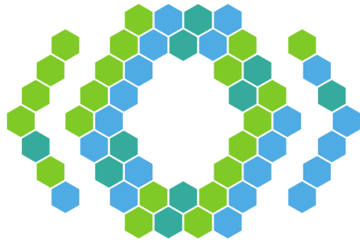


Deliverable D2.1
Spezifikation der Crawler-Komponente

Autoren: Matthias Wauer, Geraldo de Souza, Adrian Wilke, Afshin Amini

Veröffentlichung	Öffentlich
Fälligkeitsdatum	30.06.2018
Fertigstellung	30.06.2018
Arbeitspaket	AP2
Typ	Bericht
Status	Final
Version	1.01

Kurzfassung:
In this deliverable we specify the functional and non-functional requirements and architectural relationships of the the OPAL crawling component. We specify required interfaces and analyse existing crawling frameworks, including a decision on a specific framework to extend in the project. Furthermore, we briefly discuss the initial prototypical implementation for validating the specification.

Schlagworte:
crawler, focused crawler, framework, extraction, specification

mFUND
Das Startkapital für die Mobilität 4.0

Bundesministerium
für Verkehr und
digitale Infrastruktur

Inhalt

Introduction **2**
Motivation 2
Goals and non-goals 2
Requirements for the crawler component **3**
Functional requirements 3
Non-functional requirements 4
Interfaces **5**
Interaction with other components in OPAL 5
Operations 6
Data formats 7
Prototypical implementation **7**
Considered existing crawler frameworks 7
Evaluation of existing crawler frameworks 9
Initial implementation 10
Conclusions **11**

1 Introduction

The primary goal of OPAL is to enable users to find open data easily by harvesting, analysing and integrating available metadata from different Web sources. In this deliverable, we will focus on the first step towards this goal: the identification and extraction of metadata from Web pages describing data sets. We will refer to the general requirements gathered in Deliverable D1.1 and the analysis of data sources in Deliverable D1.2.

1.1 Motivation

Open Data is published on a wide range of Web sites. While several of these sites are built on standard solutions like CKAN, many use custom implementations and provide data set descriptions in very different formats. Depending on the respective portal, the metadata of these data sets is placed in varying HTML elements, sometimes hidden in semi-structured data. Hence, the harvesting has to be flexible enough so it can be adapted to these different sources. An extreme example for this case is the MDM portal, which requires certificate-based authentication to crawl the metadata.

As shown in Deliverable D1.2, many data sets relevant for OPAL have metadata scattered in different locations. For example, metadata for the "RadwegeGis Hamburg" data set (see Section 6.3.4 in D1.2) is available on mCLOUD, Transparenzportal Hamburg, European Data Portal and GovData.de. Additionally, there is a processed version of the dataset available at the ESRI portal. Thus, the harvesting approach in OPAL thus must be open to new sources identified ad-hoc. In other cases, such as data sets on the Portal data.deutschebahn.com, the crawler has to follow several links to access all relevant Web pages.

At the same time, open data and its metadata is a small fraction of the content of the Web. Hence, a generic Web crawler would not be a suitable method for gathering the metadata, as it would visit too many irrelevant Web pages.

1.2 Goals and non-goals

In this deliverable, we want to specify the functionality, as well as non-functional requirements, of the crawler component responsible for extracting the metadata from the data sources. Each requirement will be complemented with evaluation criteria that have to be met for this requirement to be successfully implemented. This specification includes the interfaces and data formats used to exchange information with other components in the OPAL platform (see Deliverable D1.3 on the system architecture), as far as these can be specified at this point in time. Furthermore, we will document a first prototypical implementation of the crawling component. For this, we also discuss available software frameworks that could be used as a foundation.

We will **not** specify details on how to learn crawling strategies from extracted metadata. This will be part of Deliverable D2.4. Also, in contrast to the functional and non-functional evaluation criteria we will not define the details of comprehensive benchmarks on which the crawler will be evaluated. The benchmark definition will be part of Deliverable D2.3.

