Knowledge Base Shipping to the Linked Open Data Cloud

Natanael Arndt
Universität Leipzig
Augustusplatz 10
04109 Leipzig, Germany
arndt@informatik.uni-leipzig.de

Markus Ackermann
Universität Leipzig
Augustusplatz 10
04109 Leipzig, Germany
ackermann@informatik.uni-leipzig.de

Martin Brümmer
Universität Leipzig
Augustusplatz 10
04109 Leipzig, Germany
bruemmer@informatik.uni-leipzig.de

Thomas Riechert
Hochschule für Technik, Wirtschaft und Kultur Leipzig
Gustav-Freytag-Str. 42A
04277 Leipzig, Germany
thomas.riechert@htwk-leipzig.de

ABSTRACT

Popular knowledge bases that provide SPARQL endpoints for the web are usually experiencing a high number of requests, which often results in low availability of their interfaces. A common approach to counter the availability issue is to run a local mirror of the knowledge base. Running a SPARQL endpoint is currently a complex task which requires a lot of effort and technical support for domain experts who just want to use the SPARQL interface. With our approach of containerised knowledge base shipping we are introducing a simple to setup methodology for running a local mirror of an RDF knowledge base and SPARQL endpoint with interchangeable exploration components. The flexibility of the presented approach further helps maintaining the publication infrastructure for dataset projects. We are demonstrating and evaluating the presented methodology at the example of the dataset projects DBpedia, Catalogus Professorum Lipsiensium and Sächsisches Pfarrerbuch.

Keywords

Docker, Knowledge Base, Linked Open Data, Web of Data, Semantic Web, SPARQL Service, Container-based Virtualisation

1. INTRODUCTION

Modelling and representing knowledge using RDF has become an established tool throughout diverse domains. However, the process of publishing and maintaining RDF knowledge bases on the World Wide Web in general and on the Linked Open Data cloud (*LOD cloud*) in particular requires technical expertise. While allocating resources to knowledge engineers who model the domain of interest and create the knowledge bases is easy to justify, making these datasets available as Linked Data is equally important. Although this should be a mere technicality to Linked Data experts, setup and maintenance of knowledge bases published as Linked Data are time consuming tasks that are often overlooked. These complications become clearer by looking at evaluations of LOD cloud datasets, such as by Schmachtenberg et al. [13], where only 19.47% of proprietary vocabularies were fully dereferenceable and only 9.96% of datasets provided SPARQL endpoints.

Another facet of this issue is reproducibility of experiments performed on RDF knowledge bases. Properly comparing own approaches to prior research entails working with the same data. However, knowledge bases evolve and public endpoints may either stop serving the data that was originally used in experiments, or they might not be powerful enough to provide the data in reasonable time spans. Finding the data, loading it into a local triple store and using this local mirror to perform the experiments is the usual way to counter this problem. Publication of knowledge bases to the LOD cloud usually comprises the following steps:

1. Installation of a triple store
2. Loading the data to be published into the triple store
3. Setting up a (publicly available) SPARQL endpoint
4. Providing a presentation application to support the exploration of the knowledge base
5. Ensuring de-referencability of IRIs occurring in the published knowledge base
6. Maintain the setup and ensure its availability

Performing these steps requires a certain level of technical knowledge and understanding of the individual server components. This often requires a knowledge engineer resp. domain expert to either consult a system administrator or to invest a significant amount of time to selectively acquire DevOp-competences that often diverge far from the original domain and core research interests of the knowledge engineer.

SEMANTiCS '15, September 15 - 17, 2015, Vienna, Austria

 ISBN 978-1-4503-3462-4/15/09…$15.00

DOI: http://dx.doi.org/10.1145/2814864.2814885

To alleviate these issues, we propose the methodology of *Dockerizing Linked Data* (DLD, http://dld.aksw.org/): an approach of a containerised Linked Data publication infrastructure. By using *Docker* container virtualisation, it provides benefits regarding the maintainability and ease of setup, through modularisation of individual components following the principle of micro services[1]. In addition, it enables easy mirroring of a setup on other computer systems. Apart from the pure presentation of the knowledge base we are also taking a look at use cases where write access is needed. A detailed description of the overall architecture is given in section 3. In section 3.1 we specify a way of containerising a triple store together with the used access and management interfaces. Different ways for loading the actual data into a containerised triple store and how to backup the data are presented in section 3.2. In section 3.3 we show how to create and attach containers for a HTML representation of the data, in addition to the SPARQL endpoint. We are using the presented approach to provide a ready-to-run DBpedia SPARQL endpoint setup to create a local mirror as replacement for the highly demanded central endpoint (section 4.1). The approach is also used to run and maintain the two prosopographical knowledge bases *Catalogus Professorum Lipsiensium* (section 4.2) and *Sächsisches Pfarrerbuch* (section 4.3) which both share many requirements to the software setup and thus can benefit from a modular reusable software stack.

2. BACKGROUND AND TOOLING

Container-based operating system virtualisation is a technology used to provide an isolated execution environment for multiple individual applications sharing the complete hardware and the host's core operating system components while each container has its own filesystem [12]. In contrast, with full- and para-virtualisation technologies each virtual machine brings its own operating system kernel which increases the resource footprint on the host system. Examples of container-based operating system virtualisation technologies are e. g. FreeBSD jail[2], Linux Containers (LXC)[3] and *Docker*[4].

Docker, albeit a comparatively young project started in 2013, experienced a very rapid increase in its popularity and community uptake [10]. Initially, the main contribution of *Docker* was not advancing operating system or virtualisation technology itself, but rather providing a feature-rich and straightforward way to use APIs and interfaces by combing pre-existing and maturing open-source virtualisation technologies. At its inception, *Docker* utilised LXC to create kernel-level process namespaces and control groups for each container to establish an isolated process tree with holistic facilities to control and modify its resource consumption and network traffic. Current releases of *Docker* now use a process container library called *libcontainer*, a sub-project of the original *Docker* initiative.

Another important *Docker* feature is the manner in which filesystems for the isolated processes are managed. On bootstrapping, each container has its own replica of a root filesystem defined by a *Docker image*. Harnessing layered and versioning copy-on-write filesystems (AuFS or btrfs), changed parts of the filesystem (*diffs*) are stored alongside a copy of the corresponding filesystem parts before destructive operations, allowing to restore any filesystem state during the lifecycle of a container and to trace the history of write operations. Each of these states of the filesystem can be tagged and reused as a new *image*, allowing new containers to start their lifecycle with that exact filesystem state.

In addition to the aforementioned prototypical filesystem tree, a *Docker image* groups information about which root process resp. *entry point* to invoke, potentially required adjustment to the execution environment of the process (i. e. the working directory and environment variables) and which resource limits must be respected. The preferred way to create an image is defining aforementioned details in a *Dockerfile*[5]. This *Dockerfile* also allows to declare process invocations and to add additional files to the *base image* to adjust aspects of the configuration in the *base image*. *Dockerfiles* also have the advantage of an even more transparent formal description of steps to obtain a desired initial configuration environment for processes to be run in containers.

With *volumes Docker* provides the possibility to mount directories of the host filesystem or other containers into the filesystem of a container[6], exempt from the versioning by the copy-on-write filesystem in favour of increased I/O performance. Each container receives its own virtual network interface, allowing it to connect to other nodes on the internal virtual network and to the Internet through a host bridge. With the `EXPOSE` keyword a container can select which ports are available to the host system. To provide services via network connections from within a container to other containers, *Docker* offers a linking mechanism[7]. Using *links*, stable host names and ports can be achieved for inter-container network communication, further environment variables from upstream containers are made available as well. It provides a simple but effective way to interchange small pieces of information required for successful orchestration.

Container-based operating system virtualisation has the advantage of keeping all dependencies necessary to run an application together while major parts (hardware and operating system kernel) are shared between instances. We expect that the main technical setup for users of this project will be commodity hardware with moderate performance characteristics. In such a context, the desired orchestration of numerous micro services in isolated environments with a complete full- or para-virtualised system for each service would be prohibitively taxing on limited computation power. However, recent comparative research also indicates significant performance advantages in using container-based

[1] We understand *micro services* as small independently deployable services, each responsible for a well defined sub-task of the whole setup; cf. http://martinfowler.com/articles/microservices.html

[2] FreeBSD jails in the FreeBSD handbook: http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/jails.html

[3] LinuxContainers.org project webpage: http://linuxcontainers.org/

[4] The *Docker* project webpage: http://docker.io/

[5] Reference documentation about the syntax of a Dockerfile: http://docs.docker.com/reference/builder/

[6] *Docker* documentation about *volumes*: https://docs.docker.com/en/latest/use/working_with_volumes/

[7] *Docker* documentation about the linking system: http://docs.docker.com/userguide/dockerlinks/#connect-with-the-linking-system

virtualisation on server hardware [6], incentivising this approach also for scenarios when a DLD setup is deployed to cater services for a broader user audience. *Docker* in particular provides tooling which allows simple setup of new images and containers and provides means for portable and reproducible image creation with *Dockerfiles*. Furthermore, *Docker* can be installed on the three mayor operating systems Linux, Mac OS X and Windows and can be deployed to mayor cloud platforms as well[8].

Docker Compose[9] (formerly known as *fig*[10]; in this paper just *Compose*) is a command line tool for configuring and managing complex setups of *Docker* containers. These setups are declared in YAML[11] configuration files that declare a set of *images* for which containers should be spawned. They also specify desired *links*, shared *volumes* and other parameters for image instantiation. These additional configuration options would otherwise always needed to be manually (re-)specified on the command line interface for the *Docker* process. *Compose* coordinates simultaneous starting and stopping of all containers belonging to a *Compose*-configuration as a group and offers helpful aggregation of the log outputs from member containers.

3. ARCHITECTURE

In our architecture we are dividing the components needed for shipping a knowledge base in a number of general types of containers: *store*, *load & back-up* and *presentation & edit* components as depicted in fig. 1. Each of those components is exchangeable by containers implementing the necessary tasks resp. interfaces. The *triple store* component can be considered the core of the setup while all other components are reading from it or writing to it. It contains an RDF graph database, is responsible for persisting the knowledge base and provides interfaces for querying and optionally altering the contained data. A *load* component is responsible for loading the knowledge base into the triple store if it is not shipped with preloaded data. *Back-up* components are responsible for frequently saving the contained data in a secure place to avoid data loss, if writing access is implemented in the setup. The *presentation & edit* components can be any service which provides means for browsing and exploring the data of the knowledge base. In an authoring use case, these components also provide services for updating the knowledge base.

When designing the architecture we were following the following principles: *Convention-over-configuration:* to decrease the amount of configuration items required for a DLD setup. We introduce conventions (especially naming conventions) that the configuration tool expects to be fulfilled. *Docker images as component boundaries (resp. micro services):* whenever possible, create the individual component images self-sufficient, with little or no assumptions of the modes of operation of accompanying components in a DLD setups. *Agnostic to the choice of a programming language:* specifications are formulated and data formats were chosen so that a broad choice of programming languages provide the capabilities and libraries to allow implementation of component configuration and supervision for additional component submissions by third parties. Another simplifying convention in the current state of the project is allowing for at most one *load*, *backup* and *store* component each per setup, that will be referenced by exactly that name in the *Compose*-configuration. (However, an arbitrary number of *present & edit* components is possible.)

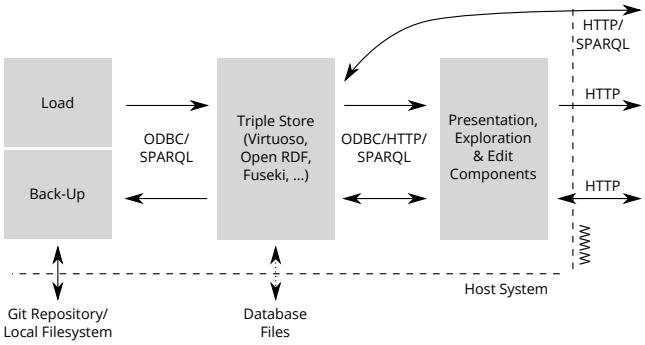

Figure 1: Architecture and data-flow of the containerized micro services for publishing knowledge bases

3.1 Triple Storage

A *triple store* is a graph database management system capable of storing the RDF triples of RDF knowledge bases. It usually provides a SPARQL Query [14] interface which can also be used with SPARQL Update for adding and removing triples. Containers for the *store* component typically consist of the store binaries and their dependencies together with its configuration files and a script-based management process. This process, if necessary, adjusts the configuration files prior to store initialisation and supervises the store process, possibly also restarts the store process on (unexpected) termination for basic failover. To ensure the persistent storage of saved triples independent of the lifetime of the store container, data directories for the provided store are kept in a volumes (depending on configuration either a filesystem location of the host system or a *Docker*-managed volume).

For the proof of concept we started with component images for the triple store implementations *Open Link Virtuoso*[12], *Apache Jena Fuseki*[13] and *Sesame Native Store*[14]. Other DLD-compliant store images can also be provided by third parties if they can be set up in a Linux environment and configured to work according to a small set of conventions detailed in section 3.4.

3.2 Load and Back-Up

A *load component* is responsible for pre-loading a triple store with an RDF knowledge base when initialising a container setup. The actual RDF knowledge base file can either be provided as a file on the host system, can be fetched

[8] *Docker* installation instructions for Linux, Mac OS X, Windows, various cloud computing platforms and other operating systems: https://docs.docker.com/installation/

[9] *Docker Compose* in the *Docker* Documentation: http://docs.docker.com/compose/

[10] Project webpage of the *Docker Compose* predecessor *fig*: http://www.fig.sh/

[11] The YAML specification in the current version 1.2: http://www.yaml.org/spec/1.2/spec.html; YAML is a superset of JSON

[12] Open Link Virtuoso product webpage: http://virtuoso.openlinksw.com/

[13] Apache Fuseki documentation:http://jena.apache.org/documentation/serving_data/

[14] Sesame alias RDF4J project webpage: http://rdf4j.org/

from an online source or can come from some backup location. During the *data loading phase*, right after the *configuration phase*, a loading container starts to connect to the configured triple store and injects the necessary data, which is then available at the triple store during the *service phase* (cf. section 3.4). A load container can either connect to the triple store via the standard SPARQL interface or use proprietary side-loading mechanisms.

Back-up containers provide a service for querying the data of a triple store and storing it in a safe location. They have the same lifecycle as the triple store to backup. This is usually done by configuring a *cron* job which is executed in fixed intervals to dump all data from the triple store. Components responsible for synchronising different setups also belong to this category of images.

3.3 Presentation and Publication

Presentation and publication images are meant to provide exploration interfaces to the knowledge base. This can be any application capable of fetching data from a triple store to provide the user with some generic or special view to the data. In this proof of concept we are providing the generic exploration interfaces pubby [5], snorql[15] and OntoWiki [7]. OntoWiki also provides the possibility to create very specific exploration and publication interfaces with its *site extension*[16]. The capability of providing domain specific views and editing components is used in the Catalogus Professorum Lipsiensium (cf. section 4.2) and Sächsisches Pfarrerbuch (cf. section 4.3) use cases. A limitation of OntoWiki is that it currently only supports the Virtuoso triple store.

A presentation container is linked to the triple store and connects to the database with the credentials given in the environment variables from the triple store container. It is available throughout the whole *service phase* (cf. section 3.4). Communication with the triple store is implemented in SPARQL or any custom interfaces available. The presentation interface finally is usually exposed as HTTP interface (at port 80) which than can be made available to the WWW through the host network bridge.

3.4 Container Design and Conventions

DLD-compatible images expose a set of required metadata items using the *Docker* `LABEL`[17] declarations. Labels are arbitrary key/value-pairs that can be attached to docker images and containers. The DLD labels are used for defining the type, special requirements and configuration options of an image. Special configuration options can be for example expected environment variables or volumes required to be provided.

Label values for DLD keys are JSON strings, which also allow to define *Compose* configuration fragments to be declared. Table 1 exemplifies the usage of labels in DLD.

Settings adjusting the behaviour of component containers during their execution are defined by declaring environment variables. This implicitly allows component containers to inspect settings of components that it is linked to. This allows for instance load components to be informed about credentials that have been set for the store to gain additional privileges for side-loading operations[18]. Some environment variables like `SPARQL_ENDPOINT_URL` and `DEFAULT_GRAPH` are expected by convention for all components and can be defined universally in the DLD configuration file.

Usually the lifecycle of component *containers* is split into a short *configuration phase* immediately after their instantiation, which is followed by an optional *data loading phase* before each container starts serving their encapsulated service in the *service phase*. The configuration phase allows for adjustments of the service setup, e. g. desired authentication details for a triple store, which named graph should be presented by default by a presentation component or the host name under which such a component is exposed to the WWW (e. g. to ensure correct minting of de-referenceable Linked Data descriptions for resources).

In addition to aforementioned conventions applicable to all components, DLD also builds on some conventions for specific component types. For instance RDF data dumps to be processed by load components are expected to be placed in a mounted volume named `/import` inside the container.

Label Key	
Component	`Label Value` *Description of Semantics*
`org.aksw.dld.type`	
VOS 7	`"store"` *Virtuoso Open Source 7 is a triple store.*
OntoWiki	`"present"` *OntoWiki is for authoring (implies presentation).*
`org.aksw.dld.subtype`	
VOS 7	`"vos7"` *This store is specifically Virtuoso Open Source 7.*
`org.aksw.dld.component-constraints`	
VOS 7	`{"load": "vos"}` *VOS should be filled with suitable load component using it's bulk laoding facilities.*
Ontowiki	`{"store": "vos[67]"}` *OntoWiki is only compatible with recent VOS versions (value interpreted as regular expression).*
`org.aksw.dld.volumes-from`	
VOS load	`["store"]` *VOS load needs to copy data into the filesystem of the store, thus mount volumes from the store to obtain a shared part of the store filesystem.*

Table 1: Examples for the usage of *Docker* labels for meta-information on component images for Virtuoso Open Source (VOS) and OntoWiki

[15]The original SNORQL source code: http://d2rq-map.cvs.sourceforge.net/viewvc/d2rq-map/d2r-server/webapp/snorql/ and the currently active source code fork at GitHub: https://github.com/kurtjx/SNORQL

[16]OntoWiki Site Extension at GithHub: https://github.com/AKSW/site.ontowiki

[17]Labels on images were introduced in *Docker* 1.6: https://docs.docker.com/userguide/labels-custom-metadata/

[18]A `PASSWORD` environment variable from the store e.g. is provided as `STORE_ENV_PASSWORD` in the linking load container.

3.5 Provisioning of a Container Setup

The requirements for the presented architecture are the Dockerizing Linked Data (DLD) tools as described at our project webpage (http://dld.aksw.org), including a running *Docker* engine installation as well as the *Docker Compose*[19] tool.

```
datasets:
    dbpedia-2015-endpoint:
        graph_name: "http://dbpedia.org"
        location_list: "dbp-2015-download.¬
          list"

components:
    store:
        image: aksw/dld-store-virtuoso7
        ports: ["8891:8890"]
        environment:
            PWDDBA: tercesrepus
        mem_limit: 8G

    load: aksw/dld-load-virtuoso
    present:
        ontowiki:
            image: aksw/dld-present-¬
              ontowiki
            ports: ["8088:80"]

environment_global:
    DEFAULT_GRAPH: "http://dbpedia.org"
    SPARQL_ENDPOINT_URL: "http://store¬
      :8890/sparql"
```

Listing 1: A DLD configuration file for a setup with the Virtuoso Open Source triple store, a suitable load component and OntoWiki for presentation/edits, retrieving DBpedia data via downloads listed in a separate file

	in-lined in config file	listed in external file
local file(s)	`file:`	`file_list:`
download(s)	`location:`	`location_list`

Table 2: Options for specifiying RDF data dumps to be imported

Listing 1 shows an example of a DLD configuration file. In addition to specifications for desired components, data to be served by the store can be specified in the `datasets`-section. In this case, a separate file is referenced that lists URLs of dataset dumps to be retrieved and loaded (Table 2 also enumerates other alternative directives for specifying RDF data to import). The DLD configuration tool, provided with this configuration, will perform several tasks to prepare the setup:

1. A working directory is created that will contain the compiled *Compose* configuration and (if specified) data dumps to load.
2. Referenced images will be pulled.
3. The DLD-specific `LABEL` meta-data of the images will be extracted, checks for declared and implicit setup constraints will be performed, and entailed additional *Compose* configuration items are incorporated to the final setup. E. g. the image for the *store* component must carry the `org.aksw.dld.type` label with the value `store`.
4. Specified Linked Data dumps are downloaded and aggregated into the `models` directory inside the working directory, augmented with files specifying the named graph URIs according to the conventions of the *Virtuosos* bulk loader[20].

The procedure of creating, fetching and configuring the individual components of a container setup is depicted in fig. 2.

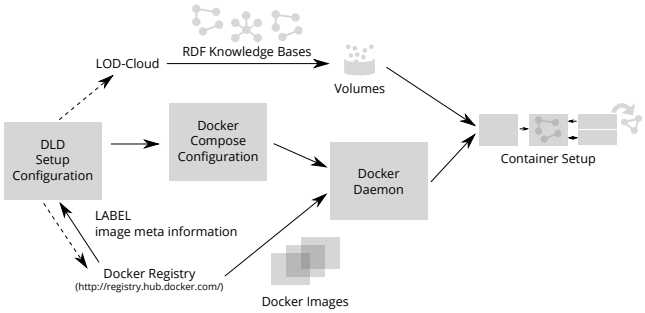

Figure 2: Workflow for creating a container setup

```
load:
    environment: {DEFAULT_GRAPH: 'http://¬
      dbpedia.org', SPARQL_ENDPOINT_URL: '¬
      http://store:8890/sparql'}
    image: aksw/dld-load-virtuoso
    links: [store]
    volumes: ['/opt/dld/wd-dbp2015-ontowiki¬
      /models:/import']
    volumes_from: [store]
presentontowiki:
    environment: {DEFAULT_GRAPH: 'http://¬
      dbpedia.org', SPARQL_ENDPOINT_URL: '¬
      http://store:8890/sparql'}
    image: aksw/dld-present-ontowiki
    links: [store]
    ports: ['8088:80']
store:
    environment: {MEM_LIMIT: '8G', ¬
      DEFAULT_GRAPH: 'http://dbpedia.org', ¬
      SPARQL_ENDPOINT_URL: 'http://store¬
      :8890/sparql', PWDDBA: 'tercesrepus'}
    image: aksw/dld-store-virtuoso7
    mem_limit: 8G
    ports: ['8891:8890']
```

