraerlo de los esquemas particulares de las diferentes bases de datos, el uso de una ontología se presenta como una opción especialmente interesante. Por un lado, los lenguajes de definición de ontologías como RDF u OWL son amplios y generales y permiten un modelado rico del esquema general de la federación; por otro lado, dichos lenguajes son recomendaciones del W3C, lo que asegura interoperabilidad y estandarización, dos de los pilares fundamentales necesarios para el éxito de la aplicación.

Por ello, el esquema general de la federación que la aplicación ofrecerá estará lejos de un esquema *tradicional* de una base de datos. Los esquemas globales ofrecidos por los SGBD multi-base de datos no alcanzan el nivel de abstracción que Antive debe ofrecer, modelando los recursos en clases pertenecientes a una jerarquía, con propiedades de tipos básicos y de tipo objeto, muy lejos de presentar los característicos elementos de los sistemas relacionales de tablas y columnas.

Uno de los mayores problemas en la federación de bases de datos son los diferentes conflictos que se pueden generar entre los diferentes esquemas de cada instancia [Sheth and Larson, 1990]: conflictos de nombrado (al emplear diferentes nombres para el mismo concepto), de dominio (al asignar diferentes valores para el mismo concepto), de datos (al contar con instancias en las que faltan algunos atributos), etc. En resumen, el empleo de diversas fuentes de datos heterogéneas y no coordinadas da lugar inevitablemente a ausencias y conflictos de datos.

Al emplear una ontología para crear el esquema general de la federación se obtienen unos mecanismos que permiten solventar los problemas generados por estos conflictos como son las

propiedades multievaluadas, la herencia múltiple o el tipado de las propiedades.

3.1.4. Interfaz de mapeado

Uno de los requisitos fundamentales de la aplicación es la necesidad de una interfaz sencilla que permita realizar la correspondencia entre las clases y propiedades de una ontología y los campos de las diferentes fuentes de datos. De esta necesidad, junto con la intención de poder emplear ontologías ricas y bien definidas, se derivan dos pilares fundamentales de la aplicación:

- En primer lugar la creación de la ontología se dejará en manos de una aplicación dedicada a ello. No se pretende crear un editor de ontologías tan completo y flexible como puede serlo cualquier aplicación disponible (como puede ser Protégé), pero sí se pretende que la aplicación sea capaz de cargar una ontología hecha con una de esas herramientas.
- En segundo lugar el proceso de hacer corresponder las distintas fuentes de datos con las diferentes clases y propiedades de la ontología debe ser fácil, sencillo y rápido. Se usará una interfaz web para permitir el acceso remoto a la aplicación sin necesidad de instalar ningún software adicional en local; por otro lado se hará hincapié en el empleo de una interfaz rica y muy usable.

Por último, cabe esperar que la aplicación sea suficientemente sencilla de utilizar como para que pueda ser empleada por un usuario medio (ver sección 3.4.4).

3.1.5. Interfaz de búsqueda

Una vez que la aplicación cuenta con una ontología y sus correspondencias a las fuentes de datos, se debe contar con una interfaz para realizar consultas sobre ella. Como punto de partida se plantean las siguientes consultas:

1. Conocer los tipos de recursos disponibles (que serán las clases existentes en la ontología).
2. Localizar los recursos de un tipo.
3. Obtener la información de un recurso concreto.
4. Listar los recursos relacionados con un recurso dado.

Para enfrentarse a estas consultas el usuario se servirá de una interfaz web que mostrará como punto de partida el árbol de la ontología y permitirá, mediante diferentes enlaces, ir accediendo a los distintos recursos y a sus propiedades.

Por otro lado la aplicación deberá permitir el acceso a estas consultas mediante una URL bien formada, alineando así el funcionamiento de la aplicación con las arquitecturas REST y con la propia Web Semántica: un recurso disponible en Antive debe estar identificado mediante una URL única que será a su vez una URL de consulta para Antive.

3.1.6. Actualizaciones

La herramienta debe permitir la actualización de los distintos mapeados existentes de forma que puedan añadirse o cambiarse nuevas relaciones, fuentes de datos, etc. sin destruir los ya existentes. Esto es de especial interés en dos casos fundamentales:

1. La adición de nuevas fuentes de datos, de forma que se puedan agregar nuevos datos al repositorio con facilidad.
2. La actualización de alguna fuente, como cambios en los parámetros de conexión, en los esquemas locales, etc.

3.2. Contexto de la herramienta

La figura 3.1 muestra el contexto general del uso de Antive. Se distinguen tres actores:

1. Los administradores de las bases de datos, que no tienen relación directa con Antive.
2. Los administradores de Antive, responsables de definir las ontologías y las correspondencias con las distintas fuentes de datos.
3. Los usuarios, que hacen uso de Antive para consultar las distintas ontologías.

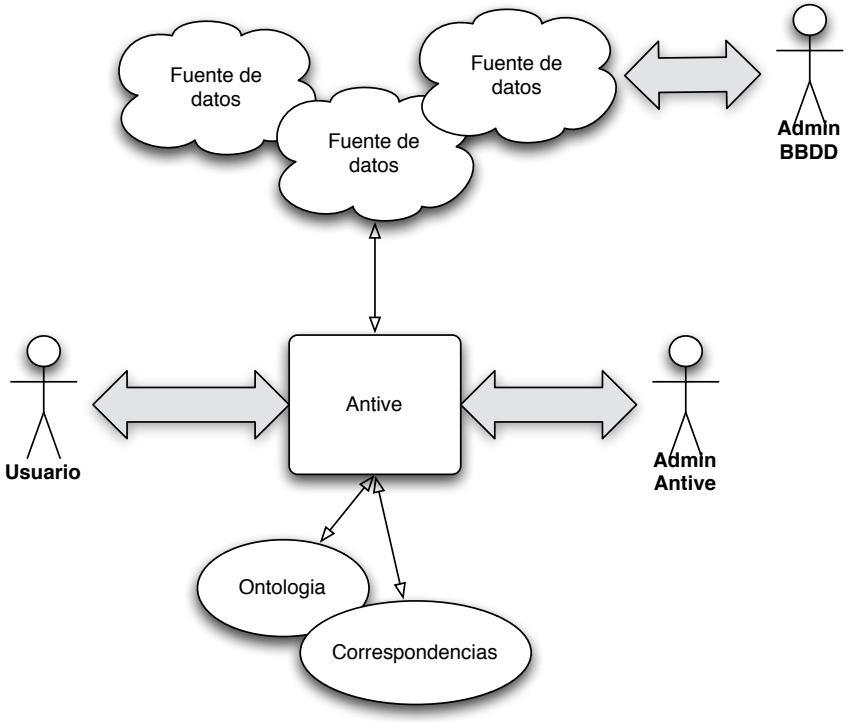

Figura 3.1: Contexto general de Antive

Nótese que no existe una relación directa entre los administradores de las bases de datos ni de Antive, lo que resulta esperable. La información que ambos administradores comparten son los

datos de conexión a los repositorios y el esquema de cada fuente de datos.

3.3. Requisitos funcionales

Teniendo en cuenta los problemas a alto nivel y el contexto general de la herramienta que se han abordado en las anteriores secciones, a continuación se realizará un análisis funcional.

Una de las herramientas más valiosas para realizar una especificación funcional de una herramienta consiste en el empleo de *casos de uso*: descripciones del comportamiento del sistema en base a su respuesta a una petición que se origina fuera del mismo. En otras palabras, un caso de uso determina el comportamiento que la aplicación debe presentar en unas condiciones determinadas.

La figura 3.2 muestra un diagrama UML de casos de uso. Se consideran dos tipos de usuarios: el *Usuario* y el *Administrador* (*Admin antive* en la figura 3.1). El uso general de la aplicación, que consiste en realizar consultas a la ontología, puede realizarse por ambos usuarios, mientras que la administración de la herramienta se lleva a cabo por el administrador únicamente.

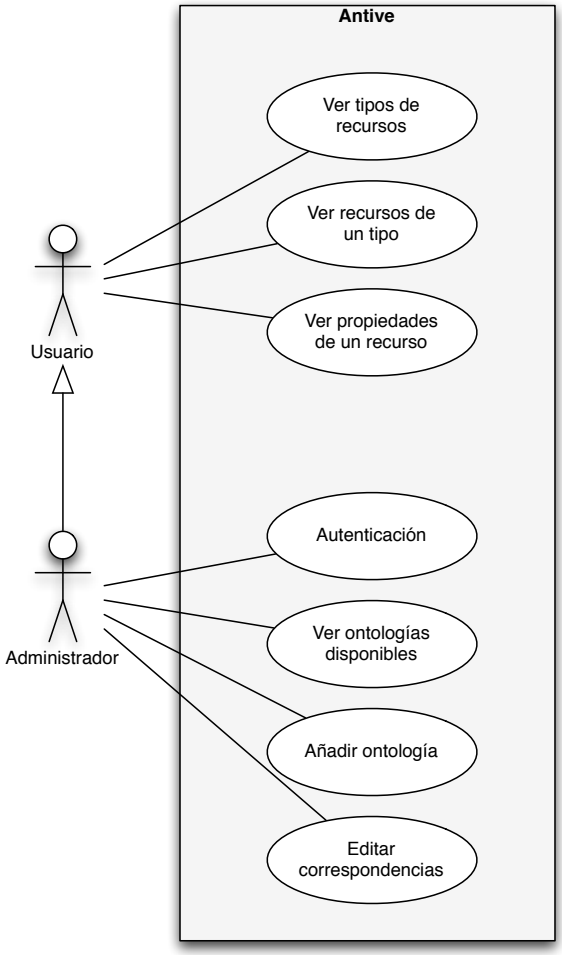

Figura 3.2: Casos de uso de Antive

3.3.1. Ver tipos de recursos

El punto de partida para un usuario será acceder a los tipos de recursos disponibles, que corresponden con la jerarquía de clases de una ontología. Para ello será necesario acceder con un navegador web a una URL determinada. Cada ontología tendrá su URL propia, que será de la forma:

```
http://servidor/ontologia
```

3.3.2. Ver los recursos de un tipo

Mediante una URL, se podrá acceder a la lista de recursos pertenecientes a un tipo (a una clase de la ontología). La URL de esta consulta tendrá la forma:

```
http://servidor/ontologia/clase
```

y será configurable como se verá en la sección 3.3.4.

3.3.3. Ver propiedades de un recurso

Por último se podrá acceder a las propiedades de un recurso mediante una URL de la forma:

```
http://servidor/ontologia/clase/id
```

Al acceder a esta URL se obtendrá una lista de las propiedades *datatype*, cada una de ellas con los valores recuperados de las fuentes de datos; también se tendrá una lista de las propiedades de tipo objeto que podrán consultarse mediante una URL como la siguiente:

```
http://servidor/ontologia/clase/id/propiedad
```

3.3.4. Ejemplo

El mecanismo de consulta mediante URLs se basa en la capacidad del administrador de configurar una URL de consulta para cada una de las clases de una ontología. A modo de ejemplo se considerará una ontología que se identificará con el nombre:

```
familia
```

Esta ontología tiene, entre otras, una clase con el siguiente nombre:

```
http://servidor/Persona
```

Esta clase tendrá un conjunto de propiedades, de las que se destacan las siguientes:

1. Una propiedad *datatype* que será identificador único con nombre:

   ```
   http://servidor/dni
   ```

2. Una propiedad *object* con nombre:

   ```
   http://servidor/hijos
   ```

   cuyo rango será la clase `Persona`.

El administrador de Antive, a la hora de crear la correspondencia entre las distintas propiedades y las bases de datos, configura que la clase `Persona` tendrá la siguiente URL de consulta:

```
http://servidor/familia/personas/[http://servidor/dni]
```

Gracias a esta configuración, se tiene que:

1. La jerarquía de clases se obtiene en la URL:

   ```
   http://servidor/familia
   ```

2. La lista de personas encontradas en las bases de datos se accede con la URL:

   ```
   http://servidor/familia/personas
   ```

3. Las propiedades de la persona con DNI `123456789` se pueden conocer mediante la URL:

   ```
   http://servidor/familia/personas/123456789
   ```

4. La lista de personas que son hijos del sujeto anterior se puede visualizar con la URL:

   ```
   http://servidor/familia/personas/123456789/hijos
   ```

Cabe destacar, por un lado, que las URLs construidas así son URLs de tipo REST, por lo que pueden emplearse en cualquier aplicación que ofrezca esta interfaz y que tenga alguna relación de uso con Antive. Por otro lado, con este mecanismo se puede conseguir que los diferentes recursos localizados por Antive tengan una URL única como nombre, lo cual se alinea con el concepto de objeto en los términos de la Web Semántica.

3.3.5. Autenticación

De cara al uso de Antive por parte de un administrador, es necesario un proceso de autenticación para poder realizar cualquier tarea administrativa. En principio no se considera ningún mecanismo para la administración de usuarios (base de datos propia de Antive, servidor LDAP, etc).

3.3.6. Ver ontologías disponibles

El administrador de Antive podrá conocer el conjunto de ontologías disponibles y visualizar la jerarquía de clases de cada una de ellas.

3.3.7. Añadir ontología

Como se vio en la sección 3.1.3, uno de los pilares de Antive consiste en su capacidad para cargar ontologías creadas en alguna herramienta experta en lugar de implementar un editor de ontologías

propio. Por ello es necesario un mecanismo para añadir una nueva ontología a Antive mediante un *asistente* que le ayude en el proceso.

3.3.8. Editar correspondencias

La principal función de Antive para el administrador será editar las correspondencias entre las distintas clases y sus propiedades con los campos de las fuentes de datos. Para ello se usará un editor que permitirá:

1. Añadir un conjunto de fuentes de datos, introduciendo los parámetros de conexión.
2. Crear una correspondencia entre los campos de las fuentes de datos y las propiedades *datatype* de cada clase.
3. Editar la URL de consulta para cada clase.
4. Crear las relaciones entre clases que sean dominio y rango de una propiedad de tipo objeto.

3.4. Requisitos no funcionales

A continuación se enumeran algunos requisitos no relacionados con ninguna funcionalidad concreta.

3.4.1. Interfaz web

La interfaz web que ofrecerá la herramienta debe ser accesible en cualquier plataforma y con los navegadores más utilizados. Se generará un código limpio y basado en XHTML.

3.4.2. Formato de almacenamiento

La información de mapeado de cada ontología se almacenará en formato XML.

3.4.3. Almacenamiento

Antive contará con un repositorio local donde se almacenarán las definiciones de las ontologías y de sus mapeados formando pares, de forma que puedan ser fácilmente accesibles para un administrador de la máquina para realizar tareas de mantenimiento, backup, etc.

3.4.4. Nivel de conocimiento de los usuarios

Como se ha visto en el análisis del contexto de la herramienta Antive (ver sección 3.1) se distinguen los perfiles de *administrador* y de *usuario*. El nivel de conocimientos esperado de estos usuarios es el siguiente:

1. Los administradores deben tener conocimientos básicos sobre Web Semántica y ontologías. También deben tener experiencia en el uso o administración de bases de datos (u otras fuentes que se estén empleando: servicios WEB, servidores LDAP, etc.)
2. No se requiere que los usuarios tengan ningún conocimiento específico respecto a bases de datos u ontologías.

Capítulo 4

Diseño

En el capítulo 3 se ha realizado un análisis del sistema completo para obtener así el conjunto de funcionalidades y requisitos que se deben cumplir. En este capítulo se realizará un estudio de Antive para mostrar el diseño de los diferentes componentes y subsistemas que lo forman.

4.1. Requisitos generales de diseño

Para el diseño e implementación de Antive debe mantenerse el enfoque de que el objetivo del desarrollo no es crear una solución final sino que es posible que Antive sea retomado en el futuro para ampliar su funcionalidad y adaptarlo a nuevas necesidades y entornos.

Como consecuencia de esto surge la necesidad de diseñar Antive de forma modular, creando subsistemas independientes que puedan ser considerados, desacoplados, administrados y actualizados sin la necesidad de abordar la solución completa. Esto conlleva, además, la necesidad de que estos componentes ofrezcan una interfaz clara y bien definida, lo que supone un requisito básico para la *neutralidad tecnológica* imprescindible para el éxito de las tecnologías relacionadas con la Web Semántica.

De aquí se deduce la necesidad de que el usuario de la herramienta pueda acceder a ella sin emplear una solución software determinada; bastará con un navegador web sin necesidad de instalar ningún software adicional. Nótese que nos referimos a los navegadores más empleados, como Mozilla Firefox o similares.

Por último, como criterio de diseño general, no se deberá perder de vista el principio de mínima sorpresa: será necesario que la interfaz que Antive ofrezca al usuario sea semejante a las que un usuario puede emplear en su trabajo diario. Se deberá evitar el empleo de metáforas originales o mecanismos confusos o poco extendidos.

4.2. Componentes del sistema

La figura 4.1 muestra los diferentes componentes que forman Antive. Los componentes marcados en gris no son objeto de implementación de este proyecto.

Se puede comprobar que se ha dividido en dos subsistemas:

1. El subsistema de consulta, dedicado a obtener los resultados a partir de los parámetros dados por el usuario.
2. El subsistema de mapeado, cuya función es ayudar al administrador a crear las correspondencias entre las bases de datos y las clases y propiedades de las ontologías.

Por otro lado, cada subsistema se ha subdividido en tres capas siguiendo el paradigma MVC propuesto por Trygve Reenskaug y que actualmente tiene tanta aceptación en el mundo de las aplicaciones web [Ruby et al., 2008]. Las siguientes secciones consideran cada uno de los subsistemas para explicar sus distintos componentes.

4.2.1. Subsistema de consultas

El Subsistema de consultas es una aplicación construida sobre Rails, un marco de desarrollo para aplicaciones web fiel al paradigma MVC. A continuación se verá cada una de las capas del subsistema de consulta para analizar los componentes que lo forman.

Capa de Vista

La vista del Subsistema de consultas se encarga de crear las páginas XHTML a partir de los resultados de una consulta. Las diferentes peticiones que el usuario puede realizar, expuestas en la sección 3.3, se corresponden con una vista. En primera instancia puede entenderse una vista como una plantilla XHTML donde se incorporan etiquetas con código Ruby ejecutable, de forma semejante a JSP o PHP.

Capa de Modelo

Típicamente en Rails un modelo es un objeto que encapsula la funcionalidad de acceso a base de datos; en el caso de Antive el modelo que se emplea es el componente Antive Core que realiza el mismo papel pero consultando a la ontología y sus bases de datos correspondientes.

Capa de Controlador

Esta capa se encarga de gestionar el flujo de ejecución del subsistema, obteniendo la petición del usuario, actuando sobre el modelo y enviando los resultados a la vista.

4.2.2. Subsistema de mapeado

Este subsistema es accesible por los administradores de Antive y ofrece una aplicación web que permite realizar la correspondencia entre los diferentes campos de las bases de datos y las clases y propiedades de la ontología.

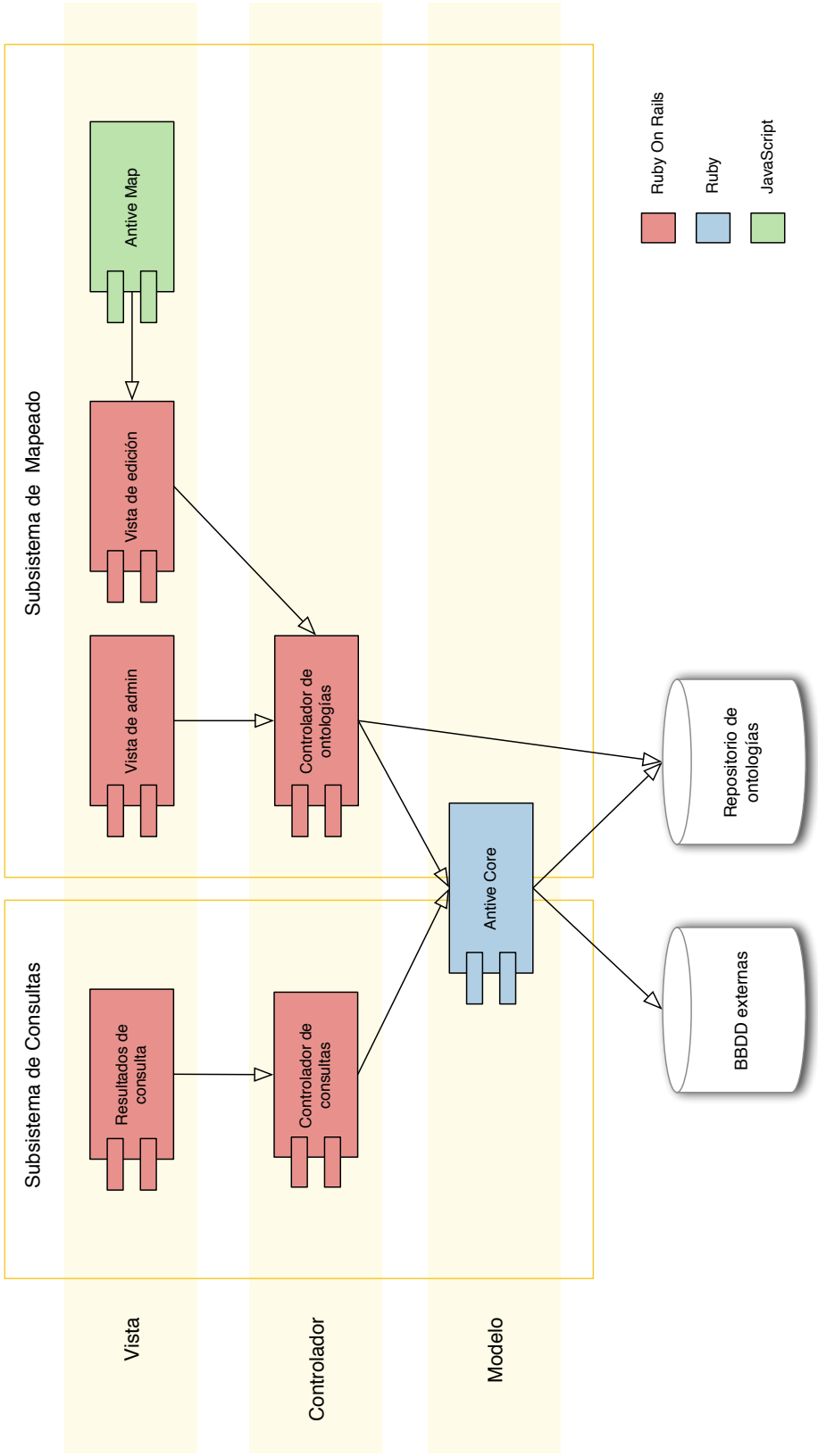

Figura 4.1: Componentes de Antive

Capa de vista

En esta capa se localizan tres componentes:

- Vista de administración: Permite realizar operaciones administrativas sobre las ontologías, como altas, bajas, o redirigir al usuario a la página de consultas.
- Vista de edición: Ofrece una interfaz para realizar la correspondencia entre las diferentes fuentes de datos y las clases de una ontología.
- Antive Map: Implementa la funcionalidad de los controles mostrados por la vista de edición. Este componente está íntimamente relacionado con la vista de edición.

Capa de Modelo

El modelo que se emplea en el subsistema de mapeado es el mismo que en el caso del subsistema de consulta, Antive Core. En este caso, el modelo aporta información sobre cómo está construida la ontología, en términos clases y propiedades definidas, la jerarquía de clases, etc.

Capa de controlador

El último componente de este subsistema es un controlador Rails encargado de realizar operaciones de administración sobre las ontologías (altas, bajas, ediciones). También es responsable de recuperar las ontologías consultando a Antive Core para su posterior empleo en la vista de edición y en Antive Map.

4.3. Despliegue del sistema

La figura 4.2 muestra el diagrama de despliegue de la aplicación en el que se observan los diferentes componentes software y hardware a un nivel muy superior al visto hasta ahora. Se observan los siguientes elementos:

- *Máquina cliente*: Donde se despliega el componente *Navegador web* que empleará el usuario.
- *Máquina servidor*: La máquina que incorpora los siguientes componentes:
  1. Una instancia de Rails que contiene la aplicación Antive y se encarga de analizar las peticiones HTTP, ejecutar la lógica de Antive correspondiente y generar el XHTML de respuesta.
  2. Un servidor web que comunica la aplicación Rails con el navegador del cliente. Se puede emplear cualquier servidor que pueda comunicarse con la instancia de Rails mediante CGI o FastCGI, o bien servidores especializados en aplicaciones Ruby on Rails que ofrecen un mejor rendimiento.
  3. Un sistema de archivos que sirve de repositorio de Antive donde se almacenan las ontologías y sus correspondencias en archivos XML.

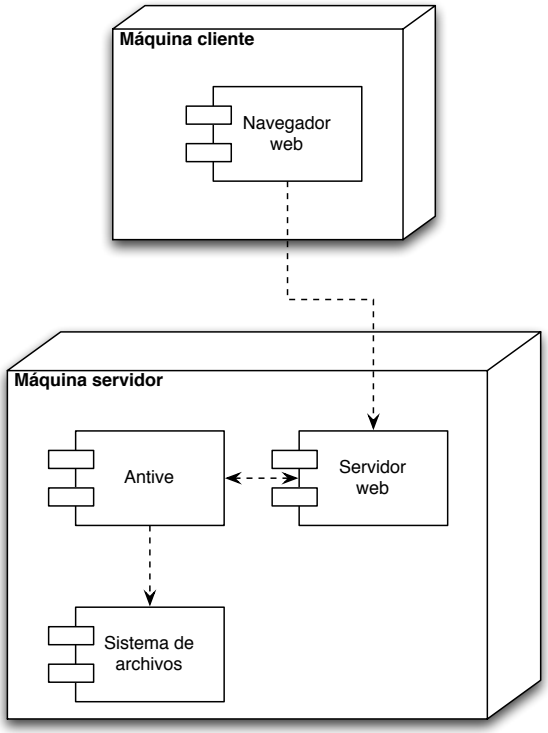

Figura 4.2: Despliegue de Antive

Nótese que en este diagrama de despliegue no aparece ningún componente de base de datos propio del sistema. Este componente se requiere en las instalaciones típicas de Rails, por lo que será necesario configurar el entorno convenientemente. Consultar el apéndice D para más información.

