Chapter 4
Contributing to the European Language Grid as a Provider

Dimitris Galanis, Penny Labropoulou, Ian Roberts, Miltos Deligiannis, Leon Voukoutis, Katerina Gkirtzou, Rémi Calizzano, Athanasia Kolovou, Dimitris Gkoumas, and Stelios Piperidis

Abstract The ELG platform enables producers of language resources and language technology tools and services to upload, describe, share, and distribute their services and products as well as to describe their companies, academic organisations and projects. This chapter presents the functionalities offered through web-based user interfaces for describing LT resources or related entities with metadata and for managing their publication. It gives a detailed description of the options that providers of LT tools can exploit to integrate them into ELG as ready-to-deploy services and the tools that ELG offers in their support during the preparation, upload and integration phases. The tools and packaging recommendations for resources to be uploaded in ELG are also presented. The chapter concludes with a discussion of functionalities offered to providers by ELG and other related platforms.

1 Introduction

The European Language Grid platform (Rehm et al. 2021) offers various functionalities for providers of Language Resources and Technologies (LRTs) through which they can share their assets with the Language Technology (LT) community and interested clients, customers or users of these technologies. The minimum requirement is that they make them accessible (by uploading them to ELG or through another website) and describe them with a metadata record that complies with the ELG specifications (see Chapter 2), where they specify the access location and licensing con-

Dimitris Galanis · Penny Labropoulou · Miltos Deligiannis · Leon Voukoutis · Katerina Gkirtzou · Athanasia Kolovou · Dimitris Gkoumas · Stelios Piperidis
Institute for Language and Speech Processing, R. C. "Athena", Greece,
galanisd@athenarc.gr, penny@athenarc.gr, mdel@athenarc.gr, leon.voukoutis@athenarc.gr, katerina.gkirtzou@athenarc.gr, akolovou@athenarc.gr, dgkoumas@athenarc.gr, spip@athenarc.gr

Ian Roberts
University of Sheffield, UK, i.roberts@sheffield.ac.uk

Rémi Calizzano
Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Germany,
remi.calizzano@dfki.de

G. Rehm (ed.), *European Language Grid*, Cognitive Technologies,
https://doi.org/10.1007/978-3-031-17258-8_4

ditions under which they can be used. To take advantage of the advanced features of ELG, providers can also integrate LT tools as ready-to-deploy services, following the ELG specifications, or upload the resource itself, in which case it will be stored and preserved according to the Data Management Plan (see Chapter 8) and made readily available to LRT consumers. Furthermore, descriptions of organisations that are active in the LT area can be added in order to promote their activities and products. Descriptions of projects that have been funded in the broader LT area can also be included in the ELG catalogue. LRTs, organisations that have provided or created them and projects that have contributed to their funding are linked together.

Detailed documentation is provided and a suite of helper tools have been developed aiming to make the contribution and integration of all entities briefly sketched above as simple as possible, taking into account the technical expertise and preferences of users. In ELG, the provision and management of catalogue entries is supported through web user interfaces (UIs) and REST application programming interfaces (APIs). Section 2 describes the steps a provider must take to contribute entries to the catalogue, and the tools provided by ELG to support this process. The ELG catalogue intends to be a reliable source for resources that can be accessed and (re-)used by commercial and non-commercial, research and public organisations as well as individuals. For this purpose, management and curation policies and processes for the metadata, data and services included in ELG have been set up, albeit with variations depending on the source and type of contribution. Only authorised and authenticated individuals can add LRTs in ELG; the registration and assignment of the "provider" user role is a simple process for all interested users (see Chapter 3). In addition, all entries go through a formal publication life cycle (see Chapter 2). Before being published in the catalogue, added metadata records are validated by the ELG core team (Section 3). Section 4 looks into the requirements for the different types of resources and entities in ELG, either integrated in ELG or available remotely and added to ELG as metadata records only. Further technical specifications are set for LT services that are intended to be deployed through the ELG cloud infrastructure, and for data resources hosted in ELG. Before being published in ELG, these resources go through a process that aims to ensure their technical validity and, for services, to set up the required environment for their deployment. Section 5 presents similar platforms and infrastructures and discusses the approach and tools they offer for providers of LRTs, in analogy to the comparison made for the platform functionalities from the point of view of consumers in Chapter 3.

2 Adding Resources to the ELG Platform

LRT providers come from a variety of backgrounds, some within Language Technology fields such as NLP or Computational Linguistics, and others from neighbouring fields such as Digital Humanities. Different providers have different levels of technical knowledge and familiarity with formal metadata descriptions, so ELG attempts to offer an integrated environment suitable for both expert and non-expert users. The

functions exposed for registering and managing catalogue entries and their accompanying data files are designed to be user-friendly while still offering advanced features to users with the relevant skills.

All metadata records must comply with the ELG metadata schema (Labropoulou et al. 2020). The schema offers a rich set of metadata elements for each type of LRT or entity (organisation, project) to be added. Individual elements are either *mandatory*, *recommended* or *optional*, depending on the record type. Providers can add entries with only the mandatory elements, although they are also encouraged to add the recommended ones. See Chapter 2 for more details.

2.1 Creating Metadata Records

Providers can add records in one of two ways: either by creating and uploading XML files compliant to the ELG schema (Section 2.1.1), or by using the interactive editor offered by ELG (Section 2.1.2). In practice many users will adopt a combination of the two approaches, for example, a provider who wishes to submit many similar records (such as MT services based on the same underlying engine but with models for different language pairs) may create their first record using the editor, export it as XML, and use this file as a template to generate the remaining records.

2.1.1 Creation and Upload of Metadata Files

This first option is probably more appealing to expert and technical users, especially those that wish to register multiple related records or produce frequently updated versions of LRTs registered in ELG. To facilitate the process of adding records, prefilled metadata templates and examples (with the mandatory and recommended elements) are available in the ELG GitLab repository[1]. As mentioned above, any existing metadata record can be exported from ELG as XML to be used as a template.

A REST endpoint for metadata validation of single files or zipped archives of XML files is publicly available and offered for providers that want to validate their metadata files and ensure they comply with the ELG schema before uploading them to the platform.[2] The XSD validator checks that all mandatory elements are filled in and that filled-in values are consistent with the data type declared for the elements – for example, if elements take values from controlled vocabularies or should follow a specific pattern – and returns the results in JSON form.

Users can upload their metadata records through the provider's grid (see Section 2.3) as single files or in batch mode. The import step includes additional validation rules, which check the syntactic and, to a certain extent, semantic integrity of the record. For example, checks are performed for metadata elements that depend

[1] https://gitlab.com/european-language-grid/platform/ELG-SHARE-schema

[2] https://live.european-language-grid.eu/catalogue/#/validate-xml

on the presence or value of other elements (e. g., the element "multilinguality type" which is mandatory for bilingual and multilingual resources), or for duplicate values (e. g., the same "language" value used twice). Validation errors are reported to the user for correction. If the file is valid, it is imported to the platform and the provider can perform further edits with the editor or submit it for publication in accordance with the publication life cycle (see Chapter 2).

2.1.2 Metadata Editor

The editor can be accessed through the provider's grid (see Section 2.3). It supports users in creating new metadata records, as well as editing and updating existing ones. The editor includes the mandatory and recommended fields of the ELG schema. Chapter 2 provides a summary of all mandatory metadata elements.

The editor has been designed with non-expert users in mind, and intends to hide the richness of the ELG schema. For this reason, we offer a full-fledged UI with metadata elements grouped into semantically coherent sets and layered along horizontal and vertical tabs, following the ELG conceptual structure. Different editor forms with the same look and feel have been implemented for each resource or entity type. Figure 1 shows the editor for tools/services; the horizontal tabs correspond to the main classes of the schema – in this case, LRT, tool/service and distribution – and the vertical tabs to categories of elements within that main section. The figure shows the LRT horizontal tab, whose options include "identity" (identification metadata such as the resource name, long description, and name of the creator responsible for the record), "categories" (classification elements such as keywords and subject domain), and "documentation" (links to publications, user manuals, or other documents describing the resource).