Listing 2: *Compose* configuration file compiled by the DLD config tool based on the configuration in listing 1

[19]Installation instructions for *Docker Compose*: http://docs.docker.com/compose/install/

[20]Description of the *Virtuosos* bulk loader: http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtBulkRDFLoader

Listing 2 shows a resulting *Compose* configuration, which can either be submitted directly by the configuration tools to *Compose* to start the configured services or checked and revised by the user before a subsequent manual start by invoking `docker-compose up`. The apparent close correspondences between fragments of the DLD configuration and the resulting *Compose* is intended as this lowers the entry barrier for new users of DLD harbouring prior experience with *Compose* and allows to develop an intuition how the final configuration is created more easily. In the example presented, the user can (after completion of initialisation and import processes) reach OntoWiki at port 8088 and the SPARQL interface of Virtuoso at port 8891 of the host system.

Although not presented in detail in this paper, it should be mentioned that setups without load and backup components are possible as well. A prime example is the usage of a store component that links in a volume with database files that contain a selection of RDF datasets already pre-loaded.

4. USE CASES

We have evaluated this architecture in three use cases. The best-known scenario we investigate is setting up a local mirror of the *DBpedia* dataset. As the availability of the DBpedia SPARQL endpoint is limited and not all DBpedia data is served by it, running a mirror is the only way to effectively access all knowledge it contains. Our other use cases are the prosopographical knowledge bases *Catalogus Professorum Lipsiensium* and *Sächsisches Pfarrerbuch*. These knowledge bases, used by digital humanities researchers, are exemplary for many smaller knowledge modelling and representation projects throughout diverse domains, that do not have the resources necessary to employ Linked Data experts. These projects can benefit significantly from an easy to deploy Linked Data publication infrastructure.

4.1 DBpedia

DBpedia[21] [9] is one of the most widely used Linked Data sources on the Web of Data, a large scale knowledge base reflecting content and structure of 125 Wikipedia language editions, Wikimedia Commons and Wikidata. Major longstanding contributions of the project include a general knowledge OWL ontology for all kinds of entities described in Wikipedia and thus being reflected in DBpedia and a framework for extracting machine-actionable fact statements from Wikipedia info boxes (guided by community curated mapping definitions) as well as additional structural features of Wikipedia pages. Although the English version of DBpedia is most widely used and was designated with special canonicalisation status, the facts are extracted analogously in internationalised DBpedia versions for 124 other Wikipedia editions as well. The DBpedia project provides supplementary datasets for each language version containing structural information (e. g. redirecting structures between the Wikipedia pages, Wikipedia categories or disambiguation links) and dataset aggregation text data for NLP purposes.

DBpedia publishes major releases of holistic extraction results based on complete Wikipedia dumps at least once a year. The big volume of information to be found in the totality of a major release called for a selective decision for a subset that can still be catered by the official DBpedia SPARQL endpoints with some performance guarantees, based on frequency of requests for certain types of facts by users of the official endpoint and Linked Data services. A user of DBpedia with the intent of achieving higher availability and better performance for their queries with a local endpoint thus is facing the non-trivial and onerous task of finding the applicable subset out of hundreds of RDF dump files provided by the DBpedia download server, depending on language, release version and relevant information categories to achieve. To mitigate this problem, ready-to-use descriptions of relevant DBpedia data compilations will be provided in the context of this project to be automatically consumed and loaded into the provided triple store components, harnessing information from DataID.

DataID [3] descriptions provide machine-readable dataset level meta-data and can thus help increase the discoverability of datasets. Because DataID descriptions mandatorily contain direct links to data files used to distribute the datasets as well as additional information about these files, such as file format and modification dates, they provide a convenient way to select the files to be loaded into the *Docker* containers. At the same time, DataID can be used to ship the container setup configurations, allowing to distribute deployment meta-data with the data itself. We will further explain this future work in section 6.

4.2 Catalogus Professorum Lipsiensium

The *Catalogus Professorum Lipsiensium* (CPL) [11] is a prosopographical knowledge base of professors who have taught at Leipzig University from its foundation in 1409 to the presence. It comprises more than 14,000 entities and is tightly interlinked with other nodes in the LOD cloud. The knowledge base is curated by researchers as well as historically interested citizen scientists. The container-based infrastructure is used to run the CPL curation infrastructure which is constantly improved by software engineers to meet the requirements of the contributors and editors. Since the first release of CPL the software stack was improved and other universities are as well interested in reusing the software system setup.

