KCONNECT

www.kconnect.eu

Component Interoperability Architecture

Deliverable number	*D2.1*
Dissemination level	*Public*
Delivery date	*31 October 2015*
Status	*Final*
Author(s)	*Konstantin Pentchev, Fredrik Axelsson, Dragomir Yankov*

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644753 (KConnect)

Executive Summary

This deliverable describes the KConnect architecture for cloud and local deployment of critical services. It also defines the interfaces for these services.

There are several key services identified in the KConnect project:

- information extraction service
- data provision service (knowledge base)
- machine translation service
- semantic search service

The cloud architecture is based on the S4 architecture designed in the DaPaaS project [9]. It depends on several AWS services and is therefore tightly coupled to the EC2 cloud. Key features of the cloud deployment include:

- Pay per usage pricing model
- User authentication
- User quota management
- Automatic scalability and recovery
- Automatic backups

In order to allow organizations that are restricted from publishing their data to external locations such as the AWS, we have also specified a design for local installations of the KConnect services. It is based on the Docker platform for building, shipping and running distributed applications. Each service is packaged as a separate, self-contained Docker Image. These images can be used to start one or more instances of the service. The instances themselves can be linked together to achieve the full workflow of the KConnect system. There are features of the cloud architecture that will not be provided out-of-the box for local installations and need to be implemented by the client or a service provider. These include:

- Auto scaling
- Recovery
- Data backups
- Quota management

The reason for not providing these features is 1) their dependency on AWS and 2) their implementation will differ with the specific infrastructure of each client.

Table of Contents

List of Abbreviations

AWS	Amazon Web Services
EBS	Elastic Bean Store
HTTP	Hypertext Transfer Protocol
IE	Information Extraction
JSON	JavaScript Object Notation
KB	Knowledge Base
LD	Linked Data
MT	Machine Translation
PaaS	Platform as a Service
REST	Representational State Transfer
SaaS	Software as a Service
SNS	Simple Notification Service
SQS	Simple Queue Service
UI	User Interface
VM	Virtual Machine

1 Introduction

The Khresmoi project [1], running between 2010 and 2014, delivered a multilingual multimodal search and access system for biomedical information and documents. As part of the KConnect project some of the deliverables will be extended in order to become more commercially attractive and achieve wider adoption.

This document will describe the architecture for component productization and design the interfaces of key project services. It takes into account the requirements described in deliverables D3.1 [2] and D4.2 [3]. In order to fulfil these we propose a SOA implemented both as a cloud market offering and packages for local installation.

In Section 2 we will provide a description of all software components part of KConnect and the services they expose. Information on their system requirements and possible use cases will also be given. The service APIs will also be defined and shared between both the cloud market and local deployment components.

The cloud market architecture, described in Section 3, is focused on providing access to Khresmoi technology in a SaaS and PaaS model. It is aimed at making the services easy to access and easy to use. Combined with a more flexible pricing model this should enable quicker prototyping and adoption by clients and professional service partners.

Section 4 will define the architecture and scope of the components for local installations. These will be used in the cases when KConnect technology needs to be deployed on premises due to data privacy restrictions or other non-functional requirements.

2 Components

Components from all partners will expose their functionality as RESTful web services. These are stateless endpoints which communicate over HTTP, consume and produce JSON (or other Internet media types) and use the standard HTTP verbs: GET, POST, PUT, DELETE, HEAD, etc [4]. The advantages offered by this approach include:

- client-server separation of concern
- service independence
- performance and scalability
- portability

The services, while independent of each other, will maintain interoperability through common consumable and producible models. Interactions between the services will be subject to implementation as part of the cloud market or by professional service providers for local installations. The cloud market and local installations will expose the original service APIs in order to have easier maintenance and client interoperability.

In the subsections we give a description of each component, possible use-cases and a summary of its services. In addition, we provide information on system requirements. This is important for choosing the right offerings for VMs on the AWS EC2, calculating costs and configuring Docker containers for local installation.

2.1 Knowledge Base

The Knowledge Base is a semantic warehouse that made structured biomedical data available to other components of the Khresmoi system via a set of different services. Its data layer semantically integrates different data sources identified in concert with the use case partners which produced more than 1.2 billion facts (RDF statements). The data has already been extended with new language versions and updated since the start of KConnect. A summary of the data sets is given in Table 1.

Data set	Language	User profile
Drugbank	En	GP, MP
RadLex	En, De	RS
UMLS MeSH	En, De, Fr, Cs, Sp, Se, Hu	GP, MP, RS
UMLS ICPC	En, De, Fr, Sp, Se, Hu	MP
UMLS CPT	En, De, Fr, Sp	MP
UMLS MedDRA	En, De, Fr, Cs, Sp, Hu	MP
ICD10	Se	MP
Snomed CT	Se	GP, MP, RS

Table 1: Data sets in the Knowledge Base. For each data set the loaded language versions are listed and the suggested user profiles: GP - General Public; MP - Medical Practitioner; RS - Radiology Specialist

2.1.1 Services

The KB exposes several services which are used for querying and modifying the data. They are intended for both interacting with other components and supporting UIs. A summary of the services and their API is given in Table 2.

Method	URL	Parameters		Content Type	Description
GET	/autocomplete	q*	String	JSON	Disambiguator service that finds concepts from the KB based on user text input. The contextual information provided for each resource from the suggested result set is: • preferred label • alternative labels • types • description
		limit	Integer		
		type	String		
		language	• en • de • cs • fr • sp • se • hu		
GET POST	/sparql	query*	String	SPARQL+XML SPARQL+JSON	SPARQL is a query language for RDF data. It allows users to retrieve semantic data using complex expressions on sets of triples and optional graphs.
		Infer	Boolean		
GET	/typeahead/subject	q*	String	JSON	
GET	/typeahead/predicate	q*	String	JSON	
		subject*	URI		
GET	/typeahead/object	q*	String	JSON	
		predicate*	URI		
GET	/typeahead/sparql	subject*	URI	JSON	
		predicate*	URI		
		object*	URI		

Table 2 RESTful API of the KB services. Each service is described by the relative URL at which it can be accessed, the HTTP Method supported, the query parameters and the content type. Parameters with an asterisk are required.

2.1.2 Requirements

RAM	HDD	OS	Libs	Distribution
20 GB	200 GB	Not specific	Java 7 or 8	

2.1.3 Use cases

Data in the KB can be used to answer the following use case problems, covering the requirements in D3.1 [2] and D4.2 [3]:

- provide data on drug interaction (Drugbank)
- provide data on drug contra-indications and adverse reactions (Drugbank)
- find drugs by brand names (Drugbank)
- find diseases by symptoms and vice versa (UMLS MeSH)
- find treatments for a given disease (UMLS MeSH, Drugbank)

2.2 Information Extraction

The information extraction component exposes a service for annotating documents using GATE [5] pipelines. These pipelines contain gazetteers populated with data from the KB. They are used to extract data meaningful in a biomedical context from unstructured text, such as EHR, hospital guidelines, web sites etc.

2.2.1 Services

Method	URL	Parameters		Content Type	Description
POST	/i.e./{pipeline}	document	String	application/json	Submit a document for processing by a specific pipeline
		documentUrl	URL		
		documentType*	• text/plain • text/xml • text/html • text/x-json-twitter • application/msword • application/vnd. openxmlformats-officedocument.wordprocessingml.document • application/rtf		
		annotationSelectors			

Table 3 RESTful API of the IE services. Each service is described by the relative URL at which it can be accessed, the HTTP Method supported, the query parameters and the content type. Parameters with an asterisk are required. URL values in curly brackets denote path parameters.

2.2.2 Requirements

RAM	HDD	OS	Libs	Distribution
8GB	10GB	Not specific	Java 8, GATE 8.1	1/language; possibly 1/domain

2.2.3 Use cases

The IE pipelines developed by USFD and ONTO can be used to answer the following use case problems, covering the requirements in D3.1 [2] and D4.2 [3]:

- tagging of drugs in text by brand names
- classification of findings
- term disambiguation
- link documents to structured knowledge

2.2.4 UIMA interoperability

The core of KConnect seek to satisfy many use cases within the medical domain which requires the flexibility to adapt the functionality with respect to application, language and information constraints. An important success factor to the project is to reuse existing research and technology and in particular in the domain of analysing text. There are three dominant open source frameworks with a strong ecosystem of components for text analysis; GATE [5], UIMA [6] and OpenNLP [7]. To best cater to the existing and future use cases of KConnect each of these components play a vital part. During this section we examine how to unify and include all three components into the KConnect system.

2.2.4.1 Text analysis architectures

There is a core set of text analysis architectures widely used in the community today: GATE (General Architecture for Text Engineering) by the University of Sheffield, along with Apache UIMA (Unstructured Information Management Architecture) by the Apache Software Foundation and OpenNLP also by the Apache Software Foundation.

These architectures consist of ways of representing content and ways of analysing this content.

2.2.4.1.1 GATE

The GATE architecture [5] is a component-based architecture. For representing content in GATE a document model is used. A document model is represented as content plus annotations plus features. In turn, each annotation can also contain features in the form of key-value pairs.

The architecture has a Collection of REusable Objects for Language Engineering called CREOLE, which consists of mainly two types of components:

- Language Resources (LRs), which represent entities such as documents and corpora, and various types of annotation schemas.
- Processing Resources (PRs), which represent entities that are primarily algorithmic, such as parsers or generators.

Processing resources includes the different components which add annotations to the document model. Processing resources can be combined into applications, which are sequential pipelines of processing resources manipulating a given language resource.

2.2.4.1.2 Apache UIMA

Apache UIMA is a platform for natural language processing, also with a component-based architecture. Its main framework is Java based. It provides frameworks for running pipelines of analysis engines. The architecture mainly consists of two elements:

- Common Analysis Structure (CAS), which is a common data structure to represent the unstructured information (content) being analysed as well as the metadata (annotations) produced by the analysis workflow.
- Analysis Engine (AE), is an object that performs an analysis operation on a CAS and possibly updates its content.

Analysis Engines can be combined into an Aggregate Analysis Engine, which determines the route CASes take through multiple Analysis Engines.

2.2.4.1.3 OpenNLP

The Apache OpenNLP architecture is a machine learning-based toolkit for processing of natural language text. This includes Java-based tools for tokenization, sentence segmentation, part-of-speech tagging, named entity recognition, chunking, parsing, and coreference resolution. OpenNLP also includes machine learning tools for training models with maximum entropy and perceptron algorithms.

OpenNLP does not provide an architecture for representing content and meta-data about the content.

2.2.4.2 Unification architecture

To achieve interoperability between the KConnect text analysis components and text analysis architectures described in the previous section, a unified abstract layer has to be defined for handling the difference in representing content, and also the difference in representing the components analysing the content. A characteristic for a unified abstract layer, is a model where an application is decomposed into components., making it easy to exchange each component regardless of the originating architecture. Another characteristic is a common content model for representing content and meta-data for the content (i.e. annotations), along with ways of translating the different architecture models into the common model.

To achieve this, the abstract layer should be defined as a processing pipeline with a sequence of interchangeable components. Both UIMA and GATE has a component software architecture approach, and makes good candidates for being used as the abstract layer. However, the KConnect content enrichment is based on text processing by an application pipeline built in the GATE architecture. This makes the GATE architecture the selection of choice.

Since GATE uses a modular way of creating a processing pipeline, it is easy to add document analysers from other architectures if they can be wrapped as a GATE processing resource. All architectures in question use Java-based components. If one manages to transform the GATE document model into each architectures content model and back, there should be straight forward to wrap these analysers in a processing resources which executes the original component with the transformed content model.