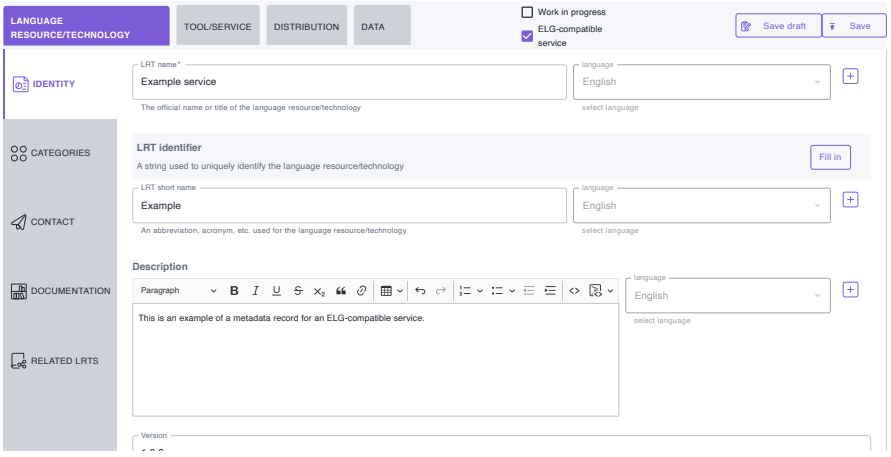

Fig. 1 ELG metadata editor

The editor guides the user to fill in at least all of the mandatory elements with appropriate values. Help tips and examples are available for metadata elements, and different editing controls are used for elements depending on their data type. For instance, the elements of controlled vocabularies are shown using dropdown lists. For vocabularies with many values (e. g., languages, service functions, etc.), we use a combination of dropdown lists with suggested values as the user types in the text.

The combination of dropdown lists and dynamically suggesting values is also applied to improve normalisation. For example, some elements such as keywords allow free text entry, however as the user types, a popup suggests matching values that have previously been used for the same element in other records, "nudging" the user to choose identical values instead of slight variations. The same lookup mechanism, of suggesting values from those already imported in the catalogue, is used for reducing the chance for duplicates of related entities such as agents, projects, documents, licences, and other resources.[3] For such entities, the ELG schema requires a set of minimal information, a name/title, and, optionally, an identifier and metadata elements that could uniquely distinguish it from similar entities (e. g., email for persons, website for organisations, a URL with the text for licences, etc.). Thus, when adding related entities through the editor, users type in a name/title, and are shown matching entries (if any) to select from; if not, they are prompted to fill in the required elements mentioned above. The same set of metadata elements is also used at the import of metadata records to uniquely identify the related entities.

Through the editor, providers have the option of saving incomplete metadata records ("draft"), for which only the data type of the metadata elements is validated (e. g., that they have entered a valid URL). When they decide to properly save the metadata record, we validate the entry using the yup library[4], implementing at least the same rules used at the import of metadata files. In case of errors, messages describe the error and location where it occurred (see Figure 2); clicking on the error, users are forwarded to its location.

2.2 Uploading and Managing Data Files

Data files, i. e., the physical files that contain the contents of a resource, must be uploaded as a ZIP file. Section 4.2.2 presents recommendations for the packaging of data resources, especially for those that can be split into subsets.

Providers can upload data files as a first step when they upload an XML file[5], or during the editing process with the editor. The editor includes a tab entitled "Data" (Figure 3) through which users can manage the files (upload, replace and delete).

---

[3] This is a well-known issue across catalogues; the adoption of unique persistent identifiers is recommended to resolve it, but not all entities are assigned such a unique identifier or it may not be known to the provider that submits the metadata record.

[4] https://github.com/jquense/yup

[5] At the time of writing, the upload of data files during the batch import of XML metadata records is not supported.

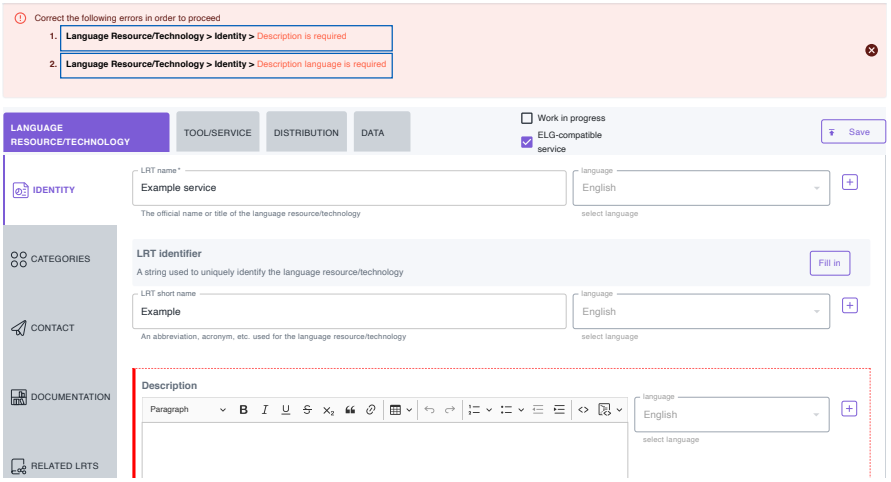

Fig. 2 ELG metadata editor with error messages

A resource may be available in a range of distributable forms ("distributions"), for example, in different file formats (e. g., as PDF, XML or TXT files). ELG supports the upload of multiple data files for the same resource. For this reason, when users upload more than one package of data files, they are prompted to associate each package with the respective distribution (i. e., the one that includes the metadata that describe the size and format of the particular set of files). This action is performed by selecting the specific package on the "distribution" tab.

2.3 Managing Catalogue Entries

The ELG platform presents users that have the "provider" role set with a "grid" (dashboard), through which they can access and manage the catalogue items they have created, as well as create new items (Figure 4). Since every provider is by definition

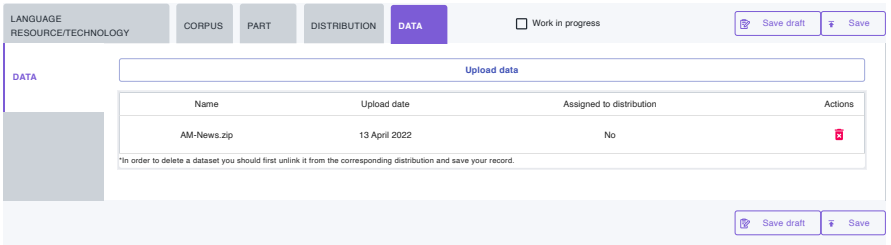

Fig. 3 ELG metadata editor – "data" tab for uploading data files

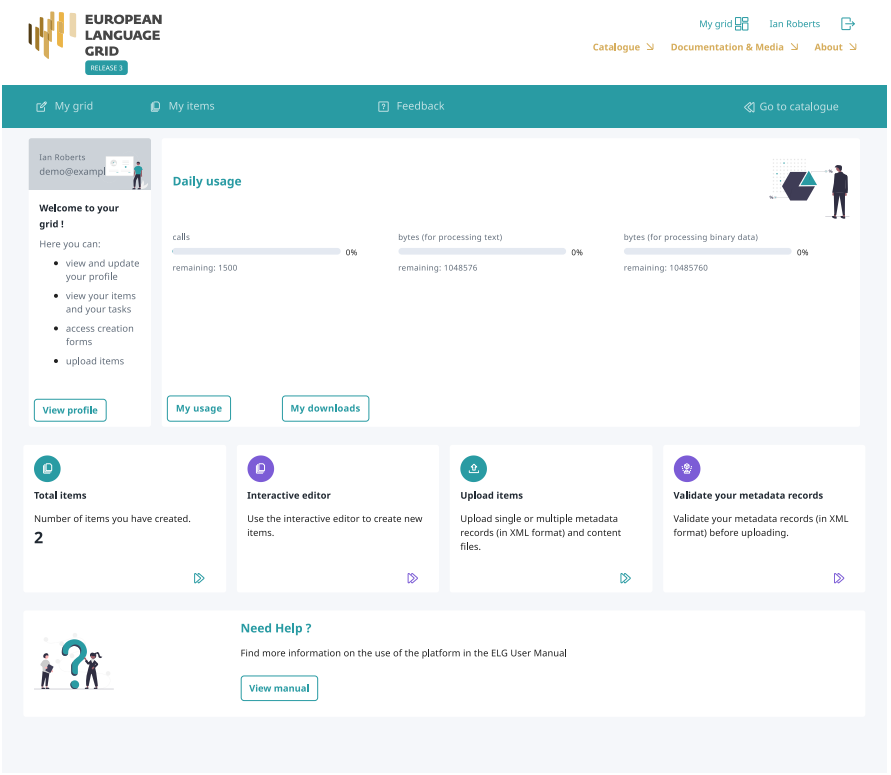

Fig. 4 Provider's grid (see Figure 6 in Chapter 3, p. 48, for the Consumer's grid)

