Proceedings of the 1st International Workshop on Language Technology Platforms (IWLTP 2020), pages 59–65
Language Resources and Evaluation Conference (LREC 2020), Marseille, 11–16 May 2020

Towards Standardization of Web Service Protocols for NLPaaS

Jin-Dong Kim[1], Nancy Ide[2], Keith Suderman[2]
[1]Database Center for Life Science (DBCLS), [2]Department of Computer Science, Vassar College
[1]Kashiwa, Chiba, Japan, [2]Poughkeepsie, New York, USA
jdkim@dbcls.rois.ac.jp, ide@cs.vassar.edu, suderman@cs.vassar.edu

Abstract

Several web services for various natural language processing (NLP) tasks ("NLP-as-a-service" or NLPaaS) have recently been made publicly available. However, despite their similar functionality these services often differ in the protocols they use, thus complicating the development of clients accessing them. A survey of currently available NLPaaS services suggests that it may be possible to identify a *minimal* application layer protocol that can be shared by NLPaaS services without sacrificing functionality or convenience, while at the same time simplifying the development of clients for these services. In this paper, we hope to raise awareness of the interoperability problems caused by the variety of existing web service protocols, and describe an effort to identify a set of best practices for NLPaaS protocol design. To that end, we survey and compare protocols used by NLPaaS services and suggest how these protocols may be further aligned to reduce variation.

Keywords: NLPaaS, web services, standards, synchronous protocols, asynchronous protocols

1. Introduction

There is considerable demand within both academia and industry for immediately available natural language processing (NLP) capabilities that can analyze and mine the vast amounts of textual data thar have become available in recent years. To answer this need, "NLP-as-a-service" (NLPaaS) web services are beginning to be developed, including Natural Language API of Google[1], Amazon Comprehend[2] and CLARIN-D NLP services[3], to name a few.

Every web service supports one or more protocols to remotely invoke its API (Application Programming Interface) in order to provide programmable access to its functionality. Among others, protocols which follow the REST (REpresentational State Transfer) architectural style (Fielding and Taylor, 2000) have become popular, due to its simplicity and flexibility. However, REST itself is a protocol design *style*, not a specific protocol, which leaves it to the implementer to decide how data objects are exchanged in client-server communication. This flexibility, while attractive to web service developers, has led to a lack of consistency in the protocols used by different NLPaaS services. As a result, those implementing clients for NLPaaS services that come from different developers often have to accommodate different communication protocols.

In this paper, we describe an effort to identify a minimal common protocol for NLPaaS based on best practices, with the aim of raising awareness of the interoperability problems caused by the variety of existing web service APIs and soliciting input for a standard set of NLPaaS service APIs. To that end, we survey and compare APIs used by NLPaaS services and provide a draft proposal intended to serve a basis for the eventual development of an NLPaaS API standard. We restrict the scope of NLP services to those that take texts as input and return the result of some NLP process as a result, as a starting point; however, we feel that an acceptable minimal common protocol for services ingesting textual data could be generally applicable across web services performing a wider variety of tasks.

[1]https://cloud.google.com/natural-language/
[2]https://aws.amazon.com/comprehend/
[3]https://weblicht.sfs.uni-tuebingen.de

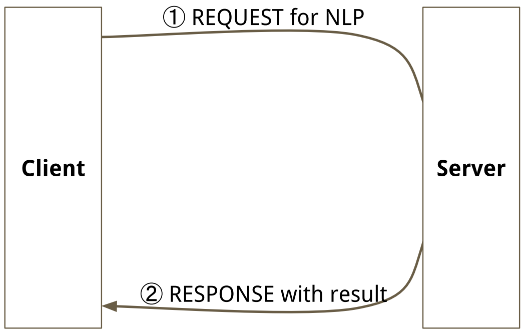

Figure 1: General synchronous protocol

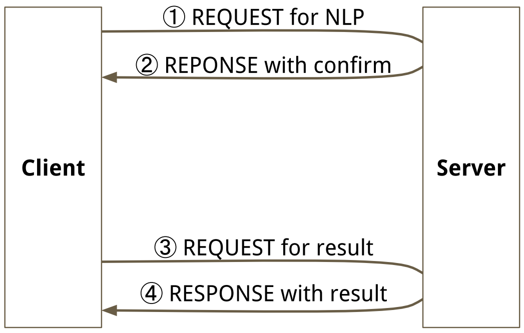

Figure 2: General asynchronous protocol

2. Synchronous and Asynchronous Protocols

There are two basic protocols for exchange of information among services and clients: *synchronous* protocols and *asynchronous* protocols,