Capítulo 5

Implementación

En este capítulo se verá la implementación de Antive centrándose en los mecanismos que emplea para representar los diferentes elementos de la ontología y cómo se utilizan para resolver consultas.

En primer lugar se verán los lenguajes, plataformas y bibliotecas empleados para la implementación de Antive. Tras ello se realizará un estudio detallado de los componentes de la aplicación presentados en el capítulo anterior. Por último, se realizará un estudio de los algoritmos y mecanismos internos de la aplicación, bajando hasta el nivel del código y del API de los diferentes componentes.

Como se ha visto en capítulos anteriores, Antive se divide en dos subsistemas:

1. Subsistema de consultas, que se encarga de la carga y análisis de ontologías y ofrece un API para realizar consultas y obtener resultados.
2. Subsistema de mapeado, que permite realizar las correspondencias entre clases y fuentes de datos mediante una interfaz web.

Las siguientes secciones detallarán cada uno de estos subsistemas en profundidad, así como el marco de trabajo que se ha empleado para crear la aplicación final.

5.1. Elección de plataforma y lenguajes

5.1.1. Ruby y jRuby

Ruby [Thomas, 2005] es un lenguaje de programación diseñado en 1995 por Yukihiro Matsumoto, quien lo creó como respuesta a las carencias encontradas en otros lenguajes como Perl, Smalltalk, Lisp o Ada. Debido a ello, Ruby es un lenguaje muy rico, flexible, potente y muy sencillo de aprender y utilizar. En el apéndice A se ofrece una introducción al lenguaje.

jRuby[1] es una implementación del intérprete de Ruby escrita en Java. Ofrece, además, la capacidad de traducir en tiempo de ejecución algunas clases Ruby a su equivalente Java, de forma

[1] http://jruby.codehaus.org/

que se pueden emplear bibliotecas Java en aplicaciones Ruby de forma transparente. Por todo ello, Ruby y jRuby son el lenguaje y su implementación elegidas para el desarrollo de la mayor parte de la aplicación.

5.1.2. JavaScript y JSON

JavaScript es un lenguaje muy extendido en el mundo de las aplicaciones web. Los principales navegadores implementan un intérprete de este lenguaje para poder crear aplicaciones web ricas y dinámicas en contenido. Igualmente, los navegadores actuales permiten emplear la notación JSON que resulta muy útil a la hora de intercambiar estructuras de datos JavaScript entre un servidor y un cliente web.

5.1.3. Bibliotecas JavaScript

Para facilitar el desarrollo de aplicaciones web existen multitud de bibliotecas JavaScript que ayudan al programador a realizar tareas comunes empleando un API único e independiente del navegador. Antive Map hace uso de dos bibliotecas que gracias a su calidad y facilidad de uso han obtenido una popularidad muy alta:

- Prototype: Permite realizar tareas como manipular el DOM, generar peticiones AJAX, etc.
- script.aculo.us: Ofrece un conjunto de herramientas para crear animaciones e interfaces gráficas de usuario.

Además de la funcionalidad que estas bibliotecas añaden, permiten al programador abstraerse del navegador que esté empleando el cliente, de forma que no es necesario adaptar el código JavaScript a los diversos navegadores del mercado [Porteneuve, 2007].

5.1.4. Jena y OWL

Jena[2] es una biblioteca Java que permite parsear y generar ontologías en diversos lenguajes como RDF u OWL. Antive emplea esta biblioteca para parsear el XML de una ontología.

En el caso de Antive se empleará el lenguaje OWL para definir las ontologías. Sin embargo, gracias a la capacidad de Jena, se podrían emplear otros lenguajes como RDF o RDFS.

5.1.5. Ruby on Rails

Ruby On Rails [Fernandez et al., 2007] es un marco de desarrollo creado por David Heinemeier para aplicaciones web escritas en Ruby cuya popularidad ha crecido exponencialmente desde su liberación en 2004. Los dos componentes principales de Antive, Antive Core y Antive Map, se ejecutan en el ámbito de una aplicación web construida sobre este entorno.

[2] http://jena.sourceforge.net/ontology/index.html

Rails se asienta en cuatro principios básicos:

1. *MVC, model-view-controller:* El patrón de diseño más común en el mundo del desarrollo web del que hablamos en la sección 4.2 está inherentemente implementado en Rails.
2. *COC, convention over configuration:* A diferencia de los marcos de desarrollo creados hasta la liberación de Rails, la configuración de los diferentes componentes que forman la aplicación era una tarea imprescindible para el programador. Con la aparición de Rails se populariza la idea de que la mayor parte de los desarrollos pueden encajar en un conjunto de convenciones que permiten acelerar el proceso de desarrollo.
3. *DRY, don't repeat yourself:* Cada componente, función, etc. ocupa un lugar único en el proyecto y no es necesario replicarlo, evitando así que el desarrollador multiplique esfuerzos.
4. El propio lenguaje Ruby. La flexibilidad del lenguaje es vital para el funcionamiento de Rails gracias a su facilidad para emplear mecanismos como las síntesis de métodos y clases en tiempo de ejecución, introspección, carga dinámica de código, ejecución de código ajeno a métodos de clase, etc.

Tradicionalmente los marcos de desarrollo como Struts, Zend, etc. abarcan la máxima funcionalidad posible para construir aplicaciones web. Por ello el programador necesita *configurar* o *informar* al marco de trabajo de cómo va a desarrollar la aplicación: las relaciones entre componentes, las diferentes vistas, etc. Rails, por contra, se centra en mantener una estructura común y permitir al desarrollador abstraerse de este tipo de tareas construyendo su aplicación basándose en *convenciones* en lugar de *configuraciones* (lo que se denomina el principio *Convention Over Configuration, COC*). Por otro lado, hace uso del paradigma Modelo-Vista-Controlador (*MVC*) para dividir la funcionalidad en diferentes *capas*:

- *Modelo*: Encapsula la funcionalidad de los datos de la aplicación. Típicamente en Rails un modelo es un punto de entrada a la base de datos; en Antive Map el modelo de datos es el ofrecido por Antive Core por lo que no se emplean los modelos Rails.
- *Vista*: Sirve de interfaz entre el usuario y la aplicación. Típicamente es un código encargado de tomar los datos y acciones y representarlos de forma que el usuario pueda entenderlos e interactuar con ellos. En Rails las vistas son fragmentos de plantillas al estilo de las *JSP* ej J2EE.
- *Controlador*: Guía los diferentes estados de la aplicación, comunicándose con los diferentes modelos y vistas para generar así la aplicación completa.

5.2. Anatomía de la aplicación Rails

Una aplicación Rails tiene una estructura de directorios donde se sitúan los diferentes componentes empleados. En el caso de Antive nos centraremos en los siguientes directorios:

- `/app/controllers`: Contiene los diferentes controladores:
  - `application`: Sirve de base para el resto de controladores y contiene un conjunto de

métodos de uso general para la carga y el cacheado de ontologías.

- `login_controller`: Implementa la funcionalidad de autenticación.
- `sources_controller`: Contiene las acciones de *metainformación* de las fuentes de datos. Las consultas a las fuentes se realizan en Antive Core por lo que este controlador se centra en la *metainformación*. Actualmente la funcionalidad de este controlador se centra en obtener las columnas disponibles de una tabla de una base de datos relacional.
- `ontologies_controller`: Contiene el grueso de la aplicación y se centra en el control de las ontologías y sus correspondencias:
  - Acciones REST para listar, mostrar, borrar y editar los diferentes mapeados.
  - Un asistente para importar una nueva ontología y, opcionalmente, un archivo de mapeado.
  - Una acción de búsqueda sobre Antive Core.

- `/app/helpers`: Contiene *modules*[3] de ayuda para los distintos controladores. Actualmente solo se emplea un *helper* para generar una cadena de caracteres que informa del estado de una ontología.
- `/app/views`: Contiene el conjunto de vistas agrupadas por controladores:
  - `/app/views/login`: Contiene las vistas para la funcionalidad de autenticación. Actualmente incluye únicamente el formulario de entrada.
  - `/app/views/sources`: Contiene las vistas que muestran la metainformación de las fuente de datos. Actualmente incluye una vista que ofrece los campos de una fuente de datos en formato JSON.
  - `/app/views/ontologies`: Contiene las vistas de las diferentes acciones del controlador de ontologías:
    - `/app/views/ontologies/index`: Muestra las ontologías disponibles.
    - `/app/views/ontologies/new...`: Muestra el asistente de importación de una nueva ontología.
    - `/app/views/ontologies/show`: Muestra información de una ontología.
    - `/app/views/ontologies/preview`: Sirve de punto de entrada para la consulta sobre una ontología.
    - `/app/views/ontologies/search`: Muestra los resultados de una consulta sobre una ontología.
- `/config`: Contiene algunos archivos de configuración:
  - `/config/initializers/shared_cache`: Contiene la inicialización de la caché compartida de ontologías. Una de las limitaciones de Rails es que el servidor web que incluye por defecto no es multihilo por lo que si se desea lanzar la aplicación en un contenedor como Tomcat se puede acelerar las consultas configurando una caché que permite compartir

[3]La palabra *módulo* se emplea con mucha asiduidad en el ámbito de la programación y dado que un *módulo* es un mecanismo concreto en Ruby (conjunto de métodos genéricos susceptibles de ser incorporados a una clase en tiempo de ejecución) emplearemos la palabra inglesa *module* para distinguirlo del concepto general de *unidad funcional*.

las distintas ontologías y sus mapeados entre todos los hilos.

- `/config/routes`: Contiene un conjunto de rutas de acceso que relacionan los formatos de las URLs solicitadas con los controladores, acciones y parámetros que se deben invocar.

- `/engine`: Contiene Antive Core.
- `/repository`: Contiene las ontologías cargadas y sus archivos de mapeado generados por Antive.
- `/public`: Contiene las páginas estáticas, hojas de estilo CSS y scripts JavaScript.

5.3. Diseño detallado de componentes

En el capítulo anterior se hizo un diseño arquitectónico de los componentes que se emplearán en Antive. Una vez conocidos, además, los diferentes lenguajes y bibliotecas que se van a emplear, se procederá a estudiar de forma detallada cada uno de los componentes.

Aquí se abordarán los dos subsistemas por separado, dividiendo cada uno de ellos en las diferentes capas del modelo MVC.

5.3.1. Subsistema de consultas

Capa de modelo

El modelo del Subsistema de consultas es el componente Antive Core. Su función consiste en cargar los archivos XML que definen una ontología y sus correspondencias con las bases de datos externas y, mediante un API, ofrecer la posibilidad de consultar los recursos federados empleando elementos (clases y propiedades) de la ontología.

Los resultados que arroja Antive Core son una lista de recursos que se pueden entender como un conjunto de propiedades y sus valores.

Los archivos correspondientes a este componente se localizan en la carpeta `engine` y corresponde al conjunto de clases de Antive Core. En la sección 5.4 se aborda el API y el código de este componente.

Capa de vista

Esta capa se encarga de ofrecer al usuario un punto de entrada a la aplicación y mostrarle los resultados de las consultas. Está formada por los archivos ubicados en `app/views/search` y son:

- `preview.html.erb`: Genera una página XHTML que muestra los diferentes tipos de recursos definidos en la ontología.
- `search.html.erb`: Genera una página XHTML con los resultados de una consulta, que pueden ser:

1. Una lista de recursos.
2. Las propiedades y sus valores de un único recurso.

- `search.rdb.erb`: Genera un documento RDF que describe un recurso.

Capa de controlador

Cuando el usuario genera una consulta, lo hace mediante una URL. El controlador del subsistema de consultas toma la URL y la analiza para generar la consulta correspondiente. Así, haciendo uso del modelo se obtiene el conjunto de resultados que se pasan a la vista para representarlos en la página XHTML que el usuario obtiene.

Este componente está formado por el archivo `app/controllers/search_controller.rb` y tiene las siguientes acciones:

- `preview`: Es el punto de entrada a una ontología y se invoca al recibir una petición de la forma:

  ```
  GET /familia
  ```

  Esta acción busca la ontología llamada *familia*, la carga mediante el modelo e invoca la vista `preview.html.erb` quien genera la lista de clases de la ontología representada por el modelo.

- `search`: Se invoca al recibir una petición de alguna de las siguientes formas:

  ```
  GET /familia/personas
  GET /familia/personas/1
  GET /familia/personas/1/hijos
  ```

  En función de la forma de la URL se buscarán todos los recursos de la case `#Persona` o bien se filtrará por identificador. En ambos casos los resultados finales se pasan a la vista `search.html.erb`.

  - `search_all`: Se invoca cuando se obtiene una URL de la forma:

    ```
    GET /familia/personas
    ```

    Se obtiene el nombre de la clase correspondiente a la URL y sobre ella se ejecuta una consulta de tipo `find :all`

  - `search_by_id`: Se invoca cuando se obtiene una URL como:

    ```
    GET /familia/personas/1
    GET /familia/personas/1/hijos
    ```

    Se obtiene el nombre de la clase correspondiente y el identificador (o identificadores en caso de sea un identificador compuesto) y se realiza una consulta sobre Antive Core con la condición correspondiente.
    En el caso de la última URL, además de localizar el recurso (*personas/1*), se busca en él la propiedad *hijos* y se generará una nueva consulta para localizar su rango.

5.3.2. Subsistema de mapeado

Capa de modelo

De forma semejante a como ocurre en el subsistema de consultas, esta capa corresponde al componente Antive Core. Su utilidad en este subsistema no consiste en realizar consultas y localizar recursos, sino en obtener la estructura de la ontología que se pretende mapear en términos de la jerarquía de clases disponible y las propiedades de cada una de ellas.

Capa de vista

En este subsistema existen los siguientes componentes:

- Vista de administrador: Es el punto de entrada para la administración de Antive y consiste en una lista de las ontologías disponibles con las posibles acciones:
  - Añadir: Permite importar una nueva ontología en el repositorio y, opcionalmente, un XML de mapeado.
  - Borrar: Elimina una ontología del repositorio.
  - Mapear: Inicia la aplicación de mapeado.
  - Probar: Transfiere al administrador al Subsistema de consultas para que pueda probar la validez del mapeado.

  Este componente está implementado en los archivos siguientes:
  - `app/views/ontologies/index.html.erb`: Muestra la página de inicio de la vista de edición.
  - `app/views/ontologies/show.html.erb`: Muestra información de una ontología particular.
  - `app/views/ontologies/new.html.erb` y `app/views/ontologies/new_step_x.html.erb`: Muestra un conjunto de pasos en un asistente para cargar una nueva ontologia.
- Vista de edición: Es el punto de entrada a Antive Map y muestra el árbol de la ontología y los controles necesarios para crear o editar las correspondencias entre las clases y propiedades con los campos de las fuentes de datos. Este componente se localiza en el archivo `app/views/ontologies/edit.html.erb`
- Antive Map: Es un componente dependiente de la vista de edición y su función consiste en implementar el comportamiento de los controles disponibles en la vista de edición. Está localizado en varios archivos ubicados en `public/javascripts` y se abordará con mayor profundidad en la sección 5.5.2.

Capa de controlador

En esta capa se ubica el componente controlador de ontologías, encargado de la administración de las mismas en términos de altas y bajas y corresponde al archivo `app/controllers/ontologies_controller` Este controlador ofrece los siguientes métodos:

- `index`: Es el punto de partida para la administración de ontologías. Corresponde a la vista `index.html.erb`.
- `destroy`: Elimina una ontología del repositorio.
- `show`: Muestra información de una ontología. Corresponde a la vista `show.html.erb`.
- `edit`: Inicia la aplicación Antive Map. Corresponde a la vista `edit.html.erb` responsable de la carga de la aplicación JavaScript completa.
- `new` y `new_step_x`: Asistente para importar una nueva ontología. Corresponde a las vistas `new.html.erb` y `new_step_x.html.erb`, donde `x` es un valor que hace referencia a los diferentes pasos del asistente (1..5).

5.4. Desarrollo de Antive Core

En esta sección se realizará una exposición del código de Antive Core para exponer su funcionamiento interno y permitir así su estudio o extensión de cara a un futuro desarrollo.

5.4.1. Archivos

A continuación se muestra la lista de archivos que componen el Antive Core y una breve descripción de cada uno de ellos:

- **ansi_terminal.rb**: Provee de un conjunto de códigos ANSI para su empleo en una terminal de comandos.
- **data_source.rb**: Clase base para los controladores de fuentes de datos.
- **database_source.rb**: Encapsula la funcionalidad de consulta a bases de datos SQL.
- **hash.rb**: Extiende la funcionalidad de la clase `Hash` añadiendo un cálculo de un valor *hash*.
- **mapping.rb**: Define la clase `Mapping` encargada de leer y escribir el archivo de mapeado XML.
- **ontology.rb**: Define la clase `Ontology` encargada de parsear la definición XML de una ontología.
- **raise.rb**: Define el módulo Raise para lanzar excepciones con diferente grado de criticidad.
- **resource.rb**: Define la clase `Resource` que representa un recurso como resultado de una consulta.
- **semantic_class.rb**: Define la clase `SemanticClass` que representa una clase de una ontología.

- **semantic_exception.rb**: Define una subclase de `Exception` con nivel de criticidad.
- **semantic_property.rb**: Define la clase `SemanticProperty` que representa una *DatatypeProperty* de una ontología.
- **semantic_relation.rb**: Define la clase `SemanticRelation` que representa una *ObjectProperty* de una ontología.
- **semantic_thing.rb**: Define la clase `SemanticThing` que sirve de base al resto de clases `Semantic...` incorporando funcionalidades genéricas como transacciones y cachés.
- **transaction.rb**: Modela una transacción.

A continuación se verá el funcionamiento de cada uno de estos archivos, haciendo hincapié en las clases principales que contienen la funcionalidad de Antive Core.

5.4.2. Ontology

El punto de entrada de Antive es la clase `Ontology`. Su función consiste en tomar un archivo XML con la definición de una ontología y generar un conjunto de estructuras de datos (clases Ruby) que representen los objetos y las relaciones entre las clases de la ontología. En concreto, Antive Core emplea tres clases para representar una ontología:

1. `SemanticClass`: Representa una clase de una ontología.
2. `SemanticProperty`: Representa una propiedad de tipo *Datatype*.
3. `SemanticRelation`: Representa una propiedad de tipo *Object*.

Para parsear el XML de la ontología, `Ontology` emplea la biblioteca Jena que ofrece un API Java. `Ontology` adapta el API Java de Jena a un API característico de Ruby basado en iteradores mediante un conjunto de métodos `each_xxx`:

- `each_root_class`: Itera sobre las clases base de la ontología.
- `each_named_root_class`: Itera sobre las clases base no anónimas de la ontología.
- `each_class`: Itera sobre todas las clases de la ontología.
- `each_named_class`: Itera sobre las clases nombradas de la ontología.
- `each_anon_class`: Itera sobre las clases anónimas de la ontología.
- `each_child_for(klass`[4]`)`: Itera sobre las clases hijas de `klass`, siendo `klass` una representación de una clase obtenida por Jena.
- `each_property_for(klass)`: Itera sobre las propiedades de `klass`, siendo `klass` una representación de una clase obtenida por Jena.
- `each_property`: Itera sobre todas las propiedades de tipo *Datatype* de la ontología.
- `each_object_property`: Itera sobre todas las propiedades de tipo *Object* de la ontología.
- `each_domain_for(prop)`: Itera sobre las clases del dominio de una propiedad `prop`, siendo `prop` una representación de una propiedad ofrecida por Jena.

[4]Se usa la palabra *klass* ya que *class* es una palabra reservada de Ruby

- `each_range_for(prop)`: Itera sobre las clases en el rango de una propiedad `prop`, siendo `prop` una representación de una propiedad ofrecida por Jena.
- `each_operand_for(klass)`: Itera sobre los operandos de una clase `klass` de tipo unión, intersección.
- `each_restriction_for(prop)`: Itera sobre las restricciones de una propiedad `prop`, siendo `prop` una representación de una propiedad ofrecida por Jena.

Con este API, `Ontology` parsea la ontología mediante el método:

- `load_xml(xml)`: Recibe un XML de una ontología y genera una representación de las clases y propiedades de la misma,

que, a su vez, emplea las siguientes llamadas en este orden:

- `load_classes`: Genera un objeto de tipo `SemanticClass` por cada clase nombrada.
- `load_hierarchy`: Relaciona las `SemanticClass` entre sí en función de sus relaciones de herencia.
- `load_properties`: Genera una instancia de `SemanticProperty` por cada *DatatypeProperty* de la ontología y la enlaza desde las clases que la declaran.
- `load_object_properties`: Genera una `SemanticRelation` por cada *ObjectProperty* de la ontología y la enlaza desde las clases que la declaran.
- `load_object_property_relations`: Indica a cada *ObjectProperty* su propiedad inversa.

Una vez que `Ontology` ha parseado la ontología se puede acceder a cada una de la clases mediante la función

- `class_for(uri)`: Obtiene un objeto `SemanticClass` que representa la clase identificada por `uri`.

5.4.3. Representación de una ontología

En esta sección se verá mediante un ejemplo el conjunto de objetos que Antive Core genera para representar la ontología de la figura 5.1.

La representación de una ontología se realiza de la siguiente manera:

1. Cada clase de la ontología se representa mediante una instancia de `SemanticClass`.
2. Cada propiedad *Datatype* de la ontología se representa mediante una instancia de `SemanticProperty`.
3. Cada propiedad *Object* de la ontología se representa mediante una instancia de `SemanticRelation`.

Por otro lado, cada `SemanticClass` c_i contiene varios conjuntos de punteros:

1. `@parents` que contiene los punteros a las `SemanticClass` que representan las clases padre *directas* de c_i.
2. `@children` que contiene los punteros a las `SemanticClass` que representan las clases hijo *directas* de c_i.

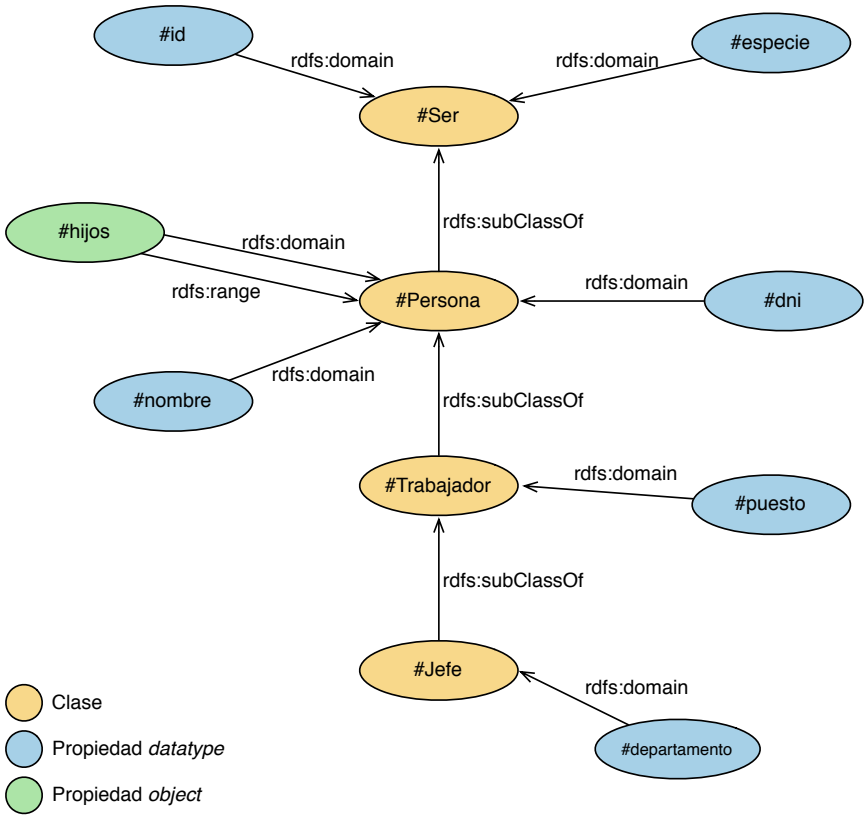

Figura 5.1: Ejemplo de ontología para el capítulo 5.4

3. `@properties` que contiene los punteros a los `SemanticProperty` que representan las *DatatypeProperties* declaradas en c_i.
4. `@relations` que contiene los punteros a los `SemanticRelation` que representan las *ObjectProperties* declaradas en c_i.

Así pues, la representación que `Ontology` hará de esta ontología es la mostrada en la figura 5.2

Los conjuntos de punteros se implementan mediante tablas *Hash* de forma que cada puntero puede obtenerse empleando como clave la URI del objeto al que apunta. Obsérvese, además, que las propiedades que contiene cada clase son las que dicha clase declara por lo que, para conocer el conjunto completo de propiedades de una clase (declaradas y heredadas) se emplea una búsqueda recursiva por el árbol de la jerarquía implementada por el método `SemanticClass.all_properties`.