also a consumer, the provider's dashboard is an extension of the consumer's dashboard shown in Chapter 3, adding a counter of the number of records this user has created and links to the editor, XML upload, and XML validator tools.

Users can manage the metadata records they have created through a dedicated page ("My items", Figure 5), and, in accordance with their user rights and the publication status of the record, perform the following actions: edit a metadata record, submit it for publication, create a new version of a published record, copy a metadata record (in order to use it as a model and create a similar record), delete a metadata record that has not yet been published, and request the unpublication of one of their records.[6] The "My items" page is a focused version of the catalogue, this time filtering records according to each user's role. This page also implements browse and search functionalities like the main catalogue page.

[6] Records cannot be completely deleted after publication except in exceptional circumstances, and then only by request to the ELG administrators.

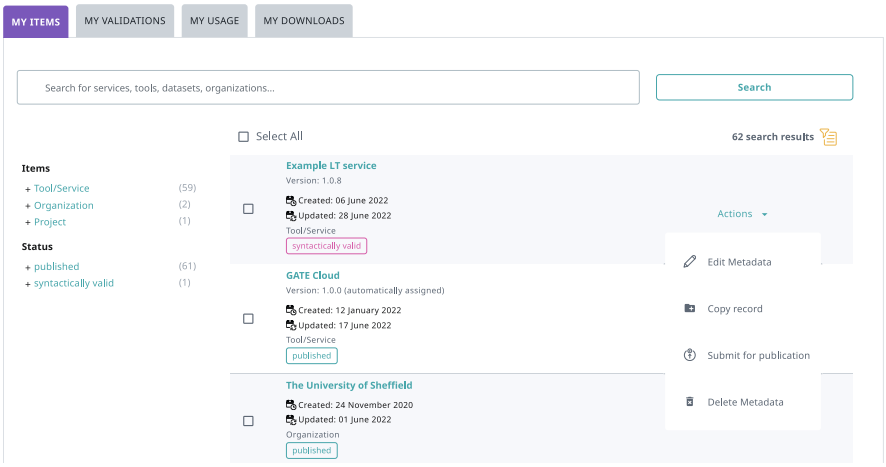

Fig. 5 "My items" page

3 Validating and Publishing Metadata Records

Metadata records added by individuals[7] enter a validation process, as specified in the ELG publication life cycle (see Chapter 2), before they are published in the catalogue: we perform technical/metadata and legal validation for ELG-compatible services and resources with uploaded data files, and validation at the metadata level only for all other metadata records. ELG-compatible services also go through a set of actions required for the registration of the service in the ELG platform (see Section 4.1.8).

Validators have access to the metadata records that have been assigned to them through the "validator's grid", and more specifically the "My validations" page (Figure 6). The validation form includes fields in which the validator can add internal comments (visible only to the other validators), and in the case of rejected records, a field for noting the reasons and suggested changes that are communicated to the provider for corrections. Providers can go through the changes and resubmit the record, which initiates a new round of validation, until final approval. When the metadata record has been approved by the responsible validator or validators, it is automatically made visible in the public catalogue.

4 Entity-Type Specific Requirements

There are several technical requirements that need to be met for LT services (Section 4.1) or resources (Section 4.2) to be deployed through or hosted in ELG successfully. We also present the requirements for metadata-only resources (Section 4.3).

[7] For harvesting and batch import functionalities from other catalogues, see Chapter 6.

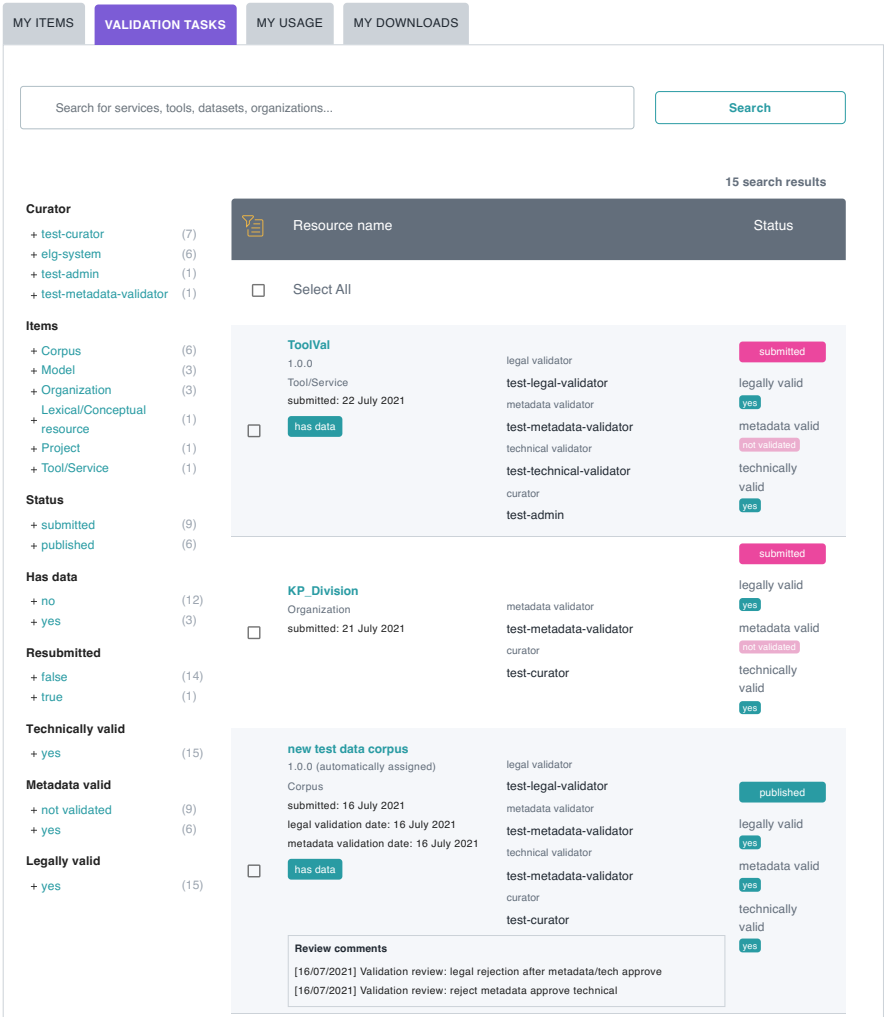

Fig. 6 "My validations" page

4.1 ELG-compatible Services

A service is ELG-compatible if it is packaged in a Docker image and follows the ELG LT internal API, i. e., the service consumes and produces messages in the ELG-specified format, as defined in Section 4.1.1 below. When a provider adds a tool or service to ELG either using XML metadata upload or through the metadata editor, they are asked if the service will actually be integrated in ELG, so that conformance to our specifications can be monitored.

