

Hands-on: Semantic Parsing-based QA using Qanary

Session Plan

- ❑ Part I: Qanary Foundations
 - ❑ Idea
 - ❑ Knowledge Representation using the qa Vocabulary
 - ❑ Qanary Methodology and Technical Framework
- ❑ Part II: Qanary Hands-on
 - ❑ Qanary QA components
 - ❑ QA Pipeline with Qanary
 - ❑ Execution and Validation of results
- ❑ Conclusions

Today's Goals

❑ You will...

- ❑ Get an overview of the Qanary methodology, the RDF vocabulary qa and the component-oriented Qanary framework
- ❑ Learn how a QA system can be built and executed using the Qanary framework

❑ Thereafter, you will...

- ❑ Run a QA pipeline using the Qanary framework
- ❑ Test several QA components and a QA pipeline with example questions

Part I: *Qanary* Foundations

Qanary vocabulary

Several common tasks required across QA systems (NLP, IR, NED, NER, ...).

Idea: share the information via a vocabulary can help to develop better QA systems and create new features on-top of it.

A framework for developing QA system by integrating various component using a pre-defined **Question Answering** vocabulary.

Goals and Impact

Qanary framework is providing you benefits aiming at rapid engineering → increase *your* focus on *your* research

- + Interoperable infrastructure
- + Exchangeable components
- + Isolation of components
- + Flexible granularity

Easy-to-build QA systems on-top of reusable components

Establish an **ecosystem** of components for QA systems

Knowledge Representation using the qa Vocabulary

Knowledge perspective

Idea: Store all the information you know (knowledge) about a question in a knowledge graph

Requirements of Knowledge perspective

- ❑ abstract knowledge representation: qa vocabulary
- ❑ align the input/output of the each component in a QA process

Abstract Knowledge Representation (KR)

Represent all the knowledge about a question using a RDF vocabulary

- ❑ Resource Description Framework (RDF)
- ❑ Web Annotation Data Model (WADM)
 - ❑ <https://www.w3.org/TR/annotation-model/>
- ❑ Question Answering vocabulary (qa)
 - ❑ custom semantics as common ground for QA community

Web Annotation Data Model (WADM)

- ❑ `oa:Annotation`
- ❑ `oa:hasTarget`
- ❑ `oa:hasBody`
- ❑ `oa:annotatedAt`
- ❑ `oa:annotatedBy`

```
<myIRI> a oa:Annotation;  
        oa:hasTarget <questionIRI> ;  
        oa:hasBody <TextSelector> ;  
        oa:annotatedBy  
<DBpediaSpotlight> ;  
        oa:annotatedAt "...^^xsd:date ;
```

QA Vocabulary

Introducing new QA-related concepts on-top of WADM

`qa:Question`

`rdfs:subClassOf oa:Annotation.`

`qa:Answer, . . .`

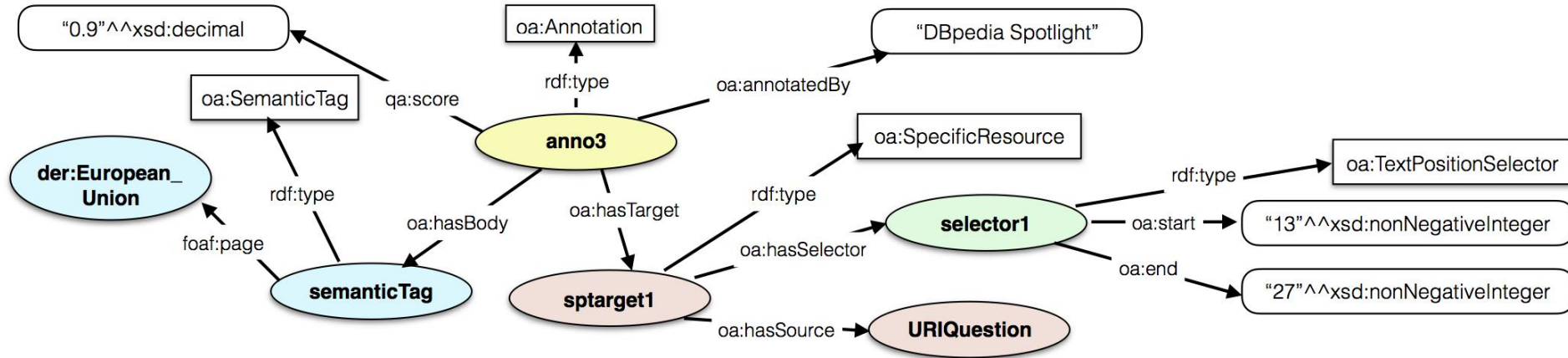
`qa:Dataset, . . .`

`qa:AnnotationQuestion, . . .`

QA Vocabulary

Idea: Use knowledge graphs as representation for the question

Example: “Where was the **European Union** founded?”