2.2.4.3 Wrapping a UIMA resource in GATE

Both Apache UIMA and GATE have similar architectures, based on components, which operates in similar ways. Both represent content separately from the added annotations. Both also define processing pipelines of components, processing content and annotations.

GATE	Apache UIMA
Document	Common Analysis Structure
Processing Resource	Analysis Engines
Applications (pipelines)	Aggregate Analysis Engines

Table 4 : UIMA and GATE respective components

To integrate UIMA into GATE, we would like to be able to execute an Analysis Engines in the GATE pipeline. Both GATE and UIMA are Java-based, so an Analysis Engine may be possible to execute inside the GATE environment. To make use of the UIMA component we need to help it process a document model it is used to. Using a UIMA component is therefore a 4 step process:

1. Transform current GATE document to a UIMA CAS.
2. Execute the Analysis Engine with the transformed CAS, updating/adding/removing annotations.
3. Transform the UIMA CAS back to a GATE document, with the added annotations.

GATE has a UIMA plugin for executing an Analysis Engine, along with the transformation of a Document into a CAS representation and back. To handle the transformation between Document and CAS, the user has to specify which annotations to transfer along with the content. This is configured for each embedded UIMA Analysis Engine.

2.2.4.3.1 Example components

Examples of wrapped UIMA components are included in the GATE UIMA plugin, which is included in the GATE download (https://gate.ac.uk/download/).

2.2.4.4 Wrapping a OpenNLP in GATE

OpenNLP processing components are tools which work directly with content, directly outputting the annotations produced by the analysis process, i.e. an array of tokens from the tokeniser or an array of tags from the PoS-tagger. Since the OpenNLP tools are provided as Java components it is possible to wrap each of them inside a GATE processing resource.

Since OpenNLP does not work with a specific content model, one does not have to transform the GATE document model before processing. However, the annotations produced by the OpenNLP component need to be correctly added to the GATE document model. Using a OpenNLP component is therefore a 3 step process in the wrapping processing resource:

- The wrapper processing resource initiates the OpenNLP component. If applicable, by loading a specified model.
- The embedded OpenNLP component process the tokens in a GATE document.
- The output from the processing is added as features on specified GATE annotation, on the processed document.

2.2.4.4.1 Example components

GATE has several OpenNLP components wrapped as GATE processing resources in a GATE OpenNLP-plugin, as described in the GATE documentation (https://gate.ac.uk/sale/tao/splitch23.html#sec:misc-creole:opennlp) . This includes the following components:

- OpenNLP Tokeniser
- OpenNLP Sentence Splitter

- OpenNLP POS Tagger
- OpenNLP Chunker
- OpenNLP Parser
- OpenNLP NER (Named Entity Recognition)

2.3 Machine Translation

Machine translation is the process of automatically translating a piece of text from one natural language to another language. The component is developed by CUNI and includes the following language pairs:

- English - German
- English - French
- English - Czech
- English - Swedish
- ...

It is based on the Moses system for statistical machine translation [8].

2.3.1 Services

Method	URL	Parameters		Content Type	Description
POST	/translate	action*	String	application/json	Submit a text for translation.
		sourceLang*	ISO 639-1 code		
		targetLang*	ISO 639-1 code		
		text*	String		
		systemId	String		
		alignmentInfo	Boolean		
		nBestSize	Integer		
		tokenize	Boolean		
		segment	Boolean		
		deTokenize	Boolean		

Table 5 RESTful API of the MT services. Each service is described by the relative URL at which it can be accessed, the HTTP Method supported, the query parameters and the content type. Parameters with an asterisk are required.

2.3.2 Requirements

RAM	HDD	OS	Libs	Distribution
16 GB	15 GB	Linux	C++, Python 2.x	1 node / language pair

2.4 Semantic Search

The semantic search component is based on the MIMIR semantic search system [5]. It provides analytics and search over annotated text.

2.4.1 Services

Method	URL	Parameters		Content Type	Description
GET	/mimir/suggest/{text}			application/json	Retrieve suggestions from the annotation schema.
POST	/mimir/search	query	String	application/json	Query the mimir index for documents.
		offset	Integer		
		size	Integer		

Table 6 RESTful API of the Semantic Search services. Each service is described by the relative URL at which it can be accessed, the HTTP Method supported, the query parameters and the content type. Parameters with an asterisk are required. URL values in curly brackets denote path parameters.

2.4.2 Requirements

RAM	HDD	OS	Libs	Distribution
2GB	Depends on data size	Not specific	Java 7	

2.4.3 Use cases

The IE pipelines developed by USFD and ONTO can be used to answer the following use case problems, covering the requirements in D3.1 [2] and D4.2 [3]:

- which patients are prescribed non-generic drugs when they can be on generics
- finding patient groups based on symptoms, treatment, disease or clinician
- find patients that have been prescribed interacting drugs
- find similar patients
- suggest articles based on medical record

2.5 Service dependencies

The services provided by KConnect components, while being independent pieces of software, exhibit dependencies between each other in order to provide the greatest value. Figure 1 gives a schematic overview of the component interactions.

