


[Quick installation](#) (Windows)
[Launching from a Lua console](#)
[Importing graphs from disk](#)
[Browsing graphs in database](#)
[Importing graphs from Lua](#)
[Importing graphs from the Web](#)
[Exporting RDF graphs](#)
[SPARQL queries](#)
[Creating rules](#)
[Using rules](#)
 [Launching a SPARQL server](#)
[Configuration of Apache httpd](#)
[Test SPARQL requests](#)
[Dynamic Lua Pages](#)

Quick installation (Windows)

Unzip the distribution archive. You will get the Lua module *StrixStore.dll*, a ready to use *Lua5.1.dll* and *lua.exe* console.

You will also get some RDF samples into the folder *samples*. The distribution zip contains also *webGet.dll*, a Lua module to download RDF files from internet, *mod_strixdb.so* the Apache module and [sparql.html](#), a tutorial file explaining how to use the SPARQL protocol (how to send SPARQL request to the Apache server).

Using the RDF store standalone

If you only want to manipulate RDF graphs you could simply use the Lua console (or any custom Lua 5.1 compatible program). This standalone use don't need the Apache HTTP server.

Launching from a Lua console

Start a Lua console, then load the StrixStore module and if you want to use Internet files, the *webGet* module.

Then open the RDF store with the function *rdf.open*. 2 parameters are expressly needed ! The database filename and the URI of default graph.

```
C:/RDF/StrixDB> lua.exe
Lua 5.1.4 Copyright (C) 1994-2008 Lua.org, PUC-Rio
> require 'StrixStore'
> require 'webGet'
> rdf.open{uri='http://mydefault/graph/uri/',file='D:/RDF/test.db'}
>
```

Importing RDF graphs

Importing from disk

We will import in the RDF Store a graph from a local disk file. We will use the sample file [foaf_sample0.rdf](#) given in the samples folder. The function *rdf.graph.import* takes a Lua table as argument. This table must have a **uri** key. On the console prompt (it is a Lua interactive console), type :

```
rdf.graph.import{uri='foaf_sample0',file='samples/foaf_sample0.rdf'}
```

We now will import an other local file ([relationship.rdf](#)) but now with an absolute **URI**.

```
rdf.graph.import {uri='http://www.perceive.net/schemas/relationship/',
file='samples/relationship.rdf'}
```

Importing from Lua

It is possible to create graphs programmatically with Lua : the Lua function *rdf.graph.create* does this. The following code create a graph with uri=[http://example/foaf_sample1](#) with a Lua string containing triples in Turtle format.

```
rdf.graph.create('foaf_sample1',
[[@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Johnny Lee Outlaw" .
_:a foaf:mbox <mailto:jlow@example.com> .
_:b foaf:name "Peter Goodguy" .
_:b foaf:mbox <mailto:peter@example.org> .
_:c foaf:mbox <mailto:carol@example.org> .
]])
```

Importing from the Web

Importing graph from the Web is elementary with [SPARQL/Update](#).

```
rdf.update [[LOAD <http://www.strixDB.com/datas/animals.rdf>
INTO <http://www.strixDB.com/datas/animals.rdf>]]
```

Custom import

To create graphs from web resources, we will use the Lua module *webGet*. The following code retrieve RDF datas from the net, then use the Lua function *rdf.graph.create* to create the graph.

```
assert( require 'webGet' )
Client = webGet.new{cache=true}
local uri = 'http://www.strixdb.com/samples/animals.rdf'
rdf.graph.create(uri,Client:GET(uri) )
```

NOTE : this Lua module has also a SAX parser and a XML/RDF parser for convenience. The next code parse the file and print start tags.

```
Client:expandNS(true)
Client:GETXML('http://www.w3.org/RDF/'
function(tag,attrs,depth)print(depth,tag)end,
null, null )
```

The next code print RDF triples :

```
Client:GETtriples('http://www.strixdb.com/samples/animals.rdf',print)
```

Browsing graphs in database

To get the list of all graphs stored in database, just use the function *rdf.graph.list* returning a Lua array describing the graphs. The following code print the list of all graphs

```
table.foreach(rdf.graph.list(),
function(_,v)table.foreach(v,print)print()end )
```

