OSLC Primer – Learning the concepts of OSLC

It has become commonplace that specifications are precise in their details but difficult to read and understand unless you already know the basic concepts. A good solution to this problem is to write a companion document called a primer, designed to be read first, whose goal is to explain the concepts while leaving precise details to the specification. This document is the **primer** for the OSLC Core Specification Version 2.0.

It is also common that specifications lack information on how to apply the concepts of the specification, especially best practices. This document also attempts to be a **usage guide** for the OSLC Core Specification Version 2.0. Usage guidance is clearly marked with the Guidance keyword.

Intended Audience

This document is intended for technical leaders who want to understand the concepts and goals of OSLC and its relationship to other standards for evaluation, as well as potential OSLC implementers who want a general overview of the OSLC concepts and an understanding of the thinking and use-cases that led to their definition.

We assume familiarity with basic web technologies such as HTTP, RDF and Linked Data. If you're not familiar with these topics, you might find the following helpful:

- http://www.w3.org/2007/02/turtle/primer/
- http://www.w3.org/TR/rdf-primer/
- http://linkeddata.org/guides-and-tutorials

Conventions

The namespace for classes and predicates defined in the OSLC Core spec is `http://open-services.net/ns/core#`, abbreviated to oslc: in this document. We also reference concepts in the RDF namespace, `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, abbreviated to rdf:, and the RDF Schema namespace, `http://www.w3.org/2000/01/rdf-schema#`>, abbreviated to rdfs:.

Most of the data samples in this document are written in Turtle notation, for its superior readability. If you prefer your samples in RDF/XML, copy the Turtle text into the validator at this URL: `http://www.rdfabout.com/demo/validator/` , set the input type to "Notation 3 (or N-Triples/Turtle)" and click "validate!". The validator will produce equivalent RDF/XML.

Table of Contents

What is OSLC?

Open Services for Lifecycle Collaboration (OSLC) is an open community creating specifications for integrating tools. These specifications allow conforming independent software and product lifecycle tools to integrate their data and workflows in support of end-to-end lifecycle processes. Examples of lifecycle tools in software development include defect tracking tools, requirements management tools and test management tools. There are many more examples in software and product development and more still in "IT operations" (sometimes called service management) where deployed applications are managed.

The OSLC community is organized into workgroups that address integration scenarios for individual topics such a change management, test management, requirements management and configuration management. These topics that have OSLC workgroups and specifications are called "domains" in OSLC. Each workgroup explores integration scenarios for a given lifecycle topic and specifies a common vocabulary for the lifecycle artifacts needed to support the scenarios.

To ensure coherence and integration across these domains, each workgroup builds on the concepts and rules defined in the OSLC Core specification, which is produced by the Core workgroup. The OSLC Core specifies the primary integration techniques for integrating lifecycle tools. This consists mostly of standard rules and patterns for using HTTP and RDF that all the domain workgroups must adopt in their specifications. The Core Specification is not intended to be used by itself. Since there is no such thing as a generic lifecycle tool – every tool is specialized to one or more domains such as requirements, defect tracking, testing and so on – the Core Specification in conjunction with one or more of the OSLC domain specifications describe the OSLC protocols offered by a domain tool.

The goal of OSLC is to create specifications for interactions *between* tools. OSLC is not trying to standardize the behavior or capability of any tool or class of tool, like a test tool or a requirements management tool. OSLC specifies a minimum amount of protocol and a small number of resource types to allow two such tools to work together relatively seamlessly. Even within a particular resource types specified by an OSLC workgroup, the goal is to define only properties that are valuable for integration, not all the properties that might be present in a particular tool's resources. OSLC also tries to accommodate a wide variety of implementation technologies, and be equally relevant to both existing tools and to newly-built ones.

OSLC Technical Foundation

OSLC is based on the W3C Linked Data. Here is a reminder of the 4 rules of linked data, authored by Tim Berners-Lee and documented on the W3C web site. (http://www.w3.org/DesignIssues/LinkedData.html)

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. (sic)

In OSLC, each artifact in the lifecycle – for example, a requirement, defect, test case, source file, or development plan and so on – is an HTTP resource that is manipulated using the standard methods of the HTTP specification (GET, PUT, POST, DELETE).

Following the third rule of linked data, each resource has an RDF representation – OSLC mandates RDF/XML, which is the most widely adopted RDF notation - but can have representations in other formats, like JSON or HTML.

The OSLC Core specification defines a number of simple usage patterns of HTTP and RDF and a small number of resource types that help tools integrate and make the lifecycle work. The OSLC domain workgroups specify additional resource types specific to their lifecycle domain, but do not add new protocol.

Primary OSLC integration techniques

OSLC offers two primary techniques for integrating tools – "Linking data via HTTP" and "Linking Data via HTML User Interface". Both of these techniques build on the HTTP and RDF foundation of OSLC.

Linking data via HTTP

OSLC specifies a common tool protocol for creating, retrieving, updating and deleting (CRUD) lifecycle data based on internet standards like HTTP and RDF using the Linked Data model. This protocol can be used by any tool or other programmatic client to talk to any other tool that implements the specifications. Linking is achieved by embedding the HTTP URL of one resource in the representation of another.

Linking Data via HTML User Interface

OSLC specifies a protocol that allows a tool or other client to cause a fragment of the web user interface of another tool to be displayed, allowing a human user to link to a new or existing resource in the other tool or see a preview of information about a resource in another tool. This enables a tool or other client to exploit existing user interface and business logic in other tools when integrating information and process steps. In some circumstances this is more efficient and offers more user function than implementing a new user interface and then integrating via an HTTP CRUD protocol.

Both of these techniques are described in this primer.

ServiceProvider – Starting point for understanding the OSLC Core

Almost all existing lifecycle tools - whether they manage defects, test cases, requirements or whatever - have organizing concepts that partition the overall space of artifacts in the tool into smaller containers. Examples of common partitioning concepts offered by tools include “projects”, “modules”, “user databases” and so on. Each artifact created in the tool is created within one of these container-like entities, and users can list the existing artifacts within one. These container-like concepts are very important to the usage of tools – which container you put artifacts into and find artifacts in is essential to the way you work and may reflect which project you are working on, or which product the artifacts pertain to. There is no agreement across tools of the lifecycle, even within a particular domain, on what these partitioning concepts should be called, but there is almost universal agreement that they exist and are fundamentally important. These concepts are also fundamentally important in the integration scenarios supported by OSLC. OSLC defines the concept of ServiceProvider to allow products to expose these containers or partitions for integration scenarios. ServiceProviders answer two basic questions, which are:

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

Tip! A common misconception of people who have read the OSLC specification is that a ServiceProvider is intended to represent a tool or tool instance. A ServiceProvider is intended to represent a “container” of resources that is hosted by a tool, not the tool itself. A single instance of a tool will typically host multiple ServiceProviders, for example one for each “project” or “product”.

Tip! OSLC references another standard – OAuth – for security. OAuth also has a concept called ServiceProvider, although OAuth appears to moving away from this term in favor of “Server”. The OAuth concept of ServiceProvider corresponds to the concept of a server, not a “resource container”. Multiple OSLC ServiceProviders would be expected to be hosted by the same OAuth ServiceProvider. OSLC has a different concept – ServiceProviderCatalog (explained later) - that is more closely analogous to OAuth’s ServiceProvider concept. Be careful not to confuse the OAuth concept of ServiceProvider and the OSLC concept of ServiceProvider.

ServiceProvider is the central organizing concept of OSLC, enabling tools to expose resources and allowing consumers to navigate to all of the resources, and create new ones. Here are some characteristics of ServiceProviders:

1) All OSLC resources live in some ServiceProvider. There is an optional property of each OSLC resource (oslc:serviceProvider) that says which ServiceProvider it is “in”.
2) Clients can retrieve the list of existing resources in a ServiceProvider

3) The only way that is defined in OSLC to create any new OSLC resources is to create them in a ServiceProvider (either directly through an HTTP POST, or via a dialog).
4) A ServiceProvider is itself an OSLC resource with an HTTP URL

Tools often have "softer" or more dynamic partitions of artifacts based on data-values like "category" or "release" or "component" that can be further used to partition the space of artifacts in a container. When adopting OSLC for a new or existing tool, deciding which container-like concept in the tool to map to an OSLC ServiceProvider is a bit of an art – you need to think about what partitions should be viewed as fundamental from the point of view of an external client, and which partitions are more dynamic and ephemeral, changing as property values of resources change. The fundamental ones are the ones that should be represented as OSLC ServiceProviders.

What does a ServiceProvider resource actually look like?

If you do an HTTP GET on a ServiceProvider resource, you will not retrieve a list of the resources it contains; you will retrieve general properties of the ServiceProvider – its "metadata" if you like that term – including the URLs you can use to find or create resources. If you do a POST to the URL of a ServiceProvider, you will likely just get an HTTP Error. Two fundamental properties of a ServiceProvider are:

1) oslc:creation: the URL of a resource to which you can POST representations to create new resources.
2) oslc:queryBase: the URL of a resource that you can GET to obtain a list of existing resources in the ServiceProvider. This URL is called the "queryBase URL" and the resource identified by this URL is called the queryBase.

In the simplest case, the creation URI and the queryBase URI will in fact be the same URL.

ServiceProviders have a third important property – dialog – that is the foundation of the second major OSLC integration technique based on invocation of HTML web user interface dialogs of one tool by another. Dialogs are discussed in a separate section of this primer.

You might think from the preceding description that the simplest ServiceProvider example might look something like the following, in Turtle notation.

```
@prefix oslc: <http://open-service.net/ns/core#>.
<http://acme.com/toolA/container1>
   a oslc:ServiceProvider;
   oslc:creation <http://acme.com/toolA/container1/contents>;
   oslc:queryBase <http://acme.com/toolA/container1/contents>.
```

The example above is true to the spirit of OSLC, and captures the essential meaning of ServiceProvider accurately, which is why it is listed here, but the real OSLC syntax is more complex, and this example is not legal.

OSLC Core supports more complex options for ServiceProviders, including the ability to have more than one creation URI and more than one queryBase URI, the ability to attach properties to each creation URI and each queryBase URI and to give hints about their intended usage and the ability to group creation URIs and queryBase URIs by the OSLC domain they are intended to support.

In order to satisfy these requirements, OSLC introduces three additional concepts – Service, CreationFactory and QueryCapability. These are not primary concepts in OSLC and their instances are never independent HTTP resources with their own URLs. These three concepts are more like "structured data types" whose instances are used as property values in the state of a ServiceProvider - you should think of Service, CreationFactory and QueryCapability as technical details of the way the state of a ServiceProvider is organized rather than central OSLC concepts. QueryCapability allows properties to be associated with a queryBase URI, CreationFactory allows properties to be associated with a creation URI and Service allows (actually, requires) creation URIs and queryBase URIs to be grouped by OSLC domain.

Because of these additional concepts, our simplest ServiceProvider actually looks like this:

Example 1:

```
@prefix oslc: <http://open-service.net/ns/core#>.
<http://acme.com/toolA/container1> a oslc:ServiceProvider;
   oslc:service
      [a oslc:Service;
         oslc:domain <http://open-services.net/ns/cm#>;
         oslc:creationFactory
            [a oslc:CreationFactory;
               oslc:creation <http://acme.com/toolA/container1/contents>];
         oslc:queryCapability
            [a oslc:QueryCapability;
               oslc:queryBase <http://acme.com/toolA/container1/contents>]
      ].
```

Guidance – Define simple ServiceProviders

We recommend that implementers of OSLC adopt the following simplified pattern of usage unless they have compelling reasons to do otherwise:

1. Focus first on identifying the basic containers implemented by the tool and expose each of these containers as a single ServiceProvider. Don't attempt to create individual ServiceProviders for each type of resource. For example, if the data partitioning concept in your tool is project, create a single ServiceProvider for each project, even if the project can contain Defects, Risks, Issues, Comments, and Approvals.
2. Identify a single URL for each ServiceProvider that will be used both as the URL to which new resources can be POSTed (the creation URI) and the URL of the RDF

container resource that lists the existing resources of the container (the queryBase URI).

