RFP 122 OSGi Bundle Repository

Draft

11 Pages

Abstract

As OSGi becomes part of the mainstream, the number of resources that a system has to define and use quickly becomes quite large. After the decision to develop large scale systems as modularized products, the problem of how to organize, categorize, search and define these systems becomes critical. A common repository definition which federates the ever growing number of resources available to the OSGi developer is desired such that it can supply the foundation for sophisticated tooling, integrated into all aspects of the software life cycle from discovery to development through QA to deployment and maintenance.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 7.1.

```
Source code is shown in this typeface.
```

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	*MAR 14 2009*	*Initial version* *Hal Hildebrand, Oracle Corporation hal.hildebrand@oracle.com*
0.1	*MAR 18 2009*	*Incorporated feedback from conference call* *Hal Hildebrand, Oracle Corporation* hal.hildebrand@oracle.com
0.2	*MAY 5 2009*	*Additional requirements from Pascal* *Hal Hildebrand, Oracle Corporation hal.hildebrand@oracle.com*
0.3	*MAY 7 2009*	*Incorporated changes from 05/07/09 conference call* *Hal Hildebrand, Oracle Corporation hal.hildebrand@oracle.com*
0.31	*MAY 7 2009*	*Transcribe additional edits from Pascal Rapicault* *Thomas Watson, IBM tjwatson@us.ibm.com*
0.32	*MAY 7 2009*	*Modifications to Pascal's edits* *Hal Hildebrand, Oracle Corporation hal.hildebrand@oracle.com*
0.4	*MAY 8 2009*	*Add requirement to associate arbitrary opaque data with entities* *Hal Hildebrand, Oracle Corporation hal.hildebrand@oracle.com*
0.5	*JULY 18 2009*	*Fixed section numbering and minor formatting and grammar. Added terminology. Added more use case details, a new use case for retirement, and a use case for secure metadata. Added a footnote disagreeing with the requirement of including optional dependencies, and thus mandating a specific resolution algorithm.* *David Kemper, TIBCO Software Inc. djk@tibco.com* *Eric Johnson, TIBCO Software Inc. eric@tibco.com*
0.6	*AUGUST 10 2009*	*Updated with review comments from F2F meeting in Dublin, Ireland, July 2009 and comments from the eeg mainling list* *Tim Diekmann, TIBCO Software Inc., tdiekman@tibco.com*

1 Introduction

This work is based on the excellent work by Richard Hall with the Oscar Bundle Repository and with the previous version of RFC 112 developed in collaboration with Peter Kriens. As RFC 112 was developed without an RFP, when the RFC 112 was taken back up, it was quickly discovered that the RFP process was critical to getting consensus around the RFC and thus this RFP was born.

2 Application Domain

OSGi specifications are being adopted at an increasing rate in the Java community. With the advent of modularization, the problem quickly becomes how to manage the large numbers of bundles both within any particular project, but across projects, companies and throughout the open source community. Repositories based on Maven and Ivy can provide some measure of support, but do not capture the richness and depth the OSGi dependency model is capable of.

When developing applications, we start from a vast base of open source and private libraries that are available to the project. The dependencies between these libraries quickly grows in complexity and developers need to quickly determine which versions of the required dependencies can be constructed to successfully run in the system being developed. To further complicate matters, a system under development is not static and evolves during its life cycle and complicated dependency relationships between different versions of the system need to be recorded and use to determine deltas, patches and the further dependency relationships of other systems on the developed modules.

Repositories provide discovery services which allow the developers and planners to browse and navigate these interrelated resources. These discovery services should be easily integrated into the tools used by developers so that they can quickly assemble existing solutions relevant to the problem being solved. These repositories not only span multiple organizations, both internal and external, but span many different mediums from shared discs to read only media to massive database backed content delivery systems.

When determining what to deploy and how to provision a system, for example, we would like to indicate only the critical and defining resources of the system and have the dependencies of this set determined automatically through a resolution process. This transitive closure over required resources should be installable from the information returned from the repository so that we can provision processes and applications with only the resources actually required. Note that the goal of this document is not to capture the requirements relative to the development of a provisioning system, but instead to provide the foundation for such a system (and others) to be built on top.

