D2.8

EUDAT service compatibility

Document information

Title	EUDAT service compatibility
ID	CLARINPLUS-D2.8 (CE-2017-1034)
Author(s)	Claus Zinn
Responsible WP leader	Erhard Hinrichs
Contractual Delivery Date	2017-06-01
Actual Delivery Date	
Distribution	Public
Document status in workplan	Deliverable

Project information

Project name	CLARIN-PLUS
Project number	676529
Call	H2020-INFRADEV-1-2015-1
Duration	2015-09-01 – 2017-08-31
Website	www.clarin.eu
Contact address	contact-clarinplus@clarin.eu

Table of contents

1 Executive Summary

The WebLicht software suite is one of the most robust and well-known workflow engines in the CLARIN infrastructure. It allows users to execute predefined workflows for common processing tasks and to define new workflows with the help of the WebLicht orchestrator. The services of a given workflow, however, are distributed geographically, and hence, there is a significant amount of data (input data and result data originating from intermediate processing stages) that is transferred between WebLicht and the individual services. The distributed nature of the WebLicht infrastructure makes it hard if not impossible to use WebLicht on big data sets, or on data sets with restrictive property rights attached to them. Here, data transfer time becomes very costly, or even prohibitive as sensitive data may not be allowed to leave the data's home institution.

The EUDAT project is currently developing the Generic Execution Framework (GEF). The GEF aims at providing an infrastructure that permits the execution of generic workflows in a computing environment located close to the data (at the data's host institution). Such an execution environment would tackle the aspects of transferring large amounts of data, or data with restrictive property and privacy rights.

In this deliverable, we report on the progress to bring together the CLARIN-based WebLicht workflow engine with the EUDAT-based Generic Execution Framework. Given the current and evolving state of the GEF, our work has to be seen as a starting point. While we attempt to give various possible integration scenarios to a good level of detail, we need to emphasize that subsequent implementations may need to deviate from our technical specification.

The potential usage of WebLicht within a GEF environment has already had a positive impact on WebLicht's design rationale, in particular, with respect to organising the transfer of data between the services of any given workflow. Our input has also benefited the GEF development team, yielding concrete GEF usage scenarios and defining specific user requirements.

2 Introduction

This deliverable reports on progress on task 2.3.3 of work package 2 of the CLARIN-PLUS project. In brief, the objectives for this subtask are to address two challenges that affect the applicability of workflows for some data sets. First, restrictive property rights may forbid research data to leave their home institution, and therefore, data transferal to other institution for processing is not allowed. The second issue concerns the size of the data, with big data often causing prohibitive overhead once it is necessary to send such data back and forth to the various tools of a scientific tool pipeline. In both cases, it is desirable to bring the workflow engine to the data, rather than having the data travel to the tools.

The EUDAT project is currently developing the Generic Execution Framework (GEF). The GEF aims at providing a framework that allows the execution of scientific workflows in a computing environment close to the data. In this deliverable, we explore a number of scenarios and technical adaptations to WebLicht to render its services compatible with the GEF.

The remainder of the deliverable is structured as follows. In Section 3, we give technical background on the Generic Execution Framework and WebLicht. In Section 4, we specify a number of scenarios that describe the adaptation of WebLicht towards a GEF integration. This includes user-centric front-end considerations as well as necessary adaptations on the WebLicht back-end and the processing services connected to WebLicht, which need to be converted to GEF-compatible web services. An integral part is a translation mechanism that liaises the WebLicht workflow engine with the GEF environment, and which ensures that all data transfers occur at the GEF's host institution, which also hosts all data. In Section 5, we review the service compatibility of WebLicht from the point of view of a GEF administrator. In Section 6, we conclude.

3 Background

3.1 Generic Execution Framework (GEF)

One underlying motivation for the GEF is that datasets have become much larger than the tools that process them. It is thus more efficient to move the tools to the data rather than the data to the tools, given that the tools do not require a large amount of processing power. This argument is often supported by restrictive property or privacy rights attached to the data; here data transfer to tools not under control by the property holder is often not possible. Also, the transfer of big data sets across the tools of a tool chain often leads to bottlenecks, and transfer time is sometimes higher than the actual processing time.

The GEF aims at a framework that allows developers to pack tools (and their computation) into movable containers that can be moved towards the data. GEF makes use of Docker, a virtualization mechanism that wraps an application into a container that capsules and sandboxes all computation. Following the explanations at https://www.docker.com/what-docker, a container image is a "lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings." Docker-based technology shows a very good performance, with container images that can have a small image size, and are fast to start. Container images are immutable by design and relatively easy to design, using Docker scripts and pre-built container images to build upon.

Figure 1 shows the design rationale of GEF. Docker software runs on the operating system of the host, on the host's hardware, and is able to execute Docker containers (holding the tools' virtualization).

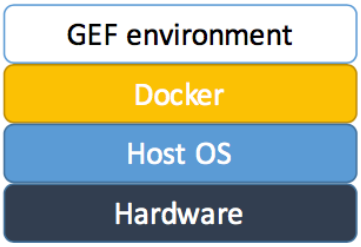

Figure 1. The GEF architecture.

GEF is built-upon Docker allowing community admin users to upload their tools as Docker files (from which Docker container images are being generated), or directly as Docker container images. Once uploaded, the Docker container image becomes a GEF service and can be invoked on any EUDAT data set.

The GEF source code is available at GitHub, see https://github.com/EUDAT-GEF/GEF. Its back-end is built upon the Go programming language (https://golang.org), and its front-end is built upon the React Javascript library for building user interfaces (https://facebook.github.io/react/).

