OASIS OPEN PROJECTS

Configuration Management Primer Version 1.0

Project Note 01
16 December 2021

This stage:

https://docs.oasis-open-projects.org/oslc-op/config-primer/v1.0/pn01/config-primer.html (Authoritative)
https://docs.oasis-open-projects.org/oslc-op/config-primer/v1.0/pn01/config-primer.pdf

Previous stage:

N/A

Latest stage:

https://docs.oasis-open-projects.org/oslc-op/config-primer/v1.0/config-primer.html (Authoritative)
https://docs.oasis-open-projects.org/oslc-op/config-primer/v1.0/config-primer.pdf

Open Project:

OASIS Open Services for Lifecycle Collaboration (OSLC) OP

Project Chairs:

Jim Amsden (jamsden@us.ibm.com), IBM
Andrii Berezovskyi (andriib@kth.se), KTH

Editor:

David Honey (david.honey@ibm.com), IBM

Related work:

This specification is related to:

- *OSLC Configuration Management Version 1.0. Part 1: Overview.* https://open-services.net/spec/config/latest

Abstract:

This primer serves as a guide to the concepts in the specification, and through the use of simple examples, explains how versioning and configurations are represented, how and when local configurations and global configurations are used, and lists the elements that an implementation should consider.

Status:

This is a Non-Standards Track Work Product. The patent provisions of the OASIS IPR Policy do not apply.

This document was last revised or approved by the Project Governing Board of the OASIS Open Services for Lifecycle Collaboration (OSLC) OP on the above date. The level of approval is also listed above. Check the "Latest stage" location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Open Project are listed at https://open-services.net/about/.

Comments on this work can be provided by opening issues in the project repository or by sending email to the project's public comment list oslc-op@lists.oasis-open-projects.org.

Citation format:

When referencing this specification the following citation format should be used:
[OSLC-Config-Primer-1.0]
Configuration Management Primer Version 1.0. Edited by David Honey. 16 December 2021. OASIS Project Note 01. https://docs.oasis-open-projects.org/oslc-op/config-primer/v1.0/pn01/config-primer.html. Latest stage: https://docs.oasis-open-projects.org/oslc-op/config-primer/v1.0/config-primer.html.

Notices

Table of Contents

1. Outline
2. Motivation
    2.1 Problem
    2.2 Solution
    2.3 Business Value
3. Concepts
4. Representation of version resources
5. Local configurations
6. Global configurations
7. UML Class diagram
8. Component skew and contribution order
9. Branching
10. Implementation elements
    10.1 Introduction
    10.2 Components
        10.2.1 Minimal elements
        10.2.2 Optional elements
    10.3 Configurations
        10.3.1 Minimal elements
        10.3.2 Optional elements
11. Concept resources and version resources
    11.1 Minimal elements
    11.2 Optional elements
12. References

Appendix A. Acknowledgements

1. Outline

The OSLC Configuration Management specification is a reference for servers that want to expose configuration management to OSLC clients. However, the specification is not organized as a tutorial or introduction about how to implement an OSLC configuration management compliant server or adapter.

This primer serves as a guide to the concepts in the specification, and through the use of simple examples, explains how versioning and configurations are represented, how and when local configurations and global configurations are used, and lists the elements that an implementation should consider. The specification remains the definitive source, and where this primer differs from the specification, the specification should be taken as the authoritative reference.

2. Motivation

2.1 Problem

Any software development project team creating anything but the simplest, short-lived solutions knows the value of source code management (SCM). Without SCM, teams could not reliably recreate the source files that were used to build a specific release, making it impossible to reliably maintain what the team has delivered.

However, a large development project involves much more shared information that just source files. Complex products and systems are mix of software, electronics, and hardware, with software taking an increasing role. Solution components are often developed by different organizations, using different method and tools, with lifecycle information stored in different repositories. The design and development of these complex systems requires many kinds of artifacts. Engineers in specialist disciplines produce these artifacts employing various engineering methods, disciplines and tools. Often these artifacts are not under Configuration Management, or if they are, it is done manually with half-measures that incur trade-offs. These challenges exist for software-only systems, yet their magnitude is much greater for products with physical, electrical and software aspects.

For example, specific requirements are often associated with a particular release of components of a system. Analysis and design artifacts are created to understand how to address the impact of, or realize those requirements. Test cases and test results are used to validate that the work done on the systems actually meets the requirements.

All this shared information is generally not available under the control of a single organization, set of tools or content management system, and it is often stored in repositories, not files in the file system. Like source files, all this shared information should be managed in a controllable, predictable way that is coordinated with specific solution releases.

Development teams need an efficient, effective means of managing versions and variants of artifacts across the whole systems development lifecycle. They need to be able to capture, preserve, compare, merge and potentially recreate specific sets of versioned information covering the whole lifecycle in order to know who did what, when, and why.

