iPlant Semantic Web Services Requirements Document

Clark & Parsia
University of Arizona

May 20, 2010

Abstract

Contents

Todo list

1 Introduction

This document describes requirements for the iPlant semantic web infrastructure work done in collaboration between Clark & Parsia and University of Arizona. In particular we describe the requirements of the HTTP and Java APIs for SSWAP, and the requirements for the Discovery Server in the SSWAP architecture. SSWAP is intended as a lightweight protocol for semantic resource discovery, query, and publishing. It is based on W3C and IETF standards, including RDF, OWL, and HTTP.

1.1 Components of the system

There are four major groups of functionalities in the SSWAP system: functionalities for SSWAP clients, functionalities for providers, the Discovery Server, and ontology management. (The functionalities of the Discover Server partially overlap with the functionalities for providers because the Discovery Server is treated as a provider in SSWAP architecture).

Additionally, the project involves developing two kinds of APIs for both clients and providers: the HTTP API, and the Java API. The HTTP API is a higher level layer that will work on the top of the lower-level Java API. The HTTP API is oriented for an audience that is not necessarily proficient in SSWAP or OWL. On the other hand, the Java API is oriented at developers, who are knowledgeable about the SSWAP protocol, and will allow direct modification of SSWAP data structures.

1.2 How this document is organized

This document first discusses the requirements for the providers (both HTTP and Java APIs), then it lists the requirements for the SSWAP clients (again, both HTTP and Java APIs). Further, the requirements for the Discovery Server are presented, which will be built by using the APIs for service providers. Later, the document includes the requirement for ontology management in SSWAP. At the end of the document, there is a table of all requirements, and a section with acronyms.

1.3 RFC 2119 Keywords

This document uses keywords as described in IETF RFC 2119[1] when they are formatted thusly: `must`, `must not`, `shall`, `shall not`, `should`, `should not`, `may`, `recommended`, `optional`, `required`.

[1] `http://www.ietf.org/rfc/rfc2119.txt`.

2 Data and Service Provider HTTP API

HTTP API for providers attempts at allowing providers of semantic services and data to participate in SSWAP architecture without significant knowledge about SSWAP internals. The implementation of HTTP API will typically be using an underlying Java API (see Section 3) for most complex tasks (esp. the ones requiring SSWAP graph parsing or semantic reasoning). Additionally, HTTP API will be a proxy invoking the provider's service implementation, when a SSWAP request (RIG) arrives.

2.1 `RG1` Functional Requirements for Provider HTTP API

`R1.1` Create or edit RDG (HTTP API)

The HTTP API should allow the provider to create an RDG for the service that provider wants to make accessible in SSWAP architecture. The API `should` limit the exposure of the provider to the underlying SSWAP, RDF, and OWL internals as much as possible. The service `must` allow the provider to create a new RDG graph, modify an existing RDG graph, and inspect the contents of an existing RDG graph (details are included below in the subrequirements `R1.1.1` and `R1.1.2`).

`R1.1.1` **Insert data into RDG (HTTP API)** The HTTP API `must` provide a method that will accept parameters describing the desired RDG graph, and will produce that RDG graph (unless the parameters provided by the user are unacceptable). Additionally, the service `must` also support the use-case when the provider optionally provides an existing RDG graph. In such a case, the service will edit that provided RDG graph by replacing the existing information with the parameters provided by the user (details provided below).

The service `must` support the following parameters (parts of the RDG graph):

- `sswap:providedBy`
- `sswap:Resource`
- `sswap:name`
- `sswap:oneLineDescription`
- `sswap:metadata`
- `sswap:aboutURI`
- `sswap:inputURI`
- `sswap:outputURI`

Additionally, the service must support the previously mentioned capability of using an existing graph (provided by the user by providing an additional, optional `sswap` parameter to the request). If the client provides this parameter, it `must` be a valid RDG.

An acceptable request to the service `must` include the `providedBy` parameter (containing the URI of the service, which does not need to be dereferencable at this time). Additionally, if the client provides a `sswap:Resource` parameter, it must be a valid resource. If the parameters provided by the user are not acceptable the service `must not` produce an RDG, but it `must` provide an HTTP error 406 (Not acceptable).

