Chapter 11

Implementation

This chapter describes the implementation and the usage of the *WIQA - Information Quality Assessment Framework*. The WIQA framework is implemented in Java. The implementation consists of two parts: *NG4J - Named Graph API for Jena* which is a general purpose extension to the Jena Semantic Web framework [CDD+04] for handling Named Graphs. Second, the *WIQA Filtering and Explanation Engine* which enables applications to filter a set of named graphs using WIQA-PL policies and to retrieve explanations about filtering decisions.

11.1 The NG4J - Named Graph API for Jena

NG4J - Named Graph API for Jena [BCW05][BC06] is a software toolkit for creating, manipulating, persisting, and exchanging sets of named graphs.

The toolkit provides an API for manipulating a set of named graphs using graph-centric and quad-centric methods. Graph sets can be stored in memory or in a relational database. NG4J provides parsers and serializers for the TriX and TriG syntaxes introduced in Section 5.3.

NG4J implements convenience methods for using the Semantic Web Publishing Vocabulary introduced in Chapter 6. The toolkit enables users to sign graph sets and to verify signatures without the need for detailed knowledge about signature methods and the SWP vocabulary.

NG4J builds on the Jena Semantic Web framework [CDD+04], a leading Semantic Web programming environment which is maintained by the Hewlett Packard Laboratories in Bristol. NG4J is available under the terms of the Berkeley Software Distribution (BSD) license [Reg99] and can be downloaded from the NG4J website[1].

[1] http://www.wiwiss.fu-berlin.de/suhl/bizer/ng4j/ (retrieved 09/25/2006)

The followings sections give an overview of NG4J's public interface and illustrate the usage of the toolkit with a code example. The complete documentation of the toolkit is available on the NG4J website[2].

11.1.1 Public Interface

The UML diagram shown in Figure 11.1 gives an overview of NG4J's public interface.

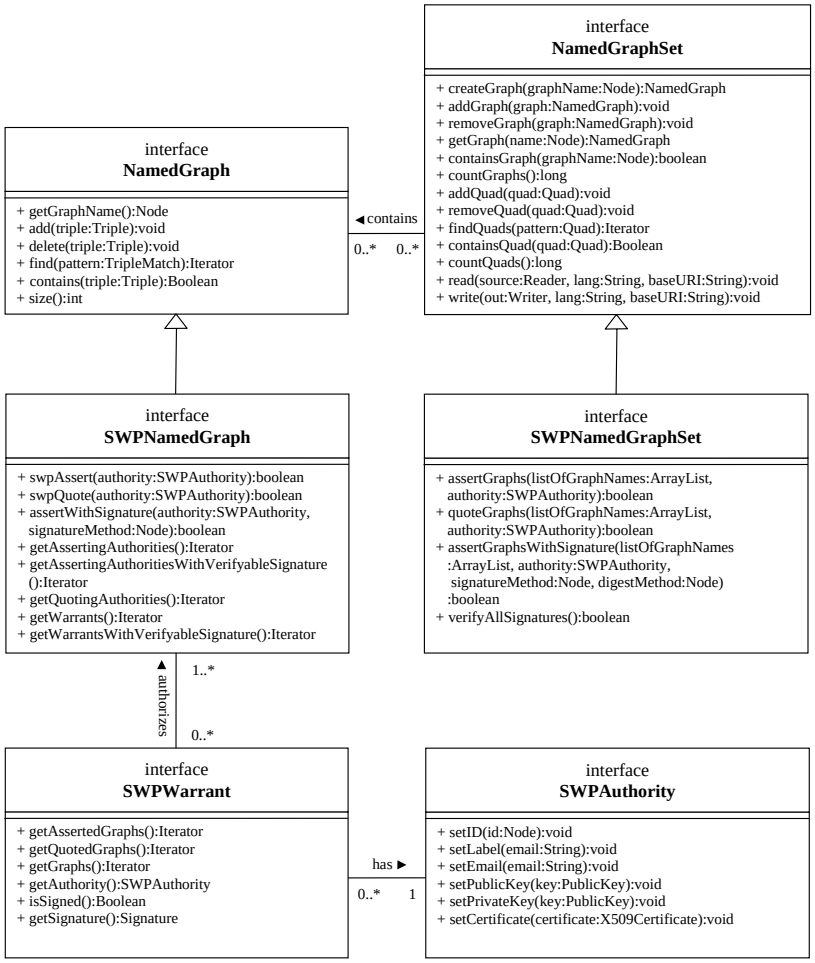

Figure 11.1: Overview of NG4J's public interfaces.

