Tabulator: Developer notes

These notes are for people developing websites using the tabulator library, or developing the Tabulator itself.

Check out the source code for the Tabulator Add-on

The tabulator is open source software, managed with Mercurial, (abbreviated to hg), a decentralized source code management system. The root of the tabulator project is `http://dig.csail.mit.edu/hg/tabulator` So make sure you have a fairly recent version of Firefox, install Mercurial, then, at the command line prompt:

```
hg clone http://dig.csail.mit.edu/hg/tabulator
```

Then, to set up your checked out code as a Firefox extension,

1. In Firefox, uninstall the Tabulator extension if you have it installed normally.
2. Quit Firefox.
3. Find your Firefox extension directory. (see Where to find your FireFox profile folder)
4. Go into the `extensions` subdirectory
5. Put a file named `tabulator@csail.mit.edu` to that firefox extensions directory. The `tabulator@csail.mit.edu` file must contain a single line which is the absolute path of the location in your computer of the directory into which you checked out Tabulator using hg above.

```
# For example, on Mac OS X:
cd ~/Library/Application?Support/Firefox/Profiles/*/extensions
#
# Assuming the tabulator has been checked out in /Users/me/tabulator
ls -d ~/tabulator > "tabulator@csail.mit.edu"
```

6. Restart Firefox.

That's it.

To update to future versions, just quite Firefox, `cd tabulator; hg pull; hg update` then restart Firefox.

Programming with RDF on the Web: APIs

Below we take you through one example application of the AJAR library. Other applications you could also look at:

- A list of people in SPB's FOAF file

Example application: Address Book

One way to start using the AJAR library is to look at a simple demo application. (Run it with Firefox set up as per tabulator)

- A trivial FOAF address book using AJAR
- The javascript book.js
- The stylesheet card.css

This just loads a FOAF file into the store, finds the friends, and loads theirs, displaying business cards for each.

It may also be useful to look at the interactive test pages which are quite simple in some cases. The notes which follow introduce the key features of the API. David Brondsema has also made some notes on how to use the RDF parser

Namespace shortcuts

The system uses Javascript objects for RDF symbols. Generating them for predicates is simplified by using a namespace function:

```
FOAF = Namespace("http://xmlns.com/foaf/0.1/")
RDF = Namespace("http://www.w3.org/1999/02/22-rdf-syntax-ns#")
RDFS = Namespace("http://www.w3.org/2000/01/rdf-schema#")
OWL = Namespace("http://www.w3.org/2002/07/owl#")
DC = Namespace("http://purl.org/dc/elements/1.1/")
RSS = Namespace("http://purl.org/rss/1.0/")
XSD = Namespace("http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/#dt-")
CONTACT = Namespace("http://www.w3.org/2000/10/swap/pim/contact#")

var foafKnows = FOAF('knows');
```

The store: IndexedFormula

This is the key class in the API. A formula is a set of statements, as you get from parsing and RDF resource. It is what is sometimes called a store, and sometimes called a graph.

To make a new one,

```
var kb = new $rdf.IndexedFormula()
```

To add data to it from a resource

```
kb.load('http://www.w3.org/People/Berners-Lee/card')
```

It will automatically add any ontologies used by the data.

To use data in the store

There are two ways to look at RDF data in a formula. You can synchronously use the the(), each() and any() methods and statementsMatching(), or you can do a query which returns results asynchronously.

The *the(), each(), any()* and *statementsMatching()* take a pattern of subject, predate, object and source ('why'), where for the() each() and any() one of s p and o are undefined and source may be undefined or not. For example, using kb.sym() to make an object for an RDF node (symbol),

```
me = kb.sym('http://www.w3.org/People/Berners-Lee/card#i');
knows = FOAF('knows')
friend = kb.any(me, knows, undefined)  // Any one person
```

or for that matter

```
me = kb.sym('http://www.w3.org/People/Berners-Lee/card#i');
friend = kb.any(me, FOAF('friend'))
```

One of the partners of the triple is set undefined and that serves as a wildcard. In this case, the formula returns the object of a matching triple.

```
m = kb.the(me, mother)
```

Here the assumption (unchecked in the application) is that there is only one match.

```
friends = kb.each(me, FOAF('knows'))
for (var i=0; i<friends.length;i++) {
    friend = friends[i];
    ...
}
```

This returns a javascript array of objects.

