SplineAPI: A REST API for NLP Services

Nuno Vieira[1], Alberto Simões[1,2(✉)], and Nuno Ramos Carvalho[1]

[1] Centro Algoritmi, Universidade do Minho, Braga, Portugal
nunovieira220@gmail.com, ambs@ilch.uminho.pt, narcarvalho@di.uminho.pt
[2] Centro de Estudos Humanísticos, Universidade do Minho, Braga, Portugal

Abstract. Modern applications often use *Natural Language Processing (NLP)* techniques and algorithms to provide sets of rich features. Researchers, who come up with these algorithms, often implement them for case studies, evaluation or as proof of concepts. These implementations are, in most cases, freely available for download and use.

Nevertheless, these implementations do not comprise final software packages, with extensive installation instructions and detailed usage guides. Most lack a proper installation mechanism and library dependency tracking. The programming interfaces are, usually, limited to their usage through command line, or with just a few programming languages support.

To overcome these shortcomings, this work aims to develop a new web platform to make available a set of common operations to third party applications that can be used to quickly access *NLP* based processes. Of course this platform still relies on the same tools mentioned before, as a base support to specific requests. Nevertheless, the end user will not need to install and learn their specific Application Programming Interfaces (API). For this to be possible, the architectural solution is to implement a *RESTful API* that hides all the tool details in a simple API that is common or, at least, coherent, across the different tools.

Keywords: Natural language processing · REST API · Web service · DSL

1 Introduction

Natural Language Processing (NLP) techniques are being used in very different types of applications.

Some companies are mining social communities to find out what their customers think about their products or services [3]. Others are making their information available in different languages by using machine translation techniques [9]. Newspapers and other news agencies, are using NLP techniques to summarise news and cluster them by specific areas, or based on their similarities [5].

Any one of these applications require a stack of NLP tools to work. This stack can be very different from tool to tool, but might include common tasks like: language identification, text segmentation, sentence tokenization, part of speech

© Springer International Publishing Switzerland 2015
J.-L. Sierra-Rodríguez et al. (Eds.): SLATE 2015, CCIS 563, pp. 205–215, 2015.
DOI: 10.1007/978-3-319-27653-3_20

tagging, dependency parsing, probabilistic translation, dictionaries querying, or named entity detection, just to mention some [6].

Although there are some NLP toolkits that include a good number of tools for most of these tasks [1,4], developers are likely to need other tools that are not directly available. This leads to the installation of different tools. If the developers need to support a wide range of languages, this list of tools is prone to grow, as some tools are not language independent or because they do not include training data for some of the required languages.

These requirements lead to the need of installing a variety of tools to have a complete NLP stack. Unfortunately, most of these installations are not as simple as they should be, as most of their developers are more interested in using the tools and adding new features than to document their usage and installation, or to provide good installation procedures. This leads to the need of dealing with different kinds of installation problems, and to learn each tool application programming interface (API).

Although our NLP team is small, we have been dealing with this problem for some time, and therefore, we are proposing a tool and a service to hide all these details from the end-user, making these libraries available as web services based in the REST philosophy. Of course that, if the web services are, themselves, using those same tools, someone will need to deal with the installation procedure, and will need to learn its usage. But if this process could be done only once, and the installed tools are available as a simple web service, application development is faster, and application deployment gets easier.

As a side benefit, having a different server running some tools, helps in distribution. Even if at the moment we have the system working on a single server, it is simple to distribute the tools between different machines.

Nevertheless, the process of making these tools available through a web service is not straightforward, as one needs to deal with timely processes, that can not be served easily using a single HTTP request, given timeouts; problems on service abuse; problems on load distribution, and others.

In this paper we present SplineAPI, that is both a service, that we are making available for free, and a platform, for anyone to replicate this kind of service in their own servers. Section 2 will compare our proposal with other services already available on the *Web*. Section 3 includes a presentation of our design goals as well as the SplineAPI architecture and implementation. Section 4 concludes with future work.

2 Related Work

The idea to make APIs available through web services is not new. There are several platforms that make NLP processes available online, each with its own characteristics and targeting different kinds of users. They range from simple tools that allow a single kind of task to be performed, to fully featured sites with a diverse set of functionalities.

In this section we compare our main goals with some of the tools already available. We focused mainly on tools that have more similarities with our approach. Therefore, we are looking mainly to tools that include more than one kind of task and targeting more than one type of user. Then, we looked up their popularity.

