Using Prolog as the fundament for applications on the semantic web

Jan Wielemaker[1], Michiel Hildebrand[2], and Jacco van Ossenbruggen[2]

[1] Human Computer Studies,
University of Amsterdam,
The Netherlands,
`wielemak@science.uva.nl`
[2] CWI, Amsterdam, The Netherlands
`firstname.lastname@cwi.nl`

Abstract. This article describes the experiences developing a Semantic Web application entirely in Prolog. The application, a demonstrator that provides access to multiple art collections and linking these using cultural heritage vocabularies, has won the first price in the ISWC-06 contest on Semantic Web end-user applications. In this document we concentrate on the Prolog-based architecture, describing experiences and vital aspects of the design.

1 Introduction

Prolog has some attractive properties for Web and Semantic Web applications. Safety and automatic memory management as well as incremental compilation are essential to web-programming, (natural) language processing, simple reasoning, constraint programming and a natural representation of the Semantic Web triple model are features that contribute to the usability of Prolog for web-programming. Disadvantages are lack of ready-to-use resources for dealing with Web protocols and documents as well as the availability of skilled Prolog programmers in this field.

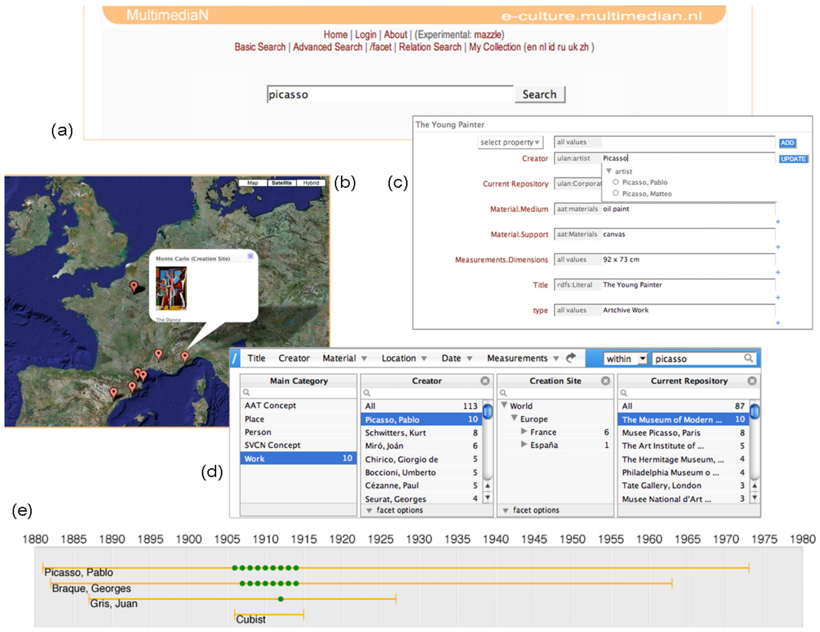
Within the E-culture research program[3] we were in the luxury position to have access to a good Prolog based starting point [13] and contributing researchers with Prolog affinity and experience. A small demonstrator was extended into a award-winning application [9] by a team of five programmers spread over three institutes.

SWI-Prolog's features for Web-programming are described in detail in [14]. This document describes practical experience using the framework in a larger project. We concentrate on design aspects to facilitate re-usability and independence between the various components of the software.

This document is organised as follows. First we introduce the E-culture demonstrator, briefly describing its functionality and software architecture. Then we describe the libraries enabling the design, concentrating on those that have

[3] http://e-culture.multimedian.nl/

been added during the project to enhance modularity and reuse. In Sect. 7 we give some practical tips for deployment of a large Prolog-based server on the Web. We conclude with problems, lessons learned, related work and plans.



The Young Painter

select property	all values	
Creator	ulan:artist	Picasso
		artist: Picasso, Pablo; Picasso, Matteo
Current Repository	ulan:Corporat	
Material.Medium	aat:materials	oil paint
Material.Support	aat:Materials	canvas
Measurements.Dimensions	all values	92 x 73 cm
Title	rdfs:Literal	The Young Painter
type	all values	Artchive Work

Main Category		Creator		Creation Site		Current Repository	
AAT Concept		All	113	World		All	87
Place		Picasso, Pablo	10	Europe		The Museum of Modern ...	10
Person		Schwitters, Kurt	8	France	6	Musee Picasso, Paris	8
SVCN Concept		Miró, Joán	6	España	1	The Art Institute of ...	5
Work	10	Chirico, Giorgio de	5			The Hermitage Museum, ...	4
		Boccioni, Umberto	5			Philadelphia Museum o ...	4
		Cézanne, Paul	5			Tate Gallery, London	3
		Seurat, Georges	4			Musee National d'Art ...	3

Fig. 1. Screendumps of the E-culture web-application. (a) simple text-based search interface, (b) geographical map visualisation, (c) resource annotation interface, (d) faceted navigation, (e) timeline visualisation.