In case the client provides both the RDG graph and the `sswap:Resource`, the service `must` use the provided `sswap:Resource` in the final graph. If the client provided only the `sswap:Resource` and no graph, the service `must` first try to treat the `sswap:Resource` as the URL to an RDG graph; if that URL is valid, it should use that graph as if this graph were provided by the by the client, and no resource were provided otherwise; if the URL is not valid, it should use the canonical graph with the `sswap:Resource` provided by the client.

The RDG graph produced by the service `must` be a valid graph. In particular, it needs exactly one `sswap:providedBy` element, exactly one `sswap:name` element, exactly one `oneLineDescription`, at most one `metadata`, at most one `aboutURI`, at most `inputURI`, and at most `outputURI`. If it is impossible to produce such a graph (because of the parameters provided by the user, including the provided version of the RDG graph), the service `must` provide an HTTP 406 error (Not acceptable).

`R1.1.2` **Retrieve data from RDG (HTTP API)** The HTTP API `must` provide a method that will process an existing RDG, and will return information about the following features of the RDG graph:

- `sswap:providedBy`
- `sswap:Resource`
- `sswap:name`
- `sswap:oneLineDescription`
- `sswap:metadata`
- `sswap:aboutURI`
- `sswap:inputURI`
- `sswap:outputURI`

The method provided by the API `must` provide all this information by using a single HTTP call of the client.

`R1.2` Create or edit PDG (HTTP API)

The HTTP API should allow the provider to create an PDG for the provider in SSWAP. The API `should` limit the exposure of the provider to the underlying SSWAP, RDF, and OWL internals as much as possible. The service `must` allow the provider to create a new PDG, modify an existing PDG, and inspect the contents of an existing PDG graph (details are included below in the subrequirements `R1.2.1` and `R1.2.2`).

`R1.2.1` **Insert data into PDG (HTTP API)** The HTTP API `must` provide a method that will allow a provider to create a PDG. The method `must` accept parameters describing the desired PDG graph, and will produce that PDG graph as the result (unless the parameters provided by the user are unacceptable). Additionally, the service `must` also support the use-case when the provider optionally provides an existing PDG graph. In such a case, the service will edit that provided PDG graph by replacing the existing information with the parameters provided by the user (details provided below).

The service `must` support the following parameters (parts of the PDG graph):

- `sswap:Provider` (URL)
- `sswap:name`
- `sswap:oneLineDescription`
- `sswap:aboutURI`
- `sswap:metadata`
- `sswap:Resource`

Additionally, the service must support the previously mentioned capability of using an existing graph (provided by the user by providing an additional, optional `sswap` parameter to the request). If the client provides this parameter, it `must` be a valid PDG.

An acceptable request must provide `sswap:Provider`, `sswap:name` and `sswap:oneLineDescription` (either as a parameters or by providing an existing PDG). Additionally, the request must include at most one `sswap:aboutURI`, at most one `sswap:metadata`, and any number of `sswap:Resource` (including no elements of this type). If the request is not acceptable, the service must return the 406 HTTP code (Not acceptable).

`R1.2.2` **Retrieve data from PDG (HTTP API)** The HTTP API `must` provide a method that will retrieve the following information from an existing PDG:

- `sswap:Provider` (URL)
- `sswap:name`
- `sswap:oneLineDescription`
- `sswap:aboutURI`
- `sswap:metadata`
- `sswap:Resource`

The method provided by the API `must` provide all this information by using a single HTTP call of the client.

R1.3 Validate SSWAP graph (HTTP API)

The HTTP API `must` provide a method that will check the validity of a SSWAP graph provided by the client. This service `must` check the syntax of the graph representation (RDF/XML), and the conformance of the encoded data to SSWAP structure. If the given graph is validated successfully the service `must` return the 202 (Accepted) HTTP code. If the graph cannot be successfully validated, the service `must` return the 406 (Not Acceptable) HTTP code.

R1.4 Accept a SSWAP request over HTTP

The HTTP API `should` provide a provider a front-end that will accept a SSWAP request, provide the support for invoking the actual service provided by the provider, and format the results back as a proper SSWAP response. This requirement consists of three sub-requirements.

R1.4.1 **Accept an HTTP request with RIG** The API `shall` accept an SSWAP HTTP request, which should contain a serialized RIG. The request `should` be accepted at the URL that is published in the RDG for that service's DS. The HTTP request `may` be either a POST or a GET request. In either case, the query parameters are provided as an RIG, which is passed as a parameter for the HTTP request. The parameter for RIG will contain the representation of the graph serialized in either RDF/XML or Turtle.