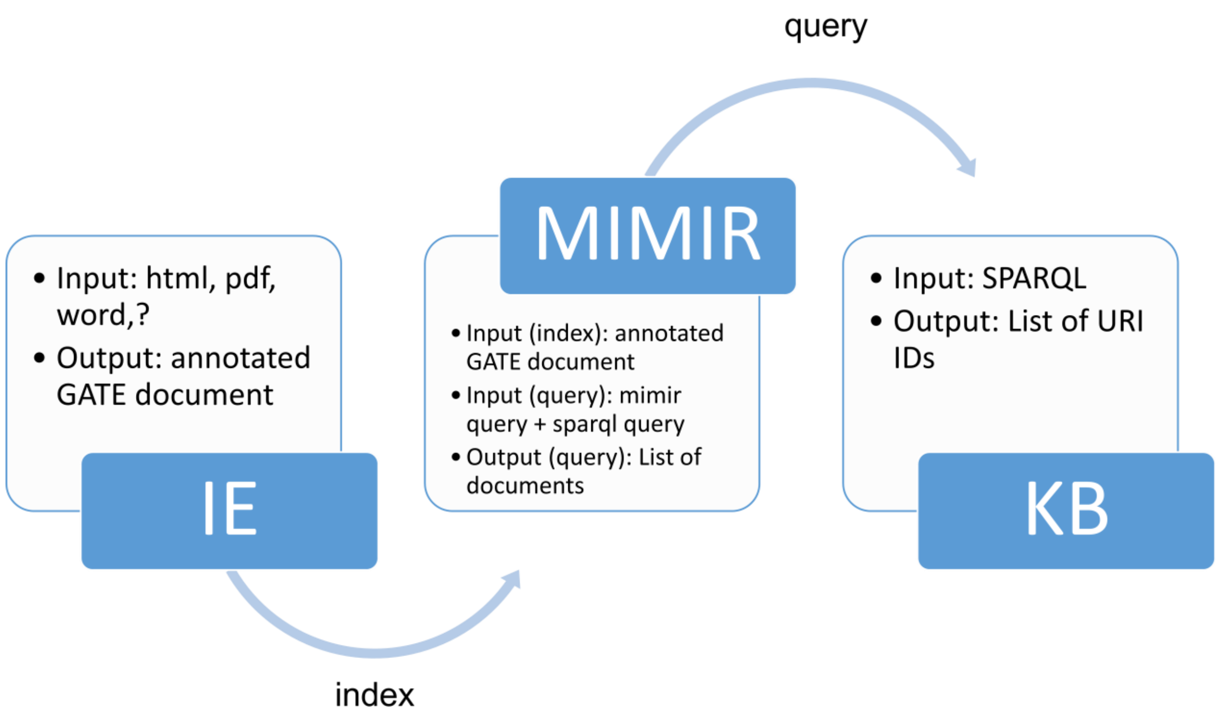
query
MIMIR
• Input: html, pdf, word,?
• Output: annotated GATE document
IE
• Input (index): annotated GATE document
• Input (query): mimir query + sparql query
• Output (query): List of documents
• Input: SPARQL
• Output: List of URI IDs
KB
index

Figure 1: KConnect service dependencies. The workflow shows the links between the components and the common inputs/outputs.

In order for the components to interact seamlessly, they need to exchange information in common models and formats:

- GATE pipelines populate their gazetteers with labels from the KB. This is done using SPARQL queries
- GATE pipelines produce annotated documents in common GATE document formats
- MIMIR consumes GATE documents and semantically indexes the annotations in them
- MIMIR queries can contain SPARQL queries which executed against the KB constrain results wrt structured data

3 Cloud Market

The cloud market will be based on the S4 platform partially funded by the DaPaaS [9], the AnnoMarket [10] and proDataMarket [11] projects. Its aim is to provide the KConnect technology as low-cost, on-demand services. Prior to KConnect it already provided a scalable architecture for IE and RDF Database packaged in a Platform-as-a-Service. As part of KConnect we will extend the platform to provide:

- Biomedical IE
- Biomedical Data-as-a-Service
- MT-as-a-Service
- Semantic Search-as-a-Service

This should enable 3rd parties to privately host relevant biomedical data sets, perform text analysis on their data, semantically index and search said data and translate it when necessary. The cloud market uses a pay-per-usage cost model. Each account is given a free quota per service per month. Usage above this quota needs to be paid for. The quotas are defined separately for each component. Below we give an overview of the general architecture as well as details for each component.

The cloud market is deployed on the AWS cloud and is based on highly reliable and flexible AWS infrastructure, e.g. the SQS, S3, DynamoDB and Glacier [12]. Each KConnect component will be packaged separately and consists of two units:

- back-end - performs the actual work. Multiple back-end nodes can be instantiated either for scalability or for providing individual clients with private instances
- front-end - exposes RESTful web services and takes care of user authentication and access control, routing traffic to the back-end nodes, monitoring and logging

Having the back-ends on separate VMs allows us to deal with the services depending on different operations systems and library versions, e.g. Java 7 vs. Java 8, while enabling us to choose the most efficient EC2 offering per component.

Separately, a common facade takes care of access control, quota management and possible auto-scaling of front-end nodes. Data about accounts, quotas and logs is stored in DynamoDB [12]. This data is automatically backed up to Glacier [12]. In addition, a separate web application is provided for accessing platform services through a graphical user interface. It also provides reports about statistics and management capabilities.

The cloud market uses API Keys for stateless authorization. The advantages include flexible access control, lower risk of stolen passwords, time based keys expiry and the option for run-time enabling disabling of keys. Each user is given a generated API Key and API Key Secret pair on registration. Further API Keys associated with the account can be generated by the owner of the account. Each time a client makes a request to one of the cloud market services, a valid key-secret pair must be provided. A schematic representation of the architecture is given in Figure 2.