2.2 Solution

The OSLC Configuration Management specification defines an RDF vocabulary and a set of REST APIs for managing versions and configurations of linked data resources from multiple domains. Using client and server applications that implement the configuration management vocabulary and REST APIs allow a team administrator to create configurations of versioned resources contributed from tools and data sources across the lifecycle. These contributions can be assembled into aggregate (global) configurations that are used to resolve references to artifacts in a particular and reproducible context.

Team members set the configuration context in each of the tools they use to refer to the particular global configuration that represents the state of the systems they are working on. They can create branches to support parallel development, and compare and merge branches to flow changes as needed. Product managers can create branches that represent different variants of a product in order to separate variation points and ensure changes are only applied to the appropriate variants.

Development teams can create baselines that preserve the state of the federated, shared information in order to be able to recreate that state for any reason, including for regulatory compliance or for applying maintenance updates to released versions and/or variants of a product.

2.3 Business Value

The primary purpose for versioning and configuration management is to be able to establish and restore or recover a particular set of related resources in order to know and manage who changed what, when, and why. Applying these techniques to other disciplines, and combining them together, offers new opportunities for parallel development, managing change, reuse, and management of product variants in complex development projects.

The benefits of source-code management are well known. However, achieving those benefits across all of the information that goes into designing, building, managing and governing the source code can be a challenge. This is because having multiple change and configuration management solutions for different artifact types requires manual coordination of versions and variants across the tools. The OSLC Configuration Management specification specifies a standard way in which different tools, developed by different suppliers, at different times, using different repositories and data representations can contribute version and configuration information across the whole lifecycle.

The resulting business value includes:

- Cross-tool version and configuration management related shared information
- Support for context-specific link navigation and management
- Automatic configuration management of complex solutions and systems
- Enable parallel, independent, development streams with branching and merging of all lifecycle information
- Save and restore consistent sets of related information for any purpose including regulatory compliance and product maintenance releases
- Manage different variants of a related set of deliverables, solutions and/or products
- Reuse utilizing commonality/variability for Product Line Engineering (PLE)

Some typical uses of versioning and configuration management capabilities include:

- Creating baselines of development streams in order to capture and preserve the state of a system at different points in time
- Use release stabilization streams to complete development of specific releases while allowing ongoing future development to continue in parallel Branching streams for different purposes including experimentation, A/B testing, gradual production rollout, importing from a supplier, etc., and delivering changes on streams to a common shared stream in a controlled manner.
- Parallel development on shared artifacts, overlapping releases or the common parts of product variants with the ability to compare different streams and flow changes as needed to meet requirements, enforce enterprise asset management and governance, and promote reuse.
- Create maintenance branches and control propagation of maintenance changes into ongoing development streams in a controlled, predictable manner
- Managing configurations of related, federated, shared information across the whole lifecycle.
- Isolate changes for different variants on different streams while also managing common, reusable components.
- Enable reuse of lifecycle artifacts including requirements and test cases, as well as the relationships between them, including regulatory and security requirements and test cases.
- Enable analysts and designers to work one iteration ahead in agile projects in order to better inform iteration planning and development.
- Enable progressive contribution of parts of a large system with baselines that capture each significant evolution of the system.
- Support different approaches to reuse by branching from product streams that are closes to the new variant, or branching from a common base that is maintained as a reusable enterprise asset.
- Enable concurrent development of shared enterprise assets in different product delivery streams (requirements delivery life-cycles) while at the same time managing and governing changes to long-lived enterprise assets (asset management life-cycles).

3. Concepts

Configuration Management on Wikipedia defines Configuration Management as follows:

> Configuration management (CM) is a systems engineering process for establishing and maintaining consistency of a product's performance, functional, and physical attributes with its requirements, design, and operational information throughout its life.

In Configuration Management, artifacts are versioned. For example, a new requirement R1 might be defined and created. Once the content of that requirement is checked in or committed, version 1 of that requirement R1 exists. At that point, the contents of version 1 cannot be changed. If changes are required, the content is changed, and when checked in or delivered, results in version 2 of requirement R1. Version 2 of R1 was created from, or derived from, version 1 of R1. The sequence of versions of that requirement comprise its *version history*. In OSLC Configuration Management, the term *concept resource* is used to mean all the versions of some artifact. In our example, requirement R1 is a concept resource (no version is specified), and requirement R1 version 1 is a specific version of that concept resource. Many versioning systems exist, and there are a number of different approaches as to how artifacts are versioned, when those versions are created, and when a version becomes non-modifiable. However, all versioning systems support some notion of when a change is committed, the version that records that change becomes non-modifiable. The OSLC Configuration Management specification does not define how versioning should be implemented. For example, some versioning systems assign *version identifiers* separately for each versioned resource, while other versioning system version an entire repository so that a version identifier is "global" across all versioned resources. Servers are free to choose existing versioning systems, or implement a versioning system of their own design.