An example RIG query (in RDF/XML syntax) is available at: `http://sswap.info/sswap/resources/queryForResources/queryForResources`

The query may also contain regular HTTP headers for content negotiation, which will specify the type of representation for the result (RRG) requested by the client. The available options for the RRG representation will be RDF/XML and Turtle. If the representation requested by the client is not available or unknown, the DS `must` return 406 "Not Acceptable" HTTP code.

R1.4.2 **Parse and forward the request to the provider** The HTTP API `must` parse the request accepted in R1.4.1 and ensure that it is syntactically valid given its MIME type for representation (e.g., RDF/XML syntax or Turtle). If the request is in a format the API cannot support, it `must` return 415 "Unsupported Media Type" HTTP code. If the request cannot be parsed according to the specified representation, the server `must` return 400 "Bad Request" HTTP code.

After it is ensured that the request is syntactically correct the API should forward the request to the provider's implementation of the service. Since the provider may be developed in a totally different architecture (e.g., programming language) than the HTTP API, it is necessary to define a standard of communication between the provider and the API.

Additionally, since the request may be using different vocabulary than the one expected by the provider, HTTP API should provide services for translating one vocabulary into the other. (In general, the translation should be done by the provider, rather than requiring the clients to perform translation on their own.)

R1.4.3 **Construct an RRG and return it as the response (HTTP API)** After the provider's implementation of the service returns it results, the HTTP API should construct an RRG based on these results, and send it back to the client in the format that the client requested (see `R1.4.2`). If the provider's implementation of service terminates with an error, HTTP API should return the 500 HTTP response code ("Internal Server Error").

2.2 `RG2` Non-Functional Requirements for Provider HTTP API

`R2.1` The implementation `should` be resilient to graph format changes as much as possible

The implementation `should not` hard code the canonical graph inside of the code (i.e., it `should` retrieve it from a predefined location). Moreover, the code `should` be able to tolerate changes in the graph format as much as possible.

`R2.2` The implementation `should` conform to REST constraints as much as possible

The HTTP Provider API `should` conform as much as possible to REST constraints; that is:

- `should` not expose internal state to clients via side-effects,
- `should` use hypertextual resource representations for navigation through program state; i.e., embedded URLs for further service operations;
- `should` be cacheable as much as possible (especially for GET requests), and the responses that `must not` not be cached, `must` be marked as such,
- the design of the system `should` allow the layered configuration (e.g., by use of proxies/caches)

3 Data and Service Provider Java API

Java API for Data and Service Providers is a lower level API than HTTP API (see Section 3). It provides more specialized services that are meant to be called by users who are more comfortable with SSWAP internals (as well as internals of semantic reasoning).

3.1 `RG3` Functional Requirements for Provider Java API

`R3.1` Create or edit RDG (Java API)

The Java API `must` allow the provider to create a Java object representing an RDG for the service that provider wants to make accessible in SSWAP architecture, and then allow to create appropriate directory structure for publishing that RDG (i.e., create a directory structure with appropriate RDF/XML files for all data in the RDG, partitioned according to SSWAP protocol; details are included in the subrequirement `R3.1.3`). The service `must` allow the provider to create a new RDG graph, modify an existing RDG graph, and inspect the contents of an existing RDG graph (details are included below in the subrequirements `R3.1.1` and `R3.1.2`).

`R3.1.1` **Insert data into RDG (Java API)** The Java API `must` provide methods to create a new object representing an RDG, as well as create an object based on an existing RDG published on the web. Additionally, the Java API `must` provide methods to insert new data into an RDG object (e.g., new subject/object mappings) and overwrite existing data (when using an already existing RDG data).