2 Introducing the E-culture demonstrator

The aim of the E-culture demonstrator is to provide a common gateway to multiple museum collections and cultural heritage documents. Museums use different database models based on different vocabularies to represent their collection. Merging this into a single datamodel is complicated, labour intensive and leads to loss of information due to inadequacy of the common model as well as errors in the transformation process. We converted [11] both vocabularies and meta-data into RDF/OWL preserving the original structure. Only where literal strings were based on a known vocabulary, we restored the mapping to the vocabulary. After this lossless transformation process, the meta-data schema is mapped to the standard VRA schema[4] using RDFS subPropertyOf relations and cross-relations between vocabularies were restored or created. Our current RDF graph contains

[4] http://www.vraweb.org/

8.6 million triples describing over 100,000 art-objects from 4 different sources and 7 vocabularies.

The RDF graph is stored in memory [15] and made accessible from Prolog by means of the predicate **rdf**(*Subject, Predicate, Object*). The web-server of the demonstrator is realised by the SWI-Prolog multi-threaded HTTP server library[5]. In this web-server, a predicate serves one (typical) or more HTTP *locations*. The handler receives the parsed HTTP request as a Prolog data structure and writes a CGI document to the current output stream. This approach is comparable to Tomcat, where a class is defined to handle an HTTP location by writing a CGI document onto a stream.

Although any Prolog predicate that produces a valid CGI document can be used, the library html_write provides a DCG-based framework to write HTML and XHTML documents from the same specification. This library ensures proper nesting of tags and escapes for special characters. The library is described in [14].

The system contains two types of reusable modules. *Reasoning* modules on top of RDF provide RDFS (Schema) and limited OWL inferencing as well as more domain specific reasoning such as various graph-search and graph-abstraction predicates. *Presentation* modules define HTML DCG rules producing reusable components of the interface, such as presenting an image thumbnail or a widget that allows for selecting a term from a vocabulary using AJAX-based [7] interactivity.

Based on these reusable modules, different interfaces to the data are realised by different HTTP locations. Currently we have four interfaces. *Basic search* performs a graph-search from literals that match at least one word with the query to target objects (art-works) and clusters the results based on the RDF properties and class of the resource in the path from literal to target object. *Relation search* describes relations between arbitrary objects. */facet* provides a traditional facetted browser [5] and *Mazzle* merges basic search with facetted browsing while providing multiple points of focus, currently art-works, artists and geographical locations. Figure 1 shows some screenshots of the application, while the architecture is summarised in Fig. 2

3 Used technologies

It is an explicit aim of the project to use Open Standards where possible. This implies RDF/OWL for representing meta-data and vocabularies, a web-server (HTTP) using W3C standards for access. Machine-access is provided by means of the SPARQL[6] or SeRQL [2] RDF query language while human access uses browser standards.

Standard HTML has two limitations: lack of graphics and lack of interactivity. Initially these were resolved using SVG for non-interactive graphics and Java applets for interactivity. Eventually both have been replaced by HTML+CSS using AJAX for interactivity. HTML+CSS has limited graphical capability, but

