EUROPEAN LANGUAGE GRID

D1.2

Base Infrastructure (first release)

Author:	Maria Moritz (DFKI), Florian Kintzel (DFKI), Ela Elsholz (DFKI), Georg Rehm (DFKI), Ian Roberts (USFD)

Dissemination Level:	Public
Date:	31-05-2019

About this Document

Project	ELG – European Language Grid
Grant agreement no.	825627 – Horizon 2020, ICT 2018-2020 – Innovation Action
Coordinator	Dr. Georg Rehm (DFKI)
Start date, duration	01-01-2019, 36 months
Deliverable number	D1.2
Deliverable title	Base Infrastructure (first release)
Type	Report
Number of pages	20
Status and version	Final – Version 1.0
Dissemination level	Public
Date of delivery	Contractual: 31-05-2019 – Actual: 31-05-2019
WP number and title	WP1: Grid Platform – Base Infrastructure
Task number and title	Task 1.2: Installation, setup and adaption of the base infrastructure
Authors	Maria Moritz (DFKI), Florian Kintzel (DFKI), Ela Elsholz (DFKI), Georg Rehm (DFKI), Ian Roberts (USFD)
Reviewers	Erinc Dikici (SAIL), Gerhard Backfried (SAIL), Jana Hamrlova (CUNI)
Consortium	Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Germany
	Institute for Language and Speech Processing (ILSP), Greece
	University of Sheffield (USFD), United Kingdom
	Charles University (CUNI), Czech Republic
	Evaluations and Language Resources Distribution Agency (ELDA), France
	Tilde SIA (TILDE), Latvia
	SAIL LABS Technology GmbH (SAIL), Austria
	Expert System Iberia SL (EXPSYS), Spain
	University of Edinburgh (UEDIN), United Kingdom
EC project officers	Philippe Gelin, Alexandru Ceausu
For copies of reports and other ELG-related information, please contact:	DFKI GmbH European Language Grid (ELG) Alt-Moabit 91c D-10559 Berlin Germany Dr. Georg Rehm, DFKI GmbH georg.rehm@dfki.de Phone: +49 (0)30 23895-1833 Fax: +49 (0)30 23895-1810 http://european-language-grid.eu

Table of Contents

List of Figures

List of Acronyms

API	Application Programming Interface
CI	Continuous Integration
HPC	Horizontal Pod Autoscaler
LR	Language Resource
LT	Language Technology
MQ	Message Queue
MVP	Minimum Viable Product
PO	Product Owner
TSC	Tools, Services, Components

Summary

This report outlines the progress of planning and developing the ELG infrastructure. After the introduction (Chapter 1), Chapter 2 presents the selected infrastructure provider and the details of the contract with the provider (see D1.1 for more details). Chapter 3 reports on the Task Force Infrastructure meetings, their discussions and outcomes. Details on the architecture and ways to access it, made available by the chosen provider, are presented in Chapter 4. Chapter 5 provides details on the design of the next integration steps and documents integration steps already undertaken. Chapter 6 consists of a brief user manual on how to get started with the infrastructure. Finally, in Chapter 7, the document reports on the hackathon which was conducted at DFKI Berlin on 20–24 May 2019 to start setting up a preliminary minimum viable product (MVP). This MVP is envisioned to be presented at the first annual ELG conference, i.e., at META-FORUM 2019 conference on 8/9 October 2019 in Brussels, Belgium.

1 Introduction

This report is a direct follow up to ELG Deliverable D1.1 ("Requirements and architectural specification of the base infrastructure"), in which we presented, among others, key design choices for the ELG base infrastructure and some of the foundational technical concepts the ELG platform will be based upon. Here, the present ELG Deliverable D1.2 picks up and describes the progress made with regard to the base infrastructure and putting together a first rudimentary but working end-to-end system.

Great effort (in terms of team size, time or money) does not necessarily guarantee the success of a large software development project, especially when many different modules and a distributed group of development parties are involved. To avoid misunderstandings in communication, misleading task descriptions, and fuzzy definitions of programming interfaces, we regularly schedule meetings in the ELG Task Force Infrastructure, established during the ELG kick-off meeting in January 2019, and discuss the progress of all work packages that directly concern the base infrastructure. In these meetings, the partners report on the progress of their tasks and explain details on how a task was or is carried out. We also exchange drafts of architectural diagrams and other working materials, and discuss details on the design of modules that the infrastructure needs to run in a reliable manner. In what follows, we report on recent progress and summarize the next steps.

2 External Cloud Service Provider

2.1 Description of the Selected Cloud Service Provider SysEleven

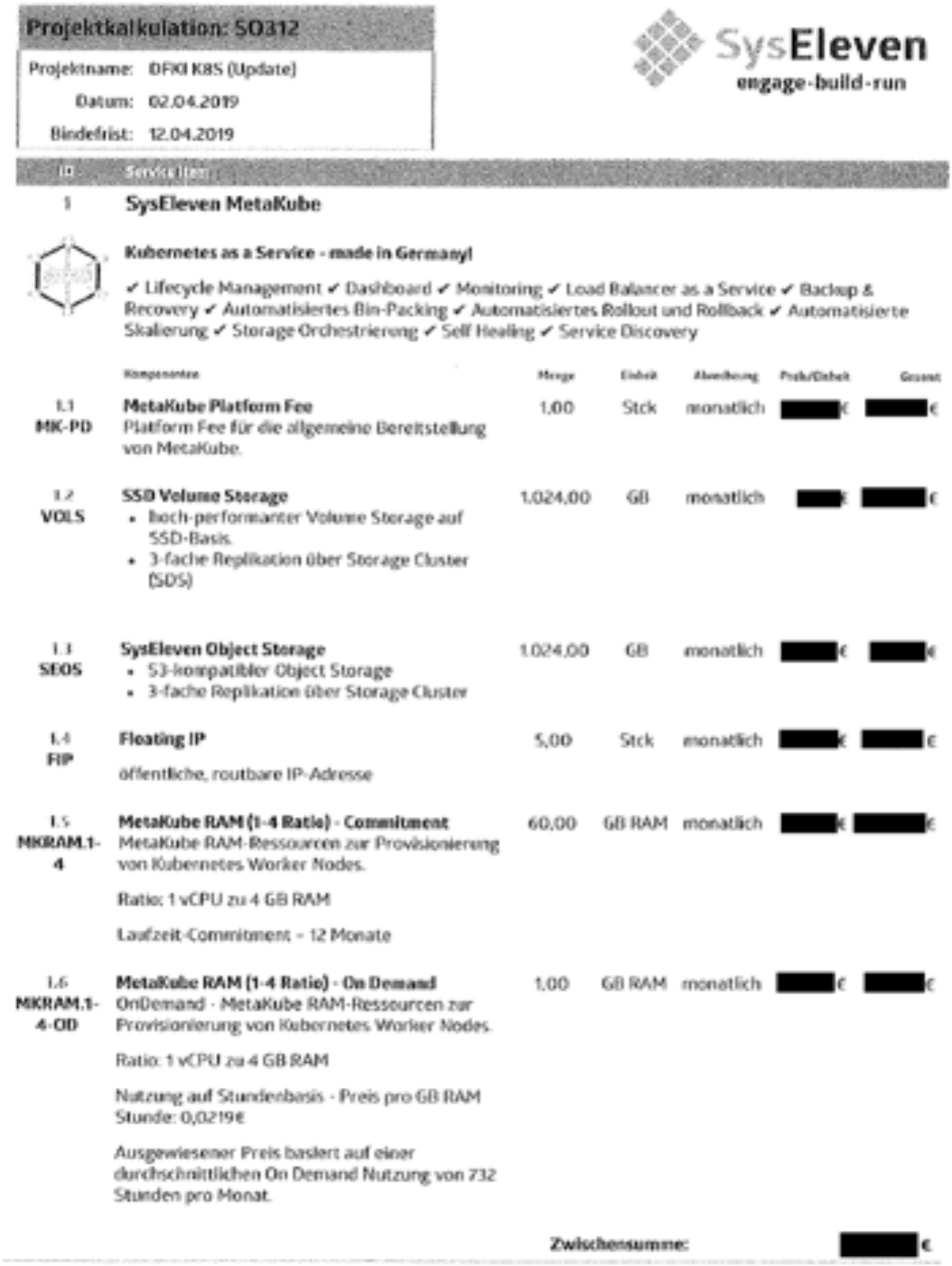
Founded in 2007, SysEleven is one of the most experienced cloud providers in Germany that offers Kubernetes as a Service. They offer in-house hosting using their own fibre optics city network and operate two data centres in Berlin. Since 2015, SysEleven has been operating the SysEleven Stack, which is their own cloud stack solution based on OpenStack, hence having their own Infrastructure as a Service.