The CPL infrastructure consists of several web applications that provide specific adapted interfaces for domain users – the project team members (Content Editors), experienced users (Researchers) and general web users. Figure 3 depicts the architecture of CPL that is based on a protected web interface for the project team and two public interfaces. According to the specific interfaces, CPL is build on a stable OntoWiki Framework application, that provides precise authoring and visioning information, and an up-to-date experimental version of OntoWiki to provide latest exploration features for researchers that consume data from the knowledge base. The experiential OntoWiki version is also using inferred information and is linked to other datasets on the LOD cloud. The third component is to provide historical information about Leipzig University integrated in the university's website.

The CPL *Docker* infrastructure[22] provides a setup that supports software engineers in replacing and updating components, such as the RDF editing forms and HTML output without needing to touch the knowledge base store. We can

[21] The DBpedia project webpage: http://dbpedia.org/

[22] *Docker* infrastructure of the Catalogus Professorum Lipsiensium at GitHub: https://github.com/AKSW/dockerinfrastrukturecpl

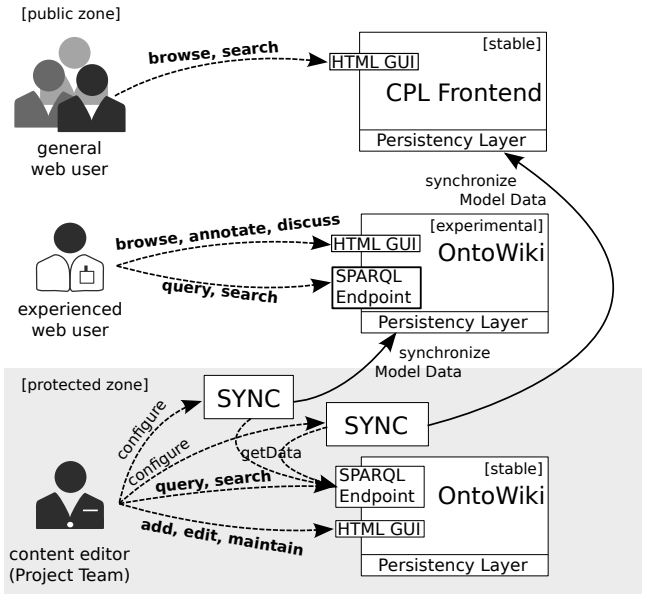

Figure 3: Architecture of CPL (cf. [11])

also easily deploy the complete setup with a new knowledge base for further tenants without a need to adapt it to the given server environment, while all required components can be reused independently.

4.3 Sächsisches Pfarrerbuch

The project "Sächsisches Pfarrerbuch" (engl. *Saxonian pastors book*)[23] is a catalogue of all pastors serving in the Lutheran church in Saxony since the reformation in 1517. Currently the dataset is under curation and only a very small excerpt of the uncurated data is published. All other data will be released as Linked Open Data after the curation phase. Due to the similarity in the domains, the setup of the software system is highly inspired by the Catalogus Professorum Lipsiensium, while the audience is different. By using the approach of the containerised Linked Data publication infrastructure we can reuse most of the containers among the two projects while still loading different datasets to the triple stores.

5. RELATED WORK

Linked Data [1] publishing has mainly been tackled in foundational works. Heath et al. [8] stresses the importance of publishing Linked Data according to its criteria and explains basic publishing recipes, such as hosting RDF/XML files or serving them via custom server-side scripts or, ideally, using a triple store. However, the technical deployment issues are not stressed further.

On the subject of *Docker* containers, Merkel [10] details that *Docker* containers are a lightweight solution to resolve dependency and security problems as well as platform differences. *Docker* containers have also been explored to enhance reproducible research (Chamberlain et al. [4] and Boettiger [2]). Chamberlain et al. [4] detail that software and experimental setups in research are often not well-documented enough to provide exact information about the systems they were executed on, hindering reproducibility of experiments. Computational environments are very complex and containerisation provides a way to isolate experimental setups from some external variables of the systems they are executed on. Boettiger [2] further states that many existing solutions, such as virtual machines, workflow software and continuous integration services provide significant barriers to adoption by being hard to realise and often not sufficiently low-level to solve the mentioned problems of dependencies and documentation. Both works therefore use *Docker* as local development environments for reproducible scientific research.

The web service Dydra (http://dydra.com/) is a graph database hosting service on the World Wide Web. It offers to the user the possibility to create graphs, load RDF data and query it using SPARQL. Currently it is in its beta period. In contrast to our approach the data can not be hosted on a local machine and thus will not give the user control over the availability of the services or the data. Further a local service has the advantage of full access-control, while for a cloud hosting service it is still in doubt whether it is suitable for sensible data, even though the service provides an authentication mechanism.

