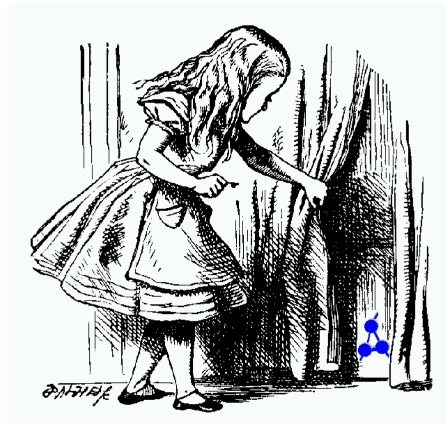
KAON SERVER Prototype

Daniel Oberle[1], Raphael Volz[1,2], Boris Motik[2], Steffen Staab[1]

[1]University of Karlsruhe
Institute AIFB
D-76128 Karlsruhe
email: {lastname}@aifb.uni-karlsruhe.de

[2]FZI - Research Center for Information Technologies
Haid-und-Neu-Strasse 10-14
D-76131 Karlsruhe
email: {lastname}@fzi.de



Identifier Del 6

Class Deliverable

Version 1.1

Date 01-13-2002

Status Final

Distribution Public

Lead Partner AIFB

WonderWeb Project

This document forms part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-2001-33052.

For further information about WonderWeb, please contact the project co-ordinator:

Ian Horrocks
The Victoria University of Manchester
Department of Computer Science
Kilburn Building
Oxford Road
Manchester M13 9PL
Tel: +44 161 275 6154
Fax: +44 161 275 6236
Email: wonderweb-info@lists.man.ac.uk

Contents

1 Introduction **1**

2 Architectural Overview **4**

3 Components **6**
3.1 RDF Mainmemory Implementation 7
3.2 RDF Server 8
3.3 KAON API on RDF API 9
3.4 Engineering Server 9
3.5 Integration Engine 10
3.6 External Services 10

4 Management **10**
4.1 Kernel 11
4.1.1 The JBoss JMX implementation 11
4.1.2 Management usage scenarios 13
4.2 Registry 15
4.3 Interceptors 16

5 Data Access **17**
5.1 RDF API 17
5.2 Ontology API 19

6 Security **20**

7 Connectors **21**

8 Scenario **23**

9 Related Work, Conclusion and Outlook **26**

1 Introduction

With the Semantic Web, computer science is progressing from isolated islands of data repositories towards a web of data, which functions in many ways like a global database. In database research and practice, however, there is now a clear separation between the database proper and the database management system. While the former describes the data itself and its structure the latter captures all the functionality required to actually store, modify and extract data.

In the Semantic Web, a system that constitutes the analogon to a full-fledged database management system is completely lacking so far. In a way, this situation is unavoidable as the functionality of such a system is far from being well understood at the current point in time with many theoretical foundations concerning such a system currently being under research, such as views [17], evolution [15], or versioning [8] to name but three[1].

Nevertheless, instead of a flock of tools that only interoperate by reading and writing data into files in some Semantic Web format, there is an immediate need for a Semantic Web Management System that

- provides interoperability between Semantic Web components;
- is flexibly configurable; and
- is open to new developments in theory and implementation.

Our Semantic Web Management System **KAON SERVER** is developed and implemented in order to respond to this need. It will complement the static part of the Semantic Web layer cake by managing components that cover the dynamic aspects of the Semantic Web (cf. Figure 1). The principal concept of KAON SERVER is developed in a way that it may easily accommodate minor changes in the structure of the Semantic Web layer cake or in the dynamic requirements. Additionally, it will offer the integration of tools that support different individual layers of the layer cake. KAON SERVER also will support to channel information between these components and to coordinate the information flow. The server basically realizes

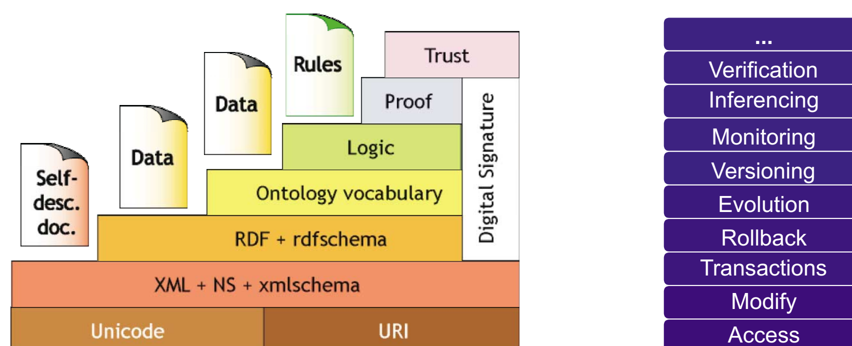


Figure 1: Static and dynamic aspects of the Semantic Web layer cake

[1]None of the three mentioned approaches solves its problem for the full Semantic Web data layer cake.

- *Component Management*, which is able to
  - load and unload components on the fly as needed,
  - register and manage the components,
  - provide communication between components through a flexible event mechanism.

Unlike in classical database management systems, not a single data model is employed for the Semantic Web (cf. Figure 1). Instead, several data models are used, some of them are even unspecified by now, e.g. the data models employed to implement the logic, proof and trust layers. At first glance, this appears not to be of any problem since all data models rely on the same meta-grammar, namely XML. However, any practical usage requires to talk about the specific data model addressed, e.g. when trying to implement transactional behavior or security. Almost all functionality in the domain of the management system shows such a dependency. Hence, all those characteristics directly lead to the need of an open, flexible and extensible architecture, such as shown prototypically for KAON SERVER. The development of the server is therefore accompanied by a

- *Semantic Web Data API*, which
  - allows for dynamic instantiations by components that extend the functionality of the core KAON SERVER modules towards a full-fledged Semantic Web Management System.
  - answers to the needs put forward by the static part of the Semantic Web layer cake, viz. the definitions of Semantic Web languages.

In addition to the extensibility, the interoperation of components must be ensured since upper layers often rely on functionality specified and provided by lower layers, e.g. data typing for RDF and OWL taken from XML Schema. However, due to the inherent complexity no single server was yet able to support everything simultaneously. In order to build a reasonably complete system to support the Semantic Web one must draw functionalities on existing components and must be able to include and manage them.

This deliverable presents a first prototype of our Semantic Web Management System, KAON SERVER. It is developed in context of the Karlsruhe Ontology and Semantic Web tool suite (KAON, http://kaon.semanticweb.org, [3]) which is an open-source project and joint effort by both the Institute AIFB[2] and the Research Center for Information Technologies (FZI)[3]. While the development of the KAON SERVER is funded by WonderWeb, some parts of the KAON tool suite are developed in other projects: SWAP - Semantic Web and Peer-to-Peer, OntoWeb - Ontology based information exchange for knowledge management and electronic commerce, OntoLogging - Corporate Ontology Modelling and Management System, SWWS - Semantic Web Enabled Web Services. Sourcecode and executable prototype can be downloaded at http://kaon.semanticweb.org.

[2] AIFB, http://www.aifb.uni-karlsruhe.de/WBS

[3] FZI, http://www.fzi.de

In order to give a more concrete motivation for the need of a Semantic Web Management System, we present the following scenario which is further elaborated in section 8. Imagine a simple genealogy application. Apparently, the domain description, viz. the ontology, will include concepts that talk about Persons and make a distinction between Males and Females. An ontology editor like OILed [1] or OntoEdit [16] may be in need of an XML processor in order to validate the XML schema datatypes of some attributes (e.g. birthday being of type `xsi:dateTime`). It may also be in need of an RDF Store to finally save the ontology along its instances. On the other hand, imagine a genealogy portal relying on that particular ontology as a conceptual backbone. To present the contents of the RDF Store, the capabilities of a description logic reasoner would be of advantage. Implicit facts could be deduced by utilizing a rule-based reasoner and would give some value-addition to the portal application. To implement the system, all the required components, i.e. a rule-based inference engine, a DL reasoner, a XML Schema data type component etc., would have to be combined manually using proprietary code. While this is a doable effort, the system would be highly proprietary and not reusable across the domain.

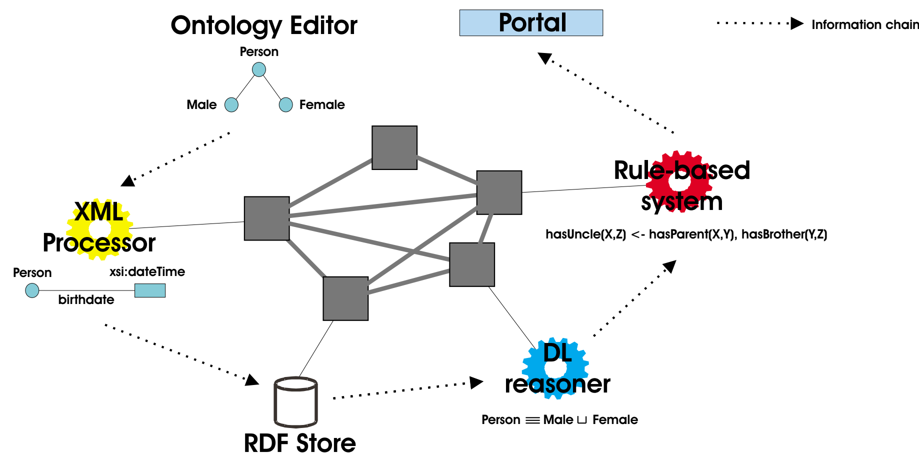


Figure 2: Scenario scheme

Apparently, the demands to a management system from this application is to hook up to all those components and to offer management of data flow between them. This also involves propagation of updates and rollback behavior, if any component in the information chain breaks.

The document is structured as follows: First, we provide an architectural overview in section 2 which conceptually divides the functionality into five layers. Those layers, viz. Functional Components, Management, Data Access, Security and Connectors Layer, are discussed in section 3 to 7, respectively. Therein, the focus always lies on the components already implemented. The remaining components are described as detailed as possible at this point in time. After all technical details we decided to elaborate on the scenario which should ease the overall understanding in section 8. Related work, conclusion and outlook are presented in section 9.

2 Architectural Overview

A first architectural overview of the KAON SERVER is depicted in Figure 3. We follow the well-known principle of layering functionalities such as known from the Internet protocol stack or from database architectures. That resulted in the Connectors Layer, the Security Layer, Data Access Layer, Management Layer as well as Functional Components, where layers featuring + are a mandatory part of KAON SERVER, all the others are optional (*).

When a client first connects to the KAON SERVER it will typically query for the components it is in need of. That could be an XML processor, an RDF store, an inference engine and so on. The client should be enabled to state precisely what it wants, e.g. an RDF store that holds a certain RDF model and allows for transactions. The KAON SERVER tries to find a registered component fulfilling the stated requirements and returns a reference. From now on, the client can seamlessly work with that component, as if it were locally instantiated.

Requests sent to that component are then processed in a top down manner. First, they are embraced by a connector, as the client may reside in another process than the KAON SERVER. After that, the properness of a request is checked by some security module that may deal with authentication, authorization, auditing or even trust in the future. Mechanisms like those are realized by so-called interceptors, which are situated in the Management Layer. Almost all requests will deal with data querying or data update. For this purpose, we defined two Semantic Web Data APIs, viz. RDF API as well as an ontological API, as mentioned in the introduction. Note, that those are purely interfaces which may be implemented differently by functional components residing in the respective layer. In interaction with the registry, the management kernel tries to find a functional component that is capable of finally computing the request and, if successful, eventually delegates the request there. Hence, all functional components, like RDF Stores, inference engines and so on are aggregated in the lowest layer. Bringing it to a one-sentenced definition:

KAON SERVER can be considered as comprehensive and sophisticated software entity enabling the management of components.

As the definition puts it, the KAON SERVER basically realizes a sophisticated component management. In order to make use of such a software entity, all functionality that shall be managed has to be brought into a certain form (in the following paraphrased as *to make existing software deployable*). Another drawback is that performance will suffer slightly in comparison to stand alone use, as a request has to pass the layer and interceptor stacks first. But on the other hand, one gains a lot of benefits from component management. By bringing existing functionality, like RDF stores, inference engines etc., into a uniform framework, we are able to

- start and stop components ad lib, reconfigure, monitor and possibly distribute them dynamically
- easily realize security, auditing, trust etc. as interceptors

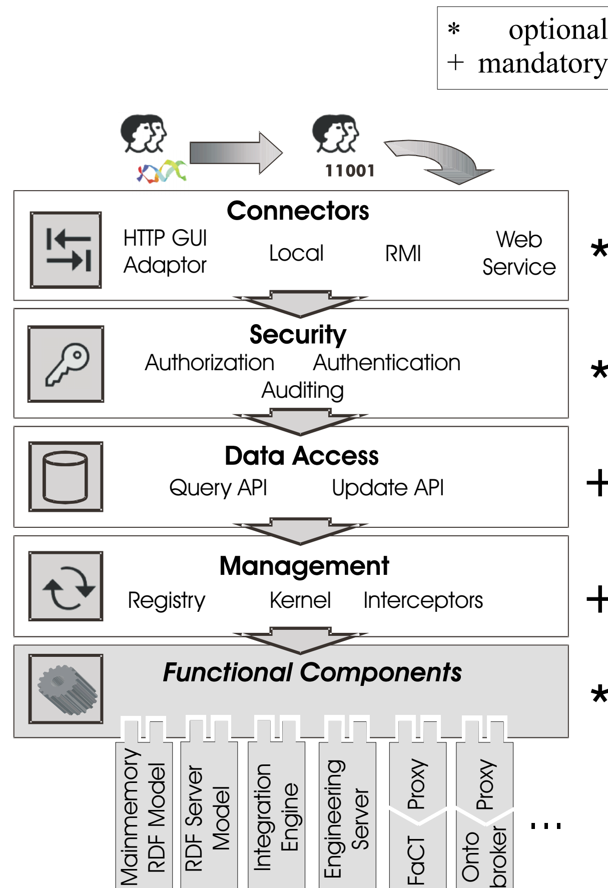


Figure 3: KAON SERVER Layers

- pose powerful queries for components a client is in need of as all of them will be registered with certain attributes
- incorporate information quality in a uniform way along the registry
- define dependencies between components and
- allow for a flexible event mechanism among them

The prototype on which this deliverable is based already features the components listed below. Each of the five layers will be discussed separately and bottom-up in the following, ranging from section 3 to 7, with the focus on the components realized within the prototype. The remaining are described as far as possible at this point in time.

- the entire Connectors Layer, i.e. HTTP GUI Adaptor, local, RMI and SOAP Connectors is already set up.
- Semantic Web Data APIs in the Data Access Layer, viz. RDF API and ontological API, are realized, yet without OWL support.

- the Kernel of the Management Layer, i.e. the JBoss JMX implementation, is already there
- and finally a first component has been realized - a concurrent main memory based RDF model which is capable of transactions.

3 Components

We will start our detailed description at the lowest layer which is populated by so-called functional components. It is crucial to be aware of the terminology we establish here. First, the process of deployment is defined, followed by the taxonomy of components.

Deployment Process of registering, possibly initializing and starting a component to the management kernel.

Component The entity of management which is deployed to the kernel. Technically speaking, a component is an MBean, i.e. a managed JavaBean, as we will learn in section 4.

System Component Component providing server logic, e.g. the registry or a connector.

Functional Component Component that is of interest to the client and can be queried for, e.g. an RDF Store.

Proxy Component Equals a functional component from a client perspective. Instead of providing the actual functionality, it manages the communication to an external service.

External Service An external service cannot be deployed and asked directly by the KAON SERVER as it may be programmed in a different language, live on a different computing platform, use interfaces unknown to the KAON SERVER, etc. Examples are inference engines, like FaCT.

According to the definitions above and also to the other deliverables of the project, we distinguish between functional components and external services in the following subsections.

All the KAON tools will be functional components in the end. Two Semantic Web Data APIs for updates and queries are defined in the KAON framework which basically populate the Data Access Layer - the RDF API and an ontological interface called KAON API (also cf. section 5). The different tools implement those APIs in different ways and will all be made deployable. One of them, namely the main-memory-based implementation of the RDF API, has already been adapted for the prototype. Figure 4 depicts an overview of the implementations. All of them are discussed in subsections 3.1 to 3.5.

On the other hand, several external services have to be made deployable as well, i.e. proxy components have to be written. Among them inference engines like FaCT, Ontobroker and Triple. Subsection 3.6 briefly talks about them. None of them have been adapted so far.

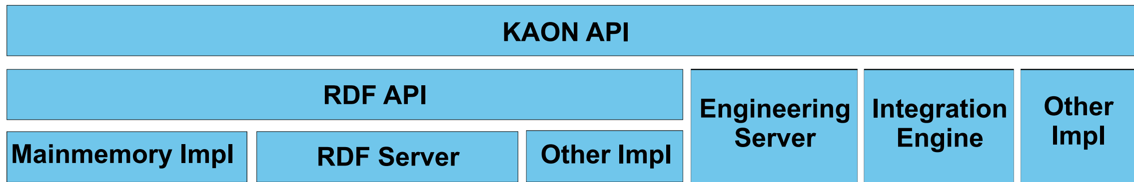


Figure 4: KAON API Implementations

3.1 RDF Mainmemory Implementation

The so-called Stanford implementation of the RDF-API [10] is primarily useful for accessing in-memory RDF models. That means, an RDF model is loaded into memory from an XML serialization on startup. After that, statements can be added, changed and deleted, all encapsulated in a transaction if preferred. Finally, the in-memory RDF model has to be serialized again.

For the prototype, we have wrapped this implementation as an MBean in order to make it deployable to the KAON SERVER. That resulted in the class `RDFComponent`, like discussed below. In the following, we highlight the details of the Java classes and interfaces that have been developed for this functional component. The reader may refer to the description of the RDF API in section 5 as it is utilized here. For details on MBeans please refer to section 4.

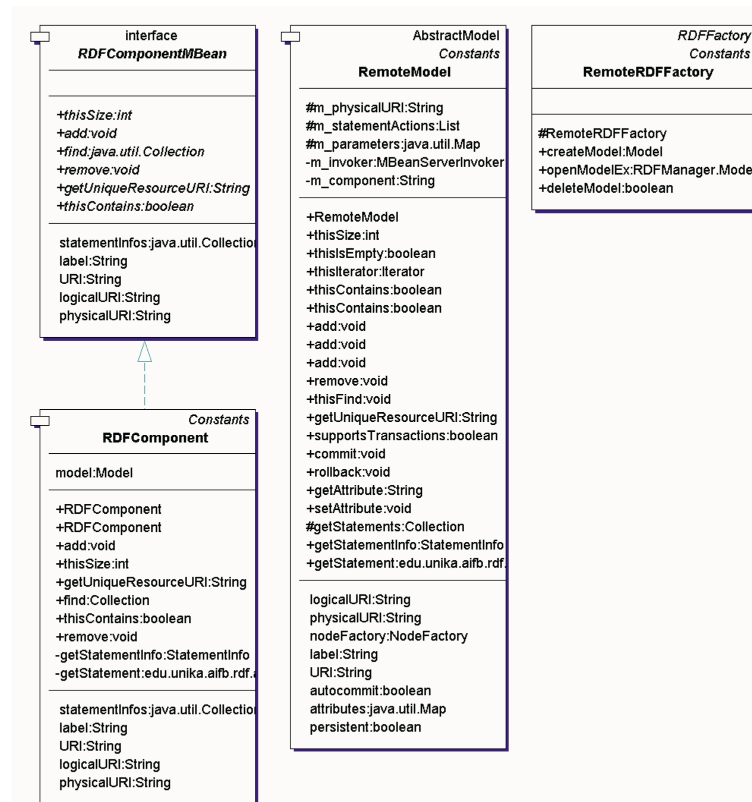


Figure 5: Interfaces and classes

Instances of `RDFComponent` are deployable to the server because they implement the interface `RDFComponentMBean` (cf. section 4 for more information on the MBean interfaces). They basically wrap main-memory based RDF models with slightly different accessor methods. It was necessary to change them because of serialization hazards. The reader may note, that a call to an `RDFComponent` can come from across the network. It would be clumsy to send comprehensive objects as parameters because they all have to be serializable. Instead, we introduced specialized parameter classes (e.g. `StatementInfo`), that make life easier for networking.