2 Requirements for the crawler component

In the following, we list two types of requirements for the OPAL crawler component. On the one hand, these are functional requirements like services or functions, which must be provided by the crawler implementation. On the other hand, non-functional requirements describe quality issues, underlying conditions, or requirements, which are open to be implemented in various ways. The lists of requirements will be used in the decision-making process for the crawler architecture and implementation as well as for the comparison of existing frameworks (see Section 4).

2.1 Functional requirements

Key	Title	Description	Evaluation Criteria	Reference (D1.1)
CF1	Focused Crawling of HTTP pages	The focused crawler uses predefined seed-lists to access and run through Web resources and provide filtering options for following links to other Web resources.	Manual comparison of meta information at data portals with crawled data.	AK11
CF2	Access via various protocols	The crawler can access data using different standard protocols, e.g. HTTPS and FTP	Test cases for crawler connectors; availability of crawled data from sites using these protocols.	AK11
CF3	Periodical crawling	A function to start periodical and iterative crawling processes, supported by the crawling framework, is available in the OPAL administration.	Availability of this function with unit/integration test.	AK4
CF4	Fetching	Relevant raw data, which was found by the crawler, is extracted and stored.	Precision/Recall evaluation with manually annotated gold standard samples of sources.	AK11
CF5	Analysis	Semi-structured, fetched data is analyzed to access only relevant parts.	Precision/Recall evaluation with manually annotated samples	AK12
CF6	Extraction	Raw data strings are extracted from surrounding structural data (e.g. HTML	The extracted data does not include structural	AK12

		tags).	information.	
CF7	Semantic, machine readable data storage	Found data should be stored in a data format which is appropriate to handle the semantics of the extracted information, to be linked in a following integration step. (E.g. Turtle, Terse RDF Triple Language)	Validation of crawled data.	AK7, AK9
CF8	Storage of time-specific data	The crawler component enriches extracted data (see CF7) with a timestamp and stores it.	Unit test.	AK17, AK4
CF9	CKAN API support	The crawler can use the CKAN API for CKAN sources for more efficient data access.	Integration test case.	

Table 1: Functional requirements for the crawling component.

Legend:

- AK: consolidated requirement (konsolidierte Anforderung) in OPAL deliverable D1.1
- CF: functional requirement for crawling

2.2 Non-functional requirements

Key	**Title**	**Description**	**Evaluation Criteria**	**Reference (D1.1)**
CN1	Configuration	A configuration of data seeds (e.g. catalogue overviews) is possible via a user interface for humans and in a programatic way (via an API).	Availability of the configuration UI and API, test cases.	
CN2	Monitoring of crawling process	The crawler component provides an appropriate overview of ongoing crawling processes.	Survey among OPAL members.	
CN3	Control of crawling process	There is an option to manually control (start, stop) the crawling component for OPAL administration in an user-friendly manner.	Survey among OPAL members.	(AK6)
CN4	Documentation of component	The developers provide a documentation for the	Document to provide.	

		configuration of the crawler. (Note: Already done by Squirrel mini-tutorial of Geraldo)		
CN5	Documentation of data flow	A description of the storage of data is provided. (Note: Useful for monitoring)	Document to provide.	
CN6	Time efficient crawling	Multiple instances or sub-components allow to crawl single or multiple resources in parallel. (Note: Squirrel workers)	Test of sequential and parallel crawling jobs.	
CN7	Polite crawling	The crawler should respect robots.txt and add reasonable pauses when between accesses.	Integration test with server mock tracking the crawler behaviour.	

Table 2: Non-functional requirements for the crawling framework.

Legend:

- AK: consolidated requirement (konsolidierte Anforderung) in OPAL deliverable D1.1
- CN: non-functional requirement for crawling

3 Interfaces

3.1 Interaction with other components in OPAL

Regarding the overall OPAL architecture, the crawler has to be controlled by components, but does not have to control sub-components of the OPAL architecture stack itself (see Figure 1). Therefore, an application interface for incoming requests and outgoing responses has to be implemented.