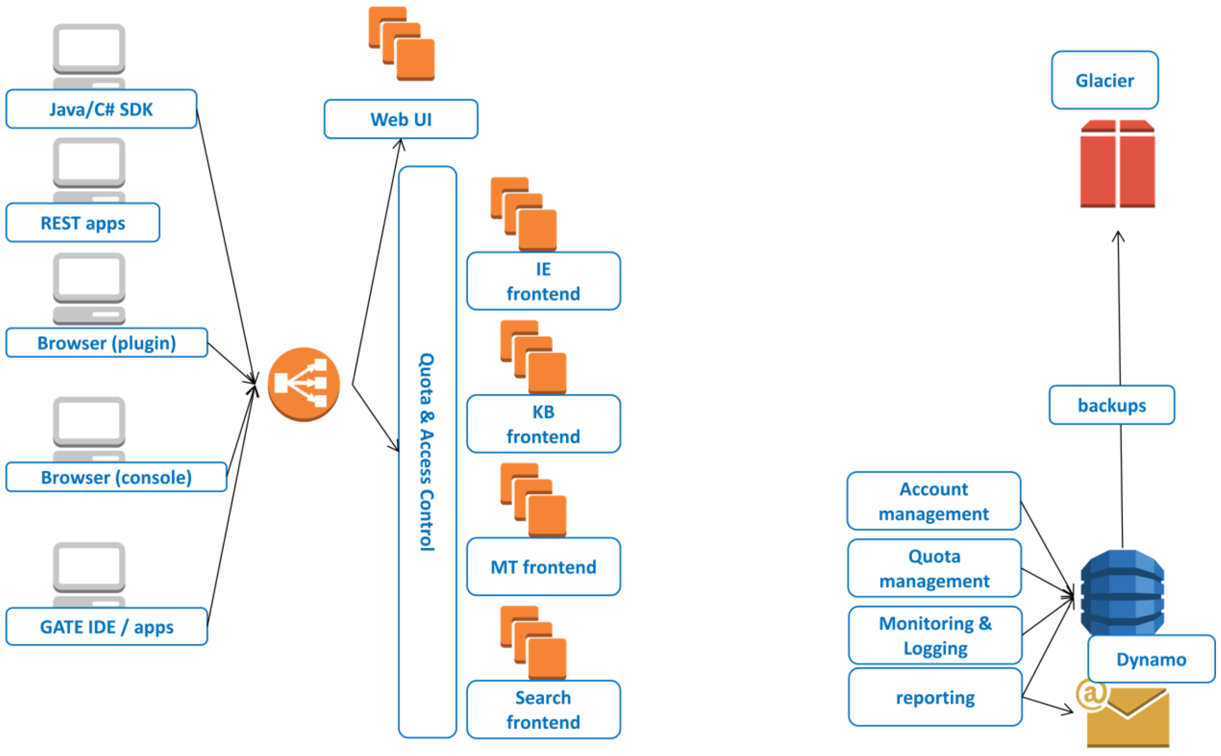

Figure 2: General architecture for the KConnect Cloud Market. Components are isolated as functional units each with its own front-end and back-end. A common access and quota management is used for all services. Data about accounts, quotas, logs and reports is stored in DynamoDB and backed up to Glacier. A separate Web UI allows users to view statistics, reports and try out the services.

3.1 Information Extraction

The information extraction component back-end will consist of VMs that run GATE, called processing nodes. Documents sent to the IE fronted are put into the SQS, Amazon's fully-managed scalable messaging queue service. The processing nodes each poll the queue for documents. Once a message is available, the back-end node checks if the desired pipeline is already instantiated on it. If not, it checks a DynamoDB based registry and loads the corresponding entry from S3. Processed documents are sent back to the front-end. The auto-scaling service checks the size of the queue and if it reaches a threshold a new processing node is set up.

Improvements planned for the IE cloud service focus on providing pre-processing capabilities similar to the current processing nodes and allow corpora of documents to be uploaded to S3 buckets and annotated in bulk.