El funcionamiento de Antive Core se basa en búsquedas recursivas. Uno de los inconvenientes a los que se enfrenta es la posibilidad de que una ontología pueda tener un ciclo en la jerarquía de clases, es decir, que una clase sea padre e hija *a la vez* de otra. Esta situación, que implica que las clases involucradas en el ciclo son equivalentes, daría lugar a una recursividad infinita. Para evitar este problema las clases cuentan con un mecanismo de protección mediante transacciones implementado en la clase base `SemanticThing`.

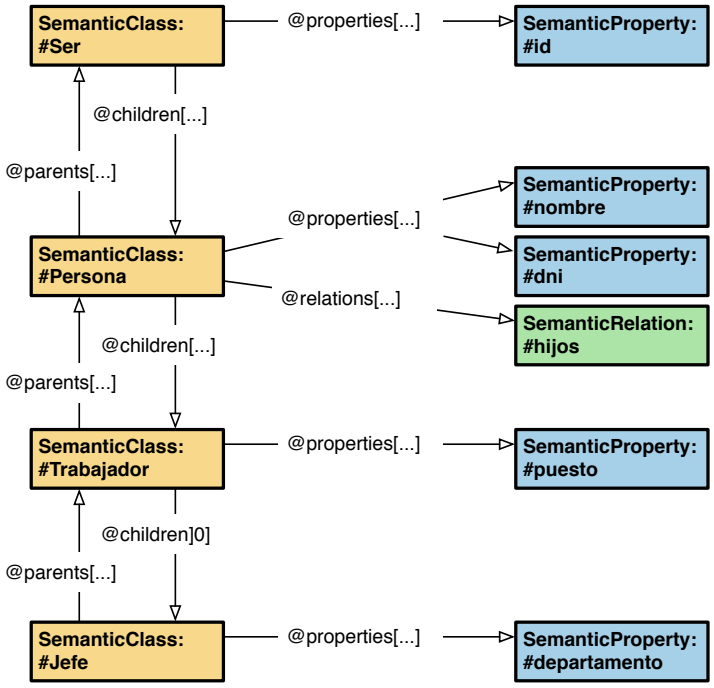

Figura 5.2: Representación interna de la ontología de la figura 5.1

5.4.4. Transacciones y cachés: `SemanticThing`

`SemanticThing` ofrece un API para el uso de transacciones que se detalla a continuación:

- `new_transaction(t)`: Es el punto de entrada de una transacción, siendo `t` la transacción en curso. Si `t` es `nil` se genera una nueva transacción. Opcionalmente se le puede pasar un bloque de código que se ejecutará en la clase que inicie la transacción, típicamente para instanciar el contenido de la transacción (normalmente un *array* o *hash*) que almacenará los resultados de la consulta.
- `accept_transaction(t)`: Marca una transacción como realizada en una clase.
- `did_transaction?(t)`: `true` si una transacción se ha realizado en una clase.
- `end_transaction(t)`: Finaliza una transacción. Opcionalmente se le puede pasar un bloque de código que se ejecutará en la clase que ha iniciado la transacción.

El mecanismo empleado para el uso de transacciones es el siguiente:

1. Cuando se inicia una consulta se genera una nueva transacción mediante el método `new_transaction`, representada por un objeto de tipo `Transaction`. Dicha transacción tiene como contenido un objeto que alojará los resultados de las consultas. Este contenido es accesible mediante el método `content` del objeto que representa la transacción.
2. En cada búsqueda el objeto de tipo `Transaction` creado se pasa como parámetro.
3. Al recibir una transacción se puede comprobar si ya se ha realizado anteriormente mediante el método `did_transaction?(t)`.

4. Si la transacción no se ha realizado se acepta mediante `accept_transaction`.
5. El resultado de la búsqueda se añade a `Transaction.content`.
6. Para finalizar la transacción se emplea `end_transaction` que recibe un bloque de código que se ejecutará únicamente en la clase que inició la transacción.

De esta manera:

1. Cuando *varias clases* se invocan mutuamente para realizar una consulta y presentan un ciclo (como un ciclo en la jerarquía de una ontología) el mecanismo de transacciones permite evitar los bucles infinitos.
2. Las consultas muy repetidas (como obtener todas las propiedades de una clase) pueden realizarse una vez y almacenar la transacción en una caché.

Nótese que este mecanismo permite evitar ciclos en una consulta en la que *intervengan varias clases*, ya que las transacciones se almacenan a nivel de clase. Si *una sola clase* realiza una consulta recursiva será necesario emplear una condición de salida que no sea `did_transaction`.

Para aprovechar la caché que `SemanticThing` ofrece, se emplea el siguiente API:

- `cache(id)`: Almacena o recupera una transacción etiquetada con el nombre `id`. Se le pasa un bloque de código que se ejecuta en caso de que la caché no contenga ninguna transacción con el nombre `id`.
- `save_in_cache(t, id)`: Almacena la transacción `t` con el nombre `id`.

Así, el uso de cachés se reduce a:

1. Se escoge un nombre para almacenar la transacción en la caché. Se recomienda emplear un objeto `Symbol`.
2. Se emplea el método `cache` con dos parámetros:
   a) El nombre de la transacción.
   b) Un bloque de código que realiza la consulta.
   Si la caché contiene una transacción con el nombre dado, `cache` devuelve el contenido almacenado. En caso contrario, se ejecuta el código de consulta.

En el apéndice B se muestra un código de ejemplo que muestra el uso del mecanismo de transacciones y cachés.

5.4.5. SemanticClass

`SemanticClass` representa una clase de una ontología y su función básica consiste en obtener los recursos a partir de las fuentes de datos considerando las relaciones dadas por la jerarquía de clases de la ontología. De esta forma, `SemanticClass` contiene la principal funcionalidad de Antive Core.

Para obtener el objeto que representa una clase de una ontología se hace uso del objeto `Ontology`:

```
onto = Ontology.new
xml = File.new(archivo_de_ontologia).read
onto.load_xml xml
persona = onto.class_for "http://..../Persona"
```

Listado 5.1: Cargando una ontología y obteniendo una de sus clases

API de búsqueda

El API de búsqueda de `SemanticClass` consiste en dos métodos:

- `find(*conditions)`: Obtiene los recursos de las fuentes de datos que cumplen las condiciones representadas por `*conditions`.
- `find_related(uri, resource)`: Obtiene los recursos de tipo `uri` que mantienen una relación con un recurso `resource` mediante una *ObjectProperty*.

Para localizar todos los recursos de una clase se emplea como condición el símbolo `:all`. A modo de ejemplo:

```
persona.find :all
```

Si se desea localizar los recursos que cumplan una condición se puede emplear la siguiente sintaxis:

```
condition = {
    :property => "...",
    :value => "..."
}
persona.find condition
```

Para fijar varias condiciones se emplea un conjunto de condiciones. A modo de ejemplo, el siguiente código localizaría todos los hombres de más de treinta años:

```
condition_1 = {
    :property => "http://.../sexo",
    :value => "H"
}
condition_2 = {
    :property => "http://.../edad",
    :value => "30",
    :operator => ">"
}
persona.find condition_1, condition_2
```

O bien se puede emplear un *array* que encapsule las condiciones, que será de utilidad si se generan programáticamente:

```
persona.find [ condition_1, condition_2 ]
```

`find` devuelve un *array* de recursos (aunque haya un único recurso) o bien `nil` en caso de no localizar ninguna coincidencia.

Por otro lado, `find_related` se emplea para obtener, a partir de un recurso, otros recursos relacionados mediante una *ObjectProperty*. A modo de ejemplo, creamos una consulta para obtener una persona que tenga por nombre Juan:

```
condition = {
    :property => "http://.../nombre",
    :value => "Juan"
}
juan = persona.find condition
```

Listado 5.2: Ejemplo de búsqueda con condiciones

`find` devuelve un *array* de recursos, y suponiendo que obtengamos un único recurso:

```
juan = juan.first
```

Para obtener los hijos de Juan empleamos `find_related` de la siguiente manera:

```
hijos = persona.find_related "http://.../hijos", juan
```

Como se verá en la sección 5.4.6 la clase `Resource` ofrece un mecanismo para simplificar esta búsqueda.

Algoritmo de búsqueda

Para entender el algoritmo de búsqueda se pondrá como ejemplo la ontología expuesta en la sección 5.4.3 y la siguiente consulta:

```
trabajador = onto.class_for "http://....#Trabajador"
condition = {
    :property => "http://...#nombre",
    :value => "alice"
}
trabajador.find condition
```

El método `find` tiene como función ejecutar una consulta por cada condición pasada como argumento y realizar una intersección de los conjuntos de resultados. El caso base, que es el que nos ocupa, es el que corresponde a una sola condición. Se genera una nueva transacción y se invoca `find_condition`.

Tal y como se comentó en la sección 5.4.2, una clase contiene una referencia a las propiedades que *declara*. En el código de Antive Core se dice que una clase c_i es *controladora* de una propiedad p_j si c_i declara la propiedad p_j. `SemanticClass` provee de un método `controller_for` que permite obtener fácilmente la clase controladora de una propiedad[5].

[5]Nótese que puede haber varios controladores para una propiedad, en el caso en que una clase sea hija de varias clases que declaren la misma propiedad.

La búsqueda se inicia en la clase controladora de `#nombre`, que es `#Persona`, mediante el método `find_condition_unique`[6]. Este método es el encargado de realizar varias tareas:

Reenviar la consulta a la propiedad correspondiente: Siguiendo con el ejemplo, la consulta se remite a un objeto de tipo `SemanticProperty` que representa la propiedad `#nombre` y que contiene un *adaptador* para la fuente de datos. Este *adaptador* es una subclase de `DataSource` y realiza la tarea de adaptar la consulta específica (SQL para bases de datos, XML para servicios web, etc.) a una interfaz genérica. Actualmente Antive Core cuenta únicamente con un adaptador para bases de datos SQL (`DatabaseSource`).

Una vez obtenida la respuesta, la propiedad `#nombre` la transforma en un conjunto de objetos de tipo `Resource` que representa el recurso final de la consulta.

Completar los recursos: Una de las principales características de Antive consiste en que las propiedades de un recurso pueden estar distribuidas en diferentes bases de datos. `SemanticClass` permite buscar la información de un recurso recorriendo la jerarquía de clases y las distintas bases de datos relacionadas con cada una de ellas. En el código de Antive Core esta actividad se denomina *resolver*.

La *resolución* de un recurso comienza en el método `resolve`, quien inicia el algoritmo de resolución implementado por dos métodos:

1. `resolve_new_kernel`: Recorre la jerarquía de clases relacionadas con un recurso r_i y en cada clase c_i se obtiene un conjunto de pares *propiedad-valor* $\{p_i, v_i\}$ en los que p_i es una propiedad *identificador* y v_i es su valor. De esta forma, en cada clase de la jerarquía, el algoritmo cuenta con el conjunto de identificadores que se pueden emplear en una consulta a una base de datos para recuperar información adicional del recurso.
2. `resolve_new_kernel_lookup`: Realiza una consulta en una fuente de datos localizando el recurso que se desea resolver, añadiéndole las propiedades no resueltas que se hayan recuperado de la fuente.

Reenviar la consulta a las subclases: Al realizar una consulta sobre una clase c_i con una condición dada, se transfiere la consulta a las subclases de c_i ya que cualquier recurso perteneciente a una subclase c_j que cumpla la condición de la consulta será a su vez un resultado válido de la consulta sobre c_i.

Descartar recursos no válidos: En determinadas condiciones, cuando una clase realiza una consulta, se pueden obtener resultados duplicados que deben ser descartados. `SemanticClass` realiza este descarte antes de continuar el ciclo de consultas.

A continuación se mostrarán un ejemplo que ilustrará el funcionamiento general del algoritmo.

[6]Si no hay un controlador único se ejecuta la consulta en cada uno de los controladores y se agregan los resultados.

Ejemplo

A continuación se muestra un ejemplo de consulta que ilustra el funcionamiento de `SemanticClass`, de la búsqueda a través de la jerarquía y de la *resolución* de recursos.

Se parte de las siguientes bases de datos:

Personas

#DNI	#Nombre
100	Juan
200	Montse
300	Alicia
400	Soledad
500	Juan

Trabajadores

#DNI	#Sueldo
100	1000
400	4000
500	5000
600	6000

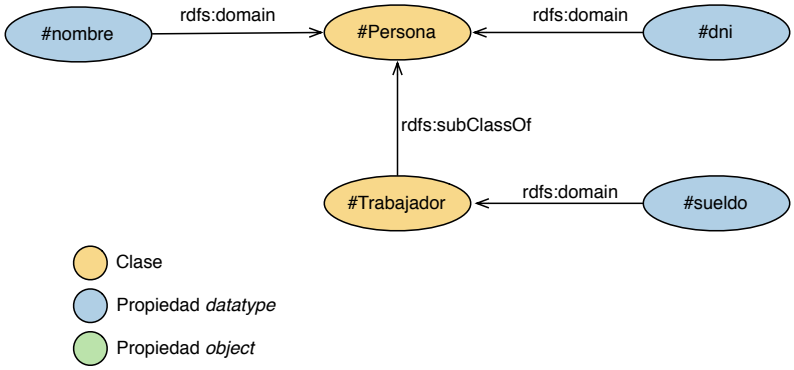

Figura 5.3: Jerarquía de clases para el ejemplo de la sección 5.4.5

La jerarquía de clases de la ontología se muestra en la figura 5.3. Se supone la siguiente búsqueda:

```
persona = onto.class_for "http://....#Persona"
condition = {
    :property => "http://...#DNI",
    :value => "100"
    :operator => ">"
}
persona.find condition
```

La consulta hace referencia a la propiedad `#DNI` que está declarada en `#Persona`. De la tabla correspondiente se obtienen los recursos siguientes:

#DNI	#Nombre
200	Montse
300	Alicia
400	Soledad
500	Juan

En este momento cada uno de estos recursos es de tipo `#Persona`, por lo que se tipan como tal:

#DNI	#Nombre	Clases
200	Montse	#Persona
300	Alicia	#Persona
400	Soledad	#Persona
500	Juan	#Persona

Tras ello se toman los recursos obtenidos uno por uno y se *resuelven*, es decir, se añaden las propiedades de las clases padre e hijas. Al ser `#Persona` una superclase el proceso comenzará por `#Trabajador`. La `SemanticClass` que corresponde a `#Trabajador` toma el recurso a resolver, obtiene los identificadores (en este caso `#DNI`) y realiza una búsqueda en sus bases de datos relacionadas.

Los recursos con DNI 200 y 300 no existen en la base de datos de trabajadores por lo que para ellos no se resuelve ninguna propiedad; sin embargo, los recursos con DNI 400 y 500 existen en esta base de datos, por lo que puede obtenerse el sueldo de cada uno de ellos. En este caso, los recursos 400 y 500 serán también de tipo `#Trabajador`:

#DNI	#Nombre	#Sueldo	Clases
200	Montse		#Persona
300	Alicia		#Persona
400	Soledad	4000	#Persona, #Trabajador
500	Juan	5000	#Persona, #Trabajador

La consulta sobre la tabla correspondiente a `#Persona` ha concluido, y todos los recursos obtenidos son de tipo `#Persona`, por lo que no se realiza ningún descarte. Este conjunto de resultados se denominará *resultados parciales 1*.

La consulta se transfiere a las subclases de `#Persona`, que en este caso será `#Trabajador`, para obtener los siguientes resultados:

#DNI	#Sueldo	Clases
400	4000	#Trabajador
500	5000	#Trabajador
600	6000	#Trabajador

Estos resultados se resuelven consultando a los padres y los hijos de `#Trabajador`. En este caso la clase `#Persona` toma los recursos y trata de resolverlos añadiendo la propiedad `#Nombre` a aquellos cuyos identificadores existan en la base de datos de personas. Se obtiene el siguiente conjunto de recursos:

#DNI	#Nombre	#Sueldo	Clases
400	Soledad	4000	#Trabajador, #Persona
500	Juan	5000	#Trabajador, #Persona
600		6000	#Trabajador

Todos estos resultados son de tipo `#Trabajador`, por lo que no hay descarte, obteniendo así el conjunto de *resultados parciales 2*. Este conjunto se agrega a los *resultados parciales 1*, eliminando

duplicados, para obtener así el conjunto de resultados final:

#DNI	#Nombre	#Sueldo	Clases
400	Soledad	4000	#Persona, #Trabajador
500	Juan	5000	#Persona, #Trabajador
600		6000	#Trabajador

Nótese que el recurso con DNI 600 se marca como de tipo `#Trabajador` y no de tipo `#Persona`, lo cual no tiene sentido ya que en esta jerarquía todo `#Trabajador` es `#Persona`. Debe entenderse este tipado como un mecanismo para realizar el algoritmo (es el conjunto de clases involucradas en la resolución de un recurso) y no como un tipado semántico.

Consideraciones sobre el algoritmo de búsqueda

1. `SemanticClass` también trabaja con herencia múltiple, trasladando las operaciones a los conjuntos de clases padre e hijas.
2. Clases anónimas: `Ontology` traduce las clases anónimas de forma que si una propiedad se declara en una clase anónima se traslada a las clases que la forman. De esta forma, si una consulta es sobre una propiedad declarada en una clase anónima se traslada la consulta a todas las clases que la declaran.
3. Se contemplan las propiedades multievaluadas: Si dos bases de datos contienen diferentes valores que se corresponden con una misma propiedad, el recurso que se obtiene contiene un *array* de valores para la propiedad multievaluada.

5.4.6. Resource

`Resource` modela un recurso como resultado de una consulta. Un recurso es un agregado de propiedades y valores donde las propiedades se especifican mediante la jerarquía de clases de una ontología y los valores se obtienen mediante consultas a diferentes bases de datos.

El resultado de realizar una consulta es un conjunto de instancias de tipo `Resource`[7] que ofrecen el siguiente API:

- `find(uri)`: Encapsula una llamada a `find_related` de `SemanticClass` de forma que la consulta

```
hijos = persona.find_related "http://.../hijos", juan
```

puede simplificarse como:

```
hijos = juan.find "http://.../hijos"
```

Listado 5.3: Ejemplo de búsqueda de *ObjectProperties*

[7]Esta clase no está relacionada con la clase Resource de RDF

- `get(prop, klass = nil)`: Obtiene los valores de la propiedad *prop*. Si la propiedad se declara en varias clases (empleando una clase anónima de tipo unión) se puede restringir la consulta a una de las clases que la forman, lo cual puede resultar útil en caso de herencia múltiple:

```
# obtiene todos los nombres de Juan
puts juan.get("http://.../nombre")

# si la propiedad nombre se define en varias superclases se
# puede restringir la consulta a una de ellas
puts juan.get("http://.../nombre", "http://.../Persona")
```

Listado 5.4: Obteniendo propiedades de un recurso

- `is_a?(klass)`: Evalúa si el recurso es de tipo *klass*.
- `each`: Itera sobre cada una de las propiedades del recurso, ofreciendo además la clase donde se han definido:

```
juan.each do |klass, property, value|
    puts "#{property} = #{value}"
end
```

- `small_print`: Muestra por STDOUT el contenido del recurso. Puede ser muy útil a la hora de realizar un desarrollo sobre Antive Core o para ampliar su funcionalidad.

5.4.7. Fuentes de datos: DatabaseSource

Para que Antive pueda federar diferentes tipos de fuentes de datos (bases de datos relacionales, servicios web, etc) se debe implementar un *adaptador* específico para cada uno de ellos que presente un API general para la consulta. En este proyecto se ha implementado un adaptador para tablas de bases de datos SQL implementado en la clase `DatabaseSource`.

`DatabaseSource` está basado en la gema ActiveRecord [Marshall et al., 2007] que permite el acceso a diferentes bases de datos de forma transparente. A su vez emplea diferentes adaptadores para generar SQL de diferentes motores (MySQL, PostgreSQL, Oracle, etc.).

El API que se debe implementar es el siguiente:

- `connect`: Conectarse a la fuente de datos. Los parámetros necesarios para la conexión son propios de la fuente, por lo que deben pasarse al adaptador empleando un API propio.
- `fields`: Obtiene los campos de los registros de la fuente de datos. En una base de datos, los campos corresponden a las columnas de una tabla.
- `find(field, operator, value)`: Realiza una búsqueda con una condición definida por un campo, un operador y un valor. Las condiciones de los ejemplos vistos anteriormente se traducen en esta consulta, como se muestra a continuación:

```
condition = {
    :property => "http://.../sueldo",
```

```
        :operator => ">",
        :value => 1000
    }
    trabajadores = trabajador.find condition
```

Esta consulta se traduce en primera instancia a:

```
    database_source.connect
    database_source.find "sueldo", ">", 1000
```

- `find_conditions(conditions)`: Realiza una consulta con un conjunto de condiciones especificadas mediante una *hash* de la forma:

```
    database_source.find_conditions {"sueldo" => 1000, "nombre" => "Juan"}
```

- `find_all`: Obtiene todos los registros de una fuente de datos.

Los resultados obtenidos se deben devolver mediante un *array* en los que cada elemento (que corresponde a un registro) se modela mediante una *hash* de la forma:

```
registro = {
    campo1 => valor1,
    campo2 => valor2,
    ...
}
```

Si se desea crear un nuevo adaptador para una fuente de datos nueva basta con crear una clase que herede de `DataSource` e implementar estos métodos convenientemente, haciendo uso de cualquier otra clase que se requiera (conexiones TCP, HTTP, etc.)

5.4.8. SemanticProperty

Se ha visto que la clase `SemanticClass` se encarga del algoritmo de búsqueda y resolución de resultados y que las consultas a las fuentes de datos se realizan en las clases adaptadoras como `DatabaseSource`. `SemanticClass` trabaja, por tanto, con clases, propiedades y herencia, mientras que los accesos a las fuentes de datos trabajan con registros, campos y valores. Ambos mundos se relacionan entre si mediante la clase `SemanticProperty`.

Cuando `SemanticClass` decide que hay que realizar una consulta sobre una propiedad (por ejemplo `#nombre` es "Juan"), esta consulta se envía a la `SemanticProperty` que representa la propiedad `#nombre`. Esta clase tiene acceso tanto a las propiedades de la clase *llamante* como a los campos de la fuente de datos que se debe consultar, por lo que realiza la tarea de traducción de unos a otros. El API que presenta es el siguiente:

- `find(condition, sender)`: Realiza una consulta del tipo *propiedad-operador-valor* enviada por `sender`, que es un objeto de tipo `SemanticClass`. `SemanticProperty` obtiene la información necesaria para traducir la consulta al API del adaptador de la fuente de datos, invoca la consulta y genera un conjunto de recursos `Resource` por cada registro obtenido.

- `find_conditions(conditions, sender)`: Es idéntico al anterior pero permite realizar una búsqueda con varias condiciones, empleando para describir las condiciones una *hash* de la forma:

```
property.find_conditions {"sueldo" => 1000, "nombre" => "Juan"}, self
```

- `find_all(sender)`: Obtiene todos los registros de la fuente de datos.

Antive permite que cada clase de una ontología se corresponda a un conjunto de fuentes de datos, de forma que diferentes bases de datos pueden corresponderse a una única clase. Así pues, la tarea de consultar a cada una de esas fuentes, así como de federar los resultados se delega en `SemanticProperty` y se realiza de forma transparente para `SemanticClass`. Esta tarea se lleva a cabo en el método `merge` de `SemanticProperty`.

5.4.9. SemanticRelation

Hemos visto que `SemanticProperty` sirve de adaptación entre una `SemanticClass` y una fuente de datos, modelando así el acceso a una *DatatypeProperty* de una clase. `SemanticRelation` realiza una tarea semejante pero modelando una *ObjectProperty*, es decir, adaptando una `SemanticClass` a otra `SemanticClass`.

La tarea de `SemanticRelation` es, por ello, tomar un recurso r_i y obtener los recursos $\{r_j\}$ que se relacionen con r_i mediante una *ObjectProperty*. Para ello, `SemanticRelation` ofrece el método:

- `find_destination_with(resource, sender)`: Obtiene el conjunto de recursos relacionados con *resource*.

`SemanticRelation` distingue entre dos casos: relaciones directas o indirectas:

- En las relaciones directas el recurso *destino* tiene el identificador del recurso *origen*, por lo que puede obtenerse el *destino* con tan solo conocer el identificador del *origen*. En un modelo entidad-relación, estas relaciones son del tipo 1:N y se implementan en el esquema relacional mediante una clave ajena, como en el siguiente ejemplo:

`Bitácoras`

id	Nombre
1	...
2	...

`Artículos`

id	blog_id	Título
1	1	...
2	1	...
3	2	...
4	2	...
5	2	...

  Se observa que la relación es de tipo 1:N (una bitácora tiene N artículos) y que, para determinar esta relación, cada artículo cuenta, además de con un identificador propio, con la clave de la bitácora a la que pertenece.
- En las relaciones indirectas existe una fuente de datos intermedia que contiene los pares de

identificadores *origen* y *destino*. Las relaciones N:N se modelan típicamente así, aunque puede emplearse este mismo esquema en relaciones 1:N o 1:1 sin ser estrictamente necesario. A modo de ejemplo, se muestran las siguientes tablas que modelan una relación N:N entre artículos y autores:

Articulos

id	Título
1	...
2	...
3	...

Autores

dni	Nombre
10001	...
20002	...
30003	...

articulos_autores

articulo_id	autor_dni
1	10001
1	30003
2	10001
3	20002
3	30003

Se observa que la relación es de tipo N:N ya que un artículo puede estar escrito por varios autores, y un autor puede escribir varios artículos.

