CMSs, Linked Data and Semantics: A Linked Data Mashup over Drupal for Personalized Search

Aikaterini K. Kalou, Dimitrios A. Koutsomitropoulos and Georgia D. Solomou

High Performance Information Systems Laboratory (HPCLab),
Computer Engineering and Informatics Dpt., School of Engineering,
University of Patras, Building B, 26500 Patras-Rio, Greece

`{kaloukat, kotsomit, solomou}@hpclab.ceid.upatras.gr`

Abstract. Semantic mashups are a representative paradigm of Web applications which highlight the novelties and added-value of Semantic Web technologies, especially Linked Data. However, Semantic Web applications are often lacking desirable features related to their 'Web' part. On the other hand, in the world of traditional web-CMSs, issues like front-end intuitiveness, dynamic content rendering and streamlined user management have been already dealt with, elaborated and resolved. Instead of reinventing the wheel, in this paper we propose an example of how these features can be successfully integrated within a semantic mashup. In particular, we re-engineer our own semantic book mashup by taking advantage of the Drupal infrastructure. This mashup enriches data from various Web APIs with semantics in order to produce personalized book recommendations and to integrate them into the Linked Open Data (LOD) cloud. It is shown that this approach not only leaves reasoning expressiveness and effective ontology management uncompromised, but comes to their benefit.

1 Introduction

Traditional mashups [7] are Web applications that aggregate data or functionality from various online third-party sources, especially Web APIs. With the prevalence of the Semantic Web, mashups are 'transformed' to semantic mashups which consume data from interlinked data sources on the cloud. Nevertheless, a semantic mashup can be considered as any mashup that employs semantic web technologies and ideas in any part of its design, architecture, functionality or presentation levels.

The Linked Open Data (LOD) project [10] has successfully brought a great amount of data to the Web. The availability of interlinked data sets encourages developers to reuse content on the Web and alleviates them from the need to discover various data sources. In the case of semantic mashups, contribution to the LOD effort can come by appropriately combining data from Web APIs with semantics and then providing them as Linked Data.

As is often the case with any Semantic Web application, semantic mashup development usually puts too much effort in the bottoms-up construction of elaborate, knowledge intensive set-ups. This kind of applications often dwells on high-end rea-

soning services, efficient rule processing and scalability over voluminous data, thus hardly leaving any room for traditional Web development.

This gap can be bridged by traditional web content management systems (CMSs) which offer an up-to-date and tailored web infrastructure and leave more room for the designer to concentrate on successful content production and delivery, rather than technical details. As they form the spearhead of Web 2.0, it might then feel natural to employ them as a basis for Semantic Web applications, but this presents a series of challenges that it is not always straightforward to overcome.

In this paper, we therefore propose how such applications and CMSs can be integrated, by presenting Books@HPClab, a semantic mashup application, which we purposely establish on top of the Drupal CMS. Books@HPClab [6, 13] has been initially developed from scratch and offers personalization features to users searching for books from various data sources. The key concept of this mashup is that it gathers information from Amazon and Half eBay Web APIs, enriches them with semantics according to an ontology (*BookShop* ontology) and then employs OWL 2 reasoning to infer matching preferences. The triplified book metadata are also linked to other resources, thus becoming more reusable and effectively more sharable on the LOD cloud.

The following text is organized as follows: in Section 2, we start by discussing the desirable properties of CMSs that make them suitable as a basis for developing Semantic Web applications. In Section 3, we describe in detail the BookShop ontology. Furthermore, in Section 4, we explain how we proceeded with the actual integration and discuss how we addressed the problems arising in this process, putting particular focus on the data workflow, reasoner integration and provision of Linked Data. Next, in Section 5, we briefly illustrate the features and the functionality of our application, now completely re-engineered over Drupal, by outlining an indicative application scenario. Finally, Section 6 summarizes our conclusions and future work.