An important element of Configuration Management is a *configuration*. A configuration defines what set of version resources are used in that configuration. For example, configuration C1 might use version 1 of requirement R1, and configuration C2 might use version 2 of requirement R1. New artifacts might be added to or obsolete artifacts removed from a configuration. The configuration therefore provides a view of the appropriate artifacts and versions of those artifacts that apply in that configuration. The OSLC Configuration Management specification defines this in terms of *selections*. A configuration might reference a *selections* resource that in turn references the specific version resources that belong in that configuration.

In OSLC Configuration Management, a *stream* is a modifiable configuration in which artifacts may be added or removed, or a different version of an artifact may be selected by a user to replace some other version of that artifact. Streams are the configurations in which ongoing work is performed. An essential element of Configuration Management is the ability to create a non-modifiable record of the set of version resources at specific milestones or points in time in order to provide traceability and auditing. In OSLC Configuration Management, a *baseline* is a non-modifiable configuration whose set of version resources are also non-modifiable. Usually a baseline is created from a stream, recording the state of that stream. The stream continues to be modifiable, but the baseline is now a non-modifiable record of the state of the stream at the time the baseline was created. Every configuration is associated with a *component*. A *component* is a unit of organization consisting of a set of version resources. When a *baseline* is created from a *stream*, both the baseline and the stream are for the same *component*. The granularity of a component is up to an application designer.

OSLC Configuration Management supports the idea that a configuration may be a container of other configurations. The term *contribution* describes a resource that has both a *contribution order* property and a reference to a configuration that is used by a parent configuration. Strictly speaking, the term *contribution* means a contribution resource, but is sometimes used informally as a shorthand for child configuration which is a *contributed configuration*. Configurations may thus form a *hierarchy* of configurations. The term *global configuration* is used to describe a configuration that aggregates configurations, especially those from other configuration management servers. For example, a global configuration might have contributions from a requirements management server, a quality management server, and a source code control server. The global configuration thus represents the state of version resources across those application servers.

There is a UML class diagram in section 7 that describes how these concepts, and those in following sections, relate to the resources described in the OSLC Configuration Management Specification.

4. Representation of version resources

We will use an example of two requirements, A and B, in order to explain some of the concepts and representations. From a concept resource point of view, one might think of the requirements as having some basic properties as shown below. Requirement B *refines* requirement A.

```
Requirement_a
    id: a
    name: "Requirement A"
    description: "A description of requirement A"


    Requirement_b
    id: b
    name: "Requirement B"
    description: "A description of requirement B"
    refines: Requirement_a
```

Let's look at a possible RDF representation of requirement A version 1 (`requirementA-v1`). The RDF graph of `requirementA-v1` might look something like:

```
:requirementA-v1
        a oslc_config:VersionResource ;
        dcterms:isVersionOf :requirementA .
    :requirementA
        a oslc_rm:Requirement ;
        oslc_config:versionId "v1" ;
        oslc_config:component :rmComponent1 ;
        dcterms:identifier "A" ;
        dcterms:title "Requirement A"^^rdf:XMLLiteral ;
        dcterms:description "A description of requirement A version 1"^^rdf:XMLLiteral .
```

There are two important things to note:

1. The main properties of the requirement are made using a subject URI of the *concept resource* `:requirementA` and **not** the URI of the specific version `:requirementA-v1`. This is done so that the same subject is used for properties of all the versions of that concept resource. A query against the concept URI `:requirementA` and `dcterms:description` would give the descriptions of requirement A across all versions.
2. There are a few statements made using the subject URI of the version resource `:requirementA-v1` that specify the artifact is a version resource and its concept resource.

Now consider that requirement A has been updated with a new description, resulting in version 2 (`requirementA-v2`):

```
:requirementA-v2
        a oslc_config:VersionResource ;
        dcterms:isVersionOf :requirementA ;
        prov:wasRevisionOf :requirementA-v1 .
    :requirementA
        a oslc_rm:Requirement ;
        oslc_config:versionId "v2" ;
        oslc_config:component :rmComponent1 ;
        dcterms:identifier "A" ;
        dcterms:title "Requirement A"^^rdf:XMLLiteral ;
        dcterms:description "A description of requirement A version 2 (changed description)"^^rdf:XMLLiteral .
```

The statements using subject URI `:requirementA-v2` include `prov:wasRevisionOf :requirement-v1`, indicating that version 2 was derived from or created from version 1 of requirement A. The `dcterms:description` of the concept resource specifies the changed description for version 2.

