

Abstract

The main goal of the **CLT Cloud project** is to equip lexica, morphological processors, parsers and other software components developed within CLT (Centre of Language Technology) with so called web API:s, thus making them available on the Internet in the form of web services. Our proof-of-concept implementation deals with the task of **composing and aggregating existing web services into new web services**, and with the problem of doing this in a declarative and flexible manner that encourages creative exploration and rapid prototyping of LT applications. Requirements such as declarativity, compositionality, security and a principled approach to the handling of ambiguity made us choose a declarative subset of **Prolog** for this purpose.

Prolog

Prolog has a number of traits that distinguishes it from most other programming languages:

- ▶ **Declarativeness**, inherited from its roots in logic
- ▶ **Inference capabilities**, inherited from its roots in theorem proving
- ▶ **Reflexivity**: no separation between data and programs
- ▶ **Nondeterminism**: a lazy a-tuple-at-a-time generation of solutions to queries, binding variables to different values on backtracking

A RESTful Prolog API

- ▶ Web services in the CLT Cloud follows the **REST** (Representational State Transfer) architectural principles.
- ▶ A client may **PUT**, **POST** or **DELETE** Prolog programs, and **GET** may be used for querying.

Is this safe?

- ▶ We carefully inspect queries and programs before allowing them to be executed on the server machine. Prolog's reflexivity makes this relatively easy.

Impedance mismatch between HTTP GET and Prolog

- ▶ Prolog is **relational**, but HTTP GET is essentially **functional**.
- ▶ To deal with the mismatch we use a **virtual index** to identify solutions, which makes a request for the *i*th solution to a query functional and deterministic.

Is this efficient and does it scale?

- ▶ To avoid regenerating solutions 0-*i* when asking for solution *i*+1, we preserve the Prolog state (stack, choice points, variable bindings) between requests by creating a thread that is associated to the HTTP session, run the state-full computation there, and send messages back and forth between the HTTP handlers and the session thread to communicate queries and results (Wielemaker et al., 2011).
- ▶ We have tried our approach with texts containing hundreds of thousands of words, making Prolog return analyses to the client sentence-by-sentence.

From client to server and back to client: A tiny example

Suppose we want to part-of-speech tag *Tom runs* from inside a browser client using the resources provided by the CLT Cloud. We start by uploading the data to the server, using the browser's XMLHttpRequest API like so:

```
var xhr = new XMLHttpRequest();
xhr.open("PUT", "cloud.clt.gu.se");
xhr.send("text('Tom runs').");
```

The data is stored in a scratch module associated with the current session and used in subsequent queries. Triggered by the JSON response to the PUT, we may now send the query that will part-of-speech tag the data:

```
var q="text(Text), tokens(Text, Tokens), tags(Tokens, Tags) ";
xhr.open("GET", "cloud.clt.gu.se?query="+q+"&cursor=0");
xhr.send(null);
```

Here, *q* is the Prolog query that we want to use, and the value of *cursor* points to the first solution in the zero-based virtual index of solutions.

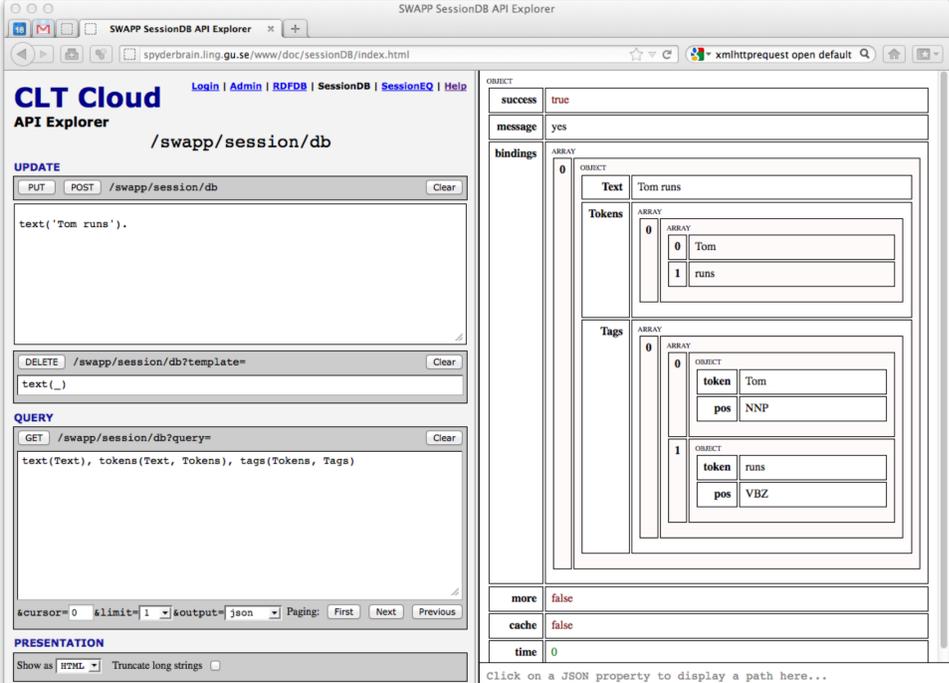
On the server side, the query is run in the scratch module containing the data that was uploaded in the previous PUT request. Note that answering the query usually involves making calls to non-Prolog processes in the cloud.

If the query is successful, the server answers with a JSON object containing fields corresponding to the query's logic variables:

```
{"success": true,
 "message": "yes",
 "bindings": [
   {"Text": "Tom runs",
    "Tokens": ["Tom", "runs"],
    "Tags": [{"token": "Tom", "pos": "NNP"},
             {"token": "runs", "pos": "VBZ"}]}
 ]
}
```

Should we want to retrieve the next solution to the query, we repeat the GET request, but this time with *cursor* set to 1. (In this example, this will probably return a JSON structure signifying failure.)

On the client side, we could for example use a combination of HTML, CSS and JavaScript to present the JSON in a more user-friendly table-based format. This is what we do in our browser-based **CLT Cloud API Explorer**:



The screenshot shows the CLT Cloud API Explorer interface. On the left, there are three sections: UPDATE (PUT /swapp/session/db), DELETE (/swapp/session/db?template=), and QUERY (GET /swapp/session/db?query=). The QUERY section contains the query: `text(Text), tokens(Text, Tokens), tags(Tokens, Tags)`. On the right, the JSON response is displayed in a table format. The response is an array of objects, with the first object containing the following data:

Index	Text	Tokens	Tags
0	Tom runs	["Tom", "runs"]	[{"token": "Tom", "pos": "NNP"}, {"token": "runs", "pos": "VBZ"}]

References

- Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager (2011) SWI-Prolog. Theory and Practice of Logic Programming: Special Issue on Prolog Systems, 12:67-96.