# An IMAP plugin for SquirrelRDF

Eynard, Davide; Recker, John; Sayers, Craig
Mobile and Media Systems Laboratory
HP Laboratories Palo Alto
HPL-2007-161
October 5, 2007*

Semantic Web, RDF, IMAP, Email

The Semantic Web aims to make information accessible to both humans and machines, using standard formats for data and making information available in a formal and structured way. Since the advent of RDF (Resource Description Framework) there have been many efforts to extract and convert existing information in this format. In this paper we describe an adapter tool for the IMAP protocol, developed as a plugin of SquirrelRDF1, which allows users to query IMAP mailboxes using SPARQL. The information returned looks like RDF, is always current, and can be reused and integrated inside other applications.

* Internal Accession Date Only

Approved for External Publication

© Copyright 2007 Hewlett-Packard Development Company, L.P.

# An IMAP plugin for SquirrelRDF

D. Eynard, J. Recker, C. Sayers

Hewlett-Packard
Palo Alto, CA, USA
davide.eynard@hp.com

**Abstract.** The Semantic Web aims to make information accessible to both humans and machines, using standard formats for data and making information available in a formal and structured way. Since the advent of RDF (Resource Description Framework) there have been many efforts to extract and convert existing information in this format. In this paper we describe an adapter tool for the IMAP protocol, developed as a plugin of SquirrelRDF[1], which allows users to query IMAP mailboxes using SPARQL. The information returned looks like RDF, is always current, and can be reused and integrated inside other applications.

## 1 Introduction

The Semantic Web is a new Web paradigm that aims to make information accessible to both humans and machines [1], using standard formats for data and making information available in a formal and structured way. This means that to make it work inside the current Web it is necessary, on the one hand, to publish new information so it's *meaningful* for machines, and on the other hand to convert old data so they're available in new, more standard and structured formats.

While both of these approaches are currently being studied by the Semantic Web community, the latter is probably the one which seems more challenging from a technical point of view. And while the task might be difficult for free, unconstrained text, it becomes much easier for information already published in a structured way. This is, fortunately, the case of many standard file formats and protocols.

This kind of conversion can be usually done in two ways: the first one is a *batch* conversion, and is run once (or periodically) on the whole data source, while the second one is an *online* conversion, which is run on the fly on the single pieces of information which need to be accessed. To be more precise, tools of this last kind allow users to query the knowledge base as if it already was described in the destination format, and translate only the results of the query while they are given as an answer to the user.

Each of the two approaches has its pros and cons: batch conversion is better suited if the data source is not going to be updated and whenever there's the need

[1] `http://jena.sourceforge.net/SquirrelRDF`

to work offline (that is, disconnected from the original source of data); online conversion requires a live connection to the data source, however the information it returns is always current and consistent with the data source.

In this paper we describe a conversion tool which allows IMAP mailboxes to be queried with SPARQL, as if they originally contained information in RDF format. The tool has been developed as a plugin of SquirrelRDF, an application which is part of the Jena Semantic Web Framework[2]. As an IMAP mailbox is a source of data that is often updated, we perform an online conversion, providing users results which are always current.

The following section is devoted to a brief description of the state of the art and SquirrelRDF. In Section 3 we describe the limits we had to face when developing the IMAP plugin and the solutions we devised to address them. Section 4 shows with a higher detail the actual implementation of the plugin. Inside Section 5 we describe the tests that we ran on the application and the evaluations of their results. Finally, in Section 6, we draw our conclusions about the project and propose some possible future developments.

## 2 Related work

Since the advent of RDF [2] there have been many efforts to extract and convert existing information to this format. The World Wide Web consortium has set up wiki pages[3] to keep track of them through a list of links. The MIT SIMILE project has developed many offline conversion tools, calling them RDFizers[4], which range from e-mail to BibTex, JPEG metadata, and XML [3]. They also provide a Web service, called Babel[5], which allows for the conversion between different formats.

The SIOC project [4] aims at interconnecting different online communities (such as the ones which gather around forums and weblogs) through a common ontology and a collection of tools (called *exporters*) that convert published information into a common format. These tools can work not only on Web-based sources like RSS feeds, blogs and forums, but also on email based ones like mailing lists (see for instance the SWAML[6] research project [5]).

The D2R project [6, 7] uses a declarative language to describe mappings between relational database schemata and OWL/RDFS ontologies. The mappings can then be used to export data from a relational database to RDF (as a batch conversion tool) or to access the content of non-RDF databases as an online tool, using Semantic Web query languages like SPARQL[7].

---

[2] `http://jena.sourceforge.net`

[3] `http://esw.w3.org/topic/ConverterToRdf`
`http://www.w3.org/2005/Incubator/mmsem/wiki/Tools_and_Resources`

[4] `http://simile.mit.edu/wiki/RDFizers`

[5] `http://simile.mit.edu/babel`

[6] `http://swaml.berlios.de`

[7] `http://sites.wiwiss.fu-berlin.de/suhl/bizer/d2rmap/D2Rmap.htm`
`http://sites.wiwiss.fu-berlin.de/suhl/bizer/D2RQ/`

The Gnowsis Email project[8] provides an adapter to extract RDF information from emails. Thanks to this adapter it's possible to transform any IMAP email object (such as a store, a folder or a message) into a standard RDF model, or extract attachments from an imap message. In a paper [8] about the Gnowsis Adapter Framework (on which the Email project is built) the authors provide an interesting classification of *adapter* tools. According to the paper, adapters are software tools that can, on request, extract data from existing structured data sources and represent them as RDF. They can follow three basic approaches:

- *Graph and query adapters*, which implement the interface of an RDF graph or a query language like RDQL, SPARQL or TRIPLE
- *Concise Bounded Description* adapters, that can return a small subgraph that describes exactly one resource in detail
- *File extractors*, that read files, parse them and return some meta-data that was extracted from the data stream

According to this classification, SquirrelRDF [9] is a Graph and Query adapter, as it allows non-RDF data stores to be queried using SPARQL. It currently includes support for relational databases (via JDBC) and LDAP servers (via JNDI). It provides an ARQ QueryEngine (for Java access), a command line tool, and a servlet for SPARQL http access. For instance, running the command line tool with the following query (directed to HP LDAP server):

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix hp: <http://jena.hpl.hp.com/schemas/hpcorp#>
SELECT ?mbox ?manager_name
WHERE
{
        ?person foaf:name "Davide Eynard" .
        ?person foaf:mbox ?mbox .
        ?person hp:manager ?manager .
        ?manager foaf:name ?manager_name .
}
```

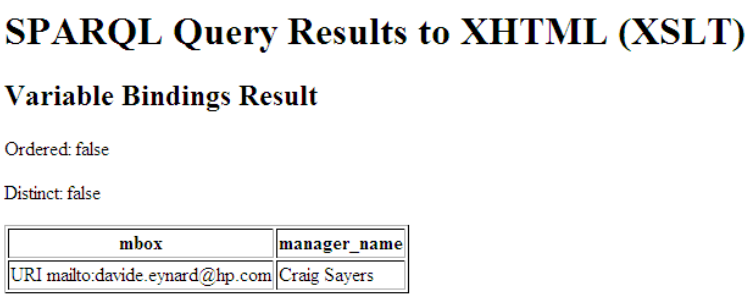
returns the following result:

```
-------------------------------------------------
| mbox                          | manager_name  |
=================================================
| <mailto:davide.eynard@hp.com> | "Craig Sayers" |
-------------------------------------------------
```

The servlet tool, instead, generates an XML file containing SPARQL query results and then uses an XSLT script to format them into XHTML. The output is shown in Figure 1, while the XML code looks like this:

[8] http://www.gnowsis.org/Projects/gnowsis_email

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="xsl/result2-to-html.xsl"?>
<sparql
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xs="http://www.w3.org/2001/XMLSchema#"
    xmlns="http://www.w3.org/2005/sparql-results#" >
  <head>
    <variable name="mbox"/>
    <variable name="manager_name"/>
  </head>
  <results ordered="false" distinct="false">
    <result>
      <binding name="mbox">
        <uri>mailto:davide.eynard@hp.com</uri>
      </binding>
      <binding name="manager_name">
        <literal>Craig Sayers</literal>
      </binding>
    </result>
  </results>
</sparql>
```



**SPARQL Query Results to XHTML (XSLT)**

**Variable Bindings Result**

Ordered: false

Distinct: false

| mbox | manager_name |
|---|---|
| URI mailto:davide.eynard@hp.com | Craig Sayers |

**Fig. 1.** The HTML output of a SPARQL query, as returned by SquirrelRDF servlet.

The advantages of such a tool are quite clear: information looks like RDF and query results can be easily integrated inside other applications. As an example, in [10] SquirrelRDF is used to solve the real-life problem of automatically extracting complex information from an LDAP directory. In [11] the techical details about this project have been described, together with a tutorial about setting the system up and expanding it to provide support to different types of SPARQL queries and output formats.

## 3 Current limits and a proposed solution

Of the many different formats which were already available on the Internet, we decided to focus on e-mails because they provide useful information not only about their actual content, but also about the people who wrote them, the relations between these people (ie. who wrote to whom) and their dynamics (ie. who replied about what).

Current adapters for emails can be roughly divided in two categories: those which work on information saved on the client side (for instance on mailboxes in *mbox* format, or some application-dependant tools), and those which work by downloading data from servers (connecting to IMAP or POP3 servers).

We decided to work on information stored on the server and not on the client side, because we didn't want to stick with some specific application or format. Also, as we wanted to access current information, we chose to build a query adapter for IMAP servers. Gnowsis Email project looks very similar to the one we had in mind, but it apparently does not allow to directly query the IMAP server with SPARQL. With this premises, SquirrelRDF instead seemed to satisfy all our prerequisites, so we chose it as a starting point to develop our adapter.

### 3.1 The Data Model

To write our IMAP plugin we chose to first study how the existing ones (RDB and LDAP) worked. In particular, the mapping used by the LDAP plugin is very compact and efficient and allows to simultaneously specify search constraints and extract the right attributes from the results returned by the LDAP server.

Unfortunately, this kind of mapping is not usable for the IMAP plugin. In fact, there's a huge asymmetry between the way messages are searched and how fields are extracted from search results, that requires us to describe the two operations in different ways. Moreover, we wanted to develop something flexible enough to allow programmers to create new plugins in an easy way, given similar kinds of problems. So we thought about a different mapping model (albeit inspired by the LDAP one), which maps properties specified in the SPARQL query with *methods* that are called by our application.

The details of our mapping model can be found in Figure 2. Every imap:Map can have one or more *server profiles* and one or more *property mappings*. The first ones are used to describe the properties of the different servers the application can connect to: each server has an alias (which is used inside SPARQL queries to refer to it), an address and a port to connect to, a username, a password, and a default folder to connect to if no folder is specified inside the query.

Property mappings are used to match properties with methods: every mapping contains the name of the property which has to be mapped; a property type can be specified, so it's possible to run specific actions over it; also, every property can be matched with three methods (through `imap:method`, `imap:extract`, and `imap:parse`) which are called inside the application (more details on it in Section B.3).

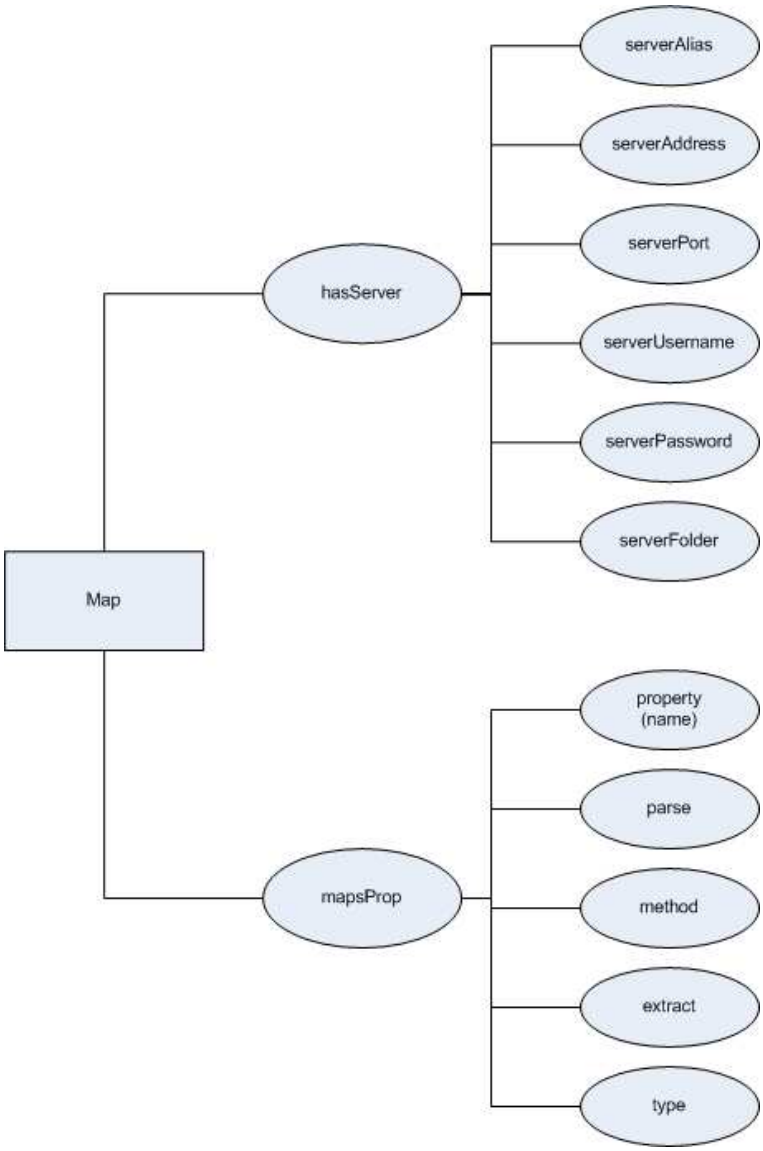



**Fig. 2.** A description of the IMAP Map and its properties.

### 3.2 The Email Ontology

As described above, our mapping describes connections between predicates used inside SPARQL queries and methods. These predicates come from an ontology which describes every single piece of information we can query: in our specific case, the ontology describes emails and their different fields.

| Field name | Predicate name | SearchTerm (JavaMail) | Extract Method (JavaMail) |
|---|---|---|---|
| Body | email:body | BodyTerm | getContent, getBodyPart |
| Subject | email:subject | SubjectTerm | getSubject |
| From | email:from | FromStringTerm | getFrom |
| To | email:to | RecipientStringTerm | getRecipients |
| Cc | email:cc | RecipientStringTerm | getRecipients |
| Bcc | email:bcc | RecipientStringTerm | getRecipients |
| MessageID | email:messageID | MessageIDTerm | getMessageNumber |
| *headers* | email:header | HeaderTerm | getAllHeaderLines |
| *flags* | email:flag* | FlagTerm | getFlags |
| Date | email:date | ReceivedDateTerm | getReceivedDate |
| Sent | email:sdate | SentDateTerm | getSentDate |

**Fig. 3.** Mapping between email field names, predicates from our email ontology, JavaMail SearchTerm subclasses and JavaMail extract methods.