2.1 Terminology + Abbreviations

- Repository – A collection of entities and their metadata.
- Resource – An entity in the repository. For example, a bundle may be placed in a repository.
- Metadata – Attributes of a resource and the repository as a whole.
- Capability – A logical attribute provided by a resource. For example, a resource provides its own contents: a bundle resource may provide one or more Java packages.
- Constraint – An expression of a dependency on one or more capabilities. For example, a bundle resource may have a constraint on a Java package.
- Configuration– A set of metadata whose capabilities satisfy both an initial set of root constraints and their own transitive set of constraints.
- Resolution – The process of analyzing constraints and capabilities to arrive at some configuration.

3 Problem Description

Whenever a problem is solved, the reality is that an additional set of problems are created which didn't exist before the solution solved the problem. As developers adopt OSGi and begin to modularize their existing systems and develop green field systems using OSGi, the explosion of modules, their interrelated dependencies and the solution to constraints of desired systems quickly become hard, if not impossible to manage. The sheer number of bundles available even in the privately controlled repositories of large software development organizations quickly becomes overwhelming. As repositories spring up in public and private institutions, individual developers and organizations want to take advantage of the innovation happening in these other organizations and need a mechanism to federate these repositories into a coherent system which can be searched, queried, annotated and organized. System administrators need sophisticated tools which can quickly tell them with a reasonably degree of accuracy what the impact of the uptake of patch sets will have on existing installations. Without a common metadata facility which organizes the disparate resources into a constantly resolved sets these problems are pushed back to the organizations to develop, hindering the uptake and exploitation of OSGi within their development organizations.

4 Use Cases

4.1 Discovery

A developer on a large project knows that there are resources available which provide the solution for the problem at hand, but the question is where are these resources and perhaps more importantly, what will these resources require as dependencies and whether the available versions are compatible with the other constraints in the system.

The developer should have sophisticated tooling which can be configured with the certified repositories the project can draw upon. These tools will provide the developer with an interface which uses the supplied metadata of the repository to guide the developer in browsing the repositories by revealing only the resources which meet the constraints of the project being worked upon. These resources can be filtered by acceptable licenses and execution environment. When the developer finds the desired resources, the developer checks to see the particular licenses that the resources are offered under. The system then calculates the resources' dependencies based on the constraints current project being developed. The discovered resources and their dependencies are added to the project's definition and the developer then goes about her business using the discovered resources in the project.

4.2 Build

Resources necessary to do development are pulled from the information contained in the repository. These resources are available through the use of various build environments such as Maven and Ivy. The build may not necessarily declare the entire set of dependencies, rather these build systems will rely on the metadata in the repositories to pull in the dependent resources necessarily to build the systems. These systems will have the flexibility to define their dependencies using the full set of resource requirements, not limiting the expressiveness to jars but describing their dependencies using packages.

Further, different build and test scenarios may have different criteria for how to resolve dependencies. For example, in a continuous integration environment, it may be useful to provision a build environment with the latest known “stable” version, or the latest known “unstable” version. It is unlikely that the solution can anticipate all of the end-user specific requirements of these advanced resolution scenarios.

4.3 Provisioning

Once a system is built, many different configurations may be pulled from the set of system artifacts. Rather than deliver the entire set of system artifacts to every provisioned system, it is desirable to indicate the critical set of resources that define the system and let the resolution process determine what other resources are required. The provisioned systems can now use these sets of transitive closures to pull the required resources from common repositories.

Multiple products are often installed on the same machine and the producers of the software would like to share common resources between the separate installations. These installations are often customized by the end user and may be extended by end user content – e.g. an application server which end customer applications are deployed. These customizations and customer deployed content may change the resolved resources due to constraints such as a particular version need on particular resources by the customer. The repositories should be able to adapt to these changing requirements and integrate into repositories created by the end customer.

Again, different OBR clients may want to provision resources based on advanced resolution scenarios. For example, an OEM may wish to provision an application using only components with specific license terms. While licensing may be a capability of a resource, the licensing requirement is **not** a constraint of the resource. Rather it is a transitive constraint of an advanced resolution.