[2] http://www.wiwiss.fu-berlin.de/suhl/bizer/ng4j/javadoc/ (retrieved 09/25/2006)

NamedGraph and NamedGraphSet

The `NamedGraph` interface defines basic methods for manipulating a named graph. The `add()` and `delete()` methods add and remove triples from the graph. The `find()` method returns an iterator over all triples that match a given triple pattern.

The basic information container in NG4J is the `NamedGraphSet`. The `NamedGraphSet` interface defines methods for manipulating and serializing a set of named graphs. A graph set can be manipulated by adding and removing entire graphs, or by working with individual *quads*. Within NG4J, quads are not understood as separate entities beside named graphs but as a different view on a graph set. The first item in a quad is a graph name; the other three constitute an RDF triple. If a quad, having a graph name that does not exists in a graph set yet, is added to the set, then a new graph with this name is created in the set. If a quad with a graph name that already exists in a graph set is added to the set, then the triple of the quad is added to the corresponding named graph.

The `write()` method serializes the graph set to a file. The `lang` argument defines the serialization syntax. NG4J supports the TRIX, TRIG, RDF/XML, N-TRIPLE, and N3 syntaxes. If the specified serialization syntax does not support graph naming, then the union graph is serialized, and knowledge about the partitioning of triples into graphs is lost.

NG4J provides two implementations of the `NamedGraphSet` and `NamedGraph` interfaces: The classes `NamedGraphSetImpl` and `NamedGraphImpl` store graph sets in memory. The classes `NamedGraphSetDB` and `NamedGraphDB` persist graph sets in a relational database. `NamedGraphSetDB` is instantiated for a given JDBC `Connection`. Database tables for storing graph sets are automatically created by NG4J if they do not exist in the database yet.

The Semantic Web Publishing API

NG4J provides a convenience API for asserting and quoting graphs using Semantic Web Publishing Vocabulary introduced in Chapter 6. The API consists of the `SWPNamedGraphSet`, `SWPNamedGraph`, `SWPWarrant` and `SWPAuthority` interfaces.

`SWPAuthority` stores the URI reference, label, email address, public and private key, and certificate of an information provider. Keys and certificates are represented using the `java.security` package. NG4J supports RSA [KS98] and DSA [FIP95b] keys, as well as X.509 certificates [HPFS02].

`SWPWarrant` represents a warrant which may assert or quote several graphs. The methods `getAssertedGraphs()` and `getQuotedGraphs()` return an iterator

over all graphs that are asserted or quoted by the warrant. The method `getAuthority()` returns the authority of the warrant. The method `isSigned()` can be used to check whether a warrant is signed.

`SWPNamedGraph` extends the NamedGraph interface. `SWPNamedGraph` defines convenience methods for asserting and signing named graphs and for verifying graph signatures.

The method `swpAssert(authority:SWPAuthority)` turns the graph into a self-asserting warrant graph by adding the triples `<GraphName> swp:assertedBy <GraphName>` and `<GraphName> swp:authority <Authority>` to the graph. `<GraphName>` stands for the URI reference identifying the current graph; `<Authority>` stands for the URI reference identifying the authority that is passed as argument to the method.

The method `assertWithSignature()` turns the graph into a self-asserting warrant graph and signs the graph with the private key of the authority. The argument `signatureMethod` specifies the combination of canonicalization, digest, and signature algorithms that should be used to calculate the signature. Signature methods are identified by the URI references listed in Table 6.3. NG4J supports the RSA [KS98] and DSA [FIP95b] signatures algorithms, SHA1 [FIP95a] and MD5 [Riv92] digests, and the RDF canonicalization algorithm proposed by Carroll [Car03]. The `assertWithSignature()` method adds a `swp:assertedBy`, a `swp:authority`, a `swp:signatureMethod`, and a `swp:signature` triple to the graph.

The method `getWarrantsWithVerifyableSignature()` returns an iterator over all warrant that assert or quote the graph and are signed with a verifiable signature. For verifying graph signatures, NG4J requires the public keys or certificates of information providers and root certification authorities that are trusted by the current user. NG4J expects trusted keys and certificated to be contained in the graph `<http://localhost/trustedinformation>`, which has to be added to a graph set before calling any of NG4J's signature verification methods. NG4J traces certification chains up to a certificate in this trusted graph. Certification chains are constructed using all certificates that are contained in the graph set.