The main differences from the analysed platforms and our main goals are:

– some of the platforms are not NLP specific, like Mashape. They just work like a proxy that hides some of the web-services requirements (like user authentication and quota management). Nevertheless, there is no information about how the real service is implemented, and if its architecture is generic enough to be configured for other requirements;
– other platforms, like Text-Processing, although allow different types of services, all of them are based on one single tool (in this case, NLTK). Again, no information is given on the system implementation and how it can be adapted to other tools, and in specific, for functionalities not available in NLTK.
– and finally, mono-application services. Some are available together in a similar place, like CORE API by TextAlytics but there is no integration or homogeneity between the different offered services.

During the development of SplineAPI our main goal is to have an extensive system, to be used by anyone interested in offering Web Services, that can be easily configured and monitored.

3 Design Goals and Architecture Details

The main goal is to create a solution that minimizes the challenges developers face, when trying to take advantage from a large set of NLP tools already available.

In today's connected world, applications are no longer running only on the client machine. Also, they are no longer running only server-side. They are distributed, both on the client machine, server machine and others that might help in the process.

Therefore, our goal is to help the conversion of NLP tools into web services. Although the tool installation may be a challenge, the administrator of these services needs to deal with it, we intend to make the API construction easy, recurring to a set of Domain Specific Languages (DSL).

With the idea of creating a web API, it was necessary to think what is the best implementable architecture to develop this idea. The easiest and the cleanest method, to make available all the NLP tools, is to build a web service. Inside the web service *world*, there are various options of architectures, depending on how do we want to provide the service. The most popular are: Simple Object Access Protocol (SOAP) and Representational State Transfer (REST), each one with its own advantages and disadvantages depending on the objective in mind. When it comes to SplineAPI, the obvious choice was REST [2,8].

REST is more and more popular, and the best benefit it offers, is the optimization for stateless interactions that, in this case, is an essential feature, because the platform handles specific requests and responses based on text data, and that, does not require a connection status. To the users, REST is the simplest way to query a service because it is less verbose and easy to understand, as it bases its interaction with the clients in well known HTTP commands.

With the platform's architecture decided, it was then fundamental to investigate the best way of developing all the connections between the tools and the service, and the software technologies needed to make everything work.

3.1 Spline Architecture

Figure 1 shows our solution architecture. The server is composed of three main components: the Spline REST server, the NLP tools and their interface definitions, and a quota database.

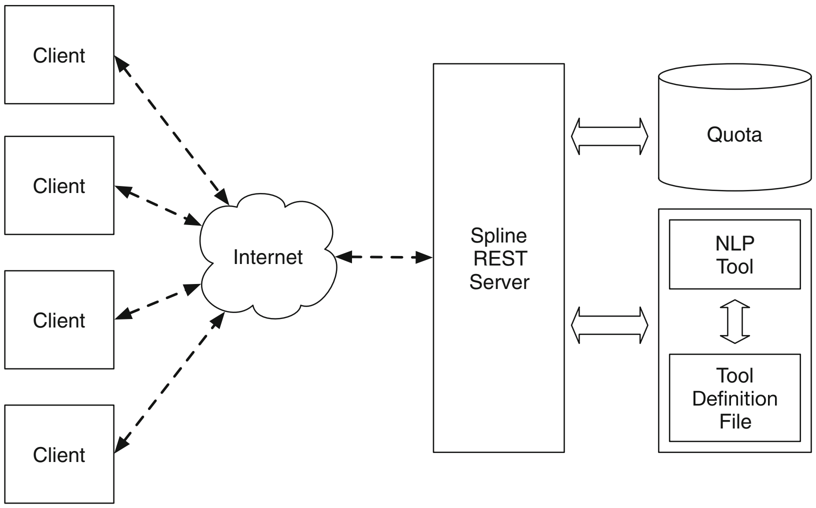

Fig. 1. Spline architecture.

NLP Tools and Definition Files. Different NLP tools communicate in different ways with the user. Some tools are command line applications that read information from a file, or from the standard input, and produce results in another file, or into the standard output. Some other are library-based, meaning that they expose an API that can be used in order to process information and obtain a desired output.

In order to be able to tackle with these different aspects of tools, each tool interface is described in an XML file.