In Figure 5, one can find `RemoteModel` which is the client-side counterpart of an `RDF Component`. It extends `AbstractModel` that finally implements `Model`. The latter forms the central interface of the RDF API and is the object-oriented representation of an RDF model. It aggregates several statement-objects and the methods allows for querying and updating. The method implementations of the class `AbstractModel`, which also belongs to the RDF API, deal with modularization issues.

A client can seamlessly work with an `RemoteModel` as if it were the common main-memory based version because it implements the same interface. All method invocations are sent to the KAON SERVER behind the scenes, as the actual model resides there. Having `RemoteModel` inheriting from `AbstractModel` means that it can include other models which can be of different implementations. The reader may note, that all modularization issues happen on the client side. `RemoteModel` has to translate its method calls into those of the `RDFComponent`.

Finally, `RemoteRDFFactory` is needed for `RemoteModel`. Like explained in section 5, the `RDFManager` eases the creation and opening of models independent of the actual API implementation. In order to work properly, the `RDFManager` has to be provided an `RDFFactory` for every implementation.

3.2 RDF Server

The RDF Server is an implementation of the RDF API that enables persistent storage and management of RDF models. This solution relies on a physical structure that corresponds to the RDF model. Hence, data is represented using four tables, one represents models and the other one represents statements contained in the model (cf. Figure 6).

However, such simplistic structures are highly suboptimal for RDF, since usually a large number of joins are made to traverse object links. Hence, indexing structures are needed on all elements of a statement to offer efficient data access. This could be achieved using hashed indexes, which could provide lookup of a particular data element in constant time. However, hashtables need a good distribution on keys for their values. This is not the fact for URIs (which have mostly similar prefixes) and also not for automatically incremented IDs.

Consequently, this representation needs further relations to represent resources and literals and provides an internal identifier that exhibits good indexing properties to provide efficient indexes on the statement relation.

The RDF Server utilizes a relational DBMS and relies on the JBoss Application

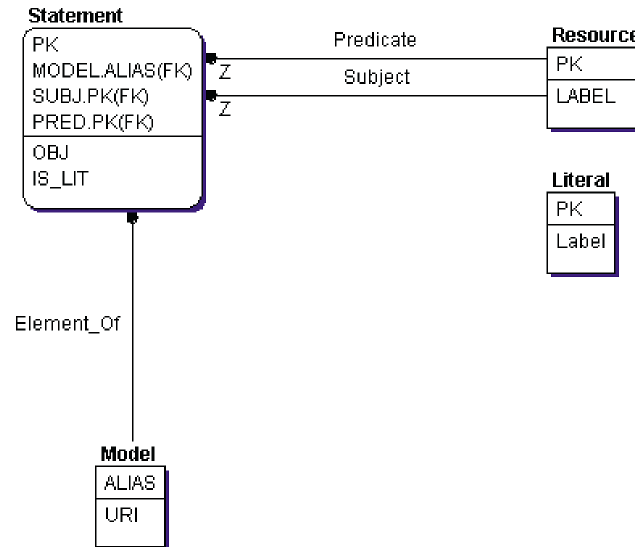


Figure 6: Schema for RDF storage

Server[4] that handles the communication between client and DBMS. The server hasn't been adapted yet but will presumably use a similar infrastructure as the main-memory variant in section 3.1

3.3 KAON API on RDF API

As depicted in Figure 4, implementations of the ontological KAON API may utilize implementations of the RDF API. None of the KAON API implementations have been made deployable so far. What will probably happen is the wrapping of a so-called OIModel as an MBean as well as a client-side counter part that handles the communication and data-transfers to such an MBean residing in the KAON SERVER. OIModel stands for Ontology-Instance-Model which is a Java interface dealing with querying and updating an ontology along its instances like further discussed in section 5.

3.4 Engineering Server

A separate implementation of the KAON API can be used for ontology engineering. This implementation provides efficient implementation of operations that are common during ontology engineering, such as concept adding and removal by applying transactions. A storage structure that is based on storing information on a metamodel level is applied here. A fixed set of relations is used, which corresponds to the structure of the used ontology language. Then individual concepts and properties are represented via tuples in the appropriate relation created for the respective meta-model element (cf. Figure 7).

This structure was not chosen before by any other RDF database, however it appears to be ideal for ontology engineering, where the number of instances (all represented in one

[4]http://www.jboss.org

table) is rather small, but the number of classes and properties dominate. Here, creation and deletion of classes and properties can be realized within transactional boundaries.

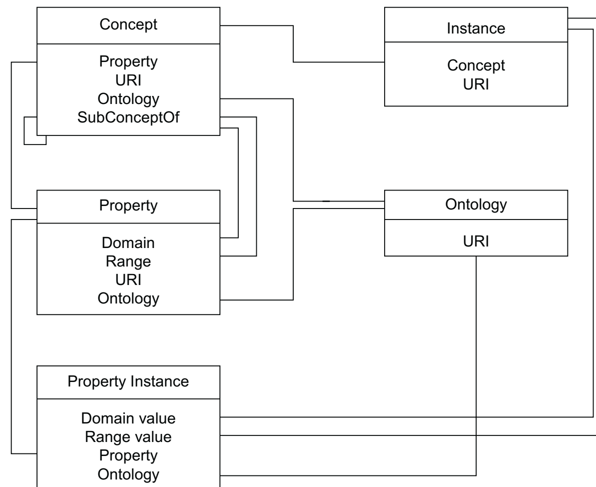


Figure 7: Schema for ontology engineering

3.5 Integration Engine

Another implementation of the KAON API can be used to lift existing databases to the ontology level. To achieve this, one must specify a set of mappings from some relational schema to the chosen ontology, according to principles described in [13]. E.g. it is possible to say that tuples of some relation make up a set of instances of some concept, and to map foreign key relationships into instance relationships.

3.6 External Services

The definitions in the beginning of this section already clarified that external services live outside the KAON SERVER. So-called proxy components have to be developed that are deployed and take care of the communication. Thus, from a client perspective, an external service cannot be distinguished from an actual functional component. The Wonderweb project sees several inference engines to be adapted - all of them are separate deliverables like listed below. None of them have been addressed so far.

- **Deliverable D8** Adaptation of the Triple inference engine [11]
- **Deliverable D9** Adaptation of the Ontobroker inference engine [6]
- **Deliverable D10** Adaptation of the FaCT inference engine [7]

4 Management

This layer is required to deal with the discovery, allocation and loading of components that are either able to compute a client request or provide server logic. It comprises a

component registry, the management kernel and so called interceptors. As this layer is the very core of the KAON SERVER, all of its components are mandatory.

Speaking in technical terms, extensibility is the strongest requirement for KAON SERVER. Therefore, the fundamental architecture follows the well-established Microkernel design pattern [4]. This pattern basically separates minimal core functionality from extended functionality. The Microkernel[5] only features a minimum of functionality that

- is required for bootstrapping the system itself
- allows the dynamic extension of the system with further components.

In our setting, the kernel of the Management Layer (cf. Figure 3) is kept very small and can be considered as a Microkernel. It basically deals with initializing, starting and stopping of components. Whether they are system or functional components makes no difference here. The kernel, the registry, which itself is a component, and the mechanism of interceptors are further described in the following subsections.

4.1 Kernel

For the kernel, we have chosen to utilize the JBoss JMX implementation whose technical details are presented in section 4.1.1. In the future, this kernel basically has to cope with two abstract usage scenarios that are highlighted in section 4.1.2.

4.1.1 The JBoss JMX implementation

The kernel exploits the Java Management Extensions (JMX), derived from the Java Dynamic Management Kit (JDMK) which is an established solution for component, application and device management for the Java platform offered by Sun Microsystems.

JMX is intended to provide tools for building distributed, Web-based, modular and dynamic solutions for managing and monitoring devices, applications and service-driven networks. It defines a universal, open API for management, and monitoring. It has already been deployed across many commercial settings, where management or monitoring were needed and is therefore the ideal platform for the construction of the management kernel. Before JMX, there was no standardized approach in the Java programming language to start, manage, monitor, and stop different software components or applications.

Technically, the JMX specification defines the instrumentation of components as MBeans (Managed JavaBeans), a so-called *agent architecture*[6] and standard components. The contract for MBeans is simple, easy to implement, and unobtrusive for managed resources, making the adoption of JMX possible also for external services. Figure 8 gives a first overview of the architecture.

Furthermore, indirection and non-typed invocation make the JMX architecture resilient to changing requirements and evolving interfaces. Components constructed as

[5] The pattern was adopted from a typical design approach for operating systems. Hence the term ”kernel”.

[6] JMX talks of agent archicture, agent level and also MBean Agent as synonym to MBeanServer, which may be confusing. We want to stress that we don’t apply the agent paradigm here.

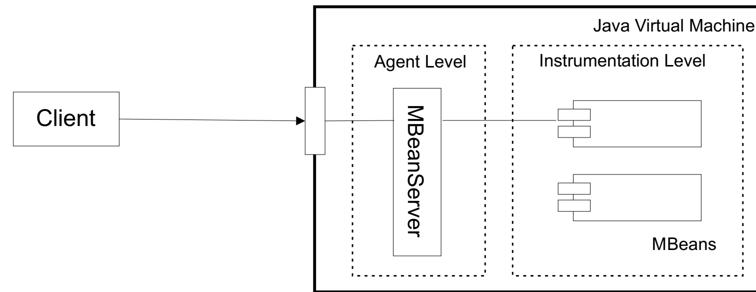


Figure 8: JMX Management Architecture

MBeans may register or unregister from the server in accordance with their respective lifecycles, and their interfaces may evolve without having to disconnect the clients.

JMX also enables the dynamic management of remote applications running on a variety of platforms, this feature can be used to manage external services. Hence, JMX is suitable for adopting existing systems by implementing new management and monitoring solutions on top of them.

Central to JMX is the so-called *MBeanServer* which acts as a simple registry for MBeans and allows management applications to discover management interfaces of the registered MBeans. The MBeanServer interface also declares the required methods for creating and querying MBeans and for invoking and manipulating MBean operations and attributes. The interface is depicted in Figure 9. We would like to point out two important methods: First is

```
registerMBean(Object object, ObjectName name)
```

which, as the name suggests, registers an object as MBean to the MBeanServer. The object has to fulfill a certain contract in the form of implementing an interface (see below). Second is

```
Object invoke(ObjectName name, String operationName,
              Object[]params, String[] signature)
```

All method invocations are tunnelled through the MBeanServer to the actual MBean by this method. The corresponding MBean is specified by `name`, whereas `operationName`, `params` and `signature` provide the rest of the information needed. Typing information gets lost and method calls are centralized. Hence, the architecture becomes resilient to changing requirements and evolving interfaces, as mentioned above. Due to this technique, it becomes easy to incorporate the mechanism of interceptors.