[5] http://www.swi-prolog.org/packages/http.html

[6] http://www.w3.org/TR/rdf-sparql-query/

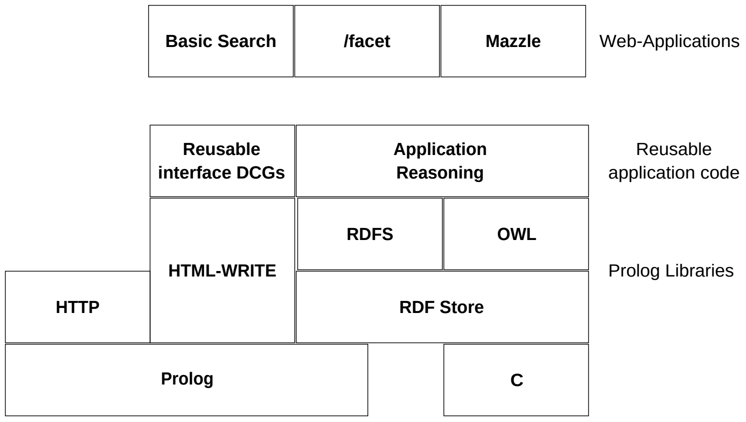


Fig. 2. Architectural components of the Prolog-based web-application

sufficient for our needs and they are much better supported by todays browsers. HTML+CSS with AJAX can deal with the interactivity we require, such as suggesting relevant vocabulary terms on each key-stroke in a text entry field. (Re)usable AJAX client scripts are widely available. Providing the required HTTP service that connects them to the data is easy.

4 Core Web libraries

In this section we describe the core libraries that enable the design. Some libraries have been described in other publications, in which case we keep the description concise.

4.1 The RDF library

The RDF library [15] is the core of SWI-Prolog's Semantic Web infrastructure. The key predicate is **rdf**(*Subject, Predicate, Object*), providing very natural access to the triple store. The predicate itself is defined in C. Because we know all clauses are ground unit clauses, resources are atoms and predicates are organised in a hierarchy using rdfs:subPropertyOf we can design an optimal representation minimising space and optimising access times. During the E-culture project we realised several enhancements to the core RDF library that are not described in previous publications and which we describe below.

Multi-threading support is enhanced by introducing *read-write locks* and *transactions*. During normal operation, multiple readers are allowed to work concurrently. Transactions are realised using **rdf_transaction**(*:Goal, +Context*). If a transaction is started, the thread waits until other transactions have finished. It then executes *Goal*, adding all write operations to an agenda. During this phase the database is not actually modified and other readers are allowed to proceed.

If *Goal* succeeds, the thread waits until all readers have completed and updates the database. If *Goal* fails or throws an exception the agenda is discarded and the failure or error is returned to the caller of **rdf_transaction/2**. Note that this behaviour is different from multi-threaded Prolog assert/retract.

– In multi-threaded (SWI-)Prolog, accessing a dynamic predicate for read or write demands synchronisation only for a short time. In particular, readers or writers with a choice-point allow other threads to operate on the same predicate. At the same time logical update semantics are realised. This is achieved using time-stamps and keeping erased clauses around until the predicate is sufficiently 'dirty' and there are no readers or writers.
– Multiple related modifications are bundled in a transaction. This is often desirable as many high-level (RDFS/OWL) changes involve multiple triples. Using transactions guarantees a consistent view of the database and avoids partial modifications.

RDF literals have been promoted to first class citizens in the database. Typed literals are supported using arbitrary Prolog terms as RDF object. Numbers (float, integer) are store in their native C representation, Unicode strings are stores as Prolog atom-handles and other Prolog terms are stored using the recorded-database access provided by SWI-Prolog through the foreign interface by means of PL_record(), PL_recorded() and PL_erase(). All literals are kept in an AVL-tree, where

$$\text{numeric-literals} < \text{string-literals} < \text{term-literals}$$

Numeric literals are sorted by value. String literals are sorted alphabetically, case insensitive and after removing diacritics. String literals that are equal after discarding case and diacritics are sorted on Unicode value. Other Prolog terms are sorted on Prolog standard order of terms. Sorted string literals are used for fast prefix search which is important for suggestions and disambiguation as-you-type with AJAX style interaction.

The literal search facilities are completed by means of *monitors*. Using **rdf_monitor**(*:Goal, +Events*) we register a predicate to be called at one or more given events. Monitors that trigger on literal creation and destruction are used to maintain a word-index for the literals as well as an index from stem to word and metaphone [8] key to word. Monitors are also used to achieve *persistency*. For persistency, each named graph is backed up by a file containing the state after initial load or last check-point and a file describing actions on the named graph, the *journal*.

4.2 Library HTML write

The HTML writer library uses Prolog DCGs in 'write' mode to translate a ground Herbrand term into a list of HTML tokens. The tokens are written to a Prolog stream using **print_html/2** to produce valid HTML. The Herbrand term can have embedded \term sequences, which causes nested invocation of the DCG referenced by *term*. We introduce the HTML library using an example from the OpenID[7] library. Note the in-line invocation of the rules **openid_title//0** and **hidden//2**. Details have been described in [14].