3. Within each ServiceProvider's representation, create a single Service with a single CreationFactory and a single QueryCapability each referencing the single URL established in step 2. Example 1 above shows exactly this pattern.

What does a queryBase resource look like?

The queryBase resource identified by a queryBase URI of a ServiceProvider is an RDF Container resource that lists all the resources in the container.

If a ServiceProvider has a single queryBase, resources that have been POSTed to a ServiceProvider's creation URI will appear in the ServiceProvider's queryBase resource if they have not been subsequently deleted. A queryBase may also contain resources that were added through other means - for example through the user interface of the tool that implements the ServiceProvider.

If a ServiceProvider has multiple queryBases, it is undefined which queryBases a resource will show up in after being POSTed to a creation URI.

The representation of the queryBase is a standard RDF Container representation using the rdfs:member predicate. (http://www.w3.org/TR/rdf-schema/#ch_member) For example, if I have an OSLC container with the URL http://acme.com/oslc/container/1, it might have the following representation:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
<http://acme.com/oslc/container/1>
   <rdfs:member> <http://acme.com/oslc/resource/000000000>;
   # … 999999998 more triples here …
   <rdfs:member> <http://acme.com/oslc/resource/999999999>.
```

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

OSLC Resources - what do the resources inside a ServiceProvider look like?

Any HTTP resource with any representation might be found inside a ServiceProvider. In many of the most interesting cases, the resources will be OSLC resources. An OSLC resource is simply a resource whose type is defined in some OSLC specification, usually one of the domain specifications created by one of the OSLC domain workgroups. All OSLC resources share some common characteristics

1) You can request an RDF/XML representation of the resource. All OSLC resources have their state[1] defined by a set of RDF properties that may be required or optional. This set of properties was designed to support OSLC integration scenarios – it is expected that any particular resource may have many more properties that are not defined in an OSLC specification. The ability of RDF to represent this kind of open extensible information model is one of the reasons it was chosen for OSLC.

2) Although an OSLC resource's state must be expressible in RDF, and it must have an RDF/XML representation, the resource may have other representations as well. You have already seen examples of Turtle - HTML and JSON[2] would be popular additions, and OSLC sets no limits.

3) OSLC protocols use standard media types. OSLC does not require and does not encourage the definition of any new media types. The goal is that any standards-based RDF or Linked Data client be able to read and write OSLC data, and defining new media types would prevent that in most cases.

4) OSLC resources use common property names for common concepts. In the current state of the art in lifecycle tools, each tool defines its own properties for common concepts like label, description, creator, last-modification-time, priority, and so on. In many cases an administrator can define these properties locally for an installation, so the tool vendors may not control the vocabulary. This is usually viewed as a good feature by practitioners who want their tools to match their local terminology and processes, but it makes it much harder for organizations to subsequently integrate tools in an end-to-end lifecycle. OSLC resolves this by requiring all tools to expose these common concepts using a common vocabulary for properties, which you will remember are identified by URIs in RDF. Tools may choose to additionally expose the same values under their own private property names in the same resources. In general, OSLC avoids inventing its own property names where possible – it uses ones from popular RDF-based standards like the RDF standards themselves, Dublin Core,

[1] The word "state" is used here to mean all of the data values for the resource, not a particular "state" property. This meaning is consistent with the meaning of state in the REST (Representational State Transfer) description of the architecture of the world-wide web.

[2] W3C has not yet standardized a JSON format for RDF data although it is expected to do so in the future. For the moment, the OSLC Core specification defines its own JSON encoding of RDF state.

and so on. In some cases, OSLC has invented property URLs where no match was found in popular standard vocabularies.

RDF classes and properties in OSLC

OSLC resource types can be used as the value of the rdf:type predicate in resources. Following the standard rules of RDF, this means that OSLC resource types are RDFS classes (by definition of rdf:type in the RDF spec).

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 and OSLC 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 some cases.

OSLC Datatypes

OSLC uses RDF concepts to define a number of standard properties. OSLC properties are RDF properties (by definition, because they are used as predicates). RDF does not by itself define datatypes to be used for property values, so OSLC lists a set of standard datatypes to be used in OSLC. The list is **Boolean, DateTime, Decimal, Double, Float, Integer, String** and **XMLLiteral.** Of course, OSLC also uses URI references and blank nodes as described in the RDF documentation.

The intention of some OSLC properties – like oslc:service, oslc:creationFactory and oslc:queryCapability – is that they only be used to reference blank nodes, never to reference true HTTP resources. To be able to express this intent, the OSLC Core specifies a number of OSLC-specific value types that are used in the specification and can also be found in ResourceShape (see later) property descriptions. These are the "resource value types", **Resource, Local Resource**, and **AnyResource. Resource** means that the value will be an RDF URI reference. **Local Resource** means the value will be an RDF blank node identifier and **AnyResource** means it can be either. The OSLC specification allows you to further specify whether the referenced resource must be defined inside the same representation that contains the reference, in a separate document, or either, using the following values for the representation property of the property: `http://open-service.net/ns/core#Reference` , `http://open-service.net/ns/core#Inline` or `http://open-service.net/ns/core#Either`

Guidance on usage of the *representation* property

There is ongoing discussion in the OSLC Core workgroup on the proper meaning and usage of the representation property. Defensive OSLC clients should always assume that a URI reference found in the representation of one resource may identify another resource on the web with its own independent representation or may – via a fragment identifier in the URL - identify a resource whose representation is part of the current representation. Similarly defensive clients should assume that an identifier may be either a URI reference or a blank node identifier - this is the standard RDF rule. Before

