Developing Semantic Web Applications with the OntoWiki Framework

Norman Heino, Sebastian Dietzold, Michael Martin, and Sören Auer

Abstract. In this paper, we introduce the OntoWiki Application Framework for developing Semantic Web applications with a strong emphasis on collaboration. After presenting OntoWiki as our main show case for the framework, we give both an architectural overview and a detailed view on the included components. We conclude this paper with a presentation of different use cases where the framework was strongly involved.

Introduction

Web application development usually begins with clarifying requirements, goals and usage scenarios. Today more than ever, requirements of modern Web applications lead to a strong need for semantic technologies. Depending on the context of the Web application, expectations for integrating those technologies vary greatly. They can range from advantages in search handling, categorization and content management to flexibility in handling different data schemes. Particularly, semantic technologies have the potential to facilitate data exchange between Web applications and allow them to be used together in unforeseeable ways.

In this paper we present a framework for developing Semantic Web applications. The OntoWiki Application Framework has been developed with applications in mind that have a strong emphasis on collaboration. As the OntoWiki Application Framework is a successor of OntoWiki, a visual Semantic Wiki [2], we will first introduce OntoWiki. The role of OntoWiki is, however, not limited

Norman Heino, Sebastian Dietzold, Michael Martin, and Sören Auer
Institute of Computer Science
University of Leipzig
Johannisgasse 26
04103 Leipzig
e-mail: {heino,dietzold,martin,auer}@informatik.uni-leipzig.de

S. Schaffert et al. (Eds.): Networked Knowledge - Networked Media, SCI 221, pp. 61–77.
springerlink.com

to a show case for the OntoWiki Application Framework, but it is intended as a generic management backend for Semantic Web applications.

The paper is structured as follows. In section 1 we will give a short introduction to the central goals of OntoWiki. Section 2 will present the OntoWiki Application Framework including its components and features. Subsequently, in section 3 we demonstrate OntoWiki usage in three different application scenarios. The article concludes with lessons learned from the use cases and future developments.

1 OntoWiki – A Visual Semantic Wiki

In this section we present OntoWiki. It is a tool which, briefly speaking, supports presentation and knowledge engineering in a Web environment. We will sketch central issues which have resulted in the development of the tool, explain why it is called OntoWiki and outline the problems while using conventional Wiki systems. Subsequently, we explain the major goals of OntoWiki, some general use cases and describe existing views and workflows.

1.1 OntoWiki – Not a Classical Wiki

The driving force behind OntoWiki development was the need of a Web tool for rapid and simple knowledge acquisition in a collaborative way. Therefore, technologies were required for presenting information in a human-readable and machine-interpretable fashion. The tool presented is called OntoWiki, since it is inspired by classical Wiki systems. Its design, however, is independent and complementary to conventional Wiki technologies. The approach taken with OntoWiki differs from previously emerged strategies to integrate Wiki systems and the Semantic Web (cf. [4, 3, 8, 10, 12]). In these works it is proposed to integrate RDF triples into text-based Wiki systems by means of a special syntax. It is a straightforward combination of existing Wiki systems and the Semantic Web knowledge representation paradigms. Yet, we see the following obstacles:

Usability: The main advantage of Wiki systems is their unbeatable usability. Adding more and more syntactic possibilities counteracts ease of use for editors.

Redundancy: To allow the answering of real-time queries to the knowledge base, statements have to be additionally kept in a triple store. This introduces a redundancy, which complicates the implementation.

Evolution: As a result of storing information in both Wiki texts and triple store, supporting evolution of knowledge is difficult.

