Introducing Kosh, a Framework for Creating and Maintaining APIs for Lexical Data

Francisco Mondaca[1], Philip Schildkamp[2], Felix Rau[2]

[1] Cologne Center for eHumanities, University of Cologne

[2] Data Center for the Humanities, University of Cologne

E-mail: f.mondaca@uni-koeln.de, philip.schildkamp@uni-koeln.de, f.rau@uni-koeln.de

Abstract

In recent years, the use of application programming interfaces (APIs) throughout the Internet has increased significantly. The main reason for this growth is the multiplicity of scenarios where APIs can be employed. In the case of APIs for lexical data, their use varies from applications for mobile devices, desktop applications to natural language processing (NLP) applications, among others. While some publishers offer their data via APIs, for most small or medium size publishers implementing and providing an API is still an obstacle due to the costs and technical expertise required for their development and maintenance. Against this background, we have developed Kosh, an open-source framework for creating and maintaining APIs for lexical data. Kosh has been conceived to minimize the technical expertise required for its use, while offering generic, flexible and efficient data management. In this article, we present the methodology employed in Kosh's development and describe its architecture and functionality.

Keywords: API; Elasticsearch; framework; GraphQL; REST

1. Introduction

The development of digital lexicography over the past decades has been focused on the production of lexical data, either by digitizing printed works or by creating born-digital lexical data from scratch. Therefore, software production within this field of expertise has been directed towards the development of tools for compiling lexical data. Lexical data access has been confined mainly to the development of web applications, which are the heirs of printed dictionaries. The emergence of NLP applications and mobile devices, among other use cases, has increased the necessity to focus on the development of efficient ways of accessing lexical data. APIs satisfy this requirement, as a single API instance can provide data for multiple applications at the same time.

Although the use of APIs seems to ease several aspects of data access, there are no software solutions focused on API development and maintenance. While it is possible for large publishers to develop their own APIs, the main problem faced by small or medium sized publishers is the absence of technical expertise in-house and expensive external solutions. Against this background we created an easy-to-use framework to serve lexical data via APIs in order to lower this technical and financial hurdle.

The structure of this article is as follows: In Section 2 the motivation and decisions made about data format, data control, data persistence and efficient data access are explained. In Section 3 the architecture and functioning of Kosh are described. Section 4 concludes with a summary of the presented work and future development of the framework. Referenced publications are listed in Section 5.

2. Development Methodology

2.1 Background and motivation

Kosh has been conceived to provide API access to any XML[1]-encoded lexical dataset, and its name *Kosh* derives from the Hindi word for dictionary or lexicon, कोश *koś* or *kosh*, which in turn derives from Sanskrit कोश *kośa* with the same meaning. Kosh's origin is related to multilateral API development for TEI[2]-encoded Sanskrit dictionaries at the University of Cologne, where the most noted digital collection of Sanskrit dictionaries worldwide is hosted.

Using the Cologne Digital Sanskrit Dictionaries web portal[3], users can query all of the 37 dictionaries available through various web applications and even download each dataset in XML format. The underlying digitization project started in 1996, when XML and Unicode were not available, while in 2003 the dictionaries had been converted into XML. During the Lazarus project[4] (2013-2015) three dictionaries were encoded in TEI-P5[5], among them the two with the most complex structure of the entire collection (Böhtlingk & Roth, 1855-1875; Monier-Williams, 1899). Those were chosen to develop a custom schema[6] to be employed for all future TEI-P5 dictionaries in the collection.

The first iteration of Kosh were the C-SALT APIs for Sanskrit Dictionaries (Mondaca, 2018), a proof-of-concept developed within the context of the currently running VedaWeb project[7]. One of this project's most important outcomes is to link each word of the Rigveda, the oldest text of the Indo-Aryan language family, to a dictionary specifically compiled for this text (Grassman, 1873). And in order to provide API access to this TEI-P5-encoded dictionary to the VedaWeb web application and other possible use cases, the C-SALT APIs for Sanskrit Dictionaries were implemented and have been

[1] Extensible Markup Language, https://www.w3.org/XML

[2] Text Encoding Initiative, https://tei-c.org

[3] Cologne Digital Sanskrit Dictionaries, https://www.sanskrit-lexicon.uni-koeln.de

[4] Cologne Center for eHumanities, Lazarus project, https://cceh.uni-koeln.de/lazarus

[5] Text Encoding Initiative, P5 Encoding Guidelines, https://tei-c.org/guidelines/P5

[6] C-SALT Dictionary Schema, https://github.com/cceh/c-salt_dicts_schema

[7] VedaWeb, https://vedaweb.uni-koeln.de

transformed into a data module[8] served by Kosh.

The guiding principle of both iterations is and has been to provide efficient access to the underlying lexical data through means of open-source software. But unlike the first iteration, the C-SALT APIs for Sanskrit Dictionaries, which were hard-coded to only serve their one designated dataset, Kosh is a *generic* solution for XML-encoded dictionaries, i.e. how each dictionary is structured is not relevant, and any XML-encoded dictionary can be indexed and accessed through Kosh's APIs.