There is no official software release of GEF available yet. Also, at the time of writing, documentation is rather sparse. The EUDAT deliverable from 2015 gives away some of GEF's design rationale (Dima, Pagé, and Budich, 2015). For general information about EUDAT's Collaborative Data Infrastructure (CDI), see https://www.eudat.eu/eudat-cdi/about.

It is easy to install a GEF environment from the GitHub repository. Figure 2 shows the front-end of the GEF. In the left pane, there are three items: "Build a service", "Browse Services" and "Browse Jobs". When choosing the first item, the GEF admin ("power user") can build a new GEF service, see (1). For this, the user needs to upload a Dockerfile, together with other files which should be part of the container. With the file upload complete, GEF's underlying Docker software will build a Docker-based container image, which the GEF will use to provide the corresponding GEF service of the application.

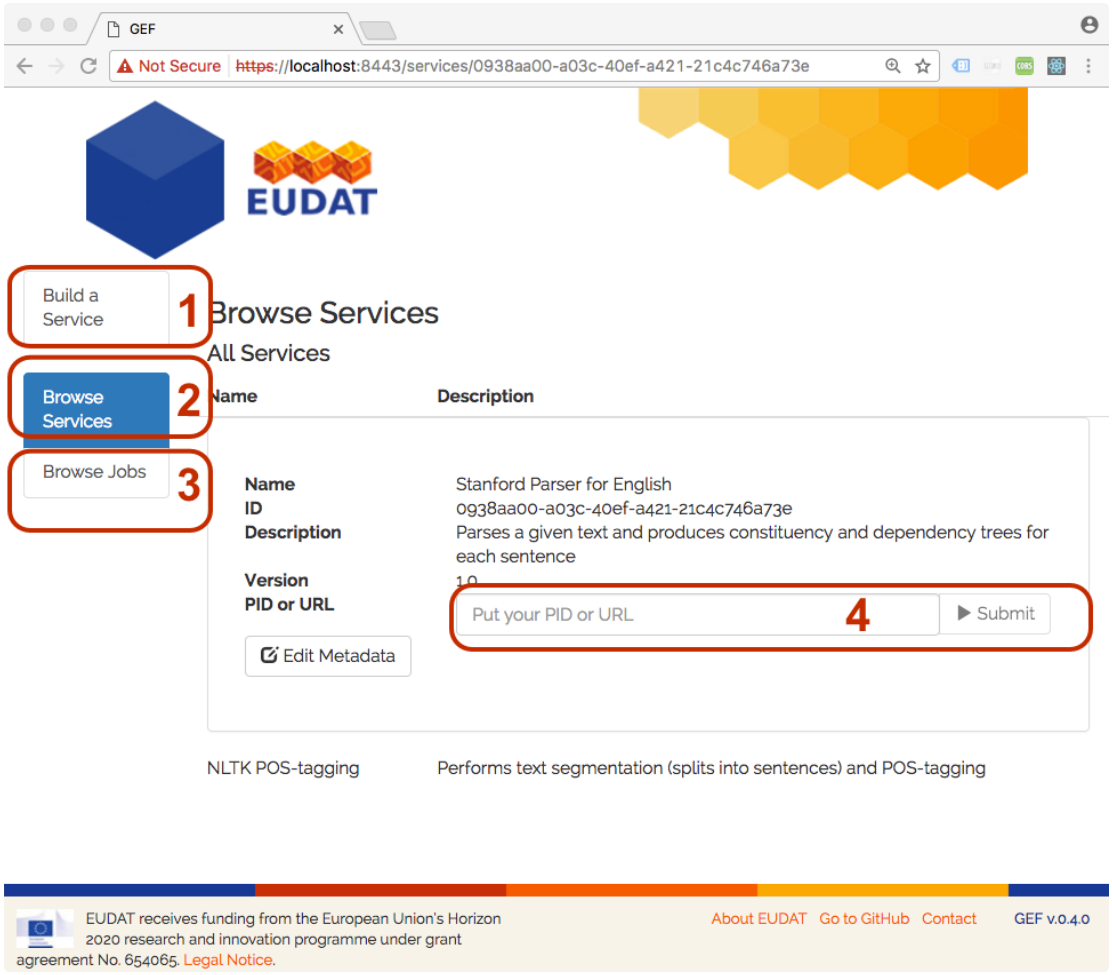

Figure 2: Main page of the GEF front-end.

The standard distribution of the GEF comes with a number of predefined Dockerfiles that can be used to test and play with the GEF. Figure 3 shows the Dockerfile for the Stanford Parser for English. It builds a Docker container based on the operation system Ubuntu, version 16.04. It updates the system's list of packages using Ubuntu's "apt-get", package manager, installs the software packages for 'curl' and 'unzip' and then uses the software to fetch the Stanford parser from a given location (the parser is precompiled and can be started with 'lex-parser.sh'). Note the number of LABEL instructions in the Dockerfile that fill in some of the metadata for the GEF-enabled parser service. In particular, it specifies the input (/root/input) and output (/root/output) service locations on the file system.

```
FROM ubuntu:16.04

MAINTAINER Alexandr Chernov <kstchernov@gmail.com>

LABEL "eudat.gef.service.name"="Stanford Parser for English"
LABEL "eudat.gef.service.description"="Parses a given text and produces constituency and dependency trees for each sentence"
LABEL "eudat.gef.service.version"="1.0"
LABEL "eudat.gef.service.input.1.name"="Input Directory"
LABEL "eudat.gef.service.input.1.path"="/root/input"
LABEL "eudat.gef.service.output.1.name"="Output Directory"
LABEL "eudat.gef.service.output.1.path"="/root/output"

RUN apt-get update
RUN apt-get install -y default-jdk curl unzip

RUN curl -Ls http://nlp.stanford.edu/software/stanford-parser-full-2016-10-31.zip > /root/stanford-parser-full-2016-10-31.zip
RUN unzip /root/stanford-parser-full-2016-10-31.zip -d /root
RUN rm /root/stanford-parser-full-2016-10-31.zip
RUN mkdir /root/input
RUN mkdir /root/output

CMD ["/root/stanford-parser-full-2016-10-31/lexparser.sh", "/root/input/*.txt"]
```

