Rapid Engineering of QA Systems Using the Light-Weight Qanary Architecture

Andreas Both[1(✉)], Kuldeep Singh[3], Dennis Diefenbach[2], and Ioanna Lytra[3,4]

[1] DATEV eG, Nuremberg, Germany
contact@andreasboth.de
[2] Laboratoire Hubert Curien, Saint Etienne, France
[3] Fraunhofer IAIS, Sankt Augustin, Germany
[4] Enterprise Information Systems, University of Bonn, Bonn, Germany

Abstract. Establishing a Question Answering (QA) system is time consuming. One main reason is the involved fields, as solving a Question Answering task, i.e., answering a user's question with the correct fact(s), might require functionalities from different fields like information retrieval, natural language processing, and linked data. The architecture used for Qanary supports the derived need for easy collaboration on the level of QA processes. The focus of the design of Qanary was to enable rapid engineering of QA systems as same as a high flexibility of the component functionality. In this paper, we will present the engineering approach leading to re-usable components, high flexibility, and easy-to-compose QA systems.

Keywords: Software reusability · Question Answering · Light-weight web architectures · Service composition · Semantic search · Ontologies · Annotation model

1 Introduction

The Web of Data is growing permanently as well as the industrial data sets. Induced by this movement the challenge for retrieving knowledge from such data sets has gained much importance in research and industry. Question Answering (QA) is tackling this challenge by providing an easy-to-use natural language interface for retrieving knowledge from large data sets. However, as QA is a challenge requiring to solve research questions from many different fields, a QA system is mostly consisting of many different components (from different research fields). Hence, enabling easy collaboration between researchers is an important engineering path while aiming at supporting the research community. Additionally, a reasonable engineering approach is required to enable a loose cooperation of different researchers.

Earlier, we established a component-oriented approach named Qanary [1] on top of a RDF vocabulary `qa` [6]. This approach provides a methodology for

J. Cabot et al. (Eds.): ICWE 2017, LNCS 10360, pp. 544–548, 2017.
DOI: 10.1007/978-3-319-60131-1_40

creating QA processes using a central knowledge base (KB) to store all available QA process data. Here, we will focus on the component model and service composition following the Qanary methodology. In the demonstration, we will use the Qanary reference implementation to show the achievement w.r.t. to the rapid engineering process that was established. We will show the engineering process for creating a Qanary Web service as well as a complete Qanary-based QA system.

2 Related Work

In the context of QA, a large number of systems and frameworks have been developed in the last years. For example, more than 20 QA systems (in the last 5 years) were evaluated against the QALD benchmark (cf., http://qald.sebastianwalter.org). These reasons led to the idea of developing component-based frameworks that make parts of QA systems reusable. We are aware of three frameworks that attempt to provide a reusable architecture for QA systems. QALL-ME [4] provides a reusable architecture skeleton for building multilingual QA systems. openQA [5] provides a mechanism to combine different QA systems and evaluate their performance using the QALD-3 benchmark. The Open Knowledge Base and Question Answering (OKBQA) challenge (cf., http://www.okbqa.org/) is a community effort to develop a QA system that defines rigid JSON interfaces between the components. In contrast, Qanary [1] does not propose a rigid skeleton for QA pipelines, instead we allow multiple levels of granularity, enable the community to develop new types of QA systems (not only pipelines), and focus on the research tasks.

3 The Qanary Component Engineering Process

Requirements. The core requirements of the Qanary architecture are: R1 programming language independent approach, R2 combining of components to different QA processes as easy as possible (no predefinition of specific pattern, e.g., QA pipeline), and R3 enabling the researches from different communities to follow their own research tasks with as few as possible restrictions.

The Qanary Component Model. Each Qanary component is an independent Web service implementing the tiny RESTful interface: `process(M)`. Via the synchronous interface the component is triggered to process the current user question. The question (and any process data) is not contained in the message `M`, instead it was stored in an RDF KB. Consequently, $\texttt{M} = (\texttt{T}, \texttt{G}_i, \texttt{G}_o)$ contains the endpoint URI of the KB `T`, and the graph $\texttt{G}_i$ in `T` containing the inbound information as well as the graph $\texttt{G}_o$ in `T` that should be used to store the computed information (i.e., outbound data flow) for further use in the QA process (by other components). Finally, the component is returning the focus to the QA process where other QA components might be called which can use the generated data. Hence, after being notified by the QA process a component will

fetch the information required for its task from G_i (in T) and perform its task using this information, cf., Fig. 1. To enable an easy data exchange on common ground, the RDF vocabulary qa [6] was established (built on top of the W3C WADM, cf., w3.org/TR/annotation-model) holding the computed information as annotations of the question. Therefore, within the process the computed data can be interpreted by any Qanary component. Note that all information stored in G_i is retrievable by each Qanary component. Hence, no restrictions w.r.t. the accessible data are imposed (cf., R3).

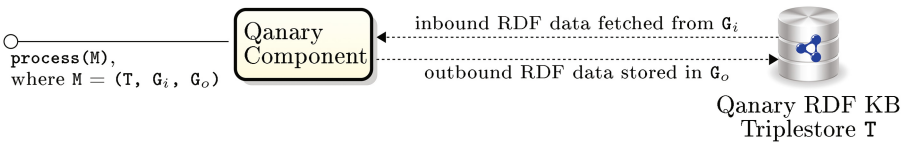

Fig. 1. Qanary component model (note that G_i might be equal to G_o).

