Luís Gomes*, Ruben Branco, João Silva, and António Branco

Open and Inclusive Language Processing

Language Processing Services by PORTULAN to Meet the Widest Needs of CLARIN users

Abstract: As a research infrastructure for human language, the mission of CLARIN is to serve its users and respond to their research needs, in all their diversity of backgrounds and aims, with the appropriate access level to the functionalities of a wide range of language processing tools. Building on solutions designed, matured, and explored at the Portuguese national node PORTULAN CLARIN, the goal of this chapter is to expand on those solutions and, by providing a detailed description of them, to report on how CLARIN has been undertaking its mission in that respect. Hopefully, this will help to further improve what the infrastructure can do for its users and for the advancement of research in the science and technology of language.

Keywords: research infrastructure, language science, language technology, language processing services, web services, PORTULAN CLARIN

Acknowledgment: The results reported here were partially supported by PORTULAN CLARIN — Research Infrastructure for the Science and Technology of Language, funded by Lisboa2020, Alentejo2020 and FCT — Fundação para a Ciência e Tecnologia under the grant PINFRA/22117/2016.

*Corresponding author: **Luís Gomes,** PORTULAN CLARIN and University of Lisbon, Departamento de Informática, Faculdade de Ciências de Lisboa, Lisbon, Portugal, e-mail: luis.gomes@di.fc.ul.pt
Ruben Branco, PORTULAN CLARIN and University of Lisbon, Departamento de Informática, Faculdade de Ciências de Lisboa, Lisbon, Portugal, e-mail: ruben.branco@di.fc.ul.pt
João Silva, PORTULAN CLARIN and University of Lisbon, Departamento de Informática, Faculdade de Ciências de Lisboa, Lisbon, Portugal, e-mail: jsilva@di.fc.ul.pt
António Branco, PORTULAN CLARIN and University of Lisbon, Departamento de Informática, Faculdade de Ciências de Lisboa, Lisbon, Portugal, e-mail: antonio.branco@di.fc.ul.pt

https://doi.org/10.1515/9783110767377-005

1 Introduction

PORTULAN CLARIN Research Infrastructure for the Science and Technology of Language[1] belongs to the Portuguese National Roadmap of Research Infrastructures of Strategic Relevance[2] and is part of the international research infrastructure CLARIN ERIC.[3] Its mission is to support researchers, innovators, citizen scientists, students, language professionals, and general users whose activities draw on research results from the Science and Technology of Language, by distributing scientific resources, supplying technological support, providing consultancy, and fostering scientific dissemination.

In this chapter, we focus in one of these mission lines, namely the provision of technological support, in particular under the form of open and inclusive language processing services. Our goal here is to expand on the solutions designed, matured, and explored at PORTULAN CLARIN and, by providing a detailed description of them, to report on how CLARIN has been undertaking its mission in that respect. We expect that this will help to further improve what the CLARIN infrastructure can do for its users and for the advancement of research in the science and technology of language, specifically in articulation with other chapters in this book, including Hajič et al. (2022), Zinn and Dima (2022), and Kupietz, Diewald, and Margaretha (2022).

Tokenization, part-of-speech tagging, parsing, or concordancing are just a few examples, among many others, of language processing tools that can serve as processing services the users of a research infrastructure for the science and technology of language. In PORTULAN CLARIN, every such web-based language processing service is accessible as an *online service*: users just need to copy the excerpt of interest to be processed from some third-party digital source, paste it into a designed text field, push a button to run the tool, then copy the result that will be displayed, and finally paste it to some digital support. The greatest advantage of this type of user interface is its unsurpassed simplicity, together with the fact that users can see the results of their requests immediately and understand the functionality of the tool at stake. This interface constrains users, however, to work with short inputs only and provides no combinatorial affordance.

In a more evolved user interface, tools are accessible as *file-processing services*. This is the type of interface that has been available through the CLARIN switchboard (Zinn 2018). Users upload files of their choice in a dialog box, push

1 https://portulanclarin.net/
2 https://www.fct.pt/apoios/equipamento/roteiro/index.phtml.en
3 https://www.clarin.eu/content/participating-consortia

the *upload* button below that box, and finally download the returned file with the output. Although they are not provided with any combinatorial affordance here, as in the online services, users are, however, not limited to short inputs, and for most practical purposes most users will not feel limited by the size of the inputs.

In another user interface, language processing tools are available under the modality of a *notebook service*. Notebooks allow users to interleave paragraphs of descriptive text with snippets of code; can be opened in a browser and the code run online by resorting to some non-local server that would otherwise have to be provided locally by the user. As in the file-processing interface, users are no longer limited to short inputs, with the added advantage that now combinatorial affordances are available by adjusting the seed code made available, for which some minimal programming skills are needed.

In yet another user interface that is more demanding in terms of technical skills, a tool can be used as a *web service* through a remote procedure call (RPC) interface. From within a program, written in any programming language of their preference, users can invoke a function to which they pass the input text to be processed and that returns the respective processed output. Like the notebook services, this is also a type of interface that is not yet available through the CLARIN switchboard (see Zinn and Dima 2022). As its greatest advantage, this interface grants users full combinatorial affordance while requiring some minimal programming skills.

This chapter is focused on the workbench with language processing services of PORTULAN CLARIN. For a broader and higher level view of PORTULAN, please refer to Branco et al. (2020).

The remainder of this chapter is organized as follows: In Section 2, we present in more detail the different types of interfaces, mentioned above, as they have been implemented in PORTULAN CLARIN. Then, in Section 3 we will expand on the technical options that were adopted and implemented, and in Section 4 we discuss the current status of the workbench formed by the collection of language processing services made available, before concluding with Section 5.

2 Language processing services for the widest user profiles

2.1 Online services

Every tool in the PORTULAN CLARIN workbench has an *online service* type of interface. This is the central interface for each tool and it serves the following purposes:

1. to allow users to experiment with the tool by changing its input and options and immediately see the effect in the output;
2. to offer one-click usage examples to help users start experimenting with the least amount of effort;
3. to grant access to several forms of documentation;
4. to provide an entry point to the *file-processing* or *web services* interfaces.