Similarly, one might have requirement B version 1 (`requirementB-v1`):

```
:requirementB-v1
        a oslc_config:VersionResource .
        dcterms:isVersionOf :requirementB .
    :requirementB
        a oslc_rm:Requirement ;
        oslc_config:versionId "v1" ;
        oslc_config:component :rmComponent1 ;
        dcterms:identifier "B" ;
        dcterms:title "Requirement B"^^rdf:XMLLiteral ;
        dcterms:description "A description of requirement B version 1"^^rdf:XMLLiteral ;
        oslc_rm:refines :requirementA.
```

Note that the `oslc_rm:refines` link is to the *concept resource* of requirement A `:requirementA` and **not** to a version of that requirement. While an implementation *could* link to a specific version resource, that's usually undesirable. Consider what would happen if `requirementB-v1` referenced `requirementA-v1`. When version 2 of requirement A is created, one might want requirement B to reference it. This would require that a new version 2 of requirement B be created in order to update that link. This results in a lot of new versions because the links themselves are referencing specific versions.

If the link references the concept resource, the *resolution* of which version of requirement A is referenced, is decoupled and deferred. A user that views requirements A and B in a particular *configuration context* will see the versions of both requirements *resolved* in that configuration context. If requirement A was updated to version 2 in that context, then `requirementB-v1` will have a *refines* link that now *resolves* to `requirementA-v2`. This avoids the need to update requirement B to reference that new version of requirement A.

Typically applications provide some means for a user to select their *configuration context* in their user interface. For work in progress, this is usually a stream. If the user wants to see the versions of artifacts that were used at an important milestone, they would typically select the baseline that was created for that milestone. From a REST API standpoint, a configuration context can be specified using either a `Configuration-Context` header or a `oslc_config.context` parameter in the request.

5. Local configurations

A configuration created in and used by an application to manage the selected versions in that configuration is called a *local configuration*. Using the previous example based on requirements, let's imagine we have a stream in a requirements management application that selects requirements `requirementA-v1` and `requirementB-v1`. The RDF representation of that stream might be as follows:

```
:rmStream1
        a oslc_config:Stream ;
        dcterms:title "First requirements management stream"^^rdf:XMLLiteral ;
        oslc_config:component :rmComponent1 ;
        oslc_config:selections :rmStream1Selections ;
        oslc_config:acceptedBy oslc_config:Configuration .
```

A configuration always has a component. The stream references zero, one, or more *selections* resource(s) that specify what version resources are selected by the stream. In our example, the selections resource might be as follows:

```
:rmStream1Selections
        oslc_config:selects :requirementA-v1, :requirementB-v1 .
```

A user might want to capture the state of that stream by creating a *baseline*. The RDF representation of such a baseline might be as follows:

```
:rmBaseline1
        a oslc_config:Baseline;
        dcterms:title "First requirements management stream (first baseline)"^^rdf:XMLLiteral ;
        oslc_config:component :rmComponent1 ;
        oslc_config:selections :rmBaseline1Selections ;
        oslc_config:baselineOfStream :rmStream1 ;
        oslc_config:acceptedBy oslc_config:Configuration .
```

The `oslc_config:baselineOfStream` property specifies the stream that the baseline was created from. The selections resource referenced by that baseline might be as follows:

```
:rmBaseline1Selections
        oslc_config:selects :requirementA-v1, :requirementB-v1 .
```

Creating that baseline from the stream results in that stream's representation including an [updated] `oslc_config:previousBaseline` property:

```
:rmStream1
        a oslc_config:Stream ;
        dcterms:title "First requirements management stream"^^rdf:XMLLiteral ;
        oslc_config:component :rmComponent1 ;
        oslc_config:selections :rmStream1Selections ;
        oslc_config:acceptedBy oslc_config:Configuration ;
        oslc_config:previousBaseline :rmBaseline1 .
```

When requirement A was changed to create version 2, the stream `rmStream1` would be updated to select that version:

```
:rmStream1Selections
        oslc_config:selects :requirementA-v2, :requirementB-v1 .
```

The baseline `rmBaseline1` continues to select `:requirementA-v1` and `:requirementB-v1`. The selections of a baseline cannot be modified. The baseline therefore serves as a non-modifiable record of the state of the stream from which it was created at a specific point in time.