dereferencing a URIRef that is found as the value of a property in an RDF graph, clients should look to see whether the information they are looking for about the resource identified by that URIRef is already included in the current graph in the form of triples with that URIRef as the subject. Also before dereferencing a URIRef, clients should verify that it does not circularly identify the current graph, perhaps with the addition only of a fragment identifier.

Unknown properties and content

As described before, OSLC depends on an open model for resource state. The specification defines some standard properties for integration, but also assumes that any given resource may have many more properties than are defined in the specification. Some tools will only support a fixed set of properties for a particular type of resource and may provide a ResourceShape that lists that set of properties. Clients should still assume that the set of properties for a type in an arbitrary tool may be open. This can happen several different ways. One is that the tool specifically supports an extensible set of properties in the sense that different resources of the same type may not all have the same properties. Another possibility is that the list of properties is fixed at any one time, but that older or newer resources may have different properties.

For OSLC Defined Resources, clients should assume that an OSLC server implementation may discard triples for properties of which it does have prior knowledge - an OSLC implementation may discard property values that are not part of the resource definition or Resource Shape known by the server.

The rule is different for clients. When doing an update, OSLC clients must preserve any unknown property-values and other content in OSLC Defined Resources.

Open Model

Many specifications have a “closed model”, by which we mean that any reference from a resource in the specification will necessarily identify a resource in the same specification or a referenced specification. UML is an example of a closed specification – every UML reference is to another UML object. By contrast, the HTML anchor tag can point to any HTTP resource, not just other HTML resources. OSLC works more like HTML in this sense. Here are some examples:

1. Any HTTP resource can be contained in a ServiceProvider, not just resources defined in OSLC specifications
2. A URL reference in one OSLC resource may in general point to any HTTP resource, not just an OSLC resource. OSLC specifications will not usually constrain the value for a property in one OSLC specification to be the URL of a resource in a different OSLC specification[3], although it is common for a property

[3] The exception is the Core specification, which is considered to be part of each of the domain specifications

defined in one OSLC specification to constrain its value to being the URI reference of a resource in the same specification.

OSLC specifications generally avoid constraining a reference to be a resource in another specification. This independence allows OSLC specifications to evolve independently, and new ones to be added, without changing existing ones, and also allows tools at different version levels to interoperate.

Tip! - **A consequence of this independence is that tool implementations that traverse URL links from an OSLC resource in one OSLC specification that may reference something outside the same specification should always code defensively and be prepared for any HTTP resource at the end of the link.** Defensive coding by OSLC tools is necessary to allow sets of tools that communicate via OSLC protocols to be independently upgraded.

Optimistic Collision Detection on Update

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, OSLC implementations should use the HTTP `If-Match` header and etags to detect collisions.

Resource Paging

It sometimes happens that a resource is too large to reasonably transmit in a single HTTP message. A client may anticipate that a resource will be too large - for example, a client tool that accesses defects may assume that an individual defect will usually be of sufficiently constrained size that it makes sense to request all of it at once, but that the list of all the defects ever created will typically be too big[4]. Alternatively, a server may recognize that a resource that has been requested is too big to return in a single message.

To address this problem, OSLC resources may support a technique called Resource Paging that enables clients to retrieve representations of resources one page at a time. For every resource whose URL is `<url>`, an OSLC implementation may define a companion resource whose URL is `<url>?oslc.paging=true`. The meaning of this resource is "the first page of `<url>`". Clients that anticipate that a particular resource will be too large may instead fetch this alternate resource. Servers that determine that a requested resource

[4] A client may also use the HTTP HEAD method to determine the size of a resource without fetching it.

is too large may respond with a 302 redirect message, directing the client to the "firstPage" resource[5].

The representation of `<url>?oslc.paging=true` will contain a subset of the triples that define the state of the resource whose URL is `<url>`. The subject of those triples will be `<url>`, not `<url>?oslc.paging=true`. In addition, the representation of `<url>?oslc.paging=true` may include a few triples whose subject is `<url>?oslc.paging=true` itself. Examples are triples whose predicate is oslc:nextPage, dcterms:description and so on.

Note that pagination is only defined for resources whose state can be expressed in RDF as a set of RDF triples. Pagination is undefined for resources whose state cannot be represented in RDF. Pure binary resources, encrypted resources, or digitally signed resources might be examples. The representation of a Page is defined by first paginating the underlying triples that express the state of the resource being paginated, and then performing whatever standard mapping is used to map from each page of triples to the requested representation. In other words, we do not paginate the representations; we paginate the RDF resource state itself and then create the representations of each page in whatever media type is requested. This provides a general specification for both RDF and non-RDF representations of pages of RDF resources. Examples of non-RDF representations are HTML and JSON.

For example, if I have an OSLC container with the URL http://acme.com/oslc/container/1, it might have the following representation (in Turtle notation):

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
<http://acme.com/oslc/container/1>
   <rdfs:member> <http://acme.com/oslc/resource/000000000>.
   # … 999999998 more triples here …
   <rdfs:member> <http://acme.com/oslc/resource/999999999>.
```

