Towards a Basic Profile for Linked Data

A collection of best practices and simple approach for a Linked Data architecture

Abstract

W3C defines a wide range of standards for the Semantic Web and Linked Data suitable for many possible use cases. While using Linked Data as an application integration technology in the Application Lifecycle Management (ALM) domain IBM has found that there often are several possible ways of applying the existing standards and little guidance is provided on how to combine them. Typical use cases include accessing, updating and creating resources from servers that expose their resources as Linked Data. This document discusses motivating background information and a proposal for a Basic Profile for Linked Data.

Motivation

There is interest in Linked Data technologies for more than one purpose. We have seen interest for the purpose of exposing information – for example public records – on the Internet in a machine-readable format. We have also seen interest in the use of Linked Data for inferring new information from existing information, for example in pharmaceutical applications or IBM Watson. The IBM Rational team has been using Linked Data as an architectural model and implementation technology for application integration.

IBM Rational is a vendor of software development tools, particularly those that support the general software development process such as bug tracking, requirements management and test management tools. Like many vendors who sell multiple applications, we have seen strong customer demand for better support of more complete business processes - in our case software development processes - that span the roles, tasks and data addressed by multiple tools. This demand has existed for many years, and our industry has tried several different architectural approaches to address the problem. Here are a few:

1. Implement some sort of application programming interface (API) for each application, and then, in each application, implement "glue code" that exploits the APIs of other applications to link them together.
2. Design a single database to store the data of multiple applications, and implement each of the applications against this database. In the software development tools business, these databases are often called "repositories".
3. Implement a central "hub" or "bus" that orchestrates the broader business process by exploiting the APIs described in option 1.

While a discussion of the failings of each of these approaches is outside the scope of this document it is fair to say that although each one of them has its adherents and can point to some successes, none of them is wholly satisfactory. So, as an alternative, we have been exploring over the last 5 years the use of Linked Data as an application integration

technology. We have shipped a number of products using this technology and are generally pleased with the result. We have more products in development that use these technologies and we are also seeing a strong interest in this approach in other parts of our company.

We are pleased – even passionate – about the results we have seen using Linked Data as an integration technology but we have found successful adoption to be difficult. It has taken us a number of years of experimentation to achieve the level of understanding that we have today, we have made some costly mistakes along the way, and we see no immediate end to the challenges and learning that lie before us. As far as we can tell, there is only a very limited number of people trying to use Linked Data technologies in the ways we are using them, and the little information that is available on best practices and pitfalls is widely dispersed. We believe that Linked Data has the potential to solve some important problems that have frustrated the IT industry for many years, or at least make significant advances in that direction, but this potential will only be realized if we can establish and communicate a much richer body of knowledge on how to exploit these technologies. In some cases, there also are gaps in the Linked Data standards that need to be addressed. To help with this process, we would like to share information on how we are using these technologies, the best practices and anti-patterns we have identified, and the specification gaps we have had to fill ourselves.

The best practices and anti-patterns can be categorized (but are not limited) to the following:

- **Resources** - a summary of the HTTP and RDF standard techniques and best practices that you should use, and anti-patterns you should avoid, when constructing clients and servers that read and write linked data.
- **Containers** - defines resources that allow new resources to be created using HTTP POST and existing resources to be found using HTTP GET
- **Paging** - defines a mechanism for splitting the information in large resources into pages that can be fetched incrementally
- **Validation** - defines a simple mechanism for describing the properties that a particular type of resource must or may have

The following sections provide details regarding this proposal for a “Basic Profile for Linked Data”.

Related Topics

This publication has a number of related efforts that accompany it. The intent of this publication is to promote some ideas and motivate specification efforts in potentially a number of communities. It is worth elaborating on these relationships:

W3C Linked Enterprise Data Patterns Workshop – This proposal is intended to elaborate on what is seen as missing or needed as referenced by one of IBM’s position papers to the workshop.

Open Services for Lifecycle Collaboration (OSLC) – The OSLC Core V2 specification defines some of these patterns and anti-patterns, perhaps not in an ideal way. This proposal can provide the basis for a simpler and more standards aligned way for future OSLC specifications.

Terminology

Terminology is based on W3C's Architecture of the World Wide Web and Hyper-text Transfer Protocol (HTTP/1.1).