4.4 Patching

Successful software systems are rarely static in that they continually evolve, producing new versions and specialization. The producers of these systems would like to be able to statically determine the delta between different versions of the delivered software and deliver these subsets as patch sets to their customers. Customers should also be able to connect to vendor's repositories and calculate the resource sets required for them to move their particular installation to include a given set of critical security patches, relevant to their installations.

Inherent to this use case is the ability to analyze what was resolved. For example, an administrator wants to apply a patch to a running system. We want to detect that a resource added by the patch satisfies an optional dependency of a resource in the runtime that is currently missing. The administrator can make a decision about whether or not to re-resolve the existing runtime resource in light of this satisfied dependency.

4.5 Retirement

Given the metadata about resource dependencies, the administrator of a solution wants to remove resources that are offered by it. For example, a company may stop providing a particular version of a product, perhaps because of security issues. The company wants to remove all resources from the solution that are only used by that specific version of the product to be removed. Without some mechanism for removal, the solution will grow without bound.

4.6 Security

Some repository resources and metadata are more sensitive than others. For example, when populating a repository, its administrator wants to associate with a resource some security-related metadata like an SHA-256 hash, or a PGP signature. The administrator does not want this sensitive metadata changed through “normal” mechanisms.

4.7 Planning and Analysis

Once an organization has a unified repository with metadata about resources, they will likely want to use that metadata for planning and analysis. Metadata indicating something like “stable/unstable” might be used in conjunction with a resolution algorithm to identify all those resources needed in order to mark a particular resource as “stable.”

4.8 Using multiple repositories

When lining up multiple repositories in order of preference, the resolver may be able to prefer local repositories over remote repositories and expand its search as needed. This saves a lot of network hops if the dependencies are already satisfiable in the closest (local) repository. For build optimization a judgement can be made whether often used resources should be promoted to local repositories for faster access.

5 Requirements

5.1 Negative Requirements

1. **Consistent resolution throughout the development cycle**
   The solution SHALL NOT be required to produce identical sets of resolved resources throughout every phase of the development process.

2. **Identical resolution between the runtime and repository**
   The solution SHALL NOT be required to produce a set of resolved resources identical to the runtime resolution of a given OSGi container.

5.2 Functional

1. The solution MUST NOT preclude the ability to browse the repository via a web browser.

2. The solution MUST provide access to the physical bits of the resources so that they may be directly accessed.

3. The solution MUST handle dependency resolution such that resources can be deployed without generating errors[1].

4. The solution SHOULD NOT mandate that the query and the resolving part are within the same service. The resolver part MUST be exchangeable, but it does NOT HAVE to define the same result.

5. The solution MUST not preclude linking repositories, creating a federated repository that spans organizations and disparate media.

6. The solution MUST NOT require federation at the repository server.

7. The solution MUST provide programatic access to the repository.

8. The solution MUST NOT require a resolution mechanism on the repository server.

5.3 Discovery

1.

2. The solution MUST provide fquerying capabilities on all fields of the solution metadata..

3. The solution MUST allow the discovery of licensing of resources before the artifacts are downloaded.

1 Note that deployment of resolved set of resources to a running system may fail because of existing resources already present in the runtime. These are resources that the resolver was not able to take into account when creating the transitive closure of resources and consequently may have impacts on the deployment which would break the otherwise correct deployment into that runtime.
Further clarification from F2F in Dublin: This requirement is not meant to say that OBR resolution is required to be supported on the server side. Instead, the resolving process is expected to be performed in the runtime hosting the solution, see 5.2.8.

4. The solution MUST allow multiple licensing offerings for resources, but MUST NOT mandate the ability to download bundles for a given license..

5. The solution MUST NOT preclude the querying of resources through a RESTful API.

5.4 Dependency Resolution

1. The solution MUST be able to resolve the unresolved resources of a supplied initial set of requirements.

2. The solution MUST make it possible to identify any optional or mandatory dependencies and whether they are satisfied or not.[2]

3. The solution MUST NOT preclude a resolver from handling all the requirements, capabilities and their directives as defined in the OSGi R4 specifications.

4. The solution MUST be able to resolve a resource whose definition is not created by the solution, taking into account local runtime bundles, execution environment

5. The solution MUST NOT PRECLUDE extensibