Collaborative Project

FROCKG - Fact Checking for Large Enterprise Knowledge Graphs

Project Number: E FROCKG Start Date of Project: 2020/01/01 Duration: 36 months

Deliverable 1.2: Component architecture and interfaces

Dissemination Level	Public
Due Date of Deliverable	Month 5, 2020/05/31
Actual Submission Date	Month 5, 2020/05/31
Work Package	WP1, Requirements Elicitation
Task	T1.2
Type	Report
Approval Status	Work in progress
Version	1.0
Number of Pages	20

Abstract:
This report presents the final version of the solution architecture and interfaces between component which are going to be developed as part of the FROCKG project.
The architecture comprises of a set of products, services, and components which can be combined in various ways to realize the FROCKG use cases.

Project by Eurostars.

History

Version	Date	Reason	Revised by
1.0	31/05/2020	Draft revised	Wolfgang Schell

Author List

Organization	Name	Contact Information
metaphacts GmbH	Peter Haase	ph@metaphacts.com
metaphacts GmbH	Wolfgang Schell	ws@metaphacts.com
metaphacts GmbH	Jeen Broekstra	jb@metaphacts.com
Paderborn University	Michael Röder	michael.roeder@upb.de
Sirma AI EAD	Nikola Rusinov	nikola.rusinov@ontotext.com
Sirma AI EAD	Todor Primov	todor.primov@ontotext.com
Zazuko GmbH	Adrian Gschwend	adrian.gschwend@zazuko.com

Contents

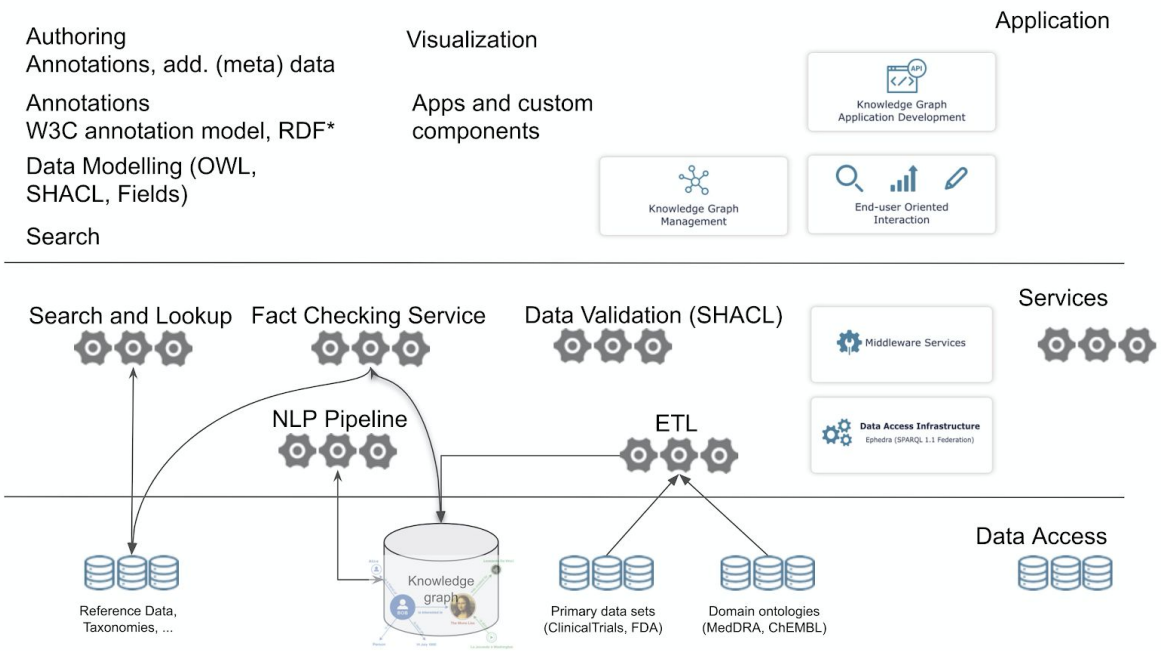
Overview

In the previous deliverable D1.1, we presented three use cases that showed the need of fact checking to ensure information correctness, especially when making decisions on data from untrusted sources. The FROCKG project aims to solve this problem by providing a system for fact checking based on knowledge graphs and text corpora with tight integration into data exploration, curation, and authoring tools.