Figure 3: A Dockerfile for the Stanford Parser for English

Figure 2 shows that we have built two services from the GEF distribution; clicking on (2) shows all available services: the "Stanford Parser for English" and the "NLTK POS-tagging" software. A GEF-enabled service can be run by supplying a Persistent Identifier (PID) or URL that points to the input to be processed. Hitting the "Submit" button – see (4) – will start the service asynchronously. All running jobs can be inspected by (3). Once the job terminates, its output volume holds the result of the computation.

At the time of writing, GEF services can only deal with a single input parameter, the input volume, and a single output parameter, the output volume. Also, note that the current version of the GEF is also not capable of executing workflows. To execute a workflow, it must be orchestrated by an external engine, which makes reference to a GEF-based repository of (immutable) application containers, each of which can be referenced by a PID. In the rest of this section, we will describe how the WebLicht orchestrator can be used to execute a GEF-based WebLicht workflow; here each processing tool of the chain is hosted on a GEF environment. Clearly, we will also need to flesh out how any data transfer between the GEF-external WebLicht orchestrator and the GEF-internal services is handled.

3.2 WebLicht

WebLicht is a workflow engine giving users a web-based access to over fifty tools for analysing digital texts (Hinrichs, Hinrichs, and Zastrow 2010). Its pipelining engine offers predefined workflows and supports users in configuring their own. With WebLicht, users can analyse texts at different levels such as morphology analysis, tokenization, part-of-speech tagging, lemmatization, dependency parsing, constituent parsing, co-occurrence analysis and word frequency analysis, supporting mainly German, English, and Dutch. Note that WebLicht does not implement any of the tools itself but mediates their use via pre-defined as well as user-configurable process pipelines. These workflows schedule the succession of tools so that one tool is called after another to achieve a given task, say for instance, named entity recognition.

WebLicht is a good step forward in increasing (web-based) tool access and usability as its TCF format mediates between the various input and output formats the tools require, and calls the tools (hosted on many different servers located nation and world-wide) without much needed user engagement. WebLicht has now been used for many years in the linguistics community; WebLicht is actively maintained, profits from regular tool updates and new tool integrations, and has recently been integrated with TüNDRA, a treebank search and visualization tool that allows WebLicht users to inspect linguistically annotated data (Chernov, Hinrichs, and Hinrichs 2017). With TüNDRA,

users can search for specific linguistic phenomena at the word and sentence level, and visualize such phenomena.

Development of WebLicht started in October 2008 as part of the D-SPIN project (https://weblicht.sfs.uni-tuebingen.de/englisch/index.shtml); in the last eight years, it has accumulated a solid user base and is the work-flow engine of choice for many national and European researchers in the CLARIN context.

WebLicht as a Service. WebLicht as a Service (WaaS) is a REST service that executes WebLicht chains.[1] Unlike the WebLicht web application, WaaS does not require a browser, and hence prevents browser-specific issues to arise such as session timeouts and file size limits. With WaaS, users can run chains from their UNIX shell, scripts, or programs. Once users have defined a chain in the WebLicht browser interface, they can download the chain, and then they execute a HTTP POST request with the multipart/form-data encoding to invoke WaaS with the chain in question and the input data, e.g., by executing the following curl command:[2]

```
curl -X POST -F chains=@chains.xml -F content=@inputFile -F apikey=apiKey URL > result
```

WebLicht Harvesting of Tool Metadata. At regular intervals, WebLicht is harvesting tool metadata from tool providers (as ingested to the metadata repositories of the centre registries). The description of a WebLicht tool is based on the CMDI Profile *WebLichtServiceProfile*.[3] Among other information, it provides information about the tool's URL and its input and output parameters. Each WebLicht tool has a persistent identifier that refers to its CMDI description. The WebLicht orchestrator uses the tools' metadata to help users define workflows.

Workflow definition and execution. WebLicht offers users two modes: an "Easy Mode" and an "Advanced Mode". The first mode offers a dozen of predefined *easy-chains* that each define a tool workflow to perform a complex textual analysis. The second mode supports users in defining their own pipelines. Each tool in the workflow is identified by a persistent identifier that WebLicht's execution system uses to invoke the tool. With each tool invocation, WebLicht passes on a TCF-compliant data file that contains the tool's input. The tool processes selected parts of the input, and enriches the TCF file with the output of the computation, which is then sent back to the WebLicht orchestrator. WebLicht then passes the enriched TCF file to the next tool of the workflow until all tools in the workflow has been executed in the given order.

Figure 4 shows the data flow between the WebLicht orchestrator and the tools. Note that each tool is invoked via the Hypertext Transfer Protocol (HTTP) using POST methods, with the tools' output data being captured in the response body of the request. This design decision makes it hard to use WebLicht on huge data sets, or on data that cannot leave the host institution for property rights or privacy concerns.

[1] https://weblicht.sfs.uni-tuebingen.de/WaaS/

[2] Note that URL points to https://weblicht.sfs.uni-tuebingen.de/WaaS/api/1.0/chain/process and that an API key to access the service needs to be obtained as well.