Regarding the *MBeans*, there are different types. Besides Standard MBeans, which we are using here, there exist Dynamic, Model and Open MBeans. Regardless of its type, an MBean must be a concrete and public Java class with at least one public constructor. The main constraint for writing a Standard MBean is the requirement to declare a statically typed Java interface that explicitly declares the management attributes and operations. The naming conventions used in the MBean interface follow closely the rules set by the JavaBeans component model. To expose the management attributes, one has to declare getter and setter methods, similar to JavaBean component properties. The MBeanServer

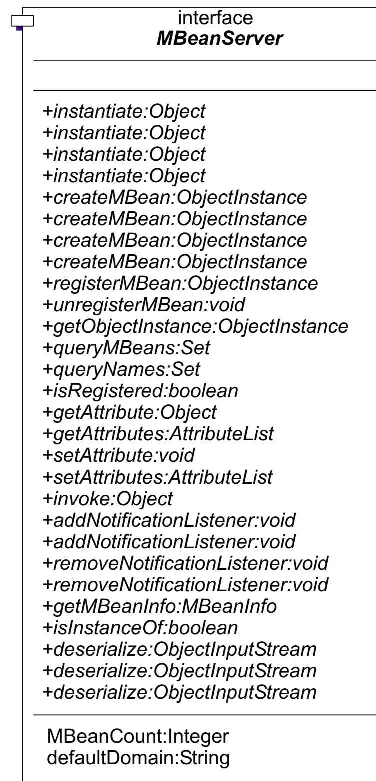


Figure 9: The MBeanServer Interface

uses introspection on the MBean class to determine which interfaces the class implements. In order to be recognized as a Standard MBean, a class `x` has to implement an interface `xMBean`. Defining the methods `getAttr()` and `setAttr()` will automatically make `Attr` a management attribute, in this case with read and write access. All the other public methods will be exposed as management operations.

JMX only provides a specification, which may be implemented by vendors. For the KAON SERVER, we have chosen the *JBoss MX*[7], a JMX implementation which is part of the comprehensive application server JBoss. The reason for this decision is (a) it fits our needs perfectly, (b) JBoss is open-source software and (c) it is needed anyway for several tools in the KAON suite.

4.1.2 Management usage scenarios

When a client connects to the KAON SERVER, it will first query for functional components it is in need of, e.g. an inference engine. We can distinguish two abstract component management usage scenarios which the server must be able to handle:

- the usage of already available, i.e. registered, components
- the usage of components that are not available at the time of the request and have to be instantiated by the management kernel

[7]http://www.jboss.org

In the following, we describe how the KAON SERVER should react in those usage scenarios. Note that the prototype is not yet able to act like that. Several parties are involved in the scenarios:

- *Clients*, issuing requests
- *Connector*, handling the communication between client and server
- *Registry*, organizing the deployed components
- *Kernel*, actual management functionality, like starting, stopping and dependency management.
- *Functional component*, providing the requested functionality
- *Invoker*, acting as a proxy for the functional component thus allowing interceptors to act in front of method invocations.

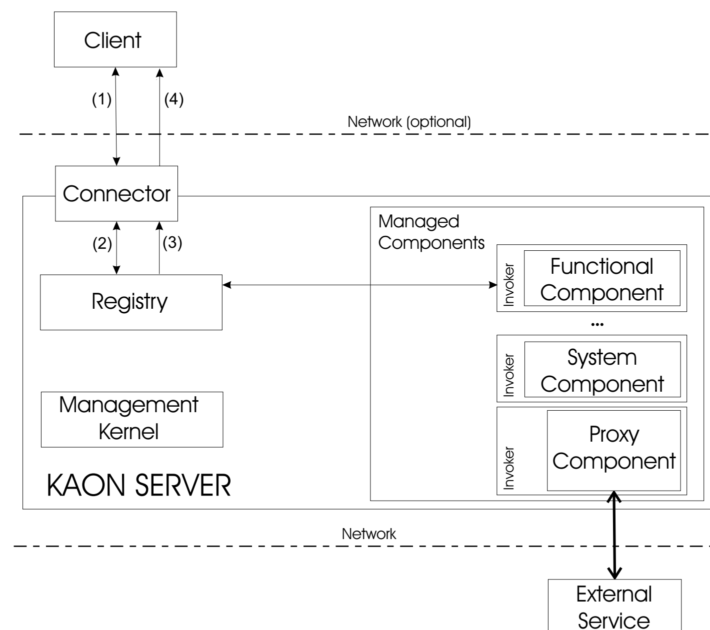


Figure 10: First usage scenario with available functional component

Figure 10 illustrates the first scenario. Note that most likely client and KAON SERVER reside in different, possibly distributed processes. Here, a client sends a request to the connector (1), i.e. RMI or Web Services (cf. section 7), which then accesses the registry (2) to resolve a component available to compute the request. A reference to the functional component is then returned to the connector (3), which then delivers the reference to the client (4). The client can now issue its request, which passes through the interceptors and should receive the appropriate response.

However, several additional steps have to be performed if a client requests a functionality that could be provided by some known functional component, but this component is

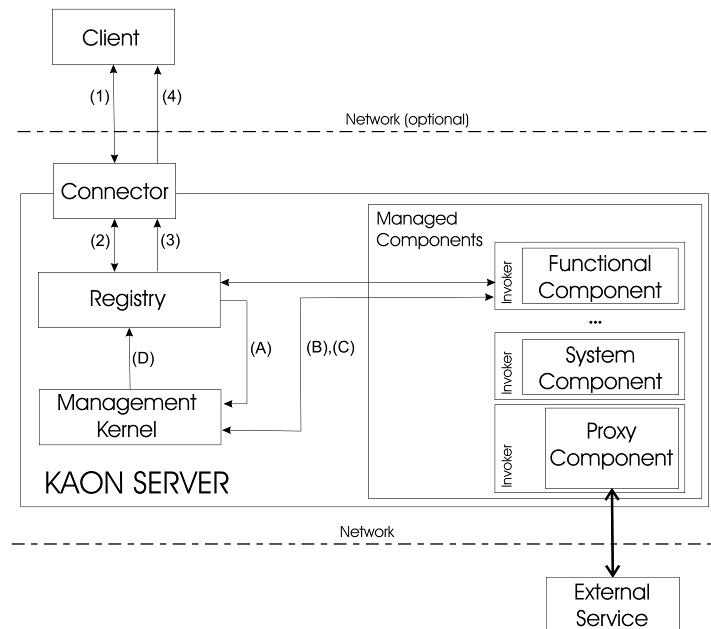


Figure 11: Second usage scenario without available functional component

not instantiated at the time of the request. This leads us to the second scenario illustrated in Figure 11.

Here, the registry consults the management kernel (A) to dynamically instantiate the appropriate functional component (B). The instantiated component is then available to the management kernel (C) and added to the registry (D). Notably, the instantiation of a component may lead to the cascading instantiation and registration of other components, if dependencies between components exist.

4.2 Registry

As mentioned in section 2, a client typically queries for the components it is in need of when it first connects to the KAON SERVER. There should be the possibility to state precisely what it wants, e.g. an RDF store that holds a certain RDF model and allows for transactions. It is the registry within the Management Layer which will provide this functionality. It hasn't been implemented so far, yet we have a good understanding of how it may look like at this point in time.

Component lookup like explained above may be realized by using existing or naming and directory services. The latter play a vital role in today's distributed systems by providing network-wide sharing of various information about users, machines, networks, components, and applications. Hence, multiple servers could share components across the network.

The Java Naming and Directory Interface (JNDI) is an API that provides naming and directory functionality to applications written in Java. Using JNDI, the implementation is able to build on functionality like the ability to store and retrieve named Java objects. For example, this can be used to store a given component temporarily to regain memory or

for later use without startup overhead, i.e. unparsing of XML. In addition, JNDI provides methods for performing standard directory operations, such as associating attributes with components and searching for components using their attributes. JNDI offers additional flexibility, since different naming and directory service providers may be seamlessly integrated behind the provided common API. Thus, the implementation can build on different existing naming and directory services, such as LDAP, NDS, DNS, and NIS(YP), and allows the server to coexist with further legacy applications and systems.

We plan to make use of the JNDI implementation which is part of the JBoss application server. It is realized as an MBean which fits very nicely into the Microkernel approach we have chosen. That means, the registry will be deployed as a system component, just like a connector for example.

In a first version, the registry will feature the components along simple attributes like names, connection parameters or the ontology an RDF store holds. In the more distant future, we envision to integrate means for information quality. That could be the reliability of a certain component, the performance of an inference engine, the timeliness of the statements in an RDF store and so on. Research on information quality deals with the definition and computing of such quality criteria. Correlating to that and one step further would be the incorporation of trust and trust management. For instance, it would be nice to have attributes that comment on the trustworthiness of an ontology store. Another thing we are fancying with is to make the querying process ontology-based. An ontology for management purposes would formally define which attributes a certain component may have and would put possible functional components into a taxonomy.

4.3 Interceptors

Intercepting, queuing, or redirecting requests becomes useful when a component is being restarted or reconfigured as part of its maintenance life-cycle, for example while loading an ontology. Also, sharing generic functionality such as security, logging, or concurrency control lessens the work required to develop individual component implementations when realized via interceptors.

Therefore each component will internally be registered with an invoker and a stack of interceptors that the request is passed through. The invoker object is responsible for managing the interceptors and sending the requests down the chain of interceptors towards the managed functional component.

For example, a logging interceptor could be inserted to implement auditing of operation requests. A security interceptor could be put in place to check that the requesting client has sufficient access rights for the component or one of its attributes or operations.

The invoker itself may additionally support the component lifecycle by controlling the entrance to the interceptor stack. When a component is being restarted, an invoker could block and queue incoming requests until the component is once again available (or the received requests time out), or redirect the incoming requests to another component that is able to service the request.