Qanary Methodology and Technical Framework

Qanary Architecture

Task: Create a Question Answering System capable of analyzing the natural language questions.



- ✓ flexible
- ✓ adaptable
- ✓ microservices-based
- ✓ distributed
- ✓ domain-specific

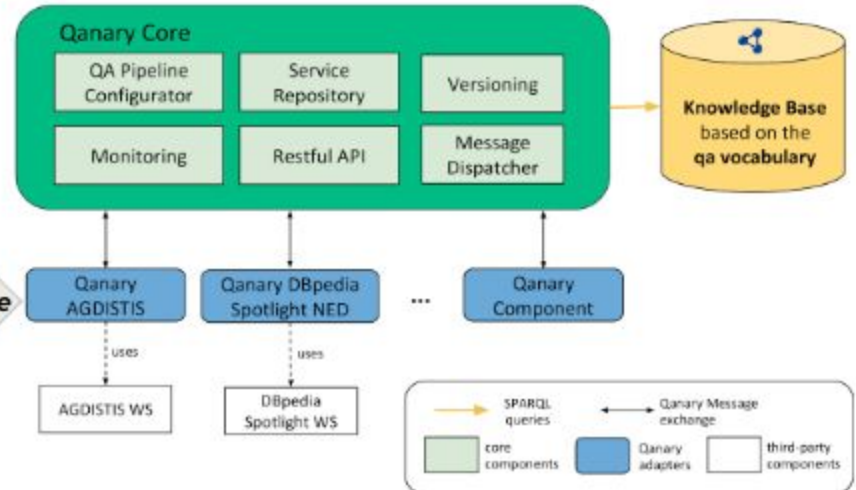
Name the municipality of Roberto Clemente Bridge

Each component creates annotations of the textual question while analyzing previously computed data.

DBpedia property
municipality

DBpedia resource
Roberto Clemente Bridge

Message-oriented Architecture for Vocabulary-driven Open Question Answering Systems



It's about the components, stupid

An agile QA framework can only provide common features

❑ e.g., central data access, tracing, logging, ...

Idea: problem solving/algorithm needs to be separated from the pipeline → create exchangeable, isolated components only communicating via data.

Component data needs to be mapped/aligned to the data of the QA process.

Running Example

- There are 30+ components integrated in Qanary.
- There are many components in Qanary, we choose AmbiverseNED (Named Entity Disambiguation), OKBQA Property Identifier (Relation Linking), and Query Builder for the query building functionality.
- Let us take the question “***Name the municipality of Roberto Clemente Bridge.***” as running example.
 - Note: Example is [not answered by Google](#).

Running Example

*Name the **municipality** of **Roberto Clemente Bridge**.*

- NED component is expected to identify **“Roberto Clemente Bridge”** as entity and link it to **dbr:Roberto_Clemente_Bridge** in DBpedia
 - http://dbpedia.org/resource/Roberto_Clemente_Bridge
 - https://en.wikipedia.org/wiki/Roberto_Clemente_Bridge
- relation linking component is expected to identify **“municipality of”** as relation, and link it to **dbo:municipality**.
 - <http://dbpedia.org/ontology/municipality>

Running Example

*Name the **municipality** of **Roberto Clemente Bridge**.*

- Goal: Query builder has to build the SPARQL query:
SELECT DISTINCT ?uri WHERE {
 <http://dbpedia.org/resource/Roberto_Clemente_Bridge>
 <http://dbpedia.org/ontology/municipality> ?uri
}
- This query returns the correct answer:
dbr:pennsylvania
 - c.f., http://dbpedia.org/resource/Roberto_Clemente_Bridge

Part II: Qanary Hands-on

Requirements

- Internet Connection
- [Git client](#)
- Java and Maven (Java 8 or higher && Maven 3+)
- [Stardog triplestore](#) (free version)

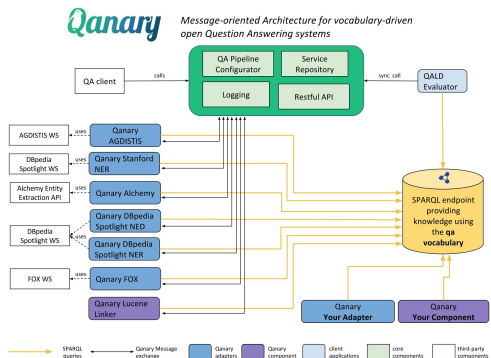
Start stardog and create db

- We use Stardog as local triplestore to store all the output, and inputs in QA process. So start Stardog by going into the “bin” folder of Stardog folder, and run:

```
stardog-admin server start
```

- After the installation is complete create a database with name “qanary”. This database stores all the data of the QA system. The command is:

```
stardog-admin db create -n qanary
```



Start stardog and create db

- Once Stardog is running, to check if its working, now log in to the Triple Store. For this, we need username and password. For the same, use (admin/admin).

Build Qanary Framework

- Clone the project from Git:

```
git clone https://github.com/WDAqua/Qanary
```

Then execute maven build of the root project folder:

```
cd Qanary
```

```
mvn install -DskipDockerBuild -P tutorial
```

The install goal will compile, test, and package your project's code and then copy it into the local dependency repository.

The above command build the project without generating corresponding Docker containers. Done here due to time restrictions.

Start the Qanary pipeline

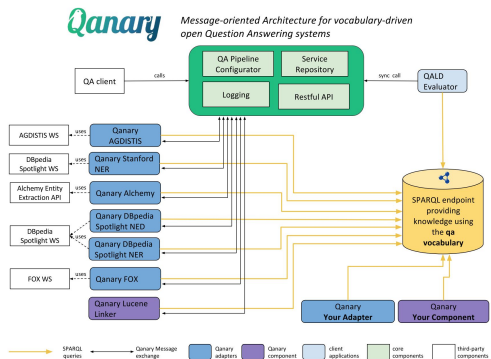
Now to start the QA process, first step is to run the main QA pipeline. The dedicated component is named:

```
qanary_pipeline-template
```

Run the following command to start Qanary pipeline:

```
cd Qanary
java -jar qanary_pipeline-template/target/qa.pipeline-1.1.2.jar
```

The pipeline was started as server so keep this terminal and do not close it.



Qanary components

Open a terminal and go to your project folder. Clone the repository of Qanary question answering components:

```
git clone
```

```
https://github.com/WDAqua/Qanary-question-answering-components
```

35+ components are now available locally.

Due to time restrictions we are just building now the three required for finishing the tutorial:

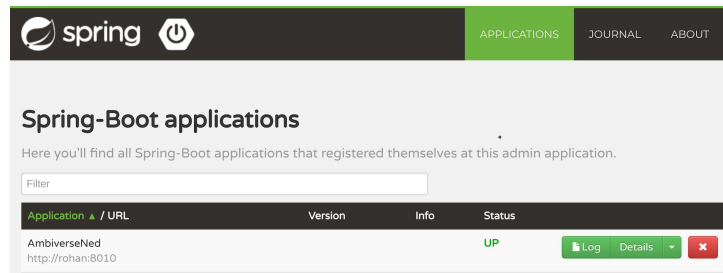
```
mvn install -DskipDockerBuild -P tutorial
```

Run NED component

Open a new terminal window, go to the project root folder, and start the JAR of the Ambiverse NED component:

```
java -jar  
qanary_component-NED-Ambiverse/target/qanary_component-NED-Ambivers  
e-1.1.2.jar
```

If the server is running, then the component will register itself to the Qanary server. You can check this by calling the Web-based Admin interface (of the integrated Spring Boot Admin Server): <http://127.0.0.1:8080/>



Run Relation Linking component

Start the Relation Linking component named DisambiguationProperty-OKBQA again from the project root folder (open a new terminal window):

```
java -jar  
qa.qanary_component-DiambiguationProperty-OKBQA/target/qa.qanary_co  
mponent-DiambiguationProperty-OKBQA-1.1.2.jar
```

Run Query Builder component

Start the created component “Query Builder” using following the command in your project root folder (same as previous components):

```
java -jar  
qa.qanary_component-QueryBuilder/target/qa.qanary_component-QueryBu  
ilder-1.1.2.jar
```

Run a QA pipeline

- Now go to our trivial interface for testing the functionality <http://localhost:8080/startquestionansweringwithtextquestion>
- you will see the three components you have ran in the previous steps appearing in a list
- Select the components and press the button :



start QA process provided by Qanary ☐

Run a QA pipeline

- On your screen you will see something like:
- {“endpoint:<http://admin:admin#>.....,
“ingraph”:”**urn:graph:XXXX**”},
- You will see the ingraph: **urn:graph:XXXX** on screen after executing pipeline, and copy this ingraph number (XXXX) in this case.