2.2 Hardware Configuration

ELG partner DFKI signed a contract for the ELG development cluster for 12 months (see Figure 1). Its hardware comprises of the MetaKube platform, which is the managed Kubernetes solution provided by SysEleven.[1] We

[1] See: https://docs.syseleven.de/metakube/de

ordered 60GB RAM per month with nodes that are equipped with a 1:4 cores:RAM ratio while additional nodes/GB RAM can be booked dynamically on top. We also booked 1TB of SSD storage, and 1TB of S3 storage.



Projektkalkulation: SO312

Projektname: DFKI K8S (Update)
Datum: 02.04.2019
Bindefrist: 12.04.2019

SysEleven
engage-build-run

Pos	Service Item	Komponente	Menge	Einheit	Abrechnung	Preis/Einheit	Gesamt
1	**SysEleven MetaKube**						
	Kubernetes as a Service - made in Germany! ✔ Lifecycle Management ✔ Dashboard ✔ Monitoring ✔ Load Balancer as a Service ✔ Backup & Recovery ✔ Automatisiertes Bin-Packing ✔ Automatisiertes Rollout und Rollback ✔ Automatisierte Skalierung ✔ Storage Orchestrierung ✔ Self Healing ✔ Service Discovery					
1.1 MK-PD	**MetaKube Platform Fee** Platform Fee für die allgemeine Bereitstellung von MetaKube.	1,00	Stck	monatlich	[redacted]€	[redacted]€
1.2 VOLS	**SSD Volume Storage** • hoch-performanter Volume Storage auf SSD-Basis. • 3-fache Replikation über Storage Cluster (SDS)	1.024,00	GB	monatlich	[redacted]€	[redacted]€
1.3 SEOS	**SysEleven Object Storage** • S3-kompatibler Object Storage • 3-fache Replikation über Storage Cluster	1.024,00	GB	monatlich	[redacted]€	[redacted]€
1.4 FIP	**Floating IP** öffentliche, routbare IP-Adresse	5,00	Stck	monatlich	[redacted]€	[redacted]€
1.5 MKRAM.1-4	**MetaKube RAM (1-4 Ratio) - Commitment** MetaKube RAM-Ressourcen zur Provisionierung von Kubernetes Worker Nodes. Ratio: 1 vCPU zu 4 GB RAM Laufzeit-Commitment - 12 Monate	60,00	GB RAM	monatlich	[redacted]€	[redacted]€
1.6 MKRAM.1-4-OD	**MetaKube RAM (1-4 Ratio) - On Demand** OnDemand - MetaKube RAM-Ressourcen zur Provisionierung von Kubernetes Worker Nodes. Ratio: 1 vCPU zu 4 GB RAM Nutzung auf Stundenbasis - Preis pro GB RAM Stunde: 0,0219€ Ausgewiesener Preis basiert auf einer durchschnittlichen On Demand Nutzung von 732 Stunden pro Monat.	1,00	GB RAM	monatlich	[redacted]€	[redacted]€
					Zwischensumme:	[redacted]€

Figure 1: Excerpt of the contract with SysEleven for the development cluster

3 Task Force Infrastructure Meetings

We organize bi-weekly meetings of the Task Force Infrastructure. Participants are all partners contributing to WP1 to WP4 that contain infrastructure matters. These topics affect, essentially, the whole ELG consortium. Primary participants of the Task Force Infrastructure are DFKI, ILSP, USFD and TILDE.

3.1 Meeting on 27 March 2019

We started with the first **Architecture Development Roadmap** as an overview to start from a common ground. Concerning the **Service Invocation Process**, we want to avoid a high number of potential services to be deployed simultaneously (to not waste resources). We agreed on instantiating them via just-in-time kept-alive for n minutes, to be specified in the configuration file. In the **Technical Service Interface**, we proposed to split a service invocation interface (e.g., command-line execution as separate docker) from the external (REST) API. Then, all services need to comply with this interface to be deployed. This interface should be located inside a base docker image that services would need to inherit from. Further, until external APIs are specified for all the services, a generic interface could be set up in the form of "call service x with params y, z". For the **Request Workflow**, a Message Queue (MQ)[2] solution for buffering incoming requests and dispatching them to worker nodes seems a good solution. Handling of the MQ would likely be possible inside the docker base image.

Partners asked for **demo workflows** to create and deploy an example service. Next, **version handling** can be covered via a parameter in the Kubernetes file to support rolling updates. The config files need to be updated with new incoming images. The fact that a follow-up version does not necessarily cover all the functionality of earlier versions was mentioned. Hence, we consider introducing a backward compatibility flag. We agreed on a compound naming rule in which parameters represent versioning. We further need guidelines/updates documented on operational change sets and we can use ISLRN for identification purposes. The question on how to keep older versions so that we can wake them up was postponed to a later stage when version becomes more urgent. We also consider stress tests for the versioning control in a separate grid instance. **Incorporation of language** resources (LR) are required for both training models and accessing the models from language technology (LT) packages. We use S3 object storage compatible space. Partners asked whether models should be stored as separate docker images. However, in general a service should have APIs to access models. Finally, next steps of each partner are quickly communicated.