5 Data Access

This layer comprises data-related functionality including interfaces for accessing and updating data, as most requests will include some form of data manipulation. Inferencing is logically situated in this layer since it handles data. However, we expect that inference itself may be provided by external services, to which appropriate interfaces must be made available. The integration of these services has to be seamless concerning the initial user request.

Our basic philosophy is to include as little implementation as possible in this layer what results in improved flexibility. We introduce an API that allows access to RDF data and another interfacing to ontologies and associated knowledge bases (the so-called KAON API). Those two Semantic Web Data APIs are a mandatory part of KAON SERVER and discussed in the following subsections. Besides, both are central to the KAON toolsuite like mentioned in the introduction.

5.1 RDF API

The RDF API is part of the KAON toolsuite and consists of interfaces for transactional manipulation of RDF models with the possibility of modularization, a streaming-mode RDF parser and an RDF serializer for writing RDF models[8].

Concern is separated from contract, thereby alternative implementations of the contract, i.e. the interfaces, are possible. Section 3 discussed the implementations of this API which become functional components in the end.

We decided to make the set of interfaces a mandatory part of the KAON SERVER, as data access to RDF models is utterly crucial for a Semantic Web Management System. Of minor interest are the RDF parser and serializer.

Figure 12 depicts the most important classes and interfaces. A `Model` is viewed as a set of `Statement` objects. Each `Statement` has a subject, predicate and a object. Subject and predicate are `Resource` objects, while the object may be either of type `Resource` or `Literal`. `Statement`, `Resource` and `Literal` objects are immutable. They may be created through a `NodeFactory` which can be obtained using `Model.getNodeFactory()` object, method. The node factory makes sure that within a single `Model` there is at most one physical `Statement`, `Resource` or `Literal` object with the same URI.

One of the important design goals for the RDF API was to isolate API clients from the implementation of the model. Hence, `Model` instances aren't created directly, but through the `edu.unika.aifb.rdf.api.util.RDFManager` class. For each RDF API implementation, a so called RDF factory must be registered with the RDF manager, which will then be responsible for opening models of different implementations. By default, the in-memory RDF factory is registered with the `RDFManager`. RDF factories for other API implementations may be registered using the `RDFManager. registerFactory(String factoryClass)` method, and passing it the name of the RDF factory class (which is dependent on the actual API implementation).

`RDFManager` contains methods for opening, creating and deleting RDF models, which

[8]We have created a Developer's Guide at http://kaon.semanticweb.org

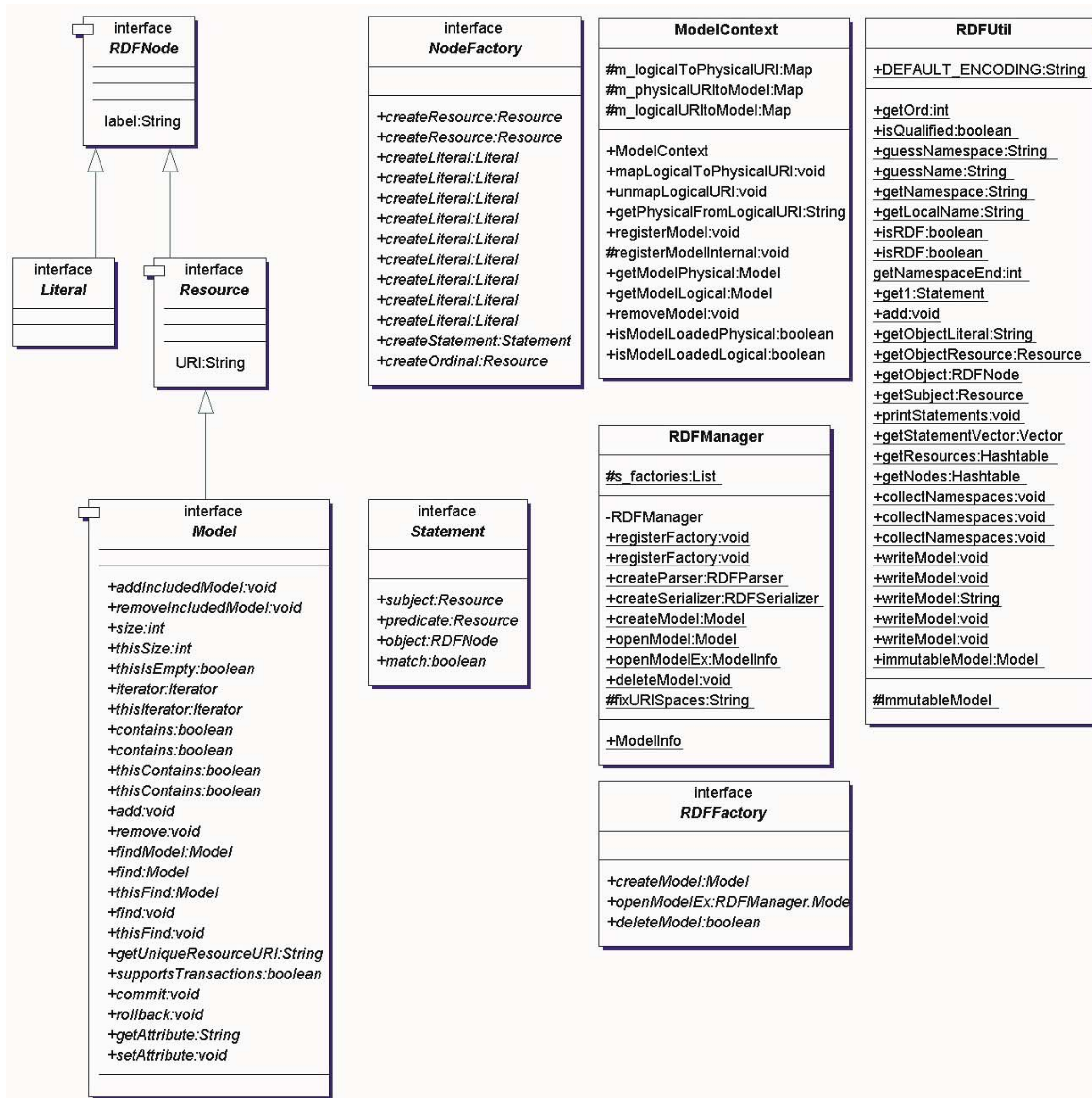


Figure 12: edu.unika.aifb.rdf.api.model

are identified by their physical URI. Based on the URI structure, `RDFManager` will locate the appropriate RDF factory and forward the call there.

`RDFManager` is responsible for loading only single models. If a model includes some other model, `RDFManager` will provide this inclusion information, but will not load the included models. The class `edu.unika.aifb.rdf.api.util.ModelContext` is responsible for that. Within each `ModelContext` there may exist exactly one Model with some logical URI. This class also provides methods for loading models, which, if a model includes some other model, makes sure that all included models are loaded as well.

5.2 Ontology API

The Web Ontology Language (OWL) [9], which is currently developed by the W3C, will play a major role in the Semantic Web as standard ontology modelling language. OWL attempts to capture many of the commonly used features of DAML+OIL. It also adds functionality beyond RDFS in order to come up with a powerful language useful for Semantic Web applications.

In order to cope with the envisioned and more powerful OWL Full layer, additional inference components will be needed. In lack of standardization and current dispute on the provided features, the KAON API currently realizes the ontology language described in [12]. Besides, we have integrated means for ontology evolution, a sophisticated transaction mechanism and views. The implementations adhere to the formal semantics given there. For the same reasons like the RDF API, we made the Java interfaces a mandatory part of the KAON SERVER. The interfaces offer access to KAON ontologies and contain classes such as `Concept`, `Property` and `Instance`.

The API decouples the ontology user from actual ontology persistence mechanisms. There are different implementations for accessing RDF-based ontologies accessible through the RDF API or ontologies stored in relational databases using the Engineering Server (cf. Figure 4). All of the implementations will become functional components in the end, like discussed in section 3.

Management of ontology changes is realized through pluggable evolution strategies [15], that the user can adapt the behavior of various operations according to her needs. For example, when deleting a concept from the ontology, it is possible to direct the API to remove its children as well, or to reconnect them to the parent. For each operation, the user is provided with detailed insight into the consequences of the operation before the operation is actually applied to the ontology. To facilitate ontology reuse, KAON API supports inclusion. An important unit of information in the API is an OI-model, and it may contain concepts, properties and instances. Further, each OI-model may include any number of other OI-models and thus reuse ontology and instances definitions in other models. KAON API implements the Observable design pattern, thus allowing users to get notifications when ontology changes.

Figure 13 shows the most important interfaces. An OI-model is represented by the interface of the same name. For clarity, the methods related to the ontology part of an OI-model are separated into the `Ontology` interface and methods related to the instance part are separated into `InstancePool` interface. However, these two interfaces are never instantiated alone, but always together as an OI-model.

All entities stored in an `OIModel` are derived from the `Entity` interface. Each entity belongs to a certain OI-model, as returned by the `getOIModel()` method. Besides, an entity can be declared in some other OI-model, which can be returned using the `getSourceOIModel()` method.