Service Composition. All Qanary components implement the same lightweight interface and retrieve/store the data using the qa vocabulary. Hence, the Qanary services can be integrated by combining these components, analogously to the Pipes and Filters [2] architecture pattern. The Qanary reference implementation (cf., github.com/WDAqua/Qanary→qanary_pipeline-template) takes advantage of the characteristics of Qanary components. It contains a service registry (AdminServer) which is called automatically during the start-up phase by all components. Hence, all Qanary components are known and can be easily composed (cf., R2), e.g., the following simple user interfaces are provided to create a QA pipeline using a textual or audio question. There components can easily be activated and combined by drag and drop (define order in QA pipeline):

http://www.wdaqua.eu/qanary/startquestionansweringwithtextquestion,
http://www.wdaqua.eu/qanary/startquestionansweringwithaudioquestion

Service Implementation. The implementation of a Qanary component is supported using a Maven archetype (cf., github.com/WDAqua/Qanary→qanary-component-archetype). It already contains the registration to the AdminServer and several other functionalities for rapid engineering. Note: There are no restrictions on the functionality nor the programming language (cf., R1); however, the reference implementation is in Java.

Demonstration. As an example, we show how to create a QA pipeline providing the functionality focusing on the engineering tasks. The pipeline is aiming at answering the question "What is the real name of Batman?"[1] (cf., QALD

[1] Full description at https://github.com/WDAqua/Qanary/wiki/ICWE-2017-demo.

question no. 92). It will use a component that already exists in the Qanary ecosystem providing functionality for Named Entity Recognition and Disambiguation (NER/NED), e.g., the Qanary DBpedia Spotlight component (cf., [3]). It will interlink the sub-string "Batman" to the DBpedia resource `dbr:Batman`. However, additional semantics is required to map the textual question to an interpretable representation. Therefore, we will interactively implement a new component C (using Qanary's Maven archetype) which adds new annotations to the Qanary KB `T` while analyzing the user's question. C will serve only the purpose to identify the relation `dbp:alterEgo` (i.e., a DBpedia property) while searching for the sub-string "real name" in the question.

The demonstration will finish while creating and executing the QA pipeline using the service composition and showing the result of the question.

Discussion. Here, we have demonstrated the main advantage a developers receives while integrating a component in the Qanary ecosystem. A rapid engineering process is provided and a created component can easily be interweaved with the already existing ones. A basic installation of a QA pipeline provided with a user interface called Trill can be found at http://www.wdaqua.eu/qa.

4 Conclusion

In this paper we presented the component model of the reference implementation of the Qanary framework. Qanary components are easy to implement as it was shown in the paper. However, one of the core features is the option to (re)combine components to QA systems without adopting the component's source code, while still having the full freedom of dedicating a (new) component to a completely new functionality. This new functionality might use data from the Qanary triplestore never used before in this particular combination. Hence, as all features are data-driven, allowing to add new functionality to the whole QA system from a local component independently. Additionally, the component model is language-independent and driven by the power of linked data which enables additional features like polymorph data types included in the inbound data. Our main contribution is a component-based architecture enabling developers to create or re-combine components following a plug-and-play approach. While aiming at an optimal system w.r.t. a given use case (scientific) developers are enabled to rapidly create new/adapted QA systems from the set of Qanary components available. Hence, we are handing the scientific QA community an easy-to-use approach reducing the investments for engineering tasks during typical tasks.

Acknowledgments. This project has received funding from the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement No 642795.

References

1. Both, A., Diefenbach, D., Singh, K., Shekarpour, S., Cherix, D., Lange, C.: Qanary – a methodology for vocabulary-driven open Question Answering systems. In: Sack, H., Blomqvist, E., d'Aquin, M., Ghidini, C., Ponzetto, S.P., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9678, pp. 625–641. Springer, Cham (2016). doi:10.1007/978-3-319-34129-3_38
2. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture - Volume 1: A System of Patterns. Wiley, New York (1996)
3. Diefenbach, D., Singh, K., Both, A., Cherix, D., Lange, C., Auer, S.: The Qanary ecosystem: getting new insights by composing Question Answering pipelines. In: International Conference on Web Engineering, ICWE. Springer (2017)
4. Ferrández, Ó., Spurk, Ch., Kouylekov, M., Dornescu, I., Ferrández, S., Negri, M., Izquierdo, R., Tomás, D., Orasan, C., Neumann, G., Magnini, B., González, J.L.V.: The QALL-ME framework: a specifiable-domain multilingual Question Answering architecture. J. Web Seman. Sci. Serv. Agents WWW **9**, 137–145 (2011)
5. Marx, E., Usbeck, R., Ngonga Ngomo, A., Höffner, K., Lehmann, J., Auer, S.: Towards an open Question Answering architecture. In: SEMANTiCS (2014)
6. Singh, K., Both, A., Diefenbach, D., Shekarpour, S.: Towards a message-driven vocabulary for promoting the interoperability of Question Answering systems. In: IEEE International Conference on Semantic Computing, ICSC (2016)