3.2 Meeting on 16 April 2019

We updated everybody on the **infrastructure design** (see also D1.1) and the selection of the cloud service provider. Everything is settled, only one last administrative step is missing within DFKI. The **next steps** to make the infrastructure ready are the creation of user accounts and to manage user authorization settings, the creation of a release plan, definition of a service execution plan (offering a test environment), and writing the documentation. We further want to clarify briefly and in the long run how does the integration of a resource happen and if the UI will be integrated as a separate container. We also discussed the MVP requirements and how contributions as well as services will be integrated into the MVP.

Big models might be mounted on drives separately. Is a shared network file storage a solution, is this available in the development instance or do we need to implement this ourselves? Partners suggested Alpine (a small version of Linux) as the operating system to reduce the resources needed. Further, the issue of containerizing LT or storing them in a shared manner was raised. This is relevant for technologies with large models. We further discussed a list-driven processing of tasks vs. waiting for **MQ requests**. We might consider a language cache that would be covered by a MQ. Partners asked whether containers can access **Licence Servers**, which is necessary for loading the respective services upfront. Further, the location of these license servers is not trivial.

[2] A message queue is an infrastrucutural software that handles the message communication among multiple components of a software architecture. Its main tasks are the identification of which LT service a message is forwarded to or pulled from.

3.3 Meeting on 30 April 2019

Topics mainly concerned the preparation of the **Hackathon** (see below). We discussed what the milestones are that everybody wants to complete in the week of the hackathon and which work packages need to collaborate more intensely than others. It was agreed to have a first version of the infrastructure with preliminary services up and running and an integration and deployment "how to" ready by the end of the hackathon.

With regard to the **Hackathon**, we agreed to set up a first working end-to-end system. We discussed the next steps to prepare the grid according to package dependencies, user credentials and ingress controller. The latter enables the exposure of deployed services directly via REST (see also https://gitlab.com/european-language-grid/platform/platform-project/boards).

3.4 Summary

The Task Force meetings of the first months have been productive and very helpful with regard to the integration of all platform parts towards a first version of a working system. We discussed and agreed how a service is to be invoked, how the service invocation interface is split into a platform-related part on the one hand, and a service-specific part on the other. Further discussion topics include how the request workflow via MQ shall be handled, how big resources are to be integrated, and, finally, how we will set up the MVP.

4 Grid Architecture

4.1 Component-driven Architecture

SysEleven's MetaKube Platform allows us to deploy a standardized managed Kubernetes cluster. Figure 2 shows an overview of the architecture in a SysEleven cluster. MetaKube provides user-friendly management features of the cluster. The SysEleven stack supports the visualization of statistics on the nodes, containers, and pods, as well as storage resources in a comfortable and flexible way.[3]

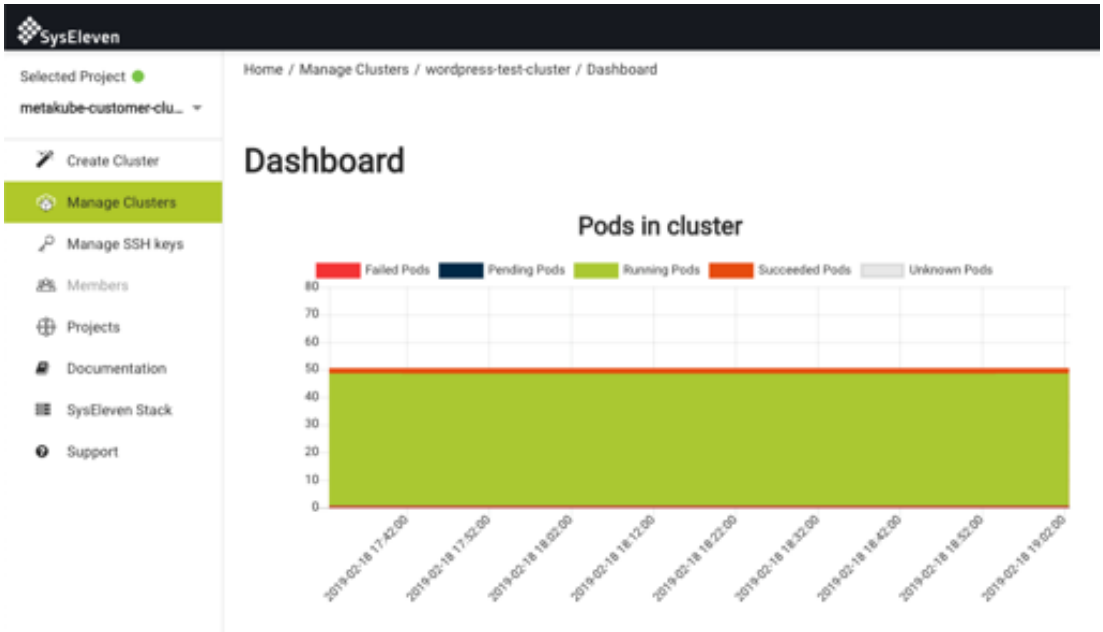


Figure 2: MetaKube Dashboard

[3] Taken from https://docs.syseleven.de/metakube/de/documentation/metakube-dashboard

4.1.1 Control Panel and Dashboard

The MetaKube Dashboard is available after logging in to the SysElvens's Stack. Every MetaKube cluster comes with a dashboard in the MetaKube web interface that shows the current state of the Kubernetes cluster, for example how many pods are running or how the cluster is used (see Figure 3).