Figure 1 illustrates a *synchronous protocol* exchange between a client and a server. The exchange is initiated by a request from the client to the server (typically, a GET or

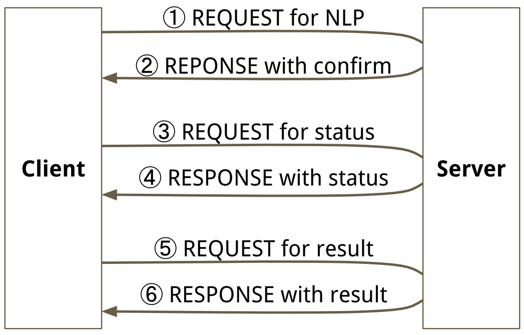

Figure 3: General asynchronous protocol with status checking

POST request) and completed by a response from the server to the client. Synchronous protocols block activity on the client as well as the server while the request is being processed; therefore, to avoid resource starvation in unexpected situations (network problems, errors, etc.), the request is typically subjected to a conservative timeout.[4] Therefore, when requests are expected to take an extended amount of time, e.g. in order to process a large amount of text or to execute a heavy task, the use of a synchronous protocol may be inappropriate.

Asynchronous protocols, illustrated in Figure 2, solves the timeout problem by separating the request from the delivery of the result, which is then handled by a separate connection. Services using asynchronous protocols are commonly coupled with an API that enables the client to check the status of the requested task, i.e., whether it is completed, still processing, or has encountered a problem. Figure 3 illustrates an asynchronous protocol with status checking.

3. Survey

This section presents our survey on the protocols used by existing NLPaaS services. For the purposes of comparison and to save space, only a selected subset of the services surveyed are included here. Note that due to the focus on NLP-related services, our survey is limited to services that take plain or annotated text as input and return a processing result as output. The result may be text or other forms of data (e.g., key-value pairs) resulting from the analysis. The subset of APIs we describe here is intended to include a variety of NLPaaS services available from different types of developers and serving a variety of audiences, including freely available services developed by academic and other non-commercial communities (CLARIN, CoreNLP, PubDictionaries), national services (PubTator, ETRI), and commercial services (Google).

Note that the focus of our survey is on the protocols used for sending and receiving data and does not consider the types of text analysis that the APIs provide (e.g., named entity recognition, sentiment analysis). For a comprehensive survey of the text analytic functions provided by different commercial services, see (Dale, 2020).

[4]Many HTTP servers, e.g., Apache, NGINX, and Tomcat have a default request timeout of 60 seconds.

3.1. Synchronous Protocols

As described above, synchronous protocols involve a simple client-server conversation consisting of a request followed by the corresponding response. Differences among servers using synchronous protocols appear primarily in their conventions for specifying input and output.

Table 1 gives an overview of the synchronous protocol APIs for several NLPaaS services, including the CLARIN-D (Hinrichs et al., 2010) and CLARIN-PL (Piasecki, 2014) services from the European CLARIN project; ETRI NLP API Korean NLP[5], developed and maintained by the Electronics and Telecommunications Research Institute (ETRI); Google Natural Language API, a commercial service provided by Google; and PubDictionaries (Kim et al., 2019), a service provided by the Database Center for Life Science (DBCLS). We also include Stanford CoreNLP (Manning et al., 2014), which is one of the most widely used NLP toolsets that is also implemented as a NLPaaS web service.

3.1.1. Methods and Content types

Most NLPaaS services receive requests using the *POST* HTTP verb (Fielding et al., 1999) in order to accommodate the need to send a (relatively) large body of text for processing. Certain services, such as PubDictionaries, support requests using the *GET* method, in this case because the service processes primarily short, natural language queries. With the *POST* method, some services require the content type to be explicitly specified, while others assume that the content type is always text (CoreNLP) or JSON (ETRI). Again, PubDictionaries is somewhat more flexible, accepting data in various formats: the content type of a POST request may be either *multipart/form-data* (for key-value pairs), *application/json* (for a hash or an array), or *text/plain* (for plain text).

