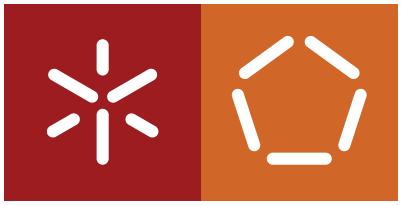
Universidade do Minho
Escola de Engenharia
Departamento de Informática

Nuno Miguel Leal Gonçalves Vieira

SplineAPI, uma API REST para serviços de Processamento de Linguagem Natural

Outubro 2015



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Nuno Miguel Leal Gonçalves Vieira

SplineAPI, uma API REST para serviços de Processamento de Linguagem Natural

Dissertação
Mestrado em Engenharia Informática

Dissertação supervisionada por
Professor Alberto Manuel Brandão Simões

Outubro 2015

Agradecimentos

Antes de mais gostaria de agradecer ao Professor Alberto Simões pela sua disponibilidade e cooperação no que diz respeito à realização desta dissertação assim como tudo o resto que a envolve.

Agradeço também à minha mãe por sempre ter acreditado em mim, nas minhas capacidades e por me ter incutido os valores e as aptidões preponderantes que contribuíram ao meu desenvolvimento pessoal e profissional.

Tenho também um agradecimento especial para todos os meus amigos, colegas e familiares que estiveram presentes quando necessitei deles e que tiveram, de forma direta ou indireta, uma influência nas escolhas e nas decisões que fui tomando ao longo da vida.

Por último, e não menos importante, agradeço a todos os professores e funcionários do Departamento de Informática da Universidade do Minho pela dedicação e conhecimentos transmitidos durante a licenciatura e mestrado que contribuíram quer para a concretização desta dissertação quer para alcançar os meus outros objetivos.

Resumo

Habitualmente os investigadores de determinada área são capazes de criar ferramentas complexas, mas raramente são capazes de as tornar fáceis de usar e de instalar. Isto é igualmente verdade na área de *Processamento de Linguagem Natural* (PLN), o que leva a que muito tempo seja perdido em processos de instalação, atualização e configuração de ferramentas.

Com esta dissertação pretendeu-se estudar a possibilidade de disponibilizar pequenos serviços de *Processamento de Linguagem Natural* usando um serviço web baseado em *Transferência de Estado Representativo* (REST). Este serviço torna acessível uma *Interface de Programação de Aplicações* (API), levando a que qualquer investigador possa usar as ferramentas de forma simples, bastando para isso que a linguagem de programação usada suporte pedidos pelo protocolo *HyperText Transfer Protocol* (HTTP).

Nesse sentido, foi desenvolvido um servidor modular, capaz de fazer o carregamento dinâmico de módulos que descrevam a interface entre ferramentas e o serviço REST. Esta interface pode funcionar como uma biblioteca, um módulo Perl ou mesmo como uma ferramenta de linha de comando. Para facilitar a descrição destes serviços foi desenhado um dialeto XML, que permite descrever meta-informação sobre o serviço mas também a sua assinatura e processo.

Para permitir que processos mais demorados pudessem também ser disponibilizados neste serviço foi implementada uma solução assíncrona.

A ferramenta implementada foi testada, com vários casos práticos, e com diferentes clientes, escritos em várias linguagens de programação.

Além disso, foi publicado um artigo que descreve o sistema e o coloca disponível para a comunidade científica.

Palavras-Chave: PLN, tempo, serviços, web, REST, API, ferramentas, DSL.

Abstract

Usually the researchers of a certain area are capable of creating complex tools but they are rarely capable of making them easy to use and install. This is also true in the *Natural Language Processing* (NLP) area which leads to a lot of time spent in installation, update and tool configuration processes.

With this dissertation it was intended to study the possibility of making some *Natural Language Processing* services available using a web service based on *Representations State Transfer* (REST). This service makes an *Application Programming Interface* (API) acessible and it gives the possibility to every researcher to use the tools in a simple way with only a constraint that the chosen programming language can support *HyperText Transfer Protocol* (HTTP) protocol requests.

With that in mind, it was developed a modular server that allows the dynamic loading of service modules that describe the interface between tools and the REST service. This interface can work as a library, as a Perl module or even as a command line tool. To make the service description easier, it was defined a XML dialect that grants the description of its meta-information as well as its signature and process.

To allow slow processes to be provided on this service too, it was developed an asynchronous solution that makes it simple to manage.

The platform created was tested, with various practical case studies and with diferent types of clients written in different programming languages.

Besides that, it was published an article that describes the system created and makes it available to the cientific community.

Keywords: NLP, time, services, web, REST, API, tools, DSL

Conteúdo

Lista de Figuras

Lista de Tabelas

Acrónimos

AJAX Asynchronous JavaScript And XML. 23

API Application Programming Interface. 4, 23–25, 35, 38, 55, 70

CLEF Cross Language Evaluation Forum. 12

CORBA Common Object Request Broker Architecture. 15, 16

CRUD Create, Read, Update, Delete. 20

DSL Domain Specific Language. 4, 16, 32, 33, 35, 78, 79

FIFO First In First Out. 51

HTTP HyperText Transfer Protocol. 4, 15, 16, 19–24, 31, 50, 52, 53, 72, 78

IDL Interface Description Language. 16

JSON JavaScript Object Notation. 22, 39, 50, 51, 54, 57, 58, 71–74

NLTK Natural Language Toolkit. 25

OMG Object Management Group. 15

ORB Object Request Broker. 16

PDF Portable Document Format. 6

PLN Processamento de Linguagem Natural. 2–6, 11, 23–25, 32, 35

POS Part of Speech. 10

REST Representational State Transfer. 4, 5, 19–25, 31, 32, 35, 38, 52, 78, 79

RPC Remote Procedure Call. 12–18, 23

SMTP Simple Mail Transfer Protocol. 16

SOAP Simple Object Access Protocol. 16–19

SWIG Simplified Wrapper and Interface Generator. 28, 30

TAP Test Anything Protocol. 47, 50, 57

TCP Transport Control Protocol. 15, 21, 78

TCP/IP Transmission Control Protocol/Internet Protocol. 14, 21

UDP User Datagram Protocol. 15

URI Uniform Resource Identifier. 18, 20, 22, 52, 55

URL Uniform Resource Locator. 52, 71, 76

W3C World Wide Web Consortium. 16, 18, 19

WSDL Web Services Description Language. 18

XML Extensible Markup Language. 16–19, 22, 32, 40, 41, 49, 55–57, 67, 69, 70, 79

1 Introdução

Com a evolução do ser humano, a comunicação começou a ser cada vez mais necessária, surgindo assim a linguagem natural. Com o passar dos tempos, com a crescente evolução tecnológica e com a grande importância da comunicação no dia a dia, começaram a surgir campos de estudo que investigam como tirar proveito das máquinas atuais para ajudar o ser humano a evoluir no que diz respeito ao processamento de um número cada vez maior de informação existente.

Com este grande progresso na área, foi surgindo software capaz de resolver variados desafios relacionados com a temática das línguas só que, infelizmente, muitas destas ferramentas tornam-se complexas (a vários níveis) na hora da sua instalação e aprendizagem por parte dos utilizadores. É por esses motivos que aparece a ideia da criação de um serviço web que seja capaz de fornecer o potencial destas funcionalidades de uma forma simples e eficiente tirando proveito das múltiplas tecnologias atuais.

1.1 História

Como qualquer outro animal, os primeiros seres humanos sentiram a necessidade de comunicar entre si. Apesar de ser difícil situar com exatidão o surgimento de algum tipo de linguagem, estudos apontam para que as primeiras linguagens tenham aparecido há cerca de dois milhões de anos, com os nossos ancestrais *homo-habilis*.

Terá sido este o nascimento do que hoje designamos como **linguagem natural** ou seja, uma linguagem originada pelo intelecto humano de uma forma espontânea. Naturalmente que tal como com qualquer outro recurso usado pelo ser humano, ao longo do tempo esta linguagem foi evoluindo. É habitual dizer-se que uma linguagem nasce, cresce (ou desenvolve-se) e, eventualmente, morre.

1.1. História

Comunicar é transmitir uma mensagem ou uma informação, usando um método que é percetível por ambas as partes. O uso de linguagens engloba, então, a comunicação pela fala, pela escrita, por gestos ou por toque e, por muito diferentes que pareçam, todas elas têm aspetos comuns.

Com a evolução das tecnologias, nomeadamente com a disseminação de máquinas capazes de processar dados, e a importância das linguagens no dia a dia, surgiram ramos da ciência e da engenharia, que pretendem tirar partido da automação destas máquinas para que o ser humano possa processar, de forma mais eficiente, os vários suportes em que se usam linguagens. É neste contexto que surge a área de **Processamento de Linguagem Natural (PLN)** [Jurafsky and Martin, 2008].

Segundo Liddy [2001], as primeiras aplicações baseadas no estudo e uso de linguagem natural datam do ano de 1946. O objetivo era, na altura por parte dos Aliados, de decifrar os códigos usados pelos inimigos durante a Segunda Grande Guerra.

As primeiras aplicações em PLN começaram a aparecer nos anos 50. Nesta altura a área estava intrinsecamente ligada à inteligência artificial, cujo principal objetivo consistia em modelar e reproduzir, recorrendo à ajuda de máquinas, a capacidade humana, fosse no seu raciocínio, mas também na capacidade em produzir e compreender enunciados linguísticos com a finalidade de comunicar. Neste sentido, o projeto icónico do início de investigação em linguagens naturais foi o Eliza [Weizenbaum, 1976], um programa computacional que simulava um ser humano com o intuito de manter uma conversa lógica.

Embora tenha sido uma área com uma evolução inicial lenta, com o drástico aumento da capacidade computacional e de armazenamento, a sua evolução foi exponencial, sendo atualmente uma área de grande interesse. Esta diferença de recursos marcou a evolução da área. Inicialmente abordava-se o problema de forma formal, por exemplo, com a tentativa de definição da gramática de uma determinada língua. No final do século passado esta abordagem foi completamente esquecida, tendo-se apostado não na modelação empírica da língua, mas na análise da língua real. Isto só foi possível graças à grande quantidade de documentos que foram surgindo em formato digital, e ao uso de métodos estatísticos para a inferência de propriedades da língua. Mais recentemente, já no início deste século, as abordagens formais voltaram a ser usadas conjuntamente com os métodos estatísticos, em abordagens híbridas [Nadkarni et al., 2011].

Com a grande quantidade de informação que produzimos atualmente em formato digital, tornou-se uma necessidade dispor de aplicações capazes de a processar e tratar eficientemente.

Os métodos ou áreas de conhecimento atualmente usadas em PLN incluem a inteligência artificial, nomeadamente a aprendizagem máquina, as ciências da computação, a lógica, a estatística, a linguística entre outras.

Neste momento, o PLN estuda múltiplos subtemas, desde a tradução automática até à análise de sentimento, da sumarização à recolha de informação, do reconhecimento de voz à geração de texto. Para além das várias áreas relativamente estanques dentro do PLN, têm surgido novas áreas de investigação como a neurociência ou a psicolinguística.

1.2 Motivação e Objectivos

Com a evolução do PLN ao longo dos anos apareceram muitas e variadas ferramentas computacionais com o objetivo de resolver diferentes problemas, desde pequenas tarefas, como a divisão de um texto em frases, até tarefas complexas como a tradução automática em várias línguas.

Infelizmente, um bom investigador em PLN não é, necessariamente, um bom programador. Do mesmo modo, um investigador, mesmo que bom programador, muito provavelmente gastará o seu tempo em tarefas de investigação e não na preparação ou documentação das ferramentas desenvolvidas, para que possam ser usadas facilmente por outros investigadores. Este é um problema que não é específico à área de PLN. No entanto, é nesta área que se pretende intervir.

Isto leva a que nem todas as ferramentas disponíveis sejam tão fáceis de usar como seria desejável, sendo que algumas se tornam difíceis de usar e até de instalar. O problema não é, apenas, a falta de documentação, mas também a falta de cuidado em tornar a interface da ferramenta (seja ela gráfica ou para programação) simples e intuitiva.

Um utilizador de uma dessas ferramentas, mesmo que tenha capacidade para resolver o desafio de instalação e aprendizagem da sua interface, perderá uma grande quantidade de tempo e de motivação ao fazer. O próprio utilizador poderá desistir e optar por implementar a sua própria ferramenta para resolver um problema já resolvido. Esta situação funciona como um freio na evolução da ciência.

Muitas vezes poderá até ser necessário alterar alguns detalhes na aplicação de modo que ela possa funcionar numa nova arquitetura ou sistema operativo. Infelizmente essas alterações, na grande maioria das vezes, não são publicitadas, e diferentes investigadores terão de lidar com

esse mesmo problema. Até é possível que o mesmo investigador tenha de reinstalar a mesma aplicação numa outra máquina (ou na mesma, passado algum tempo) e já não se lembre das alterações necessárias.

São estas as motivações para este trabalho. Pretende-se resolver ou minimizar este problema, criando uma *Application Programming Interface (API)* remota, que permita a diferentes investigadores usar um conjunto de aplicações sem a necessidade de as instalar, bastando para isso realizar um pedido ao serviço.

Pretende-se a criação de uma *API* baseada na filosofia *Representational State Transfer (REST)* [Fielding, 2000], que seja *open source* e que permita aceder a ferramentas específicas usando, para isso, pedidos simples via *HyperText Transfer Protocol (HTTP)*. O objetivo será, então, fornecer funcionalidades de diferentes ferramentas a partir de um simples endereço *web*.

Com esta abordagem todo o processo de instalação e compilação pode ser praticamente ignorado pelo utilizador comum, já que essa ferramenta terá de ser instalada apenas uma vez, pelo administrador do serviço. Também a usabilidade aumenta já que o uso de diferentes serviços se torna transparente sob uma *API* homogénea.

O trabalho tem dois objetivos diferentes:

- definir um modelo para um servidor que disponibilize uma *API REST* sobre ferramentas generalizadas, e implementar um protótipo. Este serviço deverá ser fácil de configurar e estender, por exemplo com o uso de uma *Domain Specific Language (DSL)*, e capaz de lidar com problemas de autenticação e carga computacional;
- disponibilizar sob o servidor implementado no ponto anterior um conjunto base de ferramentas de PLN.

1.3 Estrutura do Documento

Este documento é composto por cinco capítulos:

O primeiro capítulo denomina-se de *Introdução*. Inicialmente, este capítulo aborda a história do Processamento de Linguagem Natural desde o aparecimento das línguas naturais até aos dias de hoje onde as máquinas são capazes de lidar e simular situações onde as línguas naturais são fulcrais. Depois descreve também quais as razões principais para que este tema tenha sido

proposto e quais os objetivos que se tem planeados para o produto final.

O segundo capítulo é o importante *Estado da Arte* que se destaca pela revisão da literatura existente acerca dos assuntos relacionados com o tema da dissertação. Como o objetivo é criar um serviço *web* que disponibiliza as mais variadas funcionalidades investigadas no PLN, este capítulo é composto por quatros secções:

Arquiteturas de Comunicação entre Aplicações
: Como a ideia é construir um serviço *web*, baseado numa arquitetura *REST*, que não é nada mais nada menos uma maneira de comunicar entre uma aplicação cliente (utilizador) e uma aplicação servidor (serviço), então é preciso investigar como surgiram estes mecanismos e os passos evolutivos que foram sofrendo ao longo dos tempos;

Desafios no Processamento de Linguagem Natural
: Esta secção aborda um conjunto de funcionalidades que o PLN investiga e tenta resolver eficazmente. Como a plataforma que se pretende implementar tem como objetivo reunir essas tais ferramentas, convém revelar um pouco qual o intuito e a utilidade destas;

Plataformas Web para Processamento de Linguagem Natural
: Com esta secção pretende-se mostrar as plataformas que já existem no mundo web relacionadas com a temática do PLN. Cada uma destas tem as suas vantagens e desvantagens comparando com o serviço desejado e é sempre uma forma de tirar os positivos e melhorar a própria ferramenta;

O terceiro capítulo aborda o tema da arquitetura da plataforma onde se explora exemplos de ferramentas compatíveis com o que se pretende incorporar e onde se explica todo o design da arquitetura da plataforma desde a *framework* que disponibiliza os serviços até às ferramentas em si.

No quarto capítulo fala-se da implementação do sistema propriamente dito. Explica-se a escolha das mais variadas tecnologias para a implementação de todas as camadas da arquitetura e, para além disso, aborda-se todas as funcionalidades que foram criadas para facilitar o uso da plataforma.

O quinto capítulo menciona os casos de estudo utilizados para popular a plataforma e para permitir testes a todos os tipos de serviço disponibilizados. Para além disto, aprofunda também com alguns testes de usabilidade utilizando diferentes linguagens de programação e mostrando a facilidade de uso que se obtém com estes serviços.

2 Estado da Arte

Com o estado da arte é pretendido a listagem do conhecimento atual nos campos relacionados com o tema da dissertação.

Considerando que o objetivo é implementar um serviço *web* onde será possível usufruir das mais variadas ferramentas que resolvem problemas de Processamento de Linguagem Natural, é então normal que se investigue a evolução das arquiteturas que tornaram possível a comunicação entre aplicações e também a evolução do PLN, dos seus desafios e das ferramentas que os resolvem. É preciso também analisar as plataformas que já existem onde se encontram parecenças para poder melhorar o produto final.

2.1 Desafios no Processamento de Linguagem Natural

Com a evolução do PLN, especialmente nos últimos anos, a quantidade de problemas e suas soluções foram aumentando e neste momento esta área do conhecimento é aplicada nas mais diversas áreas, da economia à medicina, e conta com ferramentas e plataformas estáveis e complexas, que resolvem de forma eficaz alguns dos problemas mais comuns. Outros problemas há, no entanto, que continuam a fazer parte da investigação, e que melhoram de dia para dia.

De seguida apresentam-se vários exemplos destas várias tarefas:

Segmentação de texto em frases

A divisão de um texto em frases é um desafio aparentemente simples, mas que levanta vários problemas quando abordado de forma rigorosa. Embora se use o ponto final (ou ponto de interrogação ou exclamação) na maioria das línguas para indicar o fim de frase, nem sempre isso acontece. Por um lado, às vezes esse sinal é omitido por alguma razão (por exemplo, se o texto a processar foi obtido de forma automática a partir de fontes estranhas, como documentos *Portable Document Format (PDF)*) e por outro, estes sinais

são usados noutras situações para além da indicação do final de frase, como é o caso das abreviaturas, siglas, números (ou para marcar a casa decimal ou para o agrupar em milhares), em endereços de e-mail ou web, entre várias outras situações.

Segundo Stamatatos et al. [1999], 47% do pontos finais existentes no *corpus* do *Wall Street Journal* não têm como objetivo a delimitação de uma frase, o que demonstra que o problema pode ser mais complexo.

As abordagens passam, pelo menos para as línguas "ocidentais", pelo recurso a um conjunto de heurísticas, como o uso de uma lista de abreviaturas comuns, a deteção de início de frase pela maiúscula inicial, expressões regulares para a deteção de endereços ou números, entre várias outras.

Segmentação de texto em palavras

A divisão de um texto em palavras também parece simples, mas tal como a divisão em frases, também esta tarefa apresenta casos pouco claros. Normalmente associa-se ao espaço e a um conjunto de sinais de pontuação específicos a marca natural para a divisão do texto em palavras. Mas, para além de haver línguas ao qual o uso do espaço não é semelhante às línguas mais convencionais (como a maioria das línguas "orientais"), também há determinados caracteres, como o hífen ou o apóstrofe, que tanto podem funcionar como separadores como conectores entre duas palavras.

Dois exemplos deste problema:

- na língua inglesa, a divisão em palavras de construções com apóstrofe, como "I'm" ou "don't" ou ainda o possessivo "John's car". Enquanto que nos primeiros podemos dividir nos seus componentes naturais: "I am" ou "do not". No entanto, não existe uma solução tão clara para o terceiro caso. Será o possessivo uma única palavra? Fará parte do substantivo possuidor ou possuído?

- para a língua portuguesa não existem casos tão claros, mas se considerarmos um texto desportivo, será natural encontrar segmentos como "Porto-Benfica" o que poderá vir a ser considerado como uma única palavra (solução natural já que a maioria destes hífenes são parte integrante de palavras como "segunda- feira")

Note-se que este processo, para além de dividir o texto em palavras, também é responsável por separar não-palavras entre si. Por exemplo, um endereço de e-mail deverá ser considerado uma "palavra" (o que leva a que muitas vezes se use o termo *token* em vez de palavra,

e se chame *tokenização* à tarefa de separação em palavras).

Correcção ortográfica e sintáctica

Tal como apresentado por Peterson [1980], os primeiros corretores ortográficos eram baseados em dois processos: a divisão de um texto em *tokens* e a análise destes usando um algoritmo próprio normalmente baseado em probabilidades e números de ocorrência.

No entanto, mesmo neste estado inicial é preciso definir o que é uma palavra, que tipo de palavras devem ser corrigidas, que caracteres devem ou não devem ser ignorados (por exemplo, números e hífenes), entre outras coisas.