[3] The CMDI XSD definition is available in the CLARIN component registry, available at https://catalog.clarin.eu/ds/ComponentRegistry/rest/registry/1.x/profiles/clarin.eu:cr1:p_1320657629644/xsd .

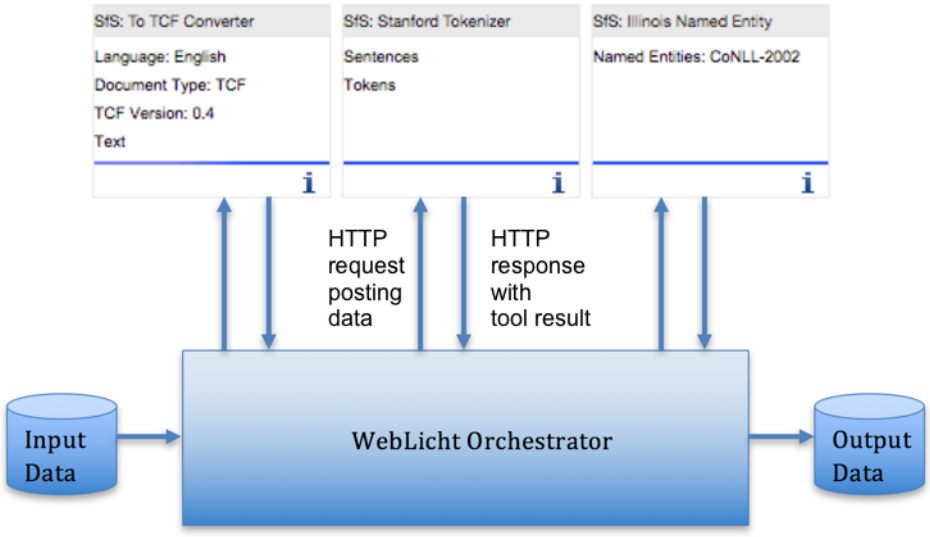

Figure 4. Data Flow in WebLicht.

In the GEF mind set, WebLicht's tools need to be "migrated" to a GEF environment that encapsulates all computation. The GEF environment is located close to the data, that is, placed on the same computer network that also hosts the data. WebLicht then calls GEF-ified services rather than the original tools. To keep all data transfer local to the GEF environment that also hosts the data, the way the WebLicht orchestrator invokes the services needs to be changed as well.

In the next section, we outline a number of possible scenarios to make WebLicht service-compatible with the GEF framework.[4]

[4] WebLicht is not a web service but a web-based/browser-based tool that provides users with a user interface to specify the input data, select a predefined easy-chain, or in advanced mode, define a new workflow. It also visualises the results of the pipeline, including information about the individual stages. At the current development stage of GEF, WebLicht cannot be "GEF-ified", but "WebLicht As a Service" could be GEF-ified.

4 WebLicht – GEF Integration Scenarios

A user with big data or with data having restrictive property rights would like to inform WebLicht that it should only consider tools as part of a processing workflow that run at his or her home institution. This assumes, however, that the given home institution has set-up a GEF environment that hosts all the "relevant" web services that the WebLicht users want to employ.

4.1 GEF Integration – Front-end Adaptation

There are two possible approaches where users could specify GEF-related information, only the first requiring a change to WebLicht's user interface.

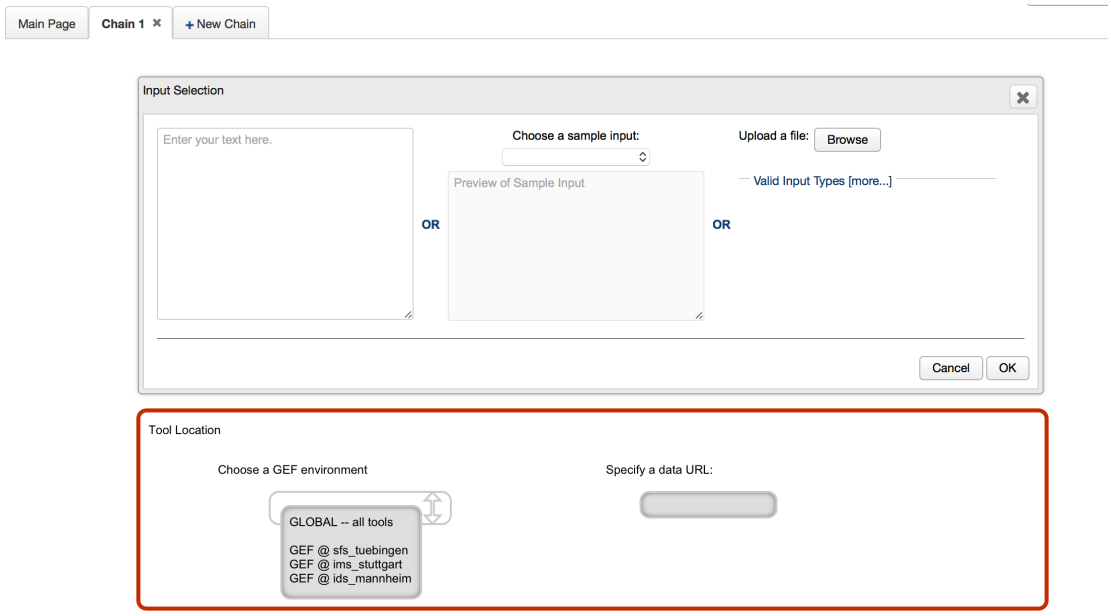

Figure 5. Sketch of WebLicht GEF extension (proposal)