3.1.2. Parameters

NLPaaS services take several parameters, including a block of *text*, the NLP *task(s)* to be run, and *user information*(e.g., for access control)

Text Services utilize two different methods to pass text to the server: through a parameter on the GET request and as the payload of a POST request. In a GET request, the (short) text to be processed is given as the value of a parameter, whose name may differ among servers; *text* is commonly used, but more abstract names such as *content* may be used for services that can process multiple content types (e.g., HTML, XML). When using a POST request, the payload is typically either key-value pairs (*multipart/form-data*), JSON object (*application/json*), or the text itself (*text/plain*). In either of the first two cases, the key name *text* is commonly used to send a block of text to a service.

Process The protocols used by some services include specification of the NLP process or processes to be invoked. This is accomplished in various ways: Google provides a different URL for each different NLP service, and ETRI receives the specification through a parameter. CLARIN-D, CLARIN-PL, and CoreNLP allow specification of a sequence of NLP processes through a parameter; however,

[5]http://aiopen.etri.re.kr/ (written in Korean)

Service	Method	Content type		Parameters		Identity
		Request	Response	Text	Process	
CLA-D	POST	multipart/form-data	n/s	*content*	*chain* (XML)	*apikey*
CLA-PL	POST	application/json	n/s	*text*	*lpmn*	*user* (email)
CoreNLP	POST	text (implicit)	multi	(payload)	*properties:annotators*	-
ETRI	POST	JSON (implicit)	application/json	*argument:text*	*argument:analysis_code*	*access_key*
Google	POST	application/json	application/json	*document:content*	(encoded in URL)	OAuth2
PubDict	GET|POST	multiple	application/json	*text*	(encoded in URL)	-

Table 1: APIs of synchronous protocols of several NLPaaS services. Note that "CLA" denotes CLARIN and "PubDict" denotes PubDictionaries. Items in italics are parameter names.

as indicated in Table 1, they use different parameter formats (XML for CLARIN-D, pipe ('|')-separated names of NLP processes for CLARIN-PL, and comma (',')-separated names of NLP processes for CoreNLP).

User Information Some services require information concerning the user who is calling the service, e.g., for access control or billing. Services may obtain this information via a parameter of the request (e.g., *apikey* for CLARIN-D, *user* for CLARIN-PL, and *access_key* for ETRI), while others use standard authentication schemes (e.g., *OAuth2* for Google).

3.2. Asynchronous Protocols

Asynchronous protocols are typically used when it is necessary to transmit large amounts of data–in the context of NLPaaS services, a large body of text–in order to avoid the timeout problem outlined in Section 2.. Therefore, asynchronous requests typically use the HTTP *POST* method, which allows for sending texts of unlimited size using the naming and content specification conventions outlined above. The relevant differences among asynchronous protocols concern the methods used to pass information about a request and requests for metadata, e.g., status of the job. To illustrate these differences, three services are surveyed: *CLARIN-PL* (Piasecki, 2014), *PubDictionaries* (Kim et al., 2019) and *PubTator Central* (Wei et al., 2019).

The asynchronous protocols of PubTator Central and PubDictionaries follow the overall request-response flow illustrated in Figure 2. However, they use different methods to pass necessary information in order for the client to follow the flow of execution. For example, when accepting a request such as

```
POST /annotate/submit/Gene ...(parameters)
```

PubTator Central responds with the status code 200 ("OK") together with a session number in the body of the response. The client is then supposed to compose the URL for retrieving the result using the session number and send a second request to the server, e.g.,

```
GET /annotate/retrieve/{SessionNumber}
```

In contrast, PubDictionaries returns the status code 303 ("See other") for a successfully received request, together with a *Location* HTTP header that specifies the URL for retrieving the result.