Alguns problemas desta tarefa é decidir o que deve fazer o corretor ortográfico quando encontra uma palavra não reconhecida ou decidir que palavras incluir no dicionário. Fará sentido incluir todas as palavras do léxico da língua? Será que o verbo "arvorar" deve estar no dicionário? Note-se que se for contemplado pelo dicionário, então a palavra "arvore" passa a existir (imperativo do verbo "arvorar"). No entanto, é pouco provável que alguém use esta palavra, e muito mais provável que a tenha usado por engano, em vez da palavra "árvore".

A maioria dos erros encontrados em texto digital são derivados de três situações:

1. Erros por falta de conhecimento da pessoa que escreve. O mais provável é escrever palavras diferentes (ou *não palavras*) mas que se dizem da mesma maneira, ou seja, homófonas[1].

   Enquanto que as *não palavras* são fáceis de detetar, no caso das palavras homófonas reais, um corretor ortográfico não é capaz de detetar o erro, sendo necessário, para estas situações, o uso de um corretor sintático (que tire partido do contexto para detetar se há probabilidade de determinada palavra aparecer nesse contexto).

2. Erros derivados de distrações aquando da escrita num teclado. Por exemplo, quando se tecla na letra ao lado da pretendida. Existem vários estudos deste tipo de problema [Bourne, 1977];

3. Erros derivados à codificação do texto e ao mecanismo de transmissão da informação.

Com tudo isto ainda apareceu a necessidade de resolver problemas relacionados com o

[1] Por vezes não-palavras homófonas, ou seja, coisas como "pocecivo" vs "possessivo", em que a variante errada não existe

contexto. Existe verificações ao nível do plural, género, tempos verbais e outros indicadores morfo-sintácticos que devem ser feitas. Esta situação é semelhante à já explicada anteriormente com o uso de palavras homófonas.

Radicalização e Lematização

O processo de radicalização (*stemming* ou *stemização*) é a transformação de uma palavra e das suas variantes para uma *palavra-raiz*. A ideia por trás deste processo é que diferentes palavras que sejam relacionadas tenham a mesma raiz. Por exemplo, "comer," "comia," ou "comeu" podem ter como raiz a forma "com."

Esta *raiz* pode até nem ser uma palavra da língua em questão, como no exemplo acima. Este algoritmo é útil pela facilidade com que se implementa, sem necessidade de um léxico completo. E, segundo Krovetz [1993], esta funcionalidade pode contribuir para uma melhoria significativa na recuperação de informação (cerca de 35%).

Para línguas ocidentais os algoritmos de radicalização baseiam-se em oito passos: remover o plural, passar a palavra para o masculino, retirar conteúdo adverbial, retirar diminutivos e aumentativos, retirar sufixos nominais, verbais ou vogais terminais e, por fim, retirar a acentuação [Orengo and Huyck, 2001].

O principal problema deste algoritmo é que usa um conjunto de heurísticas que não funcionam para todas as palavras. Ou seja, é possível que duas palavras completamente distintas tenham a mesma raiz, o que aumenta a ambiguidade.

Isto leva a que exista um outro processo, denominado de *lematização*, que tem como objetivo obter o *lema* da palavra. O *lema* é uma palavra específica, escolhida para representar um conjunto de palavras. Por exemplo, para os substantivos usa-se a forma masculina singular (a não ser que não exista) e para os verbos usa-se o infinitivo. Embora também exista ambiguidade (a palavra "pode" tanto pode ser uma forma do verbo "podar" como do verbo "poder"), este processo é mais informativo que a radicalização.

O principal problema na implementação do processo de lematização é a necessidade de se possuir um léxico base, já que não existem regras para, a partir de uma forma, obter corretamente o seu lema (no entanto, é possível a partir de um lema calcular todas as suas formas derivadas).

Modelos de língua

Aparecendo como grande contributo para o problema do reconhecimento de fala [de Kok

and Brouwer, 2011], um modelo de língua é basicamente um método de obtenção de uma função que, recebendo um texto ou uma frase, apresente quão provável esse texto ou frase faz parte das frases válidas de determinada língua.

Um modelo de língua é construído com a análise das frequências das palavras e da sua co-ocorrência. Assim, um modelo de língua pode ser visto como um conjunto de regras que indicam como as palavras de determinada língua são habitualmente ligadas.

Análise morfológica

A análise morfológica tem como objetivo obter, para determinada palavra, a sua classe gramatical (assim como alguns atributos, como o género, número, tempo verbal, entre outros). Este é um processo dependente de língua, o que leva a que diferentes tipos de línguas tenham soluções relativamente diferentes.

Tipicamente dividem-se as línguas em:

1. Isoladas: quando a maioria das suas palavras são invariáveis, como é o caso do Inglês;
2. Aglutinantes: onde algumas palavras são o resultado da combinação de várias palavras mais simples, como é o caso do Alemão;
3. Línguas flexivas, onde as palavras apresentam muitas formas distintas, como é o caso do Português ou do Húngaro.

Note-se que embora se tenham apontado exemplos típicos para cada uma das classes, a grande maioria enquadra-se em mais que uma classe.

O principal problema da análise morfológica é a necessidade de um léxico de suporte e a incapacidade de desambiguação de palavras homógrafas (existem vários exemplo, alguns dos quais até introduzidos recentemente com o acordo ortográfico de 1990, como a preposição “para” e a forma imperativa do verbo “parar”).

Etiquetação

Um processo semelhante à análise morfológica referida no ponto anterior, é a etiquetação morfo-sintática, ou *Part of Speech (POS) tagging*, que pretende etiquetar um texto com classes morfológicas (e respetivos atributos). Embora este processo possa ser visto como a simples aplicação do método anterior para cada palavra de um texto, é necessário ter em conta que nesta tarefa não é suposto existir ambiguidade, o que obriga a que estes algoritmos sejam capazes de detetar palavras homógrafas e classificá-las corretamente de

acordo com o contexto. Atualmente esta tarefa apresenta resultados na ordem dos 98% de precisão para determinadas línguas [Toutanova et al., 2003].

Tradução automática

Este é talvez o exemplo paradigmático do PLN. O desafio é simples, mas a solução, nem tanto. Pretende-se a tradução de um texto, de uma determinada língua, a língua de origem, para uma outra, a língua de destino ou língua alvo.

Muito se poderia aqui dizer sobre a tradução automática, mas sugere-se a leitura do excelente resumo da sua história e evolução, por Hutchins [2005].

Convém no entanto chamar à atenção que este processo obriga a um conjunto de tarefas nada simples: por um lado compreender o texto a traduzir, saber como representar esse semântica computacionalmente, e conseguir gerar, de novo, o texto traduzido, usando uma estrutura sintática semelhante à original.

Se este procedimento já é complicado com construções frásicas convencionais, imagine-se o problema de traduzir provérbios ou expressões idiomáticas. Isto para não falar na tradução de conceitos específicos de determinada língua (a célebre "saudade" portuguesa, por exemplo).

Para se ter uma ideia da importância deste problema, segundo Knight [2012], a cada mês, 200 milhões de pessoas usam o tradutor da Google onde se pode traduzir, neste momento, texto entre 90 línguas diferentes.

Sumarização

Com a grande quantidade de informação que existe e continua exponencialmente a aparecer, e a limitação temporal do ser humano, surgiu a necessidade de resumir este texto, de modo a que rapidamente se possa obter informação das mais diversas fontes. Este desafio também é, atualmente, uma área de investigação em PLN.

Assim, a *sumarização* corresponde ao processo de redução de um texto num outro, mais pequeno, baseando-se nos aspetos relevantes do texto inicial [Mani and Maybury, 1999]. Apesar de parecer simples, aparecem obviamente problemas, como a coerência contextual, na possível redução do texto em demasia, removendo informação útil, entre vários outros problemas [Mani, 2001].

Análise de Sentimento

A análise de sentimento é outro dos tópicos inseridos no PLN onde a popularidade é alta.

Para se compreender a sua importância basta perguntar como é que determinada empresa pode recorrer às redes sociais (como, por exemplo, o *Twitter*) para saber a opinião dos seus consumidores.

Nesta área de estudo pretende-se identificar a polaridade contextual no conteúdo de determinado texto. Assim, estas aplicações retornam, habitualmente, um de três possíveis resultados de avaliação: positiva, negativa ou neutra, dependendo do sentimento agradado, aborrecido ou indiferente do autor do texto.

Existem atualmente dezenas de ferramentas disponíveis online com o objetivo de resolver este problema.

Resposta a Perguntas

A resposta a perguntas, conhecida por *question answering* ou sistemas *question-answer*, corresponde ao desafio de encontrar em informação possivelmente não estruturada respostas a uma qualquer pergunta. No entanto, não se pretende simplesmente apresentar ao utilizador um texto ou um endereço de onde essa resposta existe (essa é uma tarefa da *recuperação de informação*), mas construir um texto, fluente, que responda a essa questão.

Tal como explicado por Prager et al. [2000], cada vez se procura mais este tipo de funcionalidade e cada vez mais aparecem novos artigos e *websites* relacionados com esta temática.

Habitualmente, o suporte deste tipo de ferramenta é uma base de dados estruturada, construída automaticamente a partir de texto não estruturado, e um conjunto de mecanismos, heurísticas ou regras para a partir de uma pergunta entender o que se pretende realmente responder. Existem atualmente várias iniciativas de avaliação de sistemas de resposta a perguntas, como é o caso do *Cross Language Evaluation Forum (CLEF)*[2].

2.2 Arquiteturas de Comunicação entre Aplicações

O *Remote Procedure Call (RPC)* é uma ideia pioneira na comunicação entre aplicações, tendo aparecido em 1976. Como explicado por Birrell and Nelson [1984], o seu objetivo é que uma aplicação possa invocar uma função remota como se de uma função local se tratasse, bastando para isso saber em que endereço essa função está disponível, e qual a sua assinatura.

[2]Ver http://www.clef-initiative.eu/.

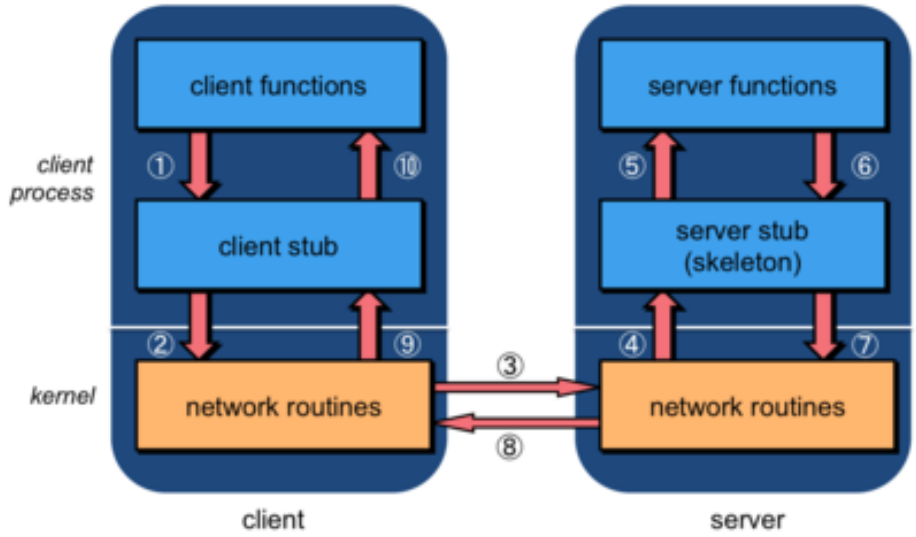


Figura 2.1: Sequência de passos de um modelo RPC [Krzyzanowski, 2010].

Como a figura 2.1 ilustra, cada interveniente numa comunicação baseada em RPC é constituído por três camadas: funções nativas implementadas na própria aplicação, protótipos de funções e funcionalidades ligadas à comunicação de dados.

A aplicação cliente, durante a sua execução, irá precisar, eventualmente, de invocar um método remoto. Nessa altura, a aplicação despertará algum método de bloqueio de modo a parar o sistema, se necessário, enquanto é feito um pedido ao servidor. O pedido, propriamente dito, é enviado pela implementação da tecnologia RPC. Esta implementação irá interpretar a estrutura do pedido e transformá-la numa estrutura especificada pelo protocolo dessa implementação de RPC (*marshalling*), que será enviada usando a comunicação de dados escolhida.

Do lado do servidor, essa estrutura é recebida pela camada de comunicação e a implementação de RPC irá extrair e interpretar o pedido do cliente para que seja executado (*unmarshaling*). A camada aplicacional irá, então, executar esse pedido. O resultado dessa invocação será depois enviado para o cliente usando um mecanismo semelhante.

Como descrito por Andrews [1991], a aplicação cliente é o processo desencadeador enquanto que a aplicação servidor é o processo reator, ou seja, o cliente encaminha processos para o servidor que, após estar continuadamente à espera, reagirá e depois de responder ao pedido, voltará a ficar a aguardar novo motivo de reação.

As vantagens de implementar e usar um modelo destes são:

- Tornar o desenvolvimento de programas com sistemas distribuídos mais simples;
- Aumentar a eficiência da aplicação empurrando alguma carga computacional para outro sistema (pode ser negativo a alguns níveis);
- Permitir o uso de linguagens de programação diferentes de maneira simples e eficaz;
- Usar a maneira mais comum de comunicação entre aplicações: a invocação de funções.

No entanto, e contando com o avanço dos tempos, o modelo RPC tem também algumas desvantagens como, por exemplo:

- É necessário ter em conta todo o tipo de erros que podem surgir, principalmente ao nível da rede e ao nível do funcionamento do servidor;
- Limitações ao nível do protocolo de transporte, como sejam problemas de segurança ou autenticação;
- Necessidade de uma especificação rígida, pela possibilidade de se implementar o modelo de diferentes formas;
- Diminuição da eficiência nomeadamente relacionado ao processo de *(un)marshaling* e à camada de comunicação.

Ao longo dos tempos surgiram várias implementações deste conceito, variando a sua complexidade tecnológica. No entanto, e por mais díspares que possam parecer, a grande maioria continua ativa e usada. De seguida apresentam-se algumas destas implementações.

2.2.1 Sockets

A ligação entre dois sistemas computacionais obriga a que cada um realize uma ligação com o exterior. Esta ligação designa-se por *Socket*. Atualmente, visto que a comunicação entre computadores é, na grande maioria das vezes, realizada usando a família de protocolos *Transmission Control Protocol/Internet Protocol (TCP/IP)* [Comer and Stevens, 2003], os *sockets* são habitualmente designados por *Internet Sockets*.

Na sua base, a comunicação via *socket* pode ser comparada a um canal onde um dos intervenientes escreve dados, e o outro os recebe. Os *sockets* podem ser usados para diferentes tipos de ligação, usando vários dos protocolos da família de protocolos TCP/IP.

A grande maioria das comunicações realizadas na Internet usam um de dois protocolos de transporte, o User Datagram Protocol (UDP) ou o Transport Control Protocol (TCP):

- O *UDP* é usado para a realização de comunicações esporádicas já que é um protocolo de transporte sem conexão: os dados são enviados, uma única vez, num único datagrama, na esperança que o recetor o receba. No entanto, este protocolo de transporte não tem qualquer garantia de entrega. É especialmente útil quando se dá preferência à velocidade de transmissão em comparação com a sua confiabilidade. É um protocolo usado, por exemplo, no *streaming* de vídeo.

- Por sua vez, o *TCP* é um protocolo de transporte que garante a entrega da informação, nem que para isso seja necessário reenviar a mesma informação várias vezes. Nesse sentido, a informação é enviada em pacotes que podem ser mais pequenos que o tamanho dos dados a transmitir. Isto permite que, caso algum deles se perca, seja possível retransmitir apenas parte da informação. É um protocolo de transmissão usado quando é imprescindível que a informação seja entregue, mesmo que com algum atraso. É um protocolo usado, por exemplo, nos protocolos a que nos habituamos na Internet, como o *HTTP*.

Qualquer uma das duas abordagens permite a implementação de protocolos RPC, dependendo do tipo de função remota que se pretende invocar.

É de realçar que o uso de *sockets* é de baixo nível. A API disponível baseia-se, praticamente, na escrita e leitura de bytes (ou carateres), sendo que algumas linguagens disponibilizam funções mais complexas que permitem a escrita e leitura de tipos de dados base, como sejam inteiros, reais ou, em alguns casos, até *strings*.

Isto significa que o uso de um protocolo RPC terá de ser, obrigatoriamente, implementado pelo utilizador. Será ele o responsável por desenhar o protocolo, definindo que dados são transmitidos, em que formato, e com que protocolo base (TCP ou UDP). Não se pode, portanto, considerar que o uso de *sockets* é usar um protocolo RPC, mas sim que é a base sobre a qual os protocolos RPC são implementados.

2.2.2 CORBA

O *Common Object Request Broker Architecture (CORBA)* [Object Management Group, 2006] é uma solução RPC, proposta pelo Object Management Group (OMG), que define uma aborda-

gem baseada no paradigma de orientação a objetos, onde se pretende simplificar a invocação remota de funções, tornando-a uma invocação de métodos remotos sobre objetos locais, ou mesmo métodos remotos sobre objetos remotos.

A incorporação deste tipo de funcionalidade numa linguagem orientada a objetos, de alto nível, permite a distribuição de métodos ou o uso de funcionalidades remotas de uma forma simples e transparente.

O *CORBA* não é uma implementação, mas um modelo de arquitetura, embora este modelo defina um conjunto de detalhes tecnológicos. Ao ser um modelo permite que possa ser implementado em qualquer linguagem, que diferentes autores possam implementá-lo de forma diferente, e que diferentes implementações em diferentes linguagens possam comunicar de forma transparente.

O principal mecanismo para permitir que diferentes implementações consigam comunicar é a definição da interface de comunicação através de uma *DSL* denominada *Interface Description Language (IDL)*. Esta linguagem pode ser vista como semelhante aos célebres ficheiros *header* das bibliotecas C: definem a estrutura dos objetos que são transmitidos e as assinaturas dos métodos remotos.

Para permitir todo o processo de transporte, o *CORBA* utiliza um *Object Request Broker (ORB)* que juntamente com um *Object Adapter* permite todo o processo de *marshaling* e permite que haja capacidade de usar objetos remotamente, sendo que este processo é, todo ele, transparente à aplicação cliente.

2.2.3 WebServices baseados em SOAP

O *Simple Object Access Protocol (SOAP)* é um protocolo que usa as potencialidades de diferentes protocolos já existentes, como o *HTTP* ou o *Simple Mail Transfer Protocol (SMTP)*, para a implementação de uma arquitetura *RPC*. Este protocolo é uma recomendação do *World Wide Web Consortium (W3C)*, e baseado em *Extensible Markup Language (XML)*.

Segundo Box et al. [2000], este protocolo é divido em três partes:

Envelope SOAP

Qualquer pedido via SOAP tem obrigatoriamente incorporado, no respetivo documento XML, um envelope e o seu conteúdo, podendo ter um cabeçalho complementar. A estru-

tura da mensagem é baseada na criação, em primeiro lugar, do envelope, sendo posteriormente adicionado o cabeçalho (se existente) e finalmente o corpo do documento, tal como indica na figura 2.2.

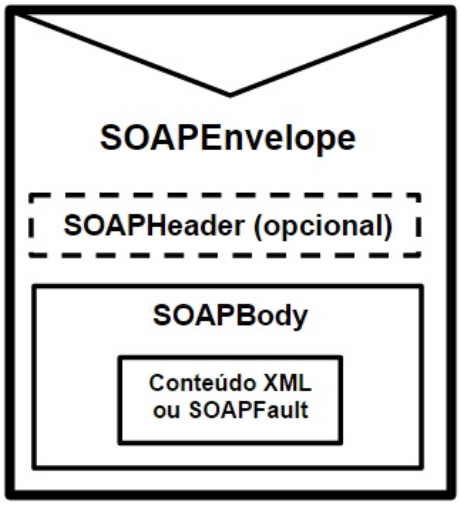


Figura 2.2: Estrutura de um envelope SOAP.

O envelope SOAP, como o nome indica, é quem descreve quem é o destinatário e o remetente da invocação, o que a mensagem contém, e quais os parâmetros que são opcionais.

O cabeçalho tem como objetivo conter informação que não está diretamente relacionada com o conteúdo da mensagem. Exemplos em que faz sentido usar este cabeçalho pode ser quando se pretende fazer autenticação na invocação de mensagens, ou quando se quer limitar o tempo de envio.

O corpo é, logicamente, o conteúdo da mensagem. Este elemento é o responsável por permitir a troca, entre o cliente e o servidor, da informação obrigatória para que o pedido remoto possa ser executado com sucesso. É neste campo que se inclui as invocações RPC e também é aqui que se integram os relatórios de erros;

Regras de codificação do SOAP

As regras de codificação do SOAP são baseadas no sistema de tipos normalmente encontrados em linguagens de programação, bases de dados e informação semi-estruturada, onde cada tipo pode ser simples ou composto por vários tipos simples. Como este protocolo é suportado por XML, esta codificação é feita de forma maioritariamente textual.

Embora se reconheça a verbosidade dos documentos XML, o facto de este ser independente de plataforma torna todo o processo de comunicação mais simples.

A conversão dos dados do seu tipo primitivo (na linguagem de programação usada) para o formato XML e vice-versa (*marshalling*) é realizada pelas aplicações cliente e servidor e, portanto, a definição exata de como as estruturas de dados são convertidas em conteúdos SOAP é praticamente independente do próprio protocolo.