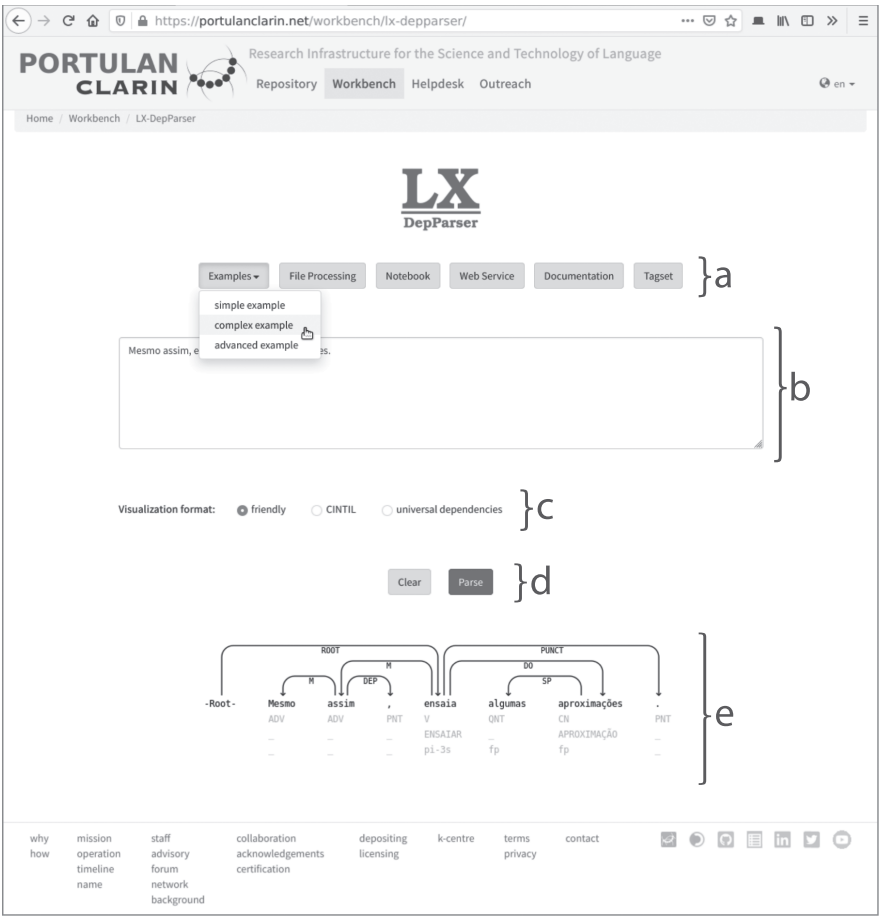

Figure 1: Example of the online service interface. Our guidelines for positioning elements in the interface follows a top-down layout with five groups, (a) to (e), superimposed to this screen shot, and not part of the interface.

As an example, Figure 1 presents the interface of the LX-DepParser tool,[4] which is a prototypical interface for sentence-based text-processing tasks, such as POS tagging, dependency and constituency parsing, or semantic role labelling, etc.

Every online service interface follows the same general layout, which can be sectioned vertically in 5 groups of elements, identified in Figure 1 using letters (a) to (e) for easier reference. In the topmost position, in group (a), we find a row of buttons that give access to examples, the file-processing and web services interfaces, and documentation. The subsequent groups follow the order of user interaction with each of the interface elements: input for the tool is accepted in group (b); options affecting the behaviour and output format of the tool are specified in group (c); processing of current input is started or cleared in group (d); and finally, the results are shown in group (e).

The "Examples" button is the first button on the interface, and thus one of the most prominent, because it provides the best starting point for newly arrived users to start interacting with the tool. Running an example via a simple button click requires no effort from the users, whereas if the common practice of providing examples only as part of the documentation had been followed, users would be required to copy and paste inputs and options from the documentation into the interface. Not only is copying and pasting examples a much more fastidious process than the solution adopted here, but it is also an error-prone one, particularly if the tool has several options affecting its behaviour that need to be changed, which ultimately could hinder the main purpose of examples: to aid users understand what the tool does and how they can use it.

The "File Processing", "Notebook" and "Web Service" buttons each open a dialog interface, which will be described in detail in Sections 2.2, 2.3 and 2.4, respectively.

The documentation button opens a window that will be displayed on top of the online service interface, containing relevant information about the tool, such as:

- a description of the tool, the problem it solves, and the method used;
- the datasets used to train the underlying models, when applicable;
- the tagsets used by the tool, when applicable;
- the input and output formats;
- a user manual or tutorial, where it is justified by the complexity of the tool;
- references to scientific publications describing the tool or its components;
- authorship and contact information;
- acknowledgements;
- licensing terms.

4 https://portulanclarin.net/workbench/lx-depparser/ (based on Branco et al. (2011)).

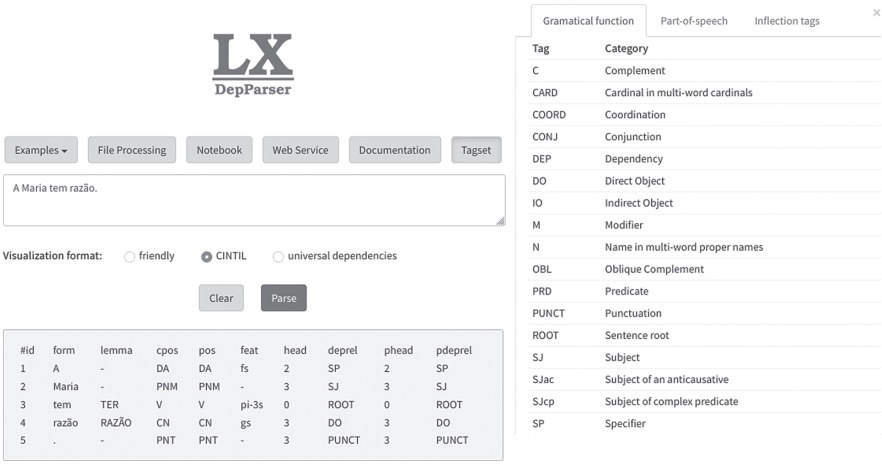

Figure 2: Tagset of LX-DepParser shown side by side with the interface, for user's convenience. Also note that a different output format was selected from the one shown in Figure 1. This interface allows the user to easily compare the output formats available for each tool by looking at the same output encoded in different formats.

The documentation window is presented in a modal form over the tool interface, which means that all page elements not belonging to the documentation window, will appear behind a semi-transparent grey background, allowing users to focus on the documentation without being disturbed by any other elements on the page.

Additionally, because the documentation is often long, a hyperlinked table of contents is automatically inserted at the top of the window, allowing users to jump to any section. A floating button, with an upward-pointing arrow, appears at the top left-hand corner of the screen whenever the document is scrolled down. By clicking this button, users may jump back to the table of contents from any point in the document. These navigation aids are implemented in the interface logic that is shared across all tools in the workbench, contributing to a more uniform and thus less distracting user experience when reading documentation.

Besides being included in the main documentation, tagsets can also be accessed directly by clicking the respective "Tagset" button, the rightmost in group (a) of Figure 1. Once pressed, this button will slightly change its appearance to indicate it has been depressed and a new panel opens on the right-hand side of the interface, sharing half of the horizontal space that was previously fully dedicated to the interface, as shown in Figure 2. Having the tagset shown side by side with the output of the tool is much more convenient to users than having to go back and forth between the documentation view and the interface. To close the

tagset panel, users either press the same button that was used to open it, which will revert to its normal appearance, or they press the "close" button, represented by a cross, at the top right-hand corner.