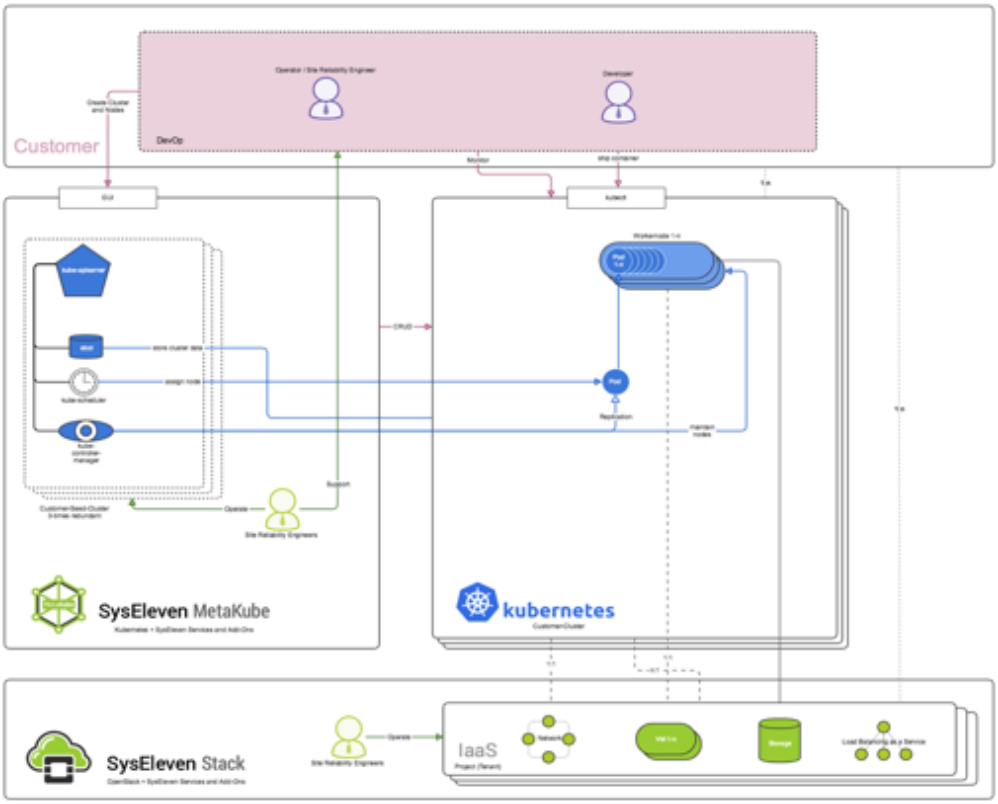


Figure 3: Overview of SysEleven's managed Kubernetes platform MetaKube

4.1.2 Package Manager and Helm Chart Repository

We use Helm as the package manager for our Kubernetes configuration.[4] Helm is a tool for managing Kubernetes charts. Charts are bundles of pre-configured Kubernetes resources (e.g., services or other components running in the grid) that help to organize a smooth interaction of grid components.

4.2 Rolling Development of the Infrastructure

This section describes the next steps to set up the initial grid, prepare configuration files and a workflow for (semi-)automatically deploying both, the core components of the grid (Kubernetes base cluster, ingress controller etc.), and the integration mechanism to add a new tool. The following tasks are stored in our GitLab projects in the form of tickets that have different states, ranging from *implemented* over *ongoing* up until *planned*.

4.2.1 User Credentials for the Development Cluster

The Kubernetes cluster will be used to host hundreds of LT services provided not only by the partners of the ELG consortium but also by other companies and research centres that develop LT services. The inclusion of new, incoming containerised services in the ELG platform is, technically, quite a challenge because it involves

[4] https://github.com/helm/helm

not only the registration of the service in the ELG catalogue but also in the Kubernetes cluster, which requires some form of access to the cluster, either by the service developer or by the ELG administration team.

If the service developer is supposed to be able to access the Kubernetes development cluster to access logs and debug their running services, several aspects need to be taken care of.

- A documentation explains how service developers can request access to the cluster.
- A workflow enables service developers to create and securely store authentication tokens.
- A secure method delivers the authentication tokens to the developer.
- A methods assigns access rights for different user groups (e.g., grid admins, service developers).

Access to the development cluster utilizes Kubernetes' default three-layer access control (Figure 4): [5]

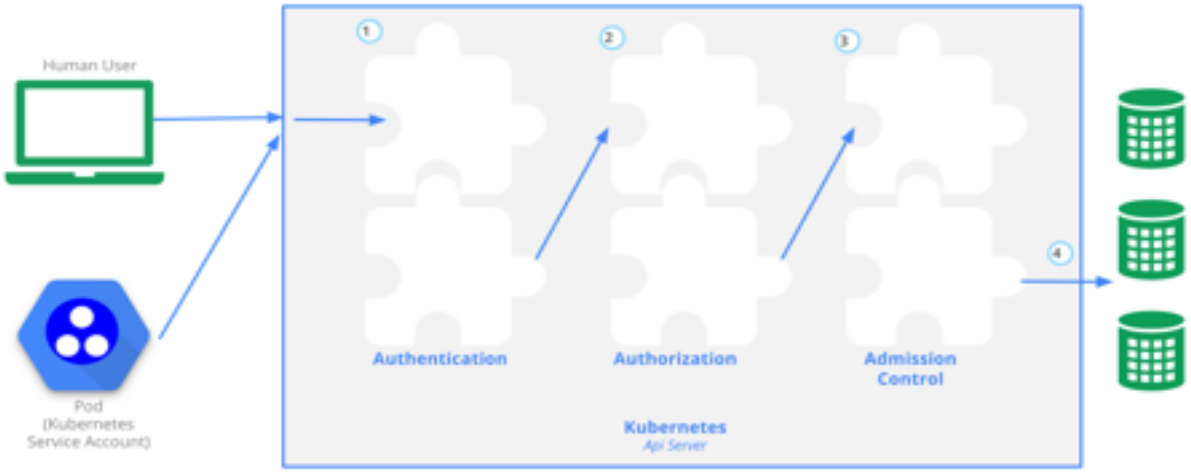


Figure 4: Controlling access to the Kubernetes API

User credentials are provided to the set of active developers in the form of *kubeconfig* files that include the certificate used for signing the credential (Transport security), the username (Authentication), the group the user belongs to (Authorization/Admission control).

The cluster is configured to use RBAC authorization (role-based access control). As of the current state of play, two roles are defined:

- Service Provider (*srv-provider-group*): has full access to the namespace *elg-srv-dev* (see below), but to no other namespaces
- Admin: has full access to all namespaces

Credentials will be distributed to the service developers via their ELG intranet accounts. Currently, no external Authentication provider is configured, so Kubernetes *service accounts* are used as user accounts.

4.2.2 Namespaces

Namespaces are used on the development cluster to distinguish between logical parts of the cluster and are also utilized for access control. Curently, the following namespaces exist:

[5] https://kubernetes.io/docs/reference/access-authn-authz/controlling-access/

- *elg-core-dev:* intended for the core part of the grid platform, e.g., for components that are utilized by all LT services as well as supporting tools
- *elg-srv-dev:* intended to be used for all LT service components
- *webterminal:* reserved by SysEleven GmbH to enable browser-based shell access to the cluster
- *velero:* reserved by SysEleven GmbH for their backup solution
- *default:* available by default in any Kubernetes system and not utilized so far in ELG
- *kube-public:* also available by default on any Kubernetes cluster; not used so far in ELG
- *kube-system:* reserved for Kubernetes-internal components

4.2.3 Initial Continuous Integration of the Grid

In the context of ELG, continuous integration primarily means to automate the insertion or update of services. We address this by offering a range of scripts that insert the right parameters and configurations in the right places. The grid administrators store the necessary configuration files in one place to set up the initial version of the ELG. Other necessary tools and the relevant documentation is stored in the ELG GitLab repository.

We provide an initial set of configuration files in GitLab together with a simple script that sets up the dev cluster, given the corresponding credentials (the execution of the script shall be idempotent). We also have a complementary script to destroy the dev cluster again. The described tooling supports multiple clusters (i.e., development, integration, production). These facilities can be used to extend the configuration so that an additional cluster can be set up when specified, and the right configuration is automatically selected.