4.1.1 Internal LT API Specification

The ELG internal LT API is closely related to the public API described in Chapter 3. The public API is a simplified derivative of the internal API. While both the internal and public APIs make use of the same JSON messages for input and output, the internal API is designed strictly around a single HTTP request-response transaction for each processing task, rather than the multi-step asynchronous mode supported by the public API.

For the internal API, services that accept text receive their requests as JSON, while services that process binary audio or image data receive a MIME "multipart/-form-data" request with the metadata in JSON and the binary data as the relevant audio or image MIME type. The endpoint must return the appropriate JSON response message depending on its function (standoff "annotations", classifications, audio, or new "texts" – which could be a single text, a series of sentences, a list of alternative translations, etc.). Examples include:

- *Information extraction (IE) services for text* accept a "text" request and return an "annotations" response; i. e., annotations whose position is described in terms of zero-based character offsets. Such services include tokenisers, sentence splitters, sentiment analysers, named entity recognisers, dependency parsers, etc.
- *Text classification services* accept a "text" request and return a "classification" response with the classes that have been assigned to the whole input text by the service. Examples are language identifiers, text-level sentiment classifiers etc.
- *Machine translation services* receive a "text" request and generate a new text or list of alternatives returned in a "texts" message. Services such as summarisation would use a similar format.
- *Information extraction services from speech* take "audio" requests and return the same standoff annotations as IE-from-text, but in this case the annotations are time segments in the audio stream, e. g., keyword spotting for audio files.
- *Speech recognition services* take "audio" requests and return a text transcription or a choice of n-best transcriptions, encoded as a "texts" message.
- *Text-to-speech services* take "text" messages and return "audio" messages, which can either include the returned audio inline as base64-encoded data, or as a URL reference to audio which has been uploaded to the temporary storage helper service (see Section 4.1.2).
- *Optical character recognition services* take "image" requests and return the extracted text as a "texts" response.
- *Image classification services* take "image" requests and return "classification" responses.

The formats of the input and output messages are generic and can be easily reused for integrating new types or classes of services. For example, Speech-to-Text services, such as a speech summariser that would consume an "audio" request and return a "texts" response in the same way as a pure speech recogniser, can easily be added. Other examples can be found in Chapter 7.

Detailed, up-to-date guidance on the process of integrating an LT service and selecting the most appropriate integration option can be found in the ELG documentation[8]; more information is provided in Section 4.1.3.

As described in Chapter 3, error, warning and progress report messages are represented as structured objects with a message *code*, representing a message that can be localised into many languages. The ELG team provides a set of standard message codes for common messages, and maintains their translations, but service providers who use their own custom messages are welcome to contribute their own localisations for integration into the public message resolver by contacting the ELG team.

Services that take a long time to process data have the option of returning a series of "progress" messages prior to generating the final response using the standard HTTP "server-sent events" format.[9]

4.1.2 Helper Services

ELG provides certain helper services that can be called at fixed URLs by LT service containers if they run within the platform. Notably, ELG provides a temporary storage helper which LT services can use in order to return data that does not naturally map on to the standard JSON-based response formats. This helper allows an LT service to store arbitrary blobs of binary data on a short-term basis (for any time from ten seconds up to 24 hours), and receive a randomly generated URL that can be included in the response JSON, and which the caller can retrieve up until its expiry time. Typical uses for this service include text-to-speech services that need to return larger chunks of audio data, or services that visualise structures such as parse trees in a binary image format. This is discussed further in the context of the Text2TCS service in Chapter 7, Section 5.1, p. 144 ff.

4.1.3 Integration Requirements and Options

The requirements for integrating an LT tool or service into ELG are as follows.

Expose an ELG-compatible endpoint: The provider needs to make sure that the LT tool or service to be integrated into ELG exposes an HTTP endpoint, i. e., either such an endpoint already exists or it needs to be implemented. The corresponding endpoint application must consume HTTP requests that follow the ELG JSON format, call the included or underlying LT tool and produce responses again in the ELG JSON format as specified in the the ELG LT internal API (Section 4.1.1). Developers working in Python or Java, Groovy, Kotlin, or other JVM-based languages, can make use of helper libraries provided by the ELG team to handle much of the boilerplate code for creating the HTTP listener, parsing and

[8] https://european-language-grid.readthedocs.io/en/stable/all/3_Contributing/Service.html

[9] https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-events

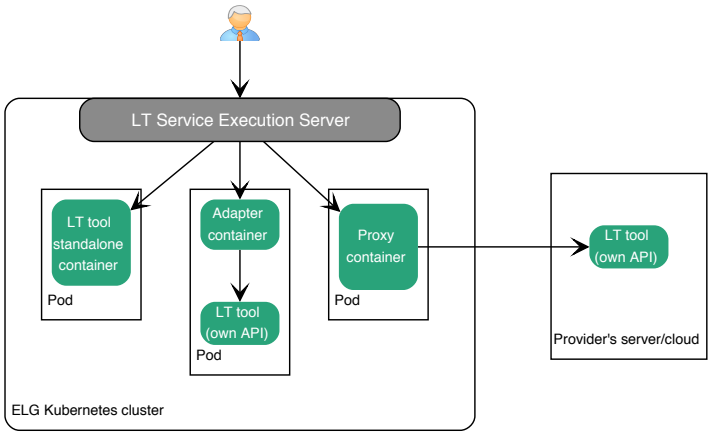

Fig. 7 Integration options

producing the JSON messages, etc., so that the provider can concentrate on their own business logic (see Sections 4.1.6 and 4.1.5 for more details).

Provide the application in the form of a Docker image: The whole application must be packaged as a container image using Docker or similar tools, and uploaded to a Docker registry, such as GitLab[10], DockerHub[11] or Azure Container Registry[12]. More than one image might be needed for one service, depending on how the service is made available. From the three options described in Fig. 7, providers can pick the one that best fits their needs.

- *LT tool packaged in one standalone image:* One image is created that contains the application that exposes the ELG-compatible endpoint and the actual LT tool. This is the most common approach when wrapping tools that are callable as libraries from custom code, such as Python machine learning models.
- *LT tool running remotely outside the ELG infrastructure:* In this case, one *proxy* image is created that exposes one (or more) ELG-compatible endpoints; the proxy container communicates with the actual LT service that runs outside the ELG infrastructure.
- *LT tool requiring an adapter:* This is a compromise between the standalone and remote approaches. A tool that is available as a Docker image but whose API is not natively ELG-compatible can be run alongside a separate ELG-compatible *adapter* image as a single pod in the ELG infrastructure. The adapter receives ELG API requests, communicates with the tool's native API in the pod, and translates the responses back to ELG format.

[10] https://gitlab.com

[11] https://hub.docker.com

[12] https://azure.microsoft.com/en-us/services/container-registry/

```
1  # Base image.
2  FROM openjdk:8-jdk-alpine
3
4  # SET TARGET DIRECTORY
5  ENV TARGETDIR /elg/
6  # This is required for wait.sh
7  RUN apk update && apk add bash
8
9  # Install tini and create unprivileged user
10 RUN apk add --no-cache tini && \
11       addgroup --gid 1001 "elg" && \
12       adduser --disabled-password --gecos "ELG User,,," \
13       --home /elg --ingroup elg --no-create-home --uid 1001 elg
14
15 # Create target directory
16 RUN install -d -o elg -g elg $TARGETDIR
17 # Copy everything to target directory
18 COPY --chown=elg:elg dockerCmd ${TARGETDIR}dockerCmd
19 # Copy/Rename server app jar.
20 ADD --chown=elg:elg  /elg-ilsp-lt-services-rest-simple-0.0.1-
       SNAPSHOT-exec.jar ${TARGETDIR}dockerCmd/app.jar
21
22 # Set working directory
23 USER elg:elg
24 WORKDIR ${TARGETDIR}dockerCmd
25
26 # Make sure script can be executed
27 RUN chmod +rx ./wait.sh
28
29 # The command that is run when the container starts
30 ENTRYPOINT ["sh", "runInContainer.sh"]
```

Listing 1 Example of a dockerfile for an integrated ELG LT service

4.1.4 Creation of Docker Images

The Docker image of an application contains the code of the tool and all dependencies required to run it, e. g., the operating system, frameworks, settings, configuration files and libraries etc. Containers are instantiations of images and can be thought of as lightweight virtual machines.