2   CMS as a Semantic Web Infrastructure

A typical CMS generally comes with the ability to help and facilitate the user, even the non-technical one, in various ways. It always ensures a set of core features [12] such as:

— *Front-end Interface*: The developer community of all available CMSs invests significantly in the layout, appearance and structure of the content that is created and delivered by a CMS. Therefore, content remains completely separate from appearance. To this end, users of CMSs can select from a great variety of well-designed templates.
— *User management*: CMSs offer also considerable advantages in regard to user administration and access issues. It can be easily controlled whether users are allowed to register on a web application as well as what kind of privileges they can have, by providing access layers and defining sections of the web application as public or private. Moreover, CMSs allow for assigning roles to users so as to involve them in the workflow of web content production.

— *Dynamic content management*: Usually a CMS relies on an RDBMS to efficiently store and manage data and settings, which are then used to display page content. So, the installation of a CMS always involves setting-up a database schema in the corresponding SQL server. The database schema actually used, varies depending on the CMS.

— *Modular design*: CMSs follow architecture styles such as *Model-View-Controller* (MVC) or *Presentation-Abstraction-Control* (PAC) that permit the organization of code in such a way that business logic and data presentation remain separate. This enables the integration of small, standalone applications, called *modules*, which accomplish a wide variety of tasks. These artifacts can be easily and simply installed/uninstalled and enabled/disabled in the core of CMSs. Modularity is one of the most powerful features and the one that saves the most development effort.

— *Caching*: It is also important that most CMSs offer cache capabilities to users/developers. Thus, CMS-based web applications can have fast response times by caching frequently requested content and reducing their overhead.

Features such as these, that contemporary CMSs unsparingly offer, are exactly the ones sometimes neglected by Semantic Web applications. In the case of our work, we chose to integrate Books@HPClab within the core of Drupal CMS [14]. Regardless of Drupal's semantic character, other significant advantages such as flexibility and scalability make it stand out from the large pool of CMSs. Besides, Drupal has been used before as a basis for offering Linked Data services [4]. Finally, Drupal can be viewed not only as a CMS, but also as a content management *framework*, by accommodating development of any type of web application.

3 Ontology Design

Taking into account the kind of metadata offered by Amazon and Half eBay responses, we designed the core ontology BookShop shown partially in Figure 1. BookShop contains five main classes *Book*, *Author*, *Offer*, *User* and *Modality*.
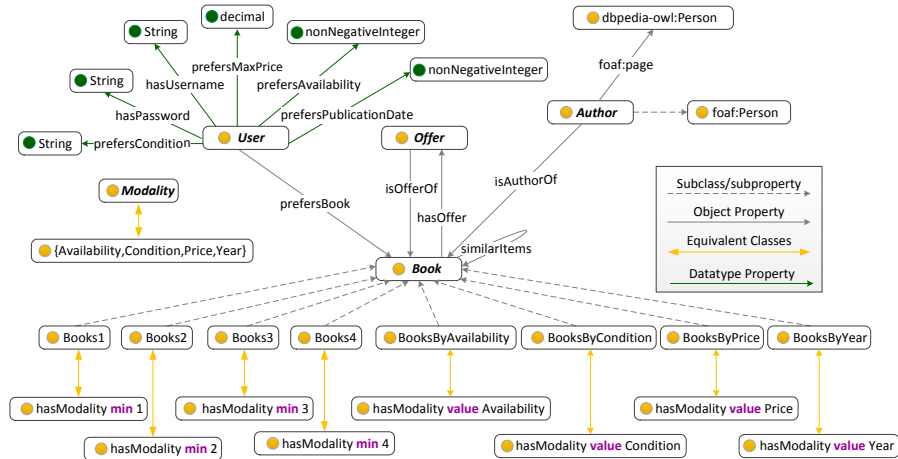

Fig. 1. BookShop Ontology

In our ontology, the class *User* is meant to express user profiles. We capture the preferences of each user in this class, such as preferable condition, preferable minimum availability, preferable minimum publication year and preferable maximum price (preference criteria). All this data about users are represented as datatype properties.

The class *Book* represents all book items that are gathered from Amazon and Half eBay sales markets. A reasoner is responsible for entailing which books match what criteria in the current user profile and classifies them accordingly (*BooksByAvailability*, *BooksBycondition, BooksByPrice, BooksByYear*). The kind of a matched criterion is represented by the members of the *Modality* class. Given the cardinality restrictions on the *hasModality* property, the books are finally classified depending on the number of satisfied preference criteria (*Books...Books4*). For example, the *Books1* class contains all the books that match at least one of the preference criteria.