4.2.4 Establishing a Development Workflow for the Grid

We are currently in the process of preparing an automated workflow for deploying the grid. A first script to homogenize configuration is in place. Its practical use for different scenarios still remains to be tested.

We want to provide developers of the grid platform infrastructure with documentation that explains how the development of the platform is coordinated. We are currently working on the following aspects and sets of information that will be thoroughly documented and then put into practice:

- Branching policy
- Merge request handling
- Commit message policy
- Code guidelines
- Preferred development language (for now we use bash and Python)
- Basic sprint planning (including organisational and review meetings)
- Roles for the scrum master and product owner to be defined

4.2.5 Autoscaling to Zero

We plan to implement a prototype configuration bundle that allows to configure the Horizontal Pod Autoscaler (HPA) so that it allows scaling down to zero replicas. It can use a custom metric for this kind of downscaling. While the custom metric is, for now, not relevant, later we will use the queue length as a base parameter to estimate scaling behaviour.

4.2.6 Configure Ingress Controller to Expose Services via REST

We will enable service developers to expose their services directly via the ingress controller unless or until an asynchronous way of exposure is implemented. The ingress controller will be configured to expose the service API under /services/.

4.2.7 Initial S3 Storage Configuration

Resource providers will be enabled to upload their LT or LR into the grid as a resource. To achieve this, S3 accounts will be configured to allow the group of service developers easy access to upload their resources into the grid. For the *dev* stage, no differentiation of user groups is necessary. For now, access to resources is in the responsibility of the service that needs the resource, i.e., the service downloads the data via S3. We currently require resources that are uploaded to be open source.

4.2.8 Basic Implementation of MQ and the Service Adapter

For service developers we want to have a basic version of the message queue and service adapter in place so that services can be called asynchronously. This solution includes the following steps:

- A basic MQ system is deployed to the grid.
- A basic version of the service adapter is implemented and connected to an example service.
- A basic version of the generic service API is implemented and deployed to the grid.
- We need to collect definitions of the policy that service developers have to follow, and of the interface structure that service developers need to adhere to in order to use the system.
- An example implementation has to be prepared as a demo for the consortium members.

5 Application Programming Interface

The following description and JSON excerpts are taken from the Platform APIs documentation hosted in our GitLab.[6] The API is still in an early state and cannot be regarded as final. It is supposed to give the reader a rough understanding on the progress of how we plan to orchestrate communication in the grid.

The APIs that we describe here are considered to communicate only between the ELG portal front end and the containerised Tools, Services, Components (TSC). Individual services do not need to concern themselves with authentication and access control, they can assume that calls they receive are permitted.

5.1 Processing of Messages

Pods for each service will be configured with the input and output queue names. Pods further must listen on the input queue for messages to process, and must send response messages to the output queue. Output messages are not required to be emitted in the same order as their corresponding inputs. A service may process messages one at a time or it may consume a batch of messages, process them all as a single unit, and then send a batch of responses. Messages are in a JSON-based message format suitable for transport over HTTP and/or message queue.

For TSCs that do not need to care about the complexity of handling the message queues, ELG will provide a service adapter in the form of helper images that deal with the queue and translate the messages into simple HTTP or stdin/out communication with the underlying component container. Where possible this will re-use the same message structures as in the main MQ interaction style.

5.2 Message Examples as Defined by the API

In the following, we present a selection of message types from the initial API available in the GitLab repository.

[6] https://gitlab.com/european-language-grid/platform/elg-apis/blob/master/doc/T2.5-api-design.md

5.2.1 Basic Request Message

Example:

```
{
  "metadata":{
    "id":"request identifier"
  },
  "request":{
    "type":"Request type code",
    "params":{ /* vendor-specific arbitrary JSON parameters */ },
    // other properties are type-specific
  }
}
```

- metadata.id: unique identifier for this request, to link progress and response messages to this request
- request.type: the type of the request, typically this will be a constant for a given service

MQ-specific metadata such as a reply-to address and MQ-level correlation ID may be provided out of band in the standard way for that queue broker.

5.2.2 Failure Message

Example:

```
{
  "metadata":{
    "id":"id of the request"
  },
  "failure":{
    "errors":[array of status messages]
  }
}
```

A failure in the service that causes it to be unable to return any meaningful results for a given request ID.

5.2.3 Success Message

Example:

```
{
  "metadata":{
    "id":"id of the request"
  },
  "response":{
    "type":"Response type code",
    "vendorData":{ /* vendor-specific arbitrary JSON data */ },
    // other properties type-specific
  }
}
```

As with the request, the response type code will likely be constant for any given service. The exact format of a successful response message depends on the type of the TSC.

5.2.4 Text Request (Service-specific message type)

This request message is for services that take plain text (or something from which plain text can be extracted, such as HTML and PDF) as their input. We expect that across the ELG from amongst the large number of possible and supported document types, a set of a smaller number of document types will emerge as being preferred and well supported (e.g., plain text, HTML, XML, PDF).

```
{
  "metadata":{...},
  "request":{
    "type":"text",
    // either content or contentLocation but not both
    "content":"The content, as a string inline",
    "contentLocation":"URI from which the content can be downloaded",
    // mimeType and encoding optional - given are the defaults
    "mimeType":"text/plain",
    "characterEncoding":"UTF-8",
    "params":{...} /* optional */
  }
}
```

Documents have to be representable in text and be short enough to fit within the MQ size limits. They may be included directly in the JSON as payload (“content”). Documents that are either too large to be included directly, or in a binary format that is not suitable for inclusion within JSON, will be placed at an HTTP(S)-accessible storage location given by the “contentLocation”.

6 Technical Documentation

6.1 Installing Docker and Kubernetes

Installing Docker and Kubernetes (and other packages) is only necessary when a user or developer wants to run the cluster locally, for example, for testing or debugging purposes. Detailed documentation for Linux and Windows is available under:

https://gitlab.com/european-language-grid/platform/infra/wikis/home

6.2 Logging in to the Cluster

Contributors to the grid and members of the ELG consortium will receive access credentials for the development cluster by the DFKI team. A detailed documentation that explains how to install the necessary software and/or using the interfaces that SysEleven offers to deploy and manage the grid is available under:

https://gitlab.com/european-language-grid/platform/infra/wikis/openstack-hints

and will be updated continuously. If a user wants to log in to a cluster via command line, *kubectl* is needed:

https://kubernetes.io/docs/tasks/tools/install-kubectl/

6.3 Deploy a Service

A core configuration (Helm chart) file is available under:

https://gitlab.com/european-language-grid/platform/infra/tree/master/elg-core

It contains parameter information for deploying the main skeleton of the cluster. Further, each service has its own specific helm chart:

https://gitlab.com/european-language-grid/platform/infra/tree/master/elg-srv

Deployment is managed, for now, by the DFKI team.