The `SWPNamedGraphSet` interface extends the NamedGraphSet interface. `SWPNamedGraphSet` defines convenience methods to assert and sign multiple graphs at once and to verify all signatures that are contained in a graph set.

The method `assertGraphsWithSignature()` asserts and signs multiple graphs. The resulting warrant is added as a new graph to the graph set. The method takes a list of graph names, an `SWPAuthority`, a signature method, and digest method as argument. The method adds a new warrant graph to the graph set which asserts all graphs from the list. The new graph is named with a Universally Unique IDentifier (UUID) [LMS05] that is created using

the Java package `com.eaio.uuid.UUID` [Bur06]. Then, the method digests all graphs from the list and adds the digests to the new warrant graph. Afterwards, the signature for the warrant graph is calculated and is added to the warrant graph.

The method `verifyAllSignatures()` verifies all signatures in the graph set. The method expects trusted keys and certificates to be contained in the graph `<http://localhost/trustedinformation>`. The verification result is added to the graph set as a new graph called `<http://localhost/verifiedSignatures>` The graph contains a `<Warrant> swp:signatureVerification swp:sucessful` triple for each warrant with a verifiable signature. This graph can be used within WIQA-PL filtering policies for checking whether content is digitally signed.

11.1.2 Usage Example

The example code shown in Figure 11.3 illustrates how NG4J is used to create a graph set, add information to the graph set, retrieve information from the graph set and finally serialize the graph set using the TriX syntax [CS04a].

11.2 The WIQA - Filtering and Explanation Engine

The WIQA - Filtering and Explanation Engine enables applications to filter a set of named graphs using WIQA-PL policies and to retrieve explanations about the filtering decisions. WIQA implementation builds on NG4J and the ARQ SPARQL query engine [Sea06].

The WIQA engine is available under the terms of the GNU General Public License [Fre91] and can be downloaded from the WIQA website[3]. The following sections give an overview of the public classes of the engine and illustrate its usage with a code example. The complete documentation of the engine is available on the WIQA website[4].

11.2.1 Public Interface

The UML diagram shown in Figure 11.1 gives an overview of the public interface of the WIQA - Filtering- and Explanation Engine.

[3] http://www.wiwiss.fu-berlin.de/suhl/bizer/wiqa/ (retrieved 09/25/2006)
[4] http://www.wiwiss.fu-berlin.de/suhl/bizer/wiqa/javadoc/ (retrieved 09/25/2006)

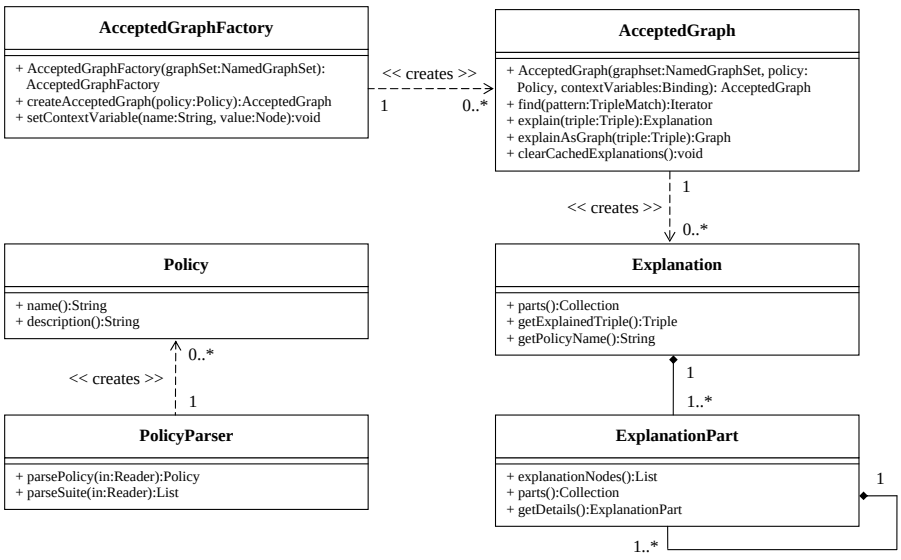

Figure 11.2: Overview of the public interface of the WIQA Filtering- and Explanation Engine.

Policy and PolicyParser

WIQA-PL policies are represented as instances of the class `Policy`. The class `PolicyParser` provides static methods for parsing WIQA-PL policies from files and strings into `Policy` objects. The method `parsePolicy()` creates a single `Policy` object from a string or a Java `Reader`. The method `parseSuite()` parses all policies in a policy suite and returns a list of `Policy` objects.