The process of packaging a service as a Docker image involves creating a dockerfile that describes the build process, running that build, and pushing, i. e., copying the resulting image to a Docker registry that is accessible to the ELG infrastructure. An example dockerfile is shown in Listing 1. The most important parts are:

- Line 2 states that an image containing a lightweight Linux-based operating system that includes Java programming language will be used as the base.
- Line 20 adds the Java-based application (.jar file) that exposes an ELG-compliant LT service to the image (see Section 4.1.5 for more details).

```
# Login to Gitlab container registry
$ docker login registry.gitlab.com

# Build the image and tag it with the name registry.gitlab.com/
    ilsp-nlpli-elg/elg-ilsp-lt-services and a version number
$ docker build -t registry.gitlab.com/ilsp-nlpli-elg/elg-ilsp-lt-
    services:1.0.0 .

# Push the image to the container registry
$ docker push registry.gitlab.com/ilsp-nlpli-elg/elg-ilsp-lt-
    services:1.0.0
```

Listing 2 Example sequence of commands to build and push a Docker image to a registry

- Line 30 specifies the script (.sh) that is run when a container is created from this image; this script starts the Java application.

A simple and robust way to build and store the image of a service in a registry is to put the service code into a source code repository such as GitHub[13] or GitLab, and then to use the repository's continuous integration (CI) mechanism. There are various examples of services built like this, i. e., using GitLab CI, in the ELG GitLab space.[14] Gitlab CI is triggered immediately after a commit on the repository or on demand and runs the build process specified in `.gitlab-ci.yml`.

An image can also be built and stored by running a set of commands locally. This option is helpful because CI services are often restricted, e. g., Gitlab has monthly quotas. In this case, users must first download the source code to a local folder (including the dockerfile), and then run a sequence of commands similar to Listing 2.

Some languages and build systems provide alternatives for building Docker images that do not require developers to write their own dockerfile, or to use Docker at all. For example, Java services based on the Micronaut[15] helper described below can use the Micronaut built-in `dockerPush` or `dockerPushNative` gradle tasks to build and push an image in one step using an automatically generated dockerfile, or Google Jib[16], which is designed specifically around the needs of Java applications and produces intelligently layered images that make more efficient use of space in the container registry. Additional files such as models can also be included.

To be deployed in ELG, a Docker image must meet the following requirements:

- It must be built for the `amd64` architecture (also known as `x86_64`); multi-architecture images may be appreciated by users who want to run the service on their own hardware, but ELG itself runs on `amd64`.
- It must be compatible with the Broadwell micro-architecture, which supports SSE4.2, AVX and AVX2 but *not* AVX512 instructions.

[13] https://github.com

[14] https://gitlab.com/european-language-grid

[15] https://micronaut.io

[16] https://github.com/GoogleContainerTools/jib

- The container must run in *at most* 6GB of RAM, but the smaller its footprint the better. By default, containers are limited to 512MB RAM; if the container requires more memory, this must be specified in the metadata record (using `additionalHWRequirements`). Services requiring more than 6GB are approved only in exceptional cases.
- It must be tagged with an explicit version number such as `:1.0.0`, not the implicit `:latest` tag which typically changes over time.
- The network socket on which the container listens for HTTP requests must bind to all the container's IP addresses (typically by using `0.0.0.0`). Some HTTP libraries only listen on the local loopback `127.0.0.1` by default, which will not be sufficient in ELG.
- Ideally the container should run without needing outgoing network connections to locations outside the hosting cluster. In particular, any model files must be cached within the image at build time, not downloaded at runtime from a repository such as Hugging Face. If outgoing network access is *required*, the target IP address ranges must be specified.

It is recommended for the service to only start listening once it is fully initialised and ready to start handling requests. If this is not possible (e. g., if the code requires some asynchronous initialisation process and the library used opens its sockets before that process is complete), then a separate "readiness" endpoint should also be provided at a separate URL path from the main service endpoint (typically `/elg-ready`) that returns the response code 503 ("service unavailable") if the service is not yet initialised, and 200 or 204 once it is ready to handle requests.

Sections 4.1.5 and 4.1.6 present Java- and Python-based libraries for easily creating an application that offers an ELG-compatible service. Some of these include utilities for creating the Docker image in which the service will be packaged.

4.1.5 Helper Libraries for Java

For LT service developers working in Java or other Java Virtual Machine (JVM) languages such as Groovy[17] or Kotlin[18], ELG provides helper libraries for two popular frameworks, Spring Boot[19] and Micronaut[20]. The programming style is similar in both cases, though Micronaut is better optimised towards creating smaller, lighter images with faster startup times, so if the service implementation does not already have a dependency on Spring, Micronaut is the recommended option. Both libraries depend on a common bindings library[21] of Java model classes that represent the various JSON message structures in a more Java-native way.

[17] https://groovy-lang.org

[18] https://kotlinlang.org

[19] https://spring.io/projects/spring-boot

[20] https://micronaut.io

[21] https://javadoc.io/doc/eu.european-language-grid/elg-java-bindings

An ELG-compatible LT service can be built in three steps[22] using Micronaut:

1. Create a blank Micronaut application using the Micronaut Launch tool.[23]
2. Add the ELG helper as a dependency, which is published to the central repository – for Gradle this means

```
implementation("eu.european-language-grid:lt-service-
    micronaut:1.0.0")
```

3. Create a controller that extends `LTService` (for services that process text-based requests) or `BinaryLTService` (for services that process requests with binary content) and implement the relevant handle or handleSync method.

The process[24] is similar for Spring Boot:

1. Create a blank Spring Boot application using the "Spring Initializr"[25] – additional dependencies are not needed, unless the specific code requires them.
2. Add the ELG helper as a dependency, which is published to the central repository – for Gradle this means

```
implementation("eu.european-language-grid:elg-spring-boot-
    starter:1.0.0")
```

3. Create one or more beans annotated `@ElgHandler`, with one or more public methods annotated `@ElgMessageHandler`. Each method should take an ELG request type such as `TextRequest` as a parameter (and for binary requests a second parameter of type `Flux<DataBuffer>` for the actual data) and return an ELG response type such as `AnnotationsResponse` or a reactive streams Publisher producing that type.

In both cases, Micronaut and Spring Boot, developers must add their code in the appropriate places to call the actual LT tool and build a response based on the tool's results, using the model classes, e. g., an `AnnotationsResponse` object in the case that the results are standoff annotations. Once the objects are created, the frameworks and libraries are able to automatically serialise them into ELG-compliant JSON response messages. Similarly, the frameworks automatically translate the received input JSON messages to objects that can be easily handled by the developer, e. g., in the Spring Boot case a "text" JSON request is deserialised to a `TextRequest` object.

4.1.6 Helper Tools for Python

Similar to Java, the ELG team provides helper tools to create an ELG-compatible service from a Python-based LT service. The helper tools are included in the ELG Pypi package presented in Chapter 3. The package provides two Python classes that

[22] https://gitlab.com/european-language-grid/platform/lt-service-micronaut

[23] https://micronaut.io/launch

[24] https://gitlab.com/european-language-grid/platform/elg-spring-boot-starter

[25] https://start.spring.io

```python
from elg import FlaskService
from elg.model import TextRequest, AnnotationsResponse
import langdetect

class ELGService(FlaskService):
    def process_text(self, request: TextRequest):
        langs = langdetect.detect_langs(request.content)
        ld = {}
        for l in langs:
            ld[l.lang] = l.prob
        return AnnotationsResponse(features=ld)

service = ELGService("LangDetection")
app = service.app
```

Listing 3 Example ELG service created using the FlaskService class of the ELG Python package

can be extended to create a simple HTTP server that exposes an ELG-compatible endpoint of the LT tool. The ELG Python package also comes with a command-line interface (CLI) that helps with the creation of the Docker image.