Link : A relationship between two resources when one resource (representation) refers to the other resource by means of a URI. (reference: WWWArch)

Linked Data : Defined by Tim Berners-Lee as four rules: 1) Use URIs as names for things 2) Use HTTP URIs so that people can look up those names. 3) When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL) 4) Include links to other URIs. so that they can discover more things.(reference: LinkedData)

Specification : An act of describing or identifying something precisely or of stating a precise requirement

Basic Profile : A specification that defines the needed specification components from other specifications as well as providing clarifications and patterns. Within the "Basic Profile for Linked Data", it is sometimes referred to as a shortened "Basic Profile".

Client : A program that establishes connections for the purpose of sending requests. (reference: HTTP)

Basic Profile Client : A client that adheres to the rules defined in the Basic Profile.

Server: An application program that accepts connections in order to service requests by sending back responses. Any given program may be capable of being both a client and a server; our use of these terms refers only to the role being performed by the program for a particular connection, rather than to the program's capabilities in general. Likewise, any server may act as an origin server, proxy, gateway, or tunnel, switching behavior based on the nature of each request. (reference: HTTP)

Basic Profile Server : A server that adheres to the rules defined in the Basic Profile.

Basic Profile Resources

Basic Profile Resources are HTTP linked data resources that conform to some simple patterns and conventions. Most Basic Profile Resources are domain-specific resources that contain data for an entity in some domain, which could be commercial, governmental, scientific, religious or other. A few Basic Profile Resources are defined by the Basic Profile specifications and are cross-domain. All Basic Profile Resources follow the rules of Linked Data, namely:

1. Use URIs as names for things

2. Use HTTP URIs so that people can look up those names.

3. When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL)

4. Include links to other URIs. so that they can discover more things.

Basic Profile adds a few rules of its own. Some of these rules could be thought of as clarification of the basic linked data rules.

1. **Basic Profile Resources are HTTP resources that can be created, modified, deleted and read using standard HTTP methods**.
   (Clarification or extension of Linked Data Rule 2.) Basic Profile Resources are created by HTTP POST (or PUT) to an existing resource, deleted by HTTP DELETE, updated by HTTP PUT or PATCH, and "fetched" using HTTP GET. Additionally Basic Profile Resources can be created, updated and deleted using SPARQL Update.

2. **Basic Profile Resources use RDF to define their state.**
   (Clarification of Linked Data rule 3.) The state (in the sense of state used in the REST architecture) of a Basic Profile Resource is defined by a set of RDF triples. Binary resources and text resources are not Basic Profile Resources since their state cannot be easily or fully represented in RDF. XML resources may or may not be suitable as Basic Profile Resources. Some XML resources are really data-oriented resources encoded in XML that can be easily represented in RDF. Other XML documents are essentially marked up text documents that aren't easily represented in RDF. Basic Profile Resources can be mixed with other resources in the same application.

3. **You can request an RDF/XML representation of any Basic Profile Resource.**
   (Clarification of Linked Data rule 3.) The resource may have other representations as well. These could be other RDF formats, like Turtle, N3 or NTriples, but non-RDF formats like HTML and JSON would also be popular additions, and Basic Profile sets no limits.

4. **Basic Profile clients use Optimistic Collision Detection on Update.**
   (Clarification of Linked Data rule 2.) Because the update process involves first getting a resource, modifying it and then later putting it back to the server there is the possibility of a conflict, e.g. some other client may have updated the resource since the GET. To mitigate this problem, Basic Profile implementations should use the HTTP `If-Match` header and HTTP ETags to detect collisions.

5. **Basic Profile Resources use standard media types.**
   (Clarification of Linked Data rule 3.) Basic Profile does not require and does not encourage the definition of any new media types. A goal of Basic Profile is that any standards-based RDF or Linked Data client be able to read and write Basic Profile data, and defining new media types would prevent that in most cases.