Representação do RPC no SOAP

Um dos principais objetivos do protocolo SOAP é tornar possível o encapsulamento e o envio de pedidos RPC através de protocolos usados no dia a dia e que, portanto, não estarão, em princípio, bloqueados por sistemas de *firewall*.

Para isto seja possível é necessário seguir um conjunto de regras de codificação que normalmente são combinadas com as regras gerais do SOAP referidas no item anterior.

Para a realização de uma invocação remota são necessários cinco argumentos essenciais: o Uniform Resource Identifier (URI) do objeto-alvo (grosso modo, a interface RPC do servidor), o nome do procedimento a invocar, os parâmetros necessários para essa invocação, uma assinatura (opcional) da função, e as informações opcionais do cabeçalho.

Como é também referido em [Box et al., 2000], o grande objetivo do aparecimento do *SOAP* foi tornar todo o mecanismo de comunicação entre aplicações mais simples e extensíveis e, para conseguir isto, prescindiu-se de algumas características tais como o envio de objetos por referência.

Outra funcionalidade interessante (embora opcional) nos serviços *web* baseados em SOAP é a possibilidade de os disponibilizar publicamente. Para isso, e também definido pela W3C, existe um formato baseado em XML denominado *Web Services Description Language (WSDL)* que define quais os métodos disponíveis em determinado serviço, bem como os argumentos e tipos respetivos necessários para a sua invocação [Christensen et al., 2001].

Estes documentos contêm ainda meta-informação de modo a que possam ser alojados em repositórios de serviços de modo a que aplicações possam, de forma dinâmica, consultar quais os serviços que, em determinado momento, permitem realizar determinada funcionalidade.

Um exemplo prático: considere-se uma aplicação de vendas, que pretende fazer o câmbio de preços para a unidade monetária do seu cliente. Quando tiver essa necessidade poderá perguntar ao repositório de serviços quais os que atualmente existem com essa funcionalidade. Selecionado

o serviço, é descarregada a sua descrição WSDL que a aplicação usará para saber como invocar remotamente o serviço.

Embora todo este processo seja interessante, o processo de encapsulamento em XML é demasiado complexo e as regras demasiado restritas para uma comunidade *web* ágil e pouco controlada.

2.2.4 WebServices baseados em REST

Sem dúvida que a facilidade de uso de serviços *web* através de protocolos já existentes, nomeadamente o *HTTP*, revolucionou a mundo deste tipo de serviço. Até então seria necessário que cliente e servidor abrissem portas específicas para a comunicação, enquanto que com o uso de um protocolo "convencional" essa necessidade passa a estar resolvida por omissão.

No entanto, os serviços baseados em *SOAP* eram algo complicados, e obrigavam a um estudo cuidado das recomendações W3C. Assim, e com o objetivo de facilitar a implementação de serviços através da *web*, definiu-se uma arquitetura sem estado (*stateless*), que foi denominada de *Representational State Transfer (REST)*.

A filosofia REST baseia-se em 7 fases, ou camadas, que definem de que modo um serviço deve ser implementado [Fielding, 2000]:

1. A fase inicial é baseada em duas perspetivas. A primeira diz que se deve começar sempre com algo vazio e construir a arquitetura a partir de elementos que sejam familiares aos utilizadores, e a segunda é que ao definir algo novo se deve pensar sempre no produto final, pelo que se deve começar com uma abordagem sem quaisqueres condições e, só quando se sente necessidade de alguma restrição é que ela deve ser implementada.

2. Numa segunda fase, considera-se a implementação de um sistema cliente-servidor, com a respetiva separação entre a interface do utilizador e o armazenamento de dados, o que permite a existência de portabilidade entre plataformas e melhora significativamente a escalabilidade do sistema;

3. A terceira fase foi tornar a comunicação sem estado, ou *stateless*, entre o cliente e o servidor. Ou seja, o pedido enviado pelo cliente tem de conter toda a informação necessária para que o servidor possa responder ao pretendido sem necessidade de guardar qualquer informação sobre os clientes e as suas operações prévias.

Este mecanismo faz que com a visibilidade, fiabilidade e escalabilidade melhore substancialmente:

a) A visibilidade aumenta pois cada pedido apresenta toda a informação necessária, e apenas essa informação sobre o cliente será usada pelo servidor para realizar a sua tarefa.

b) A fiabilidade aumenta porque torna-se simples de resolver falhas de comunicação, bastando para isso o reenvio do pedido.

c) A escalabilidade também aumenta porque, por um lado a ligação se torna mais rápida e sem dependências, e por outro, porque é possível colocar vários servidores a responder a vários clientes sem que seja necessário que estes partilhem qualquer informação.

Embora esta abordagem seja sugerida, é possível simular interações baseadas no uso de um estado (*stateful*), como seja a reescrita do URI, o uso de *cookies* ou simplesmente reenviando a informação necessária (por exemplo, usando campos escondidos em formulários *web*, que guardam o estado da aplicação) [Pautasso et al., 2008].

4. A quarta fase consiste na adição de mecanismos de *cache* com o objetivo de tornar a interação cliente-servidor mais eficiente. Esta *cache* pode eliminar completa ou parcialmente a necessidade de comunicar com o servidor, melhorando a performance do sistema. No entanto, e como em todas as *cache*, apresenta o problema da informação que contém poder ser antiga e, portanto, inválida;

5. A quinta fase consiste na inserção de interface uniforme entre componentes do sistema e é o principal destaque que diferencia a arquitetura REST de todas as outras existentes. Aqui são definidos os recursos que são a base de todo o sistema. É também nesta fase que aparecem mais algumas das características essenciais do REST [Oliveira, 2012] [Pautasso et al., 2008]:

   - **Identificação de recursos via URI:** Os serviços REST disponibilizam os seus recursos através de URI acessíveis aos utilizadores. Isto é, o URI é a chave de acesso a um recurso e é a partir dele que os recursos são disponibilizados aos utilizadores. Cada um desses URI direcionam para um controlador que retorna (via HTTP) a representação desse recurso;

   - **Manipulação dos recursos:** Os recursos oferecidos são manipulados, segundo a filo-

sofia *Create, Read, Update, Delete (CRUD)*, a partir de quatro métodos já existentes no protocolo HTTP: POST, GET, DELETE e PUT.

O método GET é responsável por ler e retornar uma representação do recurso. O método POST é responsável pela criação de novos recursos, nomeadamente daqueles que dependem de informação externa, sendo capaz de atribuir automaticamente um identificador ao novo elemento. Por sua vez, o método PUT que também é utilizado para a criação de novos recursos, mas em que o identificador é especificado pelo cliente. Estes dois últimos métodos são também capazes de realizar a atualização direta de um recurso. Por último, o método DELETE é responsável pela eliminação de um recurso do sistema;

- **Mensagens auto-descritivas:** As mensagens enviadas entre clientes e servidores são facilmente entendidas por ambos pois esta mensagem é *stateless*, e portanto, a mensagem contém toda a informação necessária para responder aos pedidos. Considera-se que a base de um serviço REST é o HTTP e os seus métodos são suficientemente vulgares para cliente e servidor saberem como se comportar perante qualquer pedido. Os recursos não estão associados a um formato próprio sendo que é possível que, para cada um, se possa obter diversas representações, se disponibilizadas. A seleção da representação é indicada pelo *mimetype* enviado no cabeçalho do protocolo HTTP.

6. A sexta fase consiste na definição de composição de serviços para permitir a construção de serviços mais complicados com base em vários serviços mais simples. Isto pode ser visto como um conjunto de camadas que interagem entre si seguindo uma organização hierárquica [Garlan and Shaw, 1994].

   Na verdade, outros protocolos, como o TCP/IP, baseiam-se nesta filosofia: um protocolo como o *HTTP*, na camada de aplicação, é implementado num protocolo (o *TCP*) na camada de transporte, e assim sucessivamente, sobre as camadas de interligação e física.

   Embora a implementação de serviços compostos adicionem um *overhead* em latência na transmissão de informação, podem surgir outros benefícios. Por exemplo, é possível a implementação de um serviço de *cache* transparente, que seja capaz de funcionar para vários outros serviços [Clark and Tennenhouse, 1990].

   Esta estrutura também permite transformar a informação enviada porque, como as mensagens são auto-descritivas, torna-se possível que as camadas intermédias possam analisar

os pedidos e filtrá-los, ou mesmo filtrar o próprio conteúdo dos pedidos [Wolman et al., 1999] [Brooks et al., 1995].

7. Por último, a sétima fase acrescenta uma funcionalidade opcional, denominada de *code-on-demand* [Fuggetta et al., 1998]. Esta restrição permite que o servidor retorne, para o lado do cliente, código para este último executar localmente. Este mecanismo é especialmente útil em situações em que a quantidade de dados a processar é grande, e é preferível transmitir a aplicação e executá-la localmente do que transmitir os dados a processar.

   As vantagens mais óbvias são que esta característica melhora a capacidade de configuração e a escalabilidade do sistema pois, estando num ambiente local, o serviço torna-se mais eficiente. No entanto, também apresenta desvantagens, pois a execução de código não é, necessariamente, multi-plataforma, e porque pode introduzir problemas de segurança, já que a aplicação local poderá ter a informação que o cliente não deseja partilhar.

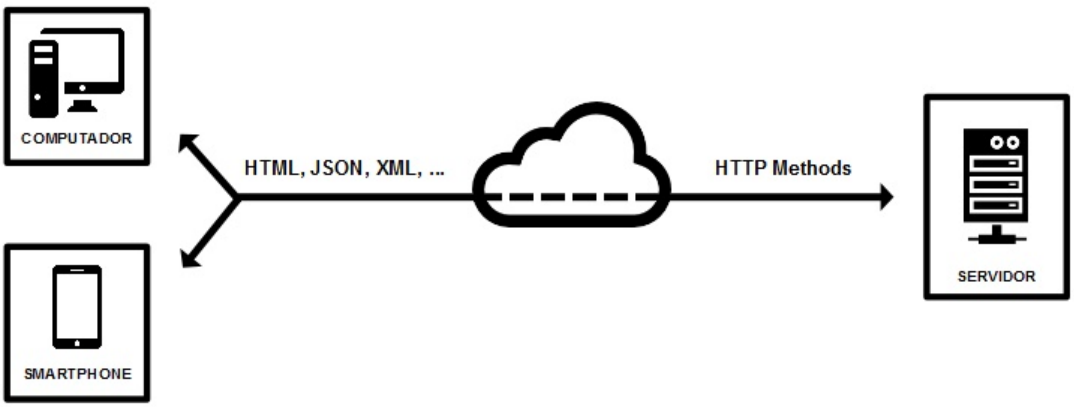


Figura 2.3: Arquitetura simplificada de um serviço REST.

A figura 2.3 representa, de uma maneira simplificada, como funciona a arquitetura REST. Havendo um serviço disponibilizado, e sabendo o utilizador do identificador (URI) do recurso pretendido, este poderá invocar essa entidade/funcionalidade usando os quatro métodos fornecidos pelo protocolo HTTP.

Cada recurso retorna uma representação que, na maioria das vezes, está construída em *JSON* ou *XML*. No entanto, qualquer tipo de resposta é válido, desde que o servidor web indique o *mimetype* da resposta no cabeçalho HTTP.

Note-se, também, que embora não tenha sido devidamente mencionado, um recurso REST também poderá falhar, ou porque o recurso não existe, ou porque o utilizador não tem permissões para invocar esse recurso com o método usado. Para indicar uma resposta de erro também são usados os códigos HTTP. Por exemplo, o serviço poderá retornar o código 404 (*not found*) caso o recurso não exista, ou o código 401 (*Unauthorized*) caso o utilizador não tenha permissão para aceder ou alterar determinado recurso.

A vantagem diferenciadora do REST em comparação com a generalidade dos modelos RPC, é a otimização de todo o processo de comunicação entre o cliente e o servidor.

O REST é mais leve, menos verboso (mensagens com, possivelmente, menos conteúdo), mais fácil de interpretar (fácil de perceber o conteúdo do pedido e da sua resposta) e é também mais fácil de implementar. O REST tem também a vantagem de aproveitar ao máximo a *cache* do HTTP (que é a base do REST) melhorando significativamente o nível de performance. O REST é a melhor opção para casos em que não se pretende que haja estado nas interações com os recursos e quando não se tem largura de banda ilimitada. É também uma ótima alternativa quando se pretende enviar pedidos *Asynchronous JavaScript And XML (AJAX)*, que são cada vez mais parte constante e essencial do mundo *web*.

2.3 Plataformas Web para Processamento de Linguagem Natural

Reconhecendo a utilidade das funcionalidades de uma interface *Web* como referido no capítulo anterior, várias são as iniciativas de plataformas *online* com acesso a serviços ou ferramentas para o Processamento de Linguagem Natural.

No sentido de comparar estes serviços com as funcionalidades que se pretendem implementar, fez-se uma pesquisa de plataformas que fornecem serviços de PLN. Foram encontradas dezenas de resultados com objetivos muito distintos. Com a necessidade de filtrar as várias opções, definiram-se critérios específicos para selecionar quais as ferramentas a analisar cuidadosamente.

O primeiro critério definido foi excluir ferramentas que permitam obter um único tipo de serviço. Muitas das plataformas encontradas permitem apenas solucionar um problema muito concreto e, portanto, não se enquadra nos objetivos definidos.

O segundo critério utilizado foi excluir ferramentas que, apesar de apresentarem vários tipos de resultado/análise, são vocacionadas para um tema muito particular. No entanto, pode-se perfeitamente salientar a API fornecida pela **Semantria**[3] (criada por *LexAlytics*) que, focando-se principalmente na área das redes sociais, consegue obter várias conclusões desde a análise de sentimento até ao tratamento de entidades.

Finalmente, ordenaram-se as restantes plataformas usando índices de popularidade (baseados na consulta de vários blogues temáticos que discutem/referem estas plataformas). Apesar de ser um critério subjetivo, pode-se esperar que os melhores serviços *web* acabem por ser os mais usados e, nesse sentido, os mais referidos e discutidos.

Após esta filtragem, selecionaram-se quatro plataformas que, sendo um pouco diferentes entre eles, contêm elementos importantes que devem ser tidos em conta na construção do serviço desejado.

Considerem-se, então, as quatro plataformas analisadas:

Mashape[4]

Apesar de não ter como base o objetivo específico de disponibilizar ferramentas ligadas ao PLN, é uma plataforma que permite a disponibilização de API sobre quaisquer ferramentas, usando a filosofia REST.

A ideia de reunir um conjunto de API de ferramentas diferentes é um dos pontos-fortes e praticamente únicos do Mashape. Na verdade, algumas das ferramentas que não vão ser aqui referidas, e outras de que se falarão a seguir, contêm algumas das suas API alojadas no Mashape.

O Mashape pode ser visto como um serviço (pago), em que os criadores de API *web* podem colocar os seus serviços disponíveis. A camada do Mashape torna possível tratar de determinados problemas comuns, como a sua monitorização, a sua rentabilização (colocando custos que são geridos pelo Mashape), autenticação, entre outras funcionalidades.

No entanto, a API do serviço tem de ser, de qualquer modo, disponibilizado pelo seu programador, já que o Mashape funciona, essencialmente, como uma *proxy*.

Text-Processing.com[5]

[3] Ver https://semantria.com/ (Consultado em 01-02-2015).
[4] Ver https://www.mashape.com (Consultado em 01-02-2015).

Esta plataforma é também baseada numa filosofia de comunicação via HTTP, mas só aceita invocações via método *POST*. Apesar de oferecer poucos serviços, a ideia implementada coincide em muitos pontos com a que se pretende com este trabalho. As funcionalidades oferecidas são a análise de sentimento, *stemming*, tratamento das classes gramaticais e reconhecimento de nomes, entre outras.

A principal diferença é que, embora forneça algumas funcionalidades, apenas disponibiliza aquelas que são possíveis de obter através da ferramenta *Natural Language Toolkit (NLTK)*.

AlchemyAPI[6]

Construída de base para disponibilizar serviços de PLN, a partir do uso de aprendizagem máquina (*machine learning*), fornece entre outros, os serviços de análise de sentimento, a extração de entidades, palavras-chave, relações, deteção de língua entre outras funcionalidades, mesmo não tão direcionadas para o PLN, como é o caso da deteção de faces em imagens.

Para além do uso direto de uma API REST, também disponibiliza um conjunto de bases de desenvolvimento para várias linguagens de programação.

CORE API[7]

A última plataforma (neste caso não é completamente independente pois encontra-se incorporada noutra grande plataforma) a mencionar foi construída pela TextAlytics e chama-se CORE API. Apesar da TextAlytics fornecer mais algumas API, todas elas disponibilizadas via REST, a CORE API é aquela que se enquadra mais com o tema e com o objetivo do serviço que se irá desenvolver.

Apresenta uma grande quantidade de pequenas API que correspondem basicamente à utilização de várias ferramentas de modo individual apesar de estarem, obviamente, disponibilizadas sob o mesmo serviço. Apresenta um pequeno inconveniente: para cada uma dessas pequenas API é preciso uma chave de licença diferente, o que torna o seu uso um pouco mais complicado.

[5] Ver http://text-processing.com/docs/index.html (Consultado em 01-02-2015).
[6] Ver http://www.alchemyapi.com/developers/ (Consultado em 01-02-2015).
[7] Ver https://textalytics.com/api-core-language-analysis (Consultado em 01-02-2015).

3 Arquitetura SplineAPI

Este capítulo apresenta a arquitetura definida para a implementação do sistema SplineAPI. Em primeiro lugar, na secção 3.1, estudam-se algumas ferramentas de processamento de linguagem natural que se pretendem disponibilizar no SplineAPI. Este estudo pretende analisar os vários tipos de interface que diferentes ferramentas disponibilizam. Posteriormente, a secção 3.2 explica qual a solução desenhada para dar resposta a esses mesmos interfaces.

3.1 Interfaces de Ferramentas para PLN

É evidente que não se pretende, nesta secção, apresentar todas as ferramentas que serão, eventualmente, adicionadas ao SplineAPI. Escolheu-se um conjunto de ferramentas que tente ser suficientemente abrangente, com diferentes requisitos de instalação, diferentes funcionalidades e diferentes interfaces de uso.

O conjunto de ferramentas escolhidas é composto por:

- o `FreeLing`, uma biblioteca C++ para a análise léxica e sintática de línguas essencialmente latinas, que inclui uma interface Perl disponível de forma independente;
- o `Lingua::Jspell`, uma biblioteca C distribuída juntamente com um módulo Perl para a análise morfológica de palavras;
- o `Lingua::NATools`, um módulo Perl que disponibiliza várias ferramentas de linha de comando para o processamento de *corpora* paralelos.

3.1.1 FreeLing

O **FreeLing** é uma biblioteca *open source* para análise linguística, desenvolvida por Lluís Padró no *TALP Research Center*. Neste momento inclui recursos linguísticos para 13 línguas diferentes, desde o português ao russo, passando pelo castelhano, galego e catalão.

As funcionalidades que melhor caracterizam esta ferramenta são as seguintes [Padró, 2012]:

- Atomização (tokenização) e segmentação (divisão em frases) de texto;
- Análise morfológica;
- Reconhecimento de termos multi-palavra;
- Etiquetação morfológica e desambiguação (disponibilizando mais que um algoritmo para essa mesma funcionalidade);
- Análise de grafo de dependências com base em regras;
- Deteção e classificação de entidade mencionadas (nomes, datas, números entre outras);
- Resolução de correferência nominal.

Apesar do grande número de úteis funcionalidades que o FreeLing oferece, o processo de instalação é bastante complexo.

Instalação

O FreeLing não se encontra disponível nem para Windows, nem para Mac OS, nem para maioria dos Linux (e mesmo para alguns Linux para o qual existe um pacote disponível, este nem sempre funciona corretamente), o que torna complicado o acesso à ferramenta.

Se o utilizador tiver a sorte de ter acesso a um pacote funcional, ainda terá de lidar com o problema das dependências. O FreeLing necessita que muitas outras bibliotecas e ferramentas estejam previamente disponíveis no sistema. E muitas dessas são também difíceis de instalar, por exemplo as bibliotecas do *libboost*.

Outra alternativa é o utilizador compilar o código fonte. Neste caso, é claro que em primeiro lugar é necessária a existência de um compilador de C++ e das bibliotecas base. Mas não é suficiente, já que as bibliotecas de que o FreeLing depende também devem estar instaladas. E se o utilizador tiver o azar de também ter de compilar a biblioteca *boost*[1], então vai perder mais algum tempo com experiências, já que esta biblioteca pode ser compilada usando opções

[1]Disponível em http://www.boost.org/

muito distintas, o que facilmente pode levar o utilizador a ter a biblioteca bem instalada mas, na verdade, lhe faltar um par de opções para que o FreeLing possa ser compilado.

Interface de uso

Esta ferramenta pode ser usada como biblioteca quando a implementação está a ser feita com a linguagem de programação C++, linguagem em que o FreeLing está implementado. No entanto, a código fonte inclui interfaces para outras linguagens como o Perl, Java e Python através do uso de interfaces Simplified Wrapper and Interface Generator (SWIG). Infelizmente os ficheiros descritores da interface não são atualizados sempre que o FreeLing muda a sua API, e o próprio SWIG não é capaz de gerar código correto para todos os métodos exportados pelo FreeLing.

3.1.2 Lingua::Jspell

O **`Lingua::Jspell`** é uma ferramenta gratuita e de código aberto bastante útil para a análise morfológica e correção ortográfica de textos.