In this document we aim at defining the architecture that will be used to implement such a system. Based on the initial requirements, and the project goals we present here the approach to create the FROCKG fact checking engine and integrated solution, together with the required components for interfacing with data sources and providing a way to interact with the fact checking component.

Solution Architecture

The FROCKG Solution Architecture is comprised of a number of components which can be used in an interchangeable way:



The solution architecture is split into multiple layers:
The **Application Layer** provides the components for end-user interaction. Additionally, it contains use-case specific content such as specific user interface views which are composed, orchestrated and customized to fulfil the specific requirements of concrete user stories.
The **Service Layer** hosts components, typically implemented as a set of independent services, which provide isolation of functional areas. Often, each component can be deployed independently of others and used in various applications or solutions. The services

are cut along process boundaries and can be deployed e.g. as Docker containers and are loosely coupled. Communication between services follows standardized protocols and means such as REST over HTTPS and similar protocols.
Each of these services has its own deployment model and can be run, operated, tuned independently.
The **Data Access Layer** holds all data in both structured and unstructured form. Triple Stores / Graph Databases and Relational or NoSQL Databases provide persistent storage, Reference Data Sets and public databases such as PubMed or Wikidata are available via remote lookup and data access protocols such as the W3C Reconciliation API[1], a SPARQL Endpoint or JDBC/ODBC. Data Ingestion and Integration is performed using Natural Language Processing (NLP) pipelines, Extract-Transform-Load (ETL) data integration tools and custom processing and automation steps.
Well-defined interfaces allow communication between the layers and individual components.

Building Blocks

Each of the components listed in the following sections is meant to be used as a building block that can be used to assemble a solution architecture for a target use case which is optimized for a certain deployment environment, target audience and organizational structure.
Besides the components described in this document, additional building blocks provided by external parties can be added as required to extend the solution architecture or interface with other systems.

Workflows

Based on the use cases outlined in deliverable D1.1 and the more detailed requirements and user stories described in deliverable D1.3 the following sections describe some workflows in a high-level technical view. These workflow diagrams provide a quick overview of the interactions between components and help in defining the required interfaces as well as documenting the flow of information and connections between individual building blocks.