6. Global configurations

A *global configuration* is a configuration used to assemble other configurations (typically from other application servers) into a hierarchy. To understand the usage, let's consider that we have a quality management application that has a test case `testCaseA` that validates `requirementA`. Version 1 of that test case might have an RDF representation as follows:

```
:testCaseA-v1
        a oslc_config:VersionResource ;
        dcterms:isVersionOf :testCaseA ;
    :testCaseA
        a oslc_qm:TestCase ;
        dcterms:title "Test case validating requirement A"^^rdf:XMLLiteral ;
        dcterms:description "Details of how the test case validates requirement A"^^rdf:XMLLiteral ;
        oslc_qm:validatesRequirement :requirementA .
```

In this example, the quality management application has its own local stream that selects `testcaseA-v1`:

```
:qmStream1
        a oslc_config:Stream ;
        dcterms:title "First quality management stream"^^rdf:XMLLiteral ;
        oslc_config:component :qmComponent1 ;
        oslc_config:selections :qmStream1Selections ;
        oslc_config:acceptedBy oslc_config:Configuration .
```

with its referenced selections resource:

```
:qmStream1Selections
        oslc_config:selects :testCaseA-v1 .
```

A user looking at test cases in the context of configuration `qmStream1` would see `testCaseA-v1`. That test case has a *validates requirement* link to requirement A. However, in that configuration context, it is not possible to *resolve* the link to a version of the requirement concept resource because that is managed by the requirement management application and not the quality management application.

This is where a *global configuration* is useful. As an example, say we have a global stream that has *contributions* from `rmStream1` and `qmStream1`. Its RDF representation might be as follows:

```
:globalStream1
        a oslc_config:Stream ;
        dcterms:title "First global stream"^^rdf:XMLLiteral ;
        oslc_config:accepts oslc_config:Configuration ;
        oslc_config:acceptedBy oslc_config:Configuration ;
        oslc_config:contribution :contribution1, :contribution2 .
    :contribution1:
        oslc_config:configuration :rmStream1 ;
        oslc_config:contributionOrder "1" .
    :contribution2:
        oslc_config:configuration :qmStream1 ;
        oslc_config:contributionOrder "2" .
```

A user is working in the configuration context `globalStream1`. In the requirements management application, when the user looks at the requirements for component `rmComponent1`, they see `requirementA-v2` and `requirementB-v1`. The requirements management application resolves the component `rmComponent1` in the context of global configuration `globalStream1`. The global configuration has only one contribution associated with `rmComponent1` and that's for the local configuration `rmStream1`. That local configuration selects `requirementA-v2` and `requirementB-v1`. Similarly, in the quality management application, when the user looks at the test cases for component `qmComponent1`, they see `testCaseA-v1`.

For that test case, the quality management user sees the *validates requirement* link to `requirementA-v2`. The quality management application only knows the link is to the test case concept resource `requirementA` and cannot resolve which version of that requirement is referenced. This is because the requirements management application that owns that requirement manages the configurations that select it. So how does the quality management application resolve the link to the requirement to a specific version?

When referencing that requirement, it uses a `oslc_config.context` query parameter or a `Configuration-Context` header with a value that is the URI of the configuration context. For example `":requirementA?oslc_config.context=':globalStream1'"`. The parameter value is shown here without URL encoding for clarity. In practice, parameter values should be URL encoded. When the requirements management application gets a request to fetch that requirement, it resolves the concept resource URI in the specified configuration context. In this example, it knows that the requirement concept resource `requirementA` is owned by component `rmComponent1`, and resolves that component in the global configuration context `globalStream1` to the local configuration `rmStream1`, and then resolves the requirement concept resource `requirementA` to `:requirementA-v2`.

How does an application resolve a component in a global configuration context to a local configuration?

Say an RM application wants to resolve component `rmComponent1` in the context of `globalStream1`. The starting point is to consider the hierarchy of the global configuration `:globalStream1` and associated components. The previous example yields the tree shown below:

```
    globalStream1 (component=globalComponent1)
```



```
rmStream1 (component=rmComponent1, order=1)
qmStream1 (component=qmComponent1, order=2)
```

An application traverses this tree looking for component `rmComponent1` and finds `rmStream1`. When the tree only contains a single configuration for a particular component, the order of traversal does not matter.

In practice, constructing a contribution tree by fetching data each time a component needs to be resolved to a local configuration does not scale well or provide the performance required for frequent resolution. Implementations commonly use some form of caching of that data and use an implementation dependent way of maintaining that cached data.