For some tools, instead of a tagset, this side panel may show other types of referencial documentation, such as a cheat sheet for a query syntax, as is the case for the CINTIL Concordancer tool.[5]

Among the output formats of each tool there is usually one termed *friendly*, which is the default and is specifically targeted at human users, as opposed to being suited to further processing by another automatic tool. This friendly format is often graphical in nature, such as the dependency tree output in Figure 1. By contrast, the other formats are generally textual, even if they encode some form of graph structure, and thus harder to interpret for humans; an example is the tabular output shown within the grey rectangle in Figure 2, which encodes a short sentence and its annotated dependency tree graph.

To conclude this section, it is worth mentioning that the layout presented in Figure 1 is a general guideline for organizing components in online service interfaces, which aims at increasing consistency across the interfaces of different tools, but ultimately, these guidelines should always be overridden as needed for the benefit of the interface.

For example, in the LX-Translator[6] online service, shown in Figure 3, which is an interface for a bi-directional machine translation system, there is not one text input box but two, one for each language, displayed side by side. Each of these boxes is used for both input and output, which breaks the guideline of displaying the output on a dedicated area at the bottom of the page. At the beginning, both text boxes are empty and the user may input text in either one, click the "Translate" button below, and the translation will appear in the other box. For providing examples, we have decided to place one "Example" button below each input/output text box, which breaks another guideline – the one that tells us to place the example button prominently in the top row of buttons. However, by breaking this rule, the new placement makes it obvious which text box will be filled with the respective example input text and which will be the translation direction triggered by each of these example buttons.

5 https://portulanclarin.net/workbench/cintil-concordancer/ (based on Barreto et al. (2006)).
6 https://portulanclarin.net/workbench/lx-translator/ (based on Santos et al. (2019)).

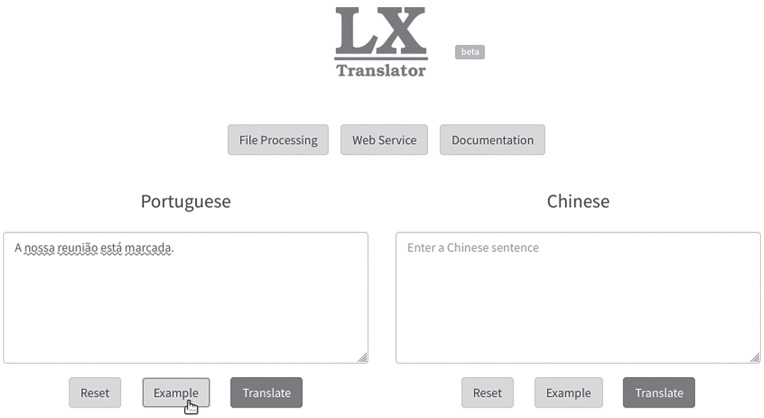

Figure 3: Interface of the LX-Translator online service, illustrating a case where the overall design guidelines may have to be weakened for the benefit of the interface usability, depending on the functionality of the service at stake.

2.2 File-processing services

The file-processing interface, or *fileproc* for short, is a multi-step workflow that is launched by clicking on the "File Processing" button at the top of the online service interface. Figure 4 depicts this workflow, using screenshots of the dialog windows presented to the user at each step.

The first dialog window allows the user to select an input file from their computer to be processed and proceed to upload the file by clicking the "Upload" button. At this point, the workflow takes one of two possible courses, depending on the size of the file that is being uploaded.

Small input files are handled by the path on the left-hand side of Figure 4, and we call these *short* (file-processing) jobs. Large input files are handled by the path on the right-hand side of Figure 4, and we call these *long* jobs. The threshold size, used to determine if a file is to be considered small or large is computed for each tool separately, based on the maximum amount of data that it can process in under two minutes. Further ahead we will discuss the reasoning that led to this specific time threshold.

If the file is small enough that it can be processed in under two minutes, then we consider this to be a short job and processing will start immediately after the file is uploaded. The user is informed of the processing progress through a progress bar, as shown in step 2(a) of Figure 4. As soon as the processing is

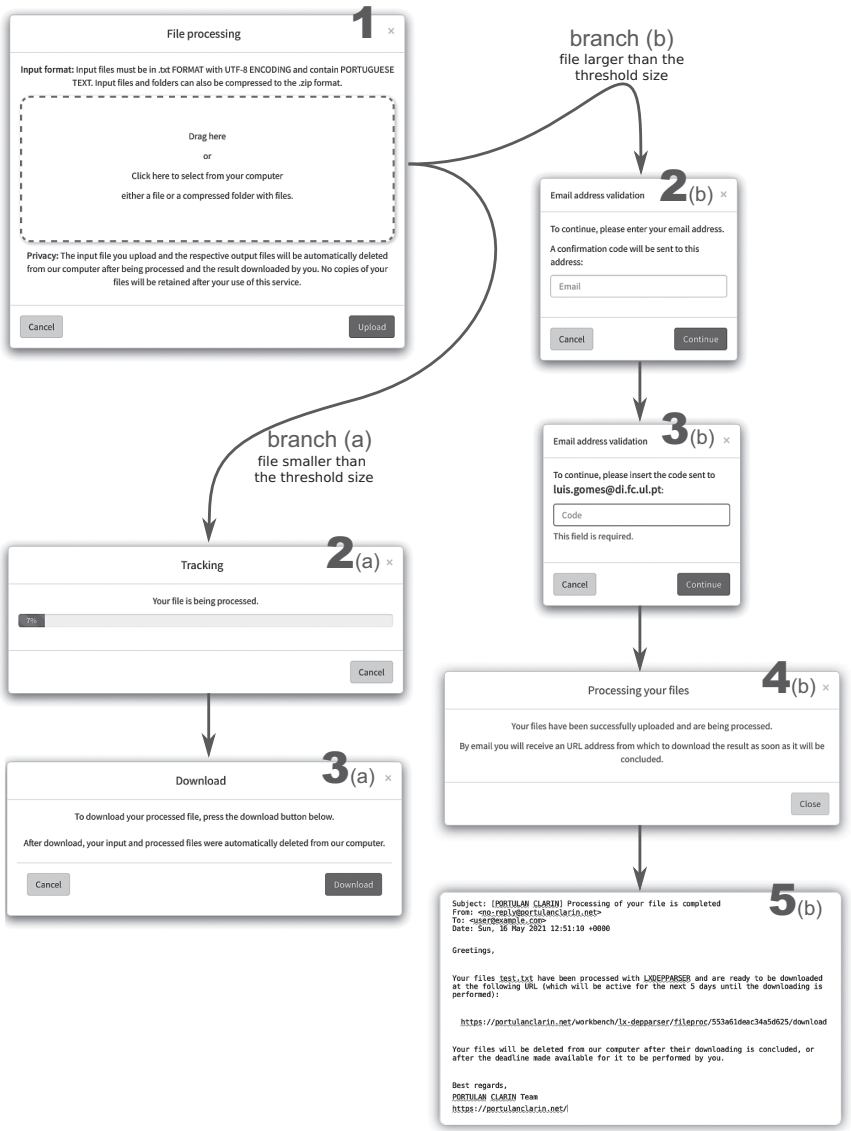

Figure 4: File-processing service interface workflow. Depending on the size of the user supplied input file, the user interaction follows one of two main branches: (a) the file is smaller than a fixed threshold, or (b) otherwise. The threshold size varies from one processing service to another and is determined as the average number of bytes that each specific service can process under two minutes.

complete, the user will be able to download the processed output files by clicking the "Download" button shown in window 3(a) of Figure 4.