AcceptedGraph and AcceptedGraphFactory

The class `AcceptedGraph` is the main interface of the WIQA engine. An `AcceptedGraph` represents a filtered view on a set of named graphs. Only statements matching a WIQA-PL policy are in the graph. The method `find()` is used to extract information from the graph. The method returns an iterator over all accepted triples that match a given triple pattern.

`AcceptedGraph`s are created using an `AcceptedGraphFactory`. The method `createAcceptedGraph()` of the factory class returns an `AcceptedGraph` for a given `Policy`. The method `setContextVariable()` is used to set WIQA-PL context variables (see Section 9.4) which may be used within policies afterwards.

Retrieving Explanations

`AcceptedGraph`s can generate textual explanations and RDF explanations why a triple was accepted into the graph. A textual `Explanation` is created by calling the method `explain()` of an `AcceptedGraph`. The method takes an accepted triple as argument.

An `Explanation` consists of a collection of `ExplanationPart`s. The method `parts()` returns the collection of `ExplanationPart`s. An `ExplanationPart` represents a text fragment and has zero or more children which are also `ExplanationPart`s. The text fragment is represented as list of RDF nodes. Most of these are literals, but some may be URI references or blank nodes which refer to some entity involved in the explanation. Applications may want to choose to make URI references or blank nodes click-able or retrieve an appropriate label for them. The method `explanationNodes()` returns the list of RDF nodes. The method `parts()` returns a collection containing the child explanation parts of the current part.

An explanation part can provide additional details about itself. These details are represented as a further explanations part. As any explanation part, this part may also have child parts. The method `getDetails()` returns this explanations part, or null if none exists.

Beside of textual explanations, the WIQA engine can also generate RDF explanations (see Section 10.3). These are simply RDF graphs that contain information about the filtering process in a vocabulary chosen by the policy. RDF explanations are retrieved by calling the `explainAsGraph()` method of an `AcceptedGraph`.

Textual and RDF explanations are cached inside `AcceptedGraph`s. This consumes memory over time. The `clearCachedExplanations()` method discards all cached explanations. The next call to the `find()` method will start to fill the cache again. The WIQA framework provides an `ExplanationToHTMLRenderer` which can be used to generate basic HTML representations of explanations.

Implementing and Registering Extension Functions

The WIQA-PL policy language can be extended with domain-specific extension functions (see Section 9.6). Extension functions are implemented as plug-ins for the WIQA - Filtering and Explanation Engine. At implementation level, two different types of extension functions are distinguished: Basic functions and extensions. The input of a basic function is a single matching solution and a number of arguments which are RDF nodes. The output is a boolean value. The input of an extension is a stream of matching solu-

tions and a number of arguments which are RDF nodes. The output is a modified version of the input stream. The `wiqa:MorePositiveRatings` and the `wiqa:TidalTrust` extension functions are implemented as basic functions. The `wiqa:count` extension function is implemented as an extension as it requires access to the complete solution set in order to count solutions.

Basic functions have to extend the abstract class `ExplainableFunction`. A basic function has to implement the following methods: `exec()` which does the actual calculation, and `returnExplanation()` which may return a custom explanation for the function.

Extensions have to extend the abstract class `ExplainableExtension`. An extension has to implement the `exec()`, `finish()`, and the `returnExplanation()` methods. The function `exec()` is called once for every solution in the solution stream. `finish()` is called after the last solution of the stream. The expected return values of both functions are iterators over matching solutions. An extension can choose when it returns the matching solutions, it might return some for each `exec()` call, or all for the `finish()` call.

Basic functions and extensions have access to the unfiltered graph set and can use it to retrieve additional information like ratings or background information about information providers. The graph set is accessed from within a basic function or extension by calling the method `getDataset()`.

Basic functions and extensions must be registered with the `FunctionRegistry` and `ExtensionRegistry` respectively, before they can be used in WIQA-PL policies. A function is registered by calling `FunctionRegistry.get().put("functionURI", Implementation.class)`, where `functionURI` is the URI identifying the function and `Implementation.class` is a Java class implementing the function. Extensions are registered by the same call on `ExtensionRegistry`.