Each graph has a timestamp : the unix time *_t* when the graph was updated for the last time. If you proceed as above, you get 3 graphs :

- the default graph (empty) with uri = <http://example/>
 - the foaf gaph with uri = http://example/foaf_sample0 Specifying a relative uri at import (or for any function) will use the default graph as URI base.
 - the 'relationship' graph.
-

Exporting RDF graphs

Exporting graphs is straightforward. For example, the next command exports '[animals.rdf](#)' now in the database in N-triples format.

```
rdf.graph.export {uri='http://www.strixdb.com/datas/animals.rdf'  
  ,file='animals.nt'}
```

Exporting the same graph in RDF/XML with namespace abbreviations (compact attribute):

```
rdf.graph.export {uri='http://www.strixdb.com/datas/animals.rdf'  
  ,file='MyAnimals.rdf',compact=2}
```

Without file specified, the export function use the Lua output (here stdout) and the Turtle format :

```
rdf.graph.export{uri='http://www.strixdb.com/datas/animals.rdf'}
```

SPARQL queries

SPARQL queries don't need a lot of explanations. The *rdf.sparql* function print the results in columns if the second argument is the string 'print' :

```
local query=[[PREFIX foaf: <http://xmlns.com/foaf/0.1/>  
SELECT ?name ?mbox  
FROM <foaf_sample1>  
WHERE {?x foaf:name ?name . ?x foaf:mbox ?mbox} ]]  
  
rdf.sparql(query,'print')
```

Without the 'print', *rdf.sparql* create a Lua iterator with the results. Next code show how to iterate into the results of a query :

```
for name,mbox in rdf.sparql(query) do
  print(name..' -> '..mbox)
end
```

Explaining queries

To show the virtual machine byte code of the compiled SPARQL query, just use the 'explain' option:

```
rdf.sparql(query,'explain')
0      CLEAR SPO
1      PRED =          <-->
2      CURSOR (P)      call 4
3      RETURN

4      ?x =      SUBJ
5      ?name =   OBJ
6      CLEAR SPO
7      PRED =          <-->
8      SUBJ =          ?x
9      CURSOR (SP)     call 11
10     RETURN

11     ?mbox =   OBJ
12     ROW_INSERT
13     CLEARSTACK
14     RETURN
```

Creating Rules

We will at first create some simple rules : the classical recursive parent/ancestor rules. In 'pure datalog', these rules are :

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
```

IMPORTANT : Datalog is not Prolog. In Datalog, the order of rules declaration has no effect. There is also no cut.

In **StrixDB**, these 2 rules are expressed in a SPAQRL like syntax. We will use the [relationship](#) namespace for the **parent** predicate and introduce a new predicate **ancestor**.

Create a new file named *transitive.log* with the above content :

```
@prefix rel: <http://www.perceive.net/schemas/relationship/>.
@prefix : <http://www.strixdb.com/2010/relationShip/>.
{ ?x rel:parentOf ?y }=> { ?x :ancestorOf ?y }.
{?x rel:parentOf ?z. ?z :ancestorOf ?y }=> { ?x :ancestorOf ?y }.
```

Now, we could load this rules into the database.

```
rdf.rules.import(uri='http://example/rules1/',
  file='transitive.rlog' }
```

For people with a Prolog or Datalog background, it is also possible to write rules as Horn Clauses (Datalog Rules):

```
rdf.rules.create(uri='http://example/rules2',
  datas=[[
  @prefix rel: <http://www.perceive.net/schemas/relationship/>.
  @prefix : <http://www.strixdb.com/2010/relationShip/>.
  ancestorOf(?X,?Y) :- parentOf(?X,?Y).
  ancestorOf(?X,?Y) :- parentOf(?X,?Z),ancestorOf(?Z,?Y).
  ]])
```

Using Rules

In pure Datalog, querying for ancestors is made as follow :

```
ancestor(X,Y) ?
```

In **StrixDB**, the rules are applied with SPARQL queries using the special key word WITH RULES. The next code illustrate a SPARQL SELECT for ancestor.