This representation has a billion triples and over 90 billion characters, which might be a bit big. Assuming that the implementation that backs this resource supports paging, a client can chose instead to GET the related resource http://acme.com/oslc/container/1?oslc.paging=true. The representation of this latter resource would look like this:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
<http://acme.com/oslc/container/1>
<rdfs:member> <http://acme.com/oslc/resource/000000000>.
# … 98 more triples here …
<rdfs:member> <http://acme.com/oslc/resource/000000099>.
# pay attention to the subject URL of the following triple
<http://acme.com/oslc/container/1?oslc:paging=true> <oslc:nextPage>
<http://acme.com/oslc/xxxxxxxxx/page2>.
```

[5] If the resource requested by the client was already a page, a server should not use this technique to force further pagination within a page.

As you can see, the representation of this smaller "firstPage" resource contains the first 100 triples that you would have got in the representation of the large resource in exactly the same form - the same subject, predicate and object - as in the representation of the large resource. In addition, it contains another triple - whose subject is the "firstPage" resource itself, not the bigger resource - that provides the URL of a third resource that will contain the next page of triples from the bigger resource. The format of the URLs of the second and subsequent pages (if they exist) is not defined by the OSLC specification – an OSLC implementation can use whichever URL it pleases. Note that although this example shows the triples in a precise order for purposes of simplicity and clarity of the example, there is no concept of ordering of triples in RDF, so the triples can be in any order both within and across pages.

As illustrated above, when a page is returned it will include the triple:

```
<url of current page> oslc:nextPage <url of next page>.
```

You can tell when you are on the last page by the absence of an oslc:nextPage triple.

Because paging is unstable (see below), by the time a client follows an `oslc:nextPage` link there may no longer be a next page. The OSLC server implementation in this case may respond with an HTTP 404 error.

The OSLC specification permits `<url>?oslc.pageSize=n` as an alias for `<url>?oslc.paging=true`. Because it is just an alias, it has exactly the same meaning and behavior. An OSLC server implementation may (but is not obliged to) adjust the number of triples on the first and subsequent pages based on the value of n.

When Resource Paging is used, the values of a multi-valued property of a single resource may be split across resource pages. All triples that reference the same blank node, must all be contained on the same page, since a blank node cannot be referenced from a different page (this is simply an observation on how RDF works, not an OSLC policy or limitation).

Unstable Paging

Because HTTP is a stateless protocol and OSLC Services manage resources that can change frequently, OSLC clients should assume that resources can change as they page through them using the `oslc:nextPage` mechanism. Nevertheless, each triple of the resource that exists when the first page is returned and is not subsequently deleted during the paging interaction must be included on at least one page. [Including the same triple more than once is permissible – identical triples are always discarded in RDF - but servers need to ensure that the same triple is not returned multiple times with different object values.] Triples that are added after the first page is returned may or may not be included in subsequent pages by a server.

OSLC ResourceShapes

In general, the validation rules that a particular container may apply in deciding to whether or not to accept a POST of a new resource representation can have unlimited complexity, and OSLC makes no attempt to describe them. Similarly the format of the representation of existing resources can be complex and variable, and the validation of a representation used to update an existing resource via PUT can also be complex. Despite the possibility of unlimited complexity, OSLC recognizes that there is a common pattern that is adopted by many, though not all tools, as described here:

1) Define a fixed set of types of resources. These are RDF Classes that can be referenced within the representation of a resource using the rdf:type predicate.
2) For each class, define a fixed list of properties whose values must or may be set when creating or updating a resource of that type. Expected datatypes and values for these properties is also useful information.
3) Similarly, for each class, define a fixed list of properties whose values may be encountered when reading a resource of that type. It is common for a "read" resource to have more properties than a "written" resource – servers often add properties like last-modification-date, creator and so on to those provided by a client on creation or update, and it's nice to know what those extra properties are because they can be used in queries.

OSLC defines the OSLC ResourceShape resource to allow the specification of a list of properties with allowed values and the association of that list with an RDFS Class. ServiceProviders that implement the simple standard model of "resources of a particular type have a fixed set of properties" can use this feature – ServiceProviders that don't can ignore it.

Note on relationship of ResourceShape to other standards.

> Although we're all very familiar with this model from relational databases and object-oriented programming, it is not the "natural" model of RDF, nor is it the model of the natural world. This model says that if you are of type X, you must have these properties. RDF and the natural world work the other way around – if you have these properties, you must be of type X. This is why there is no standard RDF vocabulary that can express the equivalent of ResourceShapes.
>
> OWL is a separate RDF-based standard which, at first glance, appears to solve the same problem as ResourceShapes. However, OWL, like RDF Schema, infers new triples from a given set of triples. OWL lets you specify inference rules about properties and types, including rules of the form "if you have these properties, you must be of type X". In contrast, ResourceShapes lets you specify the constraints that a given set of triples must have. In relational database terms, ResourceShapes are similar to TABLE definitions and integrity constraints while OWL rules are similar to VIEW definitions.

To help you understand how ResourceShapes work, below is an example shape for a fictional "Tool A" which exists at the URL `http://acme.com/toolA/resourceShape1`. You can see, below, the shape describes a Defect resource (http://acme.com/toolA/Defect) and three properties. The first is the standard Dublin Core "`title`" predicate, which is optional. The second is a Tool A-specific property called "`priority`" and the third is another Tool A-specific property called "`defectiveComponent`." The priority property definition also defines three allowed values for setting severities of high, medium and low.

```
@prefix oslc: <http://open-service.net/ns/core#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix dcterms: <http://purl.org/dc/terms/>.
<http://acme.com/toolA/resourceShape1>
     a oslc:ResourceShape;
     dcterms:title "Shape of resources of type
Defect";
     oslc:describes: <http://acme.com/toolA/Defect>;
     oslc:property [
          a oslc:Property;
          dcterms:title "details for dcterms:title
property";
          oslc:propertyDefinition dcterms:title;
          oslc:name "title";
          oslc:occurs oslc:Zero-or-many;
          oslc:valueType xsd:String
          ];
     oslc:property [
          a oslc:Property;
          dcterms:title "details for priority
property";
          oslc:propertyDefinition
<http://acme.com/toolA/priority>;
          oslc:name "priority";
          oslc:occurs oslc:Exactly-one;
          oslc:valueType oslc:Resource;
          oslc:allowedValue
<http://acme.com/toolA/high>;
          oslc:allowedValue
<http://acme.com/toolA/medium>;
          oslc:allowedValue
<http://acme.com/toolA/low>
          ];
     oslc:property [
          a oslc:Property;
          dcterms:title "details for component
property";
```

```
          oslc:propertyDefinition
<http://acme.com/toolA/defectiveComponent>;
          oslc:name "defectiveComponent";
          oslc:occurs oslc:Zero-or-many;
          oslc:valueType oslc:Resource;
          oslc:range
<http://acme.com/toolA/Component>].
```

And below is an example Defect resource representation that provides the three properties specified by the shape.