For the ELG-compatible endpoint, the developer creates a Python class extending either `FlaskService` or `QuartService` as a base class, and must implement one of the four following handler methods: `process_text`, `process_structured_text`, `process_audio` or `process_image`, depending on the required input type for the LT service. This method will contain the code of the LT tool, it takes as input an ELG request object of the relevant type and should return a valid ELG response object. As a simple example, Listing 3 shows an LT tool that detects the language of the input text. The `ELGService` class inherits from the `FlaskService` class, which already contains all the code needed to create the server. This allows the developer to focus on the LT tool by only having to define the handler method. The `FlaskService` and `QuartService` classes work the same way; the first is based on Flask[26], which is more suited to CPU-bound synchronous code, the second uses the asyncio-based Quart framework[27], which is better for I/O bound code – `QuartService` is the only supported option if the handler method uses `async`/`await`[28]. Both base classes support the progress reporting mechanism and correctly handle exceptions raised by the tool, mapping them to ELG-compliant failure responses.

After having defined the HTTP server compatible with the ELG LT internal API using the `FlaskService` or `QuartService` class, the next step is to create the Docker image. The ELG CLI that comes with the Python package contains the `elg docker create` command to help during this step. The command automatically generates the dockerfile based on the arguments. Listing 4 shows an example for the language detection service presented in Listing 3. All the available options of the

[26] https://flask.palletsprojects.com/en/2.0.x/

[27] https://pgjones.gitlab.io/quart/

[28] https://www.european-language-grid.eu/2021/10/04/choose-the-right-tool-to-create-your-elg-service-in-python/

```
elg docker create -n ELGService -p elg_service.py -r langdetect
```

Listing 4 CLI command to generate the dockerfile automatically

command are accessible with `elg docker create --help`. Once the dockerfile is generated, the creation and the publication of the Docker image follows the same process as described in Section 4.1.4.

The ELG documentation includes a complete tutorial on how to create an ELG-compatible service using the Python package.[29] With these helper tools, we seek to facilitate as much as possible the creation of an ELG-compatible service from an LT tool implemented in Python. Using the Python helper ensures that the resulting service follows best practice in terms of error handling, request parsing, etc. and the construction of the dockerfile. This makes the services deployed in the ELG infrastructure efficient and secure.

4.1.7 Metadata Requirements

In addition to the metadata requirements for tools and services (see Chapter 2), the metadata records of ELG-compatible services must also include a set of technical metadata that are necessary for their deployment in the platform:

- `dockerDownloadLocation`: location of the image with the LT service;
- `serviceAdapterDownloadLocation`: location of the adapter image (if any);
- `executionLocation`: REST endpoint at which the LT tool is exposed within the Docker image (`http://localhost:{port}{/path}`);
- `additionalHWRequirements`: can be used to specify hardware requirements for this tool beyond the default limits of 512MB RAM and one CPU core;
- We also recommend providing *sample data* on which the service produces sensible results. Sample data help speed up the validation process, and can be used through the trial UIs and the "Code samples" tab by consumers who want to test the service. Providers can upload a file with samples, add a URL where the samples are located, or simply add the data in a dedicated free text element.

Figure 8 shows the mandatory elements replicating the editor (with sections horizontally and tabs vertically); elements marked with an asterisk are mandatory, given certain conditions, or required depending on the presence of another value or element.

[29] https://european-language-grid.readthedocs.io/en/stable/all/A1_PythonSDK/TutoServiceIntegration.html

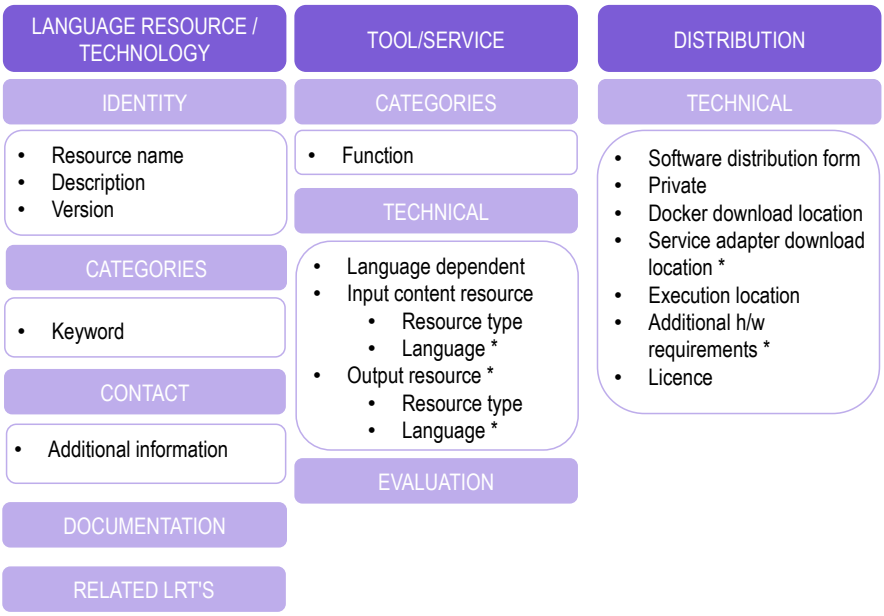

Fig. 8 Mandatory metadata for an ELG-compatible service

4.1.8 Technical Validation and Registration of ELG-Compatible Services

When LT providers have completed the packaging of their service, they can add it to ELG by supplying a metadata record via either the XML upload or editor mechanisms described in Section 2.1, specifying that it is an "ELG-compatible service" when prompted. Submitting the record initiates the validation process, which is performed internally by the ELG team.

The validation starts with the service registration process: The metadata or technical validator inspects the metadata record (accessed through the validator's grid) and deploys the service in the ELG Kubernetes cluster by creating the respective entries in the Helm charts that control the cluster. After that, the validator registers the service using a registration form (Figure 9), which specifies:

- Kubernetes-specific endpoint to be used by the LT execution server when calling the service, derived from the `executionLocation` metadata element value.
- ID of the trial UI to be used for rendering the processing results.
- Type of service (e. g., Speech Recognition, Text-to-Speech, Text Classification, etc.), which determines the appearance of the "Code samples" tab.
- Accessor ID that is used to form the public API endpoint URLs at which the service can be called. If the service was created as a new version of an existing service then it will share the same accessor ID as the service it replaces, but other than this, two distinct services must have different accessor IDs.

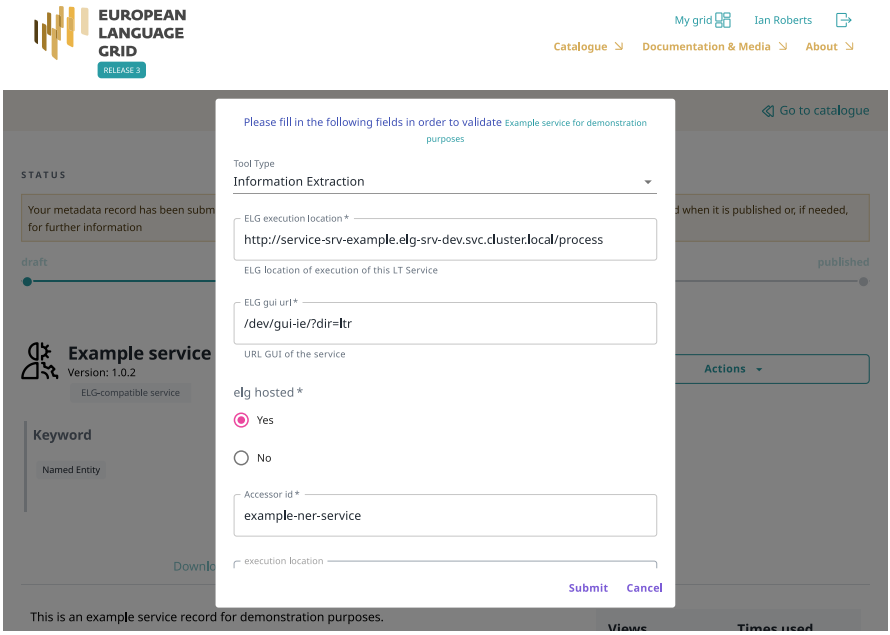

Fig. 9 Registration form for ELG-compatible LT services

When the registration is completed, the service is visible only to the validator and the provider. The technical validator and the provider check that the service behaves as expected using test input, and that the results it returns can be rendered adequately by the assigned trial UI – this is where good sample data is particularly useful. When required, the validator may communicate with the provider to recommend changes in the technical implementation of the service or metadata. When the service is finally running as it should the technical validator approves it; it will be published once it also receives approval from the legal validator (see Chapter 2 for more information on the ELG publication life cycle).