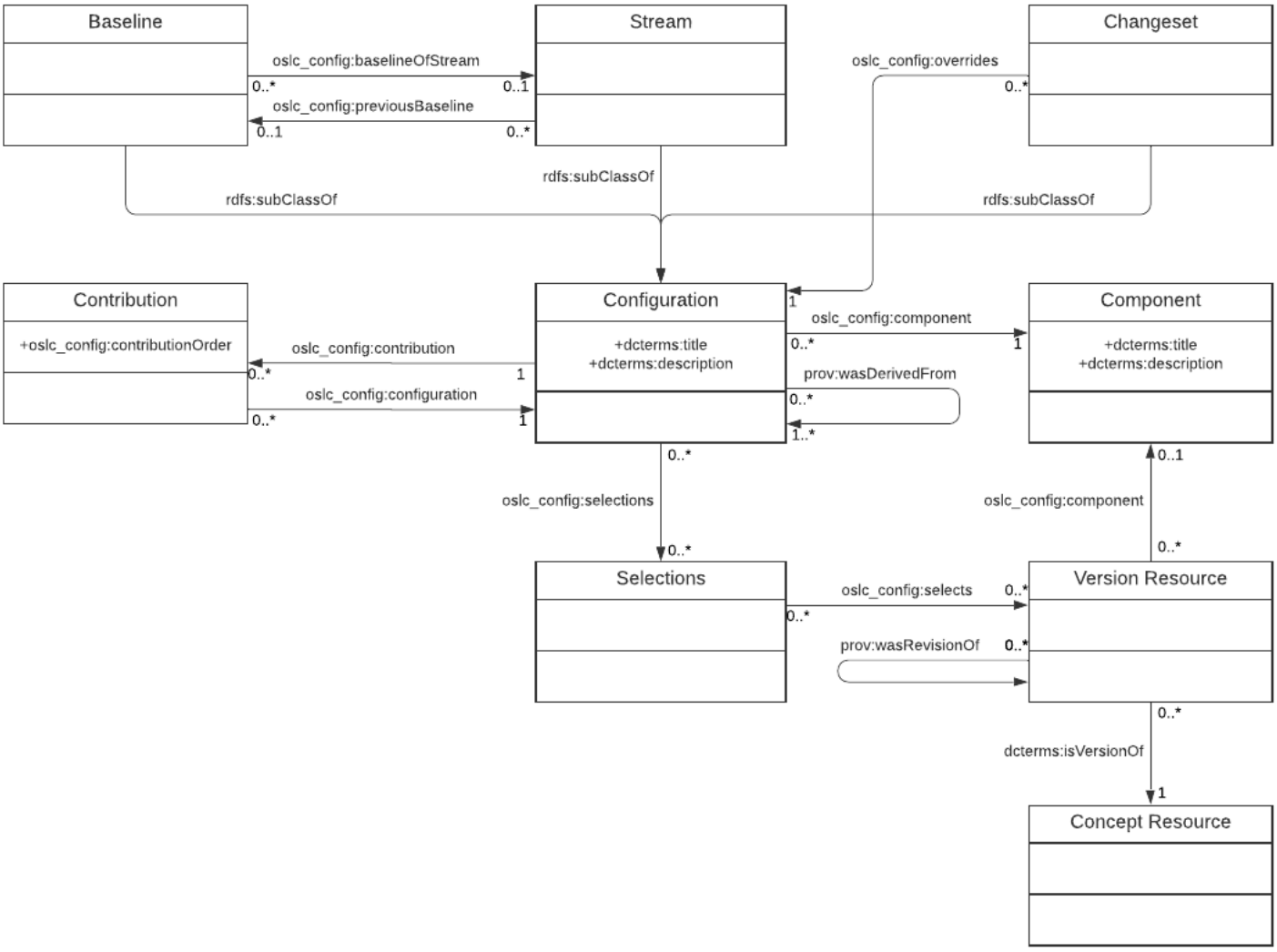
Global configurations therefore fulfill two important functions:

- They allow a configuration context to be used across applications that allow an application to resolve their concept resources to the right versions.
- They allow links between concept resources in *different* applications to be viewed and navigated to the right versions of those artifacts in that configuration context.

The distinction between a local configuration and a global configuration is a fuzzy one. Consider a requirements management server that supports configurations that have contributions from other configurations in the same server but not other servers. Should such a configuration be called a local configuration or a global configuration? The distinguishing characteristics of a global configuration are that they are normally usable as contributions in any other global configuration and may use contributions from any local or global configuration. So such a configuration would be more accurately termed a local configuration even though it has contributions.

7. UML Class diagram

The UML class diagram below shows the main elements of the configuration management specification. It intentionally omits some relationships for the sake of clarity. The resource shapes in the specification remain the definitive reference.



Some key aspects of the representation are:

- The version resources selected by a configuration are determined using the property path `oslc_config:selections/oslc_config:selects`.
- The immediate contributed configurations to a configuration are determined using the property path `oslc_config:contribution/oslc_config:configuration`. Contribution resources are embedded within configurations.
- Global configurations often only have contributions and not selections.
- Local configurations often only have selections and not contributions.

8. Component skew and contribution order

Consider a global stream `globalStream2` that has contributions from `rmStream1`, `rmBaseline1`, and `:qmStream1` in that order. `rmStream1` and `rmBaseline1` are for the same *component*. Where a configuration hierarchy has contributions from two *different* configurations of the same component, this is called *component skew*.