```
rdf.sparql( [[
PREFIX : <http://www.strixdb.com/2010/relationShip/>
SELECT ?X ?Y
FROM <foaf_sample2>
WITH RULES <http://example/rules1/>
WHERE {?X :ancestor ?Y}  ]], 'print')
```

Graphs and namespaces make the query a little more complicated. But all the power of SPARQL (OPTIONAL, UNION, FILTER and GRAPH) could be used. The next (simple) statement shows a mixed use of SPARQL and Datalog :

```
rdflib.sparql( [[PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX : <http://www.strixdb.com/2010/relationship/>
SELECT ?Xname ?Yname
FROM <foaf_sample2>
WITH RULES <http://example/rules1/>
WHERE {?X :ancestor ?Y.
      ?X foaf:name ?Xname.
      ?Y foaf:name ?Yname
} ]], 'print')
```

Launching a SPARQL server

If not already done, install an Apache HTTP Server (httpd version **2.2**).

Download a windows installer from <http://httpd.apache.org/download.cgi>

For newbie with Apache httpd, you have to modify the Apache server configuration file. The default configuration file with standard installation is:

<C:/Program Files/Apache Software Foundation/Apache2.2/conf/httpd.conf>.

Verify (and change if needed) the following parameters **Listen**, **ServerName** and **DocumentRoot**.

About Apache configuration for RDF, see more in W3C [Configuring Apache HTTP Server for RDFS/OWL Ontologies Cookbook](#)

Configuration of Apache httpd

Copy the file **mod_strixdb.so** of StrixDB distribution into the modules folder of Apache Server (with standard installation, this folder is *C:\Program Files\Apache Software Foundation\Apache2.2\modules*)

Now you have to add some entries to *httpd.conf*.

```
LoadModule strixdb_module modules/mod_strixdb.so
```

```
StrixRoot "C:/Program Files/StrixDB/"
StrixFilename "D:/RDF/strix.db"
StrixDefaultURI "http://mydefault/graph/uri/"

<Location /strixdb>
    SetHandler strix-db-handler
</Location>
```

Explanations : **LoadModule** says to Apache that we want to use StrixDB, **StrixRoot** refers to StrixDB installation folder, **StrixFilename** is the file used by our RDF store, **StrixDefaultURI** is the default graph URI.

Testing SPARQL requests

Launch Apache httpd server. Within a DOS console, run the following command :

```
httpd -k stop
httpd -k start
```

Now, Apache httpd is running as a Windows service. Copy the file [sparql.html](#) from the StrixDB distribution into your **DocumentRoot** folder specified in *httpd.conf*.

With a Web browser get the page **http://<ServerName>/sparql.html** . You could now :

- get the list of graphs stored
- download and upload RDF graphs into the RDF store (in XML or turtle format)
- make SPARQL and SPARQL/Update HTTP requests (with the SPARQL protocol)

Note: We recommend the use of Firefox with [The Tabulator Extension](#) to display RDF nicely.

Dynamic Lua Pages

With StrixDB, Lua can be used as a nice alternative to php. We propose 2 uses of Lua:

- HTTP response created with a Lua script using *print* function.
- file on server using embedded Lua code (code is put inside `<?lua ?>` tags).

First we have to configure Apache *httpd.conf* :

```
AddHandler strix-luapage-handler .hlua  
AddHandler strix-lua-handler .lua
```

Explanations:

The **strix-luapage-handler** handler associate files with a *.hlua* extension with the embedded Lua code execution.

The **strix-lua-handler** handler associate *.lua* extension with the script generation. You could change the extensions as needed. These association are only available into the specified Directory (here */scripts*).

To test, just copy the folder *tests/scripts* from the StrixDB distribution into your **DocumentRoot** folder specified in *httpd.conf*.

Embedded Lua code

Lua code must be included inside `<?lua ?>` . See [test.hlua](#) for an example. The MIME type of the file is set by the Lua function *apache.setContent*

```
apache.setContent('text/html')
```

In the sample [testCreatedRdf.hlua](#), the content type is set to **application/rdf+xml; charset=utf-8** so that the browser will interpret the response as RDF/XML.

Script generated responses

In this case, all the file is generated by the Lua script. The sample [test.lua](#) shows how to generate an HTML file. The sample [testCreateRdf.lua](#) shows how to generate a RDF/turtle file.

As for embedded code, the Lua function *apache.setContent* set the MIME type of the response.

More information in relevant [documentation](#)