4.3 Session management

The core HTTP library defines a hook to expand the HTTP request. This hook is exploited by the session management library to realise cookie-based session management. The session library also defines **http_session_assert**(*+Term*), **http_session_retract**(*?Term*) and common assert/retract variations to realise storage of session specific data which can be queried using **http_session_data**(*?Term*).

Session-data is automatically retracted after session timeout. Start and end of a session is broadcasted (see Sect. 4.6), to enable additional processing by individual modules.

4.4 The HTTP dispatching code

The core HTTP library, described in [12], handles all requests through a single predicate. Normally this predicate is defined 'multifile' to split the source of the server over multiple files. This approach proved inadequate for a larger server with multiple developers for the following reasons:

- There is no way to distinguish between non-existence of an HTTP location and failure of the predicate due to a programming error. This is an omission in itself, but with a larger project and multiple developers it becomes more serious.
- There is no easy way to tell where the specific clause is that handles an HTTP location.
- As the order of clauses in a multi-file predicate that come from different files is ill defined, it is not easy to reliably redefine the service behind a given HTTP location. Redefinition is desirable for re-use as well as for experiments during development.

To overcome these limitations we introduced a new library http_dispatch that defines the directive **http_handler**(*Location, Predicate, Options*). The directive is handled by **term_expansion/2** to manage a multi-file predicate. This predicate in turn is used to build a Prolog term stored in a global variable that provides fast search for locations. Modifications to the multi-file predicate cause re-computation of the Prolog term on the next HTTP request. *Options* can be used to specify access rights as well as a priority that allows for overruling existing definitions. Typically, each location is handled by a dedicated predicate. Based on the handler definitions, we can easily distinguish failure from non-existence as well as find, edit and debug the predicate implementing an HTTP location.

[7] http://openid.net/

```
%%      openid_login_form(+ReturnTo, +Options)// is det.
%
%       Create the OpenID form.  This is exported as a separate DCG,
%       allowing applications to redefine /openid/login and reuse this
%       part of the page.

openid_login_form(ReturnTo, Options) -->
        { option(action(Action), Options, verify)
        },
        html(div(class('openid-login'),
                 [ \openid_title,
                   form([ name(login),
                          action(Action),
                          method('GET')
                        ],
                        [ \hidden('openid.return_to', ReturnTo),
                          div([ input([ class('openid-input'),
                                        name(openid_identifier),
                                        size(30)
                                      ]),
                                input([ type(submit),
                                        value('Verify!')
                                      ])
                              ])
                        ])
                 ])).

hidden(Name, Value) -->
        html(input([type(hidden), name(Name), value(Value)])).

openid_title -->
        html(div(class('openid-title'),
                 [ a(href('http://openid.net/'),
                     img([ src('file?name=openid_logo'), alt('OpenID') ])),
                   span('Login')
                 ])).
```

Fig. 3. HTML DCG presenting OpenID login page.

4.5 Setting management

Managing settings of the application is not typical for Web-servers, but the size of this project raised the need for central management of settings. Initial management was based on a file called `parms.pl` that defined **setting/1**, containing clauses like `setting(thumbnail_size(100,100)).` As the project grew we realised it was difficult for different developers to maintain different values for the settings without corrupting the central file under CVS revision control and this central file, holding information for many modules, seriously harmed modularity of the application and we introduced two new libraries. One for declaring, storing and asking setting values and one for querying and editing settings through the web-interface.

Declaration of a setting is achieved using the directive **setting**(*:Name, +Type, +Default, +Comment*). Settings are local to a module. Settings from other modules can be defined and requested using the standard ⟨*module*⟩:⟨*name*⟩ syntax instead of using a plain atom for the name. The interface includes **setting**(*:Name, -Value*), **set_setting**(*:Name, +Value*), **save_settings**(*+File*) and **load_settings**(*+File*). When settings are saved to file, only those that have a value not equal to their default are saved. Setting default declarations provide syntactical constructs to ask for environment variables and the value of other settings. Numerical settings can use arithmetic expressions and textual settings can use the + operator for concatenation.

Whenever a setting is modified the broadcast library described in Sect. 4.6 is informed. This allows modules to react on changes to settings immediately, also for settings that are only read during initialisation of the service.

The result provides distributed declaration of settings that no longer harms modularity. Proper typing and comments simplify reuse of settings over the application and an extensible web-interface manages the application settings.