In contrast to other semantic Wiki approaches, in OntoWiki text editing and knowledge engineering (i. e. working with structured knowledge bases) are not mixed. Instead, OntoWiki directly applies the Wiki paradigm

of "making it easy to correct mistakes, rather than making it hard to make them" [9] to collaborative management of structured knowledge. This paradigm is achieved by interpreting knowledge bases as *information maps* where every node is represented visually and interlinked to related resources. Furthermore, it is possible to enhance the knowledge schema gradually as well as the related instance data agreeing on it. As a result, the following requirements have been determined for OntoWiki:

Intuitive display and editing of instance data should be provided in generic ways, yet enabling means for domain-specific presentation of knowledge.
Semantic views allow the generation of different views and aggregations of the knowledge base.
Versioning and evolution provides the opportunity to track, review and roll-back changes selectively.
Semantic search facilitates easy-to-use full-text searches on all literal data, search results can be filtered and sorted (using semantic relations).
Community support enables discussions about small information chunks. Users are encouraged to vote about distinct facts or prospective changes.
Online statistics interactively measures the popularity of content and activity of users.
Semantic syndication supports the distribution of information and their integration into desktop applications.

OntoWiki enables the easy creation of highly structured content by distributed communities. The following points summarize some limitations and weaknesses of OntoWiki and thus characterize the application domain:

Environment: OntoWiki is a Web application and presumes all collaborators to work in a Web environment, possibly distributed.
Usage Scenario: OntoWiki focuses on knowledge engineering projects where a single, precise usage scenario is either initially (yet) unknown or not (easily) definable.
Reasoning: Application of reasoning services was (initially) not the primary focus.

1.2 Generic and Domain-Specific Views

OntoWiki can be used as a tool for presenting, authoring and managing knowledge bases adhering to the RDF data model. As such, it provides generic methods and views, independent of the domain concerned. Two coarse-grained generic views included in OntoWiki are the resource view and the list view. While the former is generally used for displaying all known information about a resource, the latter can present a set of resources, typically instances of a certain concept. That concept not necessarily has to be explicitly defined as `rdfs:Class` or `owl:Class` in the knowledge base. Via its facet-based browsing, OntoWiki allows the construction of complex concept

definitions, with a pre-defined class as a starting point by means of property value restrictions. These two views are sufficient for browsing and editing all information contained in a knowledge base in a generic way.

For domain-specific use cases, OntoWiki provides an easy-to-use extension interface that enables the integration of custom components. By providing such a custom view, it is even possible to hide completely the fact that an RDF knowledge base is worked on. This permits OntoWiki to be used as a data-entry frontend for users with a less profound knowledge of Semantic Web technologies.

1.3 Workflow

With the use of RDFS [5] and OWL [11] as ontology languages, resource definition is divisible into different layers: a terminology box for conceptual information (i. e. classes and properties) and an assertion box for entities using

Fig. 1 The list and details view in OntoWiki

the concepts defined (i. e. instances). There are characteristics of RDF which, for end users, are not easy to comprehend (e. g. *classes* can be defined as *instances* of `owl:Class`). OntoWiki's user interface, therefore, provides elements for these two layers, simultaneously increasing usability and improving a user's comprehension for the structure of the data.

After starting and logging in into OntoWiki with registered user credentials, it is possible to select one of the existing ontologies. The user is then presented with general information about the ontology (i. e. all statements expressed about the knowledge base as a resource) and a list of defined classes, as part of the conceptual layer.

By selecting one of these classes, the user receives a list of resources that are instances of it. In figure 1 the class `Student` has been selected and yields a list of students being either instance of `Student` directly or of its subclass `PhDStudent`; OntoWiki applies basic `rdfs:subClassOf` reasoning automatically. After selecting an instance from the list – or alternatively creating a new one – it is possible to manage (i. e. insert, edit and update) information in the details view, which is depicted in figure 1 as well.

OntoWiki focuses primarily on the assertion layer, but also provides ways to manage resources on the conceptual layer. By enabling the visualization of schema elements, called *System Classes* in the OntoWiki nomenclature, conceptional resources can be managed in a similar fashion as instance data. One of the missing features for schema management is a knowledge base consistency check, which will be included as part of the upcoming reasoning support in the near future.