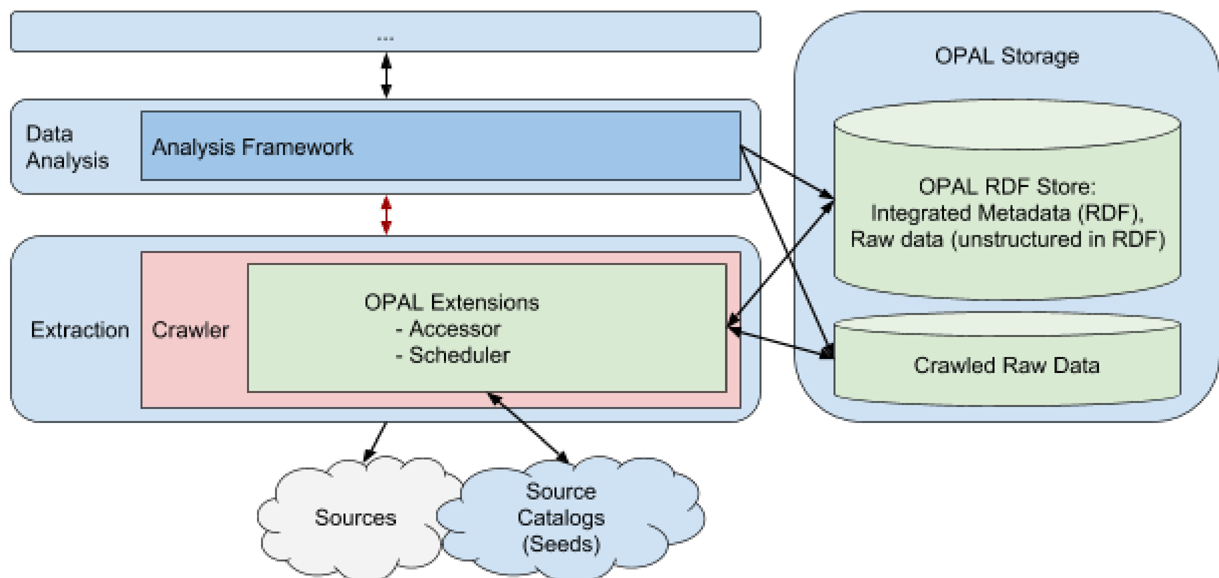


Figure 1: Architecture part of the crawler component

Crawling processes will typically be executed periodically and use known data sources, identified by the URIs of metadata portals. For these known data sources, a static configuration is needed. For instance, it has to be specified how individual parts of content can be identified inside a semi-structured data source like an HTML resource. This crawling information usually only changes, if the layout of a website changes. For resources out of structured data sources or API calls, the information about data structures is also defined in a static manner. A direct interaction with the crawling component is not required.
The accessed and downloaded data should be stored in a data store for raw data. Additionally, extracted structured data parts are written into an RDF store. The extraction is done by the analysis framework, which needs to provide a respective endpoint for this operation. The store will also be used to persist metadata about crawling processes, e.g. the time of the last successfully finished data access. Using the RDF store for writing persisted data opens the option for other OPAL components to access crawling information using the storage component and makes a direct interaction with the crawler component superfluous.
For the use case of URIs, which have to be crawled immediately, a solution for interaction is required. For those requests, the crawling component has to provide an interface to send a crawling request. For the implementation of this requirement, the crawler component to be determined has to provide a REST API or a messaging endpoint (AMQP queue etc.).
Another requirement is to control parallel crawling processes. A synchronization element or an implementation as a singleton is required by the selected crawler implementation.
When a new resource has been crawled or re-crawled, the new metadata should be processed by the upper layers in the OPAL architecture. For the first iteration, this is invoked manually, but later versions should support a non-blocking, loosely coupled way to trigger further processing, e.g., using Linked Data Notifications or sending messages on updated resources to an AMQP broker.
Further functionality, such as learning a crawler strategy, classification of crawled documents and application of natural language processing techniques is not part of this crawler component, but of the analysis framework, which in turn reconfigures the crawler to improve the focused crawling performance.