```
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix toolA:   <http://acme.com/toolA/>.
toolA:defect1
     a toolA:Defect;
     dcterms:title "Server failure during startup";
     toolA:priority toolA:high;
     toolA:defectiveComponent
      <http://acme.com/toolA/comp/Server>.
```

.

Guidance – implement simple validations for create and update

OSLC implementations should try to make it easy for programmatic clients to create and update resources. If OSLC 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 tool that needs to be communicated outside of the OSLC specifications. The preferred approach for tools is to allow creation and update based on the sort of simple validations that can be communicated programmatically through a ResourceShape. Additional checks that are 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 tools or tool installations 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 tools or tool installations should be aware that as a consequence they may be making it difficult or impossible for external software to use their OSLC protocols without extensive customization.

OSLC Query mechanisms

Finding information by performing a GET on a web of OSLC resources or on a ServiceProvider is a powerful, general mechanism, but sometimes it is inconvenient or inefficient to find specific information this way. To help with this problem, OSLC provides 2 distinct query mechanisms to find information in resources faster.

Starting from an OSLC Resource URL

OSLC implementations may support a technique called Selective Properties to enable clients to retrieve only selected property values of a specific resource. For each resource whose URL is <url>, an OSLC implementation may define a set of related resources whose URLs are of the form <url>?oslc:properties=<property list>. Each of these resources is a related resource whose representation contains the specified subset of triples of the resource whose URL is <url> and/or related resources.

Here's how the selective properties values `oslc.properties` and `oslc.prefix` work.

oslc.properties

The `oslc.properties` key=value pair portion of the URL lets you specify the resource with the set of properties you wish to retrieve. Both immediate and nested properties may be specified. A nested property is a property that belongs to the resource referenced by a property value in another resource. Nested properties are enclosed in brace brackets, and this nesting may be done recursively, i.e. a nested property may contain other nested properties.

For example, suppose we have a bug report resource at the following URL:

```
http://example.com/proj1/4242
```

Suppose this bug resource has properties such as `dcterms:title`, `dcterms:description`, and `dcterms:creator`, and that `dcterms:creator` refers to a `foaf:Person` resource that has properties such as `foaf:givenName` and `foaf:familyName`. Suppose you want to retrieve a resource that includes the `dcterms:title` of the bug report and the `foaf:givenName` and `foaf:familyName` of the person referred to by the bug report's `dcterms:creator`. The following URL illustrates the use of `oslc.properties` in the query string of a URL to include those properties:

```
http://example.com/proj1/4242?oslc.properties=dcterms:title,dcterms:cre
ator{foaf:givenName,foaf:familyName}
```

This URL identifies a resource whose representation contains exactly the desired information. The representation would look like this:

```
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix foaf: <http://http://xmlns.com/foaf/0.1/>.
<http://example.com/proj1/4242> dcterms:title "Bug 4242".
<http://example.com/proj1/4242> dcterms:creator <http://example.com/users/1>.
<http://example.com/users/1> foaf:givenName "Dave".
<http://example.com/users/1> foaf:familyName "Johnson".
```

oslc.prefix

Languages such as Turtle and SPARQL let you define URI prefixes (e.g. dcterms:) so you can represent URIs more compactly (e.g. dcterms:title). The use of compact URIs is also convenient when writing query strings. OSLC domain specifications predefine some prefixes for this purpose. For example, the oslc.properties query string assumes that dcterms: and foaf: have been predefined. However, OSLC domain implementations may add new properties and so it is not possible to predefine all prefixes. To handle this situation, you can define additional prefixes in the query string using oslc.prefix.

Suppose that foaf: was not predefined in the above example. You add its definition to the query string as follows:

http://example.com/proj1/4242?oslc.prefix=foaf=<http://xmlns.com/foaf/0.1/>&oslc.properties=dcterms:title,dcterms:creator{foaf:givenName,foaf:familyName}.

Starting from a ServiceProvider

The section entitled “Starting from an OSLC Resource URL” describes how to efficiently extract information from a graph of resources if you have the URL of a specific resource to start from. There is also an important case where the starting point is a ServiceProvider.

In addition to providing queryBases for a ServiceProvider, an OSLC implementation may provide additional resources that allow clients to find selective subsets of the resources and resource properties in queryBases. The URLs of these resources are constructed by adding a query component to the URL of the queryBase rather than to the URL of an OSLC resource. This can be much more convenient and efficient than performing a GET on the whole container, or paging through the container.

Query Syntax

The OSLC Core Spec Query Specification document defines a standard set of OSLC query parameters.

Query example

In the following examples, assume a ServiceProvider that contains bug reports (and potentially other resources) for a particular project. Assume also that those bug reports point to user accounts. The user accounts could be in the same ServiceProvider as the bug reports, but since the same accounts may be used to identify users in multiple projects, it’s more likely they are held somewhere else. The URL of the queryBase for the

ServiceProvider is `http://example.com/proj1` and the queryBase resource looks like this:

```
# the query base for proj1
<http://example.com/proj1>
   rdfs:member
      <http://example.com/proj1/4242> ,
      <http://example.com/proj1/4243> ,
      <http://example.com/proj1/4244> ,
      <http://example.com/proj1/4245> ,
      <http://example.com/proj1/4246> ,
      <http://example.com/proj1/4247> ,
      <http://example.com/proj1/4248> .
```

The actual bug report resources look like the following:

```
# representation of http://example.com/proj1/4242
<http://example.com/proj1/4242>
   dcterms:title "Bug 4242" ;
   dcterms:creator <http://example.com/users/1> .