Cada uno de estos casos se aborda en los métodos:

- `find_direct_destination_with(resource, sender)`
- `find_indirect_destination_with(resource, sender)`

Estos métodos se encargan de obtener un conjunto de condiciones que se emplean en una nueva consulta a `SemanticClass`. En el caso de la relación indirecta se realiza una consulta intermedia directamente sobre la fuente de datos para conocer los pares de claves a emplear.

5.5. Desarrollo de la interfaz web

En esta sección se explicará el código de la interfaz web con el fin de entender el diseño de esta parte de la aplicación y poder, así, depurarla, modificarla o extenderla para añadir funcionalidades.

La interfaz web se divide en dos partes:

1. Un esquema Ruby On Rails que sirve de base para la toda aplicación.
2. El componente Antive Map que implementa el comportamiento de la interfaz de mapeado.

En las siguientes secciones se verá cada una de estas partes con detenimiento.

5.5.1. De URL a consultas

Cuando se accede a la aplicación con un navegador web, el motor de Rails analiza la URL siguiendo los patrones de `config/routes` y determina el controlador, la acción y los parámetros que deben ejecutarse. El archivo `routes` no contiene el conjunto completo de rutas sino una *descripción* de las mismas. A modo de ejemplo:

```
map.resources :controlador
```

genera un conjunto de rutas REST sobre el controlador. El conjunto final de rutas se puede conocer empleando el comando de terminal:

```
$ rake routes
```

En este caso existen dos posibilidades:

1. Que la ruta corresponda a una acción de gestión de ontologías.
2. Que la ruta corresponda a una consulta.

En el primer caso se invocará el controlador y la acción correspondientes, por ejemplo mostrar las ontologías disponibles o acceder a la interfaz de mapeado. El segundo caso corresponde a una URL de una consulta, que a modo de ejemplo podría ser como:

```
http://..../publicaciones/autores/
http://..../publicaciones/autores/5
http://..../publicaciones/autores/5/articulos
```

Listado 5.5: URLs de consulta para Antive Map

El mecanismo para traducir cada una de estas URLs se basa en que cada clase de una ontología cuenta con una *URL de acceso*. Siguiendo el ejemplo anterior, para la clase `#Autor` se especifica la siguiente URL:

```
http://.../publicaciones/autores/[http://.../entities#id]
```

Listado 5.6: Ejemplo de URL pública

Nótese que `http://.../enities#id` es la URL de la propiedad `#id` de `#Autor`, de forma que cada recurso de tipo `#Autor` tiene una URL única:

```
http://..../publicaciones/autores/1
http://..../publicaciones/autores/2
http://..../publicaciones/autores/5
...
```

Comparando la URL de consulta con las URLs de acceso de las clases de la ontología se pueden distinguir los tres casos:

1. `http://..../publicaciones/autores/`: Al no especificar ningún identificador, esta URL corresponde a una consulta de *todas las instancias*. Al comprobar que la URL de acceso de `#Autor` encaja en la URL de consulta, ésta se realiza sobre dicha clase. Esta URL se traduce a la consulta sobre Antive Core siguiente:

   ```
   results = class_for("http://.../entities#Autor").find(:all)
   ```

2. `http://..../publicaciones/autores/5`: Es un caso semejante al anterior, pero al existir un identificador se puede traducir la consulta a:

   ```
   condition = {
   ```

```
        :property => "http://.../enities#id",
        :value => "5"
    }
    result = class_for("http://.../entities#Autor").find(condition).first
```

3. `http://..../publicaciones/autores/5/articulos`: De forma semejante al caso anterior, se realiza la consulta sobre la clase `#Autor` para obtener el recurso identificado cuya propiedad `#id` tenga como valor `5`. Una vez localizado el recurso se analizan sus *ObjectProperties* para encontrar la propiedad nombrada `articulos`. Ya que esta propiedad es de tipo `#Articulo`, la consulta se traduce a:

```
    condition = {
        :property => "http://.../enities#id",
        :value => "5"
    }
    resource = class_for("http://.../entities#Autor").find(condition).first
    results = resource.find("http://.../entities#Articulo")
```

   El resultado de esta consulta es un conjunto de recursos cuyas URLs serán de la forma:

```
    http://..../publicaciones/articulos/102
    http://..../publicaciones/articulos/291
    ...
```

Para conocer más sobre el funcionamiento de las rutas y la implementación de REST en Rails se recomienda leer el tutorial *Desarrollo REST con Rails* [Wirdemann and Baustert, 2007].

5.5.2. Interfaz para el mapeado

El objetivo de Antive Map es permitir al administrador relacionar los campos de las diferentes fuentes de datos con las propiedades de las clases de la ontología de una forma sencilla, muy visual e intuitiva. Para ello se ha optado por construir todo el componente en JavaScript de forma que todas las acciones se realicen de forma local, sin comunicación con el servidor, evitando así una carga innecesaria del mismo[8].

Para crear una interfaz de usuario rica se han empleado las bibliotecas *Prototype* y *Scriptaculous*.

Al iniciarse esta interfaz se genera una representación intermedia de la ontología que incluye tanto las clases y propiedades de la misma como la información del mapeado. Esta representación intermedia se basa en las estructuras básicas de JavaScript: *arrays* y tablas *hash*. Una vez el usuario finaliza su labor se genera un XML de mapeado a partir de la representación intermedia de las estructuras de datos.

Los archivos que forman este componente se sitúan en `/public/javascripts` y son:

[8]Se realizan algunas peticiones AJAX al servidor, pero no se establece ninguna sesión entre ambas partes.

- `application`: Incluye funciones generales para la aplicación como la generación del XML así como definiciones de variables globales.
- `edition_classes.js`: Incluye las funciones relacionadas con el manejo de las clases de una ontología: crear clases, añadirle propiedades, interpretar los eventos de arrastrar y soltar entre clases, etc.
- `edition_properties.js`: Incluye las funciones relacionadas con el manejo de las propiedades de las clases: crear propiedades, añadirles un mapeo desde una fuente de datos, interpretar acciones de arrastrar y soltar, etc.
- `edition_sources.js`: Incluye las funciones relacionadas con el manejo de fuentes de datos: altas, bajas, datos de conexión.

El punto de entrada para este componente es la vista `app/views/ontologies/edit.html.erb`. Esta vista genera una página HTML con una estructura de `divs` y ejecuta un script JS que genera, a partir de las estructura de datos de Antive Core, la representación intermedia de los mismos. Tras ello se genera el conjunto de elementos de la interfaz (enlaces y `divs` arrastrables) encargados de generar el conjunto de eventos que disparan los diferentes estados de la aplicación.

Estructuras de datos

La principal dificultad para enfrentarse al código de este componente consiste en conocer la representación intermedia del conjunto de clases y el papel de los *arrays* y *hashes* en ella. La figura 5.4 muestra las relaciones entre arrays y hashes que se emplean para representar las clases, sus propiedades *datatype* y las fuentes de datos.

Por otro lado las propiedades de tipo Object se representan mediante el conjunto de estructuras de datos representadas en la figura 5.5.

Para ayudar al desarrollador a entender el funcionamiento de esta representación intermedia, el apéndice C contiene unos ejemplos de código JavaScript que hacen uso de estas estructuras de datos.

Prototype y script.aculo.us

Antive Map hace un uso muy intensivo de dos bibliotecas Javascript que facilitan la creación de interfaces gráficas ricas.

Prototype: Es una biblioteca que permite realizar aplicaciones web dinámicas de forma fácil y mantenible. En Antive Map el uso de Prototype se centra básicamente en dos tareas:

1. Manipulación del DOM: La función `$(...)` permite hacer referencia a los diferentes objetos del DOM mediante el identificador único del mismo. A modo de ejemplo:

```
<div id='titulo'></div>
```

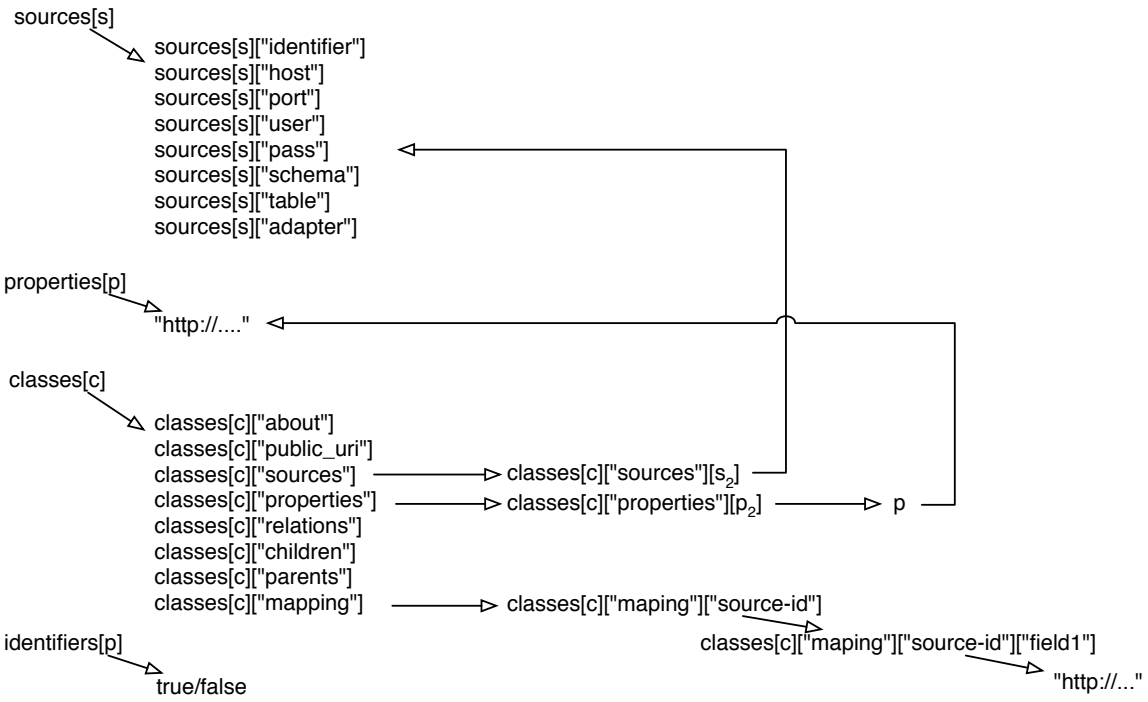

Figura 5.4: Representación intermedia de clases y *DatatypeProperties* en Antive Map

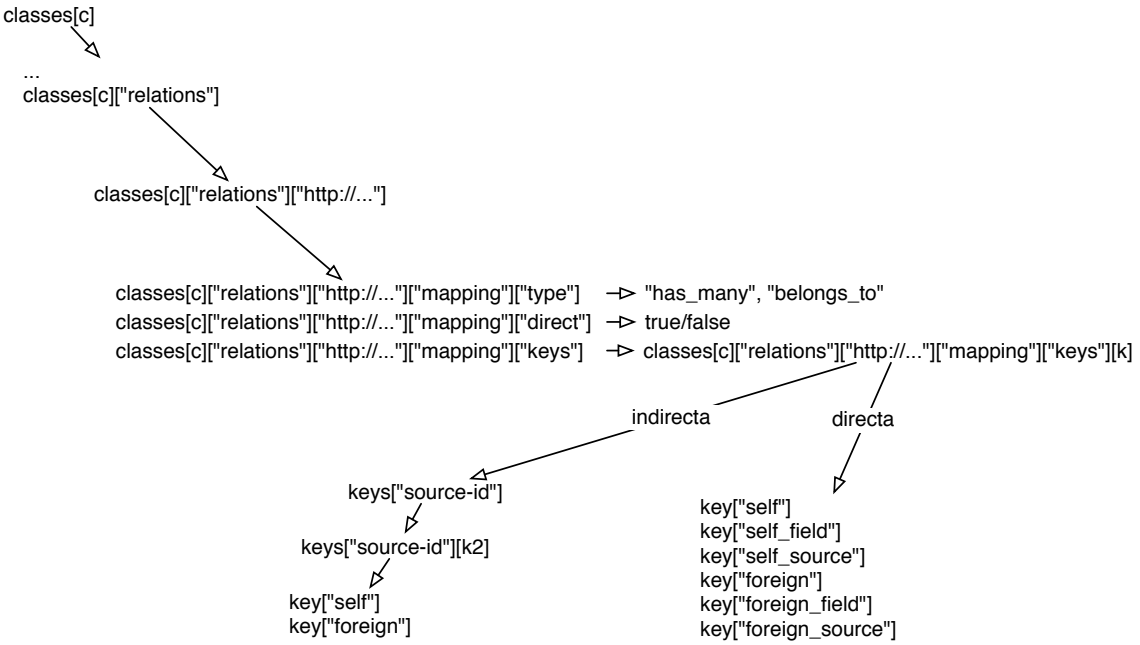

Figura 5.5: Representación intermedia de *ObjectProperties* en Antive Map

```
<div id='autor'></div>
<div id='comentarios'></div>
```

De esta manera se puede crear contenido en tiempo de ejecución de forma sencilla:

```
$("titulo").innerHTML = "El Ciclo Barroco";
$("autor").innerHTML = "Neal Stephenson";
```

Otra técnica muy empleada para crear contenido en un elemento del DOM consiste en las inserciones, de forma que se *añade* contenido al nodo:

```
div = $("comentarios");
new Insertion.Bottom(div, "...");
new Insertion.Bottom(div, "...");
new Insertion.Bottom(div, "...");
```

También es posible añadir o eliminar clases CSS a los nodos del DOM:

```
$("titulo").addClassName("actve");
$("titulo").removeClassName("inactve");
```

2. AJAX: Prototype permite el uso de peticiones AJAX de forma homogénea (*cross-browser*) y sencilla:

```
body = "...";
url  = "...";
new Ajax.Request(url, {
    method: 'post',
    postBody: body,
    asynchronous: true,
    onComplete: function(request) {
        alert("Fin de la peticion AJAX:" + request.responseText);
    }
});
```

Listado 5.7: Ejemplo de AJAX con Prototype

Script.aculo.us: Es una biblioteca que permite crear interfaces gráficas de usuario atractivas y ricas gracias a animaciones, transiciones, mecanismos de arrastrar y soltar, autocompletado de campos, etc. Los principales usos que Antive Map hace de esta biblioteca son:

1. Arrastrar y soltar son acciones intensamente empleadas en Antive Map. A modo de ejemplo se muestra el siguiente código:

```
<div id="drag_this"> ... </div>
<div id="drop_here"> ... </div>
...
new Draggable("drag_this");
Droppables.add("drop_here"), {
```

```
    onDrop:function(draggable) {
        ...
    }
});
```

Listado 5.8: Uso de arrastrar y soltar con script.aculo.us

2. Animaciones. Aunque deben emplearse con cuidado, las animaciones pueden llegar a ser muy útiles para transmitir cierta información. A modo de ejemplo, para cambiar la opacidad de un elemento se puede emplear el código siguiente:

```
new Effect.Opacity($("..."), { to:0.1 });
```

Capítulo 6

Ejemplos de uso

En este capítulo se verán unos ejemplos del uso de Antive con los que se podrá, por un lado, aprender a usar su interfaz web y, por otro, comprobar el funcionamiento del motor de federación.

Para facilitar la notación se empleará de aquí en adelante el siguiente conjunto de reglas:

1. Todos los elementos de la ontología tendrán la misma URL base (`http://it.uc3m.es/entidades`), por lo que las clases y propiedades se denominarán con la forma abreviada `#Clase`, `#propiedad`.
2. El nombre de una clase comenzará en mayúscula y el de una propiedad en minúscula.

6.1. Representando una bitácora

Las bitácoras o *blogs* se han convertido en el *Hola Mundo* de la programación web. Su funcionamiento, relativamente sencillo, y la necesidad de registrar datos y relaciones lo convierten en un buen punto de partida para este tipo de programación. En este primer ejemplo se empleará una ontología para consultar la base de datos de un sistema de bitácoras.

Las claves que se mostrarán en este ejemplo son:

1. Uso general de Antive.
2. Correspondencia entre campos de una base de datos y propiedades de tipo dato (*datatype*).
3. Mapeo de propiedades de tipo objeto.

6.1.1. Ontología

Para este ejemplo se usará una ontología muy sencilla mostrada en la figura 6.1; en ella participan dos clases:

- `#Articulo`: Representa un articulo y tiene tres propiedades:
  - `#id`: Un identificador único de tipo entero.
  - `#titulo`: Una cadena de caracteres que corresponde al título del artículo.

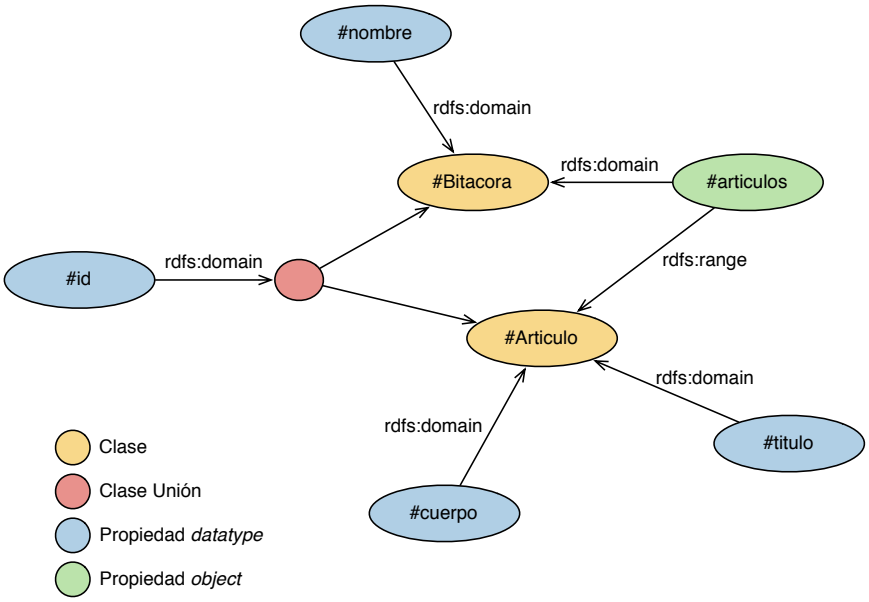

Figura 6.1: Ontología para el ejemplo 6.1

- `#cuerpo`: Una cadena de caracteres que corresponde al cuerpo del artículo.

▪ `#Bitacora`: Representa una bitácora y tiene tres propiedades:

- `#id`: Un identificador único de tipo entero.
- `#nombre`: Una propiedad *datatype* de tipo `String` que corresponde al nombre de la bitácora.
- `#articulos`: Una propiedad *object* que corresponde al conjunto de artículos de la bitácora.

6.1.2. Base de datos

La figura 6.2 muestra el diagrama entidad-relación de la base de datos de un sistema de publicación de bitácoras.

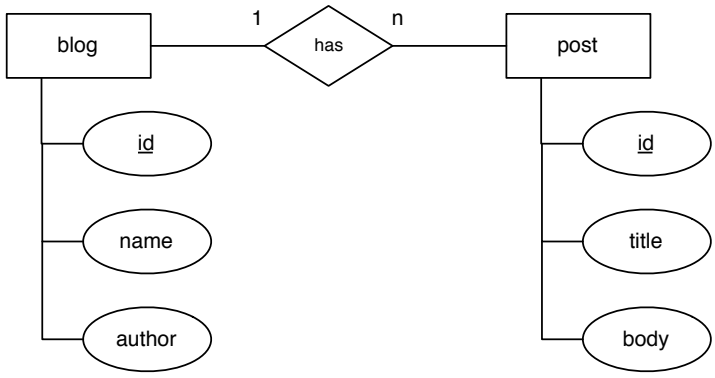

Figura 6.2: Modelo ER el ejemplo 6.1

Se poblará la base de datos con los siguientes registros:

blogs

id	name	author
1	microsiervos	Álvaro
2	planeta gul	Varios

posts

id	blog_id	title	body
1	1	Bienvenido a microsiervos	...
2	1	Canon 400D	...
3	1	Nintendo wii	...
4	2	Debian Lenny	...
5	2	Nueva Debian Lenny	...

6.1.3. Importando la ontología en Antive

Se accede a la página de administración de Antive accediendo a la dirección:

```
http://servidor:puerto/ontologies
```

En este caso el servidor será `localhost` y el puerto `3000`. Tras la autenticación se accede a la página de administración:

Figura 6.3: Página de administración de ontologías

Al pulsar en `Nueva ontología` se accede a un asistente que ayuda a importar la nueva ontología. Los pasos que se seguirán son:

1. Definir un nombre para la ontología, que será `publicaciones`.
2. Cargar el archivo OWL de la ontología.
3. Indicar que **no** se posee un archivo de mapeado.

A finalizar el asistente se accederá de nuevo a la página de administración para comprobar que la nueva ontología se ha cargado:

Figura 6.4: Página de administración de ontologías con una ontología cargada

6.1.4. Usando Antive Map

Si se pulsa en el título de la ontología se accede al árbol de clases (situado en la parte izquierda de la figura 6.5 y a un conjunto de controles (situados en la parte derecha de la misma figura):

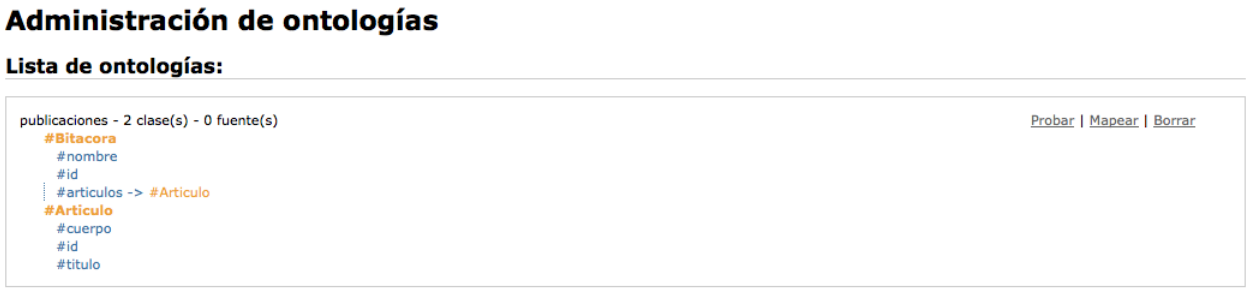

Figura 6.5: Página de administración de ontologías, controles

Para comenzar el mapeado se pulsa en `Mapear` y se accede a la interfaz de mapeado. Como muestra la figura 6.6, en esta página se observa:

1. A la izquierda, el árbol de clases y sus propiedades. Junto a cada propiedad se muestra un indicador rojo que muestra que no hay ninguna correspondencia definida.
2. A la derecha se muestra la lista de fuentes disponibles, que en este momento está vacía, y un enlace `Nueva fuente`.