3.2 Operations

The crawler interface is mainly composed by two parts: one related to statistics about the crawled uris and the other that shows the current crawled graph.
The main page shows the current status of the crawling process, displaying the pending uris, how many uris were crawled, how many workers are active and how many workers are dead. Also, it is possible to query the crawled uris by using Sparql query syntax in this page.
A worker it is considered dead if there is no more tasks available to it. Every worker registered in the Frontier, will be associated to the ip address of the first uri assigned to it. When there is no more uris related to that ip to be crawled, the task of that worker will be considered done and the Frontier will consider the worker dead, waiting for a new worker to get uris from different domains.
The crawled graph shows all the domains crawled as nodes and its edges connecting themselves. This level of details was chose due to performance issues (if the graph would render all the uris, it would require heavy processing, to not say impossible). From this view, it will be possible to export the graph to different formats and extract other graph related statistics.

3.3 Data formats

In this section we will briefly discuss the data formats to be processed by the crawler component.

With regards to the input formats, the crawler is concerned with two things. First, there's the seed file containing the initial URIs to be crawled. This simple list is augmented with an option to filter the URIs to be followed. We first define this as a WhiteList here, e.g., as a specification of acceptable URI patterns to be crawled. If necessary, this should later be extended to a grey listing approach, i.e., making it possible to also specify URI patterns to be excluded, which can often simplify this specification. Finally, we define the acceptable media types for the documents to be crawled. The crawler should support the following formats:

- HTML Documents
- RDF serialization types
    - Turtle
    - RDF/XML
    - N-Triples
    - JSON-LD
    - RDF/JSON
    - TriG
    - N-Quads
    - TriX
    - RDF Binary
- JSON (from CKAN API)

With regards to the output formats, the crawling component should support the following options:

- Triple Store (for SPARQL update requests)
- RDF N-Quad line-based, plain text format to File System
    - Compressed or not (depending of the selected implementation)
- Source file (containing the crawled raw data)
    - Storing the original fetched documents

4 Prototypical implementation

According to the description of work, the implementation of the focused crawling component in the OPAL project should be reusing an existing framework. In this section, we identify existing relevant open-source crawler frameworks and describe them briefly. Then we compare them w.r.t. the requirements in Section 2 and explain our decision for one of them before detailing the initial implementation towards the D2.2 prototype deliverable (first version of the crawler component).

4.1 Considered existing crawler frameworks

The primary criteria for the selected crawler framework include: (should be part of Section 2):

- crawling Web pages (HTML)
- extensible to different protocols (e.g., FTP required for DWD)
- focused crawling possible (via extensions)
- machine-readable output format (preferably RDF)

With a literature review and using a generic Web search engine we identified the following classes and instances of open-source crawling frameworks:

- Generic Crawlers (WebMagic, StormCrawler, Apache Nutch, REX, HTTrack)
- LD Web Crawlers (ldspider, slug)
- Web crawler with RDF output (Any23, Squirrel, TDSP)

The following list introduces each framework briefly.

- **WebMagic**[1] is a scalable open-source HTTP crawler. It supports a simple API for extracting HTML elements. It does not support protocols other than HTTP.
- **StormCrawler**[2] is an open source SDK for building distributed web crawlers based on Apache Storm. The project is under Apache license v2 and consists of a collection of reusable resources and components, written mostly in Java.
- **Apache Nutch**[3] is a highly extensible and scalable open source web crawler software project. Stemming from Apache Lucene, the project has diversified and now comprises two codebases, namely:
    - Nutch 1.x: A well matured, production ready crawler. 1.x enables fine grained configuration, relying on Apache Hadoop data structures, which are great for batch processing.
    - Nutch 2.x: An emerging alternative taking direct inspiration from 1.x, but which differs in one key area; storage is abstracted away from any specific underlying data store by using Apache Gora for handling object to persistent mappings. This means that an extremely flexibile model/stack for storing everything (fetch time, status, content, parsed text, outlinks, inlinks, etc.) into a number of NoSQL storage solutions can be implemented.