4.6 The broadcasting service

The Prolog library broadcast was initially developed for the graphical subsystem XPCE to deal with application *events* and distributed information gathering. Its function can be compared to *hooks*, but central administration makes it easier to inspect broadcasted events and check who is listening to what events. The *hooks* are called *listeners* and are owned, where the owner is represented by an arbitrary ground term. When omitted, this is the module-name making the registration. We illustrate the functionality using a simple session. The atom `me` represents the owner. Details and source can be requested from the SWI-Prolog documentation server[8].

```
?- listen(me, hello(X), format('Hello ~w~n', [X])).
?- broadcast(hello(world)).
Hello world
?- unlisten(me).
```

[8] http://gollem.science.uva.nl/SWI-Prolog/pldoc/

```
?- broadcast(hello(world)).
```

Where **broadcast/1** runs a failure driven loop over all listeners, **broadcast_request/1** is non-deterministic and succeeds on any listener that succeeds.

The web-libraries use the broadcasting service for session and setting management.

5 SWI-Prolog enabling features

Discussed with more detail in [14], we will briefly summarise the requirements on Prolog that enable its use as Semantic Web application platform.

- Scalability requires for a multi-threaded Prolog engine. Next to exploiting multi-CPU hardware efficiently, it also avoids slow queries from making the server inaccessible.
- Using unlimited-length Unicode atoms and atom garbage collection allows for uniform and simple representation of arbitrary text for web-applications.
- The system requires support for incremental compilation, so code can be modified and the server can be updated and tested without restart or loosing sessions. SWI-Prolog offers **make/0**, which reloads all modified source-files comfortably. Currently, temporal inconsistencies in the running program during reload can cause errors in services that run concurrently. We plan to enhance this using read-write locks that synchronise program update with the HTTP worker threads. Lacking these locks is generally no problem for local development or non-critical public services.

6 The role of RDF query languages

Most Semantic Web applications are modelled after relational database applications, where the application logic accesses the database through SQL. We see a number of Semantic Web equivalents to SQL, such as SeRQL [2] and the W3C recommendation SPARQL[9]. Both allow for specifying a graph expression consisting of a number of obligatory and optional edges and nodes extended with conditions on literal values, SeRQL matches the graph expression on the transitive closure using the semantics of RDFS. The SPARQL standard does not specify whether or not entailment reasoning is performed by the database engine. We implemented SeRQL and SPARQL support on top of the SWI-Prolog Semantic Web library using the HTTP infrastructure defined in this document to make the server accessible for both humans and machines.

The E-culture application, however, does not use SeRQL or SPARQL. Instead, queries by the application logic are expressed as Prolog goals on the raw RDF database and/or RDFS/OWL reasoning modules. At places where the order of executing conjunctions is critical and cannot easily be predicted by the

[9] http://www.w3.org/TR/rdf-sparql-query/

application programmer, we use the query optimiser we developed for the SeRQL server [13], which rewrites a Prolog goal involving multiple calls to **rdf/3** and tests for optimal performance. Semantic Web query languages are not used in the application logic because

- Prolog itself already provides a completely transparent and easy to use API. As the application programmer uses Prolog anyway, Prolog syntax is a natural choice. Note that a classical approach for accessing relational databases from Prolog is by translating Prolog goals into SQL statements [6]. We see only a role using a query language for access by external applications and if query expressions are used to specialise the application for a specific environment and this specialisation is done outside the application itself.
- SPARQL lacks expressiveness to construct complex path expressions. For example, SPARQL does not support regular expressions in query paths, therefore, there exists no query that gets the root of a resource given a transitive property. Note that PSPARQL [3] is being developed to support exactly this.
- For our purpose we often need specific RDFS/OWL reasoning support in different parts of the demonstrator. Partial reasoning that fulfil our requirements is easily implemented and performs well. We believe efficient complete DL-reasoning over our large and generally inconsistent RDF store is not realistic.
- We have a need for dedicated graph search in which we guarantee quick termination by limiting the 'semantic distance' based on weighted relations.
- Current Semantic Web query languages support for literal search is generally limited to regular expression search and numerical conditions. We have need for searching for keywords that can appear inside literals, possibly considering stemming. We also require fast prefix search for the suggestion interface, both on full literals and on keywords. Many applications solve this problem by populating a general text indexing engine such as Lucene[10] with the literals.
  Indexing integrated with the RDF store, however, greatly reduces memory requirements and access times, while simplifying maintenance when the RDF store is modified.