The API `must` provide methods to set the following parts of the RDG graph:

- `sswap:providedBy`
- `sswap:Resource`
- `sswap:name`
- `sswap:oneLineDescription`
- `sswap:metadata`
- `sswap:aboutURI`
- `sswap:inputURI`
- `sswap:outputURI`

`R3.1.2` **Retrieve data from RDG (Java API)** The Java API `must` provide methods that will extract data from an existing RDG. In particular, it `must` provide methods that will retrieve the following parts of the RDG graph:

- `sswap:providedBy`
- `sswap:Resource`
- `sswap:name`
- `sswap:oneLineDescription`
- `sswap:metadata`
- `sswap:aboutURI`
- `sswap:inputURI`
- `sswap:outputURI`

`R3.1.3` **Create directory structure for publishing RDG** THe Java API `must` provide a method to create the RDF/XML files necessary to publish the RDG online (along with the necessary directory structure). The files should contain semantic data contained in the RDG that is partitioned according to SSWAP protocol. The whole directory structure `should` be saved into an archive (e.g., a ZIP archive), so that it can be conveniently transferred to a host, where the data will be unpacked, and published.

R3.2 Create or edit PDG (Java API)

The Java API `must` allow the provider to create a Java object representing an PDG for the provider in SSWAP architecture, and then allow to create appropriate directory structure for publishing that PDG (i.e., create a directory structure with appropriate RDF/XML files for all data in the PDG, partitioned according to SSWAP protocol; details are included in the subrequirement R3.2.3). The service `must` allow the provider to create a new PDG, modify an existing PDG graph, and inspect the contents of an existing PDG (details are included below in the subrequirements R3.2.1 and R3.2.2).

R3.2.1 Insert data into PDG (Java API) The Java API `must` provide methods to create a new object representing an PDG, as well as create an object based on an existing PDG published on the web. Additionally, the Java API `must` provide methods to insert new data into an PDG object (e.g., URLs to new published resoures) and overwrite existing data (when using an already existing PDG data).

The API `must` provide methods to set the following parts of the PDG:

- `sswap:Provider` (URL)
- `sswap:name`
- `sswap:oneLineDescription`
- `sswap:aboutURI`
- `sswap:metadata`
- `sswap:Resource`

The design of the API `must` take into account the fact that creating a resource (a part of RDG; see requirement R3.1) requires to specify information about the provider (i.e., provide URL to the provider's PDG), and PDG can contain information about the resources (by providing URL to the RDG). The API `must` provide means to resolve this circular dependency.

R3.2.2 Retrieve data from PDG (Java API) The Java API `must` provide methods that will extract data from an existing PDG. In particular, it `must` provide methods that will retrieve the following parts of the PDG graph:

- `sswap:Provider` (URL)
- `sswap:name`
- `sswap:oneLineDescription`
- `sswap:aboutURI`
- `sswap:metadata`
- `sswap:Resource`

R3.2.3 **Create directory structure for publishing PDG** THe Java API `must` provide a method to create the RDF/XML files necessary to publish the PDG online (along with the necessary directory structure). The files should contain semantic data contained in the PDG that is partitioned according to SSWAP protocol. The whole directory structure `should` be saved into an archive (e.g., a ZIP archive), so that it can be conveniently transferred to a host, where the data will be unpacked, and published.

R3.3 **Validate SSWAP graph (Java API)**

The Java API `must` provide a method that will check the validity of a SSWAP graph. This method should parse the graph (syntactic validation) if necessary, and then the conformance of the encoded data to SSWAP structure.

R3.4 **Accept a SSWAP request (RIG)**

The Java API `must` provide a method that would accept a serialized RIG, parse it, create a corresponding RIG Java object, invoke the underlying provider's code, and allow the response to be created (in the form of RRG). This requirement consists of three sub-requirements.

R3.4.1 **Accept a service request and create RIG** The Java API `must` provide a method that will parse the serialized RIG data and create a Java object based on that data.

R3.4.2 **Validate RIG and forward the request to the provider implementation** The Java API `must` provide a method to validate the content of the created RIG object. As a part of validation, this method `must` first verify whether the RIG's structure follows all the rules of a valid SSWAP graph (see R3.3). Next, it `must` verify whether the RIG's URL matches the service URL (as described by the service's RDG). The validation `must` also check whether the subject/object pattern in the RIG matches the pattern in the RDG, and finally the method `must` check whether the RIG's subject is a subclass of RDG's subject (see R3.5). The last step `may` involve retrieving additional semantic data (since the client may use additional/extended vocabulary (see R3.6).

If the RIG is determined to be a valid RIG, the request should be forwarded to the actual provider implementation.

R3.4.3 **Construct an RRG and return it as the response (Java API)** After the provider's implementation of the service returns it results, the Java API `must` construct an RRG object, fill it with the results, serialize into the requested format (e.g., RDF/XML), and return it.