O seu núcleo é baseado no código do corretor ortográfico *ispell*, e foi desenvolvido por José João Almeida e Ulísses Pinto no âmbito do Projeto Natura.

O seu principal objetivo é que possa ser usado para a língua portuguesa, porém, podem ser usados quaisquer dicionários de qualquer outra língua[2] [Simões and Almeida, 2001]. Para além da língua Portuguesa, existem dicionários experimentais para o Inglês, Espanhol e Latim.

Instalação

A nível de instalação, e apesar de ser bem mais simples que o *FreeLing*, encontra-se também dependente de algumas ferramentas, inclusive um compilador de *C*. Também se apresentam dificuldades aquando da instalação em ambientes *Windows*, a não ser que o utilizador opte pela distribuição Strawberry Perl[3].

O principal problema é que obriga à existência, para além do compilador de *C*, do interpretador Perl. Além disso, como é disponibilizado como Módulo Perl, não é fácil de instalar para quem

[2]Desde que siga os padrões habituais das línguas ocidentais.
[3]Disponível em `http://strawberryperl.com/`

não tiver conhecimentos de como esse tipo de instalação se processa.

Depois do processo de instalação da ferramenta ainda é necessária a instalação de um dicionário num formato específico. Embora o `Lingua::Jspell` inclua uma ferramenta para a instalação destes dicionários, é necessário que o utilizador saiba dessa necessidade. Isto leva, também, a que seja necessário, periodicamente, a atualização do dicionário instalado no sistema.

Interface de uso

Para além do seu núcleo, desenvolvido em C e disponível como biblioteca C, também inclui um nível de abstração escrito em Perl, tornando a interface de programação de alto nível (desde que se use essa linguagem).

Existe também a possibilidade de usar esta ferramenta via linha de comandos através do comando `jspell/ujspell` que, combinado com os argumentos pretendidos, permitirá consultar o dicionário de forma independente da linguagem de programação.

3.1.3 Lingua::NATools

O **`Lingua::NATools`** é um conjunto de ferramentas dedicadas ao processamento de *corpora* paralelos: textos acompanhados da sua respetiva tradução.

A sua implementação em C é baseada, no Twente-Aligner [Hiemstra, 1996], alterada posteriormente por Alberto Simões [Simões and Almeida, 1998]. Para além de um conjunto de ferramentas escritas em C, existem várias interfaces escritas em Perl.

Inclui funcionalidades de alinhamento de frases e palavras, extrator de dicionários probabilísticos de tradução, terminologia bilingue, entre outras.

Instalação

Quanto à sua instalação, apesar de não ser de dificuldade elevada, encontra-se dependente de um leque grande de outros módulos Perl e bibliotecas específicas (como Berkeley DB, SQLite entre outros) para além do compilador de C e do interpretador de Perl. Com isto, e como algum do ênfase desta ferramenta é a disponibilização das suas funcionalidades num ambiente UNIX,

encontra-se também a dificuldade de obter este tipo de serviço numa plataforma Windows.

Interface de uso

A ferramenta fornece uma biblioteca em C e também um conjunto de pequenos programas implementados em Perl, que tornam disponíveis várias funcionalidades não disponíveis na biblioteca.

Para além disto, as suas componentes são utilizáveis a partir de uma linha de comandos o que obrigará o utilizador a dominar esta vertente. Por fim, variadas destas componentes demoram um tempo significativo a executar cabendo ao utilizador geri-lo.

3.1.4 Características globais das ferramentas

Na tabela 3.1 são apresentadas características das ferramentas anteriormente referidas de modo a classificá-las e compará-las quanto à dificuldade no que toca a dependências, interfaces construídas [4] e às línguas [5] que são suportadas.

Nome	Dependências	Linguagens suportadas	Línguas
FreeLing	Bibliotecas libboost, libicu e libz, compilador de C++.	C++, SWIG (Java, Python, Perl), Comando UNIX.	Português, Espanhol e derivados, Inglês, Russo, Francês, Italiano, Esloveno.
Lingua::Jspell	Compilador de C e interpretador Perl.	C, Perl, Comando UNIX, Website.	Português, Inglês, Espanhol, Latim, Francês, Italiano.
Lingua::NATools	Compilador de C, glib-2, sqlite, Berkeley DB, libz, interpretador Perl e alguns módulos.	C, Perl, Comando UNIX.	Independente de Língua (ocidentais)

Tabela 3.1: Características das ferramentas de PLN.

[4] Podem não estar 100% completas quanto às funcionalidades fornecidas.
[5] Podem não estar presentes em todas as funcionalidades que a ferramenta disponibiliza.

3.2 Design da arquitetura

O desafio inicial incluía a modelação e criação de um mecanismo flexível e extensível para descrever a interface entre um serviço REST e qualquer ferramenta instalada no sistema, permitindo que a adição de novas funcionalidades fosse o mais simples possível.

Os principais objetivos na modelação da arquitetura foram:

- serviço modular, em que diferentes serviços não possam de forma alguma interagir entre eles;
- facilidade na adição de novas ferramentas ou funcionalidades, de uma forma independente (sempre que possível) da ferramenta e do servidor REST;
- não necessidade de configurar o servidor com a adição de cada novo serviço;
- permitir contabilizar o uso das funcionalidades disponibilizadas por utilizador;

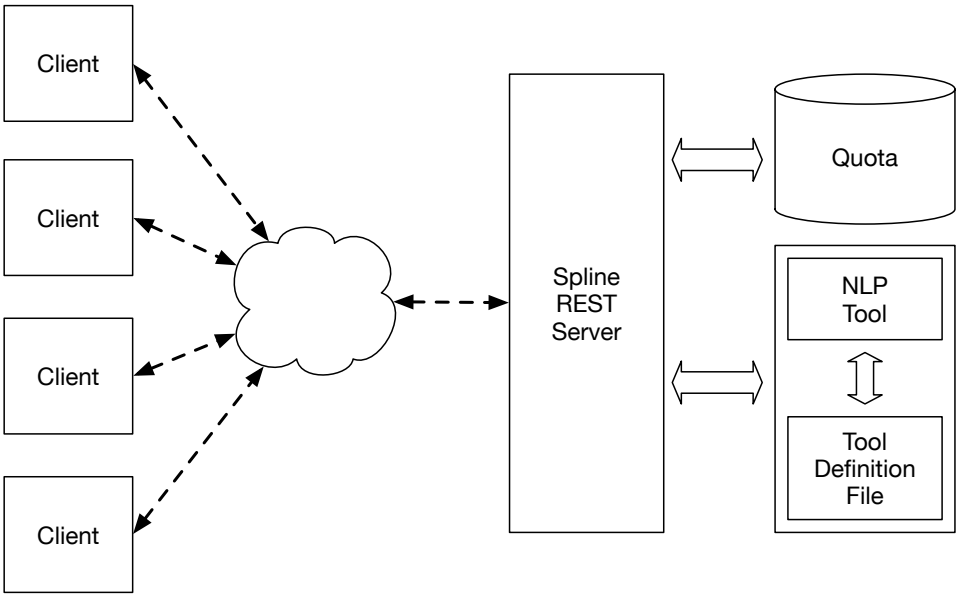


Figura 3.1: Arquitetura da plataforma

A arquitetura apresentada na figura 3.1 foi a escolhida para solucionar os problemas anteriormente referidos.

Cada cliente liga-se por HTTP ao servidor REST. O servidor interage com uma base de dados

para controlar os acessos ao serviços disponibilizados, e interage com um conjunto de definições de funcionamento de ferramentas de PLN.

A ligação entre o servidor REST e as ferramentas de PLN é realizada através de uma DSL baseada na linguagem Perl, que descreve a conexão com a ferramenta.

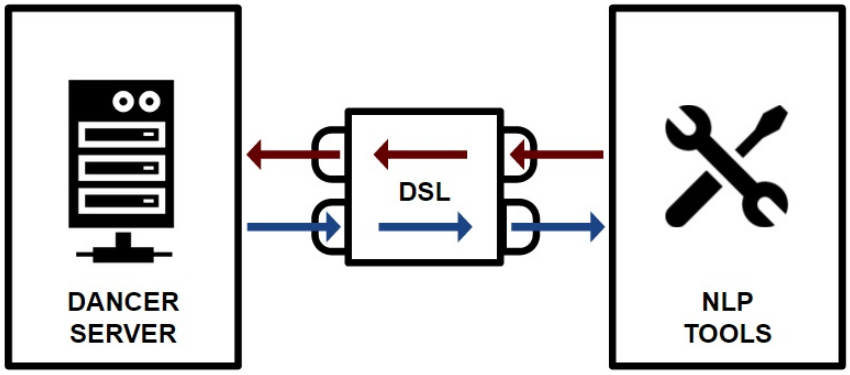


Figura 3.2: Funcionamento da DSL na plataforma.

A figura 3.2 ilustra como a DSL funciona, numa espécie de ponte entre o servidor, responsável por disponibilizar os serviços e gerir os pedidos do utilizador, e as ferramentas que serão responsáveis por executar as tarefas solicitadas, retornando o resultado final para o servidor, que o encaminha para o cliente. Para definir esta ligação existem duas hipóteses: a escrita de um *Módulo Perl*, usando uma DSL sobre a linguagem Perl, ou descrevendo a interação com a ferramenta num documento *XML*.

A opção de seguir com a segunda solução sugere uma maior facilidade na sua criação, acabando por gerar um resultado igual à primeira solução, e a possibilidade de se ter, no futuro, uma interface, gráfica ou não, capaz de gerar todo o processo de ligação de uma forma praticamente automática.

Quanto à gestão de pedidos, mesmo sendo o serviço orientado à conexão, foi necessário implementar um método para controlar o uso dos recursos computacionais da plataforma por parte dos utilizadores individualmente. Para isso ser exequível foi adicionado então uma estratégia via quotas, através de um pequeno sistema de autenticação, que, para além de permitir distinguir o consumo particular dos mais variados serviços, permite atribuir pesos diferentes para funcionalidades distintas. Este plano de quotas consistiu numa estratégia baseada em moedas onde cada utilizador tem um número de moedas para gastar num determinado tempo definido, neste caso diário, e cada serviço apresenta um custo de uso, indicado na DSL correspondente. Para

além disso, como o conteúdo textual enviado pode atingir tamanhos elevados, cada DSL das funcionalidades apresenta a capacidade de acrescentar um conjunto tamanho-custo aumentando o gasto final do pedido se determinado comprimento textual de uma determinada variável passar os limites definidos.

Para isso foi criado um pequeno modelo de base de dados, bastante simples, apenas uma entidade principal, o utilizador, e a sua lista de pedidos (ou história).

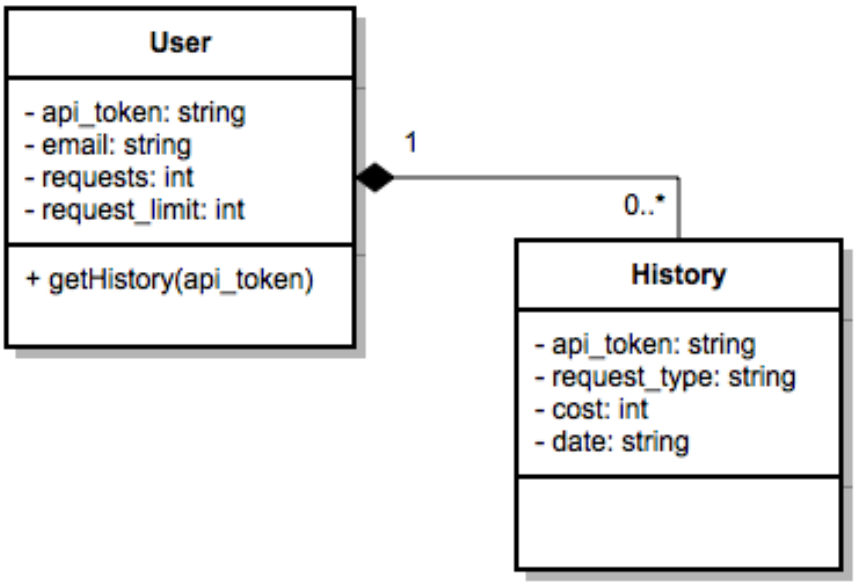


Figura 3.3: Modelo de BD para armazenamento de quotas de utilizadores.

O modelo resultante inclui duas tabelas: o utilizador '***User***' e a história '***History***'.

A tabela '***User***' contém:

- *api_token*: será o *token* responsável por especificar o utilizador num qualquer pedido pois é um parâmetro obrigatório ao utiliza-lo. Este elemento é gerado automaticamente pela plataforma e enviado ao utilizador enquanto que, no motor de base de dados, é guardada a sua informação encriptada.
- *email*: será o responsável pelo processo de autenticação, por permitir o envio ou reenvio do *token* e por tornar possível a comunicação plataforma-utilizador.
- *requests*: indica o número total de gastos diários que o utilizador em questão teve. Diariamente é executada a limpeza (reset) deste valor.

- *request_limit*: mostra qual o limite máximo que o utilizador pode gastar nesse dia. Permite que se pode distinguir utilizadores havendo diferentes níveis de liberdade a utilizar a plataforma.

A tabela '***History***' contém:

- *api_token*: será, tal como no modelo do ***User***, o que identifica o utilizador que, neste caso, foi quem realizou o pedido. Neste âmbito, possibilitará que se encontre todo o historial de um cliente em particular.
- *request_type*: é responsável por mostrar qual o tipo de pedido que o utilizador executou.
- *cost*: ilustra o custo total do pedido em questão.
- *date*: indica quando é que o pedido foi realizado.

Com esta estratégia é também possível acrescentar elementos extra à plataforma tais como a possibilidade de se vender um limite superior de pedidos, ou mesmo a possibilidade de se criar estatísticas baseadas no historial de pedidos tanto individual como globalmente.

4 Implementação da plataforma

Definida a arquitetura do sistema foi necessário proceder à implementação de um protótipo. Sendo o objetivo a criação de uma API que centraliza as mais distintas ferramentas de PLN, foi então decidido que usar uma linguagem de programação, onde o tratamento de informação textual seja o mais simples e eficiente possível, é essencial. Com isto em mente e com a ideia de implementar um serviço web com ideologia REST, a linguagem de programação *Perl* foi escolhida para criar toda a base da arquitetura pois fornece todos os utensílios desejados e permite usá-los eficientemente [Dominus, 2005].

4.1 Definição da *Domain Specific Language*

Tal como referido anteriormente, decidiu-se pela implementação de uma DSL que será a principal componente caracterizadora das funcionalidades que se pretendem disponibilizar. Ou seja, esta define a interface entre a ferramenta (esteja ela disponível como biblioteca, ferramenta de linha de comando ou outra) e a plataforma.

As linguagens de domínio específico podem ser implementadas de duas formas muito distintas. Por um lado, pode-se definir uma linguagem nova, desenhada propositadamente para determinado fim. Nesta abordagem torna-se necessária a definição também de um reconhecedor que seja capaz de processar a linguagem e de a animar, de algum modo. Por outro lado, é habitual que a linguagem de domínio específico seja definida como um subconjunto de uma linguagem já existente, ou simplesmente embutindo essa nova linguagem numa outra.

Com a escolha da linguagem Perl como base de implementação do sistema, a opção de definir uma DSL como um conjunto de funções pré-definidas Perl tornou-se óbvia, já que torna todo o processo de definição da DSL simples, reduzindo-o à implementação das funções necessárias para animar a DSL.

Para descrever cada um dos serviços REST é necessário especificar um conjunto de detalhes:

- nome que identificará o serviço;
- parâmetros, obrigatórios ou opcionais, que receberá para executar o pedido. Para cada parâmetro deve ser também possível acrescentar uma descrição e um valor por omissão, se assim se desejar.
- o próprio serviço conta também com a sua própria descrição, que serve como ajuda ao utilizador sobre os objetivos do serviço;
- contém também a informação sobre o custo de utilização da funcionalidade e, se aplicável, custos extra, para casos onde o conteúdo textual enviado seja mais extenso, através de uma combinação *tamanho-custo*.

Após a organização da informação numa estrutura própria, foi necessário implementar as funções que receberão, tratarão e retornarão os dados necessários para o sistema funcionar fazendo com que um módulo tenha:

- Duas funções base: uma função que retorna o *token* identificador (*get_token*) e uma função que retorna a informação do serviço, referida anteriormente, numa estrutura de dados específica (*get_info*):

```
sub get_token {
  return $index_info{hash_token};
}

sub get_info {
  return \%index_info;
}
```

- A função que automaticamente validará os parâmetros. A validação dos parâmetros é essencialmente a verificação da existência de todos que são obrigatórios considerando que se podem enviar argumentos opcionais ou que sejam ignoráveis caso o serviço não os contemple.

```
sub param_function {
  my ($input_params) = @_;
  my $flag = 1;
  for my $param (keys %{$index_info{parameters}}){
    if ($index_info{parameters}{$param}{required} == 1){
      $flag = 0 if (!exists($input_params->{$param}));
    }
```

```
    if ($index_info{parameters}{$param}{default}){
      $input_params->{$param} =$index_info{parameters}{$param}{default}
        if (!exists($input_params->{$param}));
    }
  }
  return $flag;
}
```

- A função que calcula o custo do pedido. A função do cálculo do custo é simplesmente o somatório do custo fixo do uso do serviço com o custo extra se o tamanho dos mais variados argumentos enviados exceder os limites impostos pelo módulo. Este custo extra pode estar associado a qualquer parâmetro do serviço.

```
sub cost_function{
  my ($input_params) = @_;
  my $cost_result = 0;

  for my $param (keys %{$index_info{text_cost}}){
    my $text_length = "";
    if($index_info{parameters}{$param}{type} eq 'file'){
      $text_length = -s "$input_params->{$param}";
    }
    else{
      $text_length = length($input_params->{$param});
    }
    for my $pair (@{$index_info{text_cost}{$param}}){
      if($text_length >= int($pair->[0])){
        $cost_result += $pair->[1];
      }
    }
  }

  my $final_cost = $cost_result + $index_info{cost};
  return $final_cost;
}
```

- A função que executará o pedido. Esta função terá características próprias ao serviço em questão. No exemplo abaixo aparece a função principal que invocará uma função particular dependente do serviço em questão. É também nestas funções que entrará, em princípio, o uso das interfaces indicadas para carregar. Por fim, esta função devolve sempre uma estrutura em formato JSON de forma genérica para ser devolvido ao utilizador.

```
sub main_function {
```

```
    my ($input_params) = @_;
    my $result = _tool_service($input_params);
    my $json = encode_json $result;
    return decode_utf8($json);
}
```

No anexo 6.1 é apresentada a implementação de um módulo completo.

4.2 Disponibilização de serviços

Como a intenção de se criar um sistema que disponibiliza serviços REST e de modo a tornar a implementação tão rápida quão eficiente, foi preciso descobrir uma *framework* que facilite todo esse processo. Como anteriormente referido, a linguagem de programação base do projeto é o Perl e portanto decidiu-se optar pelo uso da *framework Dancer2*[1] [Sukrieh, 2013].

O *Dancer2* destaca-se positivamente por ser muito leve e por disponibilizar uma API que permite abstrair imensos passos e criar um serviço eficiente em termos de implementação e de utilização.

Considerando o serviço REST como responsável por disponibilizar os serviços ao utilizador, é então requerido que seja capaz de gerir os módulos criados e instalados no servidor para que execute os pedidos recebidos. Deste modo foi necessário arranjar um algoritmo que, quando o servidor arranca, permita que este carregue toda a informação das funcionalidades existentes. Com o auxilio de um módulo que permite o carregamento de outros módulos em *runtime* e com o uso de uma estrutura baseada num *array* associativo que recolhe os dados necessários de cada um desse módulos e os guarda e organiza, é então possível utilizar-se essa informação aquando da invocação do respetivo pedido de uma maneira simples.

O algoritmo do servidor consiste, em primeiro lugar, na espera por pedidos numa qualquer rota. Quando recebe algum pedido, vai então verificar se essa rota está implementada e, se estiver, executa o conjunto de instruções associadas que normalmente corresponde à execução do serviço, tal como detalhadas na secção anterior.

Os serviços são fornecidos ao utilizador via HTTP POST. Cada pedido suporta o envio de múltiplos argumentos, inclusive múltiplos ficheiros, ou seja, contém tipo de conteúdo *multipart*.

[1] Ver http://perldancer.org/ e https://metacpan.org/release/Dancer2

Com um pedido para um serviço, o que a *framework* procura fazer é criar uma cópia local dos ficheiros (se existirem), guardando os caminhos respetivos para acesso posterior, e verificar se o conjunto de parâmetros enviados estão enquadrados com as regras estabelecidas para a funcionalidade em questão, usando uma das funções existentes no módulo com esse objetivo.

Após esta verificação, é necessário também descobrir se o utilizador ainda tem a capacidade para realizar novos pedidos no próprio dia. Para isto ser descoberto basta saber o custo total que o pedido irá ter e, para isso, existe já implementado, ao nível do módulo, uma função que calcula automaticamente esse valor. Com esse custo total, faz-se uma busca à informação do utilizador na base de dados retirando a quantidade de moedas que este ainda pode gastar e verifica-se se a diferença entre os dois números é positiva.