Check results in the qanary db

- Now to see the final results, We need to login to Stardog's qanary database where we have stored the data.
- Login to Stardog again using admin/admin credentials.
- There is a button "Database". Press this button. Now you see the database named "qanary".
- Roll your cursor on it, and select this database.










Check results in the qanary db

- Query the qanary database. For this, there is a “Query” button. Press this. You will see a placeholder to write the SPARQL query. There, type this query and press execute.

```
select * from <paste your ingraph> where { ?s ?p ?o. }
```

Scroll down. You will see the RDF triples as results (on next slide).

Check results in the qanary db

 http://www.w3.org/ns/openannotation/core/hasBody	<pre>SELECT DISTINCT ?uri WHERE { <http://dbpedia.org/resource/Roberto_Clemente_Bridge> <http://dbpedia.org/ontology/municipality> ?uri }</pre>
 http://www.w3.org/ns/openannotation/core/annotatedBy	 urn:qanary:geosparqlgenerator
 http://www.w3.org/ns/openannotation/core/AnnotatedAt	2018-02-11T16:45:37.570+01:00
 rdf:type	 http://www.wdaqua.eu/qa#AnnotationOfAnswerJSON
 http://www.w3.org/ns/openannotation/core/hasTarget	 file:///qanarySetup/Applications/workspace/qanary_qa/URIAnswer
 http://www.w3.org/ns/openannotation/core/hasBody	<pre>{ "head": { "vars": ["uri"] }, "results": { "bindings": [{ "uri": { "type": "uri" , "value": "http://dbpedia.org/resource/Pittsburgh,_Pennsylvania" } }] } }</pre>

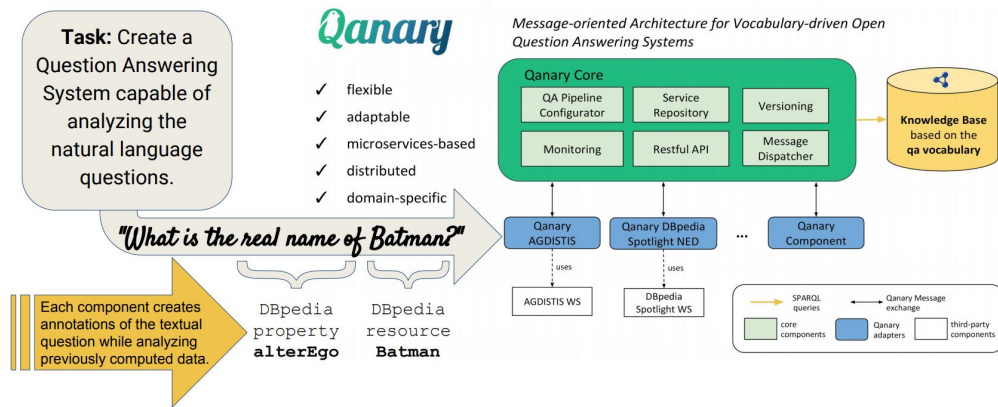
Final Remarks

Summary

Qanary methodology, the RDF vocabulary qa and the corresponding component-oriented Qanary framework.

Advantage of the Qanary ecosystem for rapid research results while provide you functionality from previous research.

Learned to iteratively build, validate and improve your own QA system using the Qanary framework.



Qanary QA System



Join us on GitHub:

<http://github.com/WDAqua/Qanary>

See WDAqua project:

<http://wdaqua.eu/>

- provides you with the methodology to create rapidly QA systems
→ helps you to create your work on the shoulder of giants
- Many extensions available, but not covered here:
 - frontend: Trill (<https://github.com/WDAqua/Trill>)
 - example: <http://www.wdaqua.eu/qa>
 - automatic component composition
 - implementation, e.g.
 - QAestro (<https://github.com/WDAqua/QAestro>)
 - Frankenstein (<https://github.com/WDAqua/Frankenstein/>)
 - papers, e.g.
 - [ESWC 2016](#), [ICWE 2017](#), [TheWebConf 2018 \(WWW'18\)](#), [ESWC 2018](#)

Take Away: Qanary methodology



Join us on GitHub:

[http://github.com/
WDAqua/Qanary](http://github.com/WDAqua/Qanary)

See WDAqua project:

<http://wdaqua.eu/>

Qanary: knowledge-driven methodology for QA systems

Agile approach for creating QA systems

Build on-top of the qa vocabulary (i.e., knowledge-driven approach) using a component-based approach

→ next session: create your own “Query Builder” task solver using deep learning

Join Qanary at GitHub
github.com/WDAqua/Qanary