In the first two columns of Table 3, the field names for a generic email and the predicate names we chose for our ontology are shown. In the third column `SearchTerm` classes are specified: these are the classes used by JavaMail (see Appendix B, "The JavaMail API") to specify the search terms inside an IMAP query. In the fourth column, extract methods are shown: these are all methods provided by the JavaMail `Message` object, which allow to extract information from the messages returned by the IMAP server.

Looking at the table, a direct mapping between the predicates and the classes/methods from JavaMail might seem the most intuitive and the easiest one. However, we decided to put one more layer of abstraction in the process, creating mappings between predicates and methods which use these objects to search and extract information. One of the main advantages of this approach is the possibility to create many different methods which get the same pieces of information, but then convert them in different ways.

For instance, the `getFrom` method is able to extract the sender from an email message. However, the From field is built up of a name and an email address: in some cases we might be interested in both, while in others we might want to extract only one of them. Using our approach it is possible to create three different methods (ie. `extractFrom`, `extractFromName`, and `extractFromAddress`) and match them with three different properties; also, one might just keep one property and update the mappings depending on what kind of information a particular query has to return.

Actually, mappings can be changed quite easily: they're saved in RDF format inside a configuration file (called `imap_map.n3` - see Section C.4 for an example) and changing them is just a matter of editing this text file. Even creating a new mapping from scratch is rather easy. Once a predicate name is chosen, its mapping can be described like in the following example:

```
imap:mapsProp [ imap:property email:body ;
                imap:method  "searchBody" ;
                imap:extract "extractBody" ;
                a imap:ExactStringProperty ;
] ;
```

In this case, for instance, we chose to map the `email:body` property with two different methods: `searchBody` to search the IMAP server for a particular body, and `extractBody` to extract body information from the results. Also, this property requires the specified search term to be matched not as a substring, but as an exact string.

As it appears in the table, the email ontology is very simple and flat and it doesn't reuse already existing ontologies. However, as it is described only through mappings inside one text file, it's very easy to change it and make it more complex and compatible with other ontologies. For instance, it is possible to map the subject with a `dc:subject`, the sender with a `dc:creator`, and the date with a `dcterms:dateSubmitted`.

## 4 System design and implementation

The plugin has a structure which is very similar to the other SquirrelRDF components: an RDF configuration file describes the mappings and all the parameters needed to connect to the server(s); an RDF schema describes how the configuration file can be built; different classes provide different ways to access the data (ARQ QueryEngine, command line, servlet). In this section we describe the main choices we made during the design phase and how we implemented them, highlighting the differences between our plugin and the already existing ones.

The structure of SquirrelRDF is shown in Figure 4. From a user perspective, there are two ways to use the application: the command line tool and the servlet. Both of them access a configuration file whose name can be specified as a parameter during startup: this file contains the mappings and the parameters needed for the plugin to run (for the IMAP plugin, this is the same `imap_map.n3` file we previously described) and its vocabulary is defined inside another RDF file containing its schema (ie. `ImapMap.n3` in our case).

As the configuration defines the type of the mapping, once it is loaded SquirrelRDF can choose which plugin has to be used. Then it parses the SPARQL queries according to the mappings, retrieving information from the servers when needed and binding them with the variables used inside the queries. Depending on the plugin, SquirrelRDF uses different protocols and APIs to connect to the

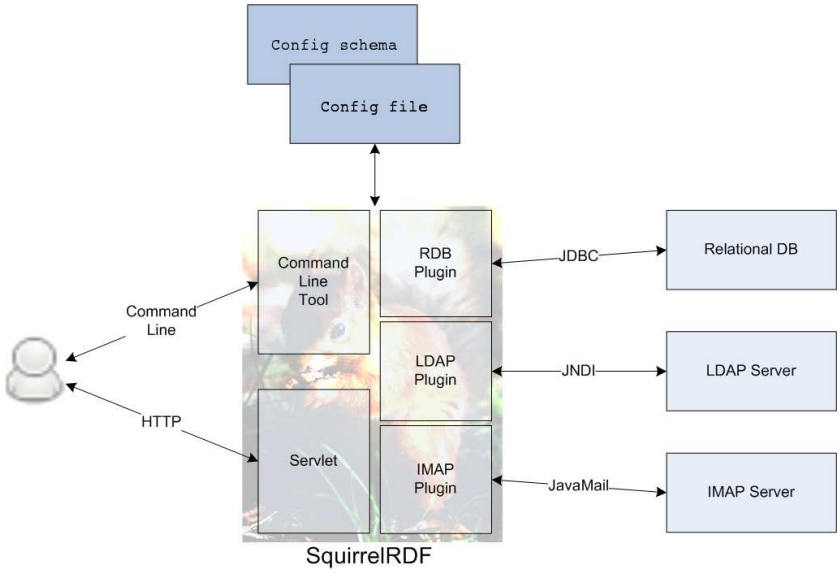



**Fig. 4.** SquirrelRDF structure

servers; also, the methods used to extract information from search results change from one plugin to another. However, all these differences are invisible to the user, who can just query the data sources as if they were RDF graphs.

### 4.1 Multiserver queries and preparsing

One of the main purposes of describing email messages with RDF is the possibility of integrating their information with different and heterogeneous data sources. However, of course, we also wanted to provide ways to integrate the same information with other *homogeneous* sources, that is other IMAP accounts, folders, and servers.

For example, one user might have different email accounts from different providers, and want to find a particular message without remembering where it is saved. In another use case, one might want to be able to see at a glance all his new emails, without having to care about where they are stored. A researcher might need to search in different mailing lists (folders) if people are writing about the same subjects, or if the same writers appear in different communities, and so on.

Allowing users to specify servers and folders inside a SPARQL query means specifying new properties for messages: given a message as a subject, two new predicates `imap:server` and `imap:folder` have been created. Two main problems arise from this feature extension: the first one is related to where and

how connection data (server IP, port, login and password) have to be specified; the second is related to how we want to manage predicates like these inside a SPARQL query.

We decided to associate connection information with a server alias, so that users just have to specify servers and folders as string literals inside their own queries. The alias is saved inside the configuration or, in the case of the servlet application, can be specified at runtime. An example of a multiserver query is shown in Section A.6.