Now consider the contribution tree for `globalStream2`:

```
globalStream2 (component=globalComponent1)
        rmStream1 (component=rmComponent1, order=1)
        rmBaseline1 (component=rmComponent1, order=2)
        qmStream1 (component=qmComponent1, order=3)
```

When resolving the concept `requirementA` in the context of `globalStream2`, should this resolve to `requirementA-v2` selected by `rmStream1` or `requirementA-v1` selected by `rmBaseline1`? The component skew is resolved by the order in which the contribution tree is traversed. If the traversal algorithm orders `rmStream1` before `rmBaseline1`, then component `rmComponent1` resolves to `rmStream1`, and the `requirementA` resolves to `requirementA-v2`.

The configuration management specification does not prescribe a traversal algorithm, but a common approach is to perform a depth-first, sibling ordered by contribution order second traversal. Consider the following configuration hierarchy:

```
globalStream2 (component=globalComponent1)
        globalStream3 (component=globalComponent2, order=1)
            rmStream1 (component=rmComponent1, order=1)
        globalStream4 (component=globalComponent3, order=2)
            rmBaseline1 (component=rmComponent1, order=1)
            qmStream1 (component=qmComponent1, order=2)
```

A depth-first, sibling ordered by contribution order second traversal results in the following traversal order:

```
globalStream2
    globalStream3
    rmStream1
    globalStream4
    rmBaseline1
    qmStream1
```

Note that the contribution order should be sorted as an ASCII string rather than being treated as a numeric string.

9. Branching

The OSLC Configuration Management specification does not define whether a versioning system should support branching and/or parallel versions, and if so how these are managed. Some versioning systems support branches as first-class objects, and there are actions to create branches. Some versioning system handling branching by copying data into different paths in the repository, referencing the shared artifacts. Other versioning systems regard branching as simply parallel versions, perhaps for different variants or purposes. Because of these different approaches, branching is not covered by the specification so that it remains implementable across a wider range of systems.

From an OSLC standpoint, clients create new versions of concept resources by a PUT on the concept URI with a configuration context specifying the stream that will be updated to use the new version. The version that was previously used in that stream is the predecessor version of the newly created version. For example, if `:rmStream1` uses `requirementA-v1`, a new version `requirementA-v2` might be created from it by a `PUT requirementA?oslc_config.context=%3ArmStream1`. If a client then requests a new version be created from `requirementA-v1` in some other stream, an implementation that supports branching might create `requirementA-v2.1` as a parallel version to `requirementA-v2`. An implementation that only supports linear versioning might fail the request.

Typically, parallel development for different branches or purposes is performed in separate *streams*. For example, parallel streams might be created from the same baseline in order to work on separate features for some release. The OSLC Configuration Management specification does not define any mechanisms for merging streams. However, a client can PUT merged content to the concept resource where the merged content has multiple `prov:wasRevisionOf` statements indicating the contents were merged from those predecessor versions.

10. Implementation elements

10.1 Introduction

When adding support for OSLC Configuration Management to an application, or writing an adapter to provide such support to an existing application, there are a number of elements to the implementation. To gain the full power of configuration management, an implementation should support both version resources and configurations. An implementation need not follow the ordering of the following sections. However, the ordering reflects some of the dependencies that an implementation is likely to face. For example, versioned artifacts belong to a component, so support for components might be better implemented before versioned artifacts. Within each section, a minimal set of elements is described, followed by optional elements. An implementation might want to tackle the minimal elements first, then revisit each of the sections and consider tackling some or all of the optional elements.

Some applications support a notion of a *project* or *project area* that serve as a container of resources and for managing access to those resource in that container. While this notion is not part of any OSLC specification, it is mentioned here because it may affect the implementation. Applications that support this or similar containers, might want to declare a separate `oslc:ServiceProvider` for each such container. For example, this is a common convention for IBM Jazz applications.

10.2 Components

10.2.1 Minimal elements

At a minimum, an implementation should provide REST support for:

- `GET` of a component URI. The RDF of a component should include a `oslc_config:configurations` statement to a *Linked Data Platform Container* (LDPC). The response should include an `Etag` header whose value represents the state of the component.
- `PUT` of a component. The implementation should require the use of an 'If-match' header whose value matches the current Etag.
- `GET` of a component's configurations LDPC, returning a container that references all the configurations of that container.

While it is valid to create a component that initially does not have any configurations, it is good practice when creating a component to do either of:

- Create an empty initial baseline, then create an empty default stream from that baseline.
- Create an empty default stream.

10.2.2 Optional elements

- Support resource shapes for components. Each component should include in its RDF representation an `oslc:instanceShape` property that references a resource shape for that component.
- Support an OSLC selection dialog of components. This should be declared in an `oslc:Service` that is discoverable from a `oslc:ServiceProviderCatalog`.
- Support for an `oslc:CreationFactory` for components. This should be declared in an `oslc:Service` that is discoverable from a `oslc:ServiceProviderCatalog`. The creation factory should reference a resource shape for components.
- Support `POST` on a component's configurations LDPC as a means of creating a new stream for that component.
- Support OSLC query on components. An `oslc:QueryCapability` should be declared in an `oslc:Service` that is discoverable from a `oslc:ServiceProviderCatalog`. The query capability should reference a resource shape for the query container, and that resource shape should reference a value shape for the components that might be returned in that container.