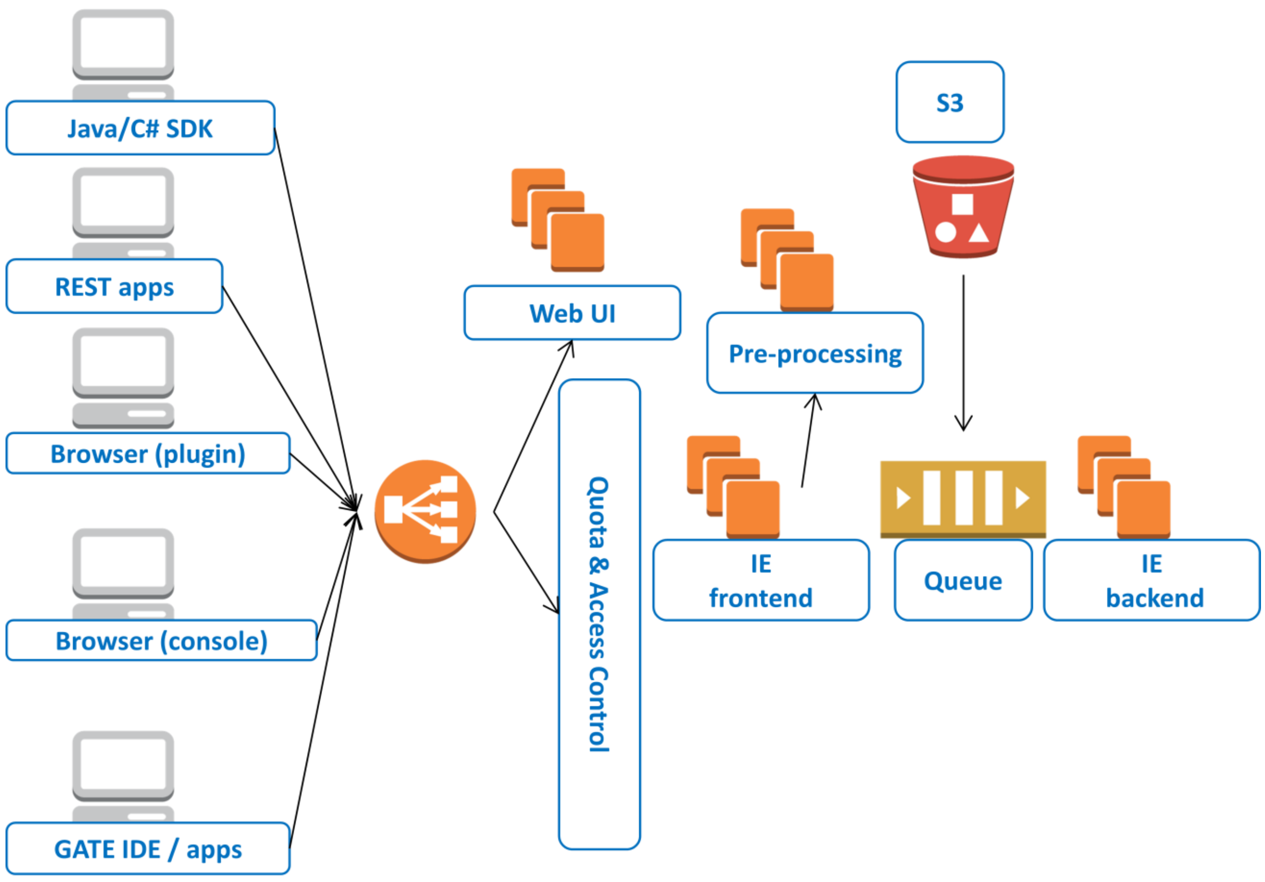

Figure 3: KConnect Cloud Market architecture for IE. The front-end component communicates with the back-end nodes via a messaging queue.

3.2 Knowledge Base

Moving the KConnect knowledge base to the cloud has the goal of making its data more easily accessible to a wider audience. As described in Section 2.1, the KB consists of a web application exposing services and a data layer integrating biomedical data sources. This unit will comprise the back-end of the Cloud Market component. The web application and a GraphDB SE will be deployed on a suitable EC2 machine, while the data itself will reside on mounted high- availability low-latency EBS volumes. This will make use of the benefits the service provides like automatic replication, encryption and scalability.

Moreover, as one of the plans of the project is to make the data sets easily available to clients, their raw RDF will be stored in S3 buckets. These buckets will be monitored by the KB for updates, which could be easily applied. This process will happen according to the knowledge sustainability methodology designed as part of the Khresmoi project. Moreover, separate data sets could be made available to clients according to their usage quota. The raw data will be backed up to Glacier as well.

The front-end component will take care of access control, quota management and data access management to S3 buckets. In addition, if multiple instances of the back-end component are present due to excessive load, it will take care of routing request appropriately. A schematic representation is given in Figure 4.

Clients of the Knowledge Base will be assigned quotas of number of queries per month. Separate accessibility quotas per data set will be assigned for direct access to the RDF data.

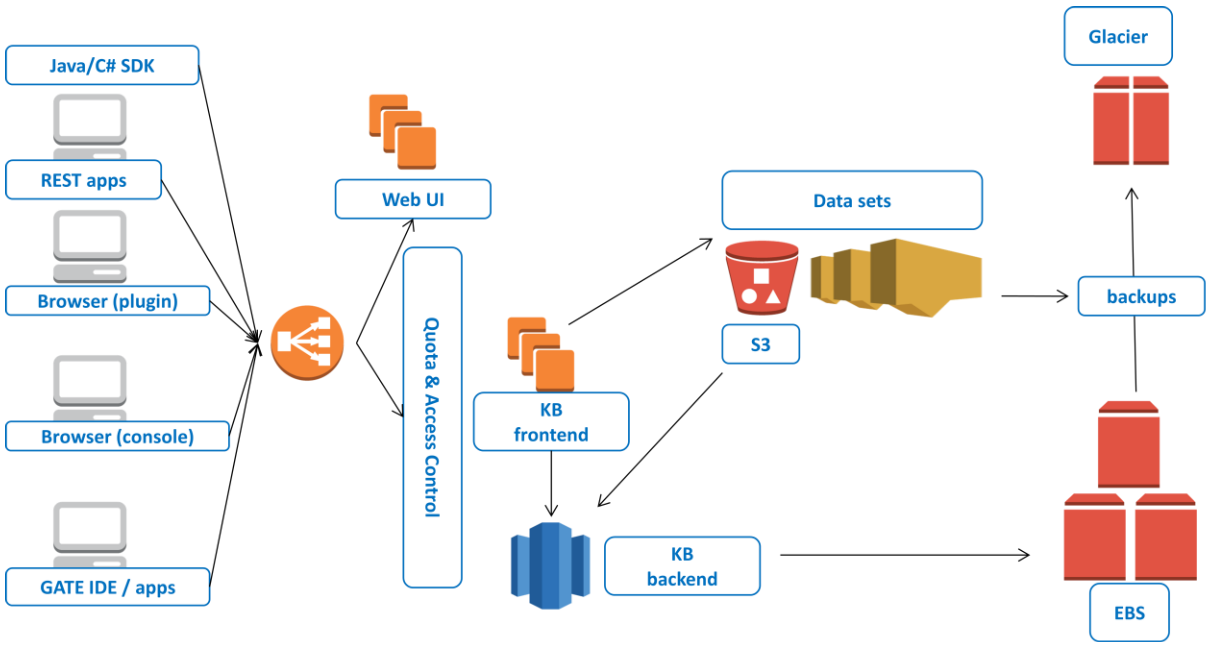

Figure 4: KConnect Cloud Market architecture for the Knowledge Base. The RDF data is stored on S3 and loaded to the KB repository when updated. The GraphDB files are stored on EBS volumes. The KB back-end services are deployed on an EC2 VM. The front-end takes care of access control to the KB and the RDF data and routing requests.

3.3 Machine Translation

As described in Section 2.3, the MT service is quite demanding with respect to RAM per node. Given the number of language pairs and the current EC2 offerings we have calculated that providing an instance for each one will be too costly in the context of the KConnect budget. Therefore, we have decided to initially not move the entire MT system to the AWS, but to provide a proxy from the Cloud Market to the service deployed at CUNI infrastructure. In this way we can apply to it the common infrastructure for access control, quota management and logging.

In the future we will analyze the demand for a fully managed MT service and move those language pairs that are required to the AWS cloud.