When a request for a result is submitted, PubTator responds with the status code 404 ("Not found") if the result is not ready, together with the warning message "[Warning : The Result is not ready" in the body of the response. PubAnnotation responds instead with status code 503 ("Service unavailable"), along with a *Retry-After* HTTP header to provide a hint to the client as to when to try to retrieve the result again. In the case where the result is ready when requested, both services respond with 200 ("OK") together with the result in the body.

CLARIN-PL uses an asynchronous protocol following the request-response flow illustrated in Figure 3. Like PubTator Central, CLARIN-PL uses the body of the response to inform the client of the task ID, with which the client can compose the URL for checking the status of the task. Below is the synopsis of the initial request:

```
POST /nlprest2/base/startTask ...(parameters)
```

The response is a task ID in the body of the response, from which a request to check the status of the task can be composed:

```
GET /nlprest2/base/getStatus/{taskID}
```

The response to this request is a JSON object:

```
{
  "status":"DONE"|"ERROR"|"QUEUING",
  "value":"..."
}
```

The client will keep checking the status until the value of the *status* key is *DONE*. When completed, the *value* key will be filled with the result ID, from which the client can compose the URL and make a request for the result, e.g.,

```
GET /nlprest2/base/download/{resultsID}
```

3.3. Summary

The differences outlined above for both synchronous and asynchronous protocols demonstrate the implementation options among services providing NLP processing. These differences complicate client development by requiring different means to handle sending requests and processing responses to different services. However, these variations are generally not due to systemic differences among services, but rather are in most cases simply a matter of arbitrary choice. It therefore seems possible to identify a set of conventions for client-server communication for NLPaaS, thereby simplifying client development for both synchronous and asynchronous processes.

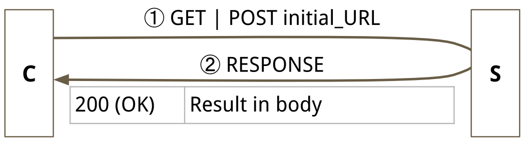

Figure 4: Synchronous protocol, proposal

4. A (Modest) Proposal

This section presents a preliminary proposal for protocols for NLPaaS services, based on the practices outlined in the previous sections. The aim is to provide a basis for continued discussion and development by members of the community at large.[6]

4.1. Criteria and Scope

The survey of differences among protocols used by NLPaaS services provides a basis for establishing the design criteria for protocol standardization.
The scope of this proposal includes:

- Request-response flow
- Request methods and headers
- The *text* parameter
- Response codes and headers

Note that the proposal does not cover input/output formats for the input text and the NLP processing results. There exist several standards for text and annotation formatting, and formatting can be dealt with in a separate layer from the protocols. Furthermore, input/output formats typically conform to the requirements of specific tools; a standard format would unnecessarily burden service developers with conversion to and from internal formats in order to be compliant. For the same reasons, we do not address user identification/authorization methods, nor do we consider parameters other than *text* since they are often tightly coupled with the functionality of a given service.
To illustrate how the proposed protocol might be used, we consider both a client-server communication environment and a server-server communication environment using *PUSH* notifications.

4.2. Synchronous protocol

Figure 4 illustrates the request and response flow of the proposed synchronous protocol. The initial request must be sent using the *POST* method. An NLPaaS service must receive a block of text through the request parameter *text*, which must be delivered either via the payload of *multipart/form-data* or as encoded in the URL. The following *cURL* command[8] illustrates this:

```
curl -F text="A␣sample␣text"
     URL_for_annotation
```

[6]To conform to the formal specifications in RFC 2119[7], in our discussion we use the verb *must* when a given practice is required and *may* when a given practice is recommended.

[8]In the example *cURL* commands, parameters other than *text* are omitted.