This XML file is processed and a Perl module is created. This Perl module is responsible for the interaction with the Spline REST server, as is detailed in Sect. 3.2.

Listing 1.1. XML example for the Tokenization Service based on FreeLing Perl library.

```
<service>
  <meta batch="false">
    <tool>FreeLing</tool>
        <name>Tokenizer</name>
        <route>tokenizer</route>
        <parameters>
          <parameter required="true" name="text" type="textarea">
            <description>The text to be tokenized</description>
          </parameter>
        </parameters>
        <definition>Process of breaking a stream of text up into tokens
            .</definition>
        <cost>1</cost>
  </meta>
  <implementation>
        <packages>
          <package>FL3 'pt'</package>
        </packages>
        <main lang="perl">
          my $pt_tok = Lingua::FreeLing3::Tokenizer->new("pt");
          my $tokens = $pt_tok->tokenize($text, to_text => 1);
          return $tokens;
        </main>
  </implementation
  <tests>
        <test>
          <param name="text">I will be tokenized.</param>
          <code>
            ok($result ->[0] eq 'I', "Test␣the␣first␣word");
          </code>
          <code>
            ok((scalar @{$result}) == 5, "Test␣the␣result␣length");
          </code>
        </test>
  </tests>
  <documentation>
    <header title="module">Spline::FreeLing::Tokenizer - a module that
         tokenizes your text.</header>
  </documentation>
</service>
```

The XML structure follows a proper XML Schema that allows the validation of the XML file. It also defines the domain of specific elements and attributes, which allow easy verification on the XML semantics.

Listing 1.1 presents an example of an XML definition file. It describes the interface for a tokenization service based on FreeLing [7] library.

The XML file is composed by three main parts:

– The meta-data for the service includes its name, the back-end tool and the service route (basically, the path used for the service URL). It also includes a brief explanation of the service goals, the service usage cost (if applicable) as well as which parameters should be used in order to request an operation. Each parameter is described in terms of its name, requiredness, data type (text, number, file and others) and default values. Note that, when a service receives file parameters, it request needs to be performed using the POST HTTP method, with multipart form data.

The file also includes documentation, adding a brief explanation of each parameter meaning.
- A description of how the parameters supplied by the users will be used to compute a result. At the moment this is done using Perl code or Bash commands. In the first case, there are two sections, one describing the Perl packages that need to be loaded, and another with the code that is executed. For Bash commands, only the executed code section should be used.
  We are aware that for different tools our generator will have different needs, and therefore this section of the XML definition file might need further options in the future.
- It includes a set of tests that allow the service programmer or the server administrator to test if all services are working properly. These tests include an input for the service and a set of assertions over the obtained output. Again, at the moment these tests are being written directly in Perl, but we have been working into incorporate a JSON querying language like JsonPath[1] or JSONiq[2].
- Finally, the file adds the possibility to document the service as a module. It divides the information by headers (like chapters) and each one has its own proper description. It is simply a way to maintain the system's organization as well as explain everything in more detail.

The structure of the Perl module generated from these XML definition files is presented later, in Sect. 3.2.

Quota Database. Although our service is designed to be stateless, meaning that the service is connection-oriented, we want to record information on service usage, in order to track users, most used services, and if possible, distribute different services by different servers, so that highly used services are hosted in different hardware.

In one hand, each service defines how much a request to it costs. This cost can be a constant or defined accordingly with the amount of data to be processed. On the other hand, each client has an amount of quota to be used based on a cost limit. This quota can differ accordingly with the status of the client or, who knows, accordingly with a paid plan. Of course there is also the possibility to turn off quota management completely.

For this to be possible it was created a coin strategy. Each user has a daily limited amount of coins he can use freely. All the functionalities are different in their processing time but have a text-based parameter that can be small or big and, based on that, we stipulated a whole panoply of cost indicators that differ with the length of the text and the functionality itself. For that to happen, it was obviously fundamental to create a stateless authentication process to identify and manage all the users and their requests.

[1] A XPath like language for JSON, available from: http://goessner.net/articles/JsonPath/ (Last visited: 15-04-2015).

[2] A very complete and expresssive query language for JSON, available from: http://www.jsoniq.org/ (Last visited: 15-04-2015).

Spline REST Server. Considering that Perl is a programming language adequate to process textual data, with a great set of interfaces to other programming languages, it was the chosen language for the back-end server implementation.