Going back to the end of step 1, if the file being uploaded is large enough such that its processing time is estimated to be longer than two minutes, then we consider this to be a long job and the processing will take place in the background, without requiring the user to suspend other activities waiting for its completion. Instead, in this type of job, when the processing is complete, the user will receive an email with a URL for downloading the output file.

Since PORTULAN CLARIN does not require its users to be registered, there is no information about the user requesting this concrete file-processing service. Thus, in order to carry on with the processing, it is necessary to know the email address where the message should be sent. For this purpose, a simple email address validation method was implemented that sends an automatically generated code into the email address specified by the user in the dialog shown in screenshot 2(b), which should then be copied over by the user from the email into a text field, as shown in screenshot 3(b) of Figure 4. Because the codes are randomly generated long strings, if the code inserted by the user matches the one that was sent, we assume that the user has had access to the specified email account and did not guess the code by chance.

Once the user's email address has been validated, the job processing begins and the user is notified that the job has been successfully submitted and that an email message will be sent upon the job's completion. See screenshot 4(b) of Figure 4.

When the processing of a long job finishes, an email like the one shown in the 5(b) screenshot of Figure 4 is sent to the user. The download URL included in the email message will be valid for five days. As soon as the user finishes downloading the output file, both the email address associated with the job and the output file will be deleted from the server (and thus the URL will no longer be valid). If, five days after the email was sent, the user did not download the output file, it will be automatically removed from the server along with the user's email address.

Now that we have considered the two workflow paths, for short and long jobs, let us take a look at the two-minute time threshold which is used to decide whether a file-processing job should be considered short or long. This threshold has been adjusted through experimentation, although in a highly subjective manner because it depends on many factors, including the users themselves. Two minutes is about the point at which we find it is more costly, in terms of inconvenience to users, to require them to go through the extra steps to validate an email address and wait for an email with the URL for downloading the output files, rather than simply wait for the processing to complete.

Compared to the online service interface, presented in detail in the previous section, here in the file-processing mode the user does not have to choose an output format. Instead, we opted to include all output formats in the output file, which will be a zip archive containing one directory for each format. The reasoning for this decision is that the time required by a tool to process the input data largely exceeds the time required to convert the processed output into all available output formats. Thus, not only this is convenient for the users, who do not have to worry about which output format to choose, but it also avoids unnecessary re-processing of the same input data if a user finds out, after a job has been processed into one output format, that a different one is needed.

The accepted formats for the input file will depend on the tool at stake, but in general, the file should be either a UTF-8 encoded plain text file, or a zip archive containing any number of UTF-8 encoded plain text files. In the case of a zip archive, the files may be organized within a directory tree structure, which will be preserved during the processing.

The output file will always be a zip archive, containing several directories, one for each output format. If the input file was a zip archive containing multiple files organized within a directory tree structure, the same structure will be replicated under each output format directory. Otherwise, if a single text file was given as input, then each directory in the output zip archive will contain a single processed output file in the corresponding output format.

2.3 Notebook services

The notebook interface is launched by clicking on the "Notebook" button at the top of the online service interface.

A Jupyter notebook (hereafter *notebook*, for short) is a type of document that contains sections of executable code, called *cells*, interspersed with visualizations of results from the execution of such cells and narrative text with rich formatting (headings, lists, bold, italic, equations, etc.). An example notebook is shown in Figure 5. Notebooks may be written in a tutorial style, embodying the *literate programming* paradigm envisioned by Knuth (1984), which also makes them a valuable tool for teaching. Furthermore, because notebooks may be modified and re-executed interactively, they are also an excellent tool for learning through experimentation.

For several tools in the workbench, the respective notebook service may be explored with only a couple of mouse clicks: a user starts by clicking the "Notebook" button in the tool's online service interface, which brings up a dialog with

relevant information and further options to launch the notebook on free supporting servers, such as the Binder offered by Project Jupyter et al. (2018) or Google.

These notebooks are intended to serve as quick and easy starting points for users to start developing their own experiments, and for that purpose, we believe that very short and artificial code examples would not be the most adequate. Instead, we often include code for downloading and cleaning example data to be processed, code for processing the data with a tool from the workbench via its web services interface, and code for some kind of subsequent analysis of the processed data.

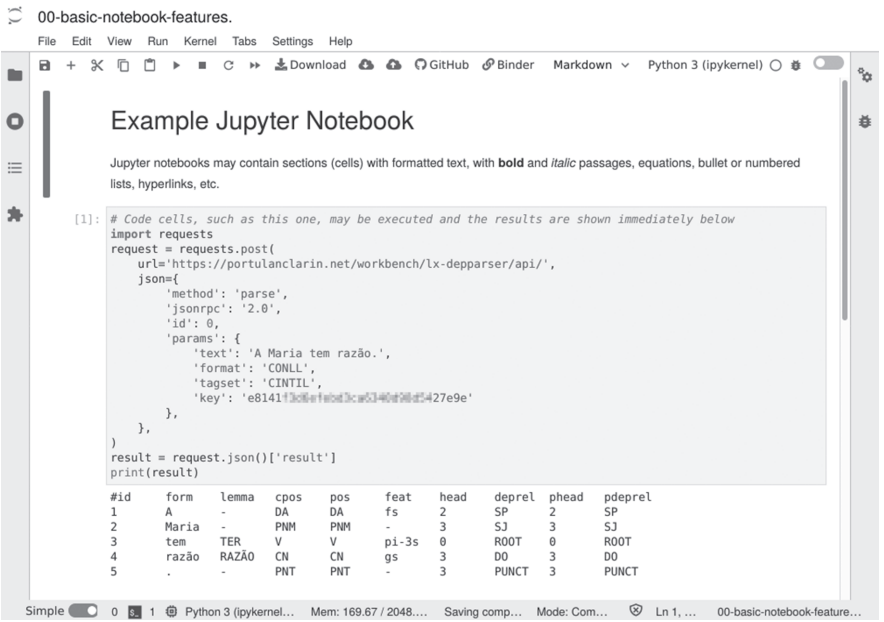

Figure 5: Example notebook illustrating basic features. At the top, there is some text with rich formatting. Within the grey rectangle there is some code. When run, it produces the output that it is displayed in the same page, and which can be input to subsequent code.