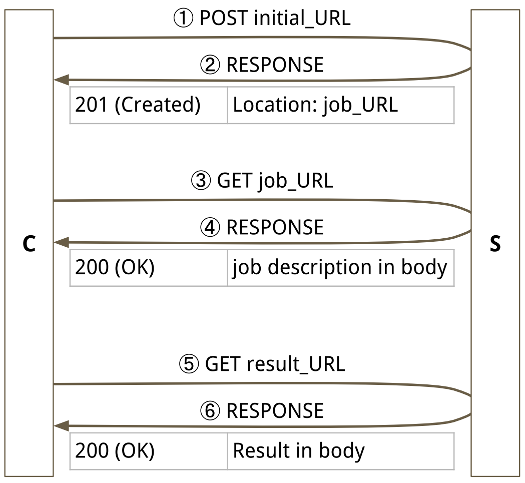

Figure 5: Asynchronous protocol

Note that specifying the request parameter *text* as a common channel for delivery of text does not prevent the service from receiving input through other channels, such as the *content* key[9]. When a request includes many parameters, and especially when it includes a structured parameter, it is common practice to include all the parameter settings in a single JSON object and send it through the payload of *application/json*; therefore, we recommend that services receive a payload of type *application/json*. Upon receiving a request, the service must execute its NLP process over the text, and, when successful, it must respond with status code 200 (OK) together with the result in the body. [10].

4.3. Asynchronous protocol with polling

Figure 5 illustrates our proposal for an asynchronous protocol for NLPaaS services, consisting of the following:

1. The initial request
   1-1. Must be sent using the *POST* method
   1-2. When successful, the response must include
      1-2-1. Status code *201 ("Created")*
      1-2-2. the *Location* header to specify the *job_URL*
      1-2-3. the description of the job, in the body
2. Second request
   2-1. Must be sent using the *GET* method
   2-2. The response must include
      2-2-1. Status code *200 ("OK")*
      2-2-2. the description of the job, in the body
3. Third request
   3-1. Must be sent using the *GET* method

[9]For example, Google uses the *content* key to receive documents as may plain-text or HTML. For Google to conform the standard, it may use *text* key to receive text, while retaining *content* to receive html.

[10]As discussed in Section 4.1., the format of the output is out of scope of this specification.

Attribute	Description	Format
submitted_at	Timestamp of submission	ISO 8601
started_at	Timestamp of execution	ISO 8601
finished_at	Timestamp of completion	ISO 8601
elapsed	Elapsed time	ISO 8601
ETR	Estimated time remaining	ISO 8601
result_location	Location of the result	URL
error_message	Error message	String
status	*IN_QUEUE* or *IN_PROGRESS* or *DONE* or *ERROR*	String

Table 2: Attributes for a job description.

3-2. The response must include

3-2-1. Status code *200 ("OK")*

3-2-2. the result, in the body

Initially, the client sends a request to a server to apply a certain NLP process to a block(s) of texts using the *POST* method (1-1). *POST* is used because the text may be very long, and, more importantly, POST is not a "safe" request[11] and therefore the response should not be cached. As for the synchronous protocol, the request parameter *text* must be used to send a block of text.

When the request is successfully accepted, the server must create a job to execute the desired NLP task and respond to the client with the status code *201* (1-2-1) together with a *Location* HTTP header (1-2-2), to indicate that the job is created and accessible via the URL specified by the header. The body of the response must contain the initial description of the job (1-2-3).

To describe a job, we propose the attributes listed in Table 2. At the time the NLP task terminates execution, the value of *finished_at* and either of *result_location* or *message* must be set. Among the attributes, *elapsed* and *status* are redundant, i.e., they can be calculated from other attributes as follows:

$$\text{elapsed} = \begin{cases} \text{current\_time} - \text{started\_at} & \text{if finished\_at} = \phi \\ \text{finished\_at} - \text{started\_at} & \text{otherwise} \end{cases}$$

$$\text{status} = \begin{cases} \text{IN\_QUEUE} & \text{if started\_at} = \phi \\ \text{DONE} & \text{result\_URL} \neq \phi \\ \text{ERROR} & \text{message} \neq \phi \\ \text{IN\_PROGRESS} & \text{otherwise} \end{cases}$$

However, because these attributes are frequently referenced they are included for convenience. The job description must be serialized into a response body of type *application/json*. This allows for structuring values, e.g., for status replies, it would be easier to define the ability to return multiple messages, possibly even with different "log levels" and with timestamps.