2.2 Modular rather than monolithic

The early-stage development of Kosh consisted partly of researching software with similar features, and we noticed a lack of tools that focus on providing API access to lexical data. Most of the dictionary writing systems (DWS), commercial as well as open-source, are focused on compiling lexical data, but bear no means of providing API access to the generated data. This is reflected in a recent survey among lexicographers (Kallas et al., 2019: 33), asking respondents to identify their wishes or needs to be solved in the next 10-15 years; API access was one of the mentioned topics.

An exception in this respect is the open-source DWS Jibiki, which provides access to lexical data contained within the platform through an API (Mangeot & Enguehard, 2018: 29). But to use this API, its clients must previously register with the system. While for many publishers this might be a desired feature, as it gives them an extra layer of control and is integrated into the DWS, we opted for a different approach to Kosh's software architecture: Modularity.

When following a modular approach to software development, resolving errors or scaling up/down specific aspects of a system is usually less complex than in the case of monolithic applications, the prime architecture in traditional software development. For example, if an API module exhibits undefined behaviour (an error), this should not affect or propagate to the whole DWS, but should be contained within the erroneous module. This is one reason why the microservices architecture, essentially modular, has reached such a high level of popularity throughout the software industry.

The task of a DWS should be focused on creating and compiling new lexical data and if required accessing external sources via standardized APIs. As is the case with Lexonomy[9], a cloud-based DWS that can access data from Sketch Engine[10], a corpus manager tool, via an API. When keeping it modular, lexical data generated with this or another DWS is published by a different software component than the DWS itself, such as Kosh.

[8] C-SALT APIs for Sanskrit Dictionaries, https://cceh.github.io/c-salt_sanskrit_data

[9] Lexonomy, https://www.lexonomy.eu

[10] Sketch Engine, https://www.sketchengine.eu

2.3 Input Data Format

In order to keep the complexity of Kosh as minimal as possible, we decided to support only the most common serialization format in lexicography: XML. At scholarly level, the use of XML-based models such as the TEI is well-known, especially in the digitization of printed dictionaries. DWS such as TLex Suite[11] or the Dictionary Production System[12] also output XML data. Other popular formats employed in dictionary compilation such as Toolbox[13], prevalent in language documentation, can be transformed into XML with open access tools[14], as is also the case for most of other formats such as JSON[15] or YAML[16]. XML is widely used for representing dictionaries modelled as trees, but it is also employed to serialize graph-based models such as RDF[17], although other serialization formats for graph-based models such as Turtle[18] or JSON-LD[19] have gained more popularity.

Kosh can handle any XML-encoded lexical dataset. We believe that developing a generic framework for APIs means that the framework should be agnostic towards the structure of the dictionaries involved: Searchable fields vary between dictionaries and they have to be defined by the publisher. Kosh can handle all types of structures as long as they are serialized in XML: Graphs, trees or graph-augmented trees, a tree-like structure that allows elements to have more than one parent (Měchura, 2016, 2018). The only limitation of our generic approach is the requirement to specify only one single XPath expression to represent an entry of the respective dictionary.

When indexing RDF datasets with Kosh, the problem that arises is to choose which nodes will be indexed, as lexical data is normally to be found in different nodes, unlike in tree-based models. If data has been encoded in OntoLex-Lemon (McCrae et al., 2017), one of the most employed graph-based models for lexical data, we would ideally index a top level node such as `LexicalEntry`. The problem then is that most of the lexicographic information such as forms and senses is normally to be found in other nodes i.e. `Form` or `LexicalSense`. So, in this case, we would need three indexes to access these nodes. For indexing the English WordNet[20], only two indexes are required

[11] TLex Suite, https://tshwanedje.com/tshwanelex

[12] Dictionary Production System, http://dps.cw.idm.fr

[13] Toolbox, https://software.sil.org/toolbox

[14] Natural Language Toolkit, Toolbox Reader, https://www.nltk.org/_modules/nltk/toolbox.html

[15] JavaScript Object Notation, https://www.json.org

[16] YAML Ain't Markup Language, https://yaml.org

[17] Resource Description Framework, https://www.w3.org/RDF

[18] Turtle, https://www.w3.org/TR/turtle

[19] JSON for Linking Data, https://json-ld.org

[20] English WordNet, https://en-word.net

for the types of nodes available: `LexicalEntry` and `Synset`[21].

2.4 Simply generic

As mentioned above, our starting point, conceptually and technically, was the C-SALT APIs for Sanskrit Dictionaries. Therefore, decisions such as which web framework, which search engine and which API paradigms to use were already made. The main issue we had to tackle was to conceive a generic method for any XML-encoded dictionary to be parsed and indexed. For this purpose, we set two goals: i) Make the configuration of this process as human-friendly as possible, and ii) from a software development perspective as elegant as possible.

Another question was: Which notation system should be used to determine the location of the nodes to be indexed? As we are parsing XML files, a rational alternative was to choose XPath[22], a query language designed for selecting nodes in an XML document. As Kosh relies on lxml[23], a library for manipulating XML documents, which supports XPath 1.0 but not XPath 2.0, all XPath notations must conform to XPath 1.0.

Regarding the human interaction required to configure Kosh, one must specify which nodes of which XML documents contain lexical entries and which subnodes contain fields to be indexed. Elasticsearch[24] indices can be configured by external JSON files (see Section 3.2); such a file is used by Elasticsearch to setup an index and its fields with their respective data types, which are specified under the property `properties`. Following this pattern, we employ the `_meta` property to store Kosh-specific data without integrating it with the respective Elasticsearch index. In conclusion, by enriching the standard Elasticsearch JSON index definition with all required Kosh-specific data, we are able to drastically minimize human configuration effort.

2.5 Searching lexical data

A crucial decision in developing Kosh has been to employ a search engine, Elasticsearch, instead of a database, relational or not, for searching through and retrieving lexical data. We abstained from using a database management system (DBMS) with a mounted search engine on top of it as our primary data storage, as this solution seemed to add a level of complexity that is too cumbersome for a framework that should deal with different datasets and update them automatically when modified. The central question here is, why would a database be useful for this purpose?

[21] English WordNet Kosh data module, https://github.com/cceh/kosh_data/tree/master/wordnet_en

[22] XML Path Language, https://www.w3.org/TR/1999/REC-xpath-19991116

[23] lxml, XML and HTML with Python, https://lxml.de

[24] Elasticsearch, https://www.elastic.co/products/elasticsearch

Databases were conceived and are employed for storing and managing data. Some of them (e.g. PostgreSQL[25]) allow full-text searches, and most of the search scenarios required by the average dictionary consumer might be covered by this functionality, but DBMS in general are not tailored to automatically hash fields to minimize response latencies nor to provide different means of fuzzy query logic as search engines are. Search engines are thus the best performing systems, and Elasticsearch is one of the most used and best documented search engines servers available, so we chose to employ it.

2.6 Tracking data changes

An ideal scenario to collaboratively edit dictionaries and track changes would be to place all the datasets on a git[26] repository. One of the main features of git is versioning, and if the modules are on a cloud repository then all users with access can track changes and contribute. This aspect is particularly useful if a dataset contains errors or is open to modifications, and as dictionaries are continuously being edited and extended, versioning is a major improvement in their compilation process.

While not being part of Kosh's core, any publisher using Kosh can easily setup data synchronization pipelines by e.g. hooking into GitHub events[27], and as soon as Kosh notices the changes being propagated to its local data modules (i.e. filesystem watches are triggered), the respective search indexes get updated.

2.7 Choosing API paradigms

Authors like Tarp (2015: 34) have pointed out that one of the central features of a dictionary is to retrieve information in an easy and efficient way. Since we second this perspective, Kosh provides access to indexed lexical data not only via a single API paradigm, but the two most popular among the request-response APIs: REST (Fielding, 2000) and GraphQL (Shevat et al., 2018: 224). Besides these two main API paradigms, there are some less-employed technologies available, e.g. XQuery[28], which we thought of implementing but refrained from at this early stage of development.

REST has been the most popular API paradigm in the last decade, but GraphQL has risen in popularity considerably during the last few years. The reduced data load that GraphQL offers towards mobile applications is an attractive factor for its implementation in such environments (see Section 3.4). And as our goal is to satisfy as

[25] PostgreSQL, https://www.postgresql.org

[26] git Source Control Management, https://git-scm.com

[27] GitHub Developer Guide, https://developer.github.com/webhooks

[28] XML Query Language, https://www.w3.org/TR/xquery-31

many consuming and publishing use cases as possible with this framework, serving endpoints for both APIs per dataset offers the highest compatibility and therefore coverage.

While Kosh's lexical input data has to be in XML format, both APIs return data in JSON format. The reason for this decision lies in the fact that parsing JSON is less cumbersome than parsing XML. This statement might be misleading, as Kosh by default indexes the whole entry in XML format, independently of the searchable fields defined by the publisher. If the client needs information that is not available through these fields, it must parse the full XML entry returned by Kosh's APIs.

2.8 Open-Source Licensing

Kosh is an open-source framework and relies extensively and exclusively on open-source technologies. It runs natively on Unix-based systems, in particular Linux (Torvalds, 1997), the operating system prevalent in server environments. Elasticsearch, the search engine server, is also open-source, as is Python, the programming language that Kosh is written in. Both API paradigms offered by Kosh, REST and GraphQL, are also open-source, as is Docker[29], which may be used to deploy Kosh (see Section 3.5). In terms of licensing, Kosh is available under the MIT Licence[30].

3. Architecture and Functioning

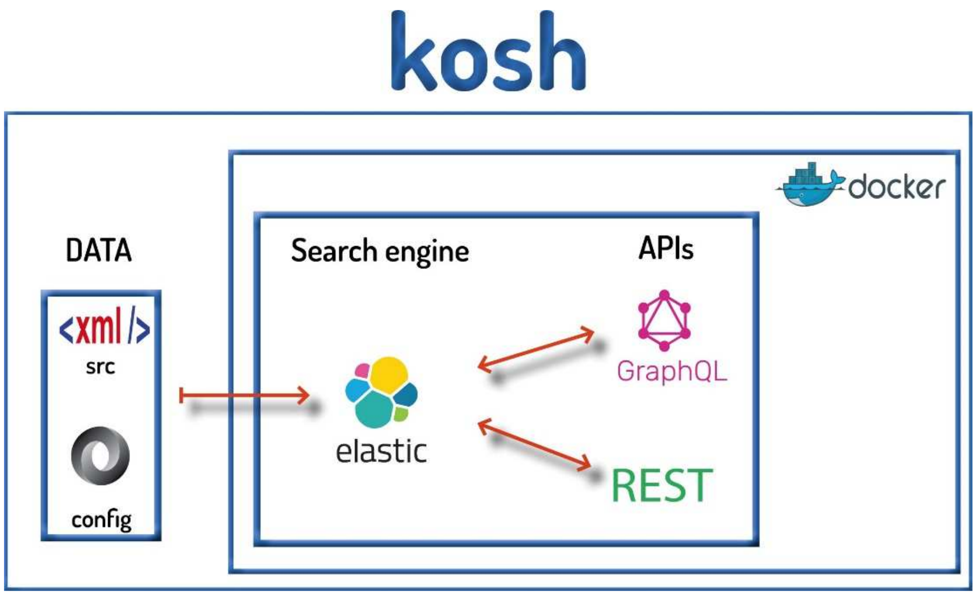


Figure 1: Overview of Kosh's Architecture.

[29] Docker, https://docs.docker.com

[30] MIT License, https://opensource.org/licenses/MIT

3.1 Overview

Kosh's core relies on the search and analytics engine Elasticsearch and access to data indexed by this search engine is provided by GraphQL and REST APIs. While currently only two API paradigms are implemented, Kosh's application structure is designed to be modular, wherefore implementing new API paradigms to provide access to the underlying lexical data is part of our vision. A Kosh data module (input data) consists of:

1. A dataset in XML format containing lexical data
2. A JSON file containing information about the elements and their data types to be extracted from the XML file(s) in XPath 1.0 notation. This information is used by the XML parser, by Elasticsearch and by the API components
3. A kosh dotfile (.kosh) providing the following information:
   - The data module(s) name(s)
   - Filesystem path(s) to the XML file(s) containing lexical data (see 1.)
   - Filesystem path(s) to the aforementioned JSON file(s) (see 2.)

Kosh is written in Python and can be deployed in Unix-based systems. XML parsing is done with lxml, the library elasticsearch-dsl[31] is employed for communicating with Elasticsearch, and Flask[32] is used as Kosh's web application framework. Kosh's core can be downloaded as a Docker image from Docker Hub[33] or accessed directly on GitHub[34].

3.2 Data and metadata

Kosh processes lexical data in XML format and datasets might be split into multiple files (see e.g. de_alcedo[35]). Further, a single Kosh instance can serve multiple data modules, while each data module is accessible through its own API endpoints. But Kosh's main innovation lies in the possibility to define the searchable fields, their respective data types and thus the perspective on each individual dataset. The only constraint is that for each index only one top-level node, i.e. entry, is allowed, but it is possible to create multiple indexes for a single XML file (see Section 2.3).

[31] Elasticsearch DSL, https://elasticsearch-dsl.readthedocs.io

[32] Flask, http://flask.pocoo.org

[33] Kosh Docker image, https://hub.docker.com/r/cceh/kosh

[34] Kosh GitHub repository, https://github.com/cceh/kosh

[35] De Alcedo Kosh data module, https://github.com/cceh/kosh_data/tree/master/de_alcedo

A lexical entry may contain different substructures, e.g. headword, part-of-speech (PoS), senses, etc., but Kosh is agnostic in this respect. The only information required for parsing and indexing a lexical entry is its XPath within the XML file(s). If no further fields (and their XPaths), e.g. headword or PoS, are specified, users can search in the whole entry but not in specific substructures, as the whole entry is indexed per default and analysed without its markup. This might be relevant for some use cases, especially when a dataset cannot be encoded in a more fine-grained manner.

```
{
  "mappings": {
    "_meta": {
      "_xpaths": {
        "id": "./@id",
        "root": "//entry",
        "fields": {
          "lemma": "./form/orth",
          "[sense_def]": "./sense/def",
          "[sense_pos]": "./sense/gramGrp/pos/q",
          "[dicteg]": "./sense/dicteg/q"
        }
      }
    },
    "properties": {
      "lemma": {
        "type": "keyword"
      },
      "sense_def": {
        "type": "text"
      },
      "sense_pos": {
        "type": "text"
      },
      "dicteg": {
        "type": "text"
      }
    }
  }
}
```

Figure 2: JSON configuration file for the Basque dictionary *Hiztegi Batu Oinarriduna*[36](HBO)

The JSON file seen in Figure 2 is used to configure the underlying Elasticsearch index. Relevant for Elasticsearch is the `mappings` property. It must contain the `properties` key, which specifies the fields to be indexed and their respective data types. For handling strings, the data types `keyword` and `text` may be chosen. The difference between them is that the latter is analysed by the standard analyser, which tokenizes the input string based on the Unicode Text Segmentation algorithm, while the former does not analyse or modify the input string. This should be taken into consideration when indexing headwords, because if they are indexed as `text` the analyser converts the input strings to lowercase and splits them if they contain spaces. In some cases this could render exact matches (`term` queries in Elasticsearch terminology) impossible.

Kosh-specific configuration values, e.g. information relevant for XML parsing, are

[36] HBO Kosh data module, https://github.com/cceh/kosh_data/tree/master/hiztegibatua

stored within the `_meta` property. It contains the XPath to lexical entries within the mandatory property `_xpaths.root` and any additional fields to be extracted within the property `_xpaths.fields`. Usually every lexical entry in parsed XML files contains a unique ID, which is also required by Kosh. This applies to the dataset as seen in Figure 2, but some datasets might not contain unique entry IDs (see e.g. freedicts[37]). In such cases, Kosh generates IDs by calculating SHA1 hashes from a normalized form of each entry, so those IDs only change when the respective entry changes and therefore are reproducible.

Data modules are identified by Kosh through the existence of a `.kosh` dotfile. Such a `.kosh` file acts as an entry point for the data module by specifying its names, file system paths to XML files that contain the lexical data to be indexed, and to the JSON metadata files containing the previously described definitions for the respective data module.

Finally, when running Kosh on an operating system capable of notifying[38] file changes, Kosh automatically updates the respective Elasticsearch index and re-binds all API components to reflect changes made to the data module definitions or its lexical data.

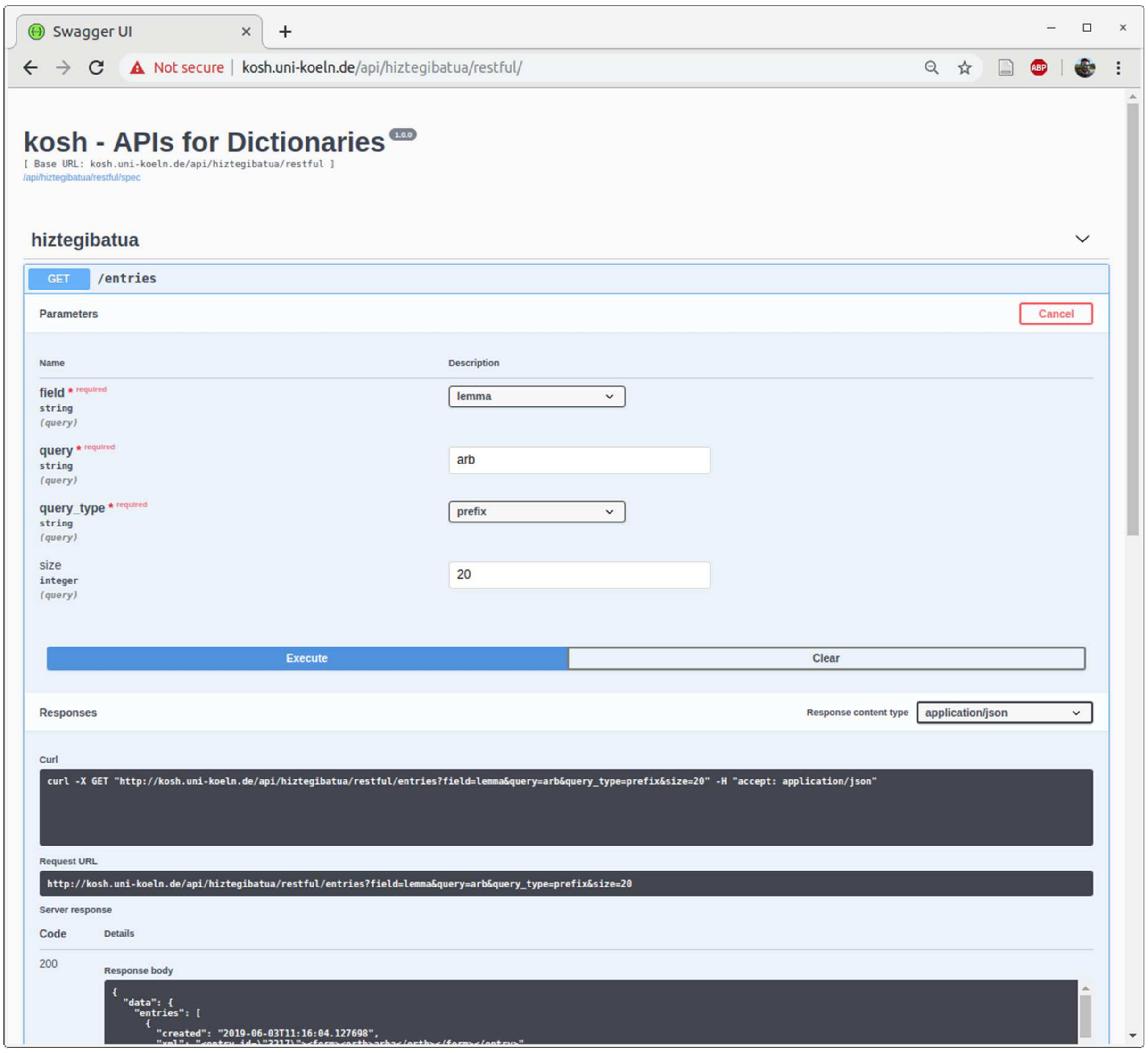
3.3 Elasticsearch engine

Kosh employs Elasticsearch as its search engine server. Currently, the supported query types are those available for unique fields with the properties `keyword` or `text` in the configuration file, e.g. `prefix`, `term` and `match`. Query types on multiple fields, e.g. `multimatch` and `bool`, have not yet been implemented but are being actively developed. String based queries can be classified as full-text or term-level, and clients can perform both types of queries on all indexed fields. Queries might return different results when using a `term` query (exact matching), if the queried field has been indexed as `text` instead of `keyword`, because `text` fields are analysed, i.e. they are tokenized and lowercased. For example, if a dictionary has uppercased lemmas which have been indexed as `text`, any uppercased term-level query on the respective field will not deliver results.