The management of "special" predicates or triples is done with a `preParseQuery` method. This method is called before the actual parsing inside the `ImapSubQuery` class: it checks all the triples in the subquery and if one of them has a property whose type is `ParseProperty` then it calls (using reflection) the matching method from the `ParseMethod` class. If the method returns true then the triple has to be deleted from the query, otherwise it can be kept.

For instance, suppose we have the following lines in the configuration file:

```
imap:hasServer [ a imap:Server ;
                 imap:serverAlias        "server1" ;
                 imap:serverAddress      "localhost" ;
                 imap:serverUsername     "user01" ;
                 imap:serverPassword     "pass01" ;
                 imap:serverFolder       "INBOX" ; ];

imap:mapsProp  [ imap:property email:server ;
                 imap:parse "setServerAlias" ;
                 a imap:ParseProperty ; ] ;
```

and suppose the user enters the following query:

```
select *
where{
        ?x email:subject        ?subj1 .
        ?x email:server         "server1" .
}
```

When the second triple is evaluated this is what happens: given that `email:server` is a `ParseProperty`, the `preParseQuery` method will call its matching method (that is `setServerAlias`, from the `ParseMethod` class). This method reads the server alias from the current triple ("server1") and whenever the program has to connect to an IMAP server it will do it using the parameters specified in the configuration file (that is, it will connect to localhost using "user01" as a login and "pass01" as a password). Finally, as the method returns the boolean value `true`, the triple will be deleted from the query.

### 4.2 SPO queries

An SPO query is a single triple pattern, with optional subject (parameter "s"), predicate (parameter "p"), and object (parameter "o"). This is usually inserted

in SPARQL queries as a test pattern, or to get all the triples for the available resources. As the other two SquirrelRDF plugins didn't support these kinds of queries, we wanted to work on this to see if we could provide some useful contribution to the project we borrowed so much from.

The default behavior of SquirrelRDF (at least for its LDAP plugin) is to build a mapping between the variables that appear inside the SPARQL query and the values that are extracted from the data source.

The mapping is done inside the `ImapSubQuery` class using a list of `HashMap` objects. Their keys are String objects containing the variable names; their values are Jena Nodes (`com.hp.hpl.jena.graph.Node`) containing the pieces of information extracted from email messages. According to this model, the results can be seen as many rows inside a database table, where each binding specifies in which column a particular value has to be saved (Figure 5).

| x | subject | from | date |
|---|---|---|---|
| nodeID b0 | email without attachment | eynardd <eynardd@localhost> | 2007-08-28T22:25:25Z |
| nodeID b1 | email with attachment | eynardd <eynardd@localhost> | 2007-08-28T22:24:27Z |
| nodeID b2 | another test | eynardd <eynardd@localhost> | 2007-08-25T22:21:31Z |
| nodeID b3 | mail with empty body | eynardd <eynardd@localhost> | 2007-08-25T22:17:36Z |

**Fig. 5.** A result from a normal SPARQL query.

| s | p | o |
|---|---|---|
| nodeID b0 | http://davide.eynard.it/rdf/email#flagNew | false |
| nodeID b0 | http://davide.eynard.it/rdf/email#flagRecent | false |
| nodeID b0 | http://davide.eynard.it/rdf/email#subject | Comment on Wikipedia page (Semantic Wikis) |
| nodeID b0 | http://davide.eynard.it/rdf/email#to | eynardd@localhost |
| nodeID b0 | http://davide.eynard.it/rdf/email#date | 2007-06-21T23:22:44Z |
| nodeID b0 | http://davide.eynard.it/rdf/email#from | eynardd <eynardd@localhost> |
| nodeID b0 | http://davide.eynard.it/rdf/email#flagDeleted | false |
| nodeID b0 | http://davide.eynard.it/rdf/email#sdate | 2007-06-21T23:22:44Z |
| nodeID b0 | http://davide.eynard.it/rdf/email#flagSeen | true |
| nodeID b0 | http://davide.eynard.it/rdf/email#flagDraft | false |
| nodeID b0 | http://davide.eynard.it/rdf/email#flagAnswered | false |
| nodeID b0 | http://davide.eynard.it/rdf/email#flagFlagged | false |
| nodeID b0 | http://davide.eynard.it/rdf/email#messageID | 1 |

**Fig. 6.** A result from an SPO query.

When an SPO query is issued, the results cannot fit into rows anymore, but rather in blocks: in our particular case, every email message has many rows describing all its properties one by one and their values. So we decided to translate this with a multiple bindings data structure (Figure 6), which can hold all the results returned by every single message (and which, in the simplest case, can just be a block made of one row).

To manage this kind of queries we took advantage of the preparsing feature: the variable triple is detected in advance, deleted from the query and a flag is set to warn that a special query has been issued. Then, for each message that satisfies the search constraints (potentially all messages), a set of fields is extracted and transformed into bindings. It is possible to change how this set is built just by toggling the `CheckProperty` type inside the configuration file: this allows to limit data transfer and optimize performances.

As an example, suppose we have the following lines in the configuration file:

```
imap:mapsProp [ imap:property email:subject;
                ...
                a imap:CheckProperty ; ];

imap:mapsProp [ imap:property email:from;
                ...
                a imap:CheckProperty ; ];

imap:mapsProp [ imap:property email:date;
                ...
                a imap:CheckProperty ; ];
```

and that all the other properties are not of this type. Then the results of the query will include only these three predicates, when present.

According to this model, the main limit in the implementation of SPO queries is the server providing the information. In case of unconstrained queries, it should return all the elements it contains: while this is feasible with IMAP, it could be impossible with other servers.

## 5 Tests and evaluations

To test the application, we chose to feed it different families of queries (see Appendix A for some examples):

- basic ones, which just ask for properties of email messages;
- queries using `OPTIONAL` clauses;
- queries using `FILTER` clauses;
- SPO queries;
- multiserver queries, mixing information between different servers and
- folders or merging it with `UNION` clauses.

The application returns the expected results; however, during the tests we had to face some IMAP limits and found some workarounds for them.

**Exact String Search** The search function within IMAP does not match exact strings, but just searches for substrings within email fields. This is particularly limiting because, for instance, when you write the triple

```
?x email:subject "This is a test" .
```

you expect only messages whose subject *is* (and not *contains*) the specified string. A workaround for this problem consists of checking the result messages before extracting information from them: if a particular field matches with a property defined as `ExactStringProperty` inside the configuration file, then it is compared with the result and if the strings do not match exactly the message is dropped before any other evaluation.

**Date Search** The date search function provided by IMAP is not precise, that is it doesn't take into account time but just returns all the messages received or sent on one particular day. Of course, a fix similar to the previous one could be implemented for date fields. However, an alternative workaround is shown in the "FILTER with dates and strings" section of Appendix A: specifying the date constraint both in the query triple and in the FILTER allows to obtain the expected result.