3.4 Semantic Search

As mentioned in Section 2.4 the Semantic Search component is based on MIMIR. For the Cloud Market we plan to make MIMIR instances available as fully managed services. This means that Cloud Market users will be able to spin up private MIMIR instances on demand and configure them according to their use case. Restrictions to the number of instances and configuration parameters will be according to the user's usage quota. Some pre-configured and pre-populated instances might be made available for usage by the consortium partners.

Each back-end node will run a Docker engine [13] (see also Section 4), able to spin up multiple containers with a pre-packaged MIMIR web application. The index data of each container will be persisted to a mounted EBS [12] volume. Thus VM resources will be optimized. Each container will register its local IP address and port to a persistent coordinator service. This is important so that the component front-end will be able to correctly route request to the back-end nodes. In addition, it provides failover capability, should a back-end node need to be replaced. A schematic representation is given Figure 5.

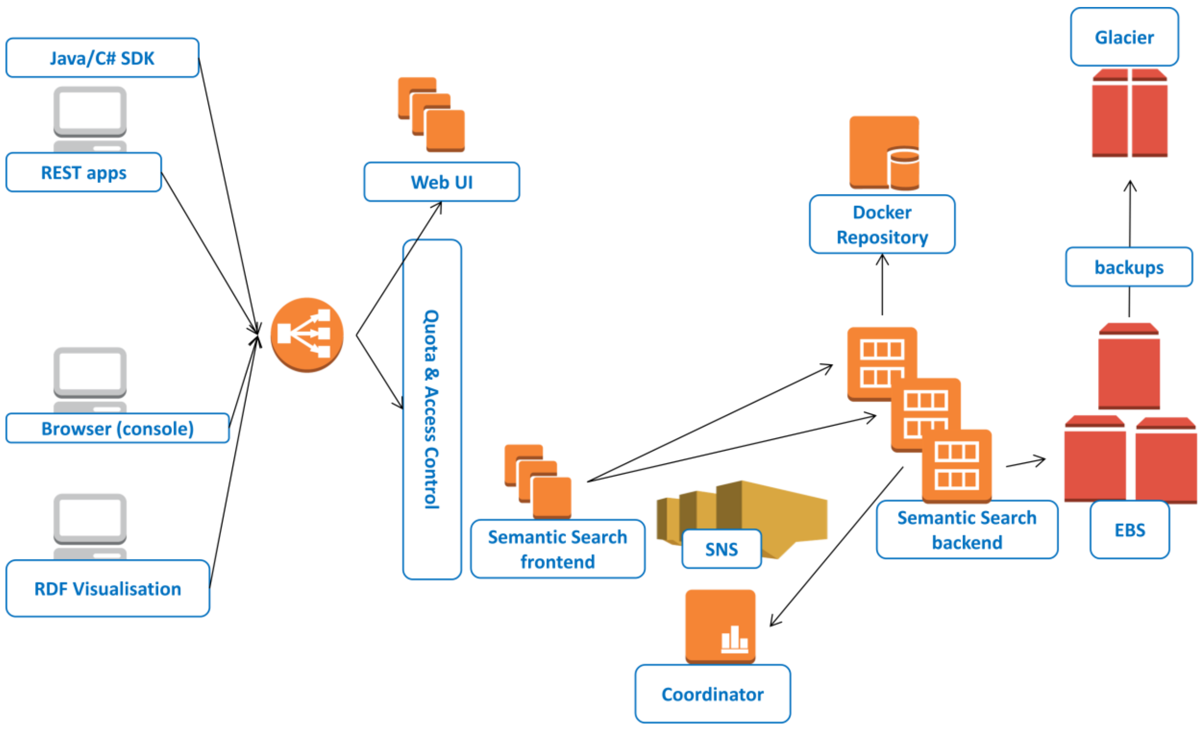

Figure 5: KConnect Cloud Market architecture for Semantic Search. The component front-end starts Docker containers with the MIMIR application on back-end nodes and routes traffic from the RESTful service to the correct container. Data is stored on EBS volumes, which are backed up to Glacier. The SNS is used to keep the front-end routing table up-to-date with back-end state.

4 Local installation

In many cases clients that are interested in using KConnect technology will not be able to do so using the Cloud Market alone. This can be due to network latency times, data privacy restrictions, legacy systems or other non-functional requirements. Currently, such adopters in the KConnect project are the KCL and healthcare providers in Sweden. Therefore, the need arises to make the components available for local installation. Such a task is not trivial, because different clients will be running different infrastructure which combined with the different requirements for each component and the fact that they are maintained by multiple vendors calls for a integrated strategy for distributing them.

One common approach is to publish pre-packaged VMs. However, these VMs can be packaged in different ways, have restrictions on network connectivity and may waste host operating system resources.

A modern alternative is Docker [13]. It is an open platform for building, shipping and running distributed applications. It allows software developers to package their applications and their dependencies in a Docker image using the Docker build system, making it a standardized process. Images contain a complete file system, including a runtime, system tools and system libraries. An image can be used to spawn one or more containers (instances) on the Docker engine. This lightweight runtime uses runC (formerly known as libcontainer) [14], which allows containers to share resources with the host operating system, e.g. kernel and RAM, while still isolating the applications from different containers . It also utilizes layered file systems like AuFS [15], which enables it to provide read-only writes to shared resources and to provide read/write mounting points to containers. Docker enforces isolation of applications by having single-process containers. Docker engine is available for Windows and Linux systems, meaning that your images are host OS agnostic. Finally, the Docker engine also manages the network connectivity of each container. Figure 6 gives a schematic overview of the differences between a VM and Docker.