By default, Elasticsearch (and thus Kosh) returns ten elements per query, but a client can request more results by providing a specific integer value in the `size` query field. Further, Kosh's default configuration offers two term-level query types that expose all the indexed entries at once: `regexp` and `wildcard`. And the prefix query type can return all entries in a couple of requests. If the publisher wishes to restrict access to his lexical data, i.e. only offer queries that return a subset of the data, these query types have to be disabled in Kosh's source code.

[37] Freedict Kosh data module, https://github.com/cceh/kosh_data/tree/master/freedict

[38] inotify Linux Manpage, http://man7.org/linux/man-pages/man7/inotify.7.html



kosh - APIs for Dictionaries 1.0.0

[Base URL: kosh.uni-koeln.de/api/hiztegibatua/restful]

/api/hiztegibatua/restful/spec

hiztegibatua

GET /entries

Parameters

Name	Description
field * required string (query)	lemma
query * required string (query)	arb
query_type * required string (query)	prefix
size integer (query)	20

Responses — Response content type: application/json

Curl

```
curl -X GET "http://kosh.uni-koeln.de/api/hiztegibatua/restful/entries?field=lemma&query=arb&query_type=prefix&size=20" -H "accept: application/json"
```

Request URL

```
http://kosh.uni-koeln.de/api/hiztegibatua/restful/entries?field=lemma&query=arb&query_type=prefix&size=20
```

Server response

Code	Details
200	Response body

```
{
  "data": {
    "entries": [
      {
        "created": "2019-06-03T11:16:04.127698",
```

Figure 3: Swagger UI[39], prefix query for ‘arb’ in the HBO, max. 20 results.