6.4 The DFKI Team

The DFKI team is responsible for all questions regarding the infrastructure and integration of components. In case of questions and issues, please consider sending an email to the Task Force Infrastructure mailing list to ease communication:

task-force-infrastructure@european-language-grid.eu

The members of the DFKI team are:

- Florian Kintzel (florian.kintzel@dfki.de)
- Maria Moritz (maria.moritz@dfki.de)
- Ela Elsholz (ela.elsholz@dfki.de)

7 ELG MVP and the first ELG Hackathon (20-24 May 2019)

Representatives of five ELG partners participated in the hackathon, which took place at DFKI Berlin from 20-24 May 2019. These were the primary technical partners DFKI, ILSP, USFD and TILDE who worked towards integrating core elements into the platform such as the GUI, the catalogue database and the API, as well as EXPSYS who worked on integrating their own services. The core participants were Maria Moritz, Florian Kintzel, Ela Elsholz (DFKI), Ian Roberts (USFD), Andis Lagzdiņš (TILDE), Dimitris Galanis, Miltos Deligiannis (ILSP), Andres Garcia, Cristian Berrio (EXPSYS).

7.1 Overall Goal – Development of the MVP

ELG organises three annual conferences (see WP7). The first one will take place on 8/9 October 2019 in Brussels, Belgium. At this event we will present an initial prototype of the ELG platform with first implemented functionality, such as a try-out area in the graphical user interface in which several running services can be tested, a first version of the ELG catalogue as well as first implementations of the user management, authentication (together with access rights to services) and an API that allows interaction between the user interface and the services. We agreed on this functionality scope during the first technical kick-off meeting on 18/19 February 2019 (see Figure 1 in D1.1). In the following, we will briefly present additional details regarding this MVP.

7.1.1 Scenario of the Demo

Assume, a person *P*, e.g., working for an e-commerce company, uses the ELG to look for a specific service *S* (say a machine translation service from language *A* to language *B*). As a result of the search, *P* is presented with a list of services to choose from. This may be a list of ten different MT services by different providers (one of those services needs to be actually working). *P* chooses this working service and is directed to a detailed view.

There, *P* will test the service in its trial GUI. An API description for the service exists and explains how to run the service from *P*'s custom code on her machine. *P* copies the offered code snippets into her IDE/editor and calls the service from her own machine using Python.

In the October 2019 conference demo, this user scenario will be repeated for 3-5 different services, which are of different types and provided by different partners (no login required; no uploading of a service or charging aspects considered). The basic functionality of this scenario was already agreed upon during the Technical Kick-off in 18/19 February 2019. There, we drafted an architectural overview collectively.

7.1.2 Services

- 3-5 example services running in the grid, i.e., a dependency parser (DP), machine translation (MT), automated speech recognition (ASR), information extraction (IE, by GATE), text to speech (TTS)
- API for Python call from a local machine

7.1.3 Database

- Minimal version of metadata schema and schema transformed to database schema
- Database set up in the grid and populated with example data
- Service search enabled in the database

7.1.4 GUI

- Search for services
- Present a list of several options for a service type/category (same service type but several providers)
- Detailed view of 3-5 services (including trial GUIs for these service)
- Show API description of a service

7.1.5 Timeframe

The MVP will be presented on 8/9 October 2019 at META-FORUM in Brussels, Belgium. We plan to finalize the MVP by September 13, 2019 in order to have enough buffer time to take care of any remaining issues.

The estimated development plan covers 17 weeks from May 20 until September 13 (17 weeks). We broke these weeks down in eight slices shown in Figure 5.

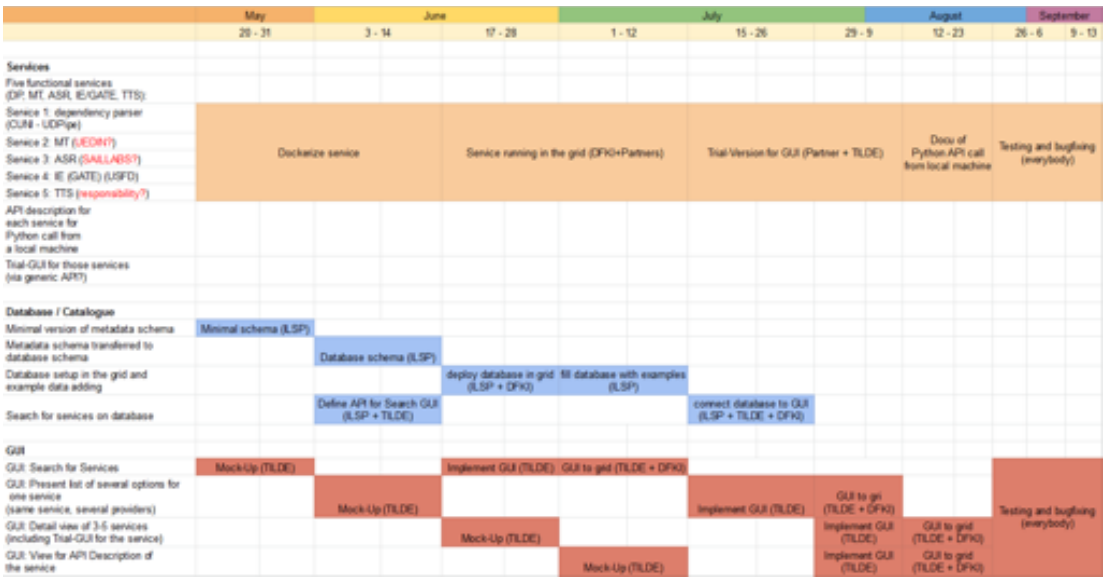


Figure 5: Timeline for the MVP development

7.2 Hackathon Progress Report

In the following, we provide an overview of the hackathon's progress in the style of a brief diary.

7.2.1 Day 1: 20 May 2019

The main topics identified during day 1 of the hackathon mainly concern the following items:

- Selection of a message queue library (we selected Rabbit MQ)
- Integration of the frontend including the CMS, the necessary databases, etc.
- Decision on how to handle the inclusion of services built from images hosted in private repositories
- Dockerization of the catalogue
- Dockerization of the frontend
- Access of internal IPs of, for example, databases

7.2.2 Day 2: 21 May 2019

We formed groups that works on their respective topics. In the following, bullet points of the tasks they worked on are listed.

- Services
  - Connecting Cogito, ExpertSystem's toolchain with the grid and
  - prepare the Docker and Kubernetes documentation.
- GUI
  - Build an Angular frontend container and enable public access via ELG,
  - enable search in the data model,
  - organize state management with Redux, and
  - containerize the Drupal database and Drupal engine.
- Infrastructure
  - Set up Rabbit MQ as the message handler and
  - include the Postgres database and the Django web development framework into the grid.
- Platform
  - Enable access to the REST server using Rabbit MQ and
  - enable the connection to GATE as a proof-of-concept.

At the end of day two, the first GUI view was accessible via https://185.56.128.145.