Adding data

The *add(s, p, o, why)* method allows a statement to be added to a formula. The optional *why* argument can be used to keep track of which resource was the source for each triple.

```
kb.add(me, FOAF('knows'), fred);
kb.add(me, FOAF('name'), "Albert Bloggs");
```

Note above where the rDF thing in question is a literal, you can just pass a Javascript literal and a string, boolean, or numeric literal of the appropriate datatype will be used.

The *the(), each(), any()* and *statementsMatching()* in fact can take a fourth argument to select data from a particular source.

```
myFoafFile = kb.sym(uri)
friendsInMyFOAF = kb.each(me, FOAF('knows'), undefined, myFoafFile)
```

RDF term types

These are types of node in the RDF graph. Why call them terms? They are terms in the language when yo think of RDF as a language.

What	To create one	x.termType
Node identified by a URI	x = kb.sym(uri)	'symbol'
Blank node	x = kb.bnode()	'bnode'
Untyped Literal	x = kb.literal('foo')	'literal'
Typed Literal	x = kb.literal('foo', undefined, dt)	'literal'
Literal with language	x = kb.literal('chat', 'en')	'literal'
Ordered list	x = kb.list([node1, node2])	'collection'

Asynchronous query

To make a query, in brief, the query is made as a separate formula, introducing variable objects in appropriate places. The query formula is then used a an argument to the query() method of the knowledge base formula being searched. A callback routine also passed to kb.query(). The callback routine is called back when each matching set of bindings is found with bindings as argument.

This is not described in detail here.

Adding views

The view system is quote modular. A form allows you to experiment adding new views to a running tabulator. Click on "New view..."

@@@@ To be continued...

Adding Panes

Panes are different views for displaying data about a particular thing. It can be application-specfic views, typically for things of a certian type or with certain properties. A pane allows you to render, for example, a document in a particular format, or add simple applications which are related to a type of thing.

The following is a partial list of fully functional panes available with Tabulator at the moment.

- Default pane - nested outline view
- Under the hood pane - similar but data the user should not normally worry about
- RDF/XML pane - for a data document, the document re-serilized as RDF/XML
- N3 pane - for a data document, an RDF document re-serialized as N3
- Data visualization pane - for a data document, the data from it as nested tables
- airPane: AIR justification pane - Viewing justifications (explations of the results of inference) from the TAMI project. (doc)

If you want to extend the functionality of Tabulator by adding a new **pane**, just follow these few simple steps:

1. Create the javascript code for the pane in the `js/panes` folder. We recommend naming your file in this format: "`[New]Pane.js`". The skeleton of the pane code should consist of the following:

```
tabulator.panes.register( {

    icon: Icon.src.icon_NewPane, //This icon should be defined in 'test.js' and 'tabulate.js'

    label: function(subject) {   //'subject' is the source of the document
        if the criteria for displaying the pane is satisfied
                return "the string that would be the title for the icon"
        else
                return null;
        },

        render: function(subject, document) {       //'subject' is the source of the document
                                                    //'document' is the HTML document element we are attaching ele

            var div = myDocument.createElement("div")

            //implement what you want to do with the pane here

            return div

        }
    } , true);
```

   icon
      The small image which is used to represent the pane in the bar. Note at the moment these have to be separately defined in both the online and the add-on version.
   label
      This has two functions. It is a test, given a subject, as to whether the pane is appopriate for viewing that object. If this is true, it returns a string to be used as a tooltip for the icon. If it is not true, it returns undefined.
   render
      This function constructs and returns a part of the DOM tree, typically a div, which is the content of the pane.
2. You will probably have to add new CSS to the tabbedtab.css file.
3. Include the newly added pane file in the `tabulator.xul` for the Tabulator to pick up. For the online version, include it in tab.html

Obviously look at exsiting panes for ideas.

.