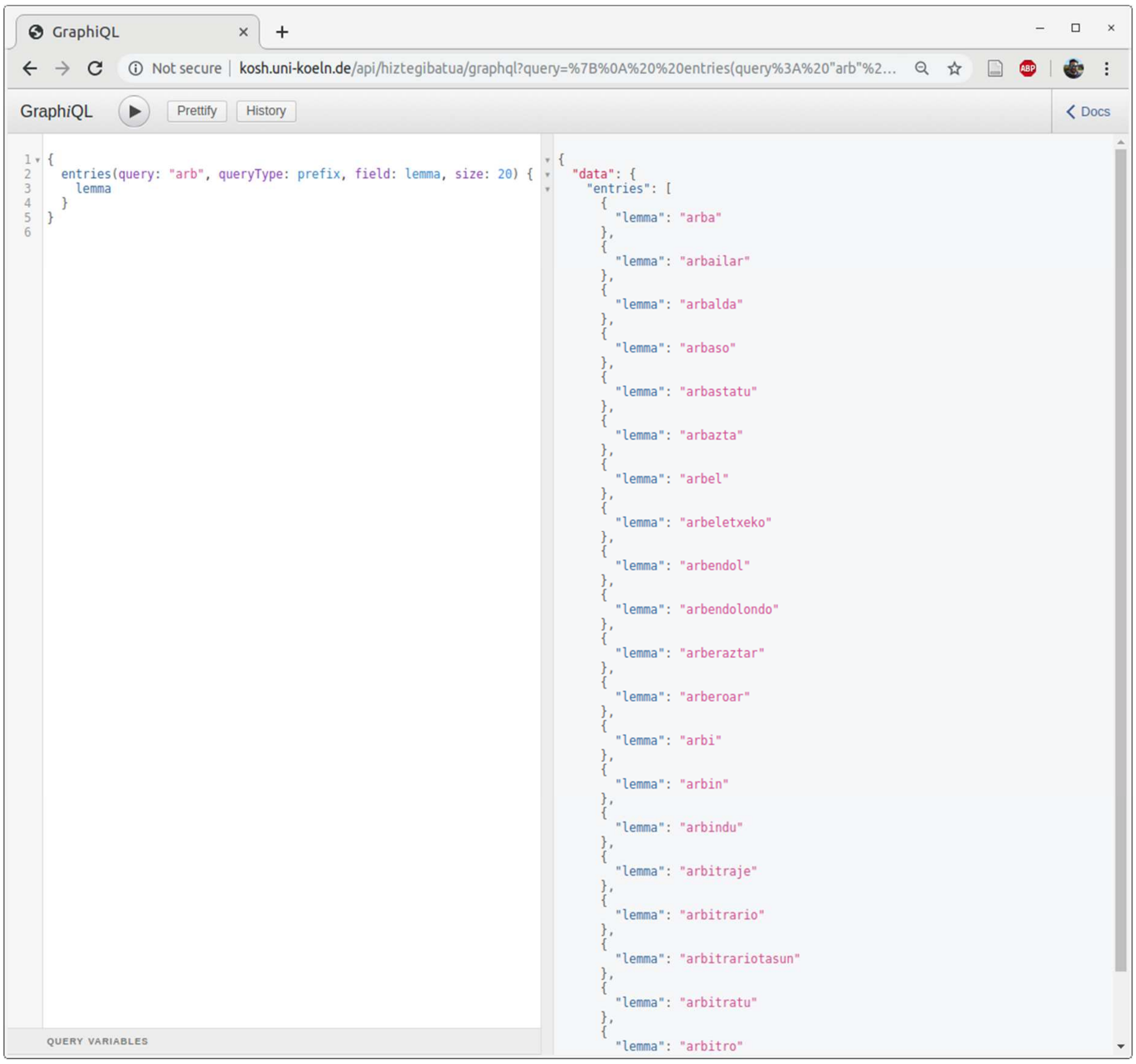
3.4 API access

While all the previously described layers of Kosh are crucial for its functioning, the APIs represent the visible and most relevant layer for clients. Kosh offers the two most popular API paradigms of the last decade for each indexed dataset: REST and GraphQL, and both return data in JSON format. The main differences between them are that GraphQL has a single endpoint, is typed, and that in a GraphQL query, unlike when using REST, the fields to be returned need to be explicitly specified. While this function can also be implemented in a REST API via sparse fieldsets[40], it is not a constraint on its implementation. For example, when using GraphQL, a client may

[39] Swagger UI, https://swagger.io/tools/swagger-ui

[40] JSON API, Sparse Fieldsets, https://jsonapi.org/format#fetching-sparse-fieldsets

retrieve only the headword field of all entries matching a specific query, e.g. all headwords that have the prefix ‘arb’, while a RESTful query would retrieve all the available fields related to the matched entries. Thus, one of the main advantages of GraphQL over REST is reduced data load, which can be relevant for mobile applications in areas with connectivity problems.


GraphiQL
Not secure | kosh.uni-koeln.de/api/hiztegibatua/graphql?query=%7B%0A%20%20entries(query%3A%20"arb"%2...
GraphiQL
Prettify
History
Docs
QUERY VARIABLES

```
{
  entries(query: "arb", queryType: prefix, field: lemma, size: 20) {
    lemma
  }
}
```

```
{
  "data": {
    "entries": [
      {
        "lemma": "arba"
      },
      {
        "lemma": "arbailar"
      },
      {
        "lemma": "arbalda"
      },
      {
        "lemma": "arbaso"
      },
      {
        "lemma": "arbastatu"
      },
      {
        "lemma": "arbazta"
      },
      {
        "lemma": "arbel"
      },
      {
        "lemma": "arbeletxeko"
      },
      {
        "lemma": "arbendol"
      },
      {
        "lemma": "arbendolondo"
      },
      {
        "lemma": "arberaztar"
      },
      {
        "lemma": "arberoar"
      },
      {
        "lemma": "arbi"
      },
      {
        "lemma": "arbin"
      },
      {
        "lemma": "arbindu"
      },
      {
        "lemma": "arbitraje"
      },
      {
        "lemma": "arbitrario"
      },
      {
        "lemma": "arbitrariotasun"
      },
      {
        "lemma": "arbitratu"
      },
      {
        "lemma": "arbitro"
```

Figure 4: GraphiQL - Prefix query for ‘arb’ in the HBO, fetching only related lemmas, max. 20 results

The framework provides user interfaces for all APIs (including documentation of available endpoints, queries, and typings). This way, all those interested in accessing the lexical data provided by the APIs can easily test and integrate them. For each data module Kosh serves an instance of Swagger UI (see Figure 3), running against all RESTful endpoints, and a GraphiQL[41] instance (see Figure 4), to allow running all available queries.

[41] GraphiQL GitHub repository, https://github.com/graphql/graphiql

Kosh offers two RESTful endpoints per data module: `entries` and `ids`. Using the `entries` endpoint, a client may search within the default available `xml` field as well as within any field defined by the respective data module. The `ids` endpoint fetches entries by specifying one or more entry IDs. For each GraphQL API endpoint the same two types of queries are available: `entries` and `ids`. All those API endpoints only offer consumption of lexical data, no modifications can be made to the underlying dataset, i.e. only HTTP GET requests are allowed.

3.5 Deployment

Kosh can be deployed in two different ways: Either natively or via Docker. The first option requires a Unix-based system with Python 3.6+ and Elasticsearch installed and running. This can be achieved by simply running the included `Makefile`, which installs Kosh and all required Python libraries, and providing a suitable configuration either on Kosh's command line or via a configuration file.

The second deployment option requires Docker and is the easiest method to deploy and maintain a Kosh instance. Docker is an operating-system-level virtualization tool which is popular among developers and administrators due to the possibility of distributing software packages as containers, i.e. isolated from each other. At the same it time offers clear and effective ways of bundling them together. To orchestrate containers and integrate them as services, Docker provides docker-compose[42], which in this case is employed to bundle an Elasticsearch and a Kosh instance together.