2 The OntoWiki Application Framework

In the previous section we have shown how OntoWiki can be used as a Semantic Wiki. In order to render its functionality, OntoWiki relies on several APIs that are also available to third-party developers. Usage of these programming interfaces enables them to extend, customize and tailor OntoWiki in several ways. In this section we describe the OntoWiki Application Framework that builds the foundation for OntoWiki and related applications. To get an idea as to what can be achieved with the framework, we refer to the use cases described in section 3.

2.1 Architecture Overview

As depicted in figure 2, the OntoWiki Application Framework consists of three separate layers. The persistence layer consists of the Erfurt API which provides an interface to different RDF stores. In addition to the Erfurt API, the application layer is built by a) the underlying Zend Framework[1] and b) an

[1] `http://framework.zend.com/`

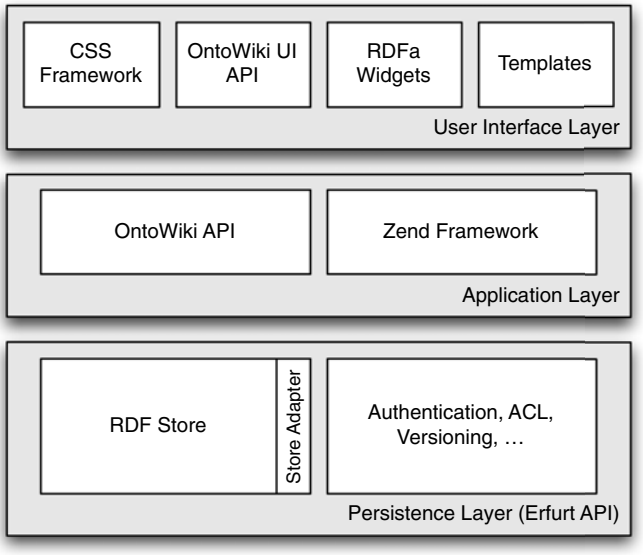

Fig. 2 The OntoWiki Application Framework with its three layers: persistence layer, application layer, user interface layer

API for OntoWiki extension development. With the exception of templates, the user interface layer is primarily active on the client side, providing the CSS framework, a JavaScript UI API, RDFa widgets and HTML templates generated on the Web-server side.

2.2 Persistence Layer

Persistent data storage as well as associated functionality such as versioning and access control are provided by the Erfurt API. This API consists of the components described in the subsequent paragraphs.

2.2.1 Authentication and Access Control Components

For Semantic Web applications it might be useful to have a means of authenticating users against an RDF store, instead of a database table. Erfurt therefore includes an authentication component that provides an API for user management.

Although Leuf and Cunningham define openness to everyone as one of the key concepts for a Wiki software [9], we think that especially in enterprise scenarios it might be useful to have access control at read and write level. Therefore, Erfurt allows fine-grained access control for both – groups of users and individuals. Access control rules can be defined in OntoWiki itself by

modifying the system configuration model. It provides a class for models as well as actions, whose instances are objects of access control statements. Since in OntoWiki, each registered user has his/her URI which can be used as subject of a rule statement, an example access control statement, which grants the admin user the right to register new users, would be as follows:

```
<http://localhost/OntoWiki/Config/Admin>
  <http://ns.ontowiki.net/SysOnt/grantAccess>
    <http://ns.ontowiki.net/SysOnt/registerNewUser>.
```

The above statement is, of course, unnecessary in OntoWiki since the admin user is granted any action by default.

2.2.2 Caching Component

Erfurt supports several caching mechanisms based on `Zend_Cache`. Almost any entity from objects to function return-values can be stored for faster retrieval. `Zend_Cache` allows the usage of several cache backends of which database and file backends are the most important. Developers are encouraged to make use of Erfurt's caching facilities as it will greatly improve user experience.