7 Deployment

Like Apache, Tomcat, etc., the Prolog based HTTP server can talk directly to a standard compliant browser. This setup, running the Prolog server interactively from a non-privileged port is normally used by the developers.

With some care, public deployment can also use the Prolog server directly. On a typical Unix system this requires the server to be started as root and make the required system calls available from Prolog to drop privileges after opening the server port. Typically this setup asks for a dedicated, possibly virtual, server machine. Due to practical considerations we opted for the option to use a public

[10] http://lucene.apache.org/

Apache server as reverse proxy. It also allows placing the Prolog server inside a firewall and realises a greater level of reliability because ill-formed requests are already blocked by the proxy server. The configuration file below makes the demo available from apache. Apache requires the standard modules `proxy` and `proxy_http` to be enabled The Prolog server listens to port 3020.

```
ProxyPass        /demo/ http://mn9c.mydomain.org:3020/demo/
ProxyPassReverse /demo/ http://mn9c.mydomain.org:3020/demo/
```

The Prolog server is started from a Unix boot script. Maintenance of the E-culture demo such as re-loading modified Prolog source files using **make/0** is realised by means of HTTP commands. The SWI-Prolog documentation server[11] is realised with a similar setup, but the Prolog server runs interactively in a terminal inside a VNC server session using an unprivileged user that is started from a Unix boot script. This setup allows easy monitoring and modifications by contacting the VNC virtual desktop.

8 Metrics

Our current RDF store contains 8.6 million triples while we plan to deal with 150 million triples on a server with 8 CPUs and 32GB main memory within 2 years. The application specific code is about 35,000 lines. The SeRQL/SPARQL infrastructure counts 18,000 lines. Finally, the SWI-Prolog HTTP library is 5,100 lines and the Semantic Web database 7,300 lines Prolog and 11,000 lines of C.

Time to load all 8.6 million triples from RDF/XML and Turtle source is 350 seconds. Time to restore from the file-based persistent database is 40 seconds. Timings are measured on an Intel core duo X6800@2.93Ghz using the 64-bit version of SWI-Prolog 5.6.34 under SuSE Linux 10.2. Initial load and restore are currently not multi-threaded.

Process' data size is 1.8GB (64-bit mode). Resources are represented as atoms. We counted 3,4 million atoms, 0.6 million for the literal index, 2.8 million for resources and literals and only 18,000 for the program.

The 8.6 million triples contain 1.9 million literals. The token and stem indices are built in 90 seconds and require 200MB memory. The token index contains 1.0 million words and numbers. The stem index has 380,000 stems.

We acquired some statistics on public server. During 3 days of operation using 8 worker threads on 2 CPUs it used 12,000 seconds CPU time, an average of 2.5% of the system capacity. Table 1 shows how calls to **rdf/3** are distributed over the possible instantiation patterns.

9 Problems experienced

Our server uses a large amount of not very well established technology. There is not much established technology in the Semantic Web world, making this unavoidable in that part of the application. For serving general web-pages however

[11] http://gollem.science.uva.nl/SWI-Prolog/pldoc/

Indexed			Calls
-	-	-	14,430
+	-	-	833,552
-	+	-	3,600
+	+	-	216,792,146
-	-	+	2,252,522
-	+	+	38,739,699
+	+	+	2,337,826

Table 1. Indexing pattern on rdf(Subject,Predicate,Object) calls after 3 days of operation.

there are many alternatives such as Tomcat servlets, jsp, php, asp, etc. Doing it all in Prolog greatly simplifies and enhances the performance in the interaction between the RDF store and the general web-page generation. It also greatly simplifies deployment. An installed version of SWI-Prolog and the hierarchy of Prolog source files are the only dependencies.

Upgrading a platform that had only be tested on small scale applications developed by one programmer to a large demanding application with multiple developers proved to be a challenge that requested the concurrent development of modules to deal with dispatching, session management and setting management. We also had to establish the best practices to use the infrastructure, notably to reach at proper re-usability of interface components. Affinity with Prolog programming in the whole team was necessary to make this work. We hope the matured Prolog libraries for web-programming with a planned Open Source release of the demonstrator provides a platform for other teams.