When using the included `docker-compose.yml` and `docker-compose.local.yml`, Kosh can be easily setup without the need to install any additional software. Docker will pull the Elasticsearch and Kosh images from Docker Hub, where they are both built automatically, i.e. the images always contain the latest versions of Kosh and Elasticsearch.

Kosh's source code is available on GitHub. For demonstration purposes, we also provide another GitHub repository, Kosh Data[43], that contains different data modules, so that users may transfer the structure of Kosh data modules onto their own datasets.

4. Conclusions and further development

In this article, we have presented Kosh and its main goal: To provide efficient and easy-to-configure access to lexical data. For this purpose, we have described the various theoretical considerations and technical decisions that have been made: i) Choosing XML as the data input format, ii) selecting Elasticsearch as Kosh's storage layer, and

[42] Docker Compose, https://docs.docker.com/compose

[43] Kosh Data GitHub repository, https://github.com/cceh/kosh_data

iii) adopting REST and GraphQL as its default API paradigms.

Kosh is a stable and high performing microservice that offers cutting-edge technologies with a relatively low learning curve for users without strong technical skills. Still, if it is used in production then aspects such as deploying a web server or user analytics should ideally be addressed by technical staff. Currently, only one field may be queried via the APIs, while the underlying search engine offers a much more fine-grained query logic. We plan to expose more of this functionality through Kosh's APIs in the future. We also envision the implementation of further API paradigms to enrich Kosh with more possibilities of serving lexical data. Besides such long-term goals, we are also committed to accomplish short-term development milestones, including continuous support in form of upstream library updates and bug fixes.

5. References

Böhtlingk, O. & Roth, R. (1855-1875). *Sanskrit-Wörterbuch.* St. Petersburg: Kaiserliche Akademie der Wissenschaften.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* Ph.D. thesis, University of California, Irvine. URL https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

Grassman, H. G. (1873). *Worterbuch zum Rig-veda.* Wiesbaden: O. Harrassowitz.

Kallas, J., Koeva, S., Kosem, I., Langemets, M. & Tiberius, C. (2019). D1.1 Lexicographic Practices in Europe: A Survey of User Needs. Deliverable D1.1, Elexis. European Lexicographic Infrastructure. https://elex.is/wp-content/uploads/2019/02/ELEXIS_D1_1_Lexicographic_Practices_in_Europe_A_Survey_of_User_Needs.pdf.

Mangeot, M. & Enguehard, C. (2018). Dictionaries for Under-Resourced Languages: from Published Files to Standardized Resources Available on the Web. Research Report, Laboratoire d'informatique de Grenoble. URL https://hal.archives-ouvertes.fr/hal-02056905.

McCrae, J. P., Bosque-Gil, J., Gracia, J., Buitelaar, P. & Cimiano, P. (2017). The OntoLexLemon Model: Development and Applications. In I. Kosem et al. (eds.) *Proceedings of the the 5th Biennial Conference on Electronic Lexicography (eLex 2017).* Leiden, the Netherlands, pp. 587– 597. https://elex.link/elex2017/wp-content/uploads/2017/09/paper36.pdf.

Mondaca, F. (2018). C-SALT APIs for Sanskrit Dictionaries: A Novel Approach for Accessing Digital Lexical Resources Online. Workshop on eLexicography: Between Digital Humanities and Artificial Intelligence. Co-located with EADH 2018 - Data in Digital Humanities. December 19, 2018. Galway, Irland. https://lexdhai.insight-centre.org/Lex_DH___AI_2018_paper_7.pdf.

Monier-Williams, M. (1899). *A Sanskrit-English dictionary: Etymologically and philologically arranged with special reference to Cognate indo-european languages.* Oxford: The Clarendon Press.

Měchura, M. (2016). Data Structures in Lexicography: from Trees to Graphs. In *The*

10th Workshop on Recent Advances in Slavonic Natural Languages Processing, RASLAN 2016, Karlova Studanka, Czech Republic, December 2-4, 2016. pp. 97–104. URL http://nlp.fi.muni.cz/raslan/2016/paper04-Mechura.pdf.

Měchura, M. (2018). Shareable Subentries in Lexonomy as a Solution to the Problem of Multiword Item Placement. In J. Čibej, V. Gorjanc, I. Kosem & S. Krek (eds.) *Proceedings of the XVIII EURALEX International Congress: Lexicography in Global Contexts.* Ljubljana University Press, Faculty of Arts, pp. 223–232. http://euralex.org/wp-content/themes/euralex/proceedings/Euralex%202018/118-4-2964-1-10-20180820.pdf.

Shevat, A., Sahni, S. & Jin, B. (2018). *Designing Web APIs.* Sebastopol: O'Reilly Media.

Tarp, S. (2015). La teoría funcional en pocas palabras. *Estudios de Lexicografía*, 4, pp. 31–42.

Torvalds, L. (1997). *Linux: a Portable Operating System.* Master of Science Thesis, University of Helsinki. https://www.cs.helsinki.fi/u/kutvonen/index_files/linus.pdf.