2.2.3 Event Dispatcher

The Erfurt event dispatcher builds the foundation of Erfurt's and of OntoWiki's plug-in system. Since the dispatcher implements the Observer pattern, extensions can register code for execution when certain events occur. The registrants can be either classes or objects. In both cases, a method with the same name as the event, must exist and will be executed once the event is triggered. Events can be triggered by using the event dispatchers `trigger` method. For a detailed description of OntoWiki's plug-in architecture, see section 2.3.

2.2.4 RDF/RDFS/OWL API

These classes provide a resource-, property- or model-centric view on the triples in an RDF store, taking into account additional inbuilt semantics that are provided by different layers of the Semantic Web stack.

Once the heart of the Erfurt API (then named pOWL [1]), they provided an easy-to-use interface which unfortunately led to extensibility and scalability problems. Thus, the current state of Erfurt contains only the most important classes with a reduced method set.

Functionality currently provided by these classes includes the following:

- adding/removing statements,
- updating models with a statement diff,
- namespace handling,
- URI handling,

- transitive closure calculation and
- `owl:imports` handling.

For performance reasons, complex retrieval tasks are not covered by the API and should be done through SPARQL [6] in combination with domain-specific MVC models instead (see section 2.3 on how this is done in OntoWiki core components).

2.2.5 Store Component

Triple storage and retrieval are provided by Erfurt's storage component. Erfurt allows for easy integration of RDF stores via adapters that mediate between the store's communication protocols and Erfurt's PHP API. The API provided by adapters is not directly exposed to framework clients. Instead, a lightweight intermediate layer (`Erfurt_Store`) is used to assure that access control rules are adhered to and that versioning information is kept along with changes to the RDF store. This architectural decision has two implications:

- Versioning and ACL enforcement is completely transparent to store adapters.
- The store architecture is open to extension, for instance different import and export formats can thus be supported by the API.

Erfurt comes with store adapters for MySQL and OpenLink Virtuoso [7]. The list of supported stores will be expanded in future versions of the framework. As a matter of fact, work is currently being done on Redland and Oracle adapters.

2.2.6 Versioning Component

As its name implies, this component is responsible for keeping versioning information on an RDF store. Versioning is handled on statement level, i. e. actions that are recorded are *statement-added*, *statement-removed* and *statement-changed*. The usual entry point is a resource URI which yields all changes that have been made to statements about that specific resource. In addition, Erfurt's versioning component provides other entry points such as user URI or model URI, where all changes are returned that have been made by a specific user or have been made to statements in a specific model, respectively.

2.3 Application Layer

OntoWiki as a Web application is based on the Zend Framework which lays out the basic architecture and is primarily responsible for request handling. In the following paragraphs we cover custom OntoWiki classes and aspects of

the Zend Framework that need to be considered when developing Semantic Web applications with the OntoWiki Application Framework.

2.3.1 OntoWiki Request Lifecycle

The single entry point to the application is the `index.php` file which sets up the basic environment and starts the `OntoWiki_Application` singleton. The latter initializes the OntoWiki application itself and serves as a global registry for objects and simple values. Thereafter, the Zend Framework takes over control and dispatches the request to an appropriate controller with an action that handles the request. The content is then rendered into templates, as described in the Templates paragraph of section 2.4.

2.3.2 OntoWiki MVC Models

One of OntoWiki's most outstanding features is that it automatically displays human-readable representations of resources instead of URI strings. The naming or title properties it uses are configurable both on a global level and per model. SPARQL queries that test all naming properties can be quite complex. OntoWiki therefore provides a model base class that builds SPARQL query fragments and fetches the correct naming property value from an Erfurt store result set.

Fig. 3 Screenshot of OntoWiki with OntoWiki Application Framework components: 1) menu, 2) toolbar, 3) navigation, 4) module window and 5) message

2.3.3 Menus

Menus in OntoWiki (see 1 in figure 3) consist of instances of `OntoWiki_Menu`. Entries are set by using the `setEntry` instance method that takes two arguments: the name of the menu entry and the content, which can be a string, another instance of `OntoWiki_Menu` or a menu separator stated by `OntoWiki_Menu::SEPARATOR`. An optional third parameter denotes whether entries of the same name should be replaced or not.