**Server lag** Usually, filtering results with just a FILTER clause is not very efficient: this is due to the fact that in this case all the messages are downloaded, then the filter is applied on them. What we would like to have, instead, is a preliminary filter on the IMAP side which would allow us to download only the messages we're interested in. To accomplish this task on message bodies (which are usually the heaviest part of emails to download), we created an ad-hoc predicate that we called `imap:bodyfilter` (for an example, look at Section A.4). This predicate maps to the very same method used for `imap:body`, but it doesn't require the string match to be exact. In this way, messages are first filtered with a normal substring search on the IMAP server, then they're FILTERed with a regular expression. Of course, this solution cannot be easily applied with very complex regular expressions, but is useful in most common cases.

**OPTIONAL subqueries** Queries which contain an OPTIONAL clause are usually managed in a more complex way than in the basic case: the non-optional part is first evaluated, then for all the results the OPTIONAL ones are, resulting in our case in $N \times M$ searches on the IMAP server (where $N$ is the number of results, and $M$ the number of optional clauses). This raises a big performance problem, which can become more serious depending on the connection speed between the client on which SquirrelRDF runs and the IMAP server. Our solutions worked primarily on disabling optionals when possible and trying to return empty results instead of *null* ones, but we think that further study is needed on this topic.

### 5.1 A mashup test

One of the main advantage of SquirrelRDF is that the information extracted from the IMAP server is provided in very standard and common formats such as RDF and XML, so it can be easily reused within other applications. As an example, we developed some very simple mashups with Yahoo Pipes[9] which get email data from the servlet, augment them with Internet searches, and then publish everything in standard formats such as RSS.

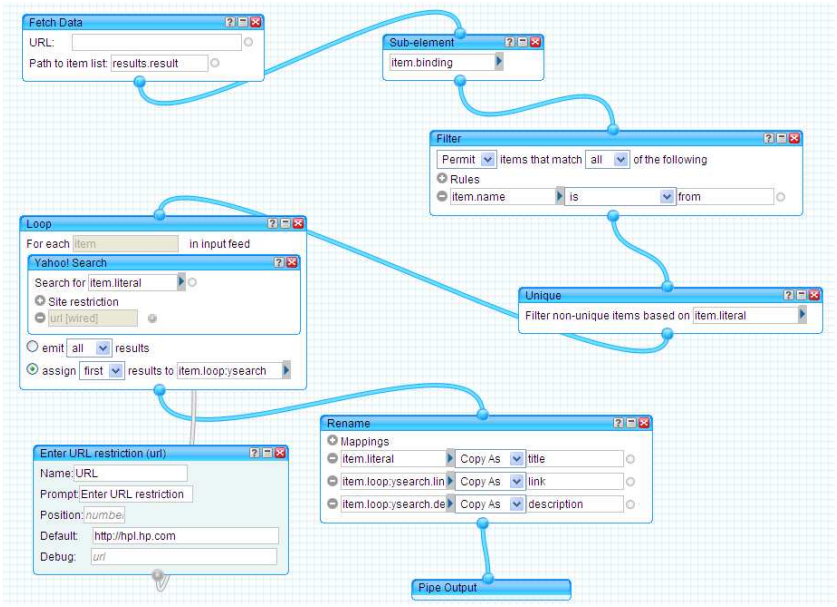



**Fig. 7.** A *Pipe* extracting email authors from a mailbox and searching for them on Yahoo.

These tests gave us some very interesting results: first of all, as SquirrelRDF allows to automatically convert structured information into RDF, querying the IMAP server and connecting the results to a mashup tool like Yahoo Pipes is really easy and fast. In fact, the *Fetch Data* widget allows users to load any XML page and extract information from it: we used this feature to download SquirrelRDF servlet results page and automatically extract the list of bindings. Then, information can be parsed, filtered, and reused inside other plugins.

This chance of redirecting information from one service to another immediately proved to be quite useful: for instance, we used it to automatically extract all the authors from the emails, search them on the internet and provide the results of these searches as additional information about them. The final product

[9] `http://pipes.yahoo.com`

is a document which contains the list of authors, followed by a link to a page which is related to them, and a brief summary of that page.

Finally, as these new data can be published in very common formats such as RSS, they can be read, shown or furtherly processed by many different applications. As an example, we imported the results of the previously described pipe as Firefox Live Bookmarks, so the additional information about email authors is always available within the browser.

## 6 Conclusions and future work

The plugin we developed gives the expected results and integrates well with the SquirrelRDF application. Performances are good on the local IMAP server, and they're expected to be more than acceptable on a remote one. Information returned by the application is current and provided in a standard (RDF/XML) format, so it can be easily shared and reused inside other applications.

As a planned future work, as there are still some limitations in the model (in this very moment it only works with flat email ontologies), we've begun to implement a new version which will enable more complex ones. By *flat*, we mean that we still can't deal with second-level relationships: for instance, while we can answer a query like

```
select ?from where
{
        ?x email:from           ?from .
}
```

where `?from` is a variable containing an email address, we can't answer another like the following:

```
select ?realname ?address where
{
        ?x email:from           ?from .
        ?from email:realname    ?realname .
        ?from email:address     ?address .
}
```

where `?from` is an anonymous node linking to a real name and an address value.

Our test queries showed that connections to the IMAP server are the bottleneck in performances of SquirrelRDF: for this reason we're planning to optimize them with caching (both of connections themselves and of search results). Also, we've already started to work on a new plugin which would use the same kind of predicate-to-method mapping: this plugin will connect via HTTP on a Twiki website and allow users to access wiki contents with SPARQL queries. Finally , we're planning to experiment more with mashup services and build some custom ones to enrich and merge all the information we can extract thanks to SquirrelRDF.

# A Test queries

## A.1 Basic query

```
# With this SPARQL query we try to ask as many properties as we can
# (almost all the ones which are defined inside the imap_map file. The
# reason of the "almost" will be explained in detail inside next queries)

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
prefix email: <http://davide.eynard.it/rdf/email#>

select * where
{
        ?x email:messageID      ?messageID .
        ?x email:from           ?from .
        ?x email:to             ?to .
        ?x email:subject        ?subject .
        ?x email:body           ?body .
        ?x email:date           ?date .
        ?x email:sdate          ?sdate .
        ?x email:header         ?headers .
        ?x email:flagAnswered   ?flagAns .
        ?x email:flagDeleted    ?flagDel .
        ?x email:flagDraft      ?flagDra .
        ?x email:flagFlagged    ?flagFla .
        ?x email:flagRecent     ?flagRec .
        ?x email:flagSeen       ?flagSee .
        ?x email:flagNew        ?flagNew .
}
```

## A.2 OPTIONAL fields

```
# Some fields, like Cc and Bcc, are often empty: this means that most
# of the times you query for "?message email:cc ?cc" you will have
# very few results. This is the reason why fields like this have to
# be kept OPTIONAL (try the difference taking away the OPTIONAL clauses
# from the query).

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
prefix email: <http://davide.eynard.it/rdf/email#>

select * where
{
        ?x email:subject        ?subject .
        ?x email:from           ?from .
        ?x email:to             ?to .
```

```
        OPTIONAL{       ?x email:cc     ?cc }.
        OPTIONAL{       ?x email:bcc    ?bcc }.
}
```