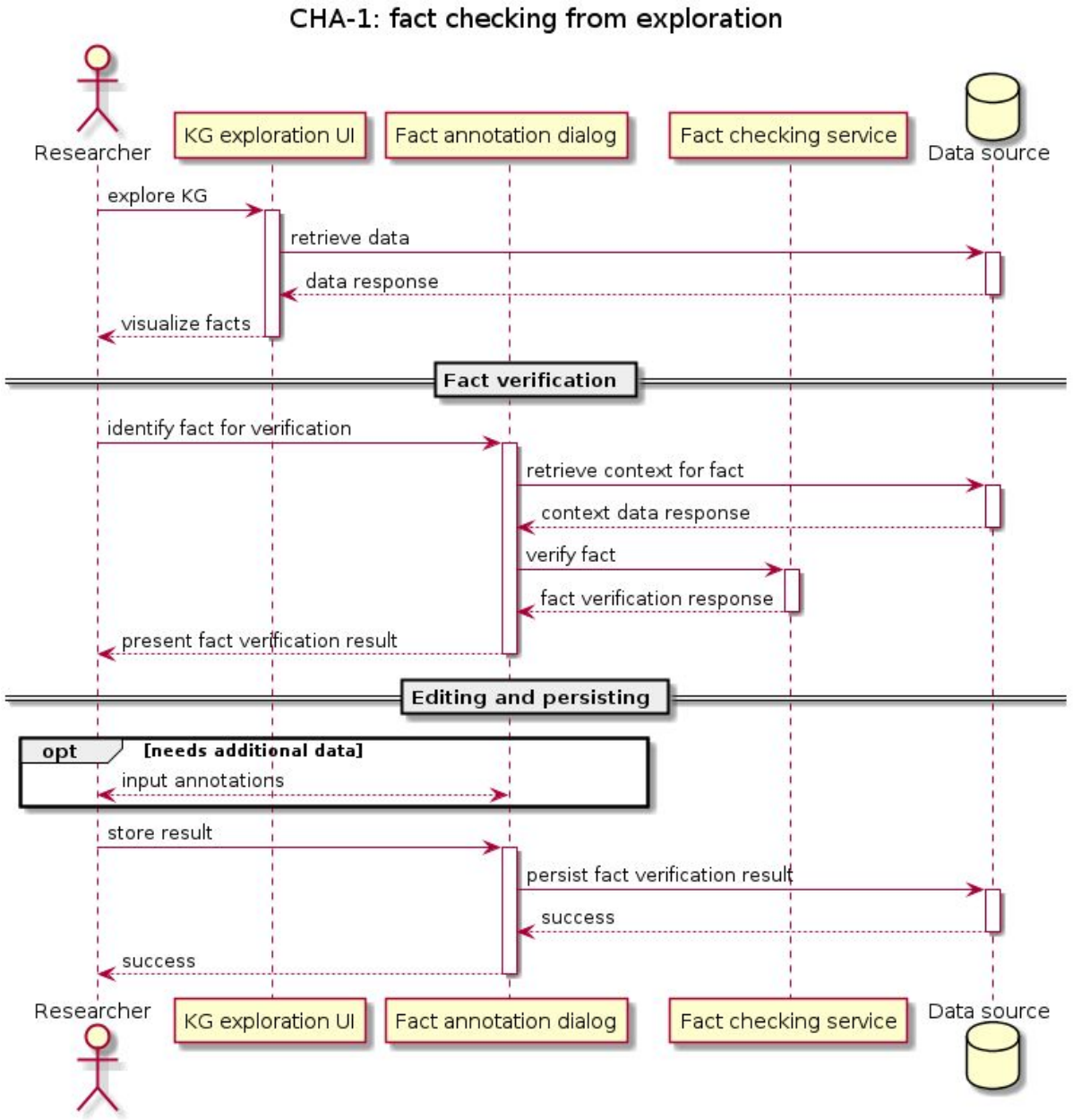
Fact checking, exploration, and annotation editing / curation

(covers use cases CHA-1, CHA-2, CHA-3)

This diagram covers the basic event sequence flow between the various actors in the proposed architecture for a researcher who explores a knowledge graph and chooses to fact-check, annotate, or curate certain facts. The components involved in these use cases interact in a sequence initiated by the researcher via the exploration user interface: the knowledge graph data is retrieved from the source and presented to the researcher. The researcher then chooses a particular fact in that knowledge graph data to verify, which initiates both a new user interface (an annotation dialog) and which activates the