R3.5 **Provide type reasoning services**

The Java API `must` provide a semantic reasoning service. The service `must` provide methods to answer queries whether one class is a subclass of another. The service `should` use Pellet OWL reasoner internally[2].

[2] http://clarkparsia.com/pellet

R3.6 Retrieve semantic closure

Since RDF documents (including PDG or RDG) can reference terms defined in other document, Java API `must` provide a method to retrieve the undefined terms. (This requirement is simplified by the fact that in SSWAP the URIs of every concept is a valid URL, from which the definition can be retrieved.) Additionally, since the retrieved definitions of the terms can reference other terms, the API should recursively try to retrieve their definitions (until all definitions are known, or a specified termination condition has been reached; e.g., time limit)

3.2 RG4 Non-functional Requirements for Provider Java API

R4.1 Hide complexities of underlying semantic framework API

The Java API `should` not expose the complexities of the underlying semantic framework (e.g., Jena or OWLAPI) to the user. Instead, it `should` wrap all the calls to the underlying framework, and expose only the functionalities necessary to accomplish the requirements described in this document.

R4.2 Support for anonymous complex classes

Java API `must` be able to handle anonymous complex classes (i.e., the classes that are expressed in terms of other classes and operators like owl:unionOf, owl:intersectionOf, or owl:complementOf).

4 Data and service requestors (clients)

Requirements for the client API

5 Discovery Server

Discovery Server (DS) is the center of SSWAP; it is used by clients to discover the services of the providers that are registered with an instance of DS. Additionally, DS is a service provider; as such, it is also discoverable through its own interface, and uses the Provider API (see Sections 2 and 3). DS accepts queries that involve they type of service requested, as well as the types of input and output parameters.

5.1 RG5 Functional requirements

R5.1 Perform semantic queries over HTTP

The DS `shall` provide a semantic query functionality via HTTP at the URL provided in its RDG (which is accessible as described in R6.1). This requirement consists of three sub-requirements.

R5.1.1 Accept an HTTP request with RIG/RQG The DS `shall` accept an HTTP request with a query for resource(s); that query `shall` be serialized as an RQG. The request `should` be accepted at the URL that is published in the RDG for the DS. The DS should use the HTTP API implementation to accept this query (see R1.4.1).

R5.1.2 **Parse and handle the query request** The DS shall parse the request accepted in R5.1.1 and ensure that it is syntactically valid given its MIME type for representation (e.g., RDF/XML syntax or Turtle). The DS should use the HTTP API implementation to perform the parsing (see R1.4.2)

After the parsing of the request is finished, and the HTTP API transfers the control to the DS implementation, the DS should consult its internal registry of providers; and, using OWL reasoning (Pellet reasoner; see R6.2, determine which individuals (RDG) match the query. A successful match between a service RDG and the request's RQG is achieved when all of the following conditions are satisfied:

- the type of service (*sswap:Resource*) is the same as in the RDG, or the type of service is a subclass of the service;
- the parameters to the service (*sswap:Subject*) are the same as in RDG, or the type of a type is a superclass of the type of the parameter provided in RQG; and
- the results of the service (*sswap:Object*) are the same as in RDG, or the type of the result is a subclass of the type of the parameter.

R5.1.3 **Construct an RRG and return it as the response** After determining which services known to the server match the query (that is, which RDG match the RQG), the server should form an RRG containing them, and return the RRG over the HTTP connection in an appropriate representation as requested by the client (see R5.1.1. The return RRG mustcontain the following information (as listed by the current RDG for the DS).

- name of the service,
- one line description of the service,
- information about the provider of the service (which will contain all the necessary information for the client to invoke the service at the provider, including the URL of the service).

The implementation of this requirement should use the HTTP API to return the RRG (see R1.4.3).

R5.2 Publish an RDG to the DS

The DS must support addition of an RDG of a published service to the DS. The addition of a new RDG to the DS consists of

1. Submitting the URL of the RDG to the DS by the provider
2. Verifying the accuracy of the RDG by the DS (see R1.3) (Note: if the given URL cannot be dereferenced, and the DS knew a service at that URL, it may lead to removing this service from the DS; see R5.3.)
3. Verifying whether there exists a service at the resource given in the RDG
4. If the DS knew another RDG published at that URL, the new RDG will replace the old RDG.