With this type of interface with language processing services in PORTULAN workbench, no software needs to be installed on the users' computers: a web browser is all that is needed. By lowering the technical requirements, we believe notebooks will foster users' interest and will help to leverage new research ideas and experimentation.

2.4 Web services

The web services interface is a remote procedure call (RPC) type of interface, through which it is possible to interact with one or several tools in the workbench by means of computer programs. We chose to implement this service using JSON-RPC, which is a lightweight and programming language-agnostic protocol for which implementations are readily available in many programming languages.

The web services interface is available for most tools in the workbench. Exceptions that do not offer this type of interface are, for example, tools that naturally lend themselves more to an interactive usage, through their online service interface, rather than to a data-processing usage scenario. For example, the CINTIL Concordancer[7] and the CINTIL Treebank Searcher[8] are examples of two such tools.

To start using web services, for any given workbench tool that supports them, a user will click the "Web Service" button in the tool's online service interface, which will bring up a dialog as the one shown in Figure 6. This dialog contains detailed information about the requirements that have to be met before this service can be used, as well as a simple and self-contained Python program that can be used as a starting point for users with little programming experience to develop their own programs.

One of the requirements to use a web service is an access key that each user must obtain by clicking the "Request key" button on this dialog. This key is used to implement a basic access control mechanism with the primary goal of preventing any individual user from abusing, either intentionally or inadvertently, the finite computational resources available on PORTULAN CLARIN to serve all its users. By clicking on the "Request key" button, users will go through an email validation process identical to the one required when submitting long file-processing jobs, as described in the previous section. After their email address has been validated, users are sent an email with an access key and information about usage quotas associated with it: the total number of requests allowed, the total number of characters allowed (accumulated over all requests), and the expiry date for the key.

Whenever a user requests a new key using an email address that was used before, if the previous key is still valid (i.e. it has not expired and its usage quotas have not been exhausted), that key is returned in the response email, along with

7 https://portulanclarin.net/workbench/cintil-concordancer/ (based on Barreto et al.(2006)).
8 https://portulanclarin.net/workbench/cintil-treebank-searcher/ (based on Branco et al.(2010)).

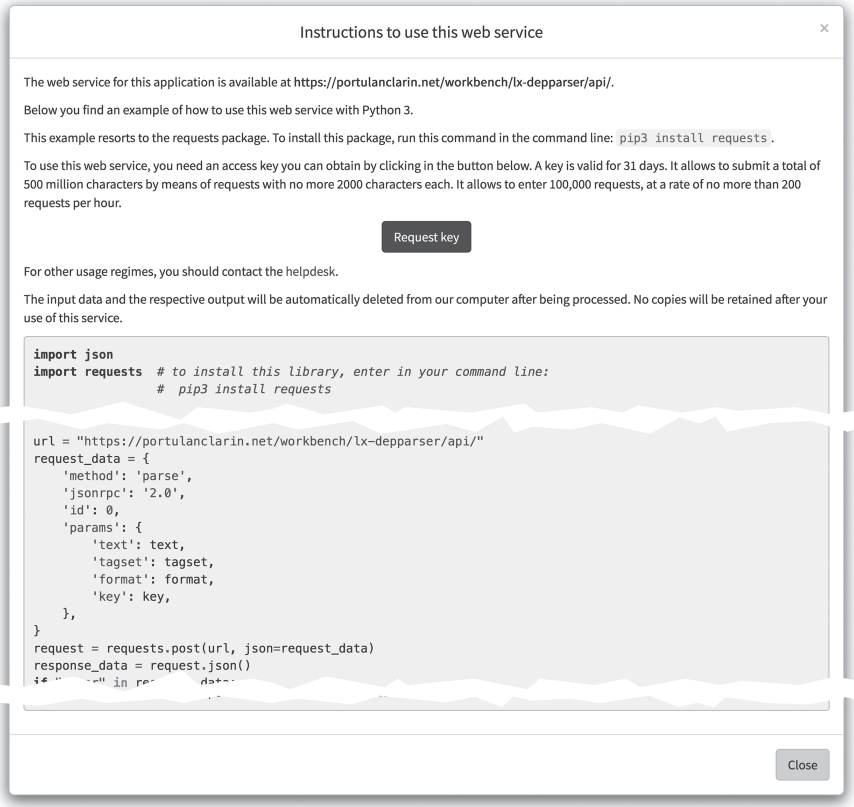

Figure 6: Example web service dialog containing detailed instructions and example Python code (truncated in this screenshot) for using the LX-DepParser web service interface.

the remainder usage quota. Thus, at any point in time, only one valid key is associated with any given email address.

Because any user can have access to several email addresses, this access control mechanism does not prevent a single user from having multiple access keys, each associated with a different address. However, creating new email addresses and requesting access keys requires some effort, which should be enough to discourage fortuitous abuse.

Besides the total number of requests and of characters allowed during the lifespan of a key, there is also a maximum number of requests and characters allowed per hour. If any of these maximum hourly rates are reached, subsequent requests will receive an appropriate error code and message, until enough time has passed since the last successful request such that both hourly rates become lower than their maximum allowed values.

3 Exploring the current stage of technological development

In order to be able to set up a computational infrastructure that seamlessly supports the four different modes of interaction described in the previous sections for dozens of different tools, non-trivial technical options need to be adopted and implemented. These options need to ensure that appropriate levels of factorization can be achieved and that sufficient levels of readability are ensured. We focus here on the design decisions that have the most impact globally.

3.1 HTTP and nginx

The PORTULAN workbench is implemented as a micro-service distributed system with a user-facing HTTP server, a frontend server and several backend servers.

The user-facing HTTP server is the only part of this distributed system that is directly exposed to the internet and it is responsible for negotiating SSL connections with the browser, serving static content such as images, CSS (Cascading Style Sheets) and JavaScript files and acting as a reverse HTTP proxy to the frontend server.

For this HTTP server, we adopted nginx[9] for its clean configuration syntax, low resource usage and excellent performance. SSL certificates are issued by Let's Encrypt,[10] a nonprofit Certificate Authority, and managed through Certbot.[11] From a security perspective, having all HTTP requests served or proxied through a single user-facing HTTP server reduces the attack surface, at least for HTTP protocol-based exploits, and eases security audits.