[1] https://reconciliation-api.github.io/specs/latest/

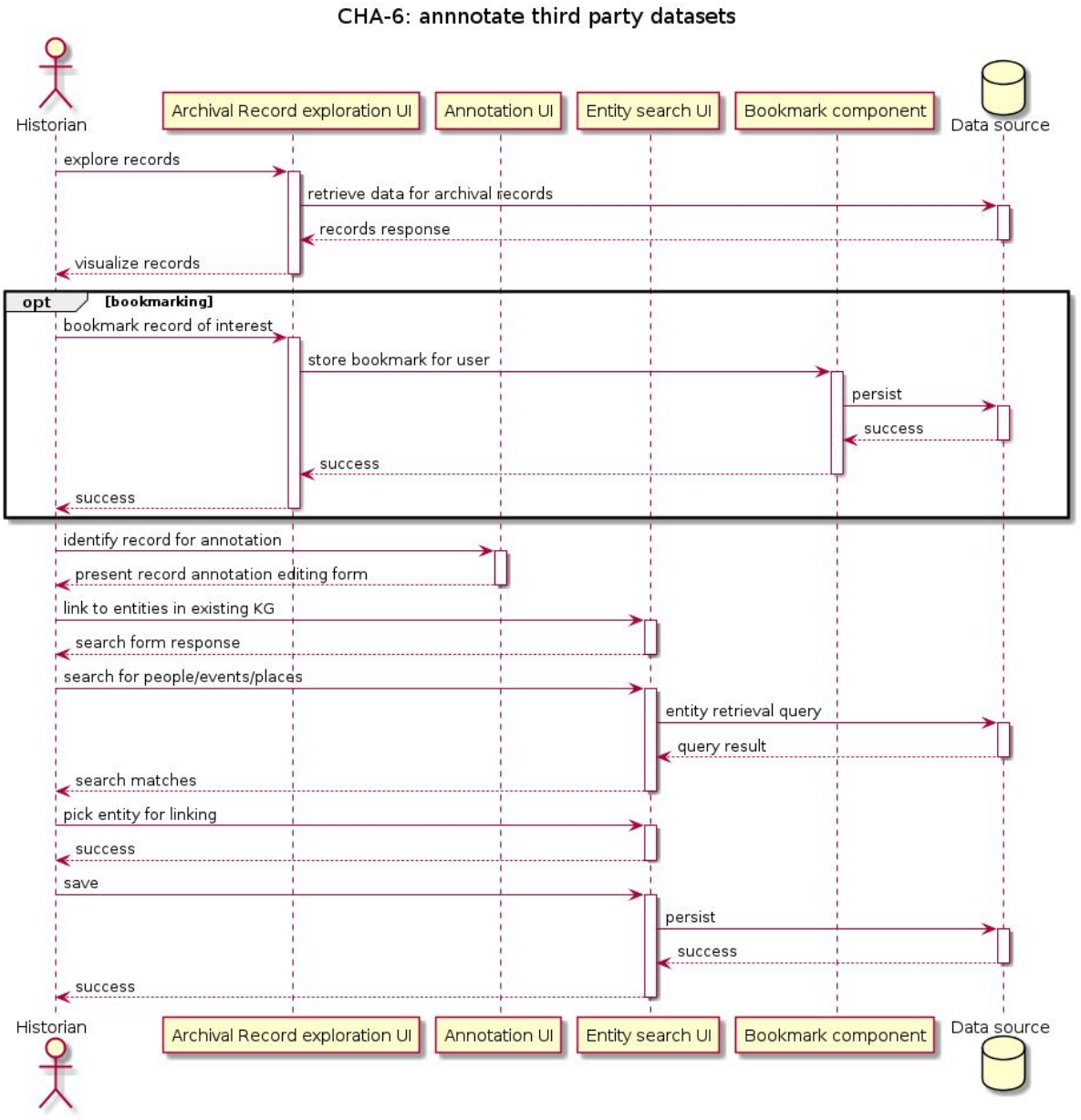
composition of a fact checking request, consisting of the initial fact and the relevant contextual information (retrieved from the data source), to supply to the fact checking service. Once the fact checking service has a result, this is presented to the researcher, who can act on this by optionally further editing or annotating, and finally, when satisfied, persisting the result as a fact annotation.



Annotating third party datasets

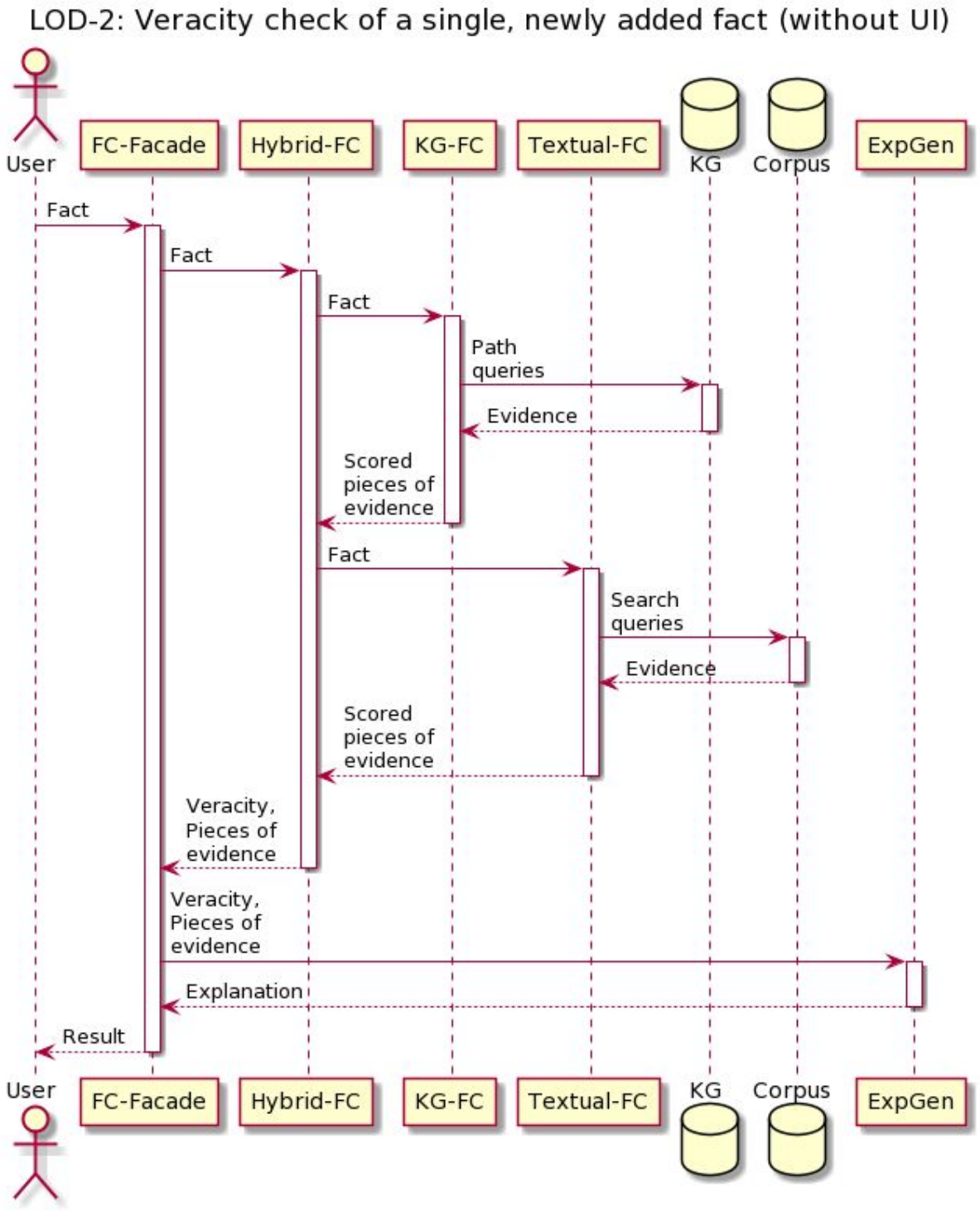
(covers use cases CHA-6)

This diagram covers the basic event sequence flow between the various actors in the proposed architecture for a researcher/historian who explores a third party set of records and chooses to bookmark, and annotate/curate certain records. Archival records are retrieved from the data source and presented to the historian. They have the option to bookmark a record of interest at any point in the exploration. This book which will be persisted for later referral. When the historian has identified a record they want to annotate, the annotation ui is opened, focused on the record in question. The historian is presented with the option to link the record to other entities in the knowledge graph (people, events, places, other records), and once done, the historian can choose to save the annotation result as an annotation.



Fact Checking Facade

This diagram shows the main steps for checking a newly added fact as described in the LOD-2 use case. In general, it can be seen as an example of how a fact is checked based on a hybrid approach, i.e., by using a corpus as well as a knowledge base as reference data. The Fact Checking Facade (FC-Facade) receives the initial request and forwards it to the hybrid Fact Checking service (Hybrid-FC). The Hybrid-FC component forwards the request to both—the knowledge-graph-based (KG-FC) and the text-based Fact Checking services (Textual-FC). Both services work in a similar way. The KG-FC component searches for connections between the subject and object of the given fact and measures whether these paths support the given fact. The Textual-FC component generates search queries and searches for the fact in the reference corpus. Both return their result including the scored pieces of evidence. The Hybrid-FC component processes them and generates a single veracity score for the fact. This score and the pieces of evidence are returned to the facade. The facade sends the pieces of evidence to the Explanation Generation component (ExpGen) which generates a natural language generation. After that, the facade returns all the results to the user.



Components

Fact Checking

The Fact Checking components offer a facade for the Fact Checking functionality. This facade is backed up by three services: 1) a corpus-based Fact Checking service, 2) a

Knowledge-Graph-based Fact Checking service and 3) a hybrid service. The facade makes use of the explanation service to generate a natural language explanation.

Fact representation

The methods of the Fact Checking components prefer RDF data as input and output format. Instead of repeating the RDF data for all single methods, we give a simple example of a result of the fact checking services for a single fact with two different pieces of evidence.

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix sh:   <http://www.w3.org/ns/shacl#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
@prefix nif:  <http://persistence.uni-leipzig.org/nlp2rdf/ontologies/nif-core#> .
@prefix ex:   <http://example.org/> .
@prefix fro:  <http://frockg.eu/vocabulary#> .

ex:fact      a      rdf:Statement ;
       fro:veracity "0.98"^^xsd:double ;
       fro:evidence ex:PathEvidence1,
<http://en.wikipedia.org/wiki/SomeNicePage#char=18,50> ;
       rdfs:comment "This comment contains a natural language explanation of the
calculated veracity and has been generated based on the provided evidence." .

ex:PathEvidence1    a      fro:PathEvidence ;
       sh:path ( ex:Property1 [ sh:inversePath ex:Property2 ] ex:Property3 ) ;
       fro:score    "0.99"^^xsd:double .