There were two main sources of bugs in the platform. One was still incomplete or false processing in both the HTML/HTTP infrastructure and the Semantic Web libraries. The other source of problems was found in the low-level RDF store, notably locking for thread-safety and memory management issues in the C-code.

10 Lessons learned

We started this project based on the SeRQL server running on top of the SWI-Prolog Semantic Web and HTTP libraries [13]. This system was fairly simple and small, handling about 50 HTTP locations that had largely be defined by the OpenRDF [2] project. It was developed by a single programmer. The E-culture project has a larger development team, is aiming at a demanding and stable server platform while the best way to support end-users based on Semantic Web data is explored using multiple prototype web interfaces.

It quickly became apparent that this required infrastructure and best-practice guidelines on how web-applications needed to be written for optimal re-usability and modularity.

– The http_dispatch library greatly enhanced the ability to find and debug code handling an HTTP location.
– The setting management library realises distributed management of application settings.
– Instead of mixing application logic, general HTML primitives and the specific code to handle a set of HTTP locations in a single file we started a libraries with HTML primitives, general primitives based on the Semantic Web libraries and more high-level application logic.

Note that the design as a web application makes it easy to deploy multiple user-interfaces concurrently on the same server from different HTTP locations. Based on a stable low-level RDF and HTML output routines, experimental code and (semi-) production code live together on the same server.

New explorations are not handled using a branch in the revision control system, but using a copy of the code running on another HTTP location. Not using CVS branches simplifies refacturing needed to deal with evolving new infrastructure such as the introduction of the dispatch, setting and session management libraries.

The HTML write library based on DCG with inline calling of other rules using the \-syntax proves to work well. It can generate both traditional HTML and XHTML from the same Prolog source and allows for easy reuse of common components. An open issue is the content of the HTML head, notably required references to CSS and Javascript files. We must consider a syntax where DCG components can specify required CSS and Javascript which is moved to the head in an extra rewriting step.

Initially interactivity and graphics was provided by means of Java applets running SeRQL queries on the server. Modifications required changing and recompiling the applet code and quite commonly restarting the browser. Later we moved the application logic from the applet to the Prolog server, only keeping the interface behaviour in the applet. With stable applets, we can now change the application logic on the server and deploy the changes using a simple **make/0** at the server.

In early development all interaction was handled server-side, which required a new HTTP request and an update of the entire page for each action. A more responsive solution is available with client-side programming in Javascript. Simple interactions for which all data is already available on the client side can be solved completely client side with Dynamic HTML, an example is the thumbnail browser in /facet.

If the interaction requires additional data, the XMLHttpRequest [7] allows this to be requested from to the server asynchronously. The server response, typically in XML or JSON, is then processed on the client side where it updates the HTML through the Document Object Model (DOM). The combination of these technologies, also known as AJAX, allows for rich interaction strategies while reducing the server workload.

Various interface widgets, such as trees and tabbed views, are publicly available in several JavaScript libraries. Furthermore, services for geographical mapping, timeline and calendar visualisations are easily integrated and updated with AJAX technology.

11 Future plans

Scalability will be tested against two axis. By incorporating more collections we plan to scale to 150 million RDF triples. As the system becomes more widely knows and serves a larger set of collections more user-friendly we anticipate higher loads. It is planned to test scalability on an 8 CPU system with 32 GB main memory.

As the connectivity between vocabularies grows, the graph-based algorithms require more selective exploration of the graph and different abstraction mechanisms to provide sufficiently simple abstractions to satisfy the user.

We also foresee that a larger part of reasoning in the system will be specified in standard (Semantic Web) languages. Notably OWL descriptions can be used to specify target objects and rules (SWRL) can be be used to express simple reasoning and mappings that cannot be expressed using subPropertyOf or owl:sameAs. Such expressions can be translated into Prolog programs and optimised before execution.

We plan to rewrite parts of the web-interface and base it on the Yahoo UI library[12]. Replacing our widgets by professional (web-)widgets enhances the look-and-feel and releases the project from browser compatibility issues. Data interchange with the server will be based on JSON[13].

12 Related work

As far as we know, there are no Prolog systems offering comprehensive support for web programming concentrating on the Semantic Web. Many Prolog systems offer some form of support for the HTTP protocol. The most widely known example is the PiLLoW library [4] developed by the Ciao Prolog team and available for at least Ciao, SWI-Prolog, SICSTus Prolog and YAP. In [14] we compare PiLLoW and the SWI-Prolog infrastructure for handling HTML documents. ProWeb [1] is an ALP-Prolog library aimed at embedded HTTP servers for controlling appliances. Its notion of *Request Processing Modules* (RPM) is probably comparable to our http dispatch library. Lack of details on RPM make an actual comparison impossible. WebLS by Amzi! [10] appears specialised for question-answering type of applications.