2.3.4 Toolbar

To ensure a consistent user interface throughout all views, the toolbar is centrally managed. In each request there exists an instance of `OntoWiki_Toolbar` to which buttons and separators can be appended or prepended. An example toolbar is depicted under 2 in figure 3. Table 1 shows default buttons that are available.

Table 1 Toolbar buttons available in OntoWiki.

Constant	Name	CSS class	Function
`CANCEL`	Cancel		Cancel an operation
`SAVE`	Save		Save current changes
`EDIT`	Edit	edit-enable	Enter editing mode
`ADD`	Add		Add a new entity
`EDITADD`			Add a new entity by editing another
`DELETE`	Delete		Delete the current selection
`SUBMIT`	Submit	submit	Save changes
`RESET`	Reset	reset	Reset changes

The name or CSS class of default buttons can be overwritten by providing the `appendButton` or `prependButton` method with a configuration array as the second parameter. If the configuration array is the only parameter, a custom button will be generated (in that case an image URL should be provided, as well).

2.3.5 Navigation

Without any customization, OntoWiki's main navigation is displayed as a tab bar in the upper part of the main window (see 3 in figure 3). Components can register one or more actions with the navigation. A component's default action is registered automatically by the component manager. Disabling the navigation is possible by calling `OntoWiki_Navigation::disableNavigation()`.

2.3.6 Extension Architecture

The OntoWiki Application Framework differentiates between three kinds of extensions:

Plug-ins are the most basic, yet most flexible types of extensions. They consist of arbitrary code that is executed on certain events. Plug-ins need to be registered for events in the `plugin.ini` config file that has to be placed in the same folder as the plug-in class.

Modules display little windows that provide additional user interface elements with which the user can affect the main window's content. Since some modules are highly dynamic extensions, they can be configured both statically and dynamically. Static configuration works in the same way as with other extensions; a module.ini file is placed in the module's root directory. In addition, a module class needs to extend `OntoWiki_Module` and can redefine several of its methods in order to allow for dynamic customization. If present, return values will overwrite static configuration settings in the `module.ini` file.

Components are pluggable MVC controllers to which requests are dispatched. Usually but not necessarily, components provide the main window's content and, in that case, can register with the navigation to be accessible by the user. In other cases components can function as controllers that serve asynchronous requests. Components are statically configured by a `component.ini` file within the component's folder.

2.3.7 Localization

`Zend_Translate` along with CSV files are used to translate user interface strings. Extensions can provide their own translation files. If done so, the folder containing the translations must be set in the configuration file. Translatable strings are printed using the the `_` member function of `OntoWiki_View`. Alternatively, the translate object that can be requested from `OntoWiki_Application` provides a `translate` method.

2.3.8 URLs and URI Parameters

By convention, the URL parameter that identifies a resource is named *r*. If this parameter contains only a URI's local part or a cURI[2], OntoWiki automatically expands it into a full URI by using namespace prefixes from imported knowledge base files.

For constructing URLs, usage of `OntoWiki_Url` is recommended. This class initializes itself with the currently active URL but all parameters including controller and action can be replaced. Apart from name and value, the `setParam` method accepts a third optional parameter that, if set to `true`, enables automatic URI compacting by replacing namespaces with their prefixes or no prefix at all for the currently active model. This behaviour allows for user-friendly short URLs with almost no extra effort for the developer.

[2] `http://www.w3.org/TR/rdfa-syntax/#s_curieprocessing`

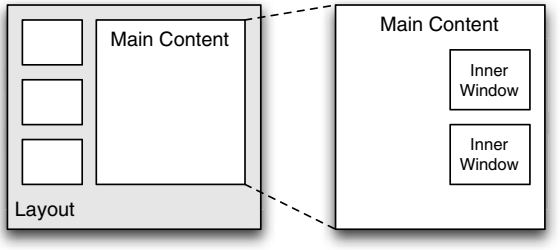

Fig. 4 OntoWiki template hierarchy. The main content is produced by the controller action. Layout content is applied automatically by the template system.