Figure 5 depicts a proposed UI change to WebLicht's entry page. In addition to the "Input Selection" pane, there is a "Tool Location" pane, where users can select the tool environment that the WebLicht orchestrator should use. The environment "GLOBAL – all tools" is the default tool environment where WebLicht assumes traditional operation (all tools are eligible for inclusion in the workflow). When the user selects the environment "GEF@sfs_tuebingen", the WebLicht orchestrator will only make use of workflows whose tools are running in the GEF environment at the Seminar für Sprachwissenschaft (SfS) in Tübingen.[5]

Note, however, that a WebLicht–GEF integration must not materialise on the front-end. In "Easy-Mode", WebLicht might offer predefined easy-chains that run entirely in a given GEF environment; those easy-chains are merely named appropriately, say "Named

[5] Sometimes, a user might be happy to have partial GEF support, that is, a user *preference* for tools in a given GEF environment. If WebLicht can identify a service available in the user's preferred GEF environment, then WebLicht would use it, otherwise WebLicht would use a service at a different location.

Entities (GEF@sfs_tuebingen)". Users will then select the GEF environment by selecting the appropriately named easy-chain.

In WebLicht, it is already the case that applicable tools are often prefixed by their location, e.g., "SfS: To TCF Converter", "SfS: Stanford Tokenizer", or "SfS: Illinois Named Entity". Here, GEF-ified variants of the tools will need to be named accordingly, say, "GEF_sfs_tuebingen: To TCF Converter". In "Advanced Mode", users can construct a workflow that suits their computing requirements, say, by selecting only tools that are part of a given GEF environment as indicated by the tool's name. The usability of this approach, however, decreases with the number of tools running in many different GEF environments, for instance, when a user has to choose among a dozen different instances of "To TCF converter", all running in different GEF environments. Here, the aforementioned front-end adaptation for WebLicht seems more suitable.[6]

4.2 GEF Integration: Back-end Implications

The backend adaptations to WebLicht are more substantial. Two changes to WebLicht's back-end are discussed: how should the orchestrator organise its tool space; and how should the orchestrator call the individual tools to minimize data transfers?

4.2.1 Making the Cut through the Tool Space.

The WebLicht Orchestrator works on the entire tool space that results from regularly harvesting tools' metadata from tool providers. Here, we assume that providers of GEF-ified tools advertise them in the same way than their non-GEF-ified equivalents: Their tools' metadata is entered in the metadata repository of the respective CLARIN centre repository so that WebLicht can harvest these tools as well.

For the following, we assume that the WebLicht user has specified a GEF environment of her choice (Figure 5). Now, the user can choose between two modes, see Figure 6.

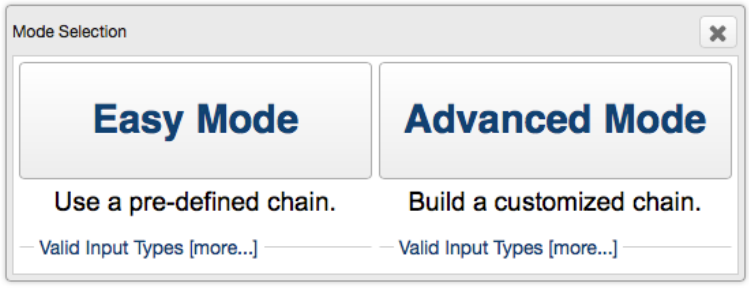

Figure 6. WebLicht: Mode Choice.

In Easy Mode, the WebLicht orchestrator needs to identify which predefined workflows are available in the GEF organisation specified by the user. Here, a workflow is available in a GEF environment if each tool of the workflow is available in the GEF environment. In Advanced Mode, the Weblicht orchestrator makes a cut through the tool space gathered from the harvester. All tools not originating in the given GEF environment can

[6] The WebLicht orchestrator could, however, take the partially constructed workflow into account; it can identify and present services first that are in the same GEF environment as those services already part of the workflow.

be ignored; only tools that are part of a given GEF environment are considered for workflow construction.

4.2.2 Transporting data through the workflow

Figure 4 displays the data flow in the current version of WebLicht. It shows the back-and-forth movement of data between the WebLicht orchestrator and the tools it is executing. If a given workflow consists of *n* tools, then the data is transferred 2*n* times between the orchestrator and the tools, potentially across networks of varying bandwidth and latency.

With data required to stay in the GEF environment, there are two options, given that all tools of a workflow live in the same GEF environment:

1. The WebLicht orchestrator becomes a GEF-ified variant of WaaS, that is, WebLicht as a Service becomes also a service running in the GEF environment.
2. The interplay between the orchestrator and the tools change; rather than passing along actual data, now references to data is being passed, with all references originating in the GEF environment.

Ad1. The first option does not affect the amount of data transfer between the players, but makes the data transfer less of a bottleneck given that all tools of a given GEF environment have access to the same high bandwidth, low latency network. On the other hand, the first option is less flexible than the second as WebLicht chains must be known and defined in advance (so that the WaaS can be called with the chain in question), but this might be acceptable for most GEF use cases.

Ad2. The second option requires either changing the tools connected to WebLicht so that they now accept *references to data* rather than the data, or building wrappers around the tool that show this behaviour. Here, a wrapper approach seems to be a better choice and it does not require interaction with and contribution from tool developers.