A concept is represented through the `Concept` interface, which provides methods for fetching information such as the set of super- or subconcepts. Similar holds for `Property` and `Instance` interfaces whose methods allow for fetching information such as the set of domain or range concepts, the set of parent concepts or related instances, respectively. For each OI-model there is exactly one `Concept`, `Property` or `Instance` object for some

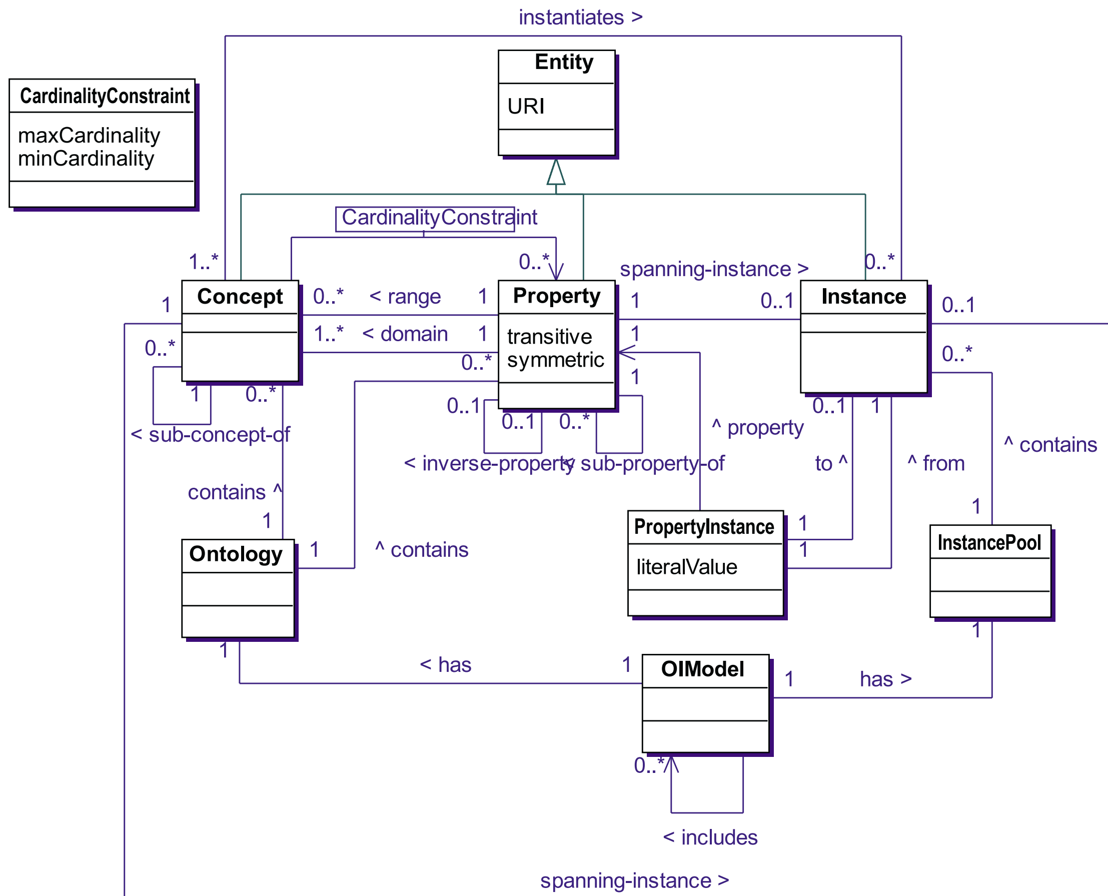


Figure 13: edu.unika.aifb.kaon.api

URI. This means that within an OI-model, each URI determines the identity of associated objects uniquely. Special types of instances are lexical entries, which are represented through the `LexicalEntry` interface, providing utility methods for accessing lexical information.

Instances can be connected using property instances, which are represented through the `PropertyInstance` interface. Each `PropertyInstance` connects the property, the source instance and the target value. The target value can be a `String` (in case a property is an attribute), or an instance (in case a property is a relation).

6 Security

The Security Layer has not been addressed so far. No functionality belonging here has been implemented for the prototype. However, what has become clear is that several interceptors will be introduced here which guarantee that operations offered by functional components in the server are only available to appropriately authenticated and authorized clients. A security interceptor could be put in place to check that the requesting client has sufficient access rights for the component or one of its attributes or operations.

Interceptors could also provide auditing, i.e. the logging of clients' activities to log files. These log files may be further used, for example by a versioning or evolution component or to operationalize data roll-back. In the more distant future we are fancying to incorporate trust issues in one or another way, possibly as an interceptor, too.

As section 4.3 already explained, intercepting of requests becomes useful when a functional component should share generic functionality such as security, logging, or concurrency control, thereby lessening the work required to develop individual component implementations.

Hence, each component will internally be registered with an invoker and a stack of interceptors that the request is passed through like depicted in Figure 14. The invoker object is responsible for managing the interceptors and sending the requests down the chain of interceptors towards the managed functional component. The invoker itself may additionally support the component lifecycle by controlling the entrance to the interceptor stack.

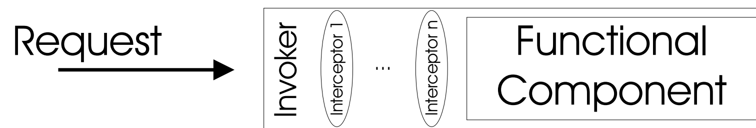


Figure 14: Invoker and interceptors

Readers familiar with the J2EE EJB specification may be aware that this functionality corresponds the EJB container contract which is usually implemented by setting a stack of interceptors in front of the EJB object. The request is passed through the interceptors that enforce the EJB specification functionality. This is a specific instance of the more generic design outlined here.

7 Connectors

Besides the local Java API, which can be used to embed the KAON SERVER into a client application, two further possibilities are provided for connection: The first offers access to the hosted functionality per Java Remote Method Invocation (RMI), a second connector offers access to the server using a web service by the Simple Object Access Protocol (SOAP). The default connector is the inherent Java API which, when used, does not necessitate remote calls.

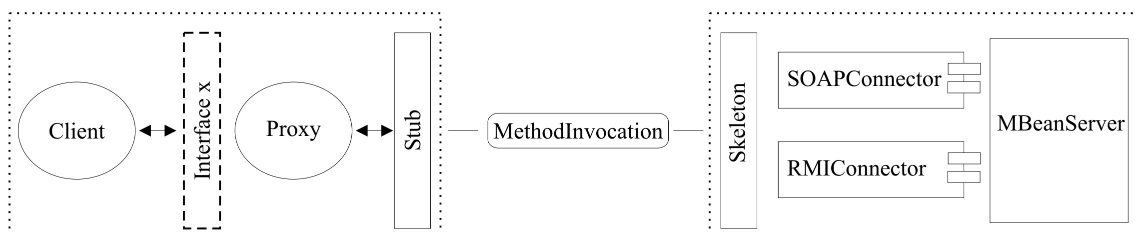


Figure 15: Connector's infrastructure

The whole connector's infrastructure already exists and is explained is this section. Figure 15 depicts the general scheme in which the two remote connectors are realized as system components. `SOAPConnector` and `RMIConnector` are both implemented as MBeans which may be deployed to the KAON SERVER, or more specifically to the MBeanServer, when needed. Basically, both export `invoke(MethodInvocation)` to the outside as defined in the `MBeanServerInvoker` interface. This is done as web service accessible per HTTP or as RMI skeleton class along an appropriate entry in the RMI registry. `MethodInvocation` is the actual payload of the respective remote call and encapsulates the call with all the necessary information. Of course, this class is designed to overcome all serialization hazards. The connectors embrace objects of this class and translate it to the MBeanServer's `invoke()` method.

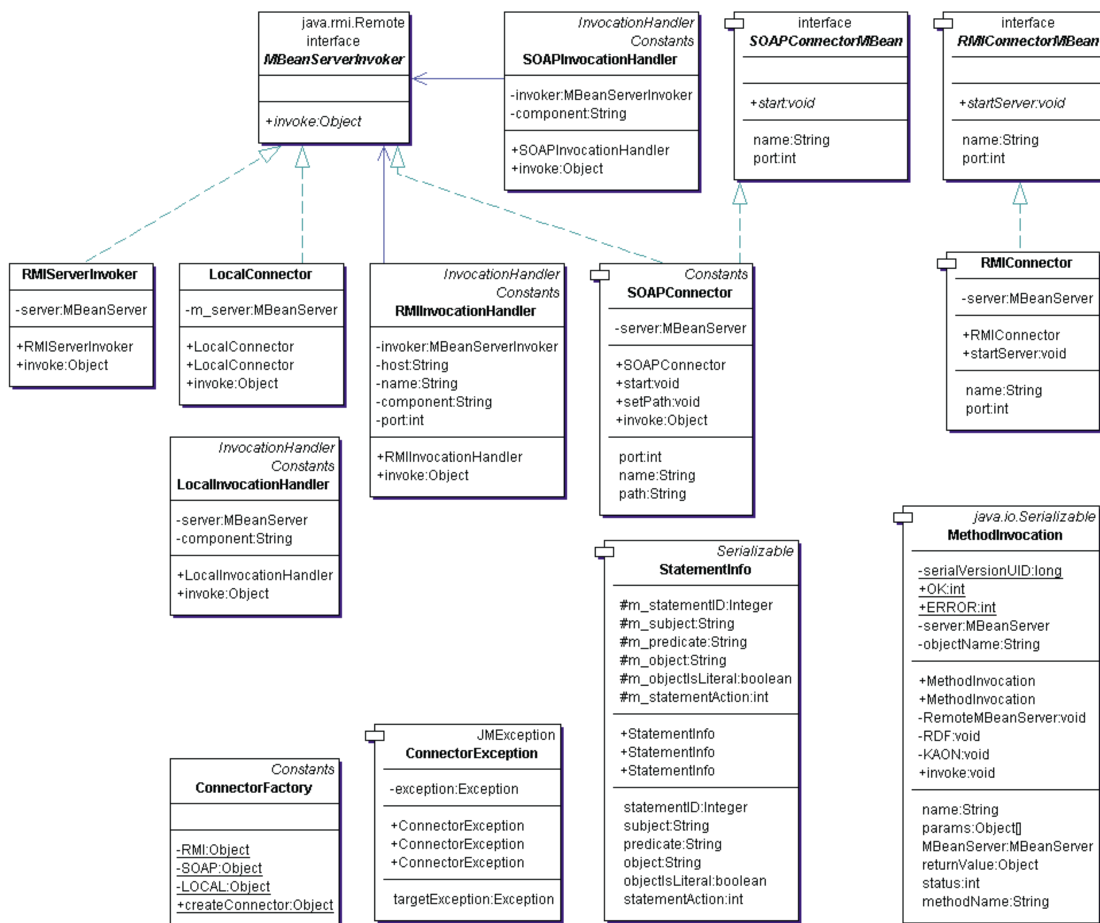


Figure 16: Classes and interfaces of the Connectors Layer

On the client side, all the connectors' details are elegantly hidden. Key to this is a helper class called `ConnectorFactory` allowing a client to create the connector it is in need of. After that, the client can work with a functional component as if it were locally instantiated. The class features basically one method `Object createConnector (properties)` which has to be fed with connection information. It returns an instance of `Object` which has to be casted adequately like shown in Table 7. Until now, there are

only two classes that are returned: `RemoteModel` and `RemoteMBeanServer` proxying for an `RDFComponent` and the MBeanServer itself for management purposes. Translation of calls is encoded within the `MethodInvocation` class. When more functional components come into play later in the project, mappings of their methods have to be incorporated, too.