2.3.9 Messages

User notifications are represented by `OntoWiki_Message` which is instantiated by passing a message text and a type constant. Recognized types are `SUCCESS`, `INFO`, `WARNING` and `ERROR`. The main `OntoWiki_Application` object keeps a message stack that is automatically displayed in the upper part of the page. In figure 3, a message is depicted under 5. Elements can be added to this stack via `OntoWiki_Message` member functions `appendMessage` or `prependMessage`.

2.4 User Interface Layer

2.4.1 Templates

Content is rendered in OntoWiki through templates, as suggested by the `Zend_View` template system. The controller action serving the request renders its output in a template. In doing so, it has control over inner windows within the main content and can explicitly include modules (see figure 4). In order to build a complete page the main content is inserted into a layout template that defines the position of main content and side windows.

2.4.2 User Interface API and CSS Framework

While much of the user interface dynamism found in OntoWiki is made available via a JavaScript API, its look and feel results from a sophisticated CSS framework that makes it almost unnecessary to provide custom style sheets. Since the API itself relies heavily on jQuery[3], large parts of it are implemented as jQuery plug-ins. This design allows, not only style, but also behaviour be used automatically on HTML elements that carry the respective CSS classes.

[3] `http://jquery.com/`

2.4.3 RDFa Widgets

By exposing RDFa, structured data is available in rendered HTML code. A set of JavaScript-based widgets that make use of statements extracted from RDFa provides editing functionality to be directly invoked from the client side (i. e. inside the user's web browser). Since complete statements are available to those widgets and they can even fetch additional metadata, e. g. `rdf:range` or `rdf:datatype` constraints, it is possible to provide the user with well-suited edit forms. Changed statements are then sent back asynchronously, so no HTML page refresh is required after performing an edit action.

3 Use Cases

In this section we introduce exemplarily three projects that make use of the OntoWiki Application Framework to different extents.

3.1 SoftWiki – Requirements Engineering the Wiki Way

SoftWiki is a specialized Wiki application for end-user-centered requirements engineering. The aim of the SoftWiki application is to support the collaboration of all stakeholders in software development processes in particular with respect to software requirements. Potentially very large and spatially distributed user groups shall be enabled to collect, semantically enrich, classify and aggregate software requirements. Thus, SoftWiki is a prime example for such knowledge-rich applications which can be developed with the OntoWiki Application Framework.

Requirements for the SoftWiki application can be outlined as follows:

- The main entity in SoftWiki is a requirement with its attributes and relations between requirements.
- Users should be enabled to create and manage requirements as well as relations between them in an easy way with respect to two different management schemes, namely topic hierarchies and tag clouds.
- Users should be supported in their collaboration, for instance, if they want to discuss and vote on particular requirements.

Based on these requirements, SoftWiki was developed as a plug-in for OntoWiki instead of developing a new Semantic Web application from scratch. SoftWiki uses the majority of the backend functionality including versioning, access control and authentication. In addition to the OntoWiki-provided backend, SoftWiki implements a dynamic user interface, built upon asynchronously loaded GUI components from the OntoWiki base system.