<http://en.wikipedia.org/wiki/SomeNicePage#char=18,50>      a
fro:TextualEvidence, nif:RFC5147String, nif:String ;
       nif:anchorOf "This sentence supports the fact."^^xsd:string ;
       nif:beginIndex      "18"^^xsd:nonNegativeInteger ;
       nif:endIndex "50"^^xsd:nonNegativeInteger ;
       nif:referenceContext
<http://en.wikipedia.org/wiki/SomeNicePage#char=0,186> ;
       fro:score    "0.8"^^xsd:double .
```

The fact itself is an RDF statement that has been given as input to a fact checking algorithm. It has a veracity score, a list of pieces of evidence and a natural language explanation attached. The subject, predicate and object of the fact have been skipped in the example. An alternative could be the usage of RDF* to annotate the fact directly. For the path evidence, the SHACL vocabulary is used to define the path that has been found to support the fact. In the example, the path comprises three properties (the second property is inverted). In addition, the path comes with a score that shows its importance with respect to the veracity. The second piece of evidence is a sentence that has been found in a document. For describing its context, content and position the NIF vocabulary is used.

Facade service

Method	checkFact
Functionality	The method checks a single fact. The fact comprises a triple of URI resources.
Type	HTTP GET (or POST)
Input	RDF comprising ● an instance of rdf:Statement that should be checked ● labels of the subject, predicate and object of the statement (rdfs:label preferred)
Output	RDF (e.g., JSON-LD) ● Veracity value ● Explanation ● Pieces of evidence

Method	checkKG
Functionality	The method checks a given knowledge graph (KG). The KG needs to be available as a SPARQL endpoint.
Type	HTTP GET (or POST)
Input	● sparql = URL of the sparql endpoint ● user = user name (optional) ● password = password (optional)
Output	RDF (e.g., JSON-LD) ● Veracity value ● Explanation ● Pieces of evidence

Hybrid Fact Checking service

Method	checkFact
Functionality	The method checks a single fact. The fact comprises a triple of URI resources.
Type	HTTP GET (or POST)
Input	RDF comprising ● an instance of rdf:Statement that should be checked ● labels of the subject, predicate and object of the statement (rdfs:label preferred)

Output	RDF (e.g., JSON-LD) ● Veracity value ● Pieces of evidence

Corpus-based Fact Checking service

Method	checkFact
Functionality	The method checks a single fact. The fact comprises a triple of URI resources.
Type	HTTP GET (or POST)
Input	RDF comprising ● an instance of rdf:Statement that should be checked ● labels of the subject, predicate and object of the statement (rdfs:label preferred)
Output	RDF (e.g., JSON-LD) ● Veracity value ● Pieces of evidence

Knowledge-Graph-based Fact Checking service

Method	checkFact
Functionality	The method checks a single fact. The fact comprises a triple of URI resources.
Type	HTTP GET (or POST)
Input	RDF comprising ● an instance of rdf:Statement that should be checked ● labels of the subject, predicate and object of the statement (rdfs:label preferred)
Output	RDF (e.g., JSON-LD) ● Veracity value ● Pieces of evidence

Explanations

We will provide a component that generates natural language explanations for a given Fact Checking result.

Method	generateExplanation
Functionality	The method takes the given evidence and produces a natural language explanation for the given veracity value-based on these evidences. The given list may contain textual evidence or RDF paths.
Type	HTTP POST
Input	RDF (e.g., JSON-LD) ● Veracity value ● Pieces of evidence
Output	RDF (e.g., JSON-LD) ● Explanation

Editing and Curation

Component	Data Authoring
Functionality	Manual data entry, authoring of meta data and annotations, authoring workflows
Type	HTTP GET/POST via manual user input (browser-based) or automated via REST, SPARQL Update[2], LDP[3] or Graph-Store[4] endpoint
Input	Schema information such as OWL, SHACL shapes, field definitions
Output	new data or meta data or annotations in RDF (data in materialized form, e.g. Turtle or as RDF4J data structures)

Component	Reference Lookup (Incoming and Outgoing)
Functionality	Lookup of reference data such as common identifiers, used for referencing and cross-linking between data sets, auto-suggestion for

[2] https://www.w3.org/TR/2013/REC-sparql11-update-20130321/
[3] https://www.w3.org/TR/ldp/
[4] https://www.w3.org/TR/2013/REC-sparql11-http-rdf-update-20130321/

	data entry and authoring
Type	HTTP GET/POST using the Reconciliation API[5]
Input	Search term or query, optional additional attributes (such as a target type) for further refinement
Output	List of candidates with ID, type(s), label, description, score

Component	Data quality / data validation
Functionality	Validation of data via shape descriptions; life cycle management (authoring, versioning, import/export) of shape definitions
Type	REST Interface, local Java Interface (RDFUnit)
Input	Shape definitions (SHACL RDF data) RDF Data to validate (data in materialized form, e.g. Turtle or as RDF4J data structures)
Output	Validation report in RDF (data in materialized form, e.g. Turtle or as RDF4J data structures) or JSON listing data constraint violations

Component	Federation to relational databases[6] (RDBMS) (Outgoing)
Functionality	Integration with external components such as Machine Learning (ML) pipelines, data sources, etc.
Type	SPARQL query translated into a JDBC/ODBC connection
Input	SPARQL query with bindings which are translated into SQL queries using query templates defined in SPIN[7] syntax.
Output	Tuple Result set for SELECT queries RDF data for CONSTRUCT / DESCRIBE queries

Component	Federation to web services[8] (REST) (Outgoing)
Functionality	Integration with external components such as Machine Learning (ML) pipelines, data sources, etc.

