Repository matters

Steve Powell and Andy Wilkinson

April 27, 2009

Second draft

We define various aspects of repository and repository chain configuration and behaviour.

This work was started under the JIRA items `DMS-347` and `DMS-340` and continues extracurricularly.

Contents

1 Introduction

Repositories of module artefacts are used to supply versions of artefacts upon request. The high-level external view of a repository is simply a map: from type, name and version to the artefact (and its properties) itself. The artefact is usually represented by a Universal Resource Identifier (URI) which may be a local file URI or an `http` URI for more general use. The properties are generally complex lists of name and value pairs, and are used to filter requests.

This high-level view masks a number of (popular) implementations. There are: simply a directory of jar files, managed by a publish and retract mechanism; a pattern of files applied to a file system to select artefacts by search; a remote system supporting a REST-like get interface; a watched directory that automatically publishes and retracts when the files change; and finally, a chain of the other types, the top-level view being presented as a distributed over-ride of the individual maps.

This document describes the top-level interface of a repository, the parts of each implementation, and a relationship between each implementation and the top-level interface by a lifting relation—showing how each needs to support the external view.

The most interesting case is probably the chain, where arbitrary repositories can be linked together in a list. The requisite support for a chain is made clear from the lifting relation that relates the external view of each repository with the external view of the chain.

Operations on a repository are limited to the top-level view, and full error conditions are not presented here, though they may be indicated in the text.

Finally, we will look at an external representation of the repositories described here: how each of them are textually specified.

2 Basic notions

Artefacts in a repository have a *Type*, a *Name* and a *Version*. Though these have a structure, and are interesting in their own right, we will not attempt to model them. There are also *VersionRange*s, which are used to search the repository, but again, it is unnecessary to model these here.

Repositories have names (*RepoName*) and hold *MetaData* and a *URI* for each artefact stored in it. Essentially, repositories are therefore simple maps from *Type*, *Name* and *Version* to the *URI* and the *MetaData* of the artefact so identified. Since the *URI* should uniquely identify the artefact in the map,

the repository map is one-to-one (an injection).

$$
\begin{array}{|l}
\textit{Repository} \\
\hline
name : RepoName \\
repo : TNV \rightarrowtail URI \times MetaData \\
\hline
first \circ repo \in TNV \rightarrowtail URI \\
\hline
\end{array}
$$

where we have used the abbreviation TNV for $Type \times Name \times Version$.

The constraint is what stipulates that the URI uniquely identifies the artefact in the map: if restricted to just URIs, the repository map is *still* one-to-one.

3 Abstract operations

The initial state of a *Repository* is not fixed in the abstract, but depends upon the particular repository implementation. Therefore we do not stipulate this. However, there are some operations defined which we can (and ought) to define. Among these there are operations of 'getting an artefact', which exposes the repository map to external scrutiny; 'search for artefacts', which exposes the meta-data; 'publish', which explains how an artefact might be added to the repository (but should allow the repository to reject the request); and 'retract', which explains how an artefact might be removed from a repository (and might similarly be rejected).

In addition to these operations, which are well understood (but only 'reasonably' so), we wish to describe operations of 'start' and 'stop' which make the repository available (and become interesting in certain implementations) and have implications for externalised repositories.

Finally, we will want to describe what the operation 'getIndex' might do on a repository—being a way to externalise all the meta-data for the artefacts of a repository, in a way that exposes the level (version) of the index and also universalises the URI for remote access. Along with this operation there is a modified version of 'get artefact' which uses the external URI and is behind a 'REST' interface.

3.1 Core operations

3.1.1 Getting an artefact

This is a simple operation that asks for an artefact: instead of a simple TNV key, with a single *Version* stipulated, a version range is supplied on

the request. The result is an artefact that satisfies that range, in some sense which we leave not described. However, it will become apparent that implementations (in particular the chained repository implementation) rely on certain properties of this 'satisfaction' relation and we will document them (as free constraints) as they become apparent.

3.1.2 Searching for artefacts

A general query is not described of course, but can be defined in very abstract terms. The chained repository will need to combine query results additively, and this is therefore stipulated as a constraint on the nature of query requests supported. Notice that this constraint is not guaranteed by the version range (get artefact) operation request, and therefore the operations are distinct in order to allow different optimisation in each case.

3.1.3 Publish an artefact

To publish an artefact we

3.1.4 Retract an artefact

3.2 Management operations

3.2.1 Start

3.2.2 Stop

3.3 Hosting operations

3.3.1 Getting an index

3.3.2 Remote get an artefact

4 Implementations

The general structure of an implementation is the same in all cases. First there is a concrete state (which may be expressed in terms of other abstract states), then there is a relationship between the concrete and the abstract state, and finally there is the re-expression of the abstract operations in concrete terms. These ought to follow from the abstract operation specification and the state relationship, and as far as possible we will attempt to derive these. Where we are not able to do this completely, the extra conditions

will be described to show how the behaviour is exposed (or not) in the real implementations.

4.1 Managed repository

4.2 Remote repository

4.3 Watched repository

4.4 External repository

4.5 Chained repository