4 System Design and Integration

In this section, we present the overall design of our application and its interaction with all necessary external and embedded components. We also describe thoroughly the main issues we had to put up with and how we addressed each one of them.

4.1 Architecture and Integration Challenges

The modular philosophy of a CMS allows us to extend its capabilities with ready-made modules and to reuse them for our purposes. To this end, we utilize the *AmazonStore module*[1] that offers an attractive wrapper and front-end for the Amazon Web API. We have extended this module so as to include support for eBay as well. We also make use of the *WebForm module*[2], which supports form-based data collection and is used as the initiating point for constructing user profiles. The architecture of our re-engineered mashup is illustrated in Figure 2.

In order to re-engineer our semantic mashup on top of Drupal so as to leverage all CMSs' core features mentioned in Section 2, we encountered a series of challenges, originating from the fact that CMSs are usually not semantics-aware. Although latest versions of Drupal offer some inherent semantic features [3], in our implementation we needed to put a strong focus on reasoning, ontology management as well as data interlinking, which is beyond Drupal's state-of-the-art (or any other CMS's for that matter). All these issues are analysed in the following subsections and summarized below:

— *User profile construction and maintenance*: Managing users as well as their profiles are common issues that have already been addressed within a web CMS. In the context of our application, the issue is how we can map and maintain the relational user profiles in terms of OWL 2 expressions (see section 4.2).

---

[1] http://drupal.org/project/amazon_store
[2] http://drupal.org/project/webform

- *Synchronizing relational and ontology back-ends*: Semantic Web applications deal with content that needs to be semantically expressed. The manipulation of semantic data should be consistent with web content management and delivery policies which are based on robust relational back-ends in the context of a web-CMS (see Section 4.2).
- *Reasoner integration*: Once embedded within a CMS, a Semantic Web application must pay special attention to the efficient and interoperable communication with a reasoning service (see Section 4.3).
- *Data linking*: A semantic mashup, which aggregates a significant amount of ontological data, can be a worthy contribution to the LOD cloud, even though it is implemented within a CMS framework (see Section 4.4).
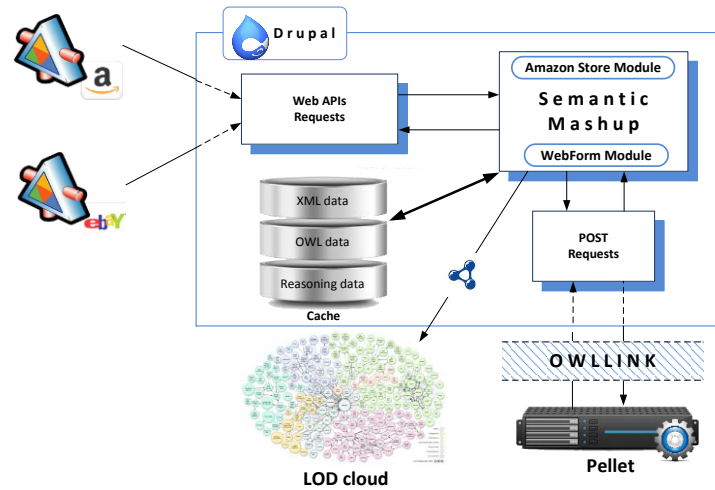
Fig. 2. Architecture and communication flow for integrating Semantic Mashup with Drupal

4.2 Data Collection and Storage

In the context of our application, with the term *data*, we mean the conjunction of user profiles, externally collected information and ontological data before and after the reasoning process. In this sub-section, we review in detail the data collection and storage workflow, and all the existing Drupal modules that we have exploited to this end.

Regarding user profile construction, user preferences are collected using web forms, designed with the aid of the WebForm module. A unique ID is assigned to each user. In addition to user preferences, each user has to set his unique password and username, as well as his e-mail address so as to get notifications from the application. All this user-related information is stored in tables of the relational database.