Step 1. Figure 7 shows the wrapping of a tool (left) resulting in a wrapped tool (right). The wrapped tool: (i) retrieves the input data from a given URL, (ii) starts the tool it wraps by posting the tool the input data via HTTP, and (iii) accepts the response of the HTTP request, (iv) stores the output data locally (no transfer), and (v) returns this location with a corresponding URL. With the workflow consisting only of tools with such wrappers no data is sent back and forth to WebLicht, but rather URL-based pointers to the data.

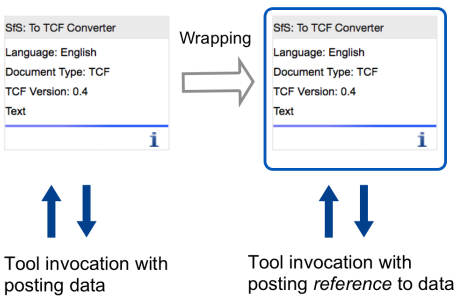

Figure 7: Wrapping a Tool.

Note that the wrappers will be required to monitor the execution of the tools they wrap. That is, when an HTTP POST request yields an error message, then this message is sent

back to the WebLicht orchestrator, which then prematurely terminates the execution of the workflow with an appropriate error message.

Step 2. The wrapped tools are dockerized.

We have dockerized a good number of tools connected to WebLicht. Each web service shown in Figure 8, for instance, has already a Docker variant. This means that there is a Dockerfile that can be used to generate the Docker container image, and hence, the web service can already be incorporated in a GEF environment, taking the approach illustrated with the help of Figure 2 and Figure 3.

Given that we have Dockerfiles for a good number of WebLicht web services, it should be investigated whether a Dockerfile for a web service can be automatically extended to a Dockerfile for a wrapped web service, see Figure 7.

Note: when a tool is integrated into WebLicht, it receives a "web service wrapping". For the GEF use case it might be profitable to unwrap its WebLicht wrapping, that is, to take the original tool as a starting place for the GEF-based wrapping. With tools being no longer web services, no http requests take place, just plain tool invocations that need to get their data somehow (depending on the tool). On the other hand, starting from the web services wrapping is easier and it serves as an abstraction shared by all tools. Wrapping the tool rather than the web service requires looking at the specifics of each tool individually.

4.3 Liaising between WebLicht and the GEF

A *translation mechanism* (working title "BridgIt") is needed to liaise between WebLicht and the GEF-enabled services in a GEF environment. We discuss the liaison by example. Consider a user wanting to annotate sensitive data, say an English text with named entities; for this, the user selects a GEF-based WebLicht workflow, cf. Figure 8.

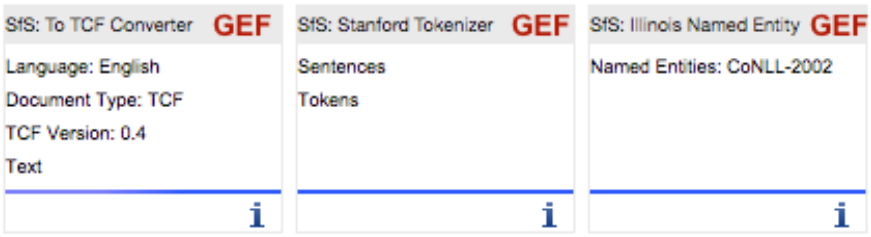

Figure 8. A WebLicht Workflow for Named Entity Recognition (English texts)

This workflow has a corresponding XML representation, see Figure 9. Here, we have used hypothetical handles.

```
<cmd:CLARIN-D xmlns:cmd="http://www.clarin.eu/cmd/1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc
    <cmd:chains>
        <cmd:CMD CMDVersion="1.2">
            <cmd:Resources>
                <cmd:ResourceProxyList/>
                <cmd:JournalFileProxyList/>
                <cmd:ResourceRelationList/>
            </cmd:Resources>
            <cmd:Components>
                <cmd:WebServiceToolChain>
                    <cmd:GeneralInfo>
                        <cmd:Descriptions>
                            <cmd:Description/>
                        </cmd:Descriptions>
                        <cmd:ResourceName>myChain</cmd:ResourceName>
                        <cmd:ResourceClass>Toolchain</cmd:ResourceClass>
                    </cmd:GeneralInfo>
                    <cmd:Toolchain>
                        <cmd:ToolInChain>
                            <cmd:PID>http://hdl.handle.net/11858/00-1778-GEF-0004-BA56-7</cmd:PID>
                            <cmd:Parameter value="en" name="lang"/>
                            <cmd:Parameter value="text/plain" name="type"/>
                        </cmd:ToolInChain>
                        <cmd:ToolInChain>
                            <cmd:PID>http://hdl.handle.net/11022/0000-GEF-2518-C</cmd:PID>
                        </cmd:ToolInChain>
                        <cmd:ToolInChain>
                            <cmd:PID>http://hdl.handle.net/11022/0000-GEF-839F-9</cmd:PID>
                        </cmd:ToolInChain>
                    </cmd:Toolchain>
                </cmd:WebServiceToolChain>
            </cmd:Components>
        </cmd:CMD>
    </cmd:chains>
</cmd:CLARIN-D>
```

Figure 9. The Named Entity Recognition EasyChain in XML.

Each of the three tools in the tool chain is referenced with a persistent URL. The persistent tool URL points to metadata that describes the tool. Consider the second tool of the chain referenced with http://hdl.handle.net/11022/0000-GEF-2518-C . This (hypothetical) handle refers to a CMDI-based [7] instance of the CMDI profile *WebLichtWebService*. The CMDI instance is located in a CLARIN centre repository.[8] The information in the CMDI instance holds the information given in Figure 10.

Stanford Tokenizer Stanford Tokenizer is a an efficient, fast, deterministic tokenizer.