3.2 Python and Django

We adopted Python as the main programming language, which not only gives one access to an immense array of high-quality libraries and frameworks, and a thriving ecosystem of development tools, but also, since it is an immensely popular and accessible language, ensures that the code base is maintainable, expandable, and accessible by a larger number of people.

9 https://www.nginx.com/
10 https://letsencrypt.org/
11 and https://certbot.eff.org/

The frontend server is implemented as a WSGI-compliant[12] application and is served by the gunicorn server.[13] We adopted the WSGI-compliant Django framework,[14] which promotes code factorization and organization, both essential aspects for large-scale projects such as the PORTULAN workbench.

A Django-based server runs a collection of Django applications,[15] and each application holds code and files for a specific part of the the frontend service as a whole. In the context of the PORTULAN CLARIN's workbench, each tool is implemented as a separate Django application. Additionally, some cross-cutting functionalites of the workbench are implemented as Django applications, such as the workbench index page where all tools are listed, email validation, and CAPTCHA validation.

Mirroring this modular organization, workbench tools and cross-cutting functionalities are developed and maintained in independent Git repositories and packaged as separate Python packages. During deployment, these packages are installed and upgraded with the Python package management tool (pip), based on a requirements file which specifies the exact version of each package to be installed.

Thus, during production, whenever a problem occurs and a bug report is filled in our GitLab[16] service, we know exactly what version of each component was installed at the time when the problem occured. This is crucial for reproducing reported errors and pinpointing their exact source within the code, because the latest development versions of packages may no longer exhibit the same error, either because the problem was fixed as part of a refactorization or because it is being masked by some other change.

At its core, a Django application is a set of views, models, and templates.

- **Views** are functions or methods responsible for handling HTTP requests. The core logic of any Django application is either implemented within views or can be traced to calls made from them.
- **Models** are classes that define the properties and structure of data that needs to be persistent in a database. Through inheritance and dynamic method

12 https://www.python.org/dev/peps/pep-3333/

13 https://gunicorn.org/

14 https://www.djangoproject.com/

15 The word *application* has several meanings in the context of web development and thus prone to generate confusion. A *WSGI application* refers to a whole web application. A *Django application* implements a part of the whole web application, which may be composed of many Django applications.

16 GitLab is an open-source development platform that provides web-based interface for managing Git-based code repositories, a ticket system, and much more. PORTULAN CLARIN hosts a private GitLab server, only accessible to staff members.

resolution, Django provides a *Pythonic* interface to its object-relational-mapper (ORM) for querying, retrieving, inserting, updating, and deleting records from a relational database. Model objects are typically instantiated and manipulated from views.

– **Templates** are, in essence, files containing static HTML code[17] enriched with special syntax describing how and where dynamic content will be inserted. The Django template syntax provides basic control flow structures, such as conditionals and loops, an inclusion mechanism that allows templates to be included as part of other templates, and an inheritance mechanism, allowing templates to inherit and extend functionality from other templates. Templates are typically used within views to generate the HTML to be sent to the browser as the body of an HTTP response.

Taking advantage of class and template inheritance, logic that is shared across all tools in the workbench is factored out, such as CAPTCHA validation, email validation, general interface layout, common components, etc. This factorization speeds up the integration of new tools into the workbench by reducing the amount of new code that has to be written for each of them, and ensuring that each bug needs to be fixed only in one place.

3.3 JavaScript, jQuery, VueJS, and Bootstrap

Equally important in building web applications, the JavaScript code running on the web browser is used to manipulate the structure and content of a page after the initial HTML has been transferred from the server.

Furthermore, by making asynchronous HTTP requests from JavaScript code, web applications can be made smoother and more efficient because only small chunks of data need to be transferred from the server, instead of reloading the entire page. For example, when a user submits a snippet of text to be processed through an online service interface, an HTTP request is sent to the server through JavaScript, containing the snippet to be processed. Likewise, through JavaScript, while the HTTP request is ongoing, a visual activity indicator may be displayed next to the button that was clicked to trigger the request, and thus letting the user know that something is happening as a consequence of the click. As soon as the server replies, the processed result will be inserted in the appropriate place

17 In fact, a template may contain any type of textual content, not only HTML, but this is the most common use for templates.

within the page and the visual activity indicator is removed. All of these page content manipulations are made using JavaScript code. Most of the HTML that makes up the page is transferred only once into the browser, when the user navigates into that page.

We have adopted the jQuery[18] library, which introduces a large set of functionalities that simplify manipulation of HTML elements programmatically. Recently, we have also been progressively adopting the VueJS framework,[19] which provides a new, more efficient, and easier-to-use mechanism to manipulate HTML elements in the browser, and enables component-based code organization and reuse.

For the styling of HTML elements, we adopted the Bootstrap[20] framework which provides a comprehensive, well-documented and easy-to-use set of CSS classes that comply with modern web design requirements, such as being able to adapt to the small screens of mobile devices.

3.4 Backend and containers

Let us now turn our attention to the backend services of the PORTULAN infrastructure. Some tools in the workbench have dedicated backend servers that encapsulate the core logic of the tool. Other tools are directly integrated into the frontend server.

Taking into consideration the architecture and inner workings of WSGI servers, for performance and reliability reasons[21] the Django worker processes should have short startup times and moderate memory usage. Thus, the decision as to whether a tool should be integrated in its own backend server depends on the following conditions:

- if it requires a CPU-heavy or long initialization;
- if it requires a large amount of memory;
- if it is multi-threaded, which becomes a problem if any other tool is not thread-safe;
- if it is not thread-safe, which becomes a problem if any other tool is multi-threaded;

18 https://jquery.com/

19 https://vuejs.org/

20 https://getbootstrap.com/

21 The two main reasons are: (1) the WSGI server may dynamically spin up/down Django worker processes depending on the number of concurrent HTTP requests and (2) the WSGI server may restart each Django worker after it serves a pre-configured maximum number of requests.

- if it is implemented in a programming language other than Python and any of the following is true:
  - it does not offer a command line interface;
  - its initialization time is not negligible in comparison to the time it takes to process a typical input unit (e.g. a snippet of text);
- if it is no longer being actively developed or maintained. The reasons underlying this condition are quite different from the previous ones, and will be detailed below, when we discuss the need for containers.

If one or more of the above conditions is true for any given tool, then it should be integrated into a separate backend server that exposes the tool functionality over an appropriate JSON-RPC or XML-RPC interface. We adopted these two standard RPC protocols because they are programming language-agnostic and implementations are readily available for most programming languages.

Other backend services include a Postgres[22] relational database server, a memcache[23] server used for Django session data, and a postfix server for sending emails.

Each server of the PORTULAN CLARIN workbench distributed system, which includes the user-facing nginx server, the Django frontend server, and all the backend services, is deployed in a separate Docker[24] container.