In order to perform reasoning however, these preferences have to be translated into semantically rich expressions, which form the ontological profile of each user. In our

case, we retrieve user preferences from the database and then we construct the profile on-the-fly, by mapping preferences to a set of OWL 2 expressions.

In order to collect book data from Amazon and Half eBay, we have extended the existing functionality of AmazonStore module by adding communication ability with the Half eBay Web API. Whenever a user types a keyword and sends a *searching call*, the searching process starts to query data from Amazon Web Services (AWS), and especially from the *US E-Commerce Service (ECS)* via functions available by the AmazonStore module. In general, a request to Amazon may have many thousands of results. Returning all these items at once may be inefficient and impractical. To this end, it is defined that Amazon operations return paginated results, 10 results per page.

Once our application completes the search process at Amazon, it starts searching Half eBay: for each book returned by Amazon, we find additional offers that may be available at Half eBay. We use the *eBay shopping Web Services* and particularly, the *FindHalfProducts* operation. The interaction with the eBay shopping API is based on the REST-protocol and the exchange of URL requests and XML files-responses. By augmenting the data storage policy of AmazonStore module, we save the Amazon XML results, enriched with additional book-offers from Half eBay, in the *XML data cache* (see Figure 2).

Next, search results need to be transformed into the OWL word in order to enable inferences. This conversion adheres to our BookShop ontology schema and is achieved via XSLT. The transformed ontological data are cached in the *OWL data cache*. In order to achieve personalization, OWL data as well as the ontological user profile are sent to the remote reasoning service. Finally, the inferred knowledge is stored at the *reasoning cache*.

An algorithm (shown in Table 1) is responsible for synchronizing between the caches, which, apart from checking for repeating queries, additionally expunges reasoning cache whenever a user updates his profile. Note that the cache can be flushed after a configurable amount of time (in this case, 24 hours). A profile update initiated by a user causes the removal from cache of all reasoning results related to the particular profile u, i.e. $\mathcal{R} \rightarrow \mathcal{R} / \{r_{*,u}\}$, where * denotes all o_q.

Table 1. Algorithm for the synchronization of data storage

$\mathcal{B}$: XML book data cache, b_q: XML book data for query q
$\mathcal{O}$: Ontological book data cache, o_q: ontological book data for query q
$\mathcal{R}$: Reasoner results cache, $r_{o_q,u}$: reasoner results for o_q and user profile u

if $\{b_q\} \nsubseteq \mathcal{B}$	**if** $\{b_q\} \subseteq \mathcal{B}$, $\{o_q\} \subseteq \mathcal{O}$ **and** $r_{o_q,u} \nsubseteq \mathcal{R}$
then $b_q \rightarrow$ **get_amazon_data** (q)	*//since b_q is in $\mathcal{B}$, o_q will always be in $\mathcal{O}$*
$\quad\quad b_q \rightarrow$ **get_ebay_data** (q)	**then** $r_{o_q,u} \rightarrow$ **invoke_reasoner** (o_q, u)
$\quad\quad \mathcal{B} \rightarrow \mathcal{B} \cup \{b_q\}$	$\quad\quad \mathcal{R} \rightarrow \mathcal{R} \cup \{r_{o_q,u}\}$
$\quad\quad o_q \rightarrow$ **triplify** (b_q)	$\quad\quad$ **return** $r_{o_q,u}$
$\quad\quad \mathcal{O} \rightarrow \mathcal{O} \cup \{o_q\}$	
$\quad\quad r_{o_q,u} \rightarrow$ **invoke_reasoner** (o_q, u)	
$\quad\quad \mathcal{R} \rightarrow \mathcal{R} \cup \{r_{o_q,u}\}$	**if** $\{b_q\} \subseteq \mathcal{B}$, $\{o_q\} \subseteq \mathcal{O}$ **and** $r_{o_q,u} \subseteq \mathcal{R}$
$\quad\quad$ **return** $r_{o_q,u}$	**then return** $r_{o_q,u}$

The adoption of the database caching and data replication strategy allows CMS modules to remain oblivious to the ontology data and lets them to operate on their own data cache. This caching idea, which is also carried over to reasoning results, actually improves the effective reasoning throughput by keeping reasoner engagement to a minimum.