> Stanford Tokenizer is a an efficient, fast, deterministic tokenizer.

PID	http://hdl.handle.net/11022/0000-0000-2518-C		
URL	http://bridgit.sfs.uni-tuebingen.de/StanfordTokenizer		
Email	wlsupport@sfs.uni-tuebingen.de	**Status**	production
Created	2014-07-07T11:45:58.789+02:00	**Modified**	2014-07-07T17:11:03.775+02:00
Input		**Output**	
type	text/tcf+ **url**	sentences	
version	0.4	tokens	
text			
lang	en		

Figure 10. Metadata for the Stanford Tokenizer.

[7] CMDI stands for Component MetaData Infrastructure, see www.clarin.eu/cmdi for details.
[8] See http://weblicht.sfs.uni-tuebingen.de/apps/harvester-beta/resources/services.

The metadata has a URL slot that is used to invoke the tool from WebLicht. Note that the URL does not point to the GEF environment, but rather to the translation mechanism, see below. Also note that the type of the input is not "text/tcf+xml" but "text/tcf+url", indicating that WebLicht will need to invoke the tool by passing a reference to the data to the tool, rather than the actual data.

For the following discussion, please consult Figure 11, which summarises the interaction between the WebLicht orchestrator and the GEF environment.

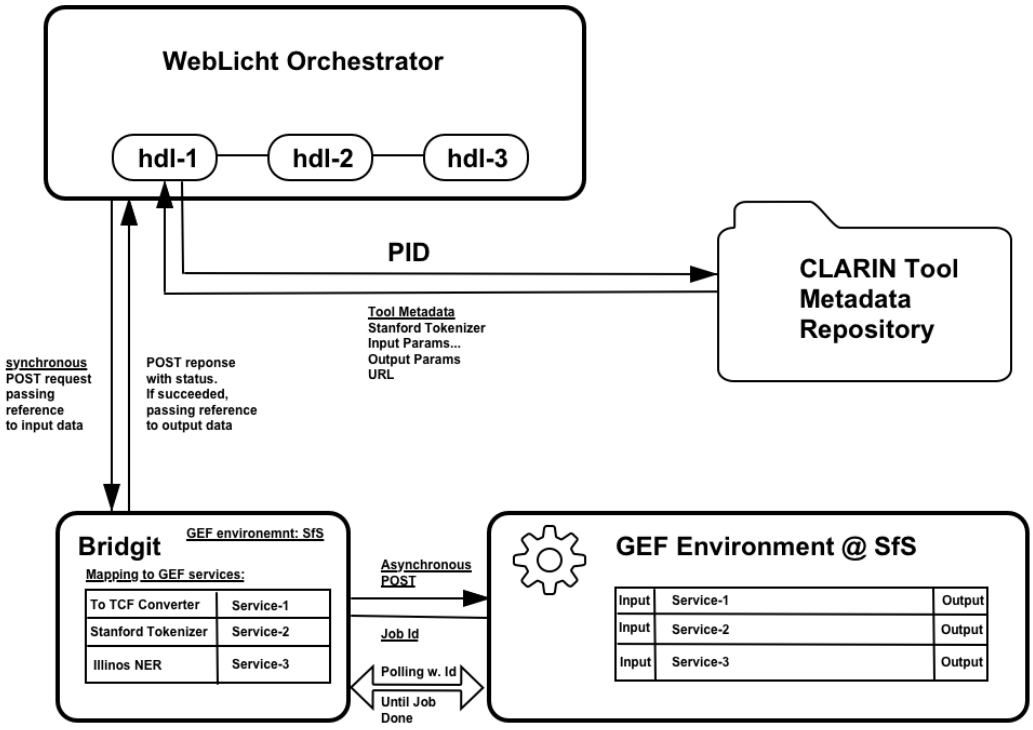

Figure 11: Architecture of WebLicht - GEF interaction.

As with all other web services connected to WebLicht, the orchestrator invokes the given URL with a synchronous POST request. Given the new type "text/tcf+url", no input data is being passed, only a reference to the data (a URL).

The synchronous request invokes BridgIt, the translation mechanism, which liaises with the GEF environment. BridgIt maintains the following information:

- A pointer to the GEF environment, say, http://gef.sfs.uni-tuebingen.de:4041.
- A JSON-based table that maps URL-encoded information such as "/StanfordTokenizer" to a service code that identifies the corresponding service in the GEF environment.

With this information a URL of the following form is constructed:

http://gef.sfs.uni-tuebingen.de:4041/jobs?serviceId=<id>&pid=<pointerToData>

BridgIt sends this URL as POST request to start the GEF service The given GEF environment answers with a JSON structure (as body of the response to the POST request) that gives the job identifier of the corresponding GEF service, e.g.,

```
{ jobId : 42 }.
```

Bridgit then polls the GEF environment at regular intervals to check whether the job has terminated. Technically, this is a GET request of the form:

http://gef.sfs.uni-tuebingen.de:4041/jobs/<jobId>

This GET request returns all information about the GEF job with the given jobID:

```
{ job:
    { ID: 42,
      State: { Status: "…."
               Error: …
               Code: -1
             }
      OutputVolume: "…"
}
```

The information includes the job's id, state, and output volume (for the result of the processing). If the job's state signals that the job is still running, BridgIt sleeps for a second, before sending another GET request, and this continues until the job terminated. When the job terminated successfully, then BridgIt returns a reference to the output volume associated with the job id; otherwise BridgIt passes on the error code encoded in the response of the GET request. Any result is returned to WebLicht as response to the synchronous POST request to BridgIt.