The server is implemented in Perl, using the Dancer2 Web Framework [10]. The interaction with the NLP tools is done using Perl modules generated automatically from the already mentioned XML Definition Files. These modules are loaded automatically by the server, making all services available.

The server is responsible for querying the quota database and update it accordingly with the user requests. When called using the standard HTTP protocol, it presents common web pages documenting the services that are available (accordingly with the loaded modules) and their interfaces.

This strategy allows the easy creation of new services, just by creating an XML definition file, converting it into a Perl Module (and in some cases, some edition of the generated module) and restarting the web server. The new module will be loaded and its description and documentation will be made available in the website automatically.

3.2 Perl Module Generation

As already mentioned, the XML definition file is processed and "compiled" into a Perl module. The Perl module includes information about the service itself (namely, the `meta` section of the XML definition file) and a set of methods that are used both for configuring the service, and to perform the required operations to provide the service.

The module generation is template based. The meta-information is converted into an associative array (*hash*, in Perl terminology), and the Perl code is embedded in a subroutine.

The generated Perl module can be edited manually, to perform any special tweaks or improvements that might be necessary.

Listing 1.2 shows the relevant portions of the generated Perl module. Each module should implement a programming interface (called Roles, in Perl world), making available functions to access some of the needed data. Some of these functions have default behaviour, and as such, the code generator creates stub functions that can be then edited by the user. This means that the XML description can be used just for the module bootstrap.

The Perl module should also include a main function that will receive the request in a dictionary, and should return an answer as a Perl structure. This structure will be then converted into JSON and sent to the client.

In the Perl community a Perl module is, usually, shipped together with a set of tests. Therefore, the test information available in the XML definition file is used to generate such tests, like the one presented in Listing 1.3.

These tests can be used both for testing the Perl module locally, as well as to test the production service (in order to guarantee all the services are running correctly).

Listing 1.2. Module generated by the XML example.

```
package Spline::FreeLing::Tokenizer;

use FL3 'pt';

my %index_info = (
  hash_token => 'tokenizer',
  parameters => {
    api_token => {
      description => "The token to identify the user",
      required => 1,
      type => 'text',
    },
    text => {
      description => "The text to be tokenized",
      required => 1,
      type => 'textarea',
    },
  },
  description => "Process of breaking a stream of text up into tokens
      .",
  cost => 1,
);

sub get_token { return $index_info{hash_token} }

sub get_info  { return \%index_info }

sub cost_function{
  # return the total cost of the request
  return $index_info{cost};
}

sub param_function {
  # return 0 or 1 depending on the validation of the request
  return 1;
}

sub main_function {
  my ($input_params) = @_;
  my $result = _freeling_tokenizer($input_params);
  my $json = encode_json $result;
  return decode_utf8($json);
}

sub _freeling_tokenizer{
  my ($input_params) = @_;
  my $text = $input_params->{text};
  return unless $text;

  my $pt_tok = Lingua::FreeLing3::Tokenizer->new("pt");
  my $tokens = $pt_tok->tokenize($text, to_text => 1);
  return $tokens;
}

1;
```

Listing 1.3. Tests generated by the XML example.

```
use strict;
use warnings;
use HTTP::Tiny;
use JSON;

use Test::More tests => 2;

my $host = $ENV{SPLINE_HOST} || 'localhost';
my $port = $ENV{SPLINE_PORT} || 8080;

my %params = ();
$params{api_token} = 'a_token';
$params{text} = 'I will be tokenized.';

my $got = HTTP::Tiny->new->post_form("http://".$host.":".$port."/
    tokenizer", \%params);
my $result = decode_json($got->{content});

ok($result->[0] eq 'I', "Test the first word");

ok((scalar @{$result}) == 5, "Test the result length");
```

3.3 Lengthy Requests

The previously presented architecture works great when the processes can be run on the fly. Unfortunately, a lot of Natural Language Processing tasks are slow, that would fire HTTP timeouts easily. Also, if concurrent users try to perform such tasks, the system will overload and be even slower (if not failing at once).

With a big range of lengthy services in the NLP area, a solution to deal with this kind of tasks was needed. It was, then, necessary to delineate a way to close the connection to the user and, after the desired process is complete, return the results.