[1] http://webmagic.io/en/
[2] http://stormcrawler.net/
[3] http://nutch.apache.org/

- **REX**[4] is an RDF extraction framework for Web data that can learn XPath wrappers from unlabelled Web pages using knowledge from the Linked Open Data Cloud.
- **HTTrack**[5] is a free (GPL, libre/free software) and easy-to-use offline browser utility. It allows you to download a World Wide Web site from the Internet to a local directory, building recursively all directories, getting HTML, images, and other files from the server to your computer.
- The **LDSpider**[6] project provides a web crawling framework for the Linked Data web. Requirements and challenges for crawling the Linked Data web are different from regular web crawling, thus the LDSpider project offers a web crawler adapted to traverse and harvest content from the Linked Data web.
- **Slug**[7] is a web crawler (or Scutter) designed for harvesting semantic web content. Implemented in Java using the Jena API, Slug provides a configurable, modular framework that allows a great degree of flexibility in configuring the retrieval, processing and storage of harvested content. The framework provides an RDF vocabulary for describing crawler configurations and collects metadata concerning crawling activity.
- **Anything To Triples (any23)**[8] is a library, a web service and a command line tool that extracts structured data in RDF format from a given source, provided in several supported formats. Crawling is supported by a plugin, but isn't focused.
- **Squirrel**[9] is an open-source crawling framework which enables extracting elements from different sources into RDF. It is component-based and can easily be extended towards supporting further fetchers, analyzers, queues, sinks etc.
- **Template-Driven Semantic Parser (TDSP)**[10] is a crawler/parser approach which is capable to provide the semantics of extracted Web data in the RDF format, based on templates defined in XML.

4.2 Evaluation of existing crawler frameworks

The crawling frameworks discussed above have been analysed and compared to the functional requirements (see Section 2). Table 3 shows a summary of the findings.

Tool/Req.	CF1	CF2	CF3	CF4	CF5	CF6	CF7	CF8	CF9
WebMagic	X (Just one link)	✔(Http only)	X	✔	✔	✔	✔	✔	X
StormCrawler	✔	✔(Http only)	X	X	X	X	X	X	X
Apache Nutch	✔	✔	X	✔	✔	✔	✔	X	X
REX	✔	✔(Http only)	X	✔	✔	✔	✔	X	X

[4] http://aksw.org/Projects/REX.html
[5] https://www.httrack.com/
[6] https://github.com/ldspider/ldspider
[7] http://www.ldodds.com/projects/slug/
[8] https://any23.apache.org/index.html
[9] https://github.com/dice-group/Squirrel
[10] https://link.springer.com/content/pdf/10.1007%2F978-3-319-15615-6_26.pdf

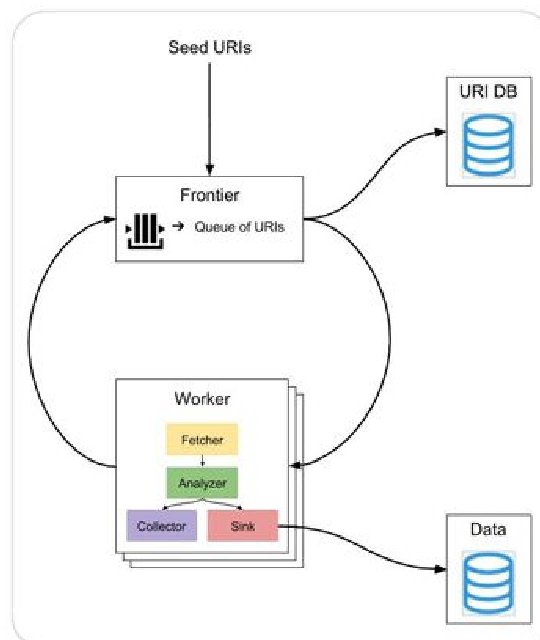
HTTrack	✔	✔(Http only)	✓	X	X	X	X	X	X
ldspider	✔	✔(Http only)	X	✔	✔	✔	✔	✔	X
slug	✔	✔(Http only)	✓	✔	✔	✔	✔	✔	X
Apache Any23	X (Just one link)	✔(Http only)	X	✔	✔	✔	X	X	X
Squirrel	✓	✔	✔	✓	✔	✔	✔	✔	✔
TDSP	X	X	X	✔	✔	✔	✔	✔	X