[5] https://reconciliation-api.github.io/specs/latest/
[6] https://help.metaphacts.com/resource/Help:EphedraSQLService
[7] https://spinrdf.org/
[8] https://help.metaphacts.com/resource/Help:EphedraRESTService

Type	SPARQL query translated into HTTP GET/POST/PUT requests
Input	SPARQL query with bindings which are translated into HTTP requests using query templates defined in SPIN[9] syntax.
Output	Tuple Result set serialized as XML or JSON for SELECT queries RDF data for CONSTRUCT / DESCRIBE queries

Component	Federation to RDF Knowledge Graphs (SPARQL) (Outgoing)
Functionality	Integration with external RDF data sources and reference databases
Type	SPARQL HTTP Protocol[10] and federation[11]
Input	SPARQL query
Output	Tuple Result set for SELECT queries RDF data for CONSTRUCT / DESCRIBE queries

Component	Exposing data via REST endpoints (Incoming)
Functionality	Exposing data defined as SPARQL queries via REST endpoints[12]
Type	HTTP GET/POST/PUT
Input	HTTP headers, body (typically form parameters)
Output	CSV, JSON or XML

Data Access & Exploration

Component	Endpoint Explorer
Functionality	Explore data in SPARQL endpoints and visualize in table-like manner
Type	HTTP GET
Input	Endpoint, optional RDF graph

[9] https://spinrdf.org/
[10] https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/
[11] https://www.w3.org/TR/2013/REC-sparql11-federated-query-20130321/
[12] https://help.metaphacts.com/resource/Help:QueryAsAService

Output	HTML, SHACL shapes (with introspection)

Component	Search-Component
Functionality	Explore data in a SPARQL endpoint in a customizable, HTML5 table-component with pivot-like filtering.
Type	HTTP GET
Input	Endpoint, additional HTML5 components for better data representation
Output	HTML

Incremental knowledge extraction

Component	RDF Knowledge Graphs (GraphDB)
Functionality	Storage and maintenance of referential knowledge graph for NLP tasks
Type	RDF triple store
Input	SPARQL query
Output	Tuple Result set for SELECT queries RDF data for CONSTRUCT / DESCRIBE queries

Component	NLP pipeline population service
Functionality	Storage and maintenance of referential knowledge graph for NLP tasks
Type	HTTP GET
Input	SPARQL query
Output	delimited result set

Component	Linked Data Gazetteer
Functionality	Named entity recognition GATE component
Type	GATE component
Input	Delimited list of knowledge graph resources
Output	Raw annotation list with with instance and start/end offsets

Component	GATE annotation pipeline
Functionality	Text processing pipeline
Type	GATE application (GAPP)
Input	Collection of documents (XML, TXT, etc)
Output	GATE XML with inline annotations

Component	Manual curation and validation
Functionality	Manual curation and validation of documents' semantic annotations.
Type	HTTP GET/POST via manual user input (browser-based) or automated via REST, SPARQL Update[13] or Graph-Store[14] endpoint
Input	Collection of documents GATE XMLs
Output	GATE XMLs

FROCKG Application Development

Method	Use Case-specific App[15] within metaphactory
Functionality	Provides content, views, configuration and optionally services to implement a use case using building blocks provided by the metaphactory Graph Data Platform
Type	Zip file containing artefacts for building web interfaces (HTML pages, CSS, Javascript, images, …) as well as configuration for various aspects of the use case and/or configuration information to integrate with external services
Input	N/A
Output	N/A

Component	RDF Mapping Generation
Functionality	Generates R2RML, RML or CSVW mapping definitions based on a lightweight mapping DSL
Type	IDE Plugin
Input	Mapping DSL
Output	R2RML, RML or CSVW mappings

[13] https://www.w3.org/TR/2013/REC-sparql11-update-20130321/
[14] https://www.w3.org/TR/2013/REC-sparql11-http-rdf-update-20130321/
[15] https://help.metaphacts.com/resource/Help:Apps

Development / Deployment

Independent services and deployable units are typically deployed in a containerized format, i.e. using Docker containers and docker-compose or in an Kubernetes environment.

The configuration is spread over all services comprising the overall solution. Ideally, the overall orchestration configuration can be externalized in parameterizable deployment templates such as docker-compose files and Kubernetes deployment descriptors specified in YAML or JSON format or as Helm[16] templates. A library of configuration templates can be used to assemble an overall solution description and deployment configuration out of smaller pieces which can be combined according to the target use case.

Method	docker-compose
Functionality	Orchestration of independent services using a standardized format
Type	docker-compose[17] templates and environment description files
Input	Description of artefacts and infrastructure configuration options
Output	Running services

Method	Kubernetes deployment descriptors
Functionality	Orchestration of independent services using a standardized format
Type	Kubernetes deployment templates[18] and environment description files
Input	Description of artefacts and infrastructure configuration options
Output	Running services

Conclusions and future work

The components and modules described in this document serve as building blocks which can be used in various scenarios to implement the user stories and use cases described in *Deliverable 1.3: Requirements specification*.

[16] https://helm.sh/
[17] https://docs.docker.com/compose/compose-file/
[18] https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.18/

The component interfaces are only listed with high-level specifications of inputs and outputs. A more detailed interface description will be provided as part of the actual work packages implementing the components.
The components will be integrated as part of Work Package 6 to evaluate the use as described in the use cases and user stories.