11.2.2 Usage Example

The example code shown in Figure 11.4 illustrates the usage of the WIQA - Filtering and Explanation Engine. Lines 10-11 create a new `NamedGraphSet` and load a TriX file into the graph set. Lines 14-17 load a policy suite and select a policy from the suite. Line 19-22 create an `AcceptedGraphFactory` and assign a value to the context variable `?USER`. Line 25 creates an `AcceptedGraph` by applying the selected policy against the graph set. Lines 28-30 create an iterator over all triples in the accepted graph that have the subject `http://richard.cyganiak.de/foaf.rdf#RC`. Line 37 creates an explanation why an accepted triple satisfies the policy. This explanation is rendered to HTML and written to `System.Out` in lines 40-42.

```
1.  import java.util.Iterator;
2.  import com.hp.hpl.jena.graph.Node;
3.  import com.hp.hpl.jena.graph.Triple;
4.  import de.fuberlin.wiwiss.ng4j.*
5.
6.  // Create a new graphset
7.  NamedGraphSet graphset = new NamedGraphSetImpl();
8.
9.  // Create a new NamedGraph in the NamedGraphSet
10. NamedGraph graph =
11.   graphset.createGraph("http://example.org/persons/123");
12.
13. // Add information to the NamedGraph
14. graph.add(new Triple(
15.    Node.createURI("http://richard.cyganiak.de/foaf.rdf#RC"),
16.    Node.createURI("http://xmlns.com/foaf/0.1/name") ,
17.    Node.createLiteral("Richard Cyganiak", null, null)));
18.
19. // Create a quad
20. Quad quad = new Quad(
21.   Node.createURI("http://www.bizer.de/InformationAboutRichard"),
22.   Node.createURI("http://richard.cyganiak.de/foaf.rdf#RC"),
23.   Node.createURI("http://xmlns.com/foaf/0.1/mbox") ,
24.   Node.createURI("mailto:richard@cyganiak.de"));
25.
26. // Add the quad to the graphset. This will create a new NamedGraph
27. // in the graphset.
28. graphset.addQuad(quad);
29.
30. // Find information about Richard across all graphs in the graphset
31. Iterator it = graphset.findQuads(
32.    Node.ANY,
33.    Node.createURI("http://richard.cyganiak.de/foaf.rdf#RC"),
34.    Node.ANY,
35.    Node.ANY);
36.
37. // Output the results of findQuads()
38. while (it.hasNext()) {
39.    Quad q = (Quad) it.next();
40.    System.out.println("Source: " + q.getGraphName());
41.    System.out.println("Statement: " + q.getTriple());
42. }
43.
44. // Serialize the graphset to System.out, using the TriX syntax
45. graphset.write(System.out, "TRIX", null);
46.
```

Figure 11.3: NG4J - Named Graph API for Jena usage example.

```
1.  import java.util.Iterator;
2.  import java.util.List;
3.  import com.hp.hpl.jena.graph.Node;
4.  import com.hp.hpl.jena.graph.Triple;
5.  import de.fuberlin.wiwiss.ng4j.NamedGraphSet;
6.  import de.fuberlin.wiwiss.ng4j.impl.NamedGraphSetImpl;
7.  import de.fuberlin.wiwiss.wiqa.*;
8.
9.  // Create a new graph set and read a TRIG file into the graph set
10. NamedGraphSet graphset = new NamedGraphSetImpl();
11. graphset.read("file:graphset.trig", "TRIG");
12.
13. // Read a WIQA policy suite and get a policy from the policy suite
14. List policysuite =
15.    PolicyParser.parseSuiteFromFile("file:policies.wiqa");
16. Policy policy =
17.    policysuite.get(Node.createURI("http://example.org/policy1");
18.
19. // Create a graph factory and set a context variable
20. AcceptedGraphFactory factory = new AcceptedGraphFactory(graphset);
21. factory.setContextVariable("USER",
22.   Node.createURI("http://www.bizer.de/i"));
23.
24. // Filter the graph set using the policy
25. AcceptedGraph acceptedGraph = factory.createAcceptedGraph(policy);
26.
27. // Find information about Richard in the accepted graph
28. Iterator it = acceptedGraph.find(
29.    Node.createURI("http://richard.cyganiak.de/foaf.rdf#RC"),
30.    Node.ANY, Node.ANY);
31.
32. // Get the first triple from the result
33. if (it.hasNext()) {
34.    Triple triple = (Triple) it.next();
35.
36.    // Get an explanation why the triple fulfills the policy
37.    Explanation explanation = acceptedGraph.explain(triple);
38.
39.    //Output a HTML representation of the explanation
40.    ExplanationToHTMLRenderer renderer =
41.        new ExplanationToHTMLRenderer(explanation, graphset);
42.    System.out.println(renderer.getExplanationAsHTML());
43. }
```

Figure 11.4: WIQA - Filtering and Explanation Engine usage example.