Table 3: Evaluation matrix of crawling frameworks.
✔: Capable and no development needed,
✓: Capable but needs development
X: Not capable

According to this analysis, we have decided to base our crawler implementation on the Squirrel framework, as it supports most requirements directly and can be extended towards the others.

4.3 Initial implementation

Squirrel comprises two major parts - a single **frontier** and *n* **workers**. The frontier manages the crawling process and is based on a queue as well as a database containing the URIs that already have been crawled in the past. The worker requests work packages from the frontier, performs the actual crawling (fetching, analysing, storage) and sends new URIs to the frontier.



The **frontier** is responsible to decide which URI's should be crawled and send it for the respective **worker.** The worker then, receives the URI, fetch the file and uses the analyzer to find new URI's in the fetched file, that will be serialized by the collector and stored by the sink. In the end, each URI found is deserialized, sent to the frontier and the process is repeated.

For crawling, the frontier will use the SEED_FILE environment variable, where should be the file that contains seeds to start the crawling process. The frontier can perform focused crawling if the environment variable URI_WHITELIST_FILE is set. The file should contains a list of domains which will be allowed to be crawled. If the frontier receives URI's from other domains, it will be ignored. If this variable is not set, the frontier will allow everything to be crawled.
The worker setup requires the following environment variables:

- **OUTPUT_FOLDER**, where the sink will store the crawled URI's.
- **HTML_SCRAPER_YAML_PATH**, the path where html scraper config files will be stored.
- **CONTEXT_CONFIG_FILE**, the spring-context xml file, storing all the implementations that will be used for the current worker.

The analyzers are responsible for analyzing the fetched file in search for triples. There are two available analyzers: **RDFAnalyzer** and **HtmlScraperAnalyzer**. The RDFAnalyzer is responsible for matching the proper RDF serialization (if the fetched file is a RDF) to extract all the triples found and requires no additional configuration. The HtmlScraperAnalyzer, however, needs a yaml configuration file to define which pages and which elements from a certain domain should be crawled. Further information on the HtmlScraperAnalyzer can be found in the Squirrel wiki.[11]

To run, the worker requires also the CONTEXT_CONFIG_FILE environment variable to run. In this file, it is defined all the implementations that should be injected by spring-framework into the worker. The possible implementations are:

- **Sink:**
  - **FileBasedSink:** Stores the triples in the file system, using the OUTPUT_FOLDER env variable.
  - **InMemorySink**: Stores the triples temporarily in memory.
  - **RDFSink**: Stores the triples in Sparql Triple Stroe
- **Collector**:
  - **SqlBasedUriCollector**: Collects serialized triples in local hsqldb.
  - **SimpleUriCollector**: Collects serialized triples in memory.
- **Serializer**:
  - **GZipSerializer**
  - **GSonSerializer**
  - **SnappyJavaSerializer**

The current implementation of the Squirrel crawler, including OPAL extensions, can be found in the official Squirrel repository:
https://github.com/dice-group/Squirrel/

5 Conclusions

In this deliverable we defined all requirements for the crawler component, identified the relationship with the analysis framework and decided to use and extend the Squirrel framework as a foundation for the OPAL crawler component.

[11] https://github.com/dice-group/Squirrel/wiki/HtmlScraper_how_to