6. DISCUSSION AND CONCLUSION

We have presented a methodology and procedure to significantly ease knowledge base deployment and maintenance by using *Docker* containers and the principle of micro services. Working with Linked Data one often encounters knowledge bases that face regular downtimes, significant load or completely lack Linked Data publication. The approach together with current proof of concept implementations for pre-configured knowledge base exploration and authoring setups shows several desirable properties.

The same setup can be reused for different knowledge bases or under recurring requirements to the software setup, while only the data to load has to be exchanged. On the other hand with the presented modular architecture it is also possible to exchange individual components without major dependencies to other components. One can select different triple store implementations with different advantages and disadvantages in respect to the data to be published and the associated use case.

Portability of knowledge base setups is improved by transferring a customised *DLD* specification and optionally the data to load to collaborators. Even though the collaborators might be using different platforms, the presented tool allows them to easily setup a mirror which significantly reduces collaboration overhead. Shipping knowledge bases together with the deployment configuration that was used during experiments also increases the reproducibility of research by eliminating errors in the data setup.

Large, well established and popular Linked Data projects like DBpedia can benefit also indirectly from the establishment of the presented approach or a similar scheme. It can incentivise a share of users to reduce load on public endpoints by choosing to setup their private mirror which entails further benefits, like stability and faster access. As a result the easy distributed deployment of mirrors can reduce the load on a central infrastructure.

[23]The Pfarrerbuch project webpage: http://pfarrerbuch.de and the current curation system: http://pfarrerbuch.aksw.org

We have also shown that small knowledge bases without large community backing can gain from dockerised data deployment. In the future, we want to further improve data deployment by providing a visual front-end for configuration. We also want to further concentrate on reproducibility by enabling data deployment *Docker* recipes to be shipped via DataID. By adding essential configuration properties in a DataID ontology module, dataset descriptions using DataID will be able to provide information on how to set up the data locally. This will make it possible to ship machine readable descriptions of the complete data backend of experiments and further improving reproducibility. Introducing a common light-weight network message bus infrastructure integrated in all provided images to allow for more detailed queries of configuration and environment parameters for tighter integration as well as status requests appears worthwhile.

7. ACKNOWLEDGEMENTS

We want to thank the students from Business Information Systems practical in summer semester 2015, especially we want to thank the students Georges Alkhouri and Tom Neumann for helping us with the preparation and implementations of the *Docker* infrastructure. We also want to thank Konrad Abicht for providing the *semicon* icons (https://github.com/k00ni/semicon) under the terms of CC BY-SA 3.0 used in fig. 2 and fig. 3. This work was supported by grants from the EU's 7th Framework Programme provided for the projects LIDER (GA no. 610782) and GeoKnow (GA no. 318159).

8. REFERENCES

[1] T. Berners-Lee. Linked Data. Design issues, W3C, June 2009. http://www.w3.org/DesignIssues/LinkedData.html.

[2] C. Boettiger. An introduction to docker for reproducible research, with examples from the R environment. *CoRR*, abs/1410.0846, 2014.

[3] M. Brümmer, C. Baron, I. Ermilov, M. Freudenberg, D. Kontokostas, and S. Hellmann. DataID: Towards semantically rich metadata for complex datasets. In *Proceedings of the 10th International Conference on Semantic Systems*, SEM '14, pages 84–91. ACM, 2014.

[4] R. Chamberlain and J. Schommer. Using Docker to Support Reproducible Research. 07 2014.

[5] R. Cyganiak and C. Bizer. Pubby - a linked data frontend for sparql endpoints, 2008.

[6] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. pages 171–172, 2015.

[7] P. Frischmuth, M. Martin, S. Tramp, T. Riechert, and S. Auer. OntoWiki—An Authoring, Publication and Visualization Interface for the Data Web. *Semantic Web Journal*, 2014.

[8] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 2011.

[9] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web Journal*, 6(2):167–195, 2015.

[10] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), Mar. 2014.

[11] T. Riechert, U. Morgenstern, S. Auer, S. Tramp, and M. Martin. Knowledge engineering for historians on the example of the catalogus professorum lipsiensis. In P. F. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J. Z. Pan, I. Horrocks, and B. Glimm, editors, *Proceedings of the 9th International Semantic Web Conference (ISWC2010)*, volume 6497 of *Lecture Notes in Computer Science*, pages 225–240, Shanghai / China, 2010. Springer.

[12] M. J. Scheepers. Virtualization and containerization of application infrastructure: A comparison. 2014.

[13] M. Schmachtenberg, C. Bizer, and H. Paulheim. Adoption of the linked data best practices in different topical domains. In *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, pages 245–260, 2014.

[14] The W3C SPARQL Working Group. SPARQL 1.1 Overview. Technical report, Mar. 2013. http://www.w3.org/TR/sparql11-overview/.