# representation of http://example.com/proj1/4243
<http://example.com/proj1/4243>
   dcterms:title "Bug 4243" ;
   dcterms:creator <http://example.com/users/2> .

# representation of http://example.com/proj1/4244
<http://example.com/proj1/4244>
   dcterms:title "Bug 4244" ;
   dcterms:creator <http://example.com/users/3> .

# representation of http://example.com/proj1/4245
<http://example.com/proj1/4245>
   dcterms:title "Bug 4245" ;
   dcterms:creator <http://example.com/users/1> .

# representation of http://example.com/proj1/4246
<http://example.com/proj1/4246>
   dcterms:title "Bug 4246" ;
   dcterms:creator <http://example.com/users/2> .

# representation of http://example.com/proj1/4247
<http://example.com/proj1/4247>
   dcterms:title "Bug 4247" ;
   dcterms:creator <http://example.com/users/3> .

# representation of http://example.com/proj1/4248
<http://example.com/proj1/4248>
   dcterms:title "Bug 4248" ;
   dcterms:creator <http://example.com/users/1> .
```

Suppose now that we use the query syntax to compose the following URL.

```
http://example.com/proj1?oslc.where=dcterms:creator=<http:example.com/u
sers/1>
```

This URL identifies a resource that contains a subset of the queryBase information, as shown here:

```
   @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.

   <http://example.com/proj1>
      rdfs:member
         <http://example.com/proj1/4242> ,
         <http://example.com/proj1/4245> ,
         <http://example.com/proj1/4248> .
```

The example above selected a subset of bug reports based on a property of the bug reports themselves. It is possible to further restrict the list by selecting based on a property of the user resources that the bug reports reference, as shown below. The URL of the query resource is this:

```
http://example.com/proj1?oslc.where=dcterms:creator{foaf:givenName="Mar
tin" and foaf:familyName="Nally"}
```

And its representation is this:

```
   @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

   <http://example.com/proj1>
      rdfs:member
            <http://example.com/proj1/4243> ,
            <http://example.com/proj1/4246> .
```

For the purposes of this example, assume the user accounts look like this:

```
# user 1
<http://example.com/users/1> a foaf:Person ,
      foaf:givenName "Dave" ;
      foaf:familyName "Johnston" .

# user 2
<http://example.com/users/2> a foaf:Person ,
      foaf:givenName "Martin" ;
      foaf:familyName "Nally" .

# user 3
<http://example.com/users/3> a foaf:Person ,
      foaf:givenName "Arthur" ;
      foaf:familyName "Ryman" .
```

It is also possible to use query to retrieve property values for the bugs and users, not just a list of bugs, as shown in the following example. The URL of the query resource is

```
http://example.com/proj1?oslc.where=dcterms:creator=<http:example.com/u
sers/3>&oslc.select=dcterms:title
```

And the resulting representation is this:

```
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

<http://example.com/proj1>
   rdfs:member
         <http://example.com/proj1/4244> ,
         <http://example.com/proj1/4247> .

<http://example.com/proj1/4244>
   dcterms:title "Bug 4244" .