Once the job is created, it must be accessed using the *GET* method (2-1). Next, the service must respond with the status code 200 ("OK") and with the job description in the body (2-2). Note that responding with the status code 200 to a *GET* request may results in caching the request somewhere between the client and the server, and it is therefore recommend to include the *Cache-Control: no-store* header. The client is expected to repeatedly access the job until it finds that the *status* is either *DONE* or *ERROR* (polling). During the loop, the value of *ETR* (Estimated Time Remaining) must provide the client with enough information to enable efficient scheduling of future requests. When the status is *DONE*, the job description includes the URL for result as the value of the *result_location* attribute. The client then accesses the result using the specified URL (3-1), after which the service must respond with status code 200 and include the result in the body (3-2).

After the result is retrieved the server may want to delete the job and the result, either immediately or after a specified period of time (e.g., 24 hours). While not required, it is generally recommended that the service explicitly state in the protocol and API documentation exactly when the job and the result will be deleted.

[11] An HTTP method is "safe" if it does not alter the state of the server.

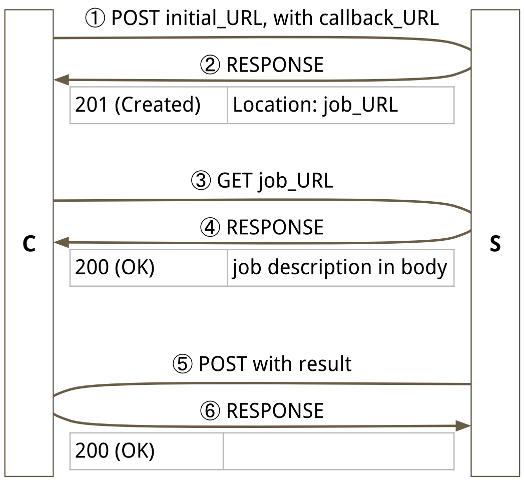

Figure 6: Asynchronous protocol with push notifications

4.4. Asynchronous protocol with callback

The protocol with *polling* proposed in Section 4.3. is necessary when a service has no way to talk to a client except by responding to the client's requests. However, if the server can talk to the client at any time, the server can instead *push* messages to the client to report when new information becomes available rather than responding to periodic client requests, thus avoiding the crush of a potentially large number of clients polling continuously. To enable this scenario, the client registers a *callback* URL as a part of the job submission. When the server has new information available, it sends this information in the same format the client would use when issuing a polling GET request (with the obvious difference that the server is issuing a POST to the client). Figure 6 illustrates our proposal for an asynchronous protocol with push notifications.

The differences from the polling model are:

- The initial request includes the callback URL, for which we propose the parameter name *callback_location*;

- When the task is completed, the server immediately sends the result to the callback URL, using the POST method;
- When the client has successfully received the result, it responds with status code 200.

Because the server will send a notification when the task is completed (successful or not), the client does not need to repeatedly check the status of the job in order to know the timing required to retrieve the result. However, the API of the service from which a client may request the job description is still useful when it is necessary to estimate when the result will be received, and, even after the client receives the result, to see the metadata associated with the job, e.g., length of execution time.

5. Discussion

As stated in Section 1., the proposal presented in Section 4. is a first draft intended to serve a basis for further development. Here we explain the rationale for various design choices over possible alternatives.

5.1. Response code for the initial request

HTTP is not designed with explicit consideration of asynchronous protocols, and therefore no existing response status code exactly fits the asynchronous scenario. The draft proposal specifies that the server must issue status code 201 ("Created") in response to an initial request for asynchronous communication. However, among existing systems and in relevant articles, some advocate for using 202 ("Accepted") or 303 ("See other"). The rationale behind our choice of 201 is that the initial request can be defined as a request for the creation of an "NLP job", which can be immediately created upon submission of the request. A drawback of this choice is that it is not user friendly, i.e., it reflects an engineering perspective rather than the perspective of end users, who simply want the result of the job. If we view the initial request from the user's perspective, it may be more reasonable for the server to respond with 202 or 303. In the case of 202, the value of the accompanying *Location* header would be interpreted as the location of the result. In the case of 303, the value of the *Location* header would be interpreted as a location for a relevant resource (e.g., the job), not the requested resource itself (e.g., the result). Although we have suggested one code over other possibilities, this topic remains open for further discussion.