Notes:

- Each GEF environment has its own associated BridgIt process.
- GEF needs to support multiple parameters (other than a pointer to the input data). For instance, for the invocation of BridgIt via

  http://bridgit.sfs.uni-tuebingen.de/StanfordTokenizer?model=<someModel>

  BridgIt will need to map the parameter "model" and its value to the corresponding parameters of the GEF-enabled service. Here, BridgIt must be aware of the metadata of the GEF-enabled service to perform the mapping.
- For various reasons, BridgIt and its associated GEF environment might be out of sync. Wrong GEF invocations (e.g., erroneous parameter associations) will yield a server error (code 500); here the POST request to the GEF environment will also return a body with the error specification, e.g., "argument mismatch".

5 GEF Perspective

It is quite likely that any GEF environment will only include a small set of WebLicht services. However, setting-up a GEF environment comes at a substantial cost. Technical and logistical support is needed to keep costs manageable. From the perspective of a GEF community manager, it is highly desirable that all tools he or she wants to integrate into a GEF environment are pre-packaged. The dockerization of a tool requires intricate knowledge about the tool and its best runtime environment. In general, such information is rarely available.

As a consequence, we assume a GEF community manager to contact the WebLicht developers and to ask them for support. There is a good chance that a WebLicht web service of interest has already been dockerized by the WebLicht development team. Ideally, the wrapping of web services to GEF-ifiable tools, as discussed in Section 4, can be automated: a script extends the existing Dockerfile for a given web service with instructions that perform the wrapping. The extended Dockerfile can then be uploaded to the GEF environment as discussed in Section 3.1.

Setting-up a GEF environment must also include the provision of metadata records that describe all GEF-ified tools using the CMDI profile *WebLichtWebService*. The metadata must be included in the GEF host's CLARIN centre repository so that WebLicht can harvest this data and hence knows about the existing of the GEF environment. For this, the existing metadata of the original web service should be copied and adapted accordingly.

The GEF community manager must also complement the GEF environment with the translation mechanism we discussed in Section 4.3. Here, we assume that a reference implementation of BridgIt will be provided by the WebLicht team so that a GEF admin can use and configure the translation mechanism with little cost.

Given the required expertise and high cost of setting up a GEF environment for WebLicht services, the WebLicht team may provide GEF environments for popular natural language processing chains. Once a GEF environment has been set up, the cost of moving it to a new location (close to the data) should be relatively low.

6 Conclusion

In this deliverable, we have sketched a potential integration of the CLARIN WebLicht workflow engine and its services with a EUDAT's Generic Execution Framework. Our integration would allow users to bring the language processing tools integrated into WebLicht to an execution environment that also hosts the data, hence allowing language processing close to the data.

From the implementation perspective, initial steps towards such integration have been done. A good number of WebLicht web services have been dockerized. The containerisation of services is the first step towards "movable computation". All other steps still await implementation. The location of the application container, for instance, needs to be advertised, of course, so that workflow engines relying on the computation can address the container accordingly. For the "advertisement", the CMDI-based WebLichtWebService profile must be used. The metadata must be added to CLARIN centre repositories for tool metadata so that WebLicht can harvest it from there.

At the time of writing, there is a significant amount of data transfer between the WebLicht orchestrator, and the individual services that define a workflow. Such transfer becomes prohibitively expensive for big data, and non-permissive for sensitive data. Independent from GEF's existence, WebLicht may decide to invoke the services in future by giving them references to the data rather than by posting the data to them.

At the time of writing, GEF offers no support for the execution of workflows. The WebLicht orchestrator is thus needed to perform this task. As we have demonstrated, a translation mechanism ("BridgIt") will need to be implemented to liaise between WebLicht and the GEF-enabled services.

There are a few steps to be taken to integrate WebLicht with GEF. A proof of concept GEF environment for a popular language processing workflow (e.g., Named Entity Recognition for English) should be set-up. So far, we have dockerized all three tools of this workflow. We need to extend the respective Dockerfiles so that the tools receive their GEF packaging with regard to data transfer. It needs to be investigated whether the Dockerfile extension from WebLicht web service to GEF-enabled service can be mechanized.

For testing purposes, we will create a WebLicht branch that offers GEF-enabled services to users. At the same time, we will need to patch the production server of WebLicht to not show those services to the users. Also, we need to build a demonstrator for the translation mechanism BridgIt.

With this work, we have demonstrated, in theory, that CLARIN web services can be exposed as GEF services, and that those GEF-enabled services can be used within the CLARIN WebLicht workflow engine. More implementation work is required to show the feasibility of our approach.

We will continue to closely cooperate with the EUDAT-GEF development team to ensure that user requirements stemming from the WebLicht use case lead to GEF feature requests that will find their way into the official GEF specification and implementation.

References

[1] E. Hinrichs, M. Hinrichs, and T. Zastrow: WebLicht: Web-Based LRT Services for German, Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (System Demonstrations), 2010.

[2] E. Dima, C. Pagé, and R. Budich: D7.5.2: Technology adaptation and development framework (final). EUDAT deliverable, retrieved from https://b2share.eudat.eu/api/files/4cc8cf0e-99a2-4b6b-981a-0ffcd870af19/EUDAT-DEL-WP7-D7%205%202-Technology_adaptation_and_development_framework-2.pdf

[3] A. Chernov, E. Hinrichs, and M. Hinrichs (2017). "Search Your Own Treebank." In: *Proceedings of the 15th International Workshop on Treebanks and Linguistic Theories (TLT15), Bloomington, IN, USA, January 20-21, 2017.* Pp. 25–34.