6. **Basic Profile Resources use standard vocabularies.**
   Basic Profile Resources use common vocabularies (classes, properties, etc) for common concepts. Many web sites define their own vocabularies for common concepts like resource types, label, description, creator, last-modification-time, priority, enumeration of priority values and so on. This is usually viewed as a good feature by users who want their data to match their local terminology and processes, but it makes it much harder for organizations to subsequently integrate information in a larger view. Basic Profile requires all resources to expose common concepts using a common vocabulary for properties. Sites may choose to additionally expose the same values under their own private property names in the same resources. In general, Basic Profile avoids inventing its own property names where possible – it uses ones from popular RDF-based standards like the RDF standards themselves, Dublin Core, and so on. Basic Profile invents property URLs where no match is found in popular standard vocabularies. A number of recommended standard properties for use in Basic Profile Resources are listed below.

7. **Basic Profile Resources set rdf:type explicitly.**
   A resource's membership in a class extent can be indicated explicitly – by a triple in the resource representation that uses the rdf:type predicate and the URL of the class - or derived implicitly. In RDF there is no requirement to place an rdf:type triple in each resource, but this is a good practice, since it makes query more useful in cases where inferencing is not supported. Remember also that a single resource can have multiple values for rdf:type. For example, the dpbedia entry for Barack Obama has dozens of rdf:types. Basic Profile sets no limits to the number of types a resource can have.

8. **Basic Profile Resources use a restricted number of standard datatypes.** RDF does not by itself define datatypes to be used for property values, so Basic Profile lists a set of standard datatypes to be used in Basic Profile. Here is the list:
   - Boolean: a boolean type as specified by XSD Boolean (http://www.w3.org/2001/XMLSchema#boolean, reference: XSD Datatypes).
   - DateTime: a Date and Time type as specified by XSD dateTime (http://www.w3.org/2001/XMLSchema#dateTime, reference: XSD Datatypes)
   - Decimal: a decimal number type as specified by XSD Decimal (http://www.w3.org/2001/XMLSchema#decimal, reference: XSD Datatypes)
   - Double: a double floating-point number type as specified by XSD Double (http://www.w3.org/2001/XMLSchema#double, reference: XSD Datatypes).
   - Float: a floating-point number type as specified by XSD Float (http://www.w3.org/2001/XMLSchema#float, reference: XSD Datatypes).

- Integer: an integer number type as specified by XSD Integer (http://www.w3.org/2001/XMLSchema#integer, reference: XSD Datatypes).
- String: a string type as specified by XSD String (http://www.w3.org/2001/XMLSchema#string, reference: XSD Datatypes).
- XMLLiteral: a Literal XML value (http://www.w3.org/1999/02/22-rdf-syntax-ns#XMLLiteral)

9. **Basic Profile clients expect to encounter unknown properties and content.** Basic Profile provides mechanisms for clients to discover lists of expected properties for resources for particular purposes, but also assumes that any given resource may have many more properties than are listed. Some servers will only support a fixed set of properties for a particular type of resource. Clients should always assume that the set of properties for a resource of a particular type at an arbitrary server may be open in the sense that different resources of the same type may not all have the same properties, and the set of properties that are used in the state of a resource are not limited to any pre-defined set. However, when dealing with Basic Profile Resources, clients should assume that a Basic Profile server may discard triples for properties of which it does have prior knowledge. In other words, servers may restrict themselves to a known set of properties, but clients may not. When doing an update using HTTP PUT, a Basic Profile client must preserve all property-values retrieved using GET that it doesn't change whether it understands them or not. (Use of PATCH or SPARQL Update instead of PUT for update avoids this burden for clients.)

10. **Basic Profile clients do not assume the type of a resource at the end of a link.** Many specifications and most traditional applications have a "closed model", by which we mean that any reference from a resource in the specification or application necessarily identifies a resource in the same specification (or a referenced specification) or application. By contrast, the HTML anchor tag can point to any resource addressable by an HTTP URI, not just other HTML resources. Basic Profile works like HTML in this sense. A HTTP URI reference in one Basic Profile resource may in general point to any resource, not just a Basic Profile resource.

    There are numerous reasons to maintain an open model like HTML's. One is that it allows data that has not yet been defined to be incorporated in the web in the future. Another reason is that it allows individual applications and sites to evolve over time - if clients assume that they know what will be at the other end of a link, then the data formats of all resources across the transitive closure of all links has to be kept stable for version upgrade.