O motor de base de dados escolhido para realizar a tarefa de armazenamento de dados foi o *SQLite*[2] pois é um motor que favorece o desenvolvimento e o teste da aplicação devido à sua facilidade de uso, a não necessidade de grandes e complicadas configurações e a sua portabilidade.

Existem três pontos em que o *SQLite* não seria, de todo, a melhor opção:

1. haver interesse em ter os dados separados da aplicação, ou seja, a existência de um sistema implementado por camadas. Na verdade, as bases de dados *SQLite* funcionam quase como que um ficheiro de dados convencional. A sua principal vantagem é o facto deste ficheiro estar preparado para responder eficientemente a pesquisas expressas em SQL.

2. existir grandes nível de concorrência no que toca à escrita na base de dados; no caso concreto do Spline, o acesso à base de dados é esporádica, apenas para a adição de um registo, pelo que a concorrência não é, ainda um fator impeditivo no uso do *SQLite*;

3. armazenamento de grandes volumes de dados. No entanto, a quantidade de dados a armazenar não é assim tão grande, sendo os limites das bases de dados *SQLite* mais que suficientes.

Para o acesso à base de dados optou-se pelo uso de um *plugin* desenvolvido propositadamente para a *framework Dancer2*, o `Dancer2::Plugin::Database`[3].

Por último, o servidor é responsabilizado por fazer a atualização das moedas usadas na base de dados e criar um novo elemento no histórico do utilizador. Após isso executa a função principal da funcionalidade e retorna o JavaScript Object Notation (JSON) resultado ao utilizador.

[2] Ver http://www.sqlite.org/
[3] Disponível em https://metacpan.org/release/Dancer2-Plugin-Database

4.3 Geração automática de módulos

Com a implementação da base do sistema, verificou-se que a criação de módulos de raiz era uma tarefa que para além de ser um tanto complexa, requeria bons conhecimentos de Perl, e era dispendiosa em termos de tempo. Decidiu-se, então, arranjar uma maneira de criar automaticamente o módulo a partir de um formato mais simples e claro.

Sendo necessário um formato que seja de fácil manipulação e compreensão e com regras muito bem definidas, a solução passou por se criar uma estrutura em XML que incorporasse toda a informação pretendida no módulo respetivo.

4.3.1 Criação de um XML Schema

A primeira etapa consistiu na criação de uma estrutura capaz de conter toda a informação necessária à criação de um novo serviço.

Para tornar mais fácil ao utilizador a construção deste tipo de ficheiros, e para permitir a validação em *runtime* do ficheiro XML gerador, criou-se um XML Schema. Com este *schema* será possível adicionar restrições à informação enviada, desde explicitando quais os dados obrigatórios e opcionais, ao número de vezes que um certo elemento deve existir ou até à tipagem da informação (texto, números inteiros, booleanos).

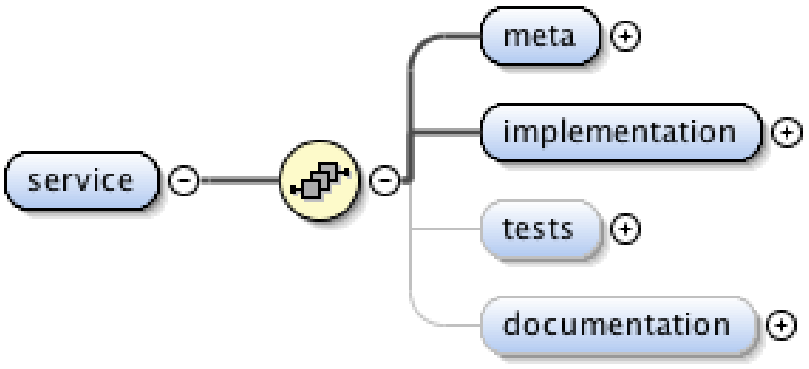


Figura 4.1: Conteúdo do elemento raiz do ficheiro XML.

A figura 4.1 mostra as quatros principais divisões do ficheiro XML e da geração de módulos:

1. **meta**: a meta-informação relacionada com o serviço;

2. **implementation**: contém a informação relativa às instruções específicas para correr a funcionalidade;
3. **tests**: contém os testes para validar a execução das instruções inseridas na implementação. Os testes não são obrigatórios;
4. **documentation**: permite criar uma documentação mais pormenorizada ao módulo criado. Esta opção também é opcional.

Aqui pode-se ver um extrato de como é representado este elemento em XML:

```
<service>
    <meta>...</meta>
    <implementation>...</implementation>
    <tests>...</tests> #opcional
    <documentation>...</documentation> #opcional
</service>
```

Como já referido anteriormente, o elemento *meta* contém toda a informação estática do que ao serviço pretendido diz respeito. Com a figura 4.2 pode-se constatar o conjunto de elementos incorporados na meta-informação. Estes são:

batch Este atributo booleano vai caracterizar o tipo de módulo que se irá criar consoante o método de uso e os problemas de desempenho associados. A secção 4.4 irá abordar mais detalhadamente as alterações correspondentes aos dois tipos de abordagem.

tool Este elemento contém o nome da ferramenta do sistema que disponibilizará o serviço em questão. Serve simplesmente para criar uma hierarquia de módulos havendo possibilidade de uma organização por ferramenta.

name Como o nome indica, aqui é pretendido o nome do serviço a disponibilizar que, na prática, será simplesmente o nome do módulo gerado (juntamente com o nome da ferramenta).

route Este campo é específico para inserir o nome que identificará o serviço aquando do seu uso. Ou seja, quando se pretender invocar a funcionalidade, o *route* é o invocador.

parameters Como em qualquer funcionalidade é normalmente necessária a existência de argumentos apesar de, por vezes, estes serem opcionais. Neste caso em particular, estes serão basicamente a informação enviada no pedido POST à plataforma excluindo o *token* de autenticação.

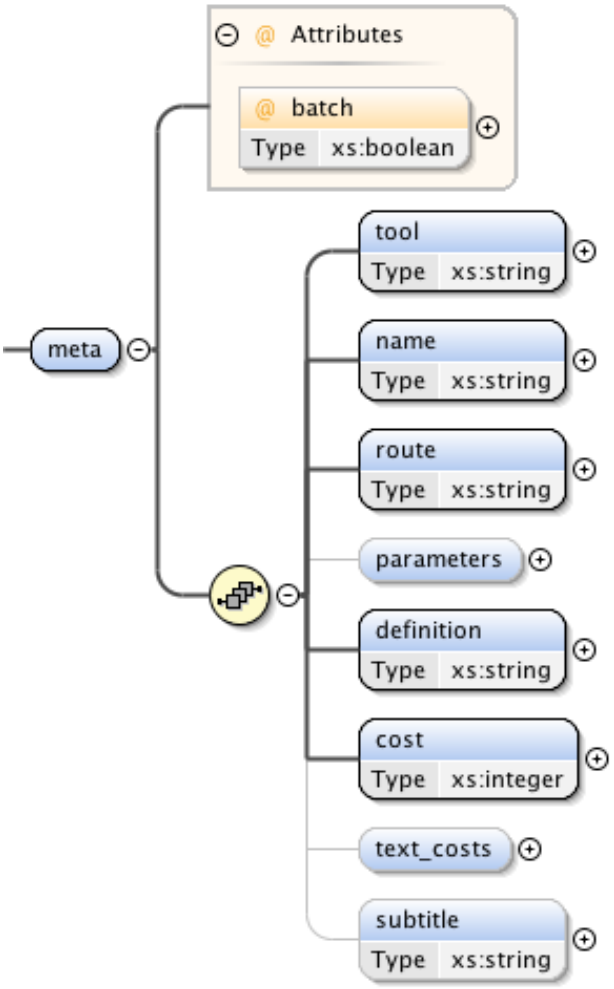


Figura 4.2: Conteúdo do elemento *meta* do ficheiro XML.

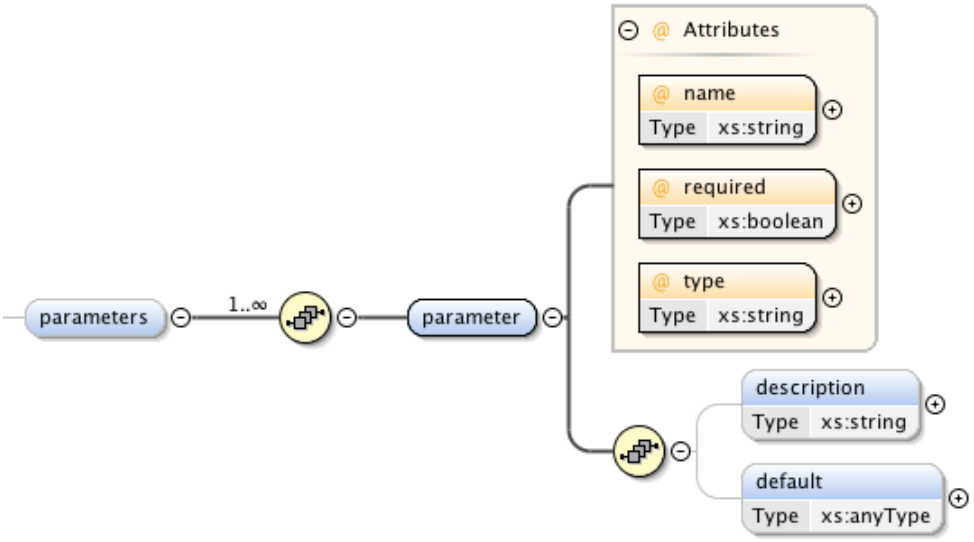


Figura 4.3: Conteúdo do elemento *parameters* do ficheiro XML.

Como a figura 4.3 permite verificar, o elemento *parameters* contém vários parâmetros onde cada um deles tem:

- **name** - O nome do parâmetro;
- **required** - Um indicador booleano que indica se o parâmetro é obrigatório ou não;
- **type** - Indicador de tipo do conteúdo do parâmetro. Serve para tornar possível alguns testes e a criação de uma demo automática via página *web*;
- **description** - Descrição do parâmetro em questão. Este campo é opcional;
- **default** - Valor por defeito atribuído ao parâmetro caso este não seja declarado. É também opcional.

definition Campo responsável por conter uma explicação sobre como a ferramenta atua, para que serve, entre outras coisas. Basicamente é o que esclarecerá mais pormenorizadamente qual o objetivo da funcionalidade.

cost Este elemento corresponde ao custo fixo de uso da funcionalidade.

text_costs Como quanto maior for o conteúdo enviado para ser tratado num pedido, maior é o tempo e o esforço computacional, este elemento permite controlar isso aumentando o custo

total consoante o tamanho do conteúdo enviado.

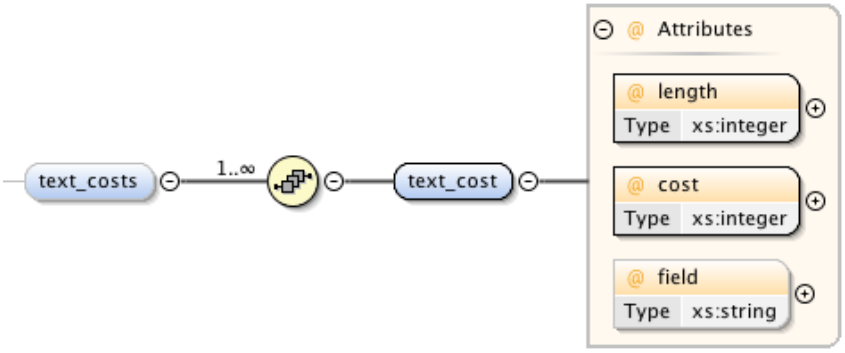


Figura 4.4: Conteúdo do elemento *text_costs* do ficheiro XML.

Como indica a figura 4.4, cada elemento filho deste conjunto contém o tamanho, o custo e o parâmetro respetivo. Ou seja, cada entrada será uma nova restrição a um parâmetro que influenciará no custo final.

subtitle Este campo opcional serve para se atribuir um pequeno subtítulo à funcionalidade, ou seja, uma espécie de segundo nome.

Este elemento da informação inicial apresenta um esqueleto XML no seguinte formato:

```
<meta batch="true|false">
    <tool>...</tool>
    <name>...</name>
    <route>...</route>
    <parameters> #opcional
        <parameter batch="..." required="true|false"
        type="number|text|textarea|file|email|...">
            <description>...</description> #opcional
            <default>...</default> #opcional
        </parameter>
        ...
    </parameters>
    <definition>...</definition>
    <cost>...</cost>
    <text_costs> #opcional
        <text_cost length="..." cost="..." field="..."/>
    </text_costs>
```

```
    <subtitle>...</subtitle> #opcional
</meta>
```

Após a inserção da meta-informação, é importante inserir o código caracteristicamente responsável por realizar o pedido, comunicando com a ferramenta que o executará.

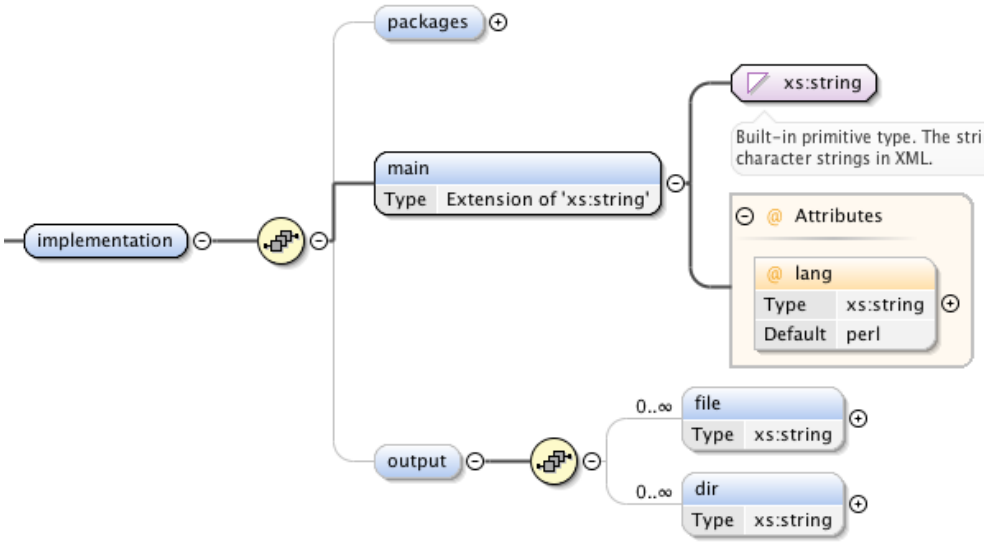


Figura 4.5: Conteúdo do elemento *implementation* do ficheiro XML.

Como mostra a figura 4.5, o campo da implementação contém dois grupos mais pequenos: o *packages* e o *main*.

O primeiro deles é responsável por carregar, para o módulo a gerar, as bibliotecas necessárias para o código funcionar. Estas bibliotecas podem ir desde as interfaces que permitem comunicar indiretamente com os programas instalados até a bibliotecas que permitem estruturar o código de maneira diferente ou facilitar processos de implementação. Pode-se carregar zero ou múltiplas bibliotecas dependendo do tipo de código que se pretende usar no próximo campo.

O segundo campo contém o código a executar na função principal do módulo pretendido. No entanto, como a informação realmente necessária para correr com sucesso é já indicada antes na meta-informação, este código não necessita de conter as declarações dos parâmetros pois estes já são gerados e atribuídos anteriormente sem o utilizador saber. Contudo, é preciso que o utilizador siga algumas normas tais como devolver uma estrutura resultado, fazer uso de bibliotecas anteriormente carregadas, entre outras coisas. Neste campo pode-se inserir, utilizando o atributo

lang, tanto código Perl como comandos UNIX que são integrados num conjunto de instruções Perl sem haver necessidade de saber como.

Como último campo deste elemento tem-se o *output*. Este elemento output, apesar de no *schema* não haver uma verificação própria, só será validado quando o atributo *batch* da meta-informação estiver a *true*. O que contém serão caminhos para os ficheiros ou pastas resultantes do pedido em questão. Desta maneira, o envio do resultado final ao utilizador será instantâneo apesar de o conteúdo em si ainda não está preenchido. Esta estratégia será melhor abordada na secção 4.4.

O código deste importante elemento na geração do módulo apresenta-se na seguinte forma:

```
<implementation>
    <packages> #opcional
        <package>...</package>
        ...
    </packages>
    <main lang="perl|bash">
        ... #código
    </main>
    <output> #opcional
        <file>...</file> #opcional
        <dir>...</dir> #opcional
        ... #opcional
    </output>
</implementation>
```

A seguir à parte essencial para o módulo funcionar, aparecem as duas partes opcionais que acabam por aprimorar o módulo facilitando a vida ao utilizador que o insere:

- **tests** - Como a estrutura indica, o campo dos testes pode ser também uma parte importante na criação e adição de um serviço na plataforma. Como em qualquer funcionalidade, testa-la para verificar se foi bem instalada, se retorna os valores desejados, se foi bem carregada no servidor ou se simplesmente contém erros no código que devem ser resolvidos, é algo que deve ser sempre tomado em conta.

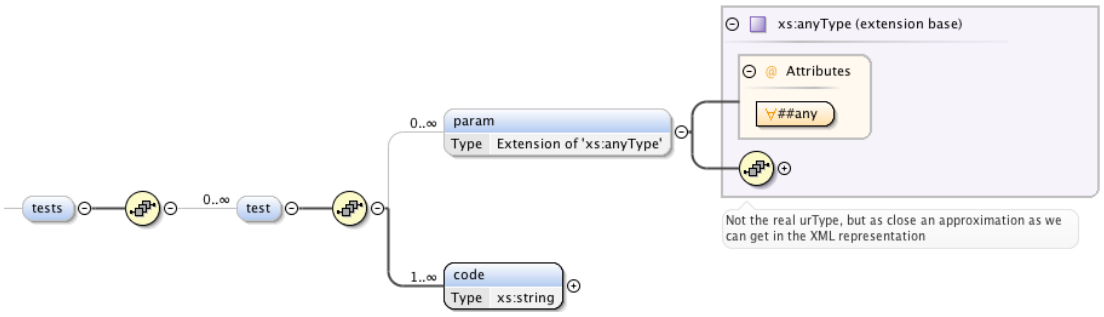


Figura 4.6: Conteúdo do elemento *tests* do ficheiro XML.

Como se verifica na figura 4.6, pode-se inserir múltiplos testes onde cada um contém um conjunto de *param* e um conjunto de *code*.

O primeiro serve simplesmente para atribuir um valor a um dos parâmetros adicionados na meta-informação. Ou seja, servirá para simular um pedido POST que será enviado à plataforma, retornando o resultado final. Com isto testar-se-à se a plataforma contém realmente o serviço integrado e se o módulo está funcional.

O segundo terá o teste em si. Atualmente, este teste baseia-se em código Perl no formato Test Anything Protocol (TAP) [4] fornecido pelo módulo *Test::More*. No entanto, no futuro, o objetivo é incorporar uma linguagem de *query* JSON como, por exemplo, JsonPath[5] ou JSONiq[6].

O TAP é um protocolo que permite fazer testes unitários num ambiente de testes automáticos. Esta interface é toda ela baseada em texto e, portanto, estes testes são facilmente transcritos num ficheiro XML definido. O TAP está disponível para diversas linguagens mas foi no Perl que surgiu e onde mais se desenvolveu através de módulos como o *Test::More* ou *Test::Simple*, por exemplo. Com a facilidade que existe para correr testes em módulos Perl e com a facilidade que existe para se criar testes unitários com o conceito do TAP, optou-se pelo seu uso.

O formato destes elementos tornam-se simples em XML tendo a seguinte estrutura:

[4]Ver https://testanything.org/

[5]Uma linguagem idêntica ao XPath, para JSON, disponível em: http://goessner.net/articles/JsonPath/ (Última visita: 15-04-2015).

[6]Uma linguagem de *queries* completa e expressiva para JSON, disponível em: http://www.jsoniq.org/ (Última visita: 15-04-2015).

```
<tests>
    <test>
        <param name="...">...</param> #opcional
        ...
        <code>...</code>
        ...
    </test>
    ...
</tests>
```

- **documentation** - Tal como mostra na figura 4.7 abaixo exibida, o campo da documentação está dividido por vários *headers* que acabam por ser tópicos onde o seu nome é definido no atributo *title*. Dentro destes, inserir-se-à o conteúdo textual do elemento principal. Este conteúdo deve ser escrito consoante a linguagem usada para documentação de módulos Perl.

Figura 4.7: Conteúdo do elemento *documentation* do ficheiro XML.

E o seu equivalente em XML é tão simples quanto o extrato a seguir ilustrado:

```
<documentation>
    <header title="...">...</>
    ...
</documentation>
```

4.3.2 Tratamento do XML

Depois da definição do *schema* e da criação de um exemplo de um ficheiro no formato XML, tal como se pode verificar no segundo anexo dos apêndices, é necessário este passar por um processo de *parsing* que se traduz na criação, e respetivo ajuste, de um módulo Perl que se incorpora diretamente no sistema arquitetural da plataforma.

Baseando novamente nas potencialidades do Perl e nas bibliotecas existentes, criou-se então um *script* responsável por toda essa transformação. Esse *script* baseia-se no uso da biblioteca *XML::DT* [Almeida and Ramalho, 1999] que facilita o processo de tratamento do XML permitindo associar código a cada elemento encontrado no documento XML enviado.