Parameter	Value	Explanation
`TYPE`	`MGMT`	Returned object to be casted to `RemoteMBeanServer`
	`RDF`	Returned object to be casted to `RemoteModel`
`CONNECTION`	`LOCAL`	Creates a local connector, MBeanServer has to run in same JVM
	`SOAP`	Creates a SOAP connector, SOAPConnector MBean has to be deployed. Properties `SOAP_HOST`, `SOAP_PORT`, `SOAP_NAME` and , `SOAP_PATH` have to be provided as connection information.
	`RMI`	Creates an RMI connector, RMIConnector MBean has to be deployed. Properties `RMI_HOST`, `RMI_NAME`, `RMI_PORT` have to be provided as connection information.
`COMPONENT`	`<JMX-NAME>`	The JMX name of the component has to be provided when `TYPE=RDF` to address a certain `RDFComponent` MBean

Table 1: createConnector(Properties)

Also belonging in the Connectors Layer is the so-called HTTP GUI Adaptor, like depicted in Figure 3. This adaptor from Sun is an MBean itself and allows to interact with the MBeanServer directly via a browser interface. Figure 17 shows the state of the MBeanServer when running the prototype. There are seven registered MBeans in this case: the two connectors, three system dependent ones, the HTTP GUI Adaptor and finally the RDFComponent. By clicking on the latter, all the management attributes and operations are exposed in a separate document. The user is able to interact and change attributes on the fly, even invoke some of the management operations.

8 Scenario

After all the abstract architectural and technical details, we decided to include a little scenario for didactic purposes, depicted in Figure 18. Imagine a simple genealogy application. Apparently, the domain description, viz. the ontology, will include concepts that talk about Persons and make a distinction between Males and Females. People are related with each other by several relations expressing parenthood and siblings. Ergo, properties like hasParent, hasSister etc. will be in place. This domain description can be

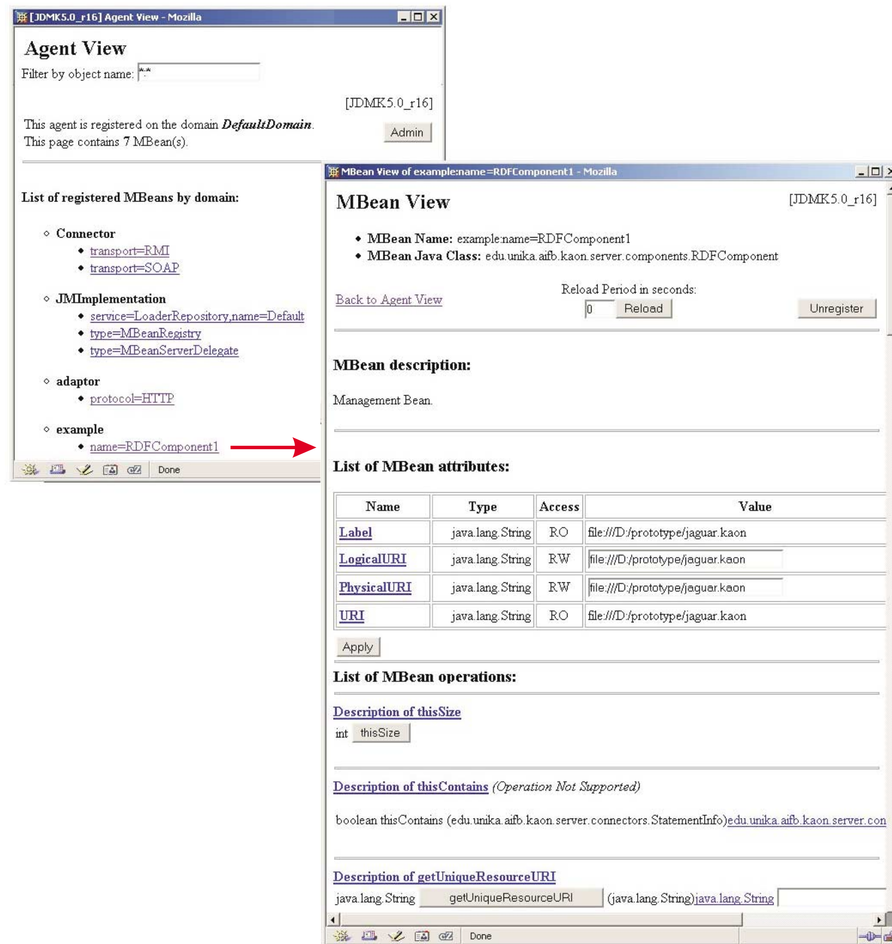


Figure 17: Screenshot of the HTTP GUI Adaptor

easily constructed with standard description logic ontologies. However, many important facts that could be inferred automatically have to be added explicitly. For example, the uncles and aunts of a person have to be stated explicitly. However, within a rule-based system, it is easy to build rules which capture those facts automatically. Furthermore, Persons will have certain properties which require data types, such as birthdates, which cannot be modelled within the current ontology systems. Such an ontology could serve as the conceptual backbone and information base of a genealogy portal, usage of it would greatly simplify the maintenance of the data and would offer machine understandability of the presented data. To implement the system, all the required components, i.e. a rule-based inference engine, a DL reasoner, a XML Schema data type component, would have to be combined manually using proprietary code. While this is a doable effort, the system would be highly proprietary and not reusable across the domain.

Apparently, the demands to an ontology management system from this application is to hook up to all those components and to offer management of data flow between them. This also involves propagation of updates and rollback behavior, if any component in the information chain breaks.

As depicted in Figure 19, the KAON SERVER could be used to solve that scenario.

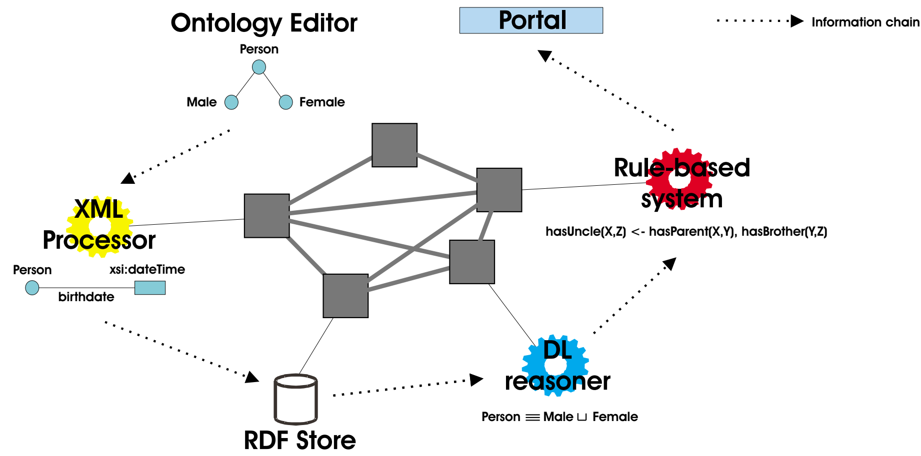


Figure 18: Scenario scheme

The ontology editor could be OntoEdit [16], querying the KAON SERVER for a particular XML Processor to validate the XML schema datatypes used in the ontology as well as an RDF store capable of saving the ontology (1). We assume that this store could be the RDF Server, viz. the persistent implementation of the RDF API. After that, OntoEdit is in possession of all it needs to validate the ontology edited (2) and finally store it (3).

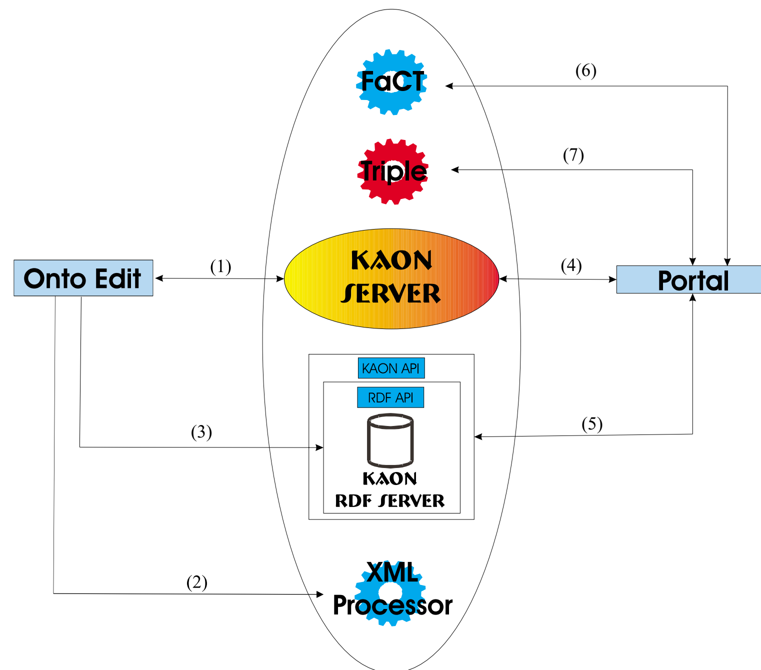


Figure 19: Solution for the scenario

On the other hand, there is the portal application which is looking for the same store but with ontological access rather than the simple RDF interface. After querying for that store, the KAON SERVER creates a new component: an implementation of the KAON

API which relies on an implementation of the RDF API[9]. By doing that, it is able to reuse the already existing RDF store component. A reference of the new component is returned to the portal application (4). What follows are queries for registered FaCT and Triple components. As we assume that those have already been registered, the KAON SERVER simply returns references (4).

Having acquired all the necessary components, the portal application takes on computing the envisioned results. First it queries the RDF Server on an ontological level (5). The results are then fed into FaCT in order to take advantage of its subsumption capabilities (6)[10]. Finally, the results from FaCT form the input to Triple. Rules allow for additional statements to be deduced (7).

9 Related Work, Conclusion and Outlook

Currently much research on middleware circles around so-called service oriented architectures (SOA) [11], which are similar to our architecture, since functionality is broken into components - so-called Web Services - and their localization is realized via a centralized replicating registry (UDDI)[12]. However, here all components are standalone processes and are not manageable by a centralized component management module. The statements for SOAs also holds for previously proposed distributed object architectures with registries such as CORBA Trading Services [2] or JINI[13].