<http://example.com/proj1/4247>

   dcterms:title "Bug 4247" .
```

Additional Query Capabilities

In addition to the capabilities illustrated in these examples, Query includes syntax for inequality operators (!=, <=, >=, < ,>,), the ability to test that a value is an element within a set of values, ordering of results, and full-text searches. See the specification for details.

Guidance on Query

OSLC implementers have a few options when it comes to query. They can implement any of the following:

1) OSLC Query

   The OSLC query capabilities are designed to be a compromise. They are powerful enough to be useful to clients, but simple enough to be implemented with reasonable effort by tools. It is not a trivial effort to implement the OSLC Query capabilities, but it's not an insurmountably difficult problem either. Common approaches to implementing the OSLC query capability are to transform an OSLC query into an existing query API implemented by the tool, or to transform an OSLC query into some query language that is understood by the underlying implementation technologies used by the tool – often SQL on an RDBMS. This mapping will have to take account of the transformation of the OSLC resource model into the target data model as well as the transformation of the OSLC query syntax into the underlying query syntax. The difficulty of this transformation may depend as much on the former as the latter.

2) SQL

   You might think that since many existing tools are built on RDBMS technologies, it would be trivially simple to offer SQL as a query language. In practice this is

usually much harder than it sounds since the internal data model of the tool is often far removed from the simple resource model of OSLC. This means that a transformation between external SQL and internal SQL is needed, and since SQL is a fully-featured query language, it is usually harder to implement this transformation than to implement OSLC Query.

3) SPARQL

   Another option is to implement SPARQL. Since SPARQL is a sophisticated query language, it is probably not reasonable to expect that tools will implement a SPARQL query capability by transforming it to a different underlying data model and query language supported by the tool's implementation data management layer, as might be practical for implementing OSLC Query. However, there are a number of freely-available open-source RDF triple-store implementations that offer complete SPARQL query implementations, so another implementation strategy is to make a copy of the most recent version of every resource and put it in an RDF triple-store, keep it up to date in real time, and then offer OSLC users SPARQL query using the SPARQL capabilities of the RDF triple store.

   Of course, if the state of the resources in the tool fits well with the relational data model, implementers could use the same technique with RDBMS technologies to expose SQL as a query language.

4) Proprietary

   Implementing a proprietary query language is always an option, but it has a much lower client value, since a client is faced with the need of "learning" the proprietary query language of each tool.

ServiceProviderCatalog

Most tools will expose more than one ServiceProvider, and there are different reasons why it might sometimes be useful for tools to expose defined sets of ServiceProviders. OSLC provides the concept of ServiceProviderCatalog for defining such sets. A ServiceProviderCatalog also has an optional property whose value is an OAuthConfiguration. If this property is set, clients can assume that all the ServiceProviders in the list can be accessed with the same OAuth tokens.

Guidance for ServiceProviderCatalog usage

OSLC provides the concept of a ServiceProviderCatalog, but does not specify how it is used, or what the meaning is of the set of ServiceProviders defined by a ServiceProviderCatalog. Clients should refer to the documentation specific to a particular tool for guidance on how to find and use any ServiceProviderCatalogs it exposes.

Delegated User Interface Dialogs

As we have seen, OSLC specifies simple HTTP protocols for locating, creating, updating, reading and deleting lifecycle resources. In many cases this is very useful, but for some integration scenarios, exploiting these protocols is not the best strategy. Suppose I'm the implementer of a test management tool that has a graphical user interface and I want to allow my users to easily create defects in a defect tracking tool when a test fails, or associate the failed test case with an existing defect if one exists. I could use the OSLC protocols already described to implement this integration, but if I did that I would have to implement the user interface needed for my users to enter all the fields of a new defect, or display lists of existing ones. In addition to being a lot of work, this could result in a poor user experience, because I cannot possibly understand all the detailed validations on new defects that a particular defect tool will demand, so I cannot help my users fill in all the appropriate fields with valid values. Fortunately, OSLC offers an additional style of integration that solves these problems. This style is based on the concept of a dialog. Continuing the example above, the idea of a dialog is that instead of the test management tool implementing the UI for creating or selecting defects, it asks the defect tracking tool to display to the user a "dialog" from its own user interface for the purpose. In the case of a dialog to create a new defect, the test tools can provide initial data from the test case to the dialog to "seed" the new defect, and the test tool can also get back the URL of the defect that finally gets created. In the case of a selection dialog, the test management tool gets back the URL of the selected defect, which it can then reference from the test case.

These are the two primary cases supported by Dialogs:

- **Resource creation**: when a user of a web application needs to create a new resource in an OSLC Service Provider. In this case the web application asks the service provider to provide a UI for resource creation and the provider notifies the application when the creation has been completed or canceled by the user.
- **Resource selection**: when a user of a web application and needs to pick a resource managed by an OSLC Service Provider. In this case the web application asks the service provider to provide a UI for resource selection and the provider notifies the application when a resource or resources has been selected or if the selection was canceled.

UI Preview

The information in an OSLC resource is composed of RDF triples, defining the values of properties. Triples whose object is the URI of another resource are often called links. When presenting a link to a human user, it is common to want to include information about the other resource referenced by the URI of the link. This information might be included in the rendering of the page of the current resource, or on conventional PCs that have a mouse as an input device it might appear in a pop-up window when the mouse is over the link. [Obviously, the pop-up window technique does not apply for touch-screen devices like smart-phones and tablets, or at least the pop-up has to be triggered by a different gesture.] OSLC defines a protocol that makes it convenient for an application to obtain a small amount of information about the resource that is the target of a link for the purposes of this sort of display.

UI preview consists solely of the definition of a new media type, and a representation format for that media type. The new media type is `application/x-oslc-compact+xml`. When a GET is performed on a resource with this media type, a conforming implementation is expected to return a small amount of information about the resource suitable for display purposes. This information is in RDF format and includes the following properties:

- The title of the resource
- A shorter title of the resource
- The URL of an icon for the resource. The client application is expected to perform a subsequent GET to retrieve the icon.
- The URL of an HTML document suitable for displaying in a small pop-up window. The client application is expected to perform a subsequent GET to retrieve the HTML.
- The URL of an HTML document suitable for displaying in a larger pop-up window. The client application is expected to perform a subsequent GET to retrieve the HTML.
- Some information on sizing for these smaller and larger pop-ups

OSLC requires that this information be returned in RDF/XML format – other RDF or non-RDF representations are not allowed.

UI Preview example

```
<?xml version="1.0" encoding="UTF-8"?>

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dcterms="http://purl.org/dc/terms/"
  xmlns:oslc="http://open-services.net/ns/core#">

 <oslc:Compact
  rdf:about="http://example.com/bugs/12345">

  <dcterms:title> 12345: &lt;s&gt;Null pointer exception during startup&lt;/s&gt; </dcterms:title>
```

```
  <oslc:shortTitle> 12345 </oslc:shortTitle>
  <oslc:icon rdf:resource="http://example.com/icons/defect.jpg" />

  <oslc:smallPreview>
    <oslc:Preview>
      <oslc:document rdf:resource="http://example.com/bugs/12345?hover=small" />
    </oslc:Preview>
  </oslc:smallPreview>


  <oslc:largePreview>
    <oslc:Preview>
      <oslc:document rdf:resource="http://example.com/bugs/12345?hover=large" />
      <oslc:hintWidth> 60em </oslc:hintWidth>
      <oslc:hintHeight> 20em </oslc:hintHeight>
    </oslc:Preview>
  </oslc:largePreview>

 </oslc:Compact>

</rdf:RDF>
```

OAuth

OSLC does not mandate a particular approach to authentication and access control, but it acknowledges the existence and use of OAuth. Use of OAuth requires a client to know 3 fixed URLs that are used to negotiate tokens. OSLC defines a resource, OAuthConfiguration, for holding these URLs, and defines an optional property on both ServiceProvider and ServiceProviderCatalog for holding OAuthConfiguration values. Clients will still need to consult the documentation specific to a tool to know whether OAuth is supported by the tool, and if so where the tool will store the OAuth URLs.