Getting the old AJAR library archived

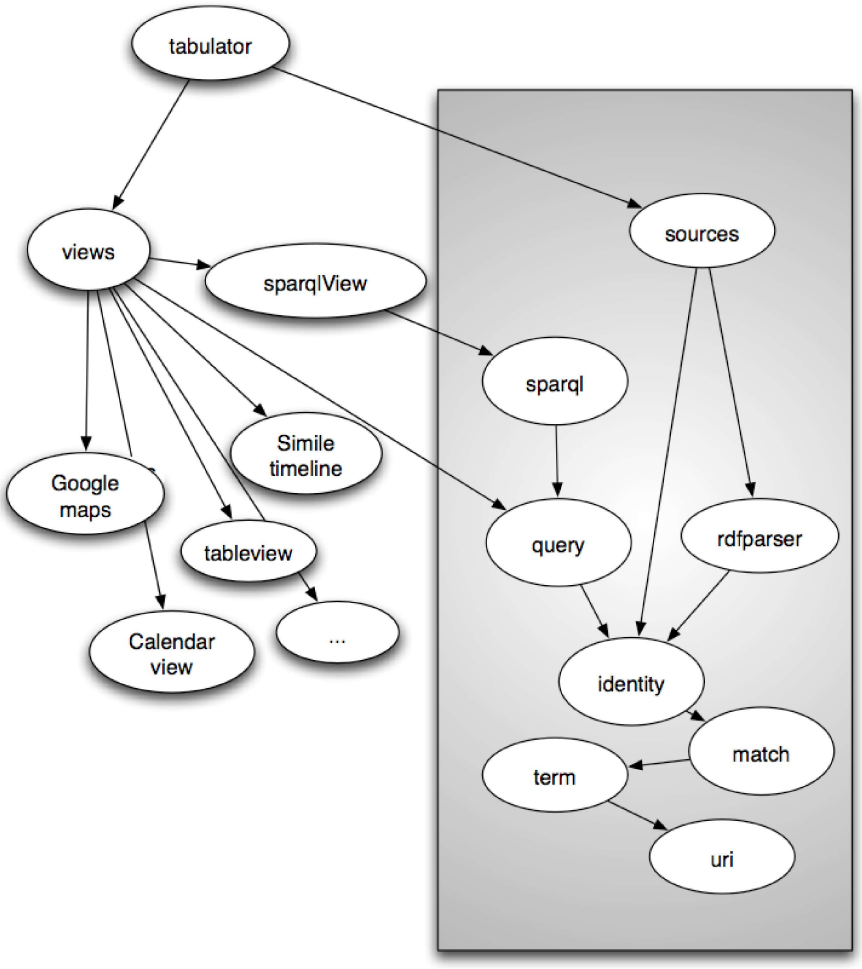
If you check out the tabulator source code as above, then you will find a general-purpose RDF library at `chrome/content/js/rdf/rdflib.js`.

Also and old The 0.8 release of tabulator as a whole is available as an archive file

You can browse the files of this version in the web.

tab.html
   The tabulator file itself - view source on it.
tabulate.js
   Top level script.
rdf/
   The AJAR library of RDF files.
test/
   The tests
doc/api/
   The documentation (incomplete).

The Tabulator RDF library for Javascript



The diagram is a simplified dependency graph of the major modules in the tabulator and the AJAR library.

Files

This library provides data handling for AJAX applications, including web access, store, and query. The modules are as follows, in increasing order of functionality. Each depends on the ones before it.

uri.js

This provides basic URI handling functions, such as calculation of absolute URIs from relative URIs. A URI test page has red blocks if the tests fail.

rdf/term.js

Objects for representing the basic RDF terms. Symbols (nodes in the graph names by URIs), blank nodes and literal values are defined, as are collections.

Statements are defined to have a subject, predicate and object, all terms. They also keep a "why" slot which is used to track the provenance of the statement.

So are formulas, sets of triples. This is a simplistic implementation of formula, RDFFormula, for testing and learning.

(The Variable class of term is not introduced here, but is added by query.js.)

rdf/match.js

This module adds comparison between terms (there is a consistent ordering across all terms) an the simple searching for a statement pattern in a formula.

rdf/rdfparser.js

This is used by sources.js to parse RDF/XML files from the DOM

See also the test/rdf/ parser test files.

rdf/n3parser.js

The N3 parser parses data in Notation 3 from a the notation3 source text. N3 is an equivalent, easier to read, syntax for RDF.