Containers are groups of one or more[25] processes running under a certain level of isolation from other processes on the same host. This isolation is managed by the operating system kernel and extends only as far as controlling access to resources such as files, memory, devices, and CPU time. Thus, all containerized and regular processes are served by the same kernel and can potentially share any resource available on the host.

By contrast, in a virtual machine, a whole new guest kernel is executed within a process running on the host kernel, and then new processes are run and managed by the guest kernel, which incurs a considerable memory and CPU overhead. Processes running within a virtual machine do not have direct access to resources available on the host (such as files, memory, devices, etc.), and vice versa. In order to share resources between the host and guest kernels there are several possible workarounds, but they always incur in yet another memory and CPU overhead.

22 https://www.postgresql.org/
23 https://memcached.org/
24 https://www.docker.com/
25 Docker containers usually run a single process.

Containers are the best fit for our needs because they are extremely lightweight, and allow us to run each server in its own tailored environment while sharing files across containers.

As mentioned above, one of the conditions that compels us to segregate a tool into its own backend server is if the tool is no longer being actively developed or maintained. The fundamental reason is because, at some point in the future, the specific versions of libraries and other dependencies of an unmaintained tool will no longer be available for installation in an up-to-date operating system, or even if they are, they may clash with more recent versions required by other tools.

Docker images are standalone executable packages that include everything needed to run a container: code, system tools, system libraries, and settings. Thus, by including all the dependencies of a tool within a dedicated docker image, we create a perfect environment for each tool.

With Docker Swarm,[26] groups of containers are configured and managed as *services*, which communicate with each other through Docker-managed private networks. Service containers can be spread across any number of available swarm *nodes*, that is networked machines that have Docker installed and have been added to the swarm. The swarm also provides some mechanisms for maintaining availability of services: should a container crash, the swarm will restart it; or if one host becomes unavailable, the swarm will relocate containers that were running on it to other available hosts.

4 Current status of the PORTULAN CLARIN workbench

At the time of writing this chapter, dozens of tools have been integrated into the workbench, with more to come.[27]

Tools are spread across the categories listed in Table 1, and new categories will be added as needed to accommodate new tools. As described in Section 3,

26 https://docs.docker.com/engine/swarm/

27 The PORTULAN CLARIN workbench comprises a number of tools that are based on a large body of research work contributed by different authors and teams, which continues to grow and is acknowledged here: Barreto et al. (2006); Branco et al. (2010); Cruz, Rocha, and Cardoso (2018); Veiga, Candeias, and Perdigão (2011); Branco and Henriques (2003); Branco et al. (2011); Branco and Nunes (2012); Silva et al. (2009); Branco et al. (2014); Rodrigues et al. (2016); Branco and Silva (2006); Rodrigues et al. (2020); Costa and Branco (2012); Santos et al. (2019); Miranda et al. (2011).

the workbench provides an automatically generated index with links to individual tools grouped by their category. In its current form, this index is a simple list of categories, with brief descriptions and hyperlinks to the tools available under each category.

This simple design is reminiscent of the initial stages of development of the workbench, when only a handful of categories was involved. Despite its simplicity, this design continues to serve its purpose adequately, even though the number of categories has nearly doubled since that initial development stage. However, as the number of categories continues to grow, albeit at a slower pace, at some point in the future we may have to redesign this index, perhaps by introducing a combination of faceted filtering, free text searching, or another level of categorization.

We bring this up to exemplify how design decisions have been made throughout the development of the workbench: if in doubt, we first try to implement the simplest design that fulfills a given purpose. We defer adding complexity to the interface, until it becomes clear, through usage, that the simpler design is not as effective as it needs to be. And at that point, we will be in a better position to design a good interface, not only because we already have a lean working base design that we can use as starting point, but also because we know its shortcomings.

In order to gather feedback from potential users, the workbench was disseminated among the PORTULAN CLARIN implementation partners and at a number events where the infrastructure has been presented. Feedback was very positive regarding the interface and its usability, even though, during the dissemination events, engaging with the audiences in a productive way may turn out to be a challenge due to the different scientific and technical backgrounds of the participants.

Suggestions that have been submitted for new tools to be incorporated in the workbench have not tended towards novel or complex language technology applications, but towards what is comparatively simple in functionality, such as a concordancer capable of running over any user-submitted corpora.[28] We find such suggestions extremely valuable and will be working towards incorporating them into the workbench.

The workbench gets roughly one-third of the unique page views in PORTULAN CLARIN,[29] with the constituency and dependency parsers being the most

28 The concordancer that is currently available runs over a pre-indexed fixed corpus.

29 The PORTULAN CLARIN repository of language resources (data and software), in turn, is only slightly more popular, with 40% of the unique page views.

popular tools. Following the parsers is LX Semantic Similarity, a tool for measuring the semantic similarity of words.

5 Conclusion

In this chapter we have described the multi-interface approach implemented at PORTULAN, which we believe opens up language processing services to a wider array of users, coming from and carrying the most diverse backgrounds and motivations. We advise against making language processing services available through a single interface, designed with a specific user profile in mind, which would necessarily be too inflexible for some users or too complex for others. Instead, we propose four different interfaces, each one demanding an increased level of technical skill from the user, but empowering the user in return.

Table 1: Tool categories.

Concordancing . . .	Retrieval of contexts of occurrence of expressions in annotated texts.
Constituency parsing . . .	Analysis of syntactic constituents in sentences.
Dependency parsing . . .	Analysis of grammatical functions in sentences.
Grammatical quantitative analysis . . .	Occurrence counting of grammatical elements in texts.
Named entity recognition . . .	Detection and semantic classification of names in texts.
Nominal inflection . . .	Lemmatization and inflection of nominal expressions.
Orthographic normalization . . .	Conversion to orthographic standard.
POS tagging . . .	Tokenization and morphosyntactic tagging of expressions in texts.
Phonological transcription . . .	Conversion of graphemic into phonological representation.
Proficiency classification . . .	Quantitative analysis and proficiency level classification of texts.
Semantic role labelling . . .	Analysis of semantic roles of syntactic constituents in sentences.
Semantic similarity . . .	Semantic similarity between words.
Sentence splitting . . .	Segmentation of texts into sentences and paragraphs.
Sentiment analysis . . .	Analysis of emotional polarity in texts.
Sub-syntactic analysis . . .	Tokenization, lemmatization, inflection analysis, and morphosyntactic tagging of expressions in texts.
Syllabification . . .	Syllabification of expressions.

Table 1 (continued)

Temporal analysis . . .	Analysis of events and of temporal information in texts.
Tokenization . . .	Segmentation of texts into lexical tokens.
Transcription . . .	Written representation of speech.
Translation . . .	Translation of a sentence from a source language to a target language.
Treebank searching . . .	Retrieval of syntactic patterns and expressions in annotated sentences.
Verbal conjugation . . .	Conjugation of verbs.
Verbal lemmatization . . .	Lemmatization of verbal expressions.
Wordnet browsing . . .	Browsing of wordnet lexical semantic network.