4.3   Reasoner Integration

Most OWL 2 reasoners (like, Pellet, FaCT++ and HermiT) are traditionally deployed directly in-memory and interaction is performed by means of a java-based API. Although a PHP-to-Java bridge[3] is available, there are many reasons why one may want to keep reasoning services logically and/or physically separated [8]. Among them, the need for interoperability and independence from the actual programming language are of particular importance for integration with a CMS.

In our implementation, we use OWLlink [9] as the reasoner communication protocol of choice and its implementation, the OWLlink API [11] that helps us deploy a true 3-tier architecture. OWLlink offers a consistent way of transmitting data to and receiving responses from the most popular Semantic Web reasoners, in a REST-like manner and over HTTP. Potential communication overhead that may be introduced with this approach can be alleviated by freeing up resources as a consequence of delegating computationally hard reasoning tasks to another tier [8]. Moreover, Drupal offers us generic function implementations that can be used to wrap and construct HTTP requests, like `drupal_http_request`. Messages are encoded in XML format and Pellet is used as the inference engine of choice.

The interaction between the OWLlink server and our client-application consists of four main request-response messages. Firstly, we allocate a Knowledge Base (KB) within the OWLlink server by sending a `CreateKB` request. The unique user id is assigned as an identifier to the KB, in order to logically separate knowledge bases under the same reasoner. In the same message, we embed a `LoadOntologies` request so as to load the BookShop ontology schema into the given KB by reading the ontology file.

Next, we add the ontological user profile and the OWL data results for a specific query by sending two distinct `Tell` requests to the OWLlink server. At this point user preferences are fetched from the DB and are used to construct the ontological user profile on the fly, which amounts to a set of OWL 2 restrictions (see Table 2). Both user profile and OWL data are encoded in OWL/XML syntax. In order to get the inferred knowledge from the reasoner, we send a `GetFlattenedInstances` request. Its purpose is to retrieve all books that satisfy up to four preference criteria (instances of *Books1*, *Books2*, *Books3* and *Books4* classes). The `direct=true` parameter ensures that the above sets will be mutually disjoint, i.e. they will include only unique book instances. Finally the KB is destroyed by issuing a `ReleaseKB` request within the same message.

---

[3]   http://php-java-bridge.sourceforge.net/pjb/

Table 2 summarizes all the messages that are exchanged between our application and the OWLlink server.

Table 2. Interaction with OWLlink server

	No. 1	No. 2	No. 3	No. 4
Request	`CreateKB` kb=[*User_ID*] `LoadOntologies` IRI=[*BookShop ontology*]	`Tell` *preferences* BooksByPrice ≡ ∃hasOffer.(∃offerPrice.[≤*user_pref*]) BooksByCondition… BooksByAvailability... BooksByYear...	`Tell` data OWL Book data from cache (query results)	`GetFlattenedInstances` direct=″true″ class IRI=*{Books1, Books2, Books3, Books4}* `ReleaseKB` kb=[*User_ID*]
Response	`ResponseMessage` OK	`ResponseMessage` OK	`ResponseMessage` OK	`SetofIndividuals` {*1..4*} `NamedIndividuals` IRI=[*Book resource URL*]

4.4 Linked Data Service

Usually, LOD can be considered as a significant data source and a Semantic Web tool can consume them in order to construct a mashup application. The reverse is also desirable and in the case of Books@HPCLab, we interlink aggregated data with other available web resources, thus contributing to the LOD cloud.

In order to publish Linked Data, we follow the Linked Data principles, as they are explicitly described in [5]. In order to identify real-world entities, either people or abstract concepts, we assign HTTP URIs to them. To encompass the book items, we mint HTTP URIs using the following pattern that is based on the application's namespace: First, each book item is uniquely identified by a single URI, describing the item itself. Then, we assign to each book another URI that describes the item and has an HTML representation, appropriate for consumption by humans. Next, another URI is given to the book item in order to describe it and provides an RDF/XML representation for machine readbility.