10.3 Configurations

10.3.1 Minimal elements

At a minimum, an implementation should provide REST support for:

- `GET` on a configuration URI. If the application manages versioned resources, the RDF representation should include a `oslc_config:selections` statement to a selections resource. The response should include an `Etag` header whose value represents the state of that stream.
- `PUT` of a configuration. The implementation should require the use of an 'If-match' header whose value matches the current Etag.
- `GET` on a selections resource (if the application manages versioned resources). The RDF representation should include a `oslc_config:select` statement to each version resource URI selected by that configuration.

10.3.2 Optional elements

- Support resource shapes for configurations. Each configuration should include in its RDF representation an `oslc:instanceShape` property that references a resource shape for that configuration.

- Support `oslc_config:streams` property in the RDF of a baseline, and support a `GET` on that LDPC. Optionally support `POST` on that LDPC as a means of creating a stream from the baseline.
- Support `oslc_config:baselines` property in the RDF of a stream, and support a `GET` on that LDPC. Optionally support `POST` on that LDPC as a means of creating a baseline from the stream.
- Support an OSLC selection dialog of configurations. This should be declared in an `oslc:Service` that is discoverable from a `oslc:ServiceProviderCatalog` and its referenced `oslc:serviceProvider` members.
- Support for an `oslc:CreationFactory` for configurations. This should be declared in an `oslc:Service` that is discoverable from a `oslc:ServiceProviderCatalog` and its referenced `oslc:serviceProvider` members. The creation factory should reference a resource shape for configurations.
- Support OSLC query on configurations. An `oslc:QueryCapability` should be declared in an `oslc:Service` that is discoverable from a `oslc:ServiceProviderCatalog` and its referenced `oslc:serviceProvider` members. The query capability should reference a resource shape for the query container, and that resource shape should reference a value shape for the configurations that might be returned in that container.
- Support `DELETE` of a local configuration.

11. Concept resources and version resources

11.1 Minimal elements

At a minimum, an implementation should provide REST support for:

- `GET` on a version resource URI. The RDF representation of the version resource should:
  - Include statements against the version URI declaring it of type `oslc_config:versionResource`, and `dcterms:isVersionOf` referencing the concept resource URI.
  - Include statements about the properties of the resource using the concept resource URI as the subject.
- `GET` on a concept resource with a local configuration context specified by a `oslc_config.context` query parameter or header. This should resolve the concept resource to the version resource and return the RDF of that version resource. This requires that the application be able to determine the component to which a specified concept resource belongs.

11.2 Optional elements

- Support `GET` on a concept resource with a global configuration context specified by a `oslc_config.context` query parameter or header. This should resolve the concept resource in the context of that global configuration to the version resource and return the RDF of that version resource. This is required if the application is to be used with global configurations. The only mechanism for resolving a local component to a local configuration in a global configuration context covered by the specification is for an application to `GET` the global configuration to discover its contributions, and then repeat this recursively on the contributions to discover the contribution hierarchy. Doing this each time resolution is required is likely to be expensive. Implementations should consider whether some form of local caching of contribution trees is required to meet performance goals of the application.
- Support `GET` on a concept resource without a configuration context specified by a `oslc_config.context` query parameter or header. The application should determine a *default configuration*, and then use that as the configuration to resolve the concept resource. should resolve the concept resource in the context of that global configuration to the version resource and return the RDF of that version resource. This is required if the application is to be used with global configurations.
- Support `POST` to a component with a `oslc_config.context` query parameter or `Configuration-Context` header to create a new concept resource.
- Support `PUT` to a concept resource with a `oslc_config.context` query parameter or `Configuration-Context` header specifying a stream in which to create a new version from a currently used version.
- Support `DELETE` of a version resource with a `oslc_config.context` query parameter or header to delete the current mutable version resource from a specified stream.
- Support an `oslc:CreationFactory` that can create new concept resources and their initial version in the context of a configuration.

12. References

https://oslc-op.github.io/oslc-specs/specs/config/oslc-config-mgt.html

Appendix A. Acknowledgements

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Project Governing Board:

James Amsden, IBM (co-chair)
Andrii Berezovskyi, KTH (co-chair)
Axel Reichwein, Koneksys

Techical Steering Committee:

James Amsden, IBM
Andrii Berezovskyi, KTH
Axel Reichwein, Koneksys

Additional Participants:

Nick Crossley
David Honey