The solution was the development a system daemon, that processes requests from a queue, and make the results available to the end-user. The algorithm is based on the following outline:

1. The REST server receives the request and detects if it is a lengthy one.
2. For lengthy requests, the server answers with a JSON that includes the URL where the answer, when ready, will reside. At the same time, it creates a task in the daemon queue, and a JSON file, accessible to the user, describing that the task is running.
3. At this point, the first HTTP connection is already closed.
4. When possible, the daemon unqueues the task and executes it, placing the resulting files in the folder created by the REST server for that effect.
5. The end-user will request, periodically, the JSON file, checking if its content changed. If so, check the URL where the results are, and fetch them. This

process might be a problem if the users check for the JSON file changes too often. Nevertheless, a simple GET request should be faster and lighter than having the processes running at the same time.

To differentiate these services from the common ones, the XML file describing the process accepts an extra attribute. This type of modules need to follow the outline above.

To make the daemon work there are four main folders, the first two private, the second two, public:

– The *logs* folder is used by the daemon to save information on each processed request. It allows the administrator to track the daemon activities, and debug them.
– The *queue* folder store files describing each process in the queue. They are organised by time, therefore allowing its use as a queue.
– The *json* folder save the JSON files with the information to be delivered to the user. When the process is running, these files show that the process is not complete. When it ends, its content changes to include the URL for the final resources.
– Finally, the *results* folder will store the output files, that are kept for a couple of days, to allow the user to collect them.

Periodically the *json* and *results* folders are cleaned for too old files.

4 Conclusions

In this document we present the architecture for a module-based server for REST services. The motivation for its development is the need to make NLP related operations available easily, without all the problems that comprise their usual configuration and installation.

Although the whole framework is ready and some services are already available (http://spline.di-um.org/) we are aware that different tools will dictate different problems to manage. In fact, we are already aware of some of the challenges we will face:

– At the moment, the lengthy services code part is not generic and it is mandatory that the admin manage a big part of the process. To improve this issue, it will be added an output section to the XML generation schema. This sections will contain all the URLs to the result files and it will be kept in the Perl Module to use when the final JSON file is created. With this feature, the user knows where to find the desired information before the request is completed (although it maintains a flag that informs the process is still running) and the admin does not have to deal with that.
– Turn the platform even more user friendly. There are some things that can be improved like error responses, interface organisation and styling and some specific features.

Other than these developing challenges we intend to implement in Spline, we will face other problems as soon as the server starts to be widely used, namely computational weight and server balancing.

Acknowledgements. This work has been partly supported by FCT - Fundação para a Ciência e Tecnologia within the Project Scope UID/CEC/00319/2013.

References

1. Cunningham, H., Maynard, D., Bontcheva, K.: Text Processing with GATE. Gateway Press, California (2011)
2. Fielding, R.T.: Representational State Transfer (REST). Ph.D. thesis, University of California, Irvine (2000). https://www.ics.uci.edu/fielding/pubs/dissertation/fielding_dissertation.pdf
3. Liu, B.: Sentiment Analysis: Mining Opinions, Sentiments, and Emotions. Cambridge University Press, New York (2015)
4. Loper, E., Bird, S.: Nltk: the natural language toolkit. In: Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics, ETMTNLP 2002, vol. 1, pp. 63–70. Association for Computational Linguistics (2002)
5. Mani, I., Maybury, M.T.: Advances in Automatic Text Summarization, vol. 293. MIT Press, Cambridge (1999)
6. Martin, J., Jurafsky, D.: Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, 2nd edn. Prentice Hall, Upper Saddle River (2009)
7. Padró, L.: Analizadores multilingües en freeling. Linguamática **3**(2), 13–20 (2011)
8. Pautasso, C., Zimmermann, O., Leymann, F.: Restful web services vs. big'web services: making the right architectural decision. In: Proceedings of the 17th International Conference on World Wide Web, pp. 805–814. ACM (2008)
9. Rychtyckyj, N.: Machine translation for manufacturing: a case study at ford motor-company. In: Proceedings of the 18th Conference on Innovative Applications of Artificial Intelligence, IAAI 2006, vol. 2, pp. 1728–1735. AAAI Press (2006). http://dl.acm.org/citation.cfm?id=1597122.1597130
10. Sukrieh, A.: Dancer2: Manual - A gentle introduction to Dancer2 (2013). http://search.cpan.org/sukria/Dancer2-0.10/lib/Dancer2/Manual.pod