Eis as etapas executadas pelo *script* baseado na ordem dos elementos XML:

1. Testar se o ficheiro XML respeita as regras, anteriormente enumeradas, que o *schema* impõe. Se não passar este teste, passa todo o processo à frente;

2. Quando encontra a informação relativa ao nome da ferramenta que executará o pedido, cria um módulo intermédio de modo a criar uma hierarquia modular. Este módulo é também essencial para tornar possível o carregamento de todos os seus módulos-filho ao nível do servidor permitindo a sua posterior disponibilização;

3. Ao deparar-se com o nome do módulo pretendido, gera-o com as definições por defeito, ou seja, aplica o comando da criação de módulos e aplica um pequeno *template* ao seu código fonte;

4. Quando se vai obtendo o conteúdo relativo à meta-informação, o gerador vai adicionando esses dados numa estrutura em memória. No final deste elemento e após a inserção das instruções que carregam as bibliotecas adicionadas pelo utilizador, insere-se, no código do módulo, a tabela de *hash* estruturada de uma forma especifica com a informação guardada;

5. Quando se encontra o conjunto de bibliotecas que se pretende que o módulo carregue, é inserido no inicio do ficheiro com o código fonte permitindo uma melhor organização e limpeza do código;

6. Após isto, é juntado o código das funções que estão, por omissão, já definidos. São as funções que testam os parâmetros enviados, a função que calcula o custo total e a função principal que invocará a função que a seguir será inserida. Estas funções são iguais em todos os módulos, ou seja, acaba por ser um *template*;

7. Depois disso adiciona-se então o código responsável por executar o pedido em si. Cria-se uma função secundária que contém uma parte inicial gerada automaticamente e que atribui a variáveis os valores dos argumentos enviados. Depois terá então a execução propriamente dita do serviço;

8. Quanto aos testes, cria-se um ficheiro de teste individual para cada elemento substituindo a

informação do XML em código num formato TAP. Existe aqui também uma parte gerada que permite criar todas as instruções necessárias para enviar um pedido POST, mesmo que localmente, para ser possível um teste completo;

9. Quanto à documentação, cria-se uma lista de *headers*, ou seja, títulos e associa-se, a cada um, um conteúdo textual que se adequa com este.

4.4 Tratamento de pedidos de maior duração

Com a possibilidade da existência de serviços que demoram mais tempo a executar, foi preciso implementar uma solução que evitasse tanto os *timeouts* do HTTP como o elevado número de conexões criadas durante o seu tempo de execução.

Para evitar os problemas referidos, é preciso então terminar a comunicação com o cliente a meio do processo, arranjando uma forma deste saber mais tarde o resultado final. A solução encontrada foi a criação de um *daemon* que, automaticamente, execute o pretendido.

O *daemon* foi implementado, mais uma vez, utilizando a linguagem Perl, com o uso do módulo *App::Daemon*[7]. O uso deste módulo torna o processo de desenvolvimento de um *daemon* mais simples já que esconde todo o processo de gestão de processos.

O processo do *daemon* coresponde a esperar que exista um novo pedido (para isso irá verificar uma fila de espera implementada no sistema de ficheiros) e, quando recebe um novo, recolhe o conjunto de instruções que deve executar. Depois de executadas, o resultado final é criado e é alterado um ficheiro JSON que indica o estado do pedido como terminado.

Para que isto possa funcionar é necessário que o módulo que implementa o serviço *batch* dê a indicação ao *daemon* que deve executar um novo pedido e dê indicação ao utilizador de onde terá acesso ao resultado pretendido. Para isso foi implementado o seguinte sistema de diretorias:

[7] Ver https://metacpan.org/release/App-Daemon.

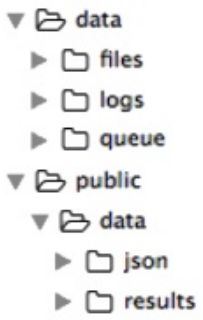


Figura 4.8: Estrutura de diretorias utilizada pelo *daemon*.

Esta estrutura de diretorias está dividida em duas grandes partes, a que deve ser acedida pelo utilizador (inserida na diretoria 'public') e a que é gerida pelo *daemon* internamente (inserida na pasta 'data').

A pasta *data* está então dividida em:

- **files:** Esta diretoria não está somente orientada à gestão do *daemon* mas, normalmente, está sim associada ao envio de ficheiros requeridos em certos pedidos mais demorados. Ou seja, é aqui que serão inseridos ficheiros enviados pelo utilizador para serem lidos através das instruções executadas pelo *daemon*.

- **logs:** Aqui estarão os *logs* criados pelo *daemon* aquando da execução de um novo pedido. Para além de permitir perceber o histórico de realização de pedidos, permite também fazer testes e *debugging* do conjunto de instruções que o administrador inserirá no módulo.

- **queue:** Como o nome indica, esta diretoria conterá uma fila de espera de ficheiros a serem corridos. Para isso ser possível sem grandes dificuldades, basta que esses ficheiros contenham, como nome, uma marca temporal criando assim uma fila First In First Out (FIFO). O conteúdo de cada ficheiro é basicamente código específico que irá ser executado pelo daemon automaticamente.

A pasta *public*, como já indicado, conterá os ficheiros que estarão disponíveis ao utilizador final. Na pasta *json* estará, desde o início do processo, um ficheiro que indica se o pedido está a ser executado ou, caso ele já tenha terminado, que o resultado já está disponível, juntamente com uma referência ao caminho onde o resultado final poderá ser acedido. Este resultado estará disponível na diretoria *results* que é responsável por armazenar este tipo de informação. O JSON que é criado e que é a ponte que mostra ao utilizador o estado do pedido tem o seguinte formato:

```
{"status":"done","result":["data/results/1441820096/target-source.dmp",
    "data/results/1441820096/source-target.dmp"]}
```


Com esta estrutura, o utilizador só tem de ir verificando o estado (campo *status*) e quando este indicar que está acabado, recolher os URI fornecidos (necessitam só que se adicione a parte inicial do Uniform Resource Locator (URL)) e utilizar a informação que estes contêm.

4.5 Componentes extra

Para além dos componentes essenciais implementados, a plataforma oferece também alguns extras com a intenção de tornar o seu uso mais simples. Por um lado, a criação de uma interface web para tornar a plataforma mais *user-friendly* ao nível de consulta e teste dos serviços disponibilizados, e por outro, a criação de uma interface Perl que permite utilizar a plataforma de forma rápida e eficaz.

4.5.1 Interface web

Para além de responder a pedidos REST, a plataforma fornece uma interface que permite ao utilizador saber quais os serviços disponibilizados a cada momento, a sua descrição, os custos de uso, os parâmetros obrigatórios e opcionais, entre outros. Para além disso, a interface permite também testar alguns dos serviços a partir de um pequeno formulário, que recebe os parâmetros e executa um pedido HTTP POST, fornecendo o resultado final de seguida.

Tal como na figura 4.9 se pode constatar, a interface é simples. Tem uma barra de navegação que contém ligações para a criação de um *token* de autenticação, para ter acesso ao histórico e ao número de moedas que ainda pode usar. Também está acessível um pequeno tutorial de como usar a plataforma.

Do lado esquerdo existem ligações diretas a cada serviço, à secção da demonstração onde se pode utilizar os serviços via web e, por fim, os contactos. A figura também mostra um caso exemplo, o atomizador, com a informação completa sobre o serviço para que o utilizador tenha conhecimento do seu modo de funcionamento e do seu custo.

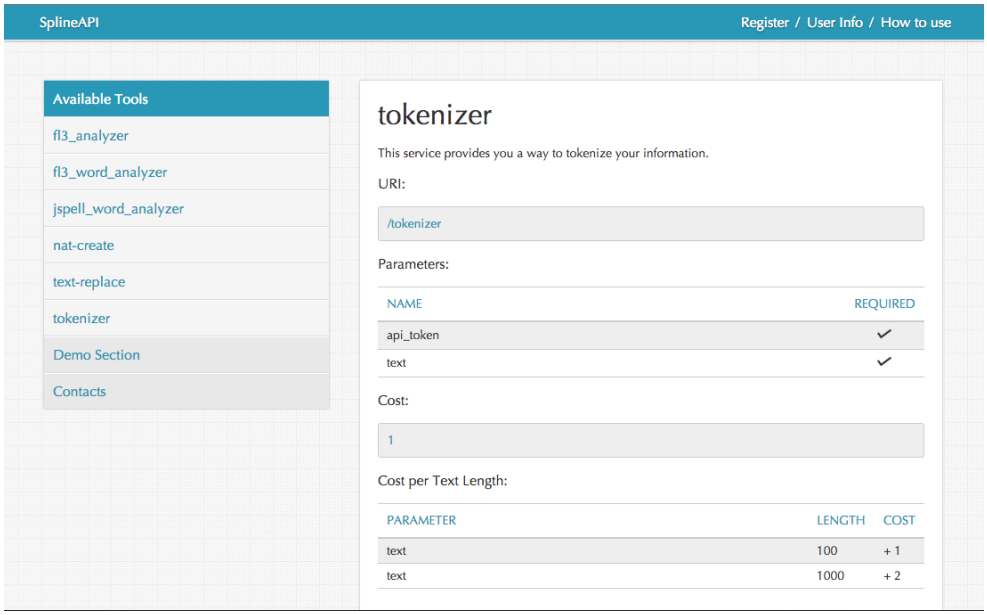


Figura 4.9: Captura de ecrã da interface web.

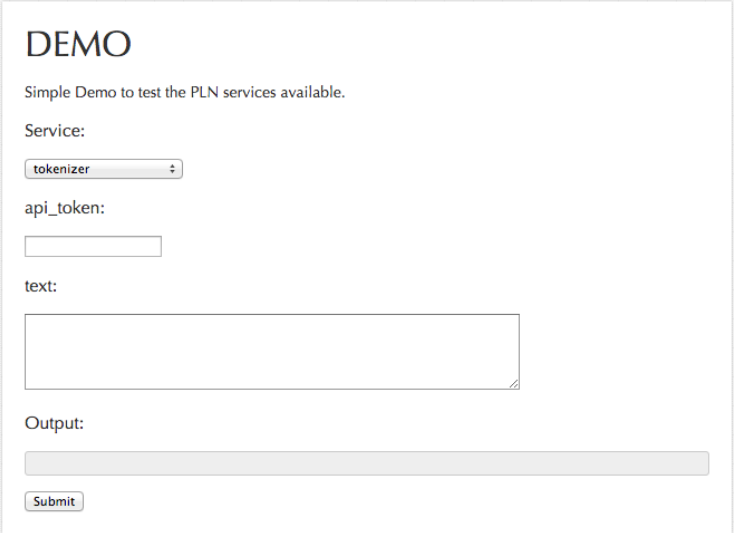


Figura 4.10: Captura de ecrã do formulário de demonstração.

O formulário de demonstração é uma maneira simples de se poder testar os serviços sem ter de se programar nada. É obviamente um formulário web que, quando submetido, envia o conteúdo que o utilizador preencheu num pedido HTTP POST para o servidor que devolverá, na caixa do output, o resultado final desejado.

Este formulário contém elementos de input variáveis, ou seja, dependendo do serviço escolhido no *select* ele troca, de imediato, os tais elementos consoante o tipo de informação a que corresponde. Ou seja, se o serviço *atomizador* recebe um texto, ele sabe que o tipo indicado é *textarea* e cria um componente HTML dessa natureza e o mesmo para qualquer outro tipo de dados que as variáveis sejam, limitado pelos elementos que existem.

4.5.2 Interface Perl

Juntamente com a plataforma desenvolvida também se desenvolveu um módulo Perl capaz de se adaptar aos serviços disponíveis, quase como se esses mesmos serviços estivessem implementados diretamente no dito módulo.

Ou seja, o que o módulo faz é enviar um pedido à plataforma para obter o nome de todos os serviços que estão atualmente disponíveis na plataforma. A partir dessa lista de nomes, faz uma instalação, em tempo de execução, de uma função para cada nome de serviço utilizando o módulo *Sub::Install* [8].

Com o uso deste módulo, é possível a partir de um *array* associativo criar das mais variadas funções atribuindo o nome do serviço como nome da nova função e executando, em código, um novo pedido POST com os argumentos que serão enviados e depois devolve descodificando o JSON resultado.

Com este método, o programador que use esta interface não tem sequer de se preocupar com a criação do pedido à plataforma usando-a como se de funções normais se tratassem.

[8] Ver https://metacpan.org/release/Sub-Install.

5 Resultados e Aplicações

Este capítulo é dedicado à análise do trabalho desenvolvido. Para isso foram criados vários serviços usando o formato XML definido, e analisada a facilidade com que essa tarefa é, ou não, possível. Além disso foram também realizados testes de usabilidade, no caso concreto, um estudo à forma como os serviços disponibilizados podem ser (facilmente) usados a partir de diferentes linguagens.

5.1 Casos de estudo

Os casos de estudo foram escolhidos tendo em conta diferentes funcionalidades que se pretendiam testar e demonstrar, nomeadamente:

- Funcionalidades baseadas em interfaces Perl, ou seja, envolvendo só o carregamento de *packages* e respetiva invocação via código;
- Funcionalidades baseadas em comandos UNIX por não existir bibliotecas que as implementem, ou por ser mais simples o uso via linha de comando do que a análise a uma API não documentada;
- Funcionalidades que recorrem a um modelo *batch*, já que obrigam a uma gestão diferente do processamento do pedido.

5.1.1 FreeLing::Tokenizer

O módulo `FreeLing::Tokenizer` é uma interface com a biblioteca *FreeLing* para a atomização de texto, ou seja, a divisão de um texto em átomos ou *tokens*: tipicamente palavras, sinais de pontuação, elementos não textuais como e-mails, URI, datas, entre outros. Esta divisão

tem como base regras que são definidas e incluídas no FreeLing. Estas regras são, normalmente, expressões regulares e listas de palavras (como abreviaturas) que são usadas para processar o texto.

Para disponibilizar o atomizador no Spline optou-se por usar uma interface Perl desenvolvida sobre a biblioteca do FreeLing, de nome `Lingua::FreeLing3`[1].

De seguida apresenta-se cada um dos blocos necessários para a definição da interface do servidor com a ferramenta (neste caso o atomizador). A versão completa de um destes ficheiros é apresentada em anexo.

```
1.  <meta batch="false">
2.      <tool>FreeLing</tool>
3.      <name>Tokenizer</name>
4.      <route>tokenizer</route>
5.      <parameters>
6.          <parameter required="true" name="text" type="textarea">
7.              <description>The text to be tokenized.</description>
8.          </parameter>
9.      </parameters>
10.     <definition>This service provides you a way
11.     to tokenize your information.</definition>
12.     <cost>1</cost>
13.     <text_costs>
14.         <text_cost field="text" length="100" cost="1"/>
15.         <text_cost field="text" length="1000" cost="2"/>
16.     </text_costs>
17. </meta>
```

Este conjunto de elementos XML é, basicamente, a caracterização ou descrição do serviço, descrevendo, por exemplo:

1. O nome da rota (e portanto, nome do serviço) que é definido pelo elemento `route`, e neste caso, definido como **tokenizer** como se verifica na linha 4;

2. É definido apenas um parâmetro, obrigatório, de nome 'text' nas linhas 6 a 8. O seu conteúdo deverá ser o texto que se pretende atomizar;

3. É definido o custo de uso do serviço que é de uma unidade por cada invocação à qual é

[1] Ver https://metacpan.org/release/Lingua-FreeLing3

acrescido um custo variável: 1 se o texto ultrapassar o tamanho de 100 caracteres; e 2 se ultrapassar o patamar de 1000 caracteres. Isto é constatado nas últimas linhas do XML.

```
1.  <implementation>
2.      <packages>
3.          <package>Lingua::FreeLing3</package>
4.      </packages>
5.      <main lang="perl">
6.          my $pt_tok = Lingua::FreeLing3::Tokenizer->new("pt");
7.          my $tokens = $pt_tok->tokenize($text, to_text => 1);
8.          return $tokens;
9.      </main>
10. </implementation>
```

Nesta parte do XML é descrita a implementação do serviço. Em primeiro lugar é necessário carregar-se a biblioteca do FreeLing que fornecerá as funções responsáveis pela execução do pedido e isso acontece na terceira linha. Depois, quanto ao código Perl, fará a atomização do texto enviado e retornará uma estrutura de dados que será depois transformada em JSON.

```
1.  <tests>
2.      <test>
3.          <param name="text">Eu serei separado.</param>
4.          <code>is($result->[0], 'Eu', "Testar o primeiro token");</code>
5.          <code>is(@{$result}, 4, "Testar o tamanho");</code>
6.      </test>
7.  </tests>
```

Por último, foi necessário fazer um pequeno teste que permitisse verificar que o serviço estava a funcionar. O teste foi simples, escolheu-se *'Eu serei separado.'* como texto de entrada e testou-se, depois do serviço ter sido executado, se a primeira palavra era o *'Eu'* e se, ao todo, se obteve quatro *tokens* separados.

No que toca à sintaxe dos testes, utilizou-se a função *is* que não é mais que um validador da igualdade (ou desigualdade) dos seus dois primeiros argumentos. É possível, opcionalmente, definir um terceiro argumento que é o nome do teste unitário em questão. A função irá imprimir, em formato TAP, se houve, ou não, sucesso no teste (ou seja, se o resultado obtido foi o esperado, ou não).

Exemplo de uso

Para se utilizar este serviço, basta criar um pedido desta forma:

```
curl -i -F "text=Eu estou prestes a ser tokenizado."
-F "api_token=MAIlGopQUt" http://spline.di-um.org/tokenizer
```

O resultado deste pedido terá a seguinte estrutura:

```
["Eu","estou","prestes","a","ser","tokenizado","."]
```

5.1.2 FreeLing::Analyzer

Este serviço permite analisar morfologicamente um texto. Este análise é, obviamente, dependente de uma determinada língua, que deve ser indicada, já que a análise morfológica de texto usa dicionários e modelos de língua treinados especificamente para cada um.

Mais uma vez foi usada a interface disponibilizada pelo módulo `Lingua::FreeLing3`. Dado um texto, o serviço irá apresentar a análise morfológica de cada palavra numa estrutura de dados JSON, que poderá posteriormente interpretado pelo cliente.

```
1.  <meta batch="false">
2.      <tool>FreeLing</tool>
3.      <name>Analyzer</name>
4.      <route>fl3_analyzer</route>
5.      <parameters>
6.          <parameter required="true" name="text" type="textarea">
7.              <description>The text to be analyzed.</description>
8.          </parameter>
9.          <parameter required="false" name="ner" type="number">
10.             <description>Named-entity recognition.</description>
11.             <default>0</default>
12.         </parameter>
13.     </parameters>
14.     <definition>This service provides you a way
15.     to analyze your text.</definition>
16.     <cost>2</cost>
```

```
17.     <text_costs>
18.         <text_cost field="text" length="1000" cost="1"/>
19.     </text_costs>
20. </meta>
```

Com este extrato de XML, no que toca à sua meta-informação, obtêm-se as seguintes características do serviço:

1. O serviço é invocado a partir da rota **fl3_analyzer**, visível na linha 4;
2. Quanto aos parâmetros, este serviço apresenta dois em que um deles é obrigatório (linhas 6-8) e outro opcional (linhas 9-12). O obrigatório é obviamente o texto que irá ser analisado enquanto que o opcional é um argumento booleano de reconhecimento de entidades mencionadas que, por defeito, será atribuído o valor 0 ou falso;
3. Tem um custo fixo de duas unidades, tal como indica na linha 16, e apresenta apenas uma restrição ao custo que adiciona uma unidade ao custo total se o tamanho do texto ultrapassar os 1000 caracteres.

```
1.  <implementation>
2.      <packages>
3.          <package>FL3 'pt'</package>
4.      </packages>
5.      <main lang="perl">
6.          my %options = ( lang => 'pt', ner => $ner );
7.          my $morph = Lingua::FreeLing3::MorphAnalyzer->new(
8.          $options{lang},
9.          NERecognition => $options{ner},);
10.
11.         my $tokens = tokenizer->tokenize($text);
12.         my $sentences = splitter->split($tokens);
13.         $sentences = $morph->analyze($sentences);
14.         $sentences = hmm->analyze($sentences);
15.
16.         my $result;
17.         foreach (@$sentences) {
18.             my @words = $_->words;
19.             foreach my $w (@words) {
20.                 push @$result, {word=>$w->form,
21.                     pos=>$w->tag, lemma=>$w->lemma};
```

```
22.             }
23.         }
24.         return $result;
25.     </main>
26. </implementation>
```

A linha 3 mostra o carregamento do módulo `FL3` que corresponde a uma forma abreviada do carregamento do módulo `Lingua::FreeLing3`. Este módulo permite que se lhe passe um único argumento que será a língua usada durante a análise morfológica. Posteriormente esta língua poderia ser indicada como um parâmetro do serviço, mas atualmente optou-se por a manter pré-definida.

A implementação utiliza as funcionalidades fornecidas pela biblioteca e manipula essa informação criando uma estrutura simples que contém, para cada palavra, a própria palavra, o seu lema e um indicador da sua classe gramatical (*part-of-speech*).

```
1.  <tests>
2.      <test>
3.          <param name="text">Eu vou ser analisado.</param>
4.          <code>is $result->[1]{'lemma'}, 'ir', "Testar lema";</code>
5.      </test>
6.  </tests>
```

Para testar se o serviço foi bem implementado, instalado e incorporado na plataforma criou-se então um teste simples que recebe um texto específico que, neste caso, é *'Eu vou ser analisado.'* e verifica-se se a segunda palavra, tem, como lema, a palavra *'ir'*.