Following this URI pattern, for the case where a book item has ASIN number 0890425558, we end up with the three next URIs:

— `http://levantes.hpclab.ceid.upatras.gr:8000/bookmashup/resource/0890425558`
— `http://levantes.hpclab.ceid.upatras.gr:8000/bookmashup/item/0890425558` (HTML)
— `http://levantes.hpclab.ceid.upatras.gr:8000/bookmashup/data/0890425558` (RDF)

Moreover, these HTTP URIs are dereferenceable by using HTTP content negotiation (HTTP *303 See Other* redirects, see Fig. 3).
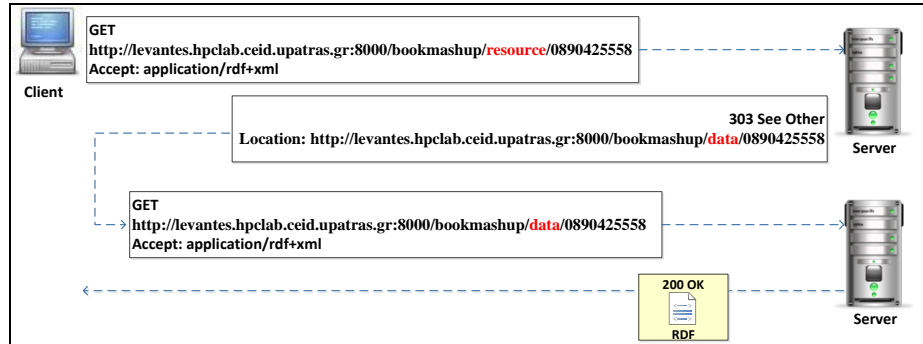
Fig. 3. A complete example of content negotiation

To associate our data with other data sets on the Web, we interlink our entities with others by adding RDF external links. More precisely, in the case of book offers, relationship links are added so as to point to the bookstore origin. We also inject DBpedia HTTP URIs into author RDF descriptions originally available from the Web APIs. The following figure (Fig. 4) depicts an excerpt of published RDF data with the external RDF links.

Fig. 4. Interlinking data set of Books@HPClab with external data sets
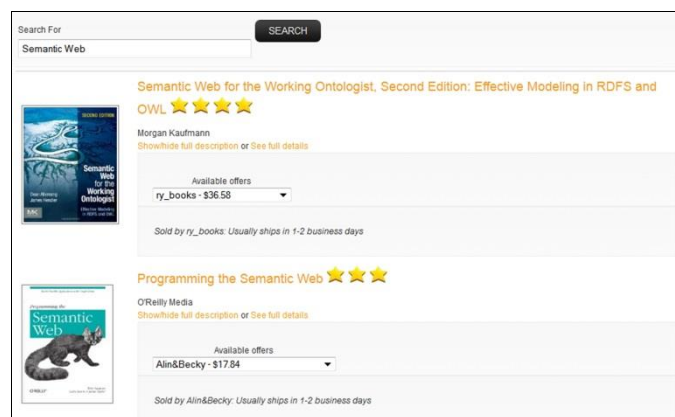
5 A Usage Scenario

When a user visits our app for the first time, he has to register by filling a form with his username and e-mail. An administrator then enables the account and a password is sent to the user at the specified mail address.

Fig. 5. Collecting user preferences

After successful authorization, logged users can set their profile using the Web-Form module. The form fields correspond to user preferences and include: book condition ("new" or "used"), maximum book price, earliest publication year and maximum availability (Fig. 5). A user can update his profile at any time. Note also that if a user does not define preferences, the application behaves as a standard book mashup and the reasoner is never engaged.

Fig. 6. Result list and preference ranking (stars)

6 Conclusions and Future Work

Integration of Semantic Web applications with a CMS is not always straightforward. In order to achieve a seamless alignment, a series of issues has first to be resolved, and in this paper we have indicated exactly how this can be achieved in the case of our semantic mashup. Primarily, the semantic-oblivious nature of most CMSs calls for the explicit manipulation of semantically enriched data, which can be far from trivial, especially when their robust relational back-end is to be taken advantage of. Additionally, incorporating a reasoning infrastructure needs to be carefully designed as there may be substantive trade-offs involved.