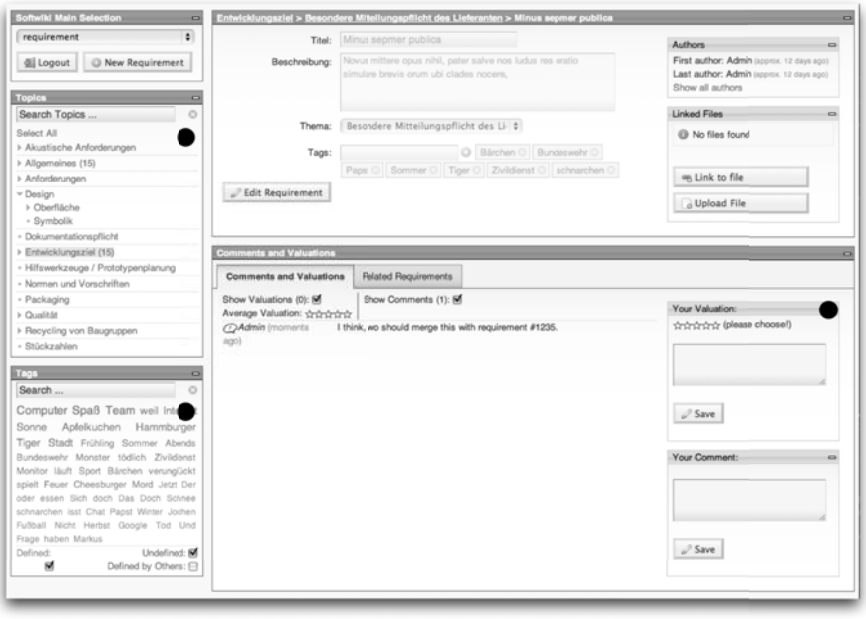

Fig. 5 SoftWiki detail view for a specific requirement. All marked windows are reused GUI components.

The list of GUI components includes

- a tag cloud which is based on the tag ontology from Ayers et al.[4],
- a generic hierarchy browser to visualize container hierarchy (this component is used for a class tree but SoftWiki uses it for a hierarchy of SKOS concepts which are used to manage requirements) and
- a generic vote and discussion component which enables users to comment on a specific resource (in SoftWiki these resources are limited to requirements.).

Fig. 5 shows a screenshot of a detailed view for a specific requirement in SoftWiki, where components reused from generic OntoWiki application are marked with a black dot. These components are integrated by Ajax calls from the SoftWiki application. Every component is a specific action and can be used from inside or outside OntoWiki.

Reusing and integrating OntoWiki core functionality was very important in the SoftWiki development. However, the main development effort was done on the SoftWiki plug-in controller which implements specialized views for listing and editing requirements. This controller uses the Erfurt API to store and query statements on a project's requirements and generates the custom views using the CSS framework. Both new and reused components are connected

[4] `http://www.holygoat.co.uk/owl/redwood/0.1/tags/`

by JavaScript functionality which handles all user input and requests the specific GUI elements.

3.2 Caucasian Spiders Database

The Caucasian Spiders database [5] is a faunistic knowledge base on the spiders of the Caucasus. It consists of several components:

- a biological taxonomy of spiders,
- concepts for records, locations and publications,
- instance data about individual spiders with localities and
- geographical information.

The knowledge base has a size of about 240k triples. It is browsable in OntoWiki with only minor tweaks to internal inference algorithms[6]. The project uses OntoWiki as both a knowledge-base editor for data entry and a browser for displaying instance data. OntoWiki supports these use cases with generic user interface components like class tree, facet-based browsing and Map component just to name a few. Significant customizations of the user interface were not required.

The project might, however, benefit from a custom hierarchy tree that can be based on arbitrary properties. This would allow tree-based browsing of the biological taxonomy instead of the `rdfs:subClassOf` hierarchy, which is more natural to biologists. Such a component has been developed for another project (see section 3.1) and will be integrated into OntoWiki in one of the upcoming versions.

3.3 Professor Catalogue of the University of Leipzig

In the course of the 600$^{\text{th}}$ anniversary of the University of Leipzig a database of Leipzig professors from the nineteenth and twentieth century[7] has been built in the department of history.

The database has been realized using Semantic Web technologies allowing it to be queried in various ways. One could for instance try to find the names of professors who were taught by Nobel Prize winners. However useful that query might be, it shows the flexibility that is gained by building upon RDF and related technologies.