5.2. Response Code for Polling

For polling, the server needs to continuously inform the client of the status and the estimated time remaining (ETR) to complete the job. Some services follow the overall request-response flow illustrated in Figure 2 and use the the status code 404 ("Not found") or 503 ("Service unavailable"). Code 503, which is an indication of a transient problem, is typically accompanied with the *Retry-After* header, an HTTP-native way to tell the client to try again within an estimated wait time. We have avoided these two codes because they are broadly understood as error codes indicating a problem with the request and/or the server. Ideally, there would exist a status code such as 309, standing for "Redirect to itself", that could be used together with *Retry-After*, but not with *Location*. With such a code the server could tell the client to make another request after a specified length of time because the request cannot be currently fulfilled.

5.3. Delivery of the result location

When the NLP task is complete and the result is ready to be served, the server responds to the request for polling with the status code 200 and the URL for the result in the *result_location* field of the response body. Some services use 303 with a *Location*, which is an HTTP-native means to inform the client of the location for the request; however, 303 was not chosen because it prevents the metadata of the job from being accessed after the job is completed.

5.4. Parameter passing

When a block of text is the single parameter of a POST request, a straightforward means to pass the parameter is to send it as payload of type "text/plain", possibly coupled with a specification of the character encoding (e.g., "text/-plain; encoding=UTF-8"). However, NLPaaS services often require additional parameters, such as the specification of the NLP process to be applied. When the payload is used to pass a block of text, the only means to pass additional parameters is to encode them in the URL, which is often unwieldy. In this case, the standard practice is to send all the parameters as key-value pairs with the content-type header "multipart/form-data".
When a value of a key is a structured value (e.g., an array of NLP processes to make up a pipeline), it may be difficult or impossible to send them as key-value pairs. For this reason, we recommend sending all parameters as a JSON object, which is a common practice.

6. Conclusion

In this paper we survey a number of NLPaaS services in order to identify current common practice and, in so doing, establish a basis for development of a standard for NLPaaS protocols. We outline a draft proposal for such a standard drawing on our observations, and offer it to the community for future consideration.
We recognize that standardization is a major endeavor that necessarily involves gathering input from the community of users in order to reach a broad consensus. We have therefore set up a GitHub repository[12] containing the draft specification so that the community can be actively involved in furthering this effort.

7. Acknowledgements

Thanks to Richard Eckart de Castilho and Jan-Christoph Klie for their insightful comments. This work was supported in part by the Database Integration Coordination Program funded by National Bioscience Database Center (NBDC) of Japan Science and Technology Agency (JST) and U.S. National Science Foundation (NSF) grant 1811123.

[12]https://github.com/jdkim/NLPaaS-Protocol

8. Bibliographical References

Dale, R. (2020). *Text Analytics APIs: A Consumer Guide*. Language Technology Group, 3 edition.

Fielding, R. T. and Taylor, R. N. (2000). *Architectural Styles and the Design of Network-Based Software Architectures*. Ph.D. thesis, University of California, Irvine. AAI9980887.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext Transfer Protocol – HTTP/1.1.

Hinrichs, E., Hinrichs, M., and Zastrow, T. (2010). WebLicht: Web-based LRT services for German. In *Proceedings of the ACL 2010 System Demonstrations*, pages 25–29, Uppsala, Sweden, July. Association for Computational Linguistics.

Kim, J.-D., Wang, Y., Fujiwara, T., Okuda, S., Callahan, T. J., and Cohen, K. B. (2019). Open Agile text mining for bioinformatics: the PubAnnotation ecosystem. *Bioinformatics*, 35(21):4372–4380, 04.

Manning, C., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S., and McClosky, D. (2014). The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, Baltimore, Maryland, June. Association for Computational Linguistics.

Piasecki, M. (2014). User-driven Language Technology Infrastructure -the Case of CLARIN-. In Jerneja Žganec Gros Tomaž Erjavec, editor, *Proceedings of the 17th International Multiconference Information Society - IS 2014*, volume G of *Language technologies*, pages 7–13. Institut Jožef Stefan.

Wei, C.-H., Allot, A., Leaman, R., and Lu, Z. (2019). PubTator central: automated concept annotation for biomedical full text articles. *Nucleic Acids Research*, 47(W1):W587–W593, 05.