4.1.9 Custom Try Out Interface

The ELG-provided trial UIs[30] have been designed to support common service types in a generic way, but there may be specific services for which the standard UIs either do not work or do not represent the results in a particularly intuitive way. If this is the case, it is possible to supply an alternative trial UI that better suits the service to be

[30] https://gitlab.com/european-language-grid/usfd/gui-ie

```
// set up message listener
window.addEventListener('message', (e) => {
  if(e.origin ===
      'https://live.european-language-grid.eu') {
    const serviceInfo = JSON.parse(e.data);
    // configure UI here - store ServiceUrl and Authorization, fetch
    // parameter metadata from ApiRecordUrl, etc.
  }
});

// request configuration from the parent frame
setTimeout(() => {
  // the content of the message is unimportant, any message will trigger
  // the configuration reply.
  window.parent.postMessage("GUI:Ready for config",
          "https://live.european-language-grid.eu");
}, 500);
```

Listing 5 Typical JavaScript setup code for a trial UI

added. The standard UIs are open source under the Apache Licence[31], and providers are free to use this code as a basis for their own UI.

A trial UI is a single-page HTML/JavaScript application which is loaded into an `<iframe>` by the catalogue page when the user views an ELG-compatible service. Trial UIs run entirely in the browser and must not send user data to anywhere other than the ELG service endpoint and the i18n message resolver service. The JavaScript inter-frame messaging mechanism is used to supply the UI with the data it needs to configure itself for use with this particular service – when the UI `<iframe>` loads it must register a message listener that expects to receive message data that can be parsed as JSON, then dispatch a message to the parent frame to trigger the configuration message in return.[32] An example of this mechanism is shown in Listing 5.

The message event `data` sent by the parent frame will be JSON containing the following properties:

`ServiceUrl` The public LT service API URL at which the service can be called. The URL may include query string parameters if the service has more than one deployed version.

`ApiRecordUrl` The catalogue API URL from which the metadata record for this service may be retrieved with a `GET` request. This provides access to service parameter declarations, sample data, etc.

`Authorization` An HTTP `Authorization` header value that will authenticate calls to the `ServiceUrl` and `ApiRecordUrl` as the user who is logged in.

[31] https://www.apache.org/licenses/LICENSE-2.0

[32] To avoid the parent frame sending the configuration data before the UI frame is ready to receive it.

`Language` (optional) ISO code for the preferred language of the user. If present, this should be used as the `lang` parameter when resolving status messages to strings using the i18n resolver (see Section 4.1.1)

The custom UI can be hosted at any HTTPS URL – the `ServiceUrl` and `ApiRecordUrl` return the appropriate CORS headers to support cross-origin requests. Trial UIs run as Docker images in the ELG Kubernetes cluster. UIs can be created either by the ELG team or by a provider that needs a custom visualisation interface for the tools they contribute. Custom UIs can be integrated into ELG together with the ELG technical team.

4.2 ELG-hosted Resources

Together with metadata descriptions, providers are encouraged to upload the corresponding data files of their language resources so that they are readily available for download through ELG. To register their resources, they can select their preferred option from the ones presented in Section 2.1 and upload the accompanying files following the instructions in Section 2.2.

4.2.1 Requirements for ELG-hosted Resources

ELG requires data files to be uploaded as compressed ZIP files. There are no other specific metadata requirements apart from those defined for records of the resource type to which they belong (i. e., corpora, models, etc.). Chapter 2, Section 5, (p. 19 ff.) describes the metadata schema in more detail.

4.2.2 Packaging Data and Splitting Metadata Records: Recommendations

Datasets are composed of files that can be organised according to different criteria. For example, a multilingual corpus of texts from various domains can be described as a whole (one metadata record) or split into subsets (with corresponding metadata records) using the language or domain criteria. Depending on their intended use, different ways of packaging datasets and making them available can be suggested.[33]

We prepared a set of recommendations for the packaging of data files to enable users, especially those accessing ELG through programmatic APIs, to automatically identify, download and use corpora as is, without having to download them and manually search among them the subsets that interest them.[34]

[33] https://www.w3.org/TR/vocab-dcat-3 provides a similar argumentation for data distributions.

[34] These recommendations can be applied in different contexts, depending on whether the resource will be uploaded in ELG: when providers upload their corpora into ELG, they can use them to package the files and register the resource as one or multiple metadata records; if they decide to

The following cases are foreseen:

Multilingual resources are recommended to be split into bilingual pairs, so that users can easily find and use them, for example, in the case of bilingual corpora, to train bilingual models.

Resources from shared tasks are usually already split into training, development, gold, and test datasets, with a direct link to each of these. This is an established practice, and adopted in ELG as is. We recommend to register them as separate metadata records.

In both cases, a parent metadata record, to which the metadata records of all subsets can point is recommended using the "isPartOf" relation.

4.3 Metadata Records for External LRTs, Organisations and Projects

When external LRTs, organisations or projects are added to ELG, the only requirement for such metadata records is that they conform to the minimal version of the ELG metadata schema, i. e., they include the mandatory metadata elements described in Chapter 2, Section 5 (p. 19 ff.). Providers can use one of the options described in Section 2.1 (p. 69 ff.). For these records, the validation process aims to ensure that the metadata description is consistent and informative for users.

5 Provider-Related Functionalities in ELG and other Platforms

In this final section of the chapter we discuss some aspects of the functionalities offered to LT providers in ELG in relation to those available in other similar platforms. This discussion cannot be exhaustive. It rather attempts to give an overview of their design and implementation, highlight the main options utilised by the platforms, and offer explanations of the adopted approaches.

5.1 Metadata Requirements

Although the use of certain metadata schemas (e. g., DC[35], DCAT[36], schema.org[37], etc.) is growing, these schemas are usually restricted to the documentation of gen-

grant access to external corpora through hyperlinks, they can follow them for splitting the resource into one or multiple records and marking the availability through a direct link (element "downloadLocation").

[35] https://www.dublincore.org/specifications/dublin-core/dcmi-terms/

[36] https://www.w3.org/TR/vocab-dcat-3/

[37] https://schema.org

eral properties and do not satisfy domain- or community-specific requirements, especially with regard to discovery. Thus, most platforms use their own metadata schemas or ask for a minimum set of elements which are community-, domain-, or resource type specific (see Chapter 6 for a discussion of metadata schemas). Technical metadata are typically mandatory when resources are deployed in a platform. ELG has a detailed schema with a minimum set of required metadata to allow for flexibility and strictness when this is mandated for operational reasons (i. e., resources deployed in ELG, added by individuals, harvested from other sources).

CLARIN has initiated the Component MetaData Infrastructure[38], which provides a framework to describe and reuse different "metadata profiles" for resource types and communities. Specific metadata profiles, e. g., those of web services, are "recommended" with an aim to ensure interoperability and operational requirements. However, these profiles may promote different mandatory elements, depending on the use of the profile by each CLARIN Centre. Hugging Face[39] uses a dataset and model card, in which part of the required information is specified via YAML[40] tags.