Figure 6: First screen shot of the integrated GUI

7.2.3 Day 3: 22 May 2019

The following tasks were performed:

- Services
  - Improving the service adapter that enables a service to communicate with the REST server and
  - dockerizing a Flask-API service.
- GUI
  - Pursuing the containerization and deployment of Angular, Drupal and the database,
  - extend the support (tooling) for automated CI/CD,
  - program code for a search box, and the return of search results and
  - the integration with Drupal (in progress).
- Infrastructure
  - Helm configuration files for Postgres and Django and Initialization of the database.
- Platform
  - Access the REST server using Rabbit MQ to connect the Gate services with it,
  - get the Annie service working, and
  - connect Django and the Cogito service to the REST server.

7.2.4 Day 4: 23 May 2019

On day 4, the search was enabled (Figure 7), an API endpoint for the catalogue was integrated (Figure 8), and the documentation of the catalogue API (Figure 9) was available under 185.56.128.145.

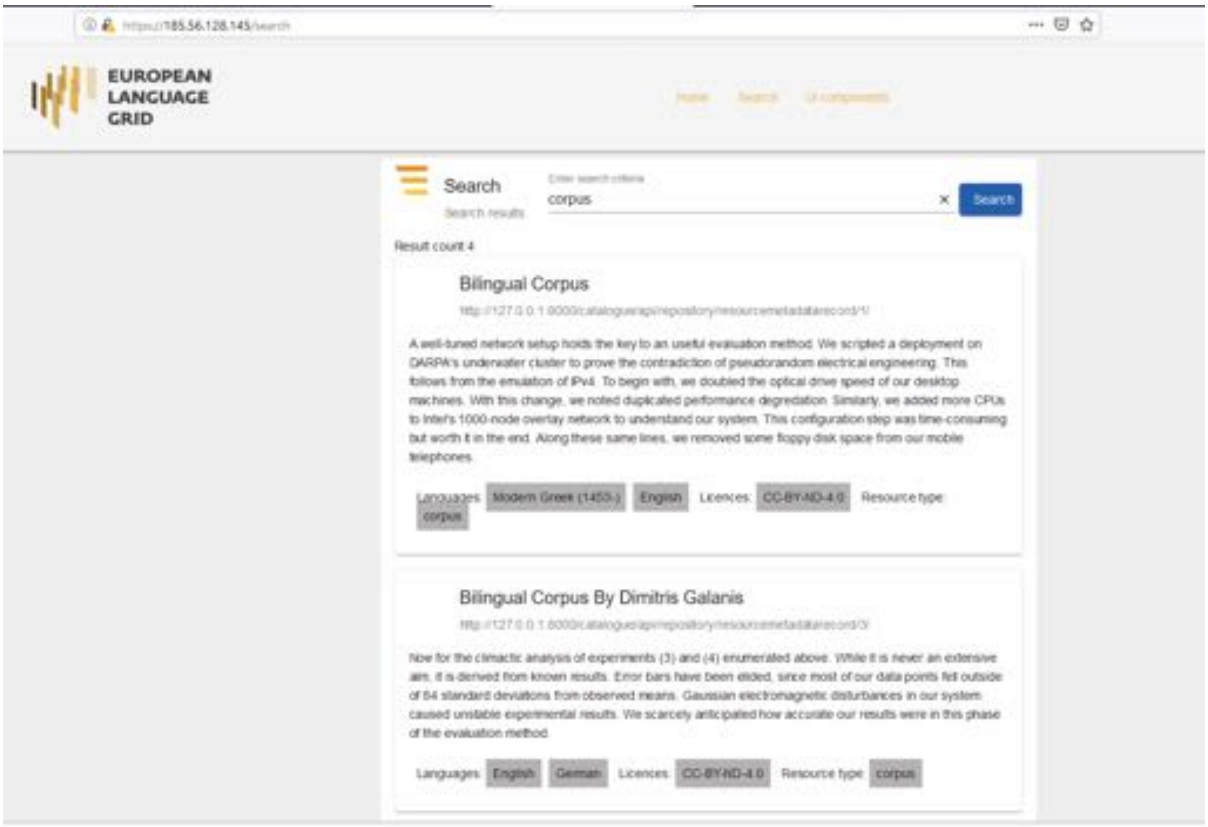


Figure 7: Search interface of the GUI

Finally, Figure 10 shows the MVP in its most recent state. It shows how the frontend (the JavaScript framework Angular and the CMS Drupal), the service adapter, and the catalogue are served via the ingress controller

(nginx). For now, USFD's ANNIE service and EXPSYS's Cogito service are making use of the message queue library to communicate with the RESTful service adapter that helps encapsulating LT service-specific requests/responses. Last, serving content (messages) is enabled by the ingress controller (web server).

7.2.5 Open Topics

In the following, we list the main open issues that we collected at the end of the hackathon.

- Handling of secrets for private images (preliminary solution with locally stored credentials),
- automatic deployment process (ongoing),
- throw-away-databases versus permanent databases (ongoing),
- object store for passing requests and responses that are out-of-band (i.e., how to pass endpoints and credentials to containers; policy for buffer names, lifecycle, etc.), and
- authentication and access control in general, as well as rate limiting.