R5.3 Delete a published RDG from the DS

The DS `must` allow removing an existing RDG from the DS. An RDG is removed from the DS when a user attempts to publish an RDG (see R5.2) with a URL that is known by the DS, but the DS will find out that the URL is no longer dereferencable.

5.2 RG6 Non-functional requirements

R6.1 RDG of the Discovery Server `must` be available via HTTP

The DS `must` provide its RDG to be downloaded via HTTP. The RDG for the DS `should` be available at `http://sswap.info/sswap/resources/queryForResources/queryForResources`.

R6.2 Semantic query `should` be performed by Pellet

Pellet reasoner `should` be included in the implementation of the server, so that it can reason about subclass and superclass relationships of the concepts provided in RQGs sent by the clients (see R5.1.2) using the ontologies known to the DS.

Add a reference here to the requirement for managing ontologies known to the discovery server, when we have written those requirements

R6.3 The implementation `should` conform to REST constraints as much as possible

The internal implementation of the Discovery Server `should` conform as much as possible to REST constraints; that is:

- `should` not expose internal state to clients via side-effects,
- `should` use hypertextual resource representations for navigation through program state; i.e., embedded URLs for further service operations;
- `should` be cacheable as much as possible (especially for GET requests), and the responses that `must not` not be cached, `must` be marked as such,
- the design of the system `should` allow the layered configuration (e.g., by use of proxies/caches)

6 Ontology providers

Requirements for ontology providers and ontology management

7 Table of all requirements

- `R1.1.2` Retrieve data from RDG (HTTP API)
- `R1.2` Create or edit PDG (HTTP API)
  - `R1.2.1` Insert data into PDG (HTTP API)
  - `R1.2.2` Retrieve data from PDG (HTTP API)
- `R1.3` Validate SSWAP graph (HTTP API)
- `R1.4` Accept a SSWAP request over HTTP
  - `R1.4.1` Accept an HTTP request with RIG
  - `R1.4.2` Parse and and forward the request to the provider
  - `R1.4.3` Construct an RRG and return it as the response (HTTP API)

`RG2` Non-functional requirements for Provider HTTP API

- `R2.1` The implementation `should` be resilient to graph format changes as much as possible
- `R2.2` The implementation `should` conform to REST constraints as much as possible

`RG3` Functional requirements for Provider Java API

- `R3.1` Create or edit RDG (Java API)
  - `R3.1.1` Insert data into RDG (Java API)
  - `R3.1.2` Retrieve data from RDG (Java API)
  - `R3.1.3` Create directory structure for publishing RDG
- `R3.2` Create or edit PDG (Java API)
  - `R3.2.1` Insert data into PDG (Java API)
  - `R3.2.2` Retrieve data from PDG (Java API)
  - `R3.2.3` Create directory structure for publishing PDG
- `R3.3` Validate SSWAP Graph (Java API)
- `R3.4` Accept a SSWAP request (RIG)
  - `R3.4.1` Accept a service request and create RIG
  - `R3.4.2` Validate RIG and forward the request to the provider implementation
  - `R3.4.3` Construct an RRG and return it as the response (Java API)
- `R3.5` Provide type reasoning services
- `R3.6` Retrieve semantic closure

`RG4` Non-functional requirements for Provider Java API

- `R4.1` Hide complexities of underlying semantic framework API
- `R4.2` Support for anonymous complex classes

`RG5` Functional requirements for Discovery Server

- `R5.1` Perform semantic queries over HTTP
  - `R5.1.1` Accept an HTTP request with RIG/RQG
  - `R5.1.2` Parse and handle the query request
  - `R5.1.3` Construct an RRG and return it as the response
- `R5.2` Publish an RDG to the DS
- `R5.3` Delete a published RDG from the DS

`RG6` Non-functional requirements for Discovery Server

- `R6.1` RDG of the Discovery Server `must` be available via HTTP
- `R6.2` Semantic query `should` be performed by Pellet
- `R6.3` The implementation `should` conform to REST constraints as much as possible

8 Acronyms

This chapter lists all the acronyms used in this document:

DS Discovery Server

PDG Provider Description Graph

RDG Resource Description Graph

RIG Resource Invokation Graph

RRG Resource Response Graph

RQG Resource Query Graph