Several of today's application servers share our design of constructing a server instance via separately manageable components, e.g. the HP-AS [14] or JBOSS[15]. However, they do not allow to manage the relations between components and their dependencies, as well as dynamic instantiation of registered components due to client requests - rather all components have to be started explicitly via configuration files or a management interface.

This deliverable presented a first prototype of KAON SERVER, a comprehensive Semantic Web Management System. It is part of the open-source Karlsruhe Semantic Web and Ontology Toolsuite (KAON) and downloadable at http://kaon.semanticweb.org. The server basically realizes a comprehensive Component Management with several Semantic Web Data APIs as mandatory part. It thus answers to the needs put forward by both the dynamic and the static parts of the Semantic Web layer cake. From our perspective, the KAON SERVER will be an important step in putting the Semantic Web into practice. Based on our experiences with building Semantic Web applications we devise that such a management system will be a crucial cornerstone to bring together so far disjoint components.

In the future, we will incorporate the Security Layer as well as all the other missing

[9]This corresponds the second management usage scenario, cf. section 4, whereas all the other queries fit into the first usage scenario

[10]The portal application can make use of the standardized interface of FaCT as designed by DIG - the Description Logic Implementation Group, http://dl.kr.org/dig/

[11]http://archive.devx.com/xml/articles/sm100901/sidebar1.asp

[12]http://www.uddi.org/

[13]http://www.jini.org

[14]http://www.bluestone.com

[15]http://www.jboss.org

parts and adapt existing inferences engines like FaCT, Ontobroker and Triple. In order to be deployed to the KAON SERVER, proxy components have to be developed for each of them. We are also considering to integrate means for semantic validation of data. Future data APIs evolving out of the Semantic Web layers not yet standardized will be taken into account, too. Besides, views, evolution and versioning of ontologies are exciting technologies to include in KAON SERVER.

In a first version, the registry will feature the components along simple attributes like names, connection parameters or the ontology an RDF store holds. In the more distant future, we envision to integrate means for information quality. That could be the reliability of a certain component, the performance of an inference engine, the timeliness of the statements in an RDF store and so on.

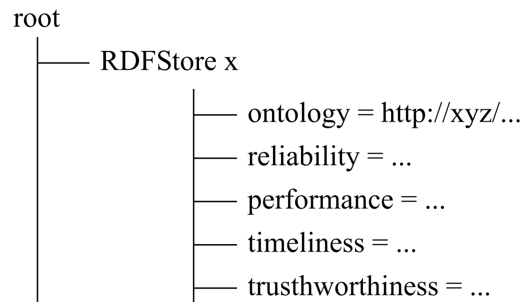


Figure 20: How the registry could look like

Figure 20 depicts how an registry entry for an RDF store could look like. A simple attribute would be ”ontology” providing information about which ontology the store holds currently. Other attributes like ”reliability”, ”performance” or ”timeliness” comment on quality criteria and allow the client to query on non-trivial features of the component.

Research on *information quality* deals with the specification and computing of such quality criteria. General definitions for information quality are ”fitness for use”, ”meets information consumers needs”, or ”user satisfaction” [14]. We conceive quality as an aggregated value of multiple criteria. With this definition information quality is flexible regarding the application domain, the sources, and the users, because the selection of criteria can be adapted accordingly. Also, assessing scores for certain aspects of information quality and aggregating these scores is easier than immediately finding a single global score. In general, criteria can be divided into three groups: subject-, object- and process-criteria. Belonging in the first group are only criteria that can be determined by individual users based on their personal views, experience, and background. On the other hand, object criteria are resolved by a careful analysis of data, i.e. one can compute them out of the store for example. The scores of process-criteria can only be determined by the process of querying.

Correlating to that and one step further would be the incorporation of trust and trust management. For instance, it would be nice to have an attribute ”trustworthiness” in the registry like depicted in Figure 20. To trust is to undertake a potentially dangerous operation *knowing that it is potentially dangerous*. A user might prefer to have proof of harmlessness, e.g. the Java interpreter for applets guarantees only harmless actions, but

weaker forms of evidence may also be sufficient. A recommendation from a close friend in the same community may convince someone to trust the contents of an ontology-store. Credentials and policies are the raw materials for making trust decisions. A credential is a statement, purportedly made by some speaker. A policy determines the conditions under which a particular action is allowed. Following [5], we use the term trust management to refer to the problem of deciding whether requested actions, supported by credentials, conform to policies. Just as a database manager coordinates the input and storage of data and processes queries, a trust manager coordinates the collection and storage of policies and credentials and processes requests for trust decisions. We refer to the processing of such a request as ”evaluating compliance with a policy”. Besides the registry, the incorporation of trust management will most likely require other means like interceptors.

Another thing we are fancying with is to make querying for required components ontology-based. The definitions in section 3 implicitly imply a taxonomy, e.g. it was stated that ”functional component” is a specialization of ”component” and so on. In the paragraphs above we learned that each component will have certain attributes, like quality criteria or just simple information. Besides, we know of associations between components, for example, an OIModel can be based on an RDFComponent, as the KAON API can be based on the RDF API. Thus, it is obvious to formalize all that in a management ontology like outlined in Figure 21. It would formally define which attributes a certain component may have and would put components into a taxonomy. Hence, it can be considered as conceptual agreement and consensual understanding between a client and the KAON SERVER. In the end, actual functional components like the RDF Server or the Engineering Server would be instantiations of corresponding concepts.

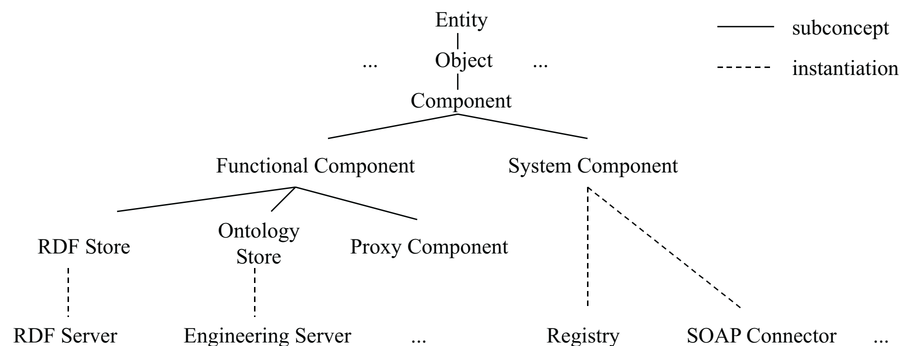


Figure 21: Management ontology

References

[1] S. Bechhofer, I. Horrocks, C. Goble, and R. Stevens, *Oiled: a reasonable ontology editor for the semantic web*, Proc. of the Joint German Austrian Conference on AI, number 2174 in Lecture Notes In Artificial Intelligence, pages 396-408, Springer, 2001.

[2] Juergen Boldt, *Corbaservices specification*, 3 1997.

[3] E. Bozsak, M. Ehrig, S. Handschuh, A. Hotho, A. Maedche, B. Motik, D. Oberle, C. Schmitz, S. Staab, L. Stojanovic, N. Stojanovic, R. Studer, G. Stumme, Y. Sure, J. Tane, R. Volz, and V. Zacharias, *Kaon - towards a large scale semantic web*, Proceedings of EC-Web 2002, Springer, 2002.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-oriented software architecture, volume 1: A system of patterns*, vol. 1, John Wiley and Son Ltd, 1996.

[5] Yang-Hua Chu, Joan Feigenbaum, Brian A. LaMacchia, Paul Resnick, and Martin Strauss, *Referee: Trust management for web applications*, Proceedings of WWW6, 1997 (Ian F. Akyildiz and Harry Rudin, eds.), vol. 29, Computer Networks and ISDN Systems, no. 8-13, Elsevier, 1997, pp. 953–964.

[6] Stefan Decker, Michael Erdmann, Dieter Fensel, and Rudi Studer, *Ontobroker: Ontology based access to distributed and semi-structured information*, DS-8, 1999, pp. 351–369.

[7] I. Horrocks, *The fact system*, Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98, Springer, 1998.

[8] Michel Klein, Atanas Kiryakov, Damyan Ognyanov, and Dieter Fensel, *Ontology versioning and change detection on the web*, 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02), Sigenza, Spain, October 1-4, 2002, 2002.

[9] Deborah L. McGuinness, *Proposed compliance level 1 for webont's ontology language owl*, Knowledge Systems Laboratory, Stanford University.

[10] Sergej Melnik, *Rdf api*, Current revision 2001-01-19.

[11] Stefan Decker Michael Sintek, *Triple - an rdf query, inference, and transformation language*, In proceedings ISWC'2002, Springer, 2002.

[12] B. Motik, A. Maedche, and R. Volz, *A conceptual modeling approach for building semantics-driven enterprise applications*, Proceedings of the First International Conference on Ontologies, Databases and Application of Semantics (ODBASE-2002), November 2002.

[13] L. Stojanovic N. Stojanovic, R. Volz, *A reverse engineering approach for migrating data-intensive web sites to the semantic web*, IIP-2002, August 25-30, 2002, Montreal, Canada (Part of the IFIP World Computer Congress WCC2002), 2002.

[14] Felix Naumann, *Quality-driven query answering for integrated information systems*, Lecture Notes in Computer Science, vol. 2261, Springer, 02 2002.

[15] L. Stojanovic, N. Stojanovic, and S. Handschuh, *Evolution of metadata in ontology-based knowledge management systems*, 1st German Workshop on Experience Management: Sharing Experiences about the Sharing of Experience, Berlin, March 7-8, 2002, Proceedings, 2002.

[16] Y. Sure, M. Erdmann, J. Angele, S. Staab, R. Studer, and D. Wenke, *Ontoedit: Collaborative ontology development for the semantic web*, Proceedings of the 1st International Semantic Web Conference (ISWC2002), June 9-12th, 2002, Sardinia, Italia, Springer, 2002.

[17] Raphael Volz, Daniel Oberle, and Rudi Studer, *Views for light-weight web ontologies*, In Proceedings of the ACM Symposium on Applied Computing SAC 2003, March 9-12, 2003, Melbourne, Florida, USA, 2003.