### A.3 FILTER with dates and strings

```
# Now on to special examples: date is the first property that has to
# be used in a particular way. If you send a query like the following,
# you might happen to receive not only the message sent at that particular
# date and time, but all the ones sent the very same day. This is due to
# IMAP server inner working and cannot be changed on the IMAP side.
# However, the desired result can be obtained just by uncommenting the
# FILTER constraint.

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
prefix email: <http://davide.eynard.it/rdf/email#>

select ?date ?subject where
{
        ?x email:subject        ?subject .
        ?x email:date           ?date .
        ?x email:date           "2007-07-06T22:48:18Z"^^xsd:dateTime .
        #FILTER (xsd:dateTime(?date) = "2007-07-06T22:48:18Z"^^xsd:dateTime) .
}

# NOTE that the same result is obtained also if you comment the line
# above the FILTER constraint. However, if you have logging enabled
# for INFO messages you'll notice that the number of downloaded messages
# changes much. In fact, when you specify a date a search is done on the
# IMAP server and the results are "pre-filtered", so performance is better.
# A more specific example on this in the next query
```

### A.4 FILTER with strings

```
# In this query a search for the BODY content is run on the IMAP server.
# The "bodyfilter" property has been created to prefilter bodies on server
# side, while "body" will contain the full message body. Note that the
# first search done on IMAP server is just a normal substring search,
# while the second one is a full working regular expression run over the
# downloaded messages

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
prefix email: <http://davide.eynard.it/rdf/email#>

select * where
{
```

```
        ?x email:subject        ?subject .
        ?x email:bodyfilter     "http://en.wikipedia.org/wiki/Semantic_Wiki" .
        ?x email:body           ?body .

        FILTER regex(?body, "http://en.wikipedia.org/wiki/Semantic_Wiki[\\s]", "i") .
}
```

### A.5 SPO Queries

```
# The following is a very simple SPO query, used to have a list of the predicates
# in our email ontology that are actually used inside the application.

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
prefix email: <http://davide.eynard.it/rdf/email#>
select distinct ?p where{
?s ?p ?o.
}
```

```
-----------------------------------------------------
| p                                                 |
=====================================================
| "http://davide.eynard.it/rdf/email#flagSeen"      |
| "http://davide.eynard.it/rdf/email#flagDraft"     |
| "http://davide.eynard.it/rdf/email#messageID"     |
| "http://davide.eynard.it/rdf/email#date"          |
| "http://davide.eynard.it/rdf/email#flagRecent"    |
| "http://davide.eynard.it/rdf/email#flagFlagged"   |
| "http://davide.eynard.it/rdf/email#sdate"         |
| "http://davide.eynard.it/rdf/email#flagNew"       |
| "http://davide.eynard.it/rdf/email#subject"       |
| "http://davide.eynard.it/rdf/email#to"            |
| "http://davide.eynard.it/rdf/email#from"          |
| "http://davide.eynard.it/rdf/email#flagAnswered"  |
| "http://davide.eynard.it/rdf/email#body"          |
| "http://davide.eynard.it/rdf/email#flagDeleted"   |
| "http://davide.eynard.it/rdf/email#header"        |
| "http://davide.eynard.it/rdf/email#cc"            |
-----------------------------------------------------
```

### A.6 Multiserver query

```
# The following query searches for email which share the same subjects and
# come from two different servers and folders (actually, emails sent from
# "server1" and emails received by "server3".

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
```

```
prefix email: <http://davide.eynard.it/rdf/email#>

select *
where{
        ?x email:subject        ?subj1 .
        ?x email:server         "server1" .
        ?x email:folder         "Sent" .
        ?y email:subject        ?subj1 .
        ?y email:server         "server3" .
}
```

---

```
# the following query shows all the subjects of messages contained in the INBOX
# folders of server1, server2, and server3

PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
prefix email: <http://davide.eynard.it/rdf/email#>

select *
where{
        { ?x email:server "server1" ;
         email:subject ?subj }
        UNION
        { ?x email:server "server2" ;
         email:subject ?subj }
        UNION
        { ?x email:server "server3" ;
         email:subject ?subj }
}
```

# B Software details

## B.1 The JavaMail API

As described in [8], an adapter tool might depend on the data source (that is, on its format and the ways to access it), so a preliminary analysis is needed. Fortunately, email is a mature standard and IMAP protocol is well described by its RFC [12]. Moreover, the JavaMail API[10] provides an easy way to access an IMAP server which is compliant enough with the protocol, so we chose it as a gateway between our tool and IMAP servers.

The main JavaMail API classes we used are `IMAPStore` and `IMAPFolder`: the first one is used to manage connection (plain or SSL, with `IMAPSSLStore`) and authentication, and to specify which is the current folder; the second one offers all the methods needed to manage messages saved inside a particular folder. Between these, the `search` method proved to be particularly useful for our purpose of building a SPARQL-to-IMAP translation. In fact, even if filtering can be done client-side by the SPARQL interpreter, we decided to take advantage of advanced IMAP search features too: this allows users to pre-filter messages on the server side, lowering the number of bytes they have to download.

The search method requires a `SearchTerm` object as input and returns an array of `Message` objects as an output. SearchTerms are objects which follow quite straightly the RFC specifics: there's one for each type of field one might want to search and it is possible to join many of them through `AND` terms. Messages are objects which provide all the methods needed to extract the different components of an email message. The third and fourth columns of table 3 show the mappings between the different email fields, SearchTerms and Message methods.

## B.2 Class structure

The IMAP plugin for SquirrelRDF is composed of eight classes (plus two "main" ones which implement the command line and the servlet tools). A brief description of them follows:

- `ImapMap` class defines the schema of the `imap_map.n3` configuration file. Actually, the RDF schema is defined inside the `ImapMap.n3` RDF file, then the class is created automatically by the `schemagen` tool. Once built, it provides all the vocabulary definitions needed to describe the RDF mapping model inside the Java application.
- `ImapQueryEngine` and `ImapSparqlMap` are mostly refactorizations of the original classes used for the LDAP plugin. The first one extends the ARQ QueryEngine object, providing access to the query plan elements; the second one analyses them, dividing the queries in blocks of triples who share the same subject, and calling the ImapSubQuery class (see below) to work on these blocks. This grouping by subject comes very useful to us, as we're mostly dealing with email messages and it's much more efficient to get all the information about one message at once.

[10] `http://java.sun.com/products/javamail/`

- `ImapSubQuery` is the heart of the IMAP adapter. It gets "subquery" elements (that is, the groups of triples which share the same subject), builds the IMAP queries, extracts the required fields from downloaded messages, and finally creates and manages the bindings between SPARQL variables and their values.
- `MatchMethod`, `ExtractMethod` and `ParseMethod` are the classes which contain the methods used in the mappings. They're all instantiated inside the ImapSubQuery class, and their usage is better described in the following section.
- `CfgManager` is the class used to manage the pieces of information which are common between all the other different classes. Currently, it contains the RDF Model object which describes the mapping and all the methods needed to access it.

### B.3 Reflection

In our particular case, reflection is used to manage the dynamic calling of methods from some particular classes, allowing users to specify the name of these methods in the `imap_map.n3` configuration file that is loaded at runtime.