Since OntoWiki can work with any RDF knowledge base, it was a natural choice as a generic data wiki for collaboratively building the database and

[5] `http://caucasus-spiders.info/`

[6] E. g. disabling inference that resource A is a class if there exists at least one resource a_i that has A as its `rdf:type` (called *implicit class* in OntoWiki terminology).

[7] `http://www.uni-leipzig.de/unigeschichte/professorenkatalog/`

entering instance data. The result was a knowledge base with about 60 schema elements and 800 entries.

4 Conclusion

In this paper we presented the OntoWiki Application Framework which can be used a basis for developing Semantic Web applications in different environments. We described the usage of the framework and gave example use-cases that were implemented with the OntoWiki Application Framework.

There is, of course, room for future improvements and refinements of the OntoWiki Application Framework. One often-requested feature is reasoning support which is currently integrated as a component. Work is also done on scalability problems that occurred when using very large datasets or complex queries. Scalability becomes paramount with statement-based access control, which heavily decelerated the use of the OntoWiki Application Framework. Further improvements could also include integration with different Semantic Web endpoints like DBpedia[8], Sindice[9] or Linked Data providers.

References

1. Auer, S.: Powl – A Web Based Platform for Collaborative Semantic Web Development. In: Proceedings of the 1st Workshop on Scripting for the Semantic Web at the ESWC, CEUR Workshop Proceedings, Heraklion, Greece (May 30, 2005)
2. Auer, S., Dietzold, S., Riechert, T.: OntoWiki - A Tool for Social, Semantic Collaboration. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 736–749. Springer, Heidelberg (2006)
3. Aumüller, D.: Semantic Authoring and Retrieval within a Wiki (WikSAR). In: Demo Session at the Second European Semantic Web Conference (ESWC 2005) (May 2005), `http://wiksar.sf.net`
4. Aumüller, D.: SHAWN: Structure Helps a Wiki Nvigate. In: Proceedings of the BTW-Workshop "WebDB Meets IR" (2005)
5. Brickley, D., Guha, R.V.: RDF Vocabulary Description Language 1.0: RDF Schema. W3C recommendation, W3C (February 2004), `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`
6. Clark, K.G., Feigenbaum, L., Torres, E.: SPARQL Protocol for RDF. W3c recommendation, W3C (January 2008), `http://www.w3.org/TR/rdf-sparql-protocol/`
7. Erling, O., Mikhailov, I.: RDF Support in the Virtuoso DBMS. In: Auer, S., Bizer, C., Müller, C., Zhdanova, A.V. (eds.) The Social Semantic Web 2007, Proceedings of the 1st Conference on Social Semantic Web (CSSW), Leipzig, Germany, September 26-28, 2007. LNI, vol. 113, pp. 59–68 (2007)

[8] `http://dbpedia.org/`
[9] `http://sindice.com/`

8. Krötzsch, M., Vrandecic, D., Völkel, M.: Wikipedia and the Semantic Web - The Missing Links. In: Voss, J., Lih, A. (eds.) Proceedings of Wikimania 2005, Frankfurt, Germany (2005)
9. Leuf, B., Cunningham, W.: The Wiki Way: Collaboration and Sharing on the Internet. Addison-Wesley Professional, Reading (2001)
10. Oren, E.: SemperWiki: A Semantic Personal Wiki. In: Decker, S., Park, J., Quan, D., Sauermann, L. (eds.) Proc. of Semantic Desktop Workshop at the ISWC, Galway, Ireland, November 6, vol. 175 (2005)
11. Patel-Schneider, P.F., Hayes, P., Horrocks, I.: OWL Web Ontology Language - Semantics and Abstract Syntax. W3c:rec, W3C (February 10, 2004), http://www.w3.org/TR/owl-semantics/
12. Souzis, A.: Building a Semantic Wiki. IEEE Intelligent Systems 20(5), 87–91 (2005)