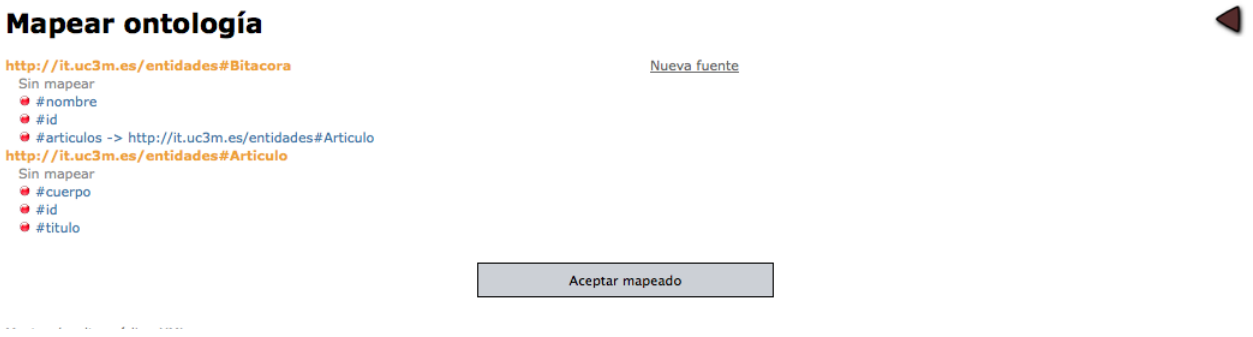

Figura 6.6: Página de Antive Map

Lo primero que se hará será crear las fuentes de datos. Se dispone de dos fuentes que corresponden a cada una de las tablas de la base de datos. Se comenzará por la tabla `blogs`; al pulsar

sobre `Nueva fuente` aparece una lista de parámetros que se debe rellenar:

Figura 6.7: Añadiendo una nueva fuente de datos

Una vez introducidos los parámetros de conexión se pulsa `Aceptar` y la nueva fuente aparece en la lista de fuentes disponibles. Se realiza el mismo procedimiento para crear una nueva fuente para la tabla `posts`:

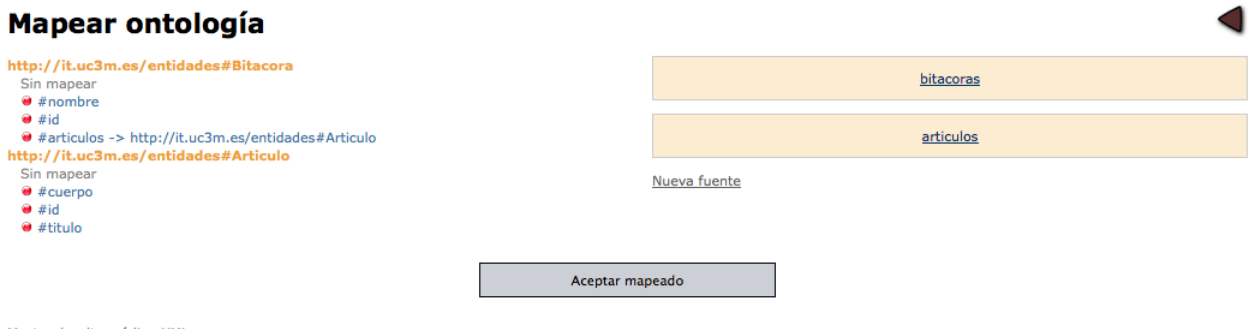

Figura 6.8: Fuentes de datos disponibles

En caso de que haya que cambiar algún dato de conexión se puede pulsar el título de la fuente para editarla:

Figura 6.9: Editando una fuente de datos

6.1.5. Mapeando propiedades *Datatype*

A continuación se hará corresponder los campos de la tabla `bitacoras` con las propiedades *datatype* de la clase `#Bitacora`. Para ello se **arrastra** la fuente `bitacoras` encima de las propiedades de la clase `#Bitacora`:

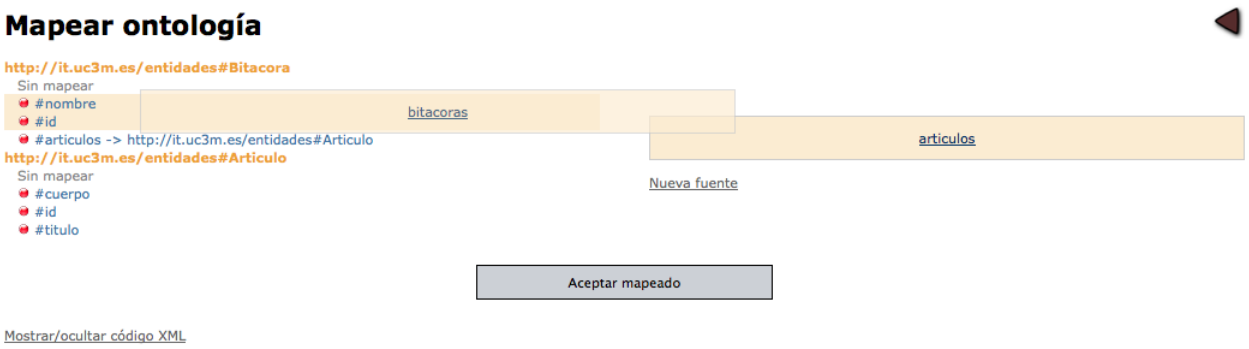

Figura 6.10: Mapeando una fuente a una clase

Al **soltar** la fuente encima de la clase aparece la ventana de mapeado:

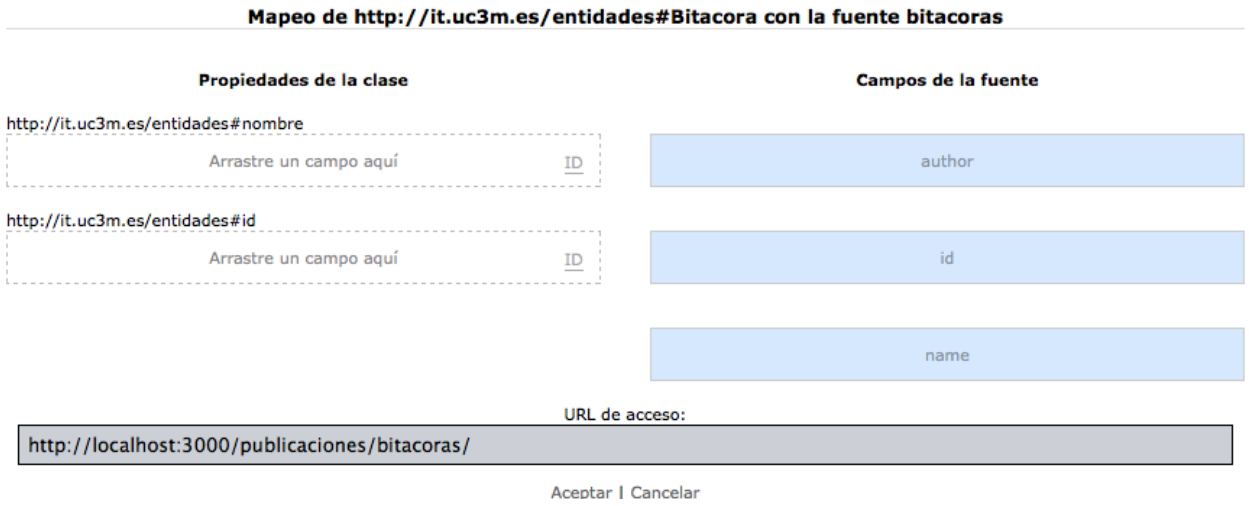

Figura 6.11: Mapeando propiedades y columnas

Como se ve en la figura 6.11, esta nueva ventana muestra los siguientes controles:

1. A la izquierda, una lista de las propiedades de la clase `#Bitacora`.
2. A la derecha, una lista de las columnas de la tabla `bitacoras`.
3. En la parte inferior, una entrada de texto titulada *URL de acceso* y los botones `Aceptar` y `Cancelar`.

Para hacer corresponder las diferentes propiedades de la clase con las columnas de la tabla, se arrastran una a una las columnas a las propiedades. Por último, se debe pulsar en el indicador `ID` de la propiedad `id` para indicar que esta propiedad es identificador único. La página queda así:

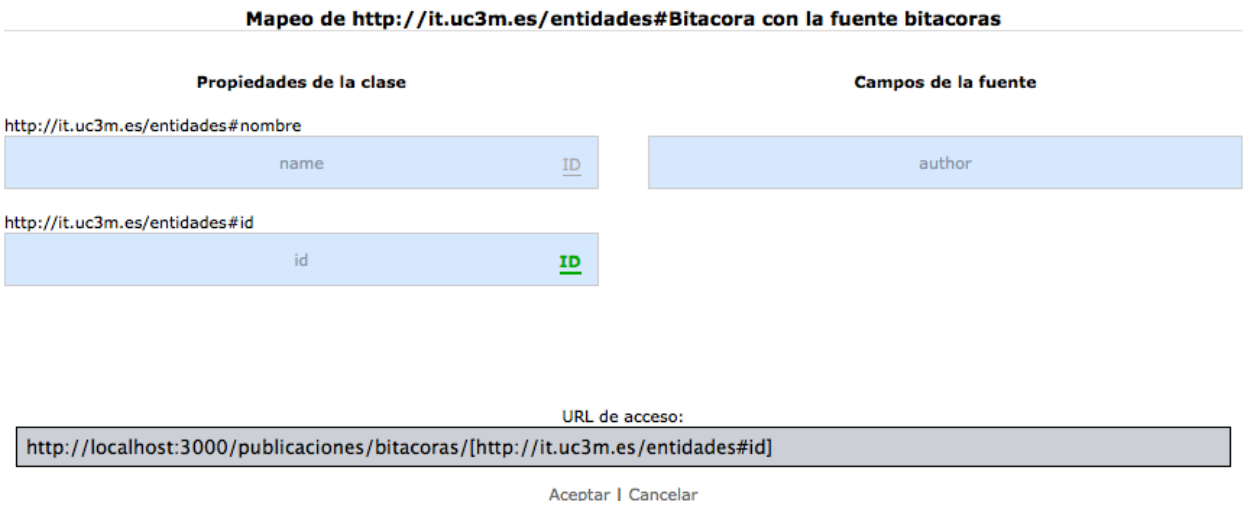

Figura 6.12: Propiedades y columnas mapeadas

Puede observarse que la URL de acceso se ha rellenado automáticamente en base a la URL del servidor y los nombres de las propiedades y las bases de datos; sin embargo, el usuario es libre de modificarla a su conveniencia. Se observa, además, que la columna `author` no se hace corresponder con ninguna propiedad de la clase: Antive ignorará esta columna.

Al aceptar este mapeado se vuelve a la página inicial, donde se puede observar que algunas propiedades ya han sido mapeadas al aparecer junto a un indicador verde:

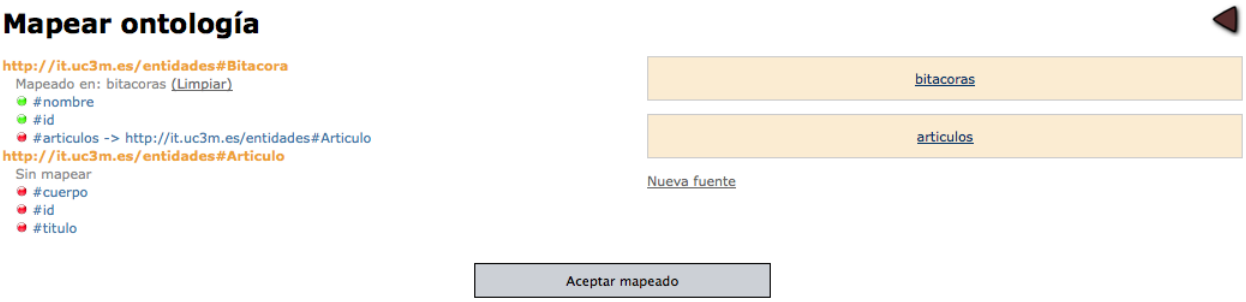

Figura 6.13: Clase `#Bitacora` mapeada

Se realiza el mismo procedimiento para la clase `#Articulo`: se arrastra la fuente `articulos` a la clase `#Articulo`:

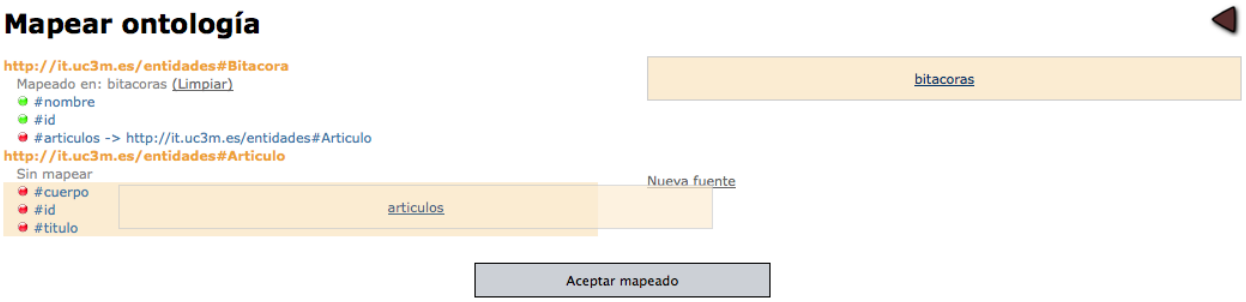

Figura 6.14: Mapeando `#Articulo` con la tabla `articulos`

para acceder así a la página de mapeado de `#Articulo`.

Figura 6.15: Mapeando `#Articulo`

Como se indicó que la propiedad `#id` es identificador único, ya aparece marcada como tal. Antive Map propone una nueva URL de acceso, aunque de nuevo el usuario es libre de editarla. De nuevo, se arrastra cada columna a la propiedad que le corresponde para configurar el mapeado como se observa en la figura 6.16:

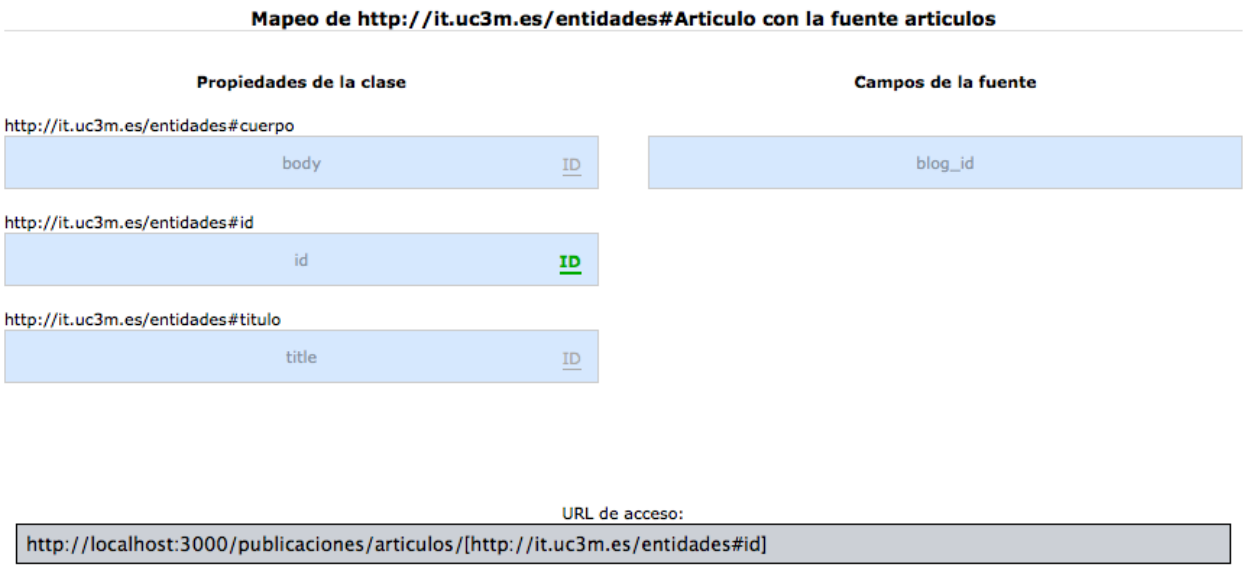

Figura 6.16: Propiedades y columnas mapeadas en `#Articulo`

Al aceptar este mapeado se vuelve a la página inicial, donde se observa que todas las propiedades tienen alguna correspondencia a excepción de la propiedad de tipo objeto `#articulos`.

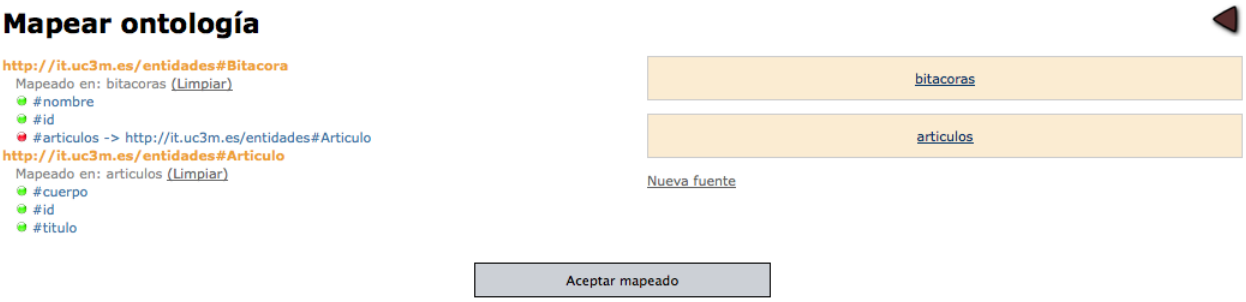

Figura 6.17: Todas las propiedades *datatype* han sido mapeadas

Hasta este momento el usuario ha creado correspondencias entre tablas y clases, o más concretamente entre columnas de tablas y propiedades *datatype* de cada clase. La propiedad que falta por mapear es de tipo *object* y por tanto no tiene ninguna correspondencia con una columna de una tabla. En realidad *las propiedades de tipo object no tienen correspondencia directa.*

El mecanismo que Antive emplea para resolver una propiedad de tipo *object* consiste en que el usuario crea *relaciones* entre las clases rango y dominio de cada propiedad de tipo *object*. Estas relaciones pueden ser entre propiedades de tipo *datatype* o bien entre columnas de las tablas relacionadas.

En este caso la correspondencia que permite resolver la propiedad `#articulos` es la siguiente: dada una bitácora b_i, los artículos que corresponden a la propiedad `#articulos` de b_i forman el conjunto $\{a_i\}_i$ tal que:

$$a_i.blog\_id = b_i.id$$

Para crear esta relación y poder resolver así la propiedad `#articulos` basta con arrastrar la clase `#Articulo` a la propiedad `#articulos`:

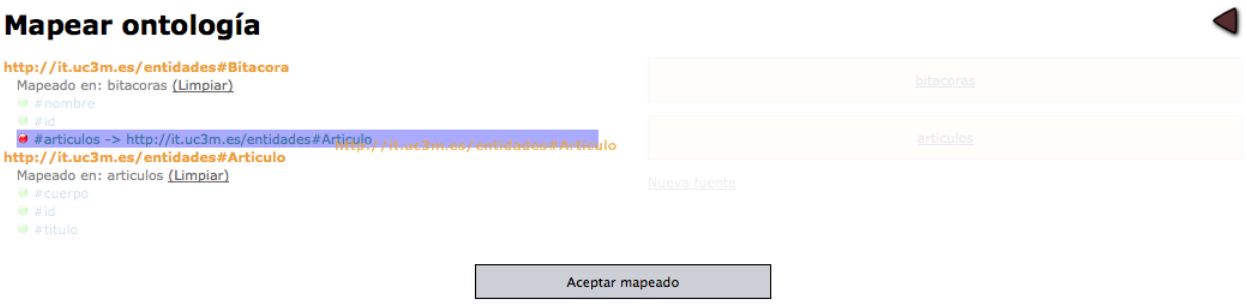

Figura 6.18: Mapeando una propiedad de tipo objeto

Aparece la ventana de mapeado de la propiedad como en la figura 6.19:

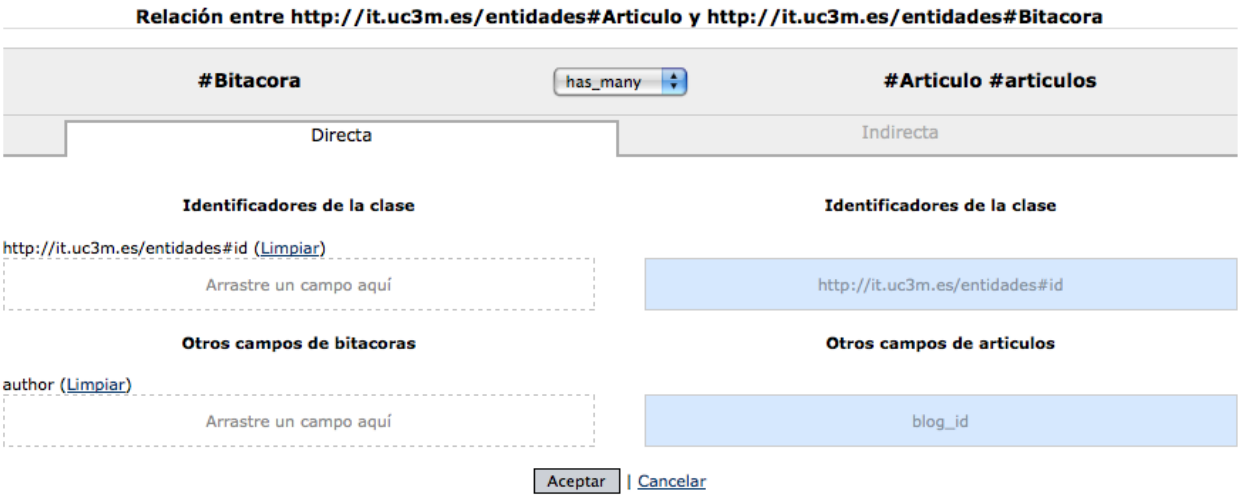

Figura 6.19: Mapeando una propiedad de tipo objeto (I)

En esta ventana aparecen varios controles:

1. Un desplegable que permite definir si la relación es de tipo `has_many` o de tipo N:1 `belongs_to`.
2. Dos pestañas que permiten emplear relaciones directas (entre dos clases) o indirectas (usando una tabla intermedia como en las relaciones N:N).
3. Dos listas que contienen las propiedades de las clases involucradas en el mapeado, así como los campos de las fuentes que no han sido mapeados a ninguna propiedad *datatype*.

Para crear la relación:

$$a_i.blog\_id = b_i.id$$

se arrastra el campo `blog_id` de los artículos a la propiedad `#id` de las bitácoras:

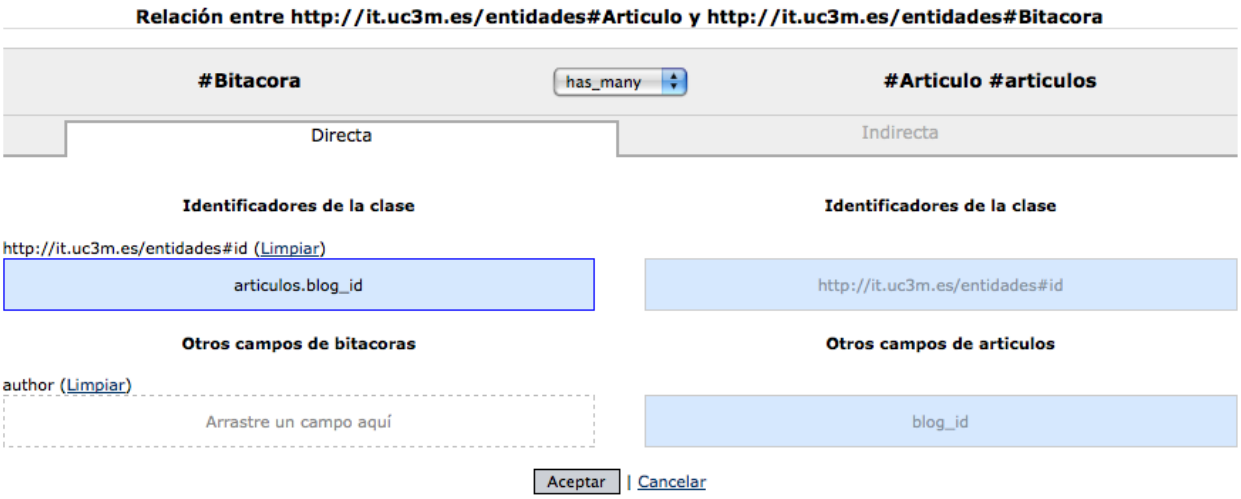

Figura 6.20: Mapeando una propiedad de tipo objeto (II)

Una vez creada esta relación, se pulsa `Aceptar` y se comprueba que todas las propiedades han sido mapeadas:

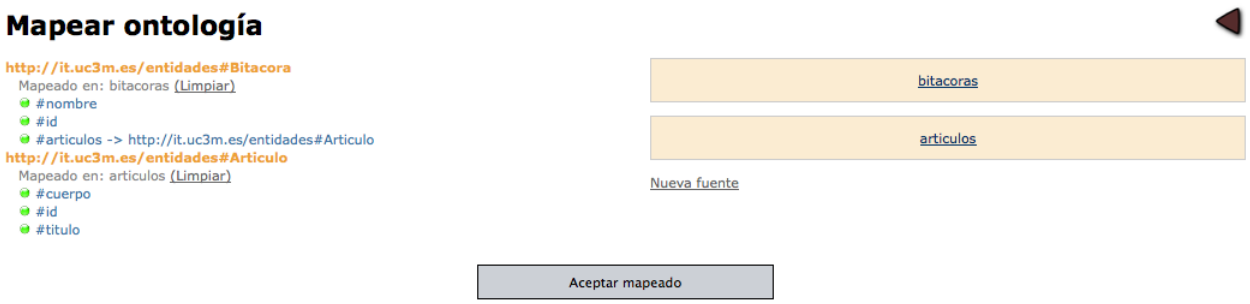

Figura 6.21: Todas las propiedades están mapeadas

Al pulsar el botón `Aceptar mapeado` se vuelve a la página de administración de ontologías.

6.1.6. Realizando consultas

Una vez creadas todas las correspondencias ya se pueden realizar consultas. Se accede a la URL:

```
http://localhost:3000/publicaciones
```

para encontrar la lista de clases disponibles en esta ontología, `#Bitacora` y `#Articulo`:

Al pulsar en el enlace de `#Bitacora` se accede a la URL:

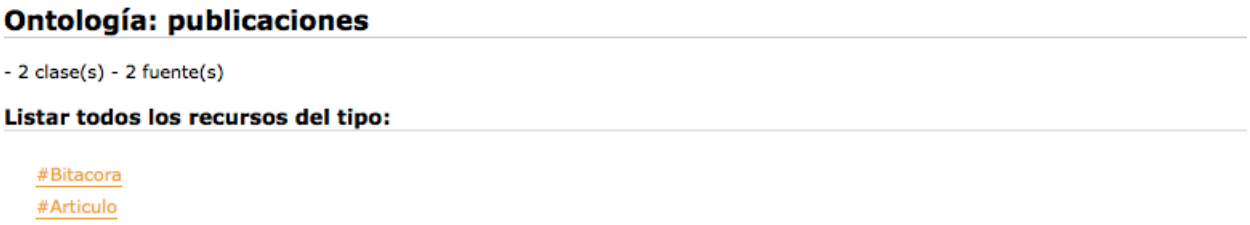

Figura 6.22: Clases disponibles

```
http://localhost:3000/publicaciones/bitacoras
```

donde se muestra la lista de bitácoras disponibles, como se observa en la figura 6.23:

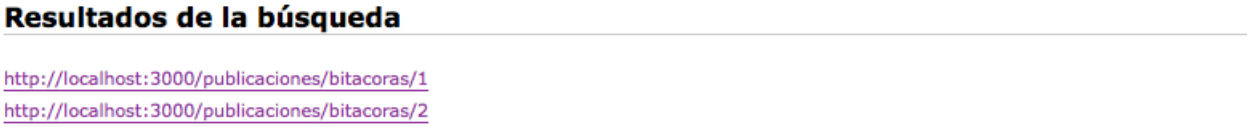

Figura 6.23: Bitácoras disponibles

Si se pulsa en el enlace de la primera bitácora se accede a la URL:

```
http://localhost:3000/publicaciones/bitacoras/1
```

donde se muestra la información de la bitácora número 1:

Figura 6.24: Información de la bitácora número 1

Se puede ver sus propiedades de tipo *datatype* (nombre e identificador) y un enlace para la propiedad de tipo *object* `#articulos`. Se pulsa en él para acceder a la URL:

```
http://localhost:3000/publicaciones/bitacoras/1/articulos
```

donde se muestra la lista de artículos de la bitácora:

Si se pulsa en el primero de ellos se accede a la URL:

```
http://localhost:3000/publicaciones/articulos/1
```

donde se muestra la información del artículo:

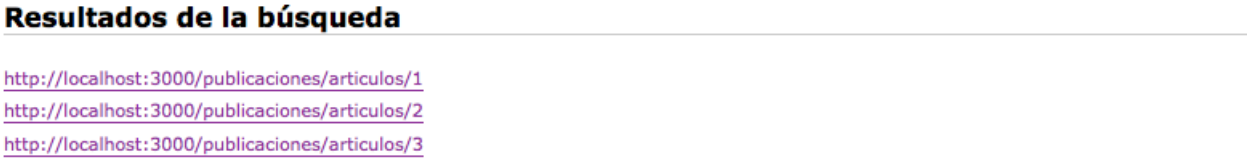

Figura 6.25: Artículos de la bitácora 1

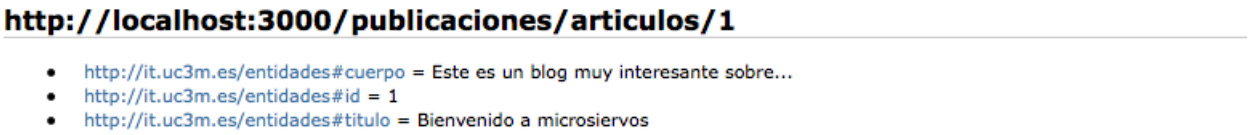

Figura 6.26: Artículo número 1

6.2. Federando varias bases de datos

Este segundo ejemplo se centra en el siguiente escenario: el Estado desea realizar un estudio sobre la capacidad de endeudamiento de los ciudadanos y sus hábitos en la compra de productos. Para ello se firma un convenio entre tres entidades:

- La Seguridad Social, cuyas bases de datos contienen la información de los ciudadanos.
- Un importante banco, que dará acceso a la base de datos de cuentas bancarias.
- Una cadena de centros comerciales, que dará información sobre sus clientes y las compras que realizan.