    A consequence of this independence is that client implementations that traverse HTTP URI links from one resource to another should always code defensively and be prepared for any resource at the end of the link. Defensive

coding by clients is necessary to allow sets of applications that communicate via Basic Profile to be independently upgraded and flexibly extended.

11. **Basic Profile servers implement simple validations for create and update.** Basic Profile servers should try to make it easy for programmatic clients to create and update resources. If Basic Profile implementations associate a lot of very complex validation rules that need to be satisfied in order for an update or creation to be accepted, it becomes difficult or impossible for a client to use the protocol without extensive additional information specific to the server that needs to be communicated outside of the Basic Profile specifications. The recommended approach is for servers to allow creation and update based on the sort of simple validations that can be communicated programmatically through a Shape (see constraints section). Additional checks required to implement more complex policies and constraints should result in the resource being flagged as requiring more attention, but should not cause the basic create or update to fail.

    It is possible that some applications or sites will have very strict requirements for complex constraints for data, and that they are unable or unwilling to allow the creation of resources that do not satisfy all those constraints even temporarily. Those applications or sites should be aware that as a consequence they may be making it difficult or impossible for external software to use their interfaces without extensive customization.

12. **Basic Profile resources always use simple RDF predicates to represent links.** By always representing links as simple predicate values, Basic Profile makes it very simple to know how links will appear in representations, and also makes it very simple to query them. When there is a need to express properties on a link, Basic Profile adds an RDF statement with the same subject, object and predicate as the original link, which is retained, plus any additional "link properties". Basic Profile resources do not use "inverse links" to support navigation of a relationship in the opposite direction, since this creates a data synchronization problem and complicates query. Instead, Basic Profile assumes that clients can use query to navigate relationships in the opposite direction from the direction supported by the underlying link.

Common Properties

The following are some properties from well-known RDF vocabularies that are recommended for use in Basic Profile Resources. Basic Profile requires none of them, but a specification based on Basic Profile may require one of these properties or more for a particular resource type.

Commonly used namespace prefixes:

```
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#>.
```

```
@prefix bp:      <http://open-services.net/ns/basicProfile#>.
@prefix xsd:     < http://www.w3.org/2001/XMLSchema#>.
```

From Dublin Core

URI: http://purl.org/dc/terms/

Property	Range	Comment
dcterms:contributor	dcterms:Agent	The identifier of a resource (or blank node) that is a contributor of information. This resource may be a person or group of people, or possibly an automated system.
dcterms:creator	dcterms:Agent	The identifier of a resource (or blank node) that is the original creator the resource. This resource may be a person or group of people, or possibly an automated system.
dcterms:created	xsd:dateTime	The creation timestamp
dcterms:description	rdf:XMLLiteral	Descriptive text about the resource represented as rich text in XHTML format. **SHOULD** include only content that is valid and suitable inside an XHTML element.
dcterms:identifier	rdfs:Literal	A unique identifier for the resource. Typically read-only and assigned by the service provider when a resource is created. Not typically intended for end-user display.
dcterms:modified	xsd:dateTime	Date on which the resource was changed.
dcterms:relation	rdfs:Resource	The URI of a related resource. This is the predicate to use when you don't know what else to use. If you know more specifically what sort o relationship it is, use a more specific predicate.
dcterms:subject	rdfs:Resource	Should be a URI (see dbpedia.org) "Typically, the subject will be represented using keywords, key phrases, or classification codes. Recommended best practice is to use a controlled vocabulary. To describe the spatial or temporal topic of the resource, use the Coverage element." (from Dublin Core)
dcterms:title	rdf:XMLLiteral	A name given to the resource. Represented as rich text in XHTML format. **SHOULD** include only content that is valid inside an XHTML element.