Exemplo de uso

Para se utilizar este serviço, basta criar um pedido desta forma:

```
curl -i -F "text=Eu vou ser analisado."
-F "api_token=MAIlGopQUt" http://spline.di-um.org/fl3_analyzer
```

O resultado deste pedido terá a seguinte estrutura:

```
[{"pos":"PP1CSN00","word":"Eu","lemma":"eu"},
{"lemma":"ir","pos":"VMIP1S0","word":"vou"},
{"word":"ser","pos":"VMN0000","lemma":"ser"},
{"pos":"VMP00SM","word":"analisado","lemma":"analisar"},
{"pos":"Fp","word":".","lemma":"."}]
```

5.1.3 FreeLing::WordAnalyzer

O Word Analyzer é uma variante do serviço anterior, mas que está preparado para etiquetar apenas uma palavra. A principal vantagem tem que ver com a eficiência deste serviço já que não irá tentar realizar algum tipo de desambiguação de sentido dependendo do contexto. Isto significa que cada palavra pode ter uma ou mais classes gramaticais (por exemplo, a palavra 'para' que tanto pode ser uma preposição como uma forma do verbo 'parar') dependendo do contexto. Esta funcionalidade retorna, então, todas essas classes para a palavra indicada.

```
1.  <meta batch="false">
2.      <tool>FreeLing</tool>
3.      <name>WordAnalyzer</name>
4.      <route>fl3_word_analyzer</route>
5.      <parameters>
6.          <parameter required="true" name="word" type="text">
7.              <description>The word to be analyzed.</description>
8.          </parameter>
9.          <parameter required="false" name="ner" type="number">
10.             <description>Named-entity recognition.</description>
11.             <default>0</default>
12.         </parameter>
13.     </parameters>
14.     <definition>This service provides you a way
15.     to analyze a single word.</definition>
16.     <cost>3</cost>
17. </meta>
```

Também no que toca à informação característica do serviço, o *Word Analyzer* é bastante semelhante ao analisador pois, apesar de partilharem a ferramenta que disponibiliza as funcionalidades, apresentam o mesmo estilo de argumentos, apesar de que aqui não é um texto mas sim uma só palavra tal como se pode verificar na linha 6. O *token* de referência aqui é o **fl3_word_analyzer**.

Quantos aos custos, como é um processo um pouco mais pormenorizado apresenta um custo mais elevado mas, como a ideia é só usar-se palavras, não apresenta qualquer custo adicional relacionado com o tamanho do conteúdo enviado.

```
1.  <implementation>
2.      <packages>
3.          <package>FL3 'pt'</package>
4.      </packages>
5.      <main lang="perl">
6.          my %options = ( lang=>'pt' );
7.          my $fl3_morph_pt = Lingua::FreeLing3::MorphAnalyzer->new('pt',
8.              ProbabilityAssignment  => 0, QuantitiesDetection    => 0,
9.              MultiwordsDetection    => 0, NumbersDetection       => 0,
10.             DatesDetection         => 0, NERecognition          => $ner,
11.         );
12.
13.         my $words = tokenizer($options{lang})->tokenize($word);
14.         my $analysis = $fl3_morph_pt->analyze(
15.             [Lingua::FreeLing3::Sentence->new(@$words)]);
16.         my @w = $analysis->[0]->words;
17.         my $result =  $w[0]->analysis(FeatureStructure=>1);
18.
19.         my @final;
20.         foreach (@$result) {
21.             my $pos = $_->{tag};
22.             my $lemma = $_->{lemma};
23.             my $cat = '_';
24.             $cat = lc($1) if $pos =~ m/^(\w)/;
25.             push @final, {lemma=>$lemma, pos=>$pos,
26.                 cat=>$cat, word=>$word};
27.         }
28.
29.         return [@final];
30.     </main>
31. </implementation>
```

Quanto à implementação do módulo, a ideia é também semelhante ao analisador só que, neste caso, trabalha-se ao nível da palavra e não do texto. Apesar do módulo permitir saber todas as classes gramaticais que uma palavra pode ter, a biblioteca do FreeLing é um pouco limitada a esse nível e foi preciso obter-se uma solução menos ortodoxa para funcionar sem, no

entanto, o utilizador saber desses pormenores. Também se adicionou, ao resultado final, um campo responsável por indicar a que categoria gramatical a palavra está inserir se estiver naquele determinado contexto, tal como se pode constatar na linha 24 e 25.

```
1.  <tests>
2.      <test>
3.          <param name="word">trabalhei</param>
4.          <code>ok($result->[0]{'lemma'} eq 'trabalhar',
5.              "Testar lemma");</code>
6.          <code>ok($result->[0]{'cat'} eq 'v', "Testar categoria");</code>
7.          <code>ok($result->[0]{'pos'} eq 'VMIS1S0', "Testar POS");</code>
8.      </test>
9.  </tests>
```

Para testar este serviço, analisou-se todos os itens que este devolve. Ou seja, dá-se a palavra *'trabalhei'* e testa-se, no final da execução do pedido, se se obtém *'trabalhar'* como lema da palavra, *verbo no pretérito perfeito* como classe gramatical.

Note-se que a definição das etiquetas de *part-of-speech* (como VMIS1S0) são definidas pelo FreeLing, e o utilizador deverá consultar a documentação do FreeLing para perceber o seu significado.

Exemplo de uso

Para se utilizar este serviço, basta criar um pedido desta forma:

```
curl -i -F "word=fui" -F "api_token=MAIlGopQUt"
http://spline.di-um.org/fl3_word_analyzer
```

O resultado deste pedido terá a seguinte estrutura:

```
[{"pos":"VMIS1S0","word":"fui","lemma":"ir","cat":"v"},
{"pos":"VMIS1S0","word":"fui","cat":"v","lemma":"ser"}]
```

5.1.4 Jspell::WordAnalyzer

Este serviço faz exatamente o mesmo trabalho que o analisador de palavras anteriormente referido executa mas, neste caso, a ferramenta que fornece a interface e a sua própria maneira de

resolver o problema é diferente. Neste caso em particular é o `Lingua::Jspell`[2] e a sua interface que tornam possível ao utilizador, para o mesmo objetivo, poder escolher qual a ferramenta que preferem.

Neste caso, tal como para o caso do Freeling, o pedido é executado e o resultado é tratado e retorna algo muito parecido estruturalmente mas em conteúdo possivelmente diferente.

```
1.  <meta batch="false">
2.      <tool>Jspell</tool>
3.      <name>WordAnalyzer</name>
4.      <route>jspell_word_analyzer</route>
5.      <parameters>
6.          <parameter required="true" name="word" type="text">
7.              <description>The word to be analyzed.</description>
8.          </parameter>
9.      </parameters>
10.     <definition>This service provides you a way
11.     to analyze a single word.</definition>
12.     <cost>2</cost>
13. </meta>
```

Comparando com o analisador de palavras que a plataforma fornece através do FreeLing, este serviço apresenta algumas pequenas diferenças:

1. A rota do serviço é **jspell_word_analyzer**, como indicado no elemento *route*;
2. Só contém um único parâmetro que é obviamente a palavra que irá ser analisada;
3. O custo é menor pois trata-se de uma ferramenta mais especializada e eficiente no que toca a este tipo de ação. No entanto, como já explicado, também não apresenta custos adicionais por tamanho pois não tem muito sentido neste contexto.

```
1.  <implementation>
2.      <packages>
3.          <package>Lingua::Jspell</package>
4.      </packages>
5.      <main lang="perl">
6.          my $jspell_dict = Lingua::Jspell->new("pt_PT");
7.          my %options = ( lang=>'pt' );
```

[2]Ver https://metacpan.org/release/Lingua-Jspell/

```
8.          my $result;
9.          foreach ( $jspell_dict->featagsrad($word) ) {
10.             my ($pos, $lemma) = split /:/, $_;
11.             my $cat = '_';
12.             $cat = lc($1) if $pos =~ m/^(\w)/;
13.             push @$result, {lemma=>$lemma, pos=>$pos,
14.                 cat=>$cat, word=>$word};
15.         }
16.         return $result;
17.     </main>
18. </implementation>
```

Para implementar esta funcionalidade foi necessário recorrer à interface Perl disponibilizada pela ferramenta que permite ter funções muito próprias e especializadas para o tratamento deste tipo de ação. Dá para verificar também que o código é muito mais simples do que o módulo derivado do FreeLing mas, no final, os dados finais terão a mesma estrutura que este. No entanto, ao depender da ferramenta, esse resultado pode conter informação num formato diferente pois depende diretamente das regras que a ferramenta tem como base.

```
1.  <tests>
2.      <test>
3.          <param name="word">trabalhei</param>
4.          <code>ok($result->[0]{'lemma'} eq 'trabalhar',
5.              "Testar lemma");</code>
6.          <code>ok($result->[0]{'cat'} eq 'v', "Testar categoria");</code>
7.          <code>ok($result->[0]{'pos'} eq 'VIP1S', "Testar POS");</code>
8.      </test>
9.  </tests>
```

Este teste é exatamente igual ao teste do analisador de palavras do FreeLing. Com isto prova-se que, apesar de se usar ferramentas diferentes, os resultados finais são os mesmos como deveria ser. Contudo, verifica-se também que, sendo diferentes, apresentam *tokens* diferentes para caracterizar o mesmo como se pode concluir pelo valor esperado do *part of speech*.

Exemplo de uso

Para se utilizar este serviço, basta criar um pedido desta forma:

```
curl -i -F "word=fui" -F "api_token=MAIlGopQUt"
http://spline.di-um.org/jspell_word_analyzer
```

O resultado deste pedido terá a seguinte estrutura:

```
[{"lemma":"ir","cat":"v","pos":"VIP1S","word":"fui"},
{"cat":"v","lemma":"ser","pos":"VIP1S","word":"fui"}]
```

5.1.5 Text::TextReplacer

O objetivo deste serviço é tão simples como substituir uma expressão num texto por outra. Para isto ser possível, recorre-se a expressões regulares responsáveis pelo *matching* de parcelas de texto que serão substituídas por outra parcela especificada pelo utilizador. Este *matching* pode acontecer múltiplas vezes no mesmo texto fazendo com que este serviço seja uma espécie de *Encontra e Substitui (ou Find & Replace)*.

Este serviço não depende de qualquer interface em específico mas sim da instalação do Perl e algumas das suas funcionalidades. Serve também como exemplo para casos onde seja necessário o uso de comandos UNIX ou quando é simplesmente conveniente devido à performance.

```
1.  <meta batch="false">
2.      <tool>Text</tool>
3.      <name>TextReplace</name>
4.      <route>text-replace</route>
5.      <parameters>
6.          <parameter required="true" name="text" type="textarea">
7.              <description>The text to be edited.</description>
8.          </parameter>
9.          <parameter required="true" name="old_expr" type="text">
10.             <description>The expression to be replaced.</description>
11.         </parameter>
12.         <parameter required="true" name="new_expr" type="text">
13.             <description>The new content.</description>
14.         </parameter>
15.     </parameters>
16.     <definition>This service provides you a way
17.     to substitute expressions in a text.</definition>
18.     <cost>2</cost>
19. </meta>
```

Como podemos constatar pelo XML acima e explicação do serviço, este apresenta:

1. **text-replace** como *token* a ser invocado;
2. um conjunto de três parâmetros, o texto que irá ser alterado (linha 6 a 8), a expressão regular que indicará o que deve ser substituído (linha 9 a 11) e o conteúdo a trocar (linha 12 a 14);
3. um custo de 2 unidades sem custos adicionais baseado no tamanho do texto enviado por opção.

```
1.  <implementation>
2.      <main lang="perl">
3.          my %res = ();
4.          $text =~ s/'/'\\''/g;
5.          $old_expr =~ s/'/'\\''/g;
6.          $new_expr =~ s/'/'\\''/g;
7.          $res{result} = `echo \"$text\" |
8.              perl -p -e 's/$old_expr/$new_expr/g'`;
9.          return \%res;
10.     </main>
11. </implementation>
```

Em termos de código é tão simples como enviar um comando UNIX executar (linha 7/8) mas, como se está a trabalhar diretamente com o sistema operativo, é necessário tomar algumas precauções quanto à injeção de comandos (ou *command-injection*). Para prevenir esse problema, foram adicionadas algumas linhas ao código, mais propriamente da 4 à 6, que manipulam aspas e pelicas evitando que o utilizador seja capaz de adicionar comandos que interfiram com o bem-estar da máquina.

```
1.  <tests>
2.      <test>
3.          <param name="text">Vou ser substituido.</param>
4.          <param name="old_expr">Vou ser</param>
5.          <param name="new_expr">Ja fui</param>
6.          <code>ok($result->{result} eq "Ja fui substituido.",
7.              "Testar troca");</code>
8.      </test>
9.  </tests>
```

Para testar este serviço é preciso algo simples. Envia-se um texto específico, uma expressão regular que, neste caso, pode ser uma palavra ou um conjunto de palavras e, por fim, um conjunto de palavras a ser substituído. Com isto, como tem no exemplo, se tivermos um texto *'Vou ser substituído.'* e quisermos trocar *'Vou ser'* por *'Já fui'* verifica-se que no final o texto será *'Já fui substituído.'*.

Exemplo de uso

Para se utilizar este serviço, basta criar um pedido desta forma:

```
curl -i -F "text=Este texto vai ser editado alterando uma expressão
por outra. A expressão é velho." -F "old_expr=velho" -F "new_expr=novo"
-F "api_token=MAIlGopQUt" http://spline.di-um.org/text-replace
```

O resultado deste pedido terá a seguinte estrutura:

```
{"result":"Este texto vai ser editado alterando uma expressão por outra.
A expressão é novo."}
```

5.1.6 NATools::NATCreate

Como já foi referido, os serviços proporcionados pela ferramenta *NATools* têm como finalidade o processamento de corpus paralelo. O `natcreate` é uma destas funcionalidades e o seu objetivo é criar um *NATools Corpora Object*, ou seja, um objeto que representa uma corpora num formato manipulável pela ferramenta.

Um objeto destes é basicamente uma diretoria que contém:

- Um ficheiro de configuração que guarda a meta-informação;
- O corpus em questão e os seus índices;
- As base de dados *n-gram*;
- Os dicionários de tradução probabilística.

Este último item é então o que o serviço irá retornar ao utilizador e consiste num par de ficheiros: *source-target.dmp* e *target-source.dmp*. O primeiro é um dicionário probabilístico da

língua de origem para a língua de destino, e o segundo é um dicionário probabilístico da língua de destino para a de origem.

```
1.  <meta batch="true">
2.      <tool>NATools</tool>
3.      <name>NATCreate</name>
4.      <route>nat-create</route>
5.      <parameters>
6.          <parameter required="true" name="file" type="file">
7.              <description>The file to be treated.</description>
8.          </parameter>
9.      </parameters>
10.     <definition>NATCreate is responsible for creating probabilistic
11.     translation dictionaries as a NATools functionality.</definition>
12.     <cost>5</cost>
13.     <text_costs>
14.         <text_cost field="file" length="1000" cost="1"/>
15.     </text_costs>
16. </meta>
```

Para se utilizar este serviço é necessário ter-se em atenção que o envio do ficheiro (que tem de estar no formato próprio para ser interpretado) é, obviamente, obrigatório e que mais nenhum argumento é necessário.

O seu custo de uso é de 5 unidades e quando o ficheiro de *input* ultrapassar os 1000 caracteres é adicionada 1 unidade ao custo total da operação (linhas 12 a 15).

Para além disso, e como é evidente, consegue-se identificar também que é um serviço *batch* na primeira linha e, portanto, o próprio XML terá características próprias de um serviço desta categoria.

```
1.  <implementation>
2.      <main lang="bash">
3.          nat-create -tokenize -id=public/data/results/$ID -tmx $file
4.      </main>
5.      <output>
6.          <file>data/results/$ID/target-source.dmp</file>
7.          <file>data/results/$ID/source-target.dmp</file>
8.      </output>
9.  </implementation>
```

Como se pode observar no XML acima, a implementação é bastante simples apesar da complexidade por trás do serviço. O que se pretende que este faça é que execute um comando *bash*, fornecido pelo *NATools*, que faça então o pretendido (linha 3). Os argumentos que este comando leva são variados mas, neste caso, envia-se a *flag tokenize*, a *flag id* e o ficheiro em questão.

O argumento `-tokenize` serve para forçar o programa a fazer uma divisão dos textos e o argumento `-id` serve para atribuir um nome que, neste caso, será basicamente a diretoria onde o conteúdo será guardado. Encontra-se também algo que não se usa em serviços normais, o $ID. Este elemento vai tornar possível guardar informação para diferentes utilizadores e este $ID é um marcador temporal que obriga a que este valor seja sempre diferente de pedido para pedido.

Por fim, tal como normalmente em qualquer serviço *batch*, encontra-se a secção do output. Esta secção serve para indicar os ficheiros e pastas que o serviço retornará ao utilizador. Ou seja, o utilizador com esta informação poderá diretamente ir buscar a informação que o serviço retornou. No caso de se indicar uma diretoria, a plataforma gera automaticamente um ficheiro *zip* que será então o que será disponibilizado ao utilizador para poder descarregar e usar.

Quanto aos testes ao serviço, como este é *batch*, optou-se pela não geração destes. Isto porque cada serviço deste género permite enviar múltiplos ficheiros e outros argumentos, obriga a manipular várias entidades e, para além de a nível de performance poder não ser rápido, pode também dificultar alguma mudança posterior que se queira fazer.

No entanto, o teste deste serviço em particular é simples. Em primeiro lugar ter-se-ia de enviar o ficheiro e o *token* da API num pedido POST *multipart*. Depois disso, a plataforma devolveria um JSON com o caminho para se aceder a outro ficheiro JSON que contém a informação desejada e o estado do pedido. Enquanto o estado do pedido não indicar que está acabado, teria de haver um ciclo que de tempo em tempo enviasse um novo pedido para ver se já estava pronto. Quando isso acontece, recolhe a informação enviada e faz o *download* dos dados.

Exemplo de uso

Para se utilizar este serviço, basta criar um pedido desta forma:

```
curl -i -F "file=@Constituicao-pt.es.tmx"
-F "api_token=MAIlGopQUt" http://spline.di-um.org/nat-create
```

O resultado deste pedido terá a seguinte estrutura:

```
{"answer":"data/json/1439690278.json","status":"processing"}
```

Com o URL obtido no JSON acima, tem-se acesso a um JSON gerado automaticamente com a seguinte forma:

```
{"status":"done","result":["data/results/1441820096/target-source.dmp",
    "data/results/1441820096/source-target.dmp"]}
```

Enquanto o *status* estiver a 'processing' o cliente deverá esperar até que os ficheiros indicados em 'result' sejam populados com o resultado final.

5.2 Testes de Usabilidade

Para testar o uso da plataforma ao nível das funcionalidades dividiram-se os testes em três grupos:

1. Casos de uso com serviços implementados de forma simples, ou seja, envia o pedido e recebe logo a resposta;
2. Casos de uso com serviços implementados com *batch* que obrigam a um passo intermédio pois a conexão é fechada a meio do processo;
3. Caso de uso a partir do cliente criado para a linguagem de programação Perl.

5.2.1 Serviços Síncronos

Como referido anteriormente, testou-se, no papel do utilizador final, os serviços mais básicos que a plataforma fornece. Para isso utilizou-se duas linguagens de programação diferentes do Perl para mostrar que, para além de perfeitamente fazível, é bastante simples de implementar e obter a solução resultado. As linguagens escolhidas foram o PHP que, para além de ser uma linguagem *scripting*, é mais centrada no mundo web e também o Java que, sendo uma das linguagens mais populares, apresenta outro tipo de paradigma, o de objetos.

Comparando esta solução com outra sem a plataforma como auxiliar, nota-se perfeitamente que o esforço é menor pois o utilizador não terá de perceber as interfaces das ferramentas, que

são normalmente complicadas, e não terá de tratar a informação que estas devolvem. A única coisa que terão de perceber é o resultado final que normalmente vem numa estrutura bem definida e de simples perceção. Quanto ao tempo e performance, apesar da dependência do HTTP ser sempre um pouco custosa, verifica-se facilmente que este peso não é suficientemente influente num possível mau estado do programa.