6.2.1. Ontología

La ontología que modela este escenario se muestra en la figura 6.27.

6.2.2. Bases de datos

Se cuenta con tres bases de datos, que se representan en la figura 6.28.

De este modelo se destacan los siguientes hechos:

- No hay cohesión entre las tablas de personas (de la Seguridad Social), clientes (del centro comercial) ni titulares (de la entidad bancaria) ya que son completamente independientes. Será común encontrar registros que no estén en las tres tablas a la vez.
- Las ventas del centro comercial se modelan mediante una relación N:N entre clientes y artículos, por lo que se implementa en la base de datos mediante una tabla intermedia (`ventas`).
- De igual manera existe una relación N:N entre titulares y cuentas bancarias, por lo que se emplea una tabla intermedia para modelar esta relación.

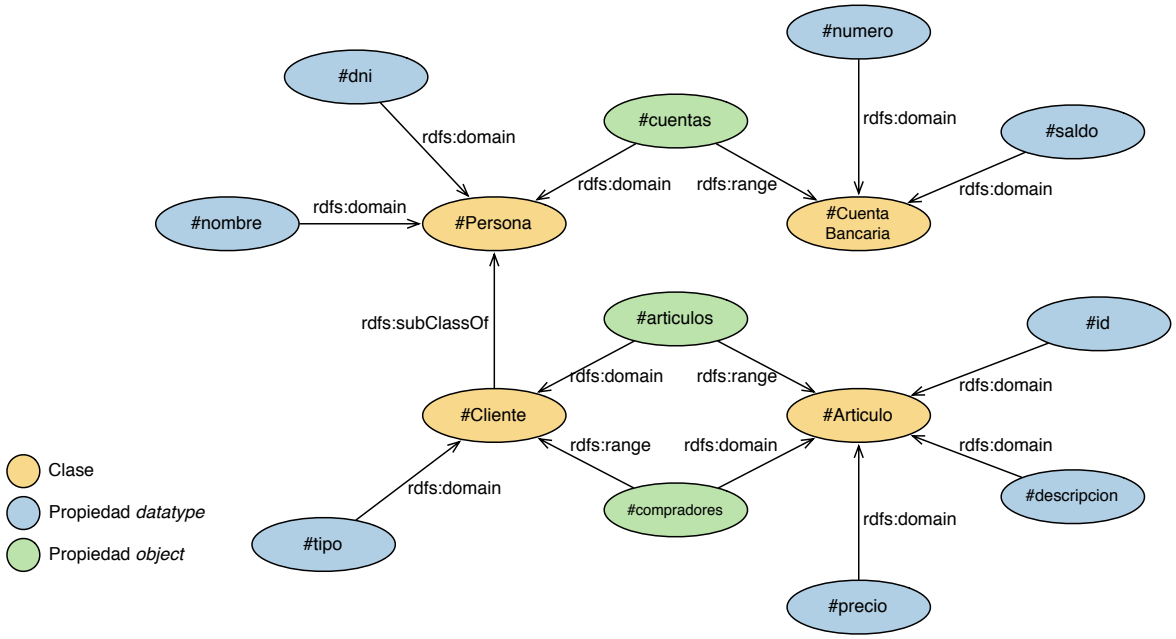

Figura 6.27: Ontología para el ejemplo 6.2

Para este ejemplo se emplearán los datos mostrados a continuación:

Base de datos de la Seguridad Social:

ciudadanos

dni	**nombre**
10001	Alice Liddell
20002	Bob Shaftoe
30003	Charlie Parker
40004	Daniel Waterhouse
50005	Eliza de la Zeur

Base de datos de la cadena de supermercados:

clientes

dni	**tipo**
50005	vip
60006	normal
70007	normal

articulos

id	**precio**	**descripcion**
1	50	Plumas de avestruz
2	40	Azogue
3	30	Madera
4	10	Manzanas
5	20	Refrescos

ventas

cliente_dni	**articulo_id**
50005	1
50005	2
50005	3
60006	4
70007	5

Base de datos de la entidad bancaria:

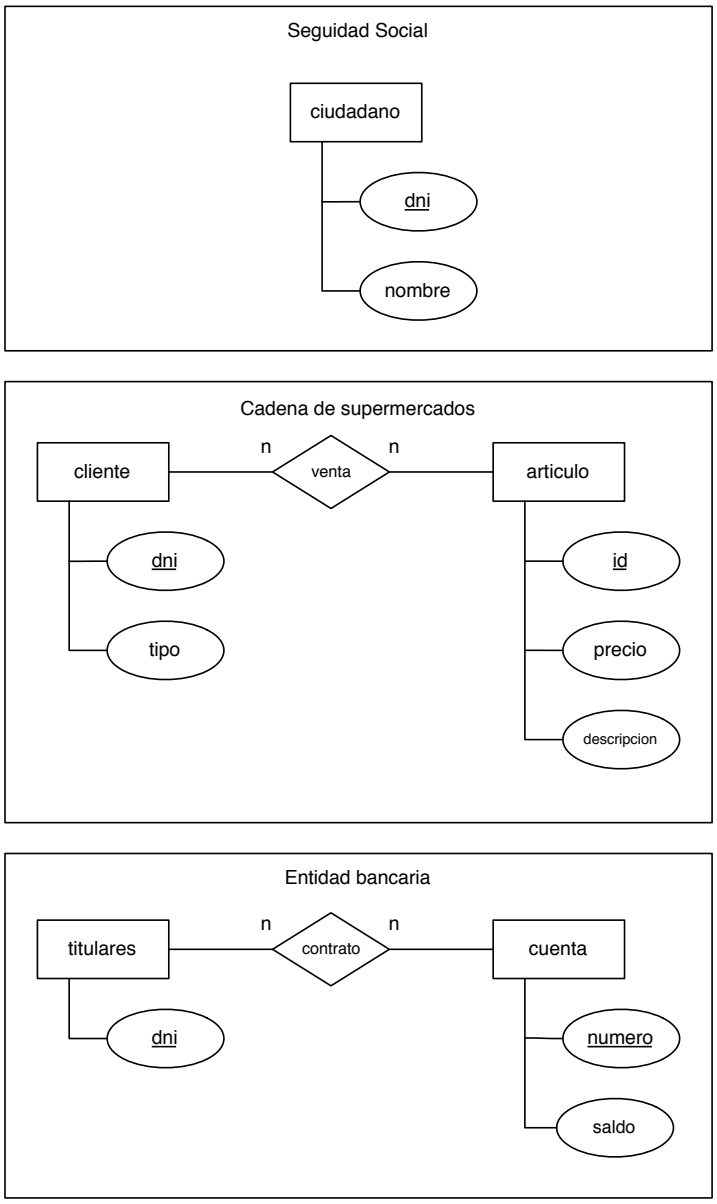

Figura 6.28: Modelo entidad-relación de las bases de datos del ejemplo 6.2

titulares

dni
10001
50005
80008

cuentas

id	saldo
100	100
200	200
300	300
400	400

contratos

dni	cuenta
10001	100
50005	200
50005	300
80008	400

6.2.3. Importando la ontología

Se realizará un proceso semejante al expuesto en el ejemplo de la sección 6.1:

1. Se accede a la página de administración, que en este caso será:

   ```
   http://localhost:3000/ontologies
   ```

2. Se pulsa en `Nueva ontología`.
3. La nueva ontología tendrá por nombre *consumo*.
4. Se cargará el archivo OWL de la ontología y *no* se contará con un archivo de mapeado.

6.2.4. Mapeando las personas

Se accede a la interfaz de mapeado y se crean tres nuevas fuentes:

1. `ss_ciudadanos`: Representa la tabla `ciudadanos` de la BBDD de la Seguridad Social.
2. `cc_clientes`: Representa la tabla `clientes` de la BBDD del centro comercial.
3. `eb_titulares`: Representa la tabla `titulares` de la BBDD de la entidad bancaria.

Estas tres fuentes contienen toda la información de los recursos de tipo `#Persona`:

Figura 6.29: Fuentes de datos para `#Persona`

A continuación se federarán las tres fuentes de datos haciendo corresponder sus columnas con las propiedades *datatype* de las clases `#Persona` y `#Cliente`:

1. Correspondencia entre `ss_ciudadanos` y `#Persona`:
   - La columna `nombre` se arrastra a la propiedad `#nombre`.
   - La columna `dni` se arrastra a la propiedad `#dni`.
   - Se marca la propiedad `dni` como identificador único.
   - La URL de acceso propuesta es mejorable, pero se cambiará más tarde.

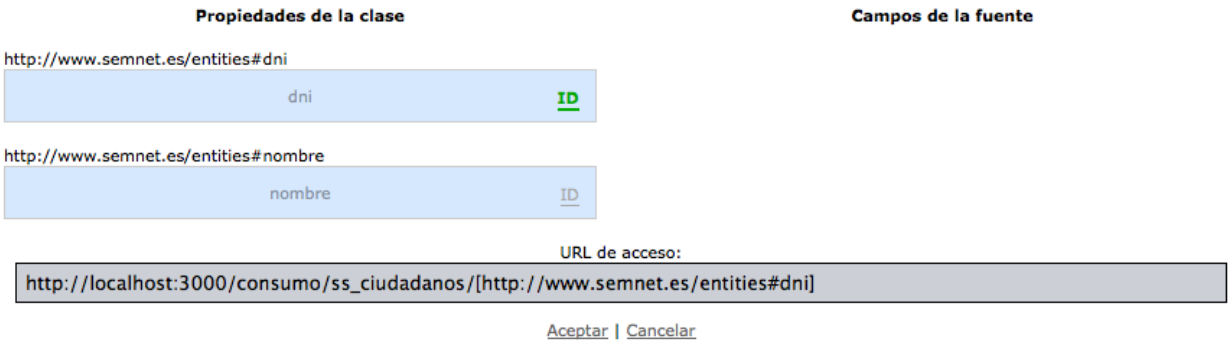

Figura 6.30: Mapeando la fuente `ss_ciudadanos`

2. Correspondencia entre `cc_clientes` y `#Cliente`:
   - La columna `dni` se arrastra a la propiedad `#dni`.
   - La columna `tipo` se arrastra a la propiedad `#tipo`.
   - La propiedad `dni` permanece como identificador único.
   - La URL de acceso se cambiará por:

     `http://localhost:3000/consumo/clientes/[http://www.semnet.es/entities#dni]`

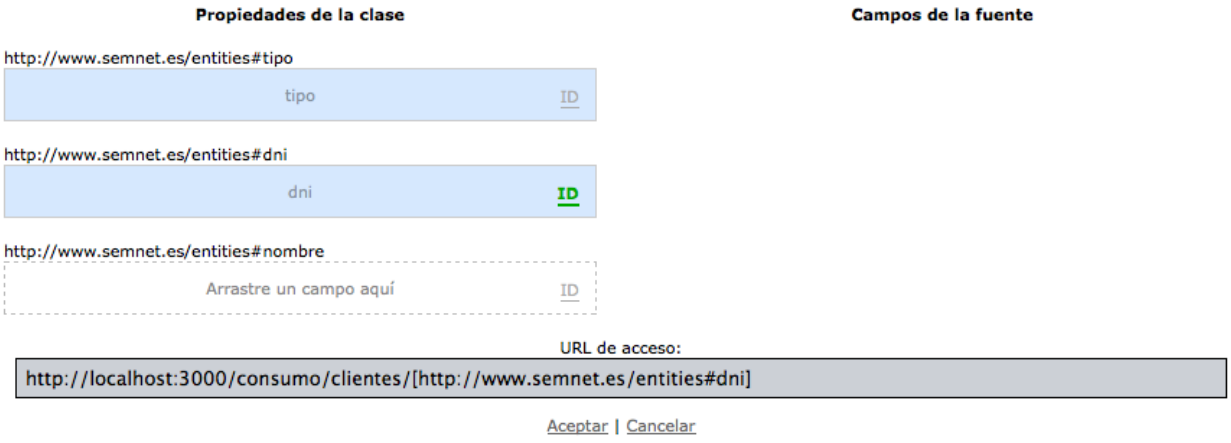

Figura 6.31: Mapeando la fuente `cc_clientes`

3. Correspondencia entre `eb_titulares` y `#Persona`:
   - La columna `dni` se arrastra a la propiedad `#dni`.
   - La propiedad `dni` permanece como identificador único.

- Se aprovecha este último mapeado para establecer la URL de acceso definitiva, que será:
  `http://localhost:3000/consumo/personas/[http://www.semnet.es/entities#dni]`

Figura 6.32: Mapeando la fuente `eb_contratos`

Antes de mapear todas las propiedades de la ontología se probará la federación de estas tres tablas en la clase `#Persona`. Para ello se pulsa en `Aceptar mapeado` para regresar a la página de administración. Allí se despliega la ontología `consumo` y se pulsa en `Probar` para acceder a la página de consultas.

Si se pulsa en el enlace correspondiente a la clase `#Persona`, o bien se accede directamente a la URL:

```
http://localhost:3000/consumo/personas
```

se accede a la lista de personas localizadas en las diferentes bases de datos:

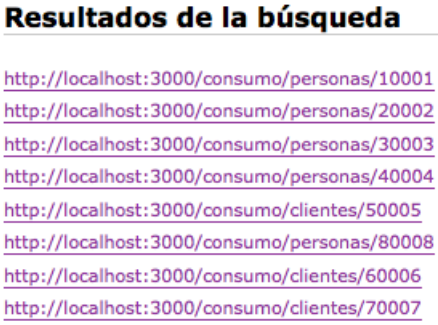

Figura 6.33: Personas encontradas en la federación de bases de datos

Se puede acceder a la información de cada una de ellas pulsando en los enlaces correspondientes. Nótese que el resto de las clases de la ontología no están mapeadas, por lo que se procederá a realizar este mapeo a continuación.

6.2.5. Mapeando el resto de las clases

Se accede de nuevo a:

```
http://localhost:3000/ontologies
```

y se pulsa en `Mapear` para finalizar el mapeado. Se añaden las siguientes fuentes:

- `cc_articulos`: Corresponde a la tabla `articulos` de la BBDD del centro comercial.
- `eb_cuentas`: Corresponde a la tabla `cuentas` de la BBDD de la entidad bancaria.

Las correspondencias con las propiedades *datatype* son:

- Correspondencia entre `cc_articulos` y `#Articulo`:

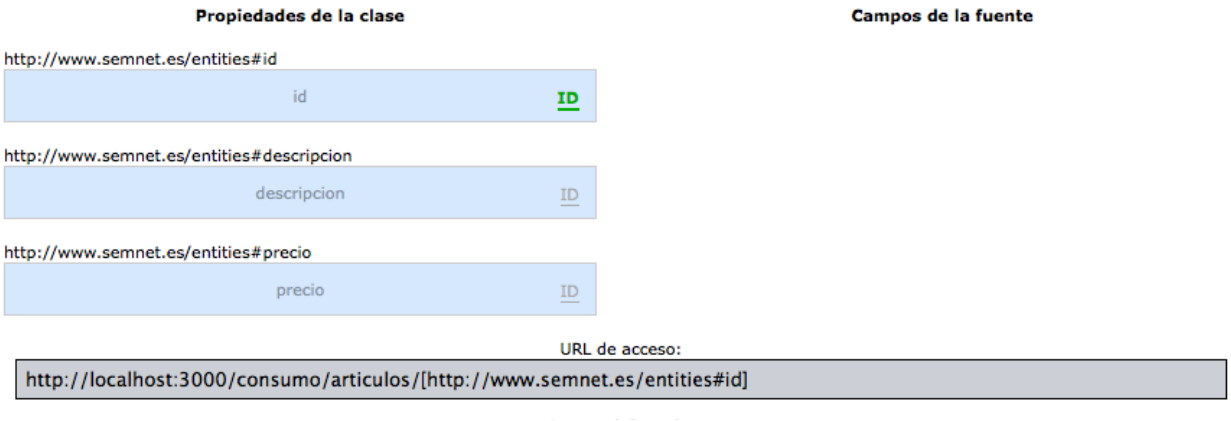

Figura 6.34: Mapeando la fuente `cc_articulos`

- Correspondencia entre `eb_cuentas` y `#CuentaBancaria`:

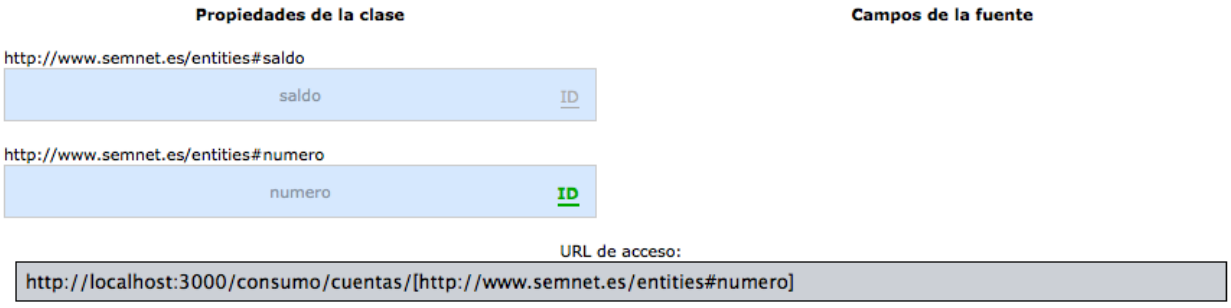

Figura 6.35: Mapeando la fuente `eb_cuentas`

6.2.6. Mapeando las propiedades de tipo *Object*

Se deben crear las siguientes fuentes de datos:

- `cc_ventas`: Corresponde a la tabla `ventas` de la BBDD del centro comercial.
- `eb_contratos`: Corresponde a la tabla `contratos` de la BBDD de la entidad bancaria.

Una vez añadidas estas fuentes se emplean en el mapeo de propiedades de tipo *Object* de la siguiente manera:

- Propiedad `#compradores` de `#Articulo`:
  1. Se arrastra la cabecera de la clase `#Cliente` a la propiedad `#compradores`.
  2. Para relacionar un artículo con un comprador se emplea una tabla intermedia (`ventas`) por lo que se pulsa la pestaña `indirecta`.
  3. En la lista desplegable de la parte superior se escoge el tipo de relación entre ambas clases. En este caso se usará `belongs_to`, que indica que un `#Articulo` *pertenece a* un `#Cliente`.
  4. Usando la lista desplegable titulada *Añadir intermediario* se escoge la fuente `cc_ventas`.
  5. Se arrastra la propiedad `#id` de `#Articulo` a la columna `articulo_id` de la tabla intermedia.
  6. Se arrastra la propiedad `#dni` de `#Cliente` a la columna `cliente_dni` de la tabla intermedia.

Figura 6.36: Mapeando la propiedad `#compradores`

- Propiedad `#cuentas` de `#Persona`:
  1. Se arrastra la cabecera de la clase `#CuentaBancaria` a la propiedad `#cuentas`.
  2. Se usará una tabla intermedia que será `eb_contratos`, por lo que se pulsa la pestaña `indirecta`.
  3. En la lista desplegable de la parte superior se usará `has_many`, que indica que una `#Persona` *tiene muchas* `#cuentas`.
  4. Usando la lista desplegable titulada *Añadir intermediario* se escoge la fuente `eb_contratos`.
  5. Se arrastra la propiedad `#dni` de `#Persona` a la columna `dni` de la tabla intermedia.
  6. Se arrastra la propiedad `#numero` de `#CuentaBancaria` a la columna `numero` de la tabla intermedia.

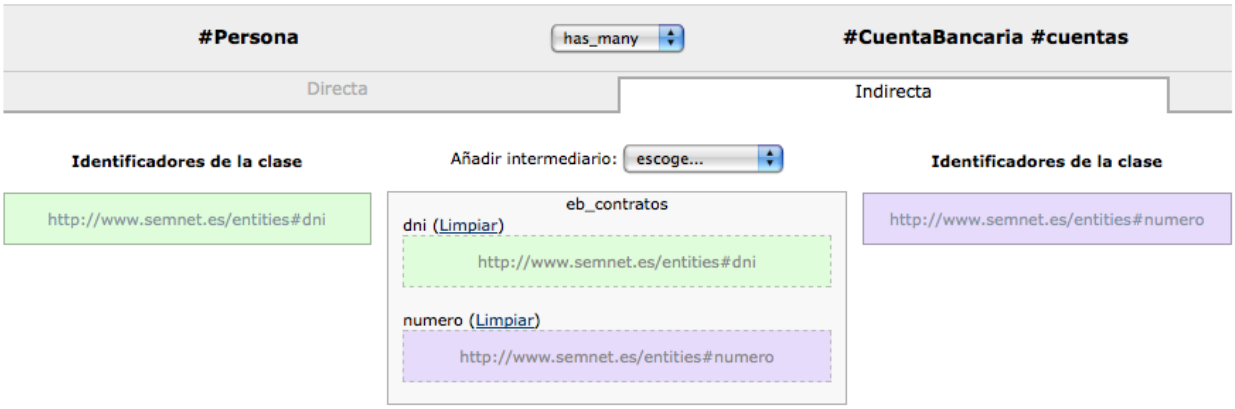

Figura 6.37: Mapeando la propiedad `#cuentas`

- Propiedad `#articulos` de `#Cliente`:
  1. Se arrastra la cabecera de la clase `#Articulo` a la propiedad `#articulos`.
  2. Se empleará una tabla intermedia (`ventas`) por lo que se pulsa la pestaña `indirecta`.
  3. En la lista desplegable de la parte superior se escoge el tipo de relación entre ambas clases. En este caso se usará `has_many`, que indica que un `#Cliente` *tiene muchos* un `#articulos`.
  4. Usando la lista desplegable titulada *Añadir intermediario* se escoge la fuente `cc_ventas`.
  5. Se arrastra la propiedad `#dni` de `#Cliente` a la columna `cliente_dni` de la tabla intermedia.
  6. Se arrastra la propiedad `#id` de `#Articulo` a la columna `articulo_id` de la tabla intermedia.

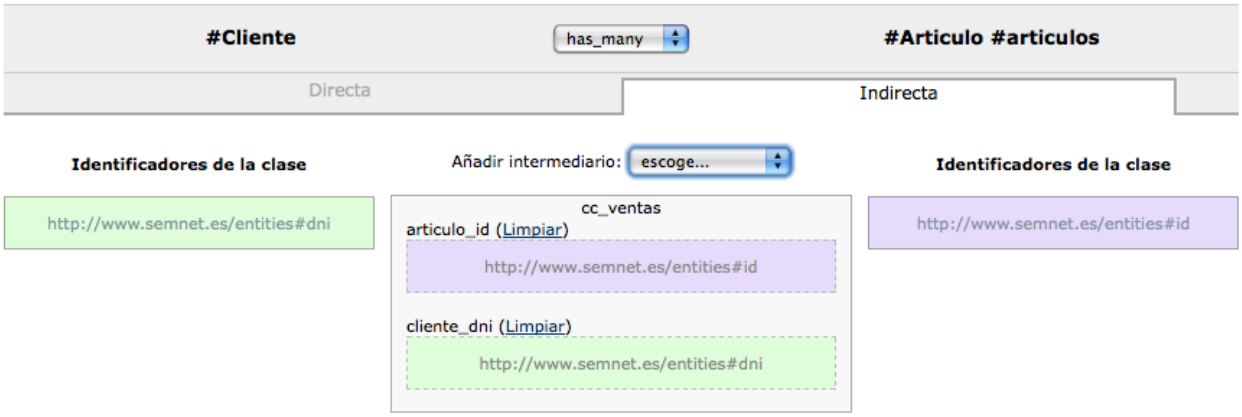

Figura 6.38: Mapeando la propiedad `#articulos`

Para finalizar se comprueba que todas las propiedades de la ontología se han mapeado (no hay indicadores en rojo), se pulsa `Aceptar mapeado` y se regresa a la página de administración.