From RDF

URI: `http://www.w3.org/1999/02/22-rdf-syntax-ns#`

Property	Range	Comment
rdf:type	rdfs:Class	The type or types of the resource. Basic Profile recommends that the rdf:type(s) of a resource be set explicitly in resource representations to facilitate query with non-inferencing query engines

From RDF Schema

URI: `http://www.w3.org/2000/01/rdf-schema#`

Property	Range	Comment
rdfs:member	rdf:Resource	The URI (or blank node identifier) of a member of a container
rdfs:label	rdf:Resource	"Provides a human-readable version of a resource name." (Fro RDFS)

Basic Profile Container

Many HTTP applications and sites have organizing concepts that partition the overall space of resources into smaller containers. Blog posts are grouped into blogs, wiki pages are grouped into wikis, and products are grouped into catalogs. Each resource created in the application or site is created within an instance of one of these container-like entities, and users can list the existing artifacts within one. There is no agreement across applications or sites, even within a particular domain, on what these grouping concepts should be called, but they commonly exist and are important. Containers answer two basic questions, which are:

1. To which URLs can I POST to create new resources?
2. Where can I GET a list of existing resources?

In the XML world, Atom Publishing Protocol (APP) has become popular as a standard for answering these questions. APP is not a good match for Linked Data - this specification shows how the same problems that are solved by APP for XML-centric designs can be solved by a simple Linked Data usage pattern with some simple conventions on posting to RDF containers. We call these RDF containers that you can POST to Basic Profile Containers. Here are some of their characteristics:

1. Clients can retrieve the list of existing resources in a Basic Profile Container.
2. New resources are created in a Basic Profile Container by POSTing to it.
3. Any resource can be POSTed to a Basic Profile Container - a resource does not have to be a Basic Profile Resource with an RDF representation to be POSTed to a Basic Profile Container.
4. After POSTing a new resource to a container, the new resource will appear as a member of the container until it is deleted. A container may also contain resources that were added through other means - for example through the user interface of the site that implements the Container.
5. The same resource may appear in multiple containers. This happens commonly if one container is a "view" onto a larger container.
6. Clients can get partial information about a Basic Profile Container without retrieving a full representation including all of its contents.

The representation of a Basic Profile Container is a standard RDF container representation using the rdfs:member predicate. For example, if you have a container with the URL http://example.org/BasicProfile/container1, it might have the following representation:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
<http://example.org/BasicProfile/container1>
   a rdfs:Container ;
```

```
rdfs:member <http://acme.com/members/000000000>;
# … 999999998 more triples here …
rdfs:member <http://acme.com/members/999999999>.
```

Basic Profile does not recognize or recommend the use of other forms of RDF container such as Bag and Seq because they are not friendly to query. This follows standard linked data guidance for RDF usage (e.g. http://linkeddatabook.com/editions/1.0/#htoc16).

rdfs:Container Properties

Basic Profile recommends the use of a set of standard Dublin Core properties with containers. The subject of triples using these properties is the container itself.

Properties whose domain is rdfs:Container:

Property	Occurs	Range	Comment
dcterms:title	zero or one	rdf:XMLLiteral	A name given to the resource. Represented as rich text in XHTML format. **SHOULD** include only content that is valid inside an XHTML element.
dcterms:description	zero or one	rdf:XMLLiteral	Descriptive text about resource represented as rich text in XHTML format. **SHOULD** include only content that is valid and suitable inside an XHTML element.
dcterms:publisher	zero or one	dcterms:Agent	An entity responsible for making the Basic Profile Container and its members available.
bp:containerPredicate	exactly one	rdfs:Property	The predicate of the triples whose objects define the contents of the container.

Retrieving non-member properties

The representation of a container that has many members will be large. When we looked at our use cases, we saw that there were several important cases where clients needed to access only the non-member properties of the Container. [The dcterms properties listed in this page may not seem important enough to warrant addressing this problem, but we have use cases that add other predicates to containers - for providing validation information and associating SPARQL endpoints for example.] Since retrieving the whole container representation to get this information is onerous, we were motivated to define a way to retrieve only the non-member property values. We do this by defining for each Basic Profile Container a corresponding resource, called the "non-member resource", whose state is a subset of the state of the container. The non-member resource's HTTP URI can be derived in the following way.

If the HTTP URI of the container is {url}, then the HTTP URI of the related non-member resource is {url}?non-member-properties. The representation of {url}?non-member-properties is identical to the representation of {url}, except that the membership triples are missing. The subjects of the triples will still be {url} (or whatever they were in the representation of {url}), not {url}?non-member-properties. Any server that does not

support non-member-resources should return an HTTP 404-NotFound error when a non-member-resource is requested.

This approach can be thought of as being analogous to using HTTP HEAD compared to HTTP GET. HTTP HEAD is used to fetch the response headers for a resource as opposed to requesting the entire representation of a resource using HTTP GET.

Here is an example:

Request:

```
GET /container1?non-member-properties
HOST: example.org
Accept: text/turtle
```

Response:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix bp: <http://open-services.net/ns/basicProfile#>.
<http://example.org/container1>
   a rdfs:Container;
   dcterms:title "An Basic Profile Container of Acme Resources";
   bp:containerPredicate rdfs:member;
   dcterms:publisher <http://acme.com/>.
```

Design motivation and background

The concept of non-member-resources has not been especially controversial, but using the URL pattern {url}?non-member-properties to identify them has been controversial. Some people feel it's an unacceptable intrusion into the URL space that is owned and controlled by the server that defines {url}. A more practical objection is that servers respond unpredictably to URLs they do not understand, especially those that have a "?" character in them. For example, some servers will return the resource identified by the portion of the URL that precedes the “?” and simply ignore the rest. This problem could perhaps be mitigated by using a character other than "?" in the URL pattern. An alternative design that was discussed uses a header field in the response header of {url} to allow the server to control and communicate the URL of the corresponding non-member-resource - presence or absence of the header field would let clients know whether the non-member-resource is supported by the server. The advantages of this approach are that it does not impinge on the server's URL space, and it works predictably for servers that do not understand the concept of a non-member-resource. The disadvantages are that it requires two server round-trips - a HEAD and a GET - to retrieve the non-member-resources, and it requires the definition of a custom HTTP header, which to some people at least seems comparatively heavyweight.

Additional considerations

Basic Profile Containers should provide guidance on:

- When dcterms:modified and/or Etag changes when container membership changes to effectively allow for caching of containers.
- Membership limitations - typically a resource will only be part of a single container, though there may be exceptions.

Basic Profile Validation and Constraints

Basic Profile resources are RDF resources and RDF has the happy characteristic that "it can say anything about anything". This means that in principle any resource can have any property and there is no requirement that any two resources have the same set of properties even if they have the same type or types. In practice, though, the properties that are set on resources usually follow regular patterns that are dictated by the usages of those resources. While a particular resource may have arbitrary properties, when viewed from the perspective of a particular application or use-case, the set of properties and property values that are appropriate for that resource in that application will often be predictable and constrained. For example, if a server has resources that represent software products and bugs, a client may want to know what properties software products and bugs have on that server, for the purposes of displaying information in tabular formats, creating and updating resources, or other purposes. The Basic Profile Validation and Constraints specification aims to capture information about those properties and constraints.
The distinction between the resource and the use-cases it participates in is important to us. Traditional technologies like relational databases constrain the total set of properties that an entity can have. In Basic Profile we aim only to define the properties a resource may have when viewed through the lens of a particular application or use-case, while retaining the ability of the same resource to have an arbitrary set of properties to support other applications and use-cases.

The set of properties that a resource can or will have is not necessarily linked to its type, but exploiting the pattern where resources of the same type have the same properties is a very traditional approach that supports the development of many useful applications. Sometimes knowledge of types and properties for the application is hard-coded in software, but there are many cases where it is desirable to represent this knowledge in data. Basic Profile provides resource types called Shape and PropertyConstraint to represent this data.

Note on relationship of Shape to other standards: Although we're all very familiar from relational databases and object-oriented programming with the model where the valid properties are constrained by the type, it is not the "natural" model of RDF, nor is it the model of the natural world. The familiar model says that if you are of type X, you will have these properties which will have values of certain types. RDF and, to a large degree, the natural world work the other way around – if you have these properties, you must be of type X. We are not aware of any OWL or RDFS construct that lets you say "from the perspective of application X, resources whose RDF type is Y will have the list of properties Z", or of constraining the types of the values of these properties.