Tokenizer em Java

O código em Java para qualquer serviço "on-the-fly" baseia-se na criação, para cada argumento pretendido, de um objeto que representa um par nome-valor (neste caso, *BasicNameValuePair*) e, com o auxílio de algumas bibliotecas de *IO* o envio de um pedido HTTP POST à plataforma para o serviço em questão. No final, executa-se a leitura, linha a linha, do resultado que, por ter formato JSON, permite à função retornar uma lista de palavras atomizadas.

```
private JSONArray tokenizer(String text, String api_token)
    throws Exception{
    String url = "http://spline.di-um.org/tokenizer";
    List<NameValuePair> params = new ArrayList<NameValuePair>();
    params.add(new BasicNameValuePair("text", text));
    params.add(new BasicNameValuePair("api_token", api_token));

    HttpClient client = new DefaultHttpClient();
    HttpPost post = new HttpPost(url);
    post.setEntity(new UrlEncodedFormEntity(params));
    HttpResponse resp = client.execute(post);

    BufferedReader rd = new BufferedReader(
        new InputStreamReader(resp.getEntity().getContent()));
    StringBuffer result = new StringBuffer();
    String line = "";
    while ((line = rd.readLine()) != null) {
        result.append(line);
    }
```

```
    JSONArray json = new JSONArray(result.toString());
    return json;
}
```

Tokenizer em PHP

Em PHP também é bastante simples de implementar o código que utiliza este tipo de serviço. Com o auxilio das funcionalidades fornecidas pela biblioteca *curl*, que são automaticamente carregadas com o PHP, só é necessário criar-se um *array* associativo que contenha os argumentos e, com isto, enviar-se o pedido desejado. No final a função abaixo, tal como no Java, devolve a lista de palavras a partir da descodificação do JSON.

```
function tokenizer($text, $api_token){
    $fields = array(
        'text' => $text,
        'api_token' => $api_token
    );
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_URL
        , "http://spline.di-um.org/tokenizer");
    curl_setopt($ch, CURLOPT_POST, 1);
    curl_setopt($ch, CURLOPT_POSTFIELDS, $fields);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    $response = curl_exec ($ch);
    curl_close ($ch);
    return json_decode($response);
}
```

5.2.2 Serviços Assíncronos

À margem das opções escolhidas para os casos mais básicos, usou-se também o PHP e o Java para testar os serviços *batch*. Estes serviços, a nível de implementação, apresentam um pouco mais de dificuldade pois contêm o tal passo intermédio que é preciso ter-se em atenção. Um serviço *batch* retorna sempre uma representação em JSON que rapidamente fornece a ligação

para o resultado final da execução. Com essa ligação, obtêm-se um novo ficheiro JSON que contém a informação desejada (normalmente ligações para ficheiros gerados entretanto pelo serviço) e contém também um sinalizador que indica se o processo está a decorrer ou não. Assim, com este mecanismo, o utilizador sabe quando o processo está acabado e, só nessa altura, pode tentar obter o resultado final.

A diferença entre esta solução apresentada e uma possível sem o uso da plataforma é grande e obviamente desfavorável à segunda. Considerando que são processos normalmente longos, que podem consumir bastantes recursos computacionais e que a plataforma fornece uma forma de evitar que a conexão entre cliente e servidor se prolongue ou até que quebre devido a *timeouts*, a escolha é simples tanto a nível de performance como de esforço. A nível de esforço, o utilizador só tem de perceber a estratégia implementada e esta é semelhante para todos os serviços desta categoria. Quanto à performance, empurra-se grande parte do esforço computacional para o lado do servidor obrigando unicamente o utilizador a esperar pelo sinal que indica que o pedido foi executado.

Uso do NATCreate em Java

Serviços batch complicam um bocado o código em Java a utilizar-se principalmente porque, com a necessidade de contemplar o upload de ficheiros, é necessário implementar recorrendo ao uso de *multipart*. No entanto o Java fornece objetos que facilitam esta tarefa tal como o *MultipartEntity* que acaba por fazer de *array* associativo de elementos em forma de ficheiro ou texto.

O facto deste tipo de pedido envolver uma estratégia que obriga a uma espera ativa até este estar terminado, também torna o código maior mas não mais difícil pois é só necessário um ciclo que vá verificando se acabou e depois, através do que foi recebido, fazer download da informação em questão. No entanto, em cada iteração deste ciclo pode-se adicionar um método de *sleep* pois evita o consumo de recursos do cliente e diminui a carga do servidor.

Esta função abaixo descrita faz todo este processo e devolve um *JSONArray*, ou seja uma lista, que contém os endereços de rede para os ficheiros gerados pelo serviço.

```
private JSONArray nat_create(File f, String api_token)
    throws Exception{
    String url = "http://spline.di-um.org/nat-create";
```

```
    String res = "";
    JSONArray result = null;

    HttpClient client = new DefaultHttpClient();
    HttpPost post = new HttpPost(url);
    MultipartEntity mpEntity = new MultipartEntity();
    ContentBody cbFile = new FileBody(f);
    mpEntity.addPart("file", cbFile);
    ContentBody cbToken = new StringBody(api_token);
    mpEntity.addPart("api_token", cbToken);

    post.setEntity(mpEntity);
    HttpResponse response = client.execute(post);
    HttpEntity resEntity = response.getEntity();
    if (resEntity != null) {
        JSONObject json = new JSONObject(
            EntityUtils.toString(resEntity));
        res = (String)json.get("answer");
        boolean flag = true;
        while(flag){
            String content =
                getContents("http://spline.di-um.org/"+res);
            JSONObject j = new JSONObject(content);
            if(((String)j.get("status")).equals("processing"))
                { Thread.sleep(10000); continue; }
            else{
                flag = false;
                result = j.getJSONArray("result");
            }
        }
    }
    return result;
}
```

Uso do NATCreate em PHP

Em PHP, o uso de serviços *batch* apresentam quase a mesma dificuldade técnica na sua implementação comparando com os síncronos. A única grande diferença é que, quando se obtém o resultado final proveniente das funcionalidades *curl*, é necessário executar o tal ciclo que vai verificando se o pedido está finalizado. Quando isto acontecer é só necessário ir buscar a informação através dos URL fornecidos pelo serviço. A função, tal como no Java, retorna a lista de URL para os ficheiros pretendidos pelo utilizador.

```
function nat_create($file, $api_token){
    $fields = array(
        'file' => '@'.$file,
        'api_token' => $api_token
    );

    $ch = curl_init();
    curl_setopt($ch, CURLOPT_URL,
        "http://spline.di-um.org/nat-create");
    curl_setopt($ch, CURLOPT_POST, 1);
    curl_setopt($ch, CURLOPT_POSTFIELDS, $fields);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    $response = curl_exec ($ch);
    curl_close ($ch);

    $obj = json_decode($response);
    $ans = $obj->{"answer"};
    $done = 0;
    $result = NULL;
    while($done == 0){
        $jsonData = json_decode(
            file_get_contents('http://spline.di-um.org/'.$ans));
        if($jsonData->{"status"} == "processing")
            { sleep(10); continue; }
        else{
            $result = $jsonData->{"result"};
```

```
            $done = 1;
        }
    }
    return $result;
}
```

5.2.3 Usando a interface Perl

A interface Perl fornecida pela plataforma é útil pois esconde grande parte do processo de implementação que o utilizador teria de fazer. Com isto, o utilizador só tem de carregar o módulo dos serviços e utilizar a função que desejar. Esta função é criada automaticamente a partir do *token* que identifica o serviço tornando simples o seu uso.

De seguida apresenta-se um exemplo extremamente simples, o uso do atomizador usando esta interface Perl.

```
use Spline::Services;

sub do_tokenizer{
    my ($text, $api_token) = @_;
    my %fields = (
        text => $text,
        api_token => $api_token,
    );
    my $result = tokenizer(%fields); # route here
    return $result
}
```

6 Conclusões

Este trabalho surgiu das dificuldades que um utilizador de ferramentas de processamento de linguagem natural tem para as poder instalar, configurar e aprender a usar. Estes obstáculos passam pela complexidade nas instalações e configurações, devido essencialmente a dependências aplicacionais que não são devidamente documentadas, e passam também pela grande dependência de um sistema operativo específico. Para além disto, a documentação nem sempre é suficiente ou fácil de interpretar, o que torna o uso correto e eficaz destas ferramentas muito complicado.

Para resolver este problema analisou-se a viabilidade da construção de um serviço REST que permitisse o uso destas ferramentas, escondendo a sua instalação e configuração, que seria necessária apenas uma vez, pelo administrador desse mesmo serviço. Pretendia-se que esse serviço fosse parametrizável, de modo a que a adição de novos serviços não obrigasse à alteração do código do próprio servidor aplicacional, mas apenas a criação de um módulo ou *plug-in*.

Com esta solução em mente, decidiu-se a implementação de um servidor capaz de carregar, dinamicamente, módulos descritos em Perl. Para isso, definiu-se um conjunto de funções específicas que cada um desses módulos terá de implementar para poder funcionar como um serviço REST. Neste caso, este conjunto de funções constitui uma DSL capaz de descrever a interface do servidor com cada uma das ferramentas.

Foi então implementado um servidor usando a linguagem de programação Perl, e a *framework* web *Dancer2*, que deteta os módulos ou serviços disponíveis e carrega-os dinamicamente. O uso desta linguagem e da *framework* tornou o desenvolvimento mais ágil, tendo sido possível dedicar tempo a futuros problemas.

Não demorou até que contratempos fossem encontrados: alguns serviços a disponibilizar não são capazes de apresentar resultados de forma suficientemente rápida para que o protocolo HTTP e o próprio TCP não dê *timeout*. Além disso, se alguns serviços demoram horas, não seria de todo adequado ter a ligação cliente/servidor ativa durante todo esse tempo. Este problema levou

à necessidade de uma estratégia de serviço em *batch*, ou *background*. Para isso foi criado um serviço, independente da interface web, que processa uma fila de pedidos. Os pedidos demorados são, então, adicionados à fila e serão executados assim que for possível. Quando os resultados estiverem prontos o utilizador será avisado.

No sentido de tornar a escrita de módulos para novas ferramentas mais simples foi modelado um XML Schema que descreve os detalhes de cada serviço, desde o nome da rota em que será disponibilizado, até testes, parâmetros obrigatórios, entre outros. Este XML é usado para, automaticamente, gerar o módulo Perl necessário para suportar o serviço via REST. Esta abstração permite que utilizadores menos familiarizados com a linguagem de programação Perl possam escrever rapidamente interfaces com diferentes ferramentas. Este XML permite, ainda, construir um sítio web em que cada serviço é devidamente documentado.

Sistematizando, as contribuições deste trabalho são:

- Um servidor REST de serviços descritos de forma modular;
- Um serviço baseado em fila para o processamento de pedidos de forma assíncrona.
- Uma linguagem de descrição de serviços, quer como uma DSL sobre a linguagem Perl, quer como dialeto XML;
- Vários módulos implementados para diferentes tipos de serviço;
- Um cliente Perl para a plataforma, capaz de se adaptar de acordo com os serviços disponíveis, tornando o uso da interface REST transparente para o utilizador, quase que como utilizasse uma biblioteca local. Esta abordagem, embora implementada em Perl, pode ser facilmente imitada noutras linguagens de programação dinâmicas;
- Disseminação desta abordagem na comunidade científica, através da publicação de um artigo [Vieira et al., 2015], a ser publicado em breve, na sua versão alargada, no volume 563 da série *Communications in Computer and Information Science*, intitulado *Languages, Applications and Technologies*.

6.1 Trabalho futuro

Como em qualquer projeto com um tempo limitado, existem sempre coisas que poderiam ser acrescentadas, alteradas ou melhoradas, de modo a ficar mais consistente e/ou eficiente.

Alguns dos pontos aqui descritos não foram resolvidos por pura e simplesmente terem sido relegados, dada a importância de outros aspetos, ou porque não faziam parte do centro temático definido inicialmente ou porque o seu desenvolvimento não seria possível no intervalo temporal de uma dissertação de mestrado.

Existem várias funcionalidades que, dependendo da evolução da plataforma e promovendo a facilidade do seu uso, podem vir a ser adicionadas. Estas são:

- Alteração da estratégia de testes utilizada, não a baseando na estrutura Perl da resposta obtida, mas usando uma linguagem de *query* sobre JSON que defina cada um dos testes (como as linguagens JsonPath ou JSONiq);
- Inserção de novos serviços de modo a popular mais a plataforma tornando-a mais completa e podendo testar outros tipos de ferramentas, que poderiam levantar problemas que não tivessem sido contemplados;
- Criação de uma interface gráfica que permita a geração do XML descritor de cada módulo, tornando possível a implementação de interfaces por utilizadores pouco aptos à tecnologia;
- Possibilidade de mudança de motor de base de dados para um que lide melhor com problemas tais como concorrência nos pedidos. Com o tempo e com o aumento de utilizadores pode ser necessário fazer a mudança para um motor com uma filosofia diferente tal como o PostgreSQL ou MySQL;
- Melhoramento do sistema de erros tornando a plataforma mais robusta;
- Realização de mais testes ao nível da performance com o objetivo de descobrir se é possível tornar a plataforma ainda mais rápida e eficiente.

Bibliografia

J.J. Almeida and José Carlos Ramalho. Xml::dt a perl down-translation module. In *XML-Europe'99, Granada - Espanha*, 1999.

Gregory R Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys (CSUR)*, 23(1):49–90, 1991.

Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

Charles P Bourne. Frequency and impact of spelling errors in bibliographic data bases. *Information Processing & Management*, 13(1):1–12, 1977.

Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1, 2000. URL http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

Charles Brooks, Murray S Mazer, Scott Meeks, and Jim Miller. Application-specific proxy servers as http stream transducers. In *Proceedings of the Fourth International World Wide Web Conference*, pages 539–548, 1995.

Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (wsdl) 1.1, 2001. URL http://wsdl2code.googlecode.com/svn/trunk/03-Literature/WSDL/wsdl.1.1..pdf.

David D Clark and David L Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM SIGCOMM Computer Communication Review*, volume 20, pages 200–208. ACM, 1990.

Douglas Comer and David L Stevens. *Internetworking With Tcp/Ip*. Prentice-Hall, 2003.

Daniël de Kok and Harm Brouwer. Natural language processing for the working programmer, 2011. URL http://www.nlpwp.org/nlpwp.pdf.

Mark Jason Dominus. *Higher-Order Perl: Transforming Programs with Programs*. Morgan Kaufmann, 2005.

Roy Thomas Fielding. *Representational State Transfer (REST)*. PhD thesis, University of California, Irvine, 2000. URL https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *Software Engineering, IEEE Transactions on*, 24(5):342–361, 1998.

David Garlan and Mary Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, 1994.

Djoerd Hiemstra. Using statistical methods to create a bilingual dictionary. Master's thesis, University of Twente, 1996.

John Hutchins. The history of machine translation in a nutshell. *Retrieved December*, 20:2009, 2005. http://www.hutchinsweb.me.uk/Nutshell-2005.pdf.

Daniel Jurafsky and James H. Martin. *An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition Paperback*. Prentice Hall, 2008.

Shawn Knight. Google translate used by 200 million people each month. *TechSpot*, 2012. http://www.techspot.com/news/48373-google-translate-used-by-200-million-people-each-month.html.

Robert Krovetz. Viewing morphology as an inference process. In *Proceedings of the 16th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 191–202. ACM, 1993. http://people.scs.carleton.ca/~armyunis/projects/KAPI/Krovetz.pdf.

Paul Krzyzanowski. Remote procedure calls. https://www.cs.rutgers.edu/~pxk/416/notes/15-rpc.html, 2010. Department of Computer Science, Hill Center, Busch Campus, Rutgers University. Acedido em 08 jan 2015.

E.D Liddy. Natural language processing. In *Encyclopedia of Library and Information Science, 2nd Ed*. NY. Marcel Decker, Inc., 2001.

Inderjeet Mani. Summarization evaluation: An overview, 2001. http://research.nii.ac.jp/ntcir/workshop/OnlineProceedings2/sum-mani.pdf.

Inderjeet Mani and Mark T Maybury. *Advances in automatic text summarization*, volume 293. MIT Press, 1999.

Prakash M Nadkarni, Lucila Ohno-Machado, and Wendy W Chapman. Natural language processing: an introduction. *Journal of the American Medical Informatics Association*, 18(5): 544–551, 2011.

Object Management Group. Corba component model 4.0 specification. Specification Version 4.0, Object Management Group, April 2006. URL http://www.omg.org/docs/formal/06-04-01.pdf.

Ricardo Ramos de Oliveira. *Avaliação de manutenibilidade entre as abordagens de web services RESTful e SOAP-WSDL*. PhD thesis, Universidade de São Paulo, 2012. URL http://www.teses.usp.br/teses/disponiveis/55/55134/tde-24072012-164751/publico/RRrevisada.pdf.

Viviane Orengo and Christian Huyck. A stemming algorithmm for the portuguese language. In *String Processing and Information Retrieval, International Symposium on*, pages 0186–0186. IEEE Computer Society, 2001. http://homes.dcc.ufba.br/~dclaro/download/mate04/Artigo%20Erick.pdf.

Lluís Padró. Analizadores multilingües en freeling. *Linguamática*, 3(2):13–20, 2012. http://nlp.lsi.upc.edu/papers/padro11.pdf.

Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. big'web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814. ACM, 2008.

James L Peterson. Computer programs for detecting and correcting spelling errors. *Communications of the ACM*, 23(12):676–687, 1980.

John Prager, Eric Brown, Anni Coden, and Dragomir Radev. Question-answering by predictive annotation. In *SIGIR '00 Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 184–191, 2000.

Alberto M. Simões and J. João Almeida. Natools – a statistical word aligner workbench. *Procesamiento del Lenguaje Natural*, 31:217–224, 1998.

Alberto Manuel Simões and José João Almeida. Jspell.pm: Um módulo de análise morfológica para uso em processamento de linguagem natural. *Actas da Associação Portuguesa de Linguística*, pages 485–495, 2001. http://repositorium.sdum.uminho.pt/bitstream/1822/638/1/jspell.pm.pdf.

Efstathios Stamatatos, Nikos Fakotakis, and George Kokkinakis. Automatic extraction of rules for sentence boundary disambiguation. In *Proceedings of the Workshop on Machine Learning in Human Language Technology*, pages 88–92, 1999.

Alexis Sukrieh. *Dancer2::Manual - A gentle introduction to Dancer2*, 2013. URL http://search.cpan.org/~sukria/Dancer2-0.10/lib/Dancer2/Manual.pod.

Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 173–180. Association for Computational Linguistics, 2003. http://delivery.acm.org/10.1145/1080000/1073478/p173-toutanova.pdf.

Nuno Vieira, Alberto Simões, and Nuno Carvalho. SplineAPI: A REST API for NLP services. In José-Luís Sierra-Rodríguez, José Paulo Leal, and Alberto Simões, editors, *IV Symposium on Languages, Applications and Technologies*, pages 101–110, 2015. ISBN 978-84-606-8762-7.

Joseph Weizenbaum. *Computer power and human reason: From judgment to calculation.* WH Freeman & Co, 1976.

Alec Wolman, Geoff Voelker, Nitin Sharma, Neal Cardwell, Molly Brown, Tashana Landray, Denise Pinnel, Anna Karlin, and Henry Levy. Organization-based analysis of web-object sharing and caching. In *the 2nd USENIX Conference on Internet Technologies and Systems (USITS)*, 1999.

Apêndices

1. Exemplo da DSL para o serviço de tokenização do *FreeLing*

```
package Spline::FreeLing::Tokenizer;
use 5.018002;
use strict;
use warnings;
use JSON;
use utf8;
use Encode qw(decode_utf8

my %index_info = (
  hash_token => 'tokenizer',
  parameters => {
    api_token => {
      description => 'The token to be indentified',
      required => 1,
      type => 'text',
    },
    text => {
      description => 'The text to be tokenized',
      required => 1,
      type => 'textarea',
    },
  },
  description => 'This service provides you a way to
  tokenize your information.',
```

```
  cost => 1,
  text_cost => {
    text => [[100,1],[1000,2],],
  },
);

sub get_token { return $index_info{hash_token}; }

sub get_info { return \%index_info; }

sub cost_function{
  my ($input_params) = @_;
  my $cost_result = 0;

  for my $param (keys %{$index_info{text_cost}}){
    my $text_length = "";
    if($index_info{parameters}{$param}{type} eq 'file'){
      $text_length = -s "$input_params->{$param}";
    }
    else{
      $text_length = length($input_params->{$param});
    }
    for my $pair (@{$index_info{text_cost}{$param}}){
      if($text_length >= int($pair->[0])){
        $cost_result += $pair->[1];
      }
    }
  }

  my $final_cost = $cost_result + $index_info{cost};
  return $final_cost;
}

sub param_function {
```

```
  my ($input_params) = @_;
  my $flag = 1;
  for my $param (keys %{$index_info{parameters}}){
    if ($index_info{parameters}{$param}{required} == 1){
      $flag = 0 if (!exists($input_params->{$param}));
    }
    if ($index_info{parameters}{$param}{default}){
      if(!exists($input_params->{$param})){
        $input_params->{$param} =
        $index_info{parameters}{$param}{default};
      }
    }
  }
  return $flag;
}

sub main_function {
  my ($input_params) = @_;
  my $tokens = _freeling_tokenizer($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;
__END__
```

2. Exemplo do XML para a geração do serviço de tokenização do *FreeLing*

```
<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>This service provides you a way to tokenize
            your information.</definition>
        <cost>1</cost>
        <text_costs>
            <text_cost field="text" length="100" cost="1"/>
            <text_cost field="text" length="1000" cost="2"/>
        </text_costs>
    </meta>
    <implementation>
        <packages>
            <package>Lingua::FreeLing3</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 tokenize your text.</header>
        <header title="synopsis">Just load the Spline main
        package to your script or send a POST request directly
        to the Spline platform using the tokenizer service
        provided.</header>
        <header title="description">This module provides a way
        to tokenize the text sent by the user. It is required
        the text information and the Spline token to use
        this functionality.</header>
    </documentation>
</service>
```