13 Conclusions

We presented the SWI-Prolog (Semantic) web application platform with the E-culture demo server. The platform combines an RDF in-core database that is

[12] http://developer.yahoo.com/yui/
[13] http://www.json.org/

seamlessly connected to Prolog with an HTTP server infrastructure, The award-winning web-application, developed by five researchers proves the applicability of Prolog for Semantic Web applications. All described infrastructure is available as Open Source under the LGPL license. The source of the application as a whole will be made available later during the project.

Acknowledgements

This research was supported by the MultimediaN project funded through the BSIK programme of the Dutch Government.

References

1. Manfred Bathelt, Ulrich Gall, Bernd Hindel, and Christian Kurzke. Accessing embedded systems via www: the proweb toolset. In *Selected papers from the sixth international conference on World Wide Web*, pages 1065–1073, Essex, UK, 1997. Elsevier Science Publishers Ltd.
2. Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: An architecture for storing and querying rdf and rdf schema. In *Proc. First International Semantic Web Conference ISWC 2002, Sardinia, Italy*, volume 2342 of *LNCS*, pages 54–68. Springer-Verlag, 2002.
3. cois Baget Jérôme Euzenat Faisal Alkhateeb, Jean-Fran˙RDF with regular expressions. Technical Report RR-6191, INRIA Rhône-Alpes, May 22 2007.
4. Daniel Cabeza Gras and Manuel V. Hermenegildo. Distributed WWW programming using (ciao-)prolog and the piLLoW library. *TPLP*, 1(3):251–282, 2001.
5. Michiel Hildebrand, Jacco van Ossenbruggen, and Lynda Hardman. /facet: A Browser for Heterogeneous Semantic Web Repositories. In *The Semantic Web - ISWC 2006*, pages 272–285, November 2006.
6. Matthias Jarke, Jim Clifford, and Yannis Vassiliou. An optimizing prolog front-end to a relational query system. *SIGMOD Rec.*, 14(2):296–306, 1984.
7. Linda Dailey Paulson. Building Rich Web Applications with Ajax. *IEEE Computer*, 38(10):14–17, 2005.
8. Lawrence Philips. The double metaphone search algorithm. *C/C++ Users J.*, 18(6):38–43, 2000.
9. Guus Schreiber, Alia Amin, Mark van Assem, Viktor de Boer, Lynda Hardman, Michiel Hildebrand, Laura Hollink, Zhisheng Huang, Janneke van Kersen, Marco de Niet, Borys Omelayenko, Jacco van Ossenbruggen, Ronny Siebes, Jos Taekema, Jan Wielemaker, and Bob J. Wielinga. Multimedian e-culture demonstrator. In Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors, *International Semantic Web Conference*, volume 4273 of *Lecture Notes in Computer Science*, pages 951–958. Springer, 2006.
10. Arvindra Sehmi and Mary Kroening. Webls: A custom prolog rule engine for providing web-based tech support. Technical report, Amzi! inc.
11. Mark van Assem, Maarten R. Menken, Guus Schreiber, Jan Wielemaker, and Bob J. Wielinga. A method for converting thesauri to rdf/owl. In *International Semantic Web Conference*, pages 17–31, 2004.
12. J. Wielemaker.

13. Jan Wielemaker. An optimised semantic web query language implementation in prolog. In Maurizio Baggrielli and Gopal Gupta, editors, *ICLP 2005*, pages 128–142, Berlin, Germany, October 2005. Springer Verlag. LNCS 3668.
14. Jan Wielemaker, Zhisheng Huang, and Lourens van der Mey. SWI-Prolog and the Web. Paper submitted to tplp, HCS, University of Amsterdam, 2006.
15. Jan Wielemaker, Guus Schreiber, and Bob Wielinga. Prolog-based infrastructure for RDF: performance and scalability. In D. Fensel, K. Sycara, and J. Mylopoulos, editors, *The Semantic Web - Proceedings ISWC'03, Sanibel Island, Florida*, pages 644–658, Berlin, Germany, october 2003. Springer Verlag. LNCS 2870.