Class: PropertyConstraint

URI: `http://open-services.net/ns/basicProfile#PropertyConstraint`

Properties whose domain is bp:PropertyConstraint:

Property	Occurs	Range	Comment
rdfs:label	zero or one	rdfs:Literal	A human-readable name for the subject. (from rdfs)
rdfs:comment	zero or one	rdfs:Literal	A description of the subject resource. (from rdfs)
bp:constrainedProperty	exactly one	rdfs:Property	The URI of the predicate being constrained
bp:rangeShape	zero or one	bp:Shape	A bp:Shape that describes the rdfs:Class that is range of the property
bp:allowedValue	zero or many	range of the subject	A value allowed for the property. If there are both `bp:allowedValue` elements and an `bp:AllowedValue` resource, then the full-set of allowed values is the union of both.
bp:AllowedValues	zero or many	bp:AllowedValues	A resource with allowed values for the property being defined.
bp:defaultValue	zero or one	range of the object	A default value for the property
bp:occurs	exactly one	rdfs:Resource	**MUST** be either `http://open-service.net/ns/basicProfile#Exactly-one`, `http://open-service.net/ns/basicProfile#Zero-or-one`, `http://open-service.net/ns/ basicProfile#Zero-or-many` or `http://open-service.net/ns/basicProfile#One-or-many`
bp:readOnly	zero or one	Boolean	`true` if the property is read-only. If not set, or set to `false`, then the property is writable. Providers **SHOULD** declare a property read-only when changes to the value of that property will not be accepted on PUT. Consumers should note that the converse does not apply: Providers **MAY** reject a change to the value of a writable property.
bp:maxSize	zero or one	Integer	For String properties only, specifies maximum characters allowed. If not set, then there is no maximum or maximum is specified elsewhere.
bp:valueType	zero or one	rdfs:Resource	For literals, see XSD Datatypes

It is debatable whether we should have a separate bp:PropertyConstraint class with a property on it called bp:constrainedProperty, or whether it would be better to use rdfs:Property itself and simply define new predicates whose domain is rdfs:Property. However, it is important to not use rdfs:range, because the semantics are different.

Class: bp:AllowedValues

Allowed values for one property.

URI: `http://open-services.net/ns/basicProfile#AllowedValues`

Properties whose domain is bp:AllowedValues:

Property	Occurs	Range	Comment
bp:allowedValue	zero or many	same as range of owning property	Allowed value

Class: bp:Shape

URI: `http://open-services.net/ns/basicProfile#Shape`

Properties whose domain is bp:Shape:

Property	Occurs	Range	Comment
dcterms:title	zero or one	rdfs:XMLLiteral	Title
bp:describedClass	exactly one	rdfs:Class	Class described
bp:propertyConstraints	zero or one	rdfs:List	The list of propertyConstraints for properties of this Shape. The domains of the PropertyConstraints must be compatible with the describedClass.

Validation Semantics

Validation semantics is expressed by mapping the property and class definitions in terms of SPARQL ASK semantics. This enables a declarative way in RDF to define the constraints while using the existing specification SPARQL ASK.

Associating shapes and containers

It is useful to be able to specify for a container what types of members it will return and accept and what properties it expects to be used with resources of those types. To enable this, Basic Profile defines 2 new container properties:

Class: rdfs:Container

Properties whose domain is rdfs:Container

Property	Occurs	Range	Comment
bp:createShape	zero or many	bp:Shape	One or more Shapes that provide information on the expected data formats of resources that can be POSTed to the container to create new members.
bp:readShape	zero or many	bp:Shape	One or more Shapes that provide information on the expected data formats of resources that can be found as members of the container. Containers often add properties of their own to POSTed and PUT resources (creation date, modification date, creator, …) and it's useful for clients to know what these might be.

Conclusion

We believe that getting to a simple basic profile will enable broader adoption of Linked Data principles for application integration. Additional development of some of the concepts will be needed to complete such a basic profile, though the intent of this publication is to initiate the development of such specifications that will fill this much needed gap.

References

- Open Services for Lifecycle Collaboration (OSLC) http://open-services.net
- Linked Data at W3C http://www.w3.org/standards/semanticweb/data
- Dublin Core Metadata Initiative http://dublincore.org
- Resource Description Framework (RDF) http://www.w3.org/TR/rdf-concepts/

Authors

Martin P. Nally - Chief Technical Officer, IBM Rational Software
Steve Speicher - IBM STSM, OSLC Lead Architect

Acknowledgements

Thanks to Arthur Ryman, Arnaud le Hors and John Arwe (and others) for review, feedback and some content.