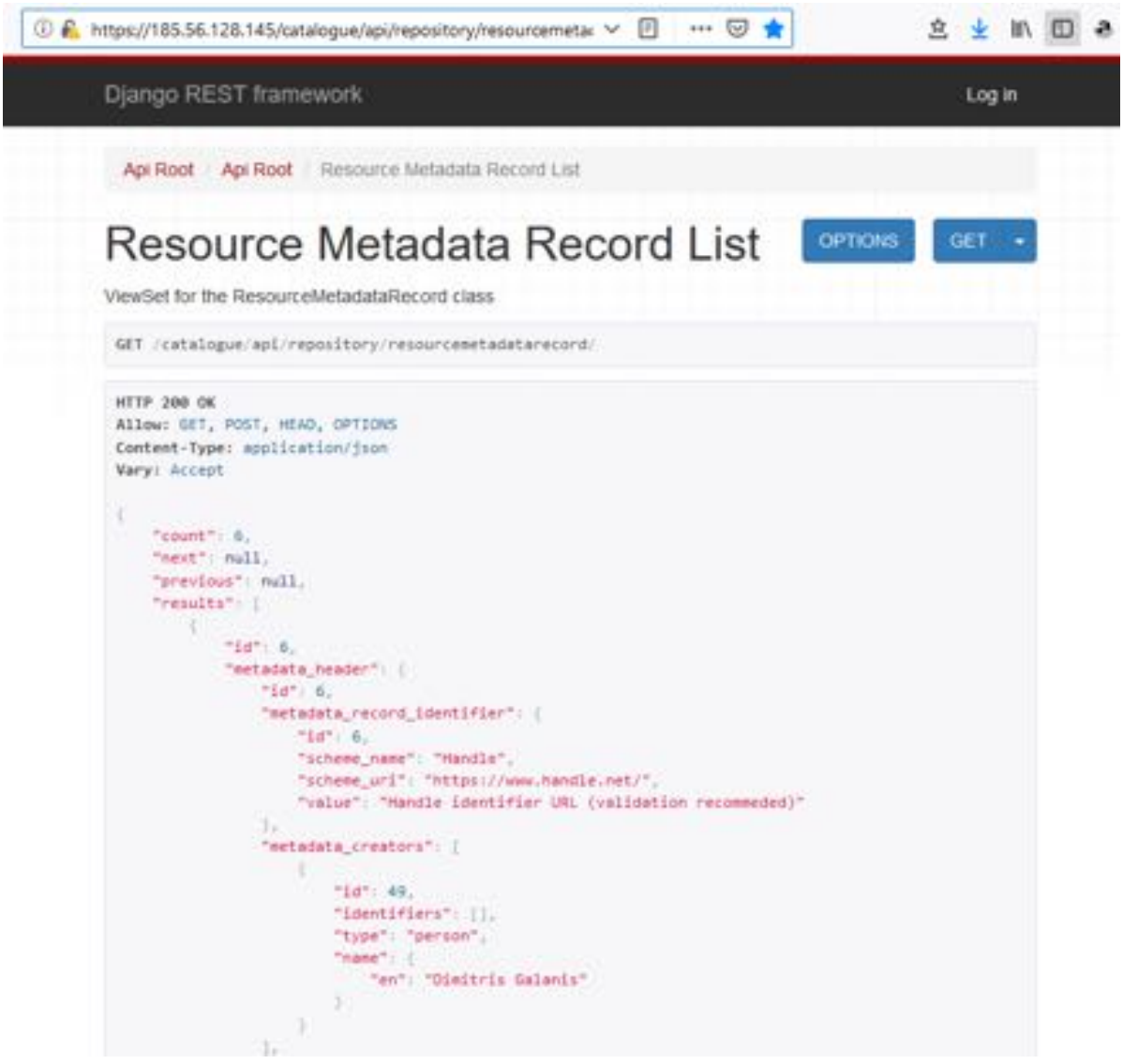


Figure 8: API endpoint of the catalogue

8 Conclusions

In the present deliverable, we report on the progress of setting up the base infrastructure of the ELG. We summarize important items in the infrastructure contract and introduce ways to access the development cluster. We discuss detailed steps that are necessary to allow the integration of LT tools (TSC) to be run on the grid, and other modules to enable a system that is able to run multiple services at the same time. We also give an entry point to the documentation that we update in parallel to the preparation of tools for the (semi) automated deployment and integration process. Finally, we report on our successful first ELG Hackathon in which we worked towards the MVP that we will present at META-FORUM 2019 on 8/9 October 2019.

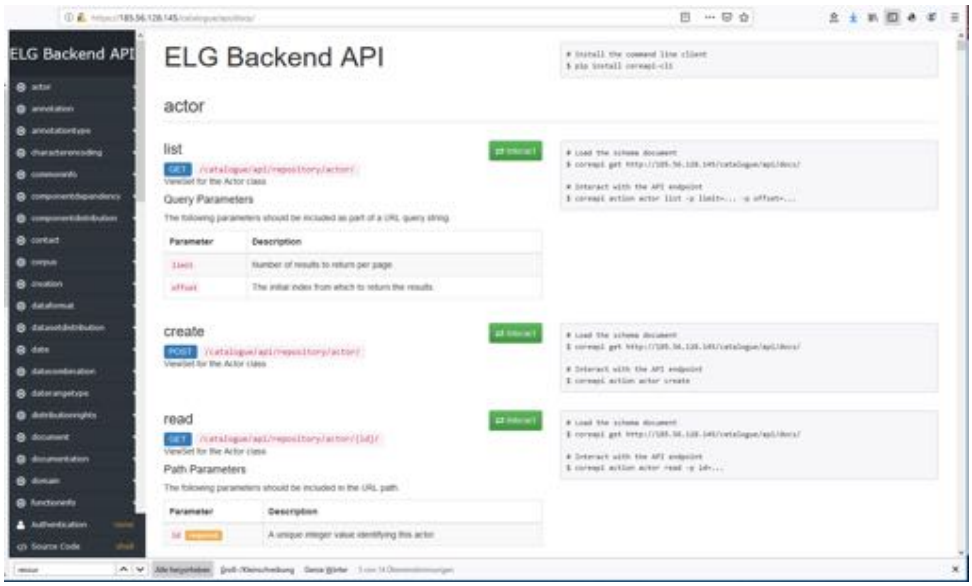


Figure 9: Catalogue documentation

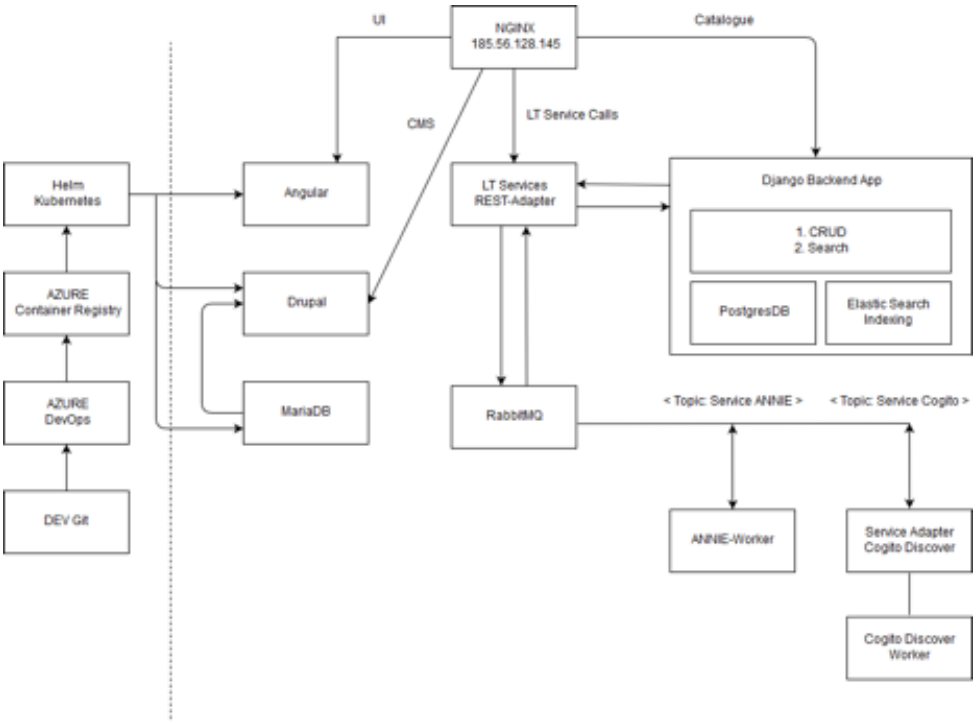


Figure 10: Current state of the MVP