Nevertheless, by combing the best of both worlds, the developer can genuinely focus on the internals of the Semantic Web implementation and assign web content management and delivery on tried and true existing frameworks, instead of wasting time and effort. It turns out that, by investing in this integration, even the semantic aspects can benefit e.g. from data caching or reasoner delegation, thus making a virtue of necessity. In addition, the CMS infrastructure can be inexpensively utilized in order to align our ontological data with the Linked Data principles, associate them with additional resources and make them available to the LOD cloud.

As a next step, we intend to pay a closer look at the deeper integration with relational data in a means to avoid data replication and to save storage space in the database. Although our caching approach appears to work well in practice, it is not clear whether the separate cache maintenance really compensates for on-the-fly transformations or how does it compare with virtualized graph access as in D2RQ [2]. The RESTful style of reasoner communication also allows for investigating potential alternatives with a view on scalability, like rule-based triple stores [13]. To this end, an assessment of our system's performance and efficiency is in order. We also intend to wrap additional RESTful web service functionality around our semantic mashup as a means for other applications to consume and exchange Linked Data without manual intervention. Finally, we plan to package our prototype as a totally independent CMS module, thus allowing its smooth installation and reuse by other developers.

7 References

1. Berrueta, D., Phipps, J. (eds.): Best Practice Recipes for Publishing RDF Vocabularies. W3C Working Group Note (2008)
2. Bizer, C., Seaborne, A.: D2RQ-treating non-RDF databases as virtual RDF graphs. In: 3rd Int. Semantic Web Conference, (2004)
3. Bratsas, C., Bamidis, P., Dimou, A., Antoniou, I., Ioannidis, L.: Semantic CMS and Wikis as Platforms for Linked Learning. In: 2nd Int. Workshop on Learning and Education with the Web of Data – 24th Int. World Wide Web Conference, (2012)
4. Corlosquet, S., Delbru, R., Clark, T., Polleres, A., Decker, S.: Produce and Consume Linked Data with Drupal!. In: 8th Int. Semantic Web Conference, pp. 763-778, Springer (2009)

5. Heath, T., Bizer, B.: Linked Data: Evolving the Web into a Global Data Space (1st edition). Synthesis Lectures on the Semantic Web: Theory and Technology, vol. 1, pp. 1-136. Morgan & Claypool (2011)

6. Kalou, K., Pomonis, T., Koutsomitropoulos, D., Papatheodorou, T.S.: Intelligent Book Mashup: Using Semantic Web Ontologies and Rules for User Personalisation. In: 4th IEEE Int. Conference on Semantic Computing - Int. Workshop on Semantic Web and Reasoning for Cultural Heritage and Digital Libraries, pp. 536-541, IEEE (2010)

7. Koschmider, A., Torres, V., Pelechano, V.: Elucidating the Mashup Hype: Definition, Challenges, Methodical Guide and Tools for Mashups. In: 2nd Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web, (2009)

8. Koutsomitropoulos, D., Solomou, G., Pomonis, T., Aggelopoulos, P., Papatheodorou, T. S.: Developing Distributed Reasoning-based Applications for the Semantic Web. In: 24th IEEE Int. Conference on Advanced Information and Networking - Int. Symposium on Mining and Web, pp. 593-598. IEEE (2010)

9. Liebig, T., Luther, M., Noppens, O., Wessel, M.: OWLlink. Semantic Web Journal, vol. 2, 23-32 (2011)

10. Linked Open Data Project. http://linkeddata.org/.

11. Noppens, O., Luther, M., Liebig, T.: The OWLlink API-Teaching OWL Components a Common Protocol. In: 7th Workshop on OWL: Experiences and Directions, 614, CEUR Workshop Proceedings, (2010)

12. Patel, S.K., Rathod, V.R., Prajapati, J.B.: Performance Analysis of Content Management Systems-Joomla, Drupal and WordPress. International Journal of Computer Applications, vol. 21, 39-43 (2011)

13. Solomou, G., Kalou, K., Koutsomitropoulos, D., and Papatheodorou, T.S.: A Mashup Personalization Service based on Semantic Web Rules and Linked Data. In: 7th Int. Conference on Signal Image Technology and Internet Information Systems, pp. 89-96. IEEE (2011)

14. Tomlinson, T.: Beginning Drupal 7. Apress (2010)