The three main uses of these methods are search, extraction and preparsing. The `MatchMethod` class contains all the methods that build up the SearchTerms used to search email messages on the IMAP server. The `ExtractMethod` class contains the methods used to extract pieces of information from an email message (such as the subject, the body and so on). The `ParseMethod` class contains methods that are called in a subquery preparsing phase, for instance the ones which set the server or the folder for the current query.

The first step we accomplish is instantiating the class with the right parameters: all of them require at least a `CfgManager` object and the current triple; `ExtractMethod` also requires the current message to extract information from, and `ParseMethod` needs the current `ImapSubquery` to change some of its parameters. Once an instance of the class is created, it is then possible to call its only public method, called `run`.

The `run` method first finds the name of the right method to call, extracting the predicate from the triple and then matching it with the configuration. Then it gets the method from its name using `getClass().getDeclaredMethod` and invokes it. Every single method has then access to the configuration model, the triple, and the additional parameters, as they had been saved into private attributes when the class was instantiated.

As an example, suppose we have the following lines inside the configuration file:

```
imap:mapsProp [ imap:property email:subject;
                imap:method "searchSubject" ;
                imap:extract "extractSubject" ;
                a imap:ExactStringProperty ; ];
```

```
imap:mapsProp [ imap:property email:body ;
                imap:method  "searchBody" ;
                imap:extract "extractBody" ;
                a imap:ExactStringProperty ;
] ;
```

and suppose the user enters the following query:

```
select *
where{
        ?x email:subject        "test" .
        ?x email:body           ?body .
}
```

When the query is parsed, predicates are extracted from the triples and the matching search methods are called if the object is not variable or if the variable is bound to some value: in this case, only the subject has been specified so only the `searchSubject` method (from the `MatchMethod` class) is called. Then, when the IMAP server returns the results of the search (in this case, all the messages whose subject *contains* the string "test"), the extract methods matching the specified predicates (`extractSubject` and `extractBody` from the `ExtractMethod` class) are called. Moreover, as both the predicates are ExactStringProperties and the subject has been specified, the application will also check that, of all the messages returned by the IMAP search, only the ones whose subject actually *is* "test" will be returned.

The main advantage of this technique is that it's very flexible and allows users to change drastically the behavior of the application with few and simple changes. For instance, alternative "searchFrom" methods can be built to extract the whole From field, only the address or only the name of the sender; then the user can change the method that has to be called just by updating the matching string in the configuration file. Moreover, methods can be used not only to return a value formatted in a particular way, but also to perform particular actions inside the system (such as modifying the configuration or retrieving information outside of the current triple).

This solution also has some drawbacks: in fact, to avoid managing too many particular cases, during the design phase it's better to define some standards and constraints in method calls, losing part of their flexibility. For instance, we decided to keep the type of the returned values fixed for all the methods in a single class (`SearchTerm` for `MatchMethod`, `String` for `ExtractMethod`, `Boolean` for `ParseMethod`) and we used roughly the same parameters for all of them. Anyway this is not a huge limitation in our case, and we think it might be relaxed in more complex cases, at the cost of more type checks inside the application code.

# C User Manual

## C.1 Installation

The source code can be checked out from SVN (email the authors for the URL of the SVN repository). Actually, on the repository a full version of SquirrelRDF is present, together with the plugin. However, as the original SquirrelRDF files have been modified for debugging purposes, it is strongly suggested that the original version of SquirrelRDF is downloaded from their SVN repository and only the new or modified files are merged with it:

- `/build-user.xml`, changed to compile ImapMap.java with schemagen
- `/ImapMap.n3`, the RDF schema for the IMAP plugin configuration
- `/examples` directory, which contains the RDF maps and a subdirectory called `queries`, containing all the test queries
- `/src/com.hp.hpl.squirrelrdf.imap`, which contains all the classes needed by the IMAP plugin
- `/src/com.hp.hpl.squirrelrdf.imap.test`, which contains the test files for the plugin
- `/squirrelrdf/Query.java`, the command line client (updated to support the IMAP plugin)
- `/squirrelrdf/Servlet.java`, the servlet application (updated to support the IMAP plugin and to run under Windows)

All the needed libraries are available in the `/lib` directory:

- HSQLDB
- Jena 2.4
- JavaMail 1.4
- Jaf 1.1
- Servlet API 2.5

## C.2 Running the command line application

The command line application can be run as follows:

```
squirrelrdf.Query <config file> "the query"        or
squirrelrdf.Query <config file> [<query file>] (reads from stdin id no file)
```

for example:

```
squirrelrdf.Query examples/imap_map.n3 examples/queries/imap01.sparql
```

However, the command line application does not parse comments out of the query files, so it will probably give an error if the default queries (which all contain comments inside them) are used. A better approach is to use the `Scratch.java` application inside `/src/com.hp.hpl.squirrelrdf.imap.test`, saving the query file name inside the source code.

### C.3 Running the servlet

The prerequisites for the servlet to run are the following ones:

- Tomcat (the application has been tested with version 5.5)
- Java 5
- Jena 2.4
- all the other libraries included in the `/lib` directory

Once deployed[11], there are still some changes that have to be applied for the *IMAP* servlet to work: first of all, the `WEB-INF/map.ttl` file content has to be updated with the content of the `imap_map.n3` file. Then, the example `index.html` page has to be changed to provide the user an example query which is compatible with the IMAP configuration (you can copy one from the `/examples/queries` directory). Finally, the needed libraries have to be copied to the `WEB-INF/lib/` directory.

Once everything's set, the servlet can be used quite easily: supposing you have deployed it inside as "squirrel", and that your local Tomcat installation answers requests on port 8080, just connect to `http://localhost:8080/squirrel` to access the Web application page.

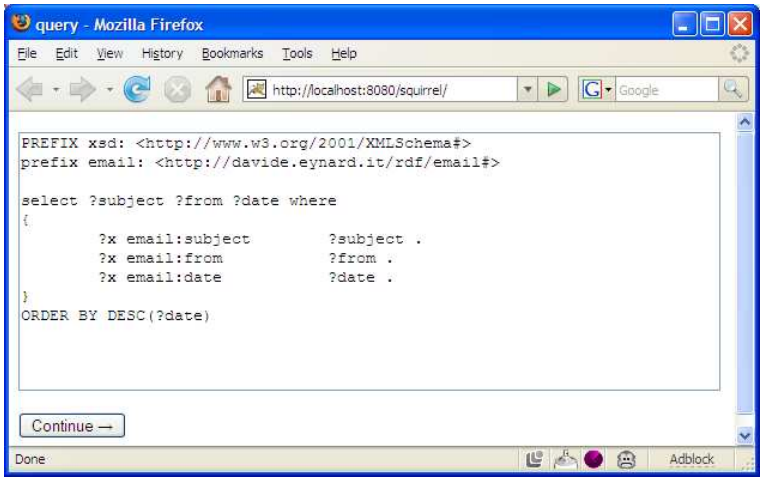



**Fig. 8.** Connection to SquirrelRDF Web application

With the IMAP servlet, different server profiles can be specified as parameters with a normal HTTP GET. That is, in the simplest case, they can be written in the adddress bar as additional parameters to the query. For instance, if the query is like the following one:

[11] Some basic instructions about how to patch the original SquirrelRDF so it works under Windows and how to deploy it can be found on HP internal wiki: just search for the page called "InstallingSquirrelRDFServlet"

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
prefix email: <http://davide.eynard.it/rdf/email#>

select *
where{
        ?x email:subject        ?subj1 .
        ?x email:server         "server1" .
}
```

then the "server1" profile can be easily changed at runtime just by specifying a different parameter in the address bar. The format for this parameter is

```
<serveralias>=<login>:<password>@<server>:<port>
```

For instance, as the URL of the results page is

```
http://localhost:8080/squirrel/model?stylesheet=
xsl%2Fresult2-to-html.xsl&query=<query>
```

if the user `user01` with password `mypass` wants to connect to the server `locahost`, default port, he can just add to the address of the results page the new parameter:

```
http://localhost:8080/squirrel/model?stylesheet=
xsl%2Fresult2-to-html.xsl&query=<query>&server1=user01:mypass@localhost
```

### C.4 Example configuration file

```
@prefix imap: <http://jena.hpl.hp.com/schemas/imapmap#> .
@prefix email: <http://davide.eynard.it/rdf/email#> .

<> a imap:Map ;
  imap:folder "INBOX" ;

  imap:hasServer [ a imap:Server ;
                   imap:serverAlias        "server1" ;
                   imap:serverAddress      "localhost" ;
                   imap:serverUsername     "user01" ;
                   imap:serverPassword     "pass01" ;
                   imap:serverFolder       "INBOX" ;
                 ];

  imap:hasServer [ a imap:Server ;
                   imap:serverAlias        "server2" ;
                   imap:serverAddress      "localhost" ;
                   imap:serverUsername     "user02" ;
                   imap:serverPassword     "pass02" ;
                   imap:serverFolder       "INBOX" ;
                 ];

  imap:mapsProp [ imap:property email:folder;     imap:parse "setFolder" ;
                                                  a imap:ParseProperty ; ];

  imap:mapsProp [ imap:property email:server;     imap:parse "setServerAlias" ;
                                                  a imap:ParseProperty ; ];

  imap:mapsProp [ imap:property email:body;       imap:method "searchBody" ;
                                                  imap:extract "extractBody" ;
                                                  a imap:CheckProperty ;
                                                  a imap:ExactStringProperty ; ];

  imap:mapsProp [ imap:property email:subject;    imap:method "searchSubject" ;
                                                  imap:extract "extractSubject" ;
                                                  a imap:CheckProperty ;
                                                  a imap:ExactStringProperty ; ];

  imap:mapsProp [ imap:property email:from;       imap:method "searchFrom" ;
                                                  imap:extract "extractFrom" ;
                                                  a imap:CheckProperty ;
                                                  a imap:EmailProperty ; ];

  imap:mapsProp [ imap:property email:to;         imap:method "searchTo" ;
                                                  imap:extract "extractTo" ;
                                                  a imap:CheckProperty ; ];

  imap:mapsProp [ imap:property email:cc;         imap:method "searchCc" ;
```

```
                                                imap:extract "extractCc" ;
                                                a imap:CheckProperty ; ];

imap:mapsProp [ imap:property email:bcc;        imap:method "searchBcc" ;
                                                imap:extract "extractBcc" ;
                                                a imap:CheckProperty ; ];

imap:mapsProp [ imap:property email:messageID; imap:method "searchMessageID" ;
                                                imap:extract "extractMessageID" ;
                                                a imap:CheckProperty ; ];

imap:mapsProp [ imap:property email:header;     imap:method "searchHeader" ;
                                                imap:extract "extractHeader" ;
                                                a imap:CheckProperty ; ];

imap:mapsProp [ imap:property email:date;       imap:method "searchDate" ;
                                                imap:extract "extractDate" ;
                                                a imap:DateProperty ;
                                                a imap:CheckProperty ; ];

imap:mapsProp [ imap:property email:flagAnswered;  imap:method "searchFlagAnswered" ;
                                                   imap:extract "extractFlagAnswered" ;
                                                   a imap:CheckProperty ; ];

imap:mapsProp [ imap:property email:flagDeleted;   imap:method "searchFlagDeleted" ;
                                                   imap:extract "extractFlagDeleted" ;
                                                   a imap:CheckProperty ; ];

imap:mapsProp [ imap:property email:flagDraft;     imap:method "searchFlagDraft" ;
                                                   imap:extract "extractFlagDraft" ;
                                                   a imap:CheckProperty ; ];

imap:mapsProp [ imap:property email:flagFlagged;   imap:method "searchFlagFlagged" ;
                                                   imap:extract "extractFlagFlagged" ;
                                                   a imap:CheckProperty ; ];

imap:mapsProp [ imap:property email:flagRecent;    imap:method "searchFlagRecent" ;
                                                   imap:extract "extractFlagRecent" ;
                                                   a imap:CheckProperty ; ];

imap:mapsProp [ imap:property email:flagSeen;      imap:method "searchFlagSeen" ;
                                                   imap:extract "extractFlagSeen" ;
                                                   a imap:CheckProperty ; ];

imap:mapsProp [ imap:property email:flagNew;       imap:method "searchFlagNew" ;
                                                   imap:extract "extractFlagNew" ;
                                                   a imap:CheckProperty ; ];
.
```

## List of References


1. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American **284**(5) (2001) 34–43
2. Lassila, O., Swick, R.: Resource description framework (rdf) model and syntax specification (February 1999) http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.
3. Butler, M.H., Gilbert, J., Seaborne, A., Smathers, K.: Data conversion, extraction and record linkage using xml and rdf tools in project simile. research report (August 2004)
4. Breslin, J., Harth, A., Bojars, U., Decker, S.: Towards Semantically-Interlinked Online Communities. In: Second European Semantic Web Conference, ESWC 2005, Heraklion,. (2005)
5. Flejter, D., ed.: Mailing Lists Meet The Semantic Web. In Flejter, D., ed.: Proceedings of the BIS 2007 Workshop on Social Aspects of the Web Poznan, Poland, April 27, 2007. (2007)
6. Bizer, C.: D2r map - a database to rdf mapping language. In: WWW (Posters). (2003)
7. Bizer, C., Seaborne, A.: D2rq - treating non-rdf databases as virtual rdf graphs. In: ISWC2004 (posters). (November 2004)
8. Sauermann, L., Schwarz, S.: Gnowsis adapter framework: Treating structured data sources as virtual rdf graphs. In Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A., eds.: Proceedings of the ISWC 2005. Number 3729 in LNCS, Galway, Ireland, Springer (November 6-10, 2005 2005) p. 1016 ff.
9. Steer, D.: Squirrelrdf (2006) http://jena.sourceforge.net/SquirrelRDF/.
10. Yung, W.: Using sparql for good: Querying ldap with squirrelrdf. Blog post (May 2007) http://wingerz.com/blog/2007/05/10/using-sparql-for-good-querying-ldap-with-squirrelrdf/.
11. Yung, W.: Bring existing data to the semantic web (May 2007) http://www-128.ibm.com/developerworks/library/x-semweb.html.
12. Crispin, M.: Internet message access protocol - version 4rev1 (March 2003) http://tools.ietf.org/html/rfc3501.