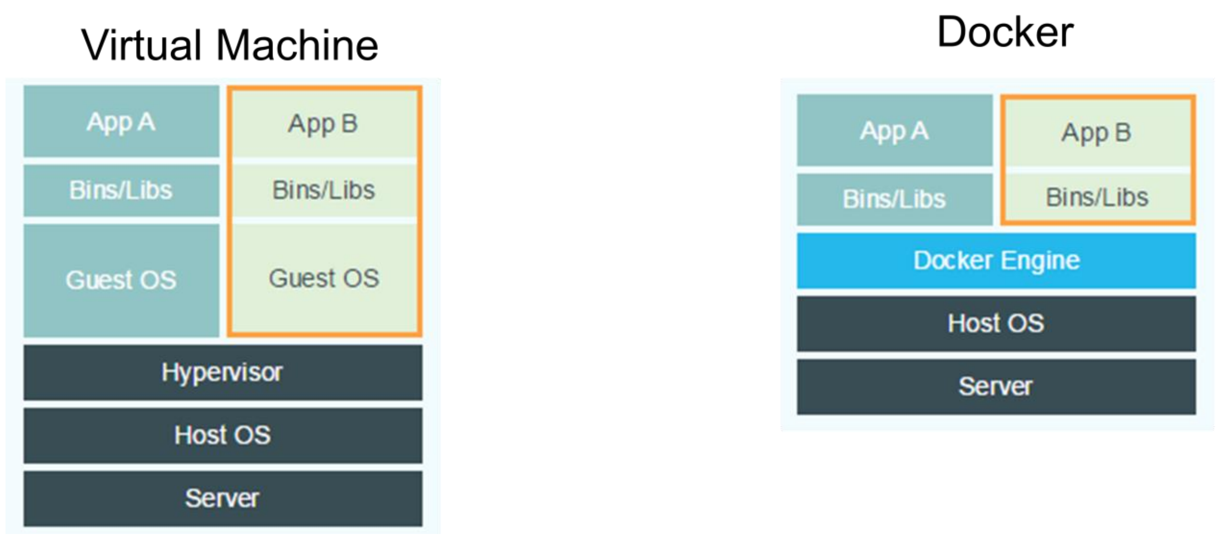

Figure 6: Comparison of full VMs and Docker Images. The major difference between running a VM and a Docker container is the absence of the Hypervisor and Guest OS in the latter, thus using less resources.

Docker images can be easily shared between users and locations via online registries. DockerHub is such a registry for storing and sharing Docker images which is publicly available. The AWS, among other vendors, provides private registries as well.

Because of the advantages mentioned above we decided to use Docker to package the core KConnect components, each in its own image. This isolation is fine because we designed the components for a

SOA, having them communicate with each other over REST services (see Figure 1). Docker containers can be "linked" to one another at run-time. This maps a containers local network IP address to a resolvable domain for another container, enabling the configuration of dependency services to happen at build time.

Data sets that are required by the components will be distributed separately as archives and depending on the client needs. It will be straightforward to make these accessible to component containers via preconfigured mount points.

It is important to also note that while we provide the packages for local installations and a detailed installation and integration guide, we will not provide some cloud features out-of-the-box:

- auto scaling
- load balancing
- backups
- failover recovery
- access control
- quota management

While some of these features are easy to implement with Docker, e.g. starting up additional instances in cases of high load, they are all dependent on the specific client use-case, procedures and infrastructure. We expect that professional service providers will take care of tasks such as integrating with an organization's LDAP, data encryption and backup etc.

5 Conclusion

In this document we presented the list of core KConnect components and their RESTful APIs for interacting in a SOA. In order to satisfy the requirements of different usage scenarios, we presented two architectures - one for a cloud platform and one for local deployment.

The cloud platform is based on the AWS, making extensive use of its services to achieve good performance, scalability and reliability. It is designed to allow clients to rapidly access KConnect technology at very little initial cost. It's architecture uses a 3 layer design, whereby common functionalities like access control, quota management, logging and reporting are shared between all components. However, the functional units - the components - are isolated from each other in separate VMs and/or Docker containers, communicating only over established web standards and APIs. This ensures their future interoperability. It also enables the addition of further services to the platform that expose a RESTful API.

The architecture for local installations re-uses the concept of component isolation in separate systems by utilizing Docker containers as well. This provides an up-to-date solution for cross-platform installation. Clients will benefit from the ability to deploy the technology on premises using a well adopted tool that optimizes hardware resource usage and makes scalability easier.

The standardized packaging and reduced discrepancy between cloud market and local installation will benefit technology providers by lowering code amount, wide community support and reduced maintenance cost.

6 References

[1] KHRESMOI, GA 257528, http://khresmoi.eu

[2] C. Boyer, L. Dolamic, J. Brassey, A. Roberts, E. Jofoldi, Z. Varju, Z. Farago, J. Palotti, V. Stefanov. Requirements for Vertical Search Solutions. KConnect project deliverable D3.1. July 2015

[3] A. Rukat. Fr. Axelsson. J. Sjöberg, R. Berger, R. Dobson. Requirements for Medical Record Analysis and Search. KConnect project deliverable D4.2. August 2015

[4] R. Fielding. Chapter 5: Representational State Transfer (REST). Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine. 2000

[5] M. A. Greenwood, V. Tablan and D. Maynard, GATE Mimir: Answering Questions Google Can't, Proceedings of the 10th International Semantic Web Conference ISWC2011, 2011.

[6] Apache UIMA, https://uima.apache.org/

[7] Apache OpenNLP, https://opennlp.apache.org/

[8] Moses, www.statmt.org/moses/

[9] DaPaas, GA 610988, http://project.dapaas.eu/

[10] AnnoMarket, GA 296322, https://annomarket.eu/

[11] proDataMarket, GA 644497, http://blog.prodatamarket.eu/

[12] AWS Cloud Products, https://aws.amazon.com/products/

[13] Docker, http://www.docker.com

[14] runC, https://github.com/opencontainers/runc

[15] AuFS File Systems, http://aufs.sourceforge.net/