The most basic type of interface, which we termed *online service*, is designed to be attractive and to invite users to self-guided exploration, for example by providing one-button-click examples. The second type of interface, termed *file processing*, is akin to the CLARIN Switchboard and allows the user to upload a large input file and have it processed with minimal effort. The third type of interface, *Jupyter notebooks*, gives users a starting point for designing and developing their own experiments. Notebooks may be edited and executed through a browser without requiring installation on users' computers. The fourth and most technically demanding, but also the most empowering interface, the *web service*, is a language-agnostic remote procedure call interface to be used from within a computer program written in any programming language.

After expanding on the design and rationale of these four types of interfaces, we shared key aspects of the implementation, which include far-reaching and long lasting decisions such as the choice of a programming language, overall architecture, frameworks, communication protocols, process containerization, code organization, and development and deployment practices.

Lastly, we reported on the current status of the workbench and feedback that we have had from users.

Bibliography

Barreto, Florbela, António Branco, Eduardo Ferreira, Amália Mendes, Maria Fernanda Nascimento, Filipe Nunes & João Silva. 2006. Open resources and tools for the shallow processing of Portuguese: The TagShare project. *Proceedings of the 5th international conference on language resources and evaluation (lrec)*, 1438–1443.

Branco, António, Sérgio Castro, João Silva & Francisco Costa. 2011. CINTIL DepBank handbook: Design options for the representation of grammatical dependencies. Technical Report DI-FCUL-TR-2011-03, University of Lisbon.

Branco, António, Francisco Costa, João Silva, Sara Silveira, Sérgio Castro, Mariana Avelãs, Clara Pinto & João Graça. 2010. Developing a deep linguistic databank supporting a collection of treebanks: the CINTIL DeepGramBank. *Proceedings of the 7th international conference on language resources and evaluation (lrec)*, 1810–1815.

Branco, António, Amália Mendes, Paulo Quaresma, Luís Gomes, João Silva & Andrea Teixeira. 2020. Infrastructure for the science and technology of language PORTULAN CLARIN. *Proceedings of the 1st international workshop on language technology platforms*, 1–7. Marseille, France: European Language Resources Association.

Branco, António & Filipe Nunes. 2012. Verb analysis in a highly inflective language with an MFF algorithm. *Proceedings of the 11th international conference on the computational processing of portuguese (propor)*, Lecture Notes in Artificial Intelligence no. 7243, 1–11. Springer.

Branco, António, João Rodrigues, João Silva, Francisco Costa & Rui Vaz. 2014. Assessing automatic text classification for interactive language learning. *Proceedings of the ieee international conference on information society (isociety)*, 72–80.

Branco, António & João Silva. 2006. A suite of shallow processing tools for Portuguese: LX-Suite. *Proceedings of the 11th conference of the european chapter of the association for computational linguistics (eacl)*, 179–182.

Branco, António & Tiago Henriques. 2003. Aspects of verbal inflection and lemmatization: Generalizations and algorithms. *Proceedings of xviii annual meeting of the portuguese association of linguistics (apl)*, 201–210.

Costa, Francisco & António Branco. 2012. Aspectual type and temporal relation classification. *Proceedings of the 13th conference of the european chapter of the association for computational linguistics*, 266–275.

Cruz, A. F., G. Rocha & H. L. Cardoso. 2018. Exploring spanish corpora for portuguese coreference resolution. *2018 fifth international conference on social networks analysis, management and security (snams)*, 290–295.

Google. Google Colab. https://research.google.com/colaboratory/faq.html (accessed 20 September 2021).

Hajič, Jan, Eva Hajičová, Barbora Hladká, Jozef Mišutka, Ondřej Košarko & Pavel Straňák. 2022. LINDAT/CLARIAH-CZ: Where we are and where we go. In Darja Fišer & Andreas Witt (eds.), *CLARIN. The infrastructure for language resources*. Berlin: De Gruyter.

Project Jupyter, Matthias Bussonnier, Jessica Forde, Jeremy Freeman, Brian Granger, Tim Head, Chris Holdgraf, Kyle Kelley, Gladys Nalvarte, Andrew Osheroff, M Pacer, Yuvi Panda, Fernando Perez, Benjamin Ragan Kelley & Carol Willing. 2018. Binder 2.0 – Reproducible, interactive, sharable environments for science at scale. In Fatih Akici, David Lippa, Dillon Niederhut & M Pacer (eds.), *Proceedings of the 17th Python in Science Conference*, 113–120.

Knuth, Donald Ervin. 1984. Literate programming. *The computer journal* 27 (2): 97–111.

Kupietz, Marc, Nils Diewald & Eliza Margaretha. 2022. Building paths to corpus data: A multi-level least effort and maximum return approach. In Darja Fišer & Andreas Witt (eds.), *CLARIN. The infrastructure for language resources*. Berlin: De Gruyter.

Miranda, Nuno, Ricardo Raminhos, Pedro Seabra, Joao Sequeira, Teresa Gonçalves & Paulo Quaresma. 2011. Named entity recognition using machine learning techniques. *Epia-11, 15th portuguese conference on artificial intelligence*, 818–831.

Rodrigues, João, Francisco Costa, João Silva & António Branco. 2020. Automatic syllabification of portuguese. *Revista da Associação Portuguesa de Linguística*, no. 1.

Rodrigues, João, António Branco, Steven Neale & João Silva. 2016. LX-DSemVectors: Distributional semantics models for the Portuguese language. *Proceedings of the 12th international conference on the computational processing of portuguese (propor'16)*, 259–270.

Santos, Rodrigo, João Silva, António Branco & Deyi Xiong. 2019. The direct path may not be the best: Portuguese-chinese neural machine translation. *Proceedings of the 19th epia conference on artificial intelligence*, 757–768.

Silva, João, António Branco, Sérgio Castro & Ruben Reis. 2009. Out-of-the-box robust parsing of Portuguese. *Proceedings of the 9th international conference on language resources and evaluation (lrec)*, 75–85.

Veiga, Arlindo, Sara Candeias & Fernando Perdigão. 2011. Generating a pronunciation dictionary for European Portuguese using a joint-sequence model with embedded stress assignment. *Proceedings of the 8th Brazilian symposium in information and human language technology*.

Zinn, Claus. 2018. The language resource switchboard. *Computational Linguistics* 44 (4): 631–639.

Zinn, Claus & Emanuel Dima. 2022. The CLARIN Language Resource Switchboard: Current state, impact, and future roadmap. In Darja Fišer & Andreas Witt (eds.), *CLARIN. The infrastructure for language resources.* Berlin: De Gruyter.