5.2 Provider User Interface and Metadata User Interface

User-friendly editors that can cover *multiple* metadata schemas are difficult to implement, especially when the schemas have a complex structure. Nevertheless, most platforms include such an option. ELG, like META-SHARE[41] (Piperidis 2012; Piperidis et al. 2014), OpenMinTeD[42] (Labropoulou et al. 2018) and the European AI-on-demand platform[43], offer provider-specific UIs and a metadata editor supporting their respective schemas for describing resources. Hugging Face offers a rather simple UI with limited functionality. LAPPS Grid[44] (Ide et al. 2016) does not provide such UIs, a provider must communicate with the technical team in order to add services to the Galaxy[45] toolbox. Various CLARIN teams have created editors that support CMDI metadata (e. g., COMEDI[46], ARBIL[47], etc.). For more technical users, platforms offer APIs through which they can upload metadata records with JSON being the most widely used format for the records.

[38] https://www.clarin.eu/content/component-metadata
[39] https://huggingface.co
[40] https://huggingface.co/docs/datasets/v1.12.0/dataset_card.html
[41] http://www.meta-share.org
[42] https://openminted.github.io
[43] https://www.ai4europe.eu
[44] https://www.lappsgrid.org
[45] http://galaxy.lappsgrid.org
[46] https://clarino.uib.no/comedi/page
[47] https://portal.clarin.nl/node/14320

5.3 Try Out User Interface

Hugging Face offers embedded trial UIs to access their public "inference API". These are similar in spirit to the ELG "try out" UI mechanism, with a publicly documented API being called by a generic user interface. In addition, Hugging Face provides "Spaces"[48] which enable users to create and deploy their own UIs for demonstrating a model. The approach followed by Hugging Face Spaces is different from ELG; it is based on developers coding their own back-end server code and front-end UI as a single unit using the Streamlit[49] or Gradio[50] Python libraries. The developer adds this source code to a Git repository and Hugging Face then deploys the code to their infrastructure directly from the source code rather than from a developer-supplied Docker image. The UI is tightly coupled to the server-side code and the "API" is an implementation detail that varies from "space" to "space". ELG does not offer this kind of option by default, but the documented APIs mean that third parties could create a similar service on top of the LT services offered by ELG.

5.4 Helper Tools for Packaging Resources

As described in the previous sections, ELG offers command line utilities and SDKs for creating and submitting metadata for resources, preparing ELG-compatible services, etc. OpenMinTeD offered only a metadata validation service, without a corresponding command line tool. The European AI-on-demand platform, however, provides such utilities through Acumos[51] an open source framework, that makes it easy to build, share, and deploy AI applications.

5.5 Packaging Data Resources

ELG has adopted a lightweight policy for the packaging of uploaded datasets, given that direct deployment is currently not foreseen. In the CLARIN infrastructure, each centre has its own processes and recommended formats for uploaded resources, taking into account preservation or deployment purposes (e. g., submitting the resources to processing). Hugging Face maintains a detailed set of instructions for the upload of datasets and models, which is crucial for ensuring that they can be deployed.

[48] https://huggingface.co/spaces
[49] https://streamlit.io
[50] https://gradio.app
[51] https://www.acumos.org

6 Conclusions

ELG enables producers of language resources and language technology tools and services to upload, describe, share, and distribute their services and products as well as to describe their companies, academic organisations and projects. ELG offers to providers web-based user interfaces for describing LT resources or related entities with metadata records and provides them with functionalities for managing the life cycle of their assets; a billing component for commercial services and resources has been implemented (see Chapter 3, Section 6, p. 59 f.) and will be activated as soon as the ELG legal entity is in place (see Chapter 13). Providers of LT tools can exploit such functionalities to integrate LT tools in the ELG platform as ready-to-deploy services. LT data and tool providers are requested to follow the specifications and recommendations for packaging tools and resources to be uploaded in ELG. In the wider language technology ecosystem, provider-related functionalities are offered by other platforms, too, respecting their own target groups, objectives and policies. ELG has built bridges to some of these platforms, see Chapter 6 for more details.

References

Ide, Nancy, James Pustejovsky, Christopher Cieri, Eric Nyberg, Denise DiPersio, Chunqi Shi, Keith Suderman, Marc Verhagen, Di Wang, and Jonathan Wright (2016). "The Language Application Grid". In: *Worldwide Language Service Infrastructure*. Ed. by Yohei Murakami and Donghui Lin. Cham: Springer, pp. 51–70. DOI: 10.1007/978-3-319-31468-6_4.

Labropoulou, Penny, Dimitris Galanis, Antonis Lempesis, Mark Greenwood, Petr Knoth, Richard Eckart de Castilho, Stavros Sachtouris, Byron Georgantopoulos, Stefania Martziou, Lucas Anastasiou, Katerina Gkirtzou, Natalia Manola, and Stelios Piperidis (2018). "OpenMinTeD: A Platform Facilitating Text Mining of Scholarly Content". In: *Proceedings of WOSP 2018 (co-located with LREC 2018)*. Miyazaki, Japan: ELRA, pp. 7–12. URL: http://lrec-conf.org/workshops/lrec2018/W24/pdf/13_W24.pdf.

Labropoulou, Penny, Katerina Gkirtzou, Maria Gavriilidou, Miltos Deligiannis, Dimitris Galanis, Stelios Piperidis, Georg Rehm, Maria Berger, Valérie Mapelli, Michael Rigault, Victoria Arranz, Khalid Choukri, Gerhard Backfried, José Manuel Gómez Pérez, and Andres Garcia-Silva (2020). "Making Metadata Fit for Next Generation Language Technology Platforms: The Metadata Schema of the European Language Grid". In: *Proceedings of the 12th Language Resources and Evaluation Conference (LREC 2020)*. Ed. by Nicoletta Calzolari, Frédéric Béchet, Philippe Blache, Christopher Cieri, Khalid Choukri, Thierry Declerck, Hitoshi Isahara, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis. Marseille, France: ELRA, pp. 3421–3430. URL: https://www.aclweb.org/anthology/2020.lrec-1.420/.

Piperidis, Stelios (2012). "The META-SHARE Language Resources Sharing Infrastructure: Principles, Challenges, Solutions". In: *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*. Ed. by Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Mehmet Uğur Doğan, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis. Istanbul, Turkey: ELRA.

Piperidis, Stelios, Harris Papageorgiou, Christian Spurk, Georg Rehm, Khalid Choukri, Olivier Hamon, Nicoletta Calzolari, Riccardo del Gratta, Bernardo Magnini, and Christian Girardi (2014). "META-SHARE: One year after". In: *Proceedings of the 9th Language Resources and Evaluation Conference (LREC 2014)*. Ed. by Nicoletta Calzolari, Khalid Choukri, Thierry Declerck,

Hrafn Loftsson, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis. Reykjavik, Iceland: ELRA, pp. 1532–1538. URL: http://www.lrec-conf.org/proceedings/lrec2014/pdf/786_Paper.pdf.

Rehm, Georg, Stelios Piperidis, Kalina Bontcheva, Jan Hajic, Victoria Arranz, Andrejs Vasiļjevs, Gerhard Backfried, José Manuel Gómez Pérez, Ulrich Germann, Rémi Calizzano, Nils Feldhus, Stefanie Hegele, Florian Kintzel, Katrin Marheinecke, Julian Moreno-Schneider, Dimitris Galanis, Penny Labropoulou, Miltos Deligiannis, Katerina Gkirtzou, Athanasia Kolovou, Dimitris Gkoumas, Leon Voukoutis, Ian Roberts, Jana Hamrlová, Dusan Varis, Lukáš Kačena, Khalid Choukri, Valérie Mapelli, Mickaël Rigault, Jūlija Meļņika, Miro Janosik, Katja Prinz, Andres Garcia-Silva, Cristian Berrio, Ondrej Klejch, and Steve Renals (2021). "European Language Grid: A Joint Platform for the European Language Technology Community". In: *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations (EACL 2021)*. Kyiv, Ukraine: ACL, pp. 221–230. URL: https://www.aclweb.org/anthology/2021.eacl-demos.26.pdf.