6.2.7. Probando la federación

Para probar el funcionamiento del mapeado creado se pulsa en el título de la ontología `consumo` y en el enlace `Probar`, o bien se accede directamente a la URL:

```
http://localhost:3000/consumo
```

Pulsando en los diferentes enlaces se puede acceder a la lista de recursos de diferentes tipos. Al acceder ahora a un cliente, por ejemplo:

```
http://localhost:3000/consumo/clientes/50005
```

aparecen las propiedades del recurso y sus valores. De nuevo, las propiedades `#articulos` y `#cuentas` son enlaces, por lo que pulsando en cada uno de ellos, o bien accediendo a las URLs:

```
http://localhost:3000/consumo/clientes/50005/articulos
http://localhost:3000/consumo/clientes/50005/cuentas
```

se accede a la lista de artículos adquiridos por el cliente, así como a las cuentas bancarias que posee.

Cada una de las páginas HTML donde se muestra la información de un recurso tiene un enlace a la definición RDF de dicho recurso. Si se desea ver este documento, basta con acceder a la misma URL del recurso agregando la extensión *.rdf*:

```
http://localhost:3000/consumo/clientes/50005.rdf
```

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:ns1="http://www.semnet.es/entities#">
<rdf:Description rdf:about="http://localhost:3000/consumo/clientes/50005">
  <ns1:dni>50005</ns1:dni>
  <ns1:nombre>Eliza de la Zeur</ns1:nombre>
  <ns1:tipo>vip</ns1:tipo>
  <ns1:articulos>
    <rdf:Bag>
      <rdf:li resource="http://localhost:3000/consumo/articulos/1" />
      <rdf:li resource="http://localhost:3000/consumo/articulos/2" />
      <rdf:li resource="http://localhost:3000/consumo/articulos/3" />
    </rdf:Bag>
  </ns1:articulos>
  <ns1:cuentas>
    <rdf:Bag>
      <rdf:li resource="http://localhost:3000/consumo/cuentas/200" />
      <rdf:li resource="http://localhost:3000/consumo/cuentas/300" />
    </rdf:Bag>
  </ns1:cuentas>
</rdf:Description>
</rdf:RDF>
```

Figura 6.39: Representación RDF de un recurso

Capítulo 7

Conclusiones y trabajo futuro

En este capítulo se resumen los principales aspectos de la aplicación desarrollada, contemplando los objetivos iniciales planteados y el grado de éxito alcanzado respecto a ellos. También se considerarán las ventajas que se han conseguido respecto a las soluciones existentes y se analizarán las limitaciones que se presentan para determinar posibles líneas futuras de desarrollo.

7.1. Conclusiones generales

En este proyecto se ha planteado como principal problema la necesidad de integrar un conjunto de fuentes de datos heterogéneas e independientes para conseguir unificar toda la información disponible en ellas en *recursos*, ofreciendo así la información de cada uno de estos de forma *abstracta*, con transparencia respecto a la realidad física del almacenamiento de dicha información.

En este contexto, se ha realizado un estudio de las principales herramientas disponibles para llevar a cabo esta *federación* y se ha obtenido como conclusión que el empleo de una ontología para describir el modelo de datos de la federación se presenta como una opción muy recomendable.

El empleo de recursos para unificar la información disponible, así como el de una ontología para describir el dominio de la aplicación, han incitado el uso de lenguajes de la Web Semántica para realizar este proyecto. Se ha comprobado que algunas soluciones existentes, aunque ofrecen un funcionamiento semejante, se alejan del empleo de estas tecnologías, ya sea empleando otros lenguajes u ofreciendo una funcionalidad divergente con el interés del proyecto.

De esta manera, se ha optado por realizar una nueva aplicación que permita crear una correspondencia entre un conjunto de fuentes de datos y una ontología descrita en el lenguaje OWL. En esta aplicación, a la que se le ha dado por nombre Antive, se han diferenciado dos aspectos:

1. Se ha realizado un gran esfuerzo para aprovechar la riqueza de las descripciones de la ontología. En este sentido, se aprovechan en gran medida las descripciones de las propiedades de tipo objeto, las relaciones de herencia, etc. que se plasman en el componente Antive Core.
2. Se ha contemplado la necesidad de emplear una interfaz gráfica muy sencilla que permita

realizar las correspondencias sin necesidad de conocer el funcionamiento de la aplicación. Este esfuerzo se ha visto materializado en Antive Map.

Desde el planteamiento de este proyecto se ha tenido como objetivo que sea retomado en el futuro como base para otros desarrollos. Por este motivo se ha hecho un esfuerzo por detallar los mecanismos internos de la aplicación para facilitar su aprendizaje por parte del futuro desarrollador.

Por último, se han ofrecido dos ejemplos que muestran el funcionamiento general de la aplicación, demostrando las capacidades de resolución de consultas e integración de datos heterogéneos y distribuidos en diferentes fuentes.

7.2. Ventajas de Antive

Una vez finalizado el desarrollo base de Antive, se puede comparar la funcionalidad conseguida con la de las soluciones existentes mostradas en el capítulo 2. En este sentido, se pueden obtener de su empleo las siguientes ventajas:

- *Uso de una única ontología*: La mayor parte de soluciones existentes emplean varias ontologías para describir tanto el esquema general como los esquemas locales de cada fuente de datos.
- *Uso de OWL*: Las soluciones existentes que emplean una única ontología para describir el modelo de datos no hacen uso del lenguaje OWL que se presenta hoy en día como una de las principales opciones.
- *Web Semántica*: El mecanismo más empleado para la consulta de información actualmente es la Web. Las soluciones existentes, a pesar de plantear la Web como fuente de datos, no contemplan relación con el empleo de las tecnologías de la Web Semántica.
- *Interfaz gráfica*: El uso de una interfaz gráfica para crear las correspondencias entre bases de datos y elementos de la ontología hace que la experiencia del usuario sea más sencilla.
- *Alta heterogeneidad*: El diseño de Antive Core se ha realizado contemplando como base una alta heterogeneidad en las bases de datos, de forma que se contempla:
  - Que la información de un recurso esté disponible en varias fuentes.
  - La inconsistencia en los datos.
  - Las propiedades multievaluadas.
- *Uso intensivo de mecanismos de jerarquía*: El algoritmo de resolución de recursos hace uso de las relaciones de herencia existentes para refinar en la medida de lo posible la cantidad de información disponible de cada recurso.
- *Relaciones entre recursos*: Las relaciones entre tablas de las bases de datos se pueden mapear a propiedades de tipo objeto, ya sea empleando claves ajenas o tablas intermedias, de forma que a partir de un recurso se obtienen los recursos relacionados con él.
- *Lenguaje de programación*: Para el desarrollo de Antive Core, la parte más compleja de la aplicación, se ha optado por un lenguaje muy natural, sencillo de aprender y mantener, que

permite al futuro desarrollador abordar su aprendizaje de forma más directa.

- *Independencia entre Antive Core y Antive Map*: No existe un acoplamiento entre Antive Core y Antive Map, por lo que se puede emplear Antive Core de forma independiente e incorporarlo en otra aplicación sin esfuerzo.

7.3. Planificación final

Como se vio en la introducción, en este proyecto se ha empleado un ciclo de vida en V para el desarrollo de los componentes, empleando el prototipado evolutivo como herramienta para el control continuo del proceso de implementación. Durante el desarrollo del proyecto se han distinguido cuatro etapas, cada una de las cuales se ha subdividido en tareas. La planificación final de estas cuatro etapas y sus tareas se muestra a continuación:

1. **Etapa de análisis general**: Se realiza un estudio del contexto del proyecto y se realiza un primer análisis de la solución.
   - Estudio del contexto: Web Semántica, ontologías, aplicaciones similares: 2 semana.
   - Estudio de bibliotecas de análisis de ontologías: 1 semana.
   - Análisis y diseño general de la aplicación: 1 semanas.
2. **Desarrollo Fase 1**: Desarrollo inicial y evolución de la herramienta. Se realizan diferentes versiones, contemplando el empleo de clases y propiedades *datatype* de una ontología.
   - Prototipo inicial de Antive Core: 1 semana.
   - Prototipado evolutivo de Antive Core, diseño e implementación: 4 semanas.
   - Estudio de bibliotecas para la interfaz web: 1 semana.
   - Prototipado evolutivo de Antive Map, mapeado de propiedades datatype, interfaz de consulta: 3 semanas.
3. **Desarrollo Fase 2**: Se contempla el soporte para propiedades de tipo *object* y la correspondencia con relaciones directas e indirectas.
   - Análisis y diseño del soporte de propiedades de tipo objeto en Antive Core: 1 semana.
   - Implementación y prototipado evolutivo del soporte de propiedades de tipo objeto en Antive Core: 3 semanas.
   - Protitopado evolutivo de mapeado de propiedades de tipo objeto en Antive Map, diseño e implementación: 3 semanas.
4. **Depuración**: Se realizan mejoras y depuraciones en el código.
   - Depuración y adaptaciones: 1 semana.

En la figura 7.1 se muestra un diagrama de Gantt de esta planificación.

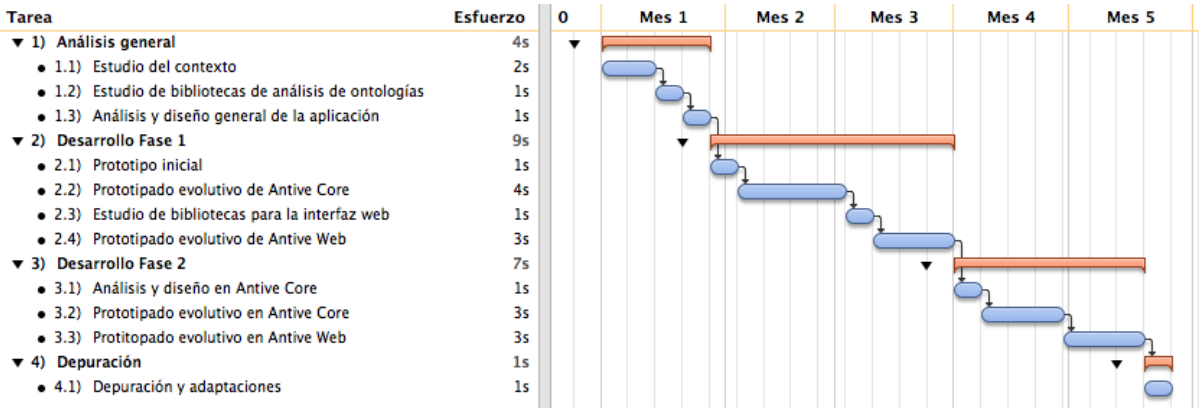

Figura 7.1: Planificación final del desarrollo de Antive

7.4. Principales dificultades encontradas

En primer lugar, el estudio de la Web Semántica ha sido uno de las mayores dificultades para la realización de este proyecto, ya que no existía un contacto previo con ella. Los conceptos derivados de las ontologías y los mecanismos involucrados en la Web Semántica suponen una diferencia muy elevada respecto a los mecanismos ya conocidos de modelos relacionales, jerarquías de clases en programación orientada a objetos, etc.

Las consideraciones particulares en el ámbito de las ontologías que no tienen un paralelismo directo con las estructuras de datos conocidas anteriormente han requerido un número elevado de prototipos, especialmente en Antive Core. La evolución de los mecanismos de búsqueda y las incorporaciones de nuevos conceptos en los diferentes prototipos han requerido varias reimplementaciones completas del algoritmo de búsqueda y resolución de recursos.

Por otro lado, la implementación de Antive Map ha requerido un gran esfuerzo para trasladar al lenguaje JavaScript las representaciones de las estructuras de datos empleadas en Antive Core. Paralelamente, el empleo de una interfaz grafica basada en arrastrar y soltar supone un manejo de grandes cantidades de eventos cuya implementación ha requerido un gran número de prototipos.

7.5. Futuras líneas de desarrollo

A partir del desarrollo actual se plantean un conjunto de posibles líneas de desarrollo. A continuación se ofrecen algunas ideas, ordenadas en función de la complejidad estimada:

- Añadir soporte para roles y autenticación: Aunque el soporte de autenticación está incorporado en Antive, no se especifica el mecanismo a emplear (LDAP, base de datos, etc.) por lo que sería conveniente incluir esta funcionalidad.
- Incorporar a Antive Map la capacidad de descargar las ontologías y los archivos de mapeado, con el fin de hacer copias de seguridad o analizar el XML de los archivos.

- Modo prueba/producción en Antive: Ya que el proceso de mapeado de una ontología puede realizarse en varios pasos, sería conveniente poder definir un *modo de pruebas* de forma que un usuario anónimo no pueda acceder a ella hasta que el mapeado esté completo.
- Añadir a Antive Map la posibilidad de clonar una ontología para crear un nuevo *proyecto* a partir de uno existente.
- Mantenimiento de Antive Map: Los navegadores Web evolucionan constantemente, en ocasiones sin compatibilidad hacia atrás, por lo que se hace necesaria una actualización constante de las bibliotecas empleadas, así como un mantenimiento general del código.
- Añadir una interfaz de consultas con filtros: Aunque el objetivo de este proyecto es federar varias bases de datos y localizar los recursos disponibles, se ha implementado la capacidad de aplicar filtros a las consultas. Se puede implementar una página de consulta con filtros en los que se determinen los valores que deben adoptar las propiedades y se obtengan los resultados correspondientes.
- Interfaz de mapeado local: Aunque actualmente las interfaces web se proponen como una solución apropiada para muchas aplicaciones, algunas herramientas de mantenimiento y configuración pueden resultar más convenientes como aplicaciones *stand-alone*.
- Extender el soporte de restricciones OWL, de forma que se contemplen a la hora de obtener los recursos. En entornos de alta heterogeneidad puede ser perjudicial, ya que no se asegura que este tipo de restricciones vayan a existir, pero puede resultar interesante permitir estas condiciones para su empleo en entornos que deban estar controlados.
- Mapeo automático de relaciones inversas: OWL permite definir que dos propiedades de tipo objeto son inversas. Actualmente hay que mapearlas de forma independiente, pero se puede investigar la capacidad de definir una de ellas y realizar automáticamente el mapeo inverso.
- Servidor de ontologías: Puede resultar interesante extender Antive Core al modelo cliente-servidor, de forma que una aplicación pueda realizar una consulta mediante un procedimiento remoto o un servicio web.
- Análisis del rendimiento y optimización: Las consultas que Antive realiza, unidas al procesado interno de los recursos, hacen que Antive tenga un rendimiento muy inferior al de una base de datos. Por ello puede resultar interesante localizar posibles optimizaciones en los algoritmos para reducir el tiempo de proceso.
- Planes de ejecución y metaprogramación: Una de las características de las bases de datos es que, antes de realizar una consulta, se crea un *plan* que determina cómo se va a llevar a cabo. Sería muy interesante que Antive Core realizara planes de ejecución a partir de una consulta, recogiendo el conjunto de fuentes de datos a consultar y las propiedades que se resuelven en cada una de ellas y estableciendo un plan para la resolución de la jerarquía de clases. En este sentido, los planes de Antive Core sería muy diferentes a los de las bases de datos, ya que en este último caso se conoce a priori la disponibilidad de los datos mientras que en entornos heterogéneos, Antive Core requiere tomar decisiones en función de los datos recuperados. Por ello, la metaprogramación se plantea como una línea de investigación muy interesante para

este desarrollo.

- Rendimiento y escalado: Las aplicaciones Rails no ofrecen un rendimiento elevado *per se*. Para solucionar este problema se emplean capas de *proxies* y *cachés* y el empleo de clústeres de instancias Rails. El estudio de estos mecanismos puede resultar muy interesante, especialmente si se contempla la posibilidad de generar los planes de ejecución expuestos anteriormente. También se propone el estudio de *memcache* para acelerar las consultas.

Apéndice A

Introducción a Ruby

Ruby es el lenguaje empleado en el desarrollo de Antive Core y parte de Antive Map. A pesar de su enorme popularidad gracias al *framework* de desarrollo web *Ruby On Rails*, Ruby sigue siendo un lenguaje poco conocido. Por ello este capítulo servirá de toma de contacto con el lenguaje.

Ruby se creó en 1993 con la finalidad de incorporar lo mejor de otros lenguajes de programación *clásicos* como Perl, Python, Smalltalk o Lisp. Algunas de las principales características del lenguaje son:

- Interpretado: No existe diferencia entre tiempo de compilación y tiempo de ejecución.
- Dinámico: Tanto el tipo de las variables como la definición de las clases, métodos, etc. puede alterarse en tiempo de ejecución.
- *Duck Typing*: El tipo de las variables se resuelve en tiempo de ejecución. Las llamadas a métodos se resuelven mediante paso de mensajes.
- Completamente orientado a objetos: Los tipos básicos (enteros, coma flotante, etc.) no existen como en otros lenguajes, sino que están representados por objetos y sus operaciones (suma, resta, igualdad) son métodos de la clase.

En las próximas secciones se hará un repaso por los principales mecanismos de Ruby que permitirán entender y extender el código de Antive.

A.1. Hola Mundo

A continuación se muestra el código de una clase que muestra en pantalla un saludo:

```
class HolaMundo
    def saluda
        puts "Hola Mundo"
    end
end
HolaMundo.new.saluda
```

Para ejecutar este programa basta con escribir en una terminal:

```
$ ruby holamundo.rb
```

Este código muestra varias claves:

- Las clases se definen con `class ... end` y los métodos con `def ... end`.
- Los métodos pueden o no recibir parámetros.
- No hay un método *main*. La ejecución comienza al principio del archivo *.rb* que se pasa como parámetro al intérprete Ruby. En este caso el código es una definición de una clase y, tras ella, el inicio del programa.
- Para crear un nuevo objeto HolaMundo se invoca el método `new` sobre el nombre de la clase.
- Cuando no hay riesgo de ambigüedad, no es necesario añadir `()` en la llamada a un método.

A.2. Métodos

En Ruby los métodos de una clase tienen una serie de características que lo diferencian de los lenguajes más comunes como Java:

- Todo método devuelve un valor: Cuando una línea de código se ejecuta, el valor devuelto es el evaluado en ella. Por ejemplo, el siguiente método devuelve la suma de dos números pasados como parámetros:

```
def suma(a, b)
    a + b
end
```

  Igualmente se puede emplear la palabra reservada `return` como en C o Java:

```
def suma(a, b)
    return a + b
end
```

  Nótese que aunque el comportamiento es idéntico, el mecanismo que se emplea para devolver un valor es conceptualmente diferente. En el primer caso el método devuelve el último valor evaluado, mientras que en el segundo caso se hace explícito el valor que se debe devolver.

- Argumentos con valor por defecto: Se pueden definir valores por defecto de los parámetros de un método:

```
def suma(a, b = 1)
    a + b
end
puts suma(5)
```

  En este caso la llamada a `suma` no indica el parámetro `b`, por lo que se toma el valor por defecto.

- Argumentos de longitud variable: Se sintetizan como elementos de un array dentro del método:

```
def suma(*op)
    r = 0;
    for i in 0..op.size - 1
        r = r + op[i]
    end
    r
end
puts suma(5)
```

  Este método recibe un número variable de argumentos y devuelve la suma de ellos. Nótese el uso del bucle `for`, así como el direccionamiento dentro de un array mediante `[]`.

- Bloques de código: Una de las principales novedades de Ruby respecto a otros lenguajes es que cada método puede recibir un bloque de código como argumento:

```
def haz_veces(veces, &block)
    for i in 0..veces - 1
        block.call
    end
end
haz_veces(5) { puts "Hola" }
```

  En este ejemplo se define un método `haz_veces` que recibe un número y un bloque de código y ejecuta dicho bloque el número de veces indicado en la variable `veces`.
  De todas las características de los métodos Ruby la capacidad de recibir bloques de código es la que permite emplear los *iteradores*, uno de los puntos más fuertes del lenguaje y que se verá en la sección A.6.

- Convenciones: Ruby permite que los métodos incluyan en su nombre caracteres como `!` o `?`. Como convención se emplea el carácter `!` para indicar que el método realiza una acción *destructiva* (por ejemplo sustituir los elementos de un array). Por otro lado, se suele emplear el carácter `?` en los métodos cuya acción consiste en realizar alguna comprobación. En cualquier caso es una convención y no una característica del lenguaje.

- Operadores: Ruby permite que una clase defina métodos con nombres como `+ - « » <=>== [] []=`. Ya que en Ruby no existen los tipos de datos básicos, en realidad las operaciones como suma o resta son en realidad llamadas a métodos. Sin embargo, para facilitar la sintaxis, Ruby permite invocar dichos métodos mediante la sintaxis de los operadores. Por ello las siguientes sentencias son equivalentes:

```
a = 5 + 4
a = 5.+(4)
```

A.3. Orientación a objetos

Como se vio al principio de este capítulo, una clase se define con la sintaxis:

```
class Clase
    ...
end
```

El constructor de una clase Ruby se denomina `initialize`:

```
class Clase
    def initialize(...)
        ...
    end
end
```

Las propiedades de una clase se definen anteponiendo el símbolo `@` al nombre de la misma:

```
class Clase
    def initialize(...)
        @variable = ...
    end
end
```

De forma semejante, las variables estáticas se definen empleando el símbolo `@@`:

```
class Clase
    def initialize(...)
        @@variable_estatica = ...
    end
end
```

Nótese que no es necesario declarar una propiedad como se hace en Java o C++. Ya que en Ruby no existe diferencia entre tiempo de compilación y tiempo de ejecución, las propiedades se añaden a la misma en el momento en que se ejecuta un código que las defina:

```
class Clase
    def initialize(...)
        @variable = ...
    end

    def metodo
        @otra_variable = ...
    end
end
```

Las variables `@...` se denominan *de instancia* y las variables `@@...` *de clase*. En el caso de Ruby, y a modo de curiosidad, las clases son en realidad instancias de la clase `Class` referenciadas

por constantes. Puede comprobarse empleando el intérprete de Ruby interactivo `irb`:

```
$ irb
>> class Cosa
>> end
=> nil
>> Cosa.object_id
=> 1787430
>> Cosa.class
=> Class
```

Ya que Ruby permite emplear el carácter `=` en el nombre de un método, es común que los métodos *accesores get* y *set* se implementen de la forma:

```
class Persona
    def nombre
        @nombre
    end

    def nombre=(n)
        @nombre = n
    end
end

p = Persona.new
p.nombre = "Juan"    # en realidad es una llamada a "nombre="
```

Ya que los métodos *accesores* son muy comunes, Ruby ofrece una macro para sintetizarlos en tiempo de ejecución:

```
class Clase
    attr_accessor :nombre
end
```

Se pueden definir los tipos de acceso a diferentes métodos de forma semejante a como se hace en C++:

```
class Clase
    public

    ...

    private

    ...

    protected
```

```
    ...
end
```

La herencia en Ruby es siempre herencia simple:

```
class Clase < ClasePadre
    ...
end
```

Sin embargo existe un mecanismo para simular la herencia compuesta mediante *mixins*. Este mecanismo consiste en emplear *módulos*, que son conjuntos de métodos:

```
module MiModulo
    def metodo
        ...
    end

    def metodo2
        ...
    end
end
```

Así, una clase puede incorporar los métodos de varios módulos:

```
class Clase < ClaseBase
    include MiModulo
    include MiModulo2

    def ...
    ...
end
```

Nótese que un módulo no alcanza el status de clase, pero permite emplear variables de clase. Estas variables solo tendrán sentido cuando se han incorporado a una clase real. De nuevo, este mecanismo es factible gracias a que toda esta síntesis se realiza en tiempo de ejecución.

A.4. Clases de interés

A.4.1. Array

La clase Array representa una colección ordenada de elementos. Su tamaño puede variar durante la ejecución del programa y no presenta ninguna restricción respecto a la clase de los objetos que contiene. Para crear un array se pueden emplear dos notaciones:

```
array = Array.new
array = []
```

Para añadir objetos al array se puede emplear una asignación:

```
array[0] = "Hola Mundo"
```

O bien se pueden añadir elementos empleando el operador <<:

```
array << "Hola Mundo"
```

Para acceder a los elementos del array se emplean índices enteros, positivos o negativos:

```
puts array[0] // primer elemento
puts array[1] // segundo elemento
puts array[-1] // ultimo elemento
```

Para hacer el código legible, Array ofrece dos métodos para acceder a los elementos primero y último:

```
puts array.first
puts array.last
```

Un array puede contener otros arrays:

```
array = [1, 2, [3, 4, 5], 6, [7, 8]]
```

En este caso puede emplearse el método `flatten` (o `flatten!`) para generar un único array que contenga todos los elementos:

```
array.flatten!   # array contiene ahora [1, 2, 3, 4, 5, 6, 7, 8]
```

Otros métodos muy útiles de Array son `uniq` (obtiene un array sin elementos repetidos), `compact` (elimina los objetos nulos de un array)

A.4.2. Hash

La clase Hash representa una colección de pares clave-valor. Para crear una Hash se pueden emplear dos notaciones:

```
hash = Hash.new
hash = {}
```

Se puede iniciar una hash empleando una notación semejante a la de Perl:

```
hash = { clave1 => valor1, clave2 => valor2 }
```

Para crear un nuevo par en la hash se emplea una notación semejante a la de Array:

```
hash[clave] = valor
```

Por último, para acceder al valor de una clave, se emplea igualmente una notación semejante a Array:

```
puts hash[clave]
```

A.4.3. Symbol

Por último, los objetos de tipo Symbol son muy empleados en los desarrollos Ruby y Ruby On Rails. Se declaran empleando la siguiente notación:

```
:identificador
```

El uso de los símbolos consiste en permitir comparar dos identificadores mediante su puntero, o dicho de otro modo, mediante su nombre: dos símbolos son el mismo objeto si tienen el mismo nombre.

A modo de ejemplo, si se emplea el intérprete de ruby interactivo `irb` y se crean dos Strings idénticas:

```
>> a = "Hola"
=> "Hola"
>> b = "Hola"
=> "Hola"
```

Aunque las strings sean idénticas, no son el mismo objeto. Esto puede comprobarse observando el identificador (la *dirección*) de cada uno de ellos:

```
>> a.object_id
=> 1787900
>> b.object_id
=> 1785570
```

Sin embargo, si se hace esta prueba con dos símbolos:

```
>> a = :hola
=> :hola
>> b = :hola
=> :hola
>> a.object_id
=> 199138
>> b.object_id
=> 199138
```

Se puede ver que, efectivamente, es el mismo símbolo. Estos objetos se suelen emplear como identificadores o claves de tablas hash, arrays, etc:

```
persona = { :nombre => "Juan", :apellido => "Alba" }
puts persona[:nombre]
```

Nota: la clase Array de Ruby implementa un operador `==` que compara dos Strings por contenido, de forma semejante al método `equals()` de Java:

```
>> a = "Hola"
=> "Hola"
>> b = "Hola"
=> "Hola"
>> a == b
=> true
```

Esto no debe confundir al programador, aunque esta igualdad sea cierta, los objetos comparados no son el mismo.

A.5. Peculiaridades

El código de Antive hace un uso intensivo de algunas peculiaridades del lenguaje que suelen sorprender al programador de Java o C:

1. Evaluaciones: Todo objeto Ruby que no sea `nil` ni `false` se evalúa a `true` en una condición, a diferencia de Java o C en los que `0` se evalúa a `false`.
2. Condicionales: Aparte de la sintaxis típica de condicionales:

   ```
   if cond
       ...
   else
       ...
   end
   ```

   Ruby permite una sintaxis de condicionales retardados al estilo Perl:

   ```
   ... if cond
   ```

   Asimismo existe la palabra reservada `unless` que equivale a `if not`:

   ```
   ... unless cond
   ```

   A modo de ejemplo, el código de Antive Core contiene una gran cantidad de mensajes de depuración que pueden activarse estableciendo la variable global `$VERBOSE` a cualquier valor; por ello, para mostrar un mensaje de depuración se emplea el código

   ```
   puts "mensaje de depuracion" if $VERBOSE
   ```

A.6. Iteradores

En los lenguajes como C++ o Java los iteradores son objetos que permiten realizar una tarea en los elementos de una colección empleando para ello un API concreto. En Java se emplean

típicamente métodos `hasNext()` y `next()` para recorrer la colección y obtener cada uno de los elementos de la misma respectivamente. El siguiente ejemplo corresponde a un fragmento de código Java que muestra por pantalla los elementos de una colección:

```
Iterator iterator = ....
while(iterator.hasNext()) {
    Object o = iterator.next();
    System.out.println("Elemento: " + o);
}
```

La forma de trabajar con colecciones en Ruby consiste en que los objetos que implementan algún tipo de colección (*arrays*, tablas *hash*, etc.) cuentan con un conjunto de métodos encargados de iterar sobre la colección y que reciben un bloque de código de forma semejante a como se vio en la sección anterior. De esta manera es el contenedor de la colección el encargado de iterar sobre los distintos elementos, por lo que el programador no necesita realizar ninguna tarea al respecto.

En el siguiente ejemplo corresponde a la versión Ruby del fragmento de código Java anterior:

```
coleccion.each { |o| puts o }
```

El objeto `coleccion` implementa un método `each` que itera sobre los elementos de la colección y, para cada uno de ellos, ejecuta el bloque de código `puts o` pasándole como parámetro el elemento `o`. Este enfoque, heredado de Smalltalk, es completamente opuesto al de Java o C pero ofrece al programador una serie de ventajas muy atractivas. A continuación se muestran algunos ejemplos de ello:

- Dada una tabla *hash* con notas de alumnos (de la forma hash[nombre] = nota), obtener una nueva *hash* con los alumnos aprobados:

  ```
  aprobados = notas.select { |nombre, nota| nota >= 5 }
  ```

- Sustituir los elementos de un array por el doble de su valor:

  ```
  array.collect! { |o| o*2 }
  ```

  Nótese el uso del método `collect!` con el carácter `!` que indica que el método es destructivo.

En Antive el uso de iteradores es muy intenso y es preciso comprender su uso para poder entender el código de la aplicación. A modo de curiosidad, Antive Core emplea bucles `while` en una sola clase que emplea un API de una biblioteca Java; el resto de la aplicación carece de bucles `for` y `while`.

A.6.1. Creando un iterador

En general, un iterador es un método que sigue el siguiente esquema:

1. Recibe un conjunto de parámetros *de entrada*.
2. Recibe un bloque de código.

3. Genera un conjunto de parámetros *intermedios*.
4. Invoca el bloque de código pasándole los parámetros *intermedios*.
5. Emplea el valor devuelto por el bloque de código.
6. Genera un valor *de retorno*.

Para crear un método iterador (o cualquier otro método que reciba como parámetro un bloque de código) se pueden emplear tres mecanismos:

- Emplear la sintaxis vista en la sección A.2:

```
def iterador(..., &block)
    ...
    block.call (param1, param2, ...)
    ...
end
iterador(...) { |param1, param2, ...| ... }
```

- Encapsular el código en un objeto de tipo Proc:

```
def iterador(..., block)
    ...
    block.call (param1, param2, ...)
    ...
end
codigo = Proc.new { |param1, param2, ...| ... }
iterador(..., codigo)
```

- Emplear la palabra reservada `yield` y obviar el parámetro que recibe el bloque de código (`block` en los ejemplos anteriores):

```
def iterador(...)
    ...
    yield param1, param2, ...
    ...
end
iterador(...) { |param1, param2, ...| ... }
```

Nótese que el tercer método es el que menos cantidad de código requiere pero no hace explícita la referencia al bloque de código pasado como parámetro, lo que puede suponer necesario en algunos casos.

A.7. JRuby

La implementación original de Ruby se hizo en C pero, a medida que el lenguaje fue adquiriendo popularidad, se iniciaron desarrollos para crear intérpretes en otros lenguajes. Probablemente el más conocido es JRuby, un intérprete Ruby escrito en Java. La principal ventaja de este proyecto

es la posibilidad de mezclar código Ruby y Java en un mismo programa. Antive Core hace uso de esta capacidad para realizar llamadas a la biblioteca Jena para parsear códigos XML de ontologias.

Para emplear un API Java desde un código JRuby basta con:

1. Importar el soporte para Java:

```
require 'java'
```

2. Importar la clase Java:

```
include_class 'com.hp.hpl.jena.rdf.model.ModelFactory'
```

3. Realizar las llamadas empleando sintaxis Ruby:

```
objetoJava = ClaseJava.new(...)
objetoJava.metodoJava(...)
```

Ya que un objeto Java recibe como parámetros otros objetos Java, por lo general no resulta un inconveniente mezclar código Ruby y Java; sin embargo, existen algunas clases (como String) que están implementadas tanto en Ruby como en Java:

```
objetoJava = ClaseJava.new(...)      # Esto es Java
cadena = "cadena_de_caracteres"      # Esto es Ruby
objetoJava.metodoJava(cadena)        # Mezcla Java y Ruby
```

En este caso el intérprete JRuby transforma las clases Ruby a sus equivalentes Java y viceversa, por lo que el programador no necesita realizar ninguna conversión explícita.

Por estas razones JRuby es una excelente herramienta que permite acceder a bibliotecas Java de forma transparente, emplear bibliotecas Ruby en aplicaciones Java, desplegar aplicaciones Ruby en contenedores como Tomcat o Glassfish, prototipado y testing de aplicaciones y componentes Java, etc.

Apéndice B

Ejemplo de uso de transacciones y cachés

A continuación se muestra un método que hace uso de cachés y transacciones. Al invocar este método sobre la clase c_i se comprueba si existe algún ciclo en las superclases de c_i:

```
1   def cycles_towards_parents?(transaction = nil)
2       cache :cycles_towards_parents do
3
4           # inicio de la transaccion
5           # (esto se ejecuta solo en la primera llamada)
6           transaction = new_transaction(transaction) do |transaction|
7               transaction.content = false
8           end
9
10          # pregunto a cada padre
11          @parents.each_pair do |uri, parent|
12              if parent.did_transaction? transaction
13                  # si ha hecho la transaccion es que ya hemos
14                  # pasado por aqui
15                  transaction.content = true
16              else
17                  # si no ha hecho la transaccion, recursividad
18                  parent.cycles_towards_parents?(transaction)
19              end
20          end
21
22          # fin de la transaccion
23          # (esto se ejecuta solo en la primera llamada)
24          end_transaction transaction do |transaction|
25              save_in_cache transaction, :cycles_towards_parents
26          end
27
```

```
28              # cada clase devuelve el valor actual de la transaccion
29              # (que sera true o false)
30              transaction.content
31          end
32      end
```

Listado B.1: Ejemplo del uso de cachés y transacciones

En la línea 1 se define el valor por defecto de `transaction` a `nil`. La llamada a `cache` de la línea 2 indica que solo se debe ejecutar el bloque de código contenido si no existe una transacción almacenada con la clave `:cycles_towards_parents`. En caso contrario se debe ignorar el bloque de código interno y se debe devolver el valor almacenado.

La línea 6 emplea `new_transaction` para generar una nueva transacción (en caso de que `transaction` sea `nil`, es decir, en caso de que el método se invoque sin argumentos) o bien reutilizar la transacción actual (en caso de que el método se invoque con una transacción como argumento). Solo en el primer caso se ejecuta el código de la línea 7, que corresponde a la inicialización de la transacción.

La línea 11 itera sobre el conjunto de padres de la clase y el contenido de este bucle corresponde a la lógica del algoritmo: una jerarquía de clases tiene ciclos si al recorrerla en un sentido (ascendente en este caso) se llega dos veces a una misma clase. Esto se observa en las líneas 12, 15 y 18: si una clase padre ya ha realizado la transacción actual es porque existe un ciclo en la ontología.

Por último, la línea 24 hace uso de `end_transaction` para finalizar la transacción mediante la línea 25, que se ejecutará únicamente en la instancia que generó la transacción. En este caso se almacena la transacción en la caché. Por último, la línea 30 evalúa el valor que debe almacenarse en la caché.

Apéndice C

Representación intermedia de ontologías en Antive Map

Una buena forma de entender este esquema es revisar el código de `XMLForClasses()`, ya que recorre todos los arrays y hashes de la representación inermedia de la ontología y sus correspondencias con las fuentes de datos, generando el XML que las representa. El siguiente fragmento de código representa la primera parte del método, que hace referencia a las estructuras de datos representadas en la figura 5.4:

```
1     // para cada clase
2     for (var i = 0; i < classes.length; i++) {
3       klass = classes[i];
4       xml += "<class about='"     +
5              klass["about"]       +
6              "' uri='"            +
7              klass["public_uri"] +
8              "'>"
9
10      // para cada fuente
11      for (var s = 0; s < klass["sources"].length; s++) {
12        sourceName = klass["sources"][s]["identifier"];
13        xml += "<properties ref='" +
14               sourceName          +
15               "'>";
16
17        // para cada campo mapeado en la fuente
18        for (field in klass["mapping"][sourceName]) {
19
20          // mapeo entre datatypeproperty y campo de la fuente
21          xml += "<property about='"                    +
22                 klass["mapping"][sourceName][field] +
23                 "' field='"                            +
```

```
24                  field                                   +
25                  "'␣/>";
26          }
27
28          xml += "</properties>";
29        }
```

Listado C.1: Generación del XML a partir de la representación intermedia de clases y *Datatype-Properties*

Puede observarse que cada iteración se realiza sobre un subconjunto definido en la iteración anterior: la línea 2 itera sobre el conjunto completo de clases; para cada una de ellas se itera sobre sus fuentes en la línea 11 y, por último, para cada fuente se itera sobre sus campos en la línea 18. Dentro de cada bucle se genera un fragmento de XML accediendo a los diferentes campos de las *hashes*.

En cuanto a la representación de propiedades de tipo *object*, la continuación del código de `XMLForClasses()` ayuda a comprender la estructura de datos representada en la figura 5.5:

```
30      // para cada objectypeproperty
31      for (var about in klass["relations"]) {
32        relation      = klass["relations"][about];
33        relation_type = relation["mapping"]["type"];
34
35        // distinguimos entre directa e indirecta
36        if (relation["mapping"]["direct"]) {
37
38            // Si hay un mapeo correcto
39            if (relation["mapping"]["keys"] &&
40                relation["mapping"]["keys"].length > 0) {
41              xml += "<"           +
42                     relation_type +
43                     "␣about='"    +
44                     about         +
45                     "'>";
46
47              // Anyadimos la etiqueta XML con los pares
48              // (clave local, clave ajena) distinguiendo si se
49              // han relacionado propiedades o campos de la fuente
50              for (var k = 0; k < relation["mapping"]["keys"].length; k++) {
51                key = relation["mapping"]["keys"][k];
52                xml += "<key␣";
53                if (key["self"])
54                    xml += "self='"    +
55                           key["self"] +
56                           "'";
57
```

```
58              if (key["self_field"])
59                  xml += "self_field='"      +
60                         key["self_field"]  +
61                         "' self_source='"  +
62                         key["self_source"] +
63                         "' ";
64
65              if (key["foreign"])
66                 xml += "foreign='"     +
67                        key["foreign"] +
68                        "' ";
69
70              if (key["foreign_field"])
71                 xml += "foreign_field='"      +
72                        key["foreign_field"]  +
73                        "' foreign_source='"  +
74                        key["foreign_source"] +
75                        "' ";
76
77              xml += " />";
78            }
79            xml += "</"+relation_type+">";
80          }
```

Listado C.2: Generación del XML a partir de la representación intermedia de *ObjectProperties* (I)

La línea 31 itera sobre las distintas *Object properties* de cada clase, y la 36 distingue entre relaciones directas (sin una tabla intermedia) o indirectas (con una tabla intermedia). Este fragmento de código corresponde a la generación de XML de relaciones directas. La línea 50 recorre todas las claves de la relación y, para cada una de ellas, se genera el XML en función de las propiedades de la clave (propiedades o campos de las clases que participan en la relación) entre las líneas 52 y 77.

Si la relación es indirecta (con una tabla intermedia) el código que se ejecuta es el siguiente:

```
81    // relacion indirecta
82    if (relation["mapping"]["keys"]) {
83      xml += "<"             +
84             relation_type +
85             " about='"     +
86             about          +
87             "'>";
88
89      // Para cada fuente intermedia
90      for (source in relation["mapping"]["keys"]) {
91        xml += "<through ref='" +
92               source +
93               "'>";
```

```
94
95          // Generamos los pares (clave local/ajena, campo de la fuente)
96          for (var k = 0; k < relation["mapping"]["keys"][source].length; k++) {
97            key = relation["mapping"]["keys"][source][k];
98            xml += "<key ";
99
100           if (key["self"])
101             xml += "self='"     +
102                    key["self"] +
103                    "' ";
104
105           if (key["foreign"])
106             xml += "foreign='"     +
107                    key["foreign"] +
108                    "' ";
109
110           xml += "field='"     +
111                  key["field"] +
112                  "' ";
113
114           xml += " />";
115         }
116         xml += "</through>";
117       }
118       xml += "</" +
119              relation_type +
120              ">";
121     }
```

Listado C.3: Generación del XML a partir de la representación intermedia de *ObjectProperties* (II)

En el caso indirecto, la generación del XML es prácticamente idéntica a la del caso directo: las claves se agrupan por fuentes (tablas intermedias), por lo que hay una iteración adicional (línea 90).

El resto del código de Antive Map, aunque extenso, consiste únicamente en relacionar los diferentes eventos de la interfaz gráfica con operaciones sobre esta estructura de datos. En caso de que una acción pueda deshacerse o cancelarse (como en las pantallas de mapeado) se crea una copia de la estructura de datos correspondiente de forma que pueda ser aceptada o rechazada.

Apéndice D

Ruby on Rails sin base de datos

La mayor parte de las aplicaciones Rails necesitan una conexión a una base de datos. Aunque Antive Core realiza consultas de este tipo, la aplicación Rails no necesita ninguna base de datos configurada.

Para indicar al entorno de trabajo que no se empleará ninguna base de datos por defecto se necesita realizar la siguiente tarea:

1. Editar `test/test_helper.rb` y cambiarlo a:

```
class Test::Unit::TestCase
  self.use_transactional_fixtures = false
  self.use_instantiated_fixtures  = false
  def load_fixtures
  end
end
```

2. Editar `config/environment.rb` y añadir la línea:

```
config.frameworks -= [ :active_record ]
```

3. Editar `config/initializers/new_rails_defaults.rb` y comentar las líneas:

```
# ActiveRecord::Base.include_root_in_json = true
# ActiveRecord::Base.store_full_sti_class = true
```

Bibliografía

[Álvarez Espinar, 2007] Álvarez Espinar, M. (2007). Introducción a la web semántica. In *V Workshop REBIUN Sobre Proyectos Digitales*, Barcelona, España. `http://www.w3c.es/Presentaciones/2005/1018-WebSemanticaREBIUN-MA/`.

[Arens et al., 1996] Arens, Y., Knobloch, C. A., Chee, C. Y., and Hsu, C. (1996). Sims: Single interface to multiple sources. Technical Report RL-TR-96-118, University of Southern California, Los Angeles, CA, USA.

[Brisaboa et al., 2002a] Brisaboa, N. R., Penabad, M. R., Ángeles Saavedra Places, and Rodríguez, F. J. (2002a). Ontologías en federación de bases de datos. *Novática*, (158):45–53.

[Brisaboa et al., 2002b] Brisaboa, N. R., Penabad, M. R., and Places, M. A. S. (2002b). *Arquitectura para Federación de Bases de Datos Documentales basada en Ontologías*. PhD thesis, Laboratorio de Bases de Datos. Dep. de Computación, Universidade da Coruña.

[Brisaboa et al., 2001] Brisaboa, N. R., Places, A. S., and Rodríguez, F. J. (2001). Arquitectura para federación de bases de datos documentales basada en ontologías. In F. Torres Rojas, J. E. Araya Monge, Y. S. S., editor, *Actas de las 4tas. Jornadas Iberoamericanas de Ingeniería de Requisitos y Ambientes del Software*, pages 252–262, Heredia (Costa Rica).

[Busse et al., 1999] Busse, S., Kutsche, R.-D., Leser, U., and Weber, H. (1999). Federated information systems: Concepts, terminology and architectures. Technical report, Technische Universität Berlin, Computergestützte InformationssystemeCIS, Berlin, Deutchland.

[Cámara, 2002] Cámara, J. C. (2002). Learning metadata standards. Recurso electrónico disponible en `http://www.iua.upf.es/~jblat/material/doctorat/students/jccbis/Ontologias.htm`. Universitat Pompeu Fabra, Barcelona, España.

[Carey et al., 1995] Carey, M., Haas, L., Schwarz, P., Arya, M., Cody, W., Fagin, R., Flickner, M., Luniewski, A., Niblack, W., Petkovic, D., Thomas, J., Williams, J., and Wimmers, E. (1995). Towards heterogeneous multimedia information systems: the garlic approach. In *Fifth International Workshop on Research Issues in Data Engineering: Distributed Object Management*, pages 124–131, Taipei, Taiwan.

[Chawathe et al., 1994] Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., and Widom, J. (1994). The tsimmis project: Integration of heterogeneous

information sources. In *Information Processing Society of Japan Conference*, pages 7–18, Tokyo, Japan.

[Codina and Rovira, 2006] Codina, L. and Rovira, C. (2006). La web semántica. In Tramullas, J., editor, *Tendencias en documentación Digital*, pages 9–54. Trea, Guijón, España.

[Cuadra Sanchez et al., 2008] Cuadra Sanchez, A., Garces Garcia, F., Fuentes Lorenzo, D., Sanchez Fernandez, L., and Reyes Ureña, M. (2008). Plataforma de monitorización avanzada basada en tecnologías web. In *XVIII Jornadas Telecom I+D*, Bilbao, España.

[Cui and O'Brien, 2000] Cui, Z. and O'Brien, P. (2000). Domain ontology management environment. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8*, page 8015, Washington, DC, USA. IEEE Computer Society.

[Davis, 1997] Davis, A. M. (1997). Software life cycle models. In Dorfman, M. and Thayer, R., editors, *Software Project Management*, pages 105–114. Institute of Electrical and Electronics Engineers Computer Society Press, Los Alamitos, CA, USA.

[Fernandez et al., 2007] Fernandez, O., Bauer, M., Black, D., Cashion, T., Pelletier, M., Showers, J., and Heinemeier Hansson, D. (2007). *The Rails Way*. Addison-Wesley Professional.

[Garcia-molina et al., 1997] Garcia-molina, H., Papakonstantinou, Y., Quass, D., Sagiv, Y., Ullman, J., Vassalos, V., and Widom, J. (1997). The tsimmis approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8:117–132.

[Genesereth et al., 1997] Genesereth, M. R., Keller, A. M., and Duschka, O. M. (1997). Infomaster: an information integration system. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 539–542, New York, NY, USA. ACM.

[Grant et al., 1993] Grant, J., Litwin, W., Roussopoulos, N., and Sellis, T. (1993). Query languages for relational multidatabases. *The VLDB Journal*, 2(2):153–172.

[Gray et al., 1997] Gray, W. A., Preece, A., Fiddian, N. J., Gray, W. A., Bench-Capon, T. J. M., Shave, M. J. R., Azarmi, N., Wiegand, M., Ashwell, M., Beer, M., Cui, Z., Diaz, B., Embury, S. M., Hui, K., Jones, D. M., Kempt, G. J. L., Lawsod, E. W., Lunn, K., Martit, P., Shaot, J., and Visser, P. R. S. (1997). Kraft: Knowledge fusion from distributed databases and knowledge bases. In *Proceedings of Database and Expert System Applications Conference (DEXA' 97*, pages 682–691. IEEE Press.

[Haas et al., 1997] Haas, L. M., Miller, R. J., Niswonger, B., Roth, M. T., Schwarz, P., Schwarz, R. P. M., and Wimmers, E. L. (1997). Transforming heterogeneous data with database middleware: Beyond integration. *IEEE Data Engineering Bulletin*, 22:31–36.

[Lakshmanan et al., 2001] Lakshmanan, L. V. S., Sadri, F., and Subramanian, S. N. (2001). Schemasql: An extension to sql for multidatabase interoperability. *ACM Trans. Database Syst.*, 26(4):476–519.

[Levy et al., 1996] Levy, A. Y., Rajaraman, A., and Ordille, J. J. (1996). Querying heterogeneous information sources using source descriptions. In *Proceedings of the Twenty-second International Conference on Very Large Databases*, pages 251–262, Bombay, India. VLDB Endowment, Saratoga, CA.
`http://www.cs.washington.edu/homes/alon/files/vldb96-im.pdf`.

[Marshall et al., 2007] Marshall, K., Pytel, C., and Yurek, J. (2007). *Pro Active Record: Databases with Ruby and Rails*. Apress.

[Mena et al., 1996] Mena, E., Kashyap, V., Sheth, A., and Illarramendi, A. (1996). Observer: An approach for query processing in global information systems based on interoperation across pre-existing ontologies. *Cooperative Information Systems, IFCIS International Conference on*, 0:14.
`http://doi.ieeecomputersociety.org/10.1109/COOPIS.1996.554955`.

[Noy and mcguinness, 2001] Noy, N. F. and mcguinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology. Technical report, Stanford University, Stanford, CA, USA.
`http://www.ksl.stanford.edu/people/dlm/papers/ontology101/ontology101-noy-mcguinness.html`.

[Porteneuve, 2007] Porteneuve, C. (2007). *Prototype and script.aculo.us*. The Pragmatic Programmers.

[Richardson and Ruby, 2007] Richardson, L. and Ruby, S. (2007). *RESTful Web Services*. O'Reilly Media, Inc.

[Ruby et al., 2008] Ruby, S., Thomas, D., and Hansson, H. (2008). *Agile Web Development with Rails*. The Pragmatic Programmers.

[Sheth and Larson, 1990] Sheth, A. P. and Larson, J. A. (1990). Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22:183–236.

[Thomas, 2005] Thomas, D. (2005). *Programming Ruby*. The Pragmatic Programmers.

[Tomasic et al., 1997] Tomasic, A., Amouroux, R., Bonnet, P., Kapitskaia, O., Naacke, H., and Raschid, L. (1997). The distributed information search component (disco) and the world wide web. *SIGMOD Rec.*, 26(2):546–548.

[Tomasic et al., 1995] Tomasic, A., Raschid, L., and Valduriez, P. (1995). Scaling heterogeneous databases and the design of disco. Technical Report 2704, Institut National de Recherche en Informatique et en Automatique, Le Chesnay, France.

[Wirdemann and Baustert, 2007] Wirdemann, R. and Baustert, T. (2007). Desarrollo rest con rails. Recurso electrónico.
`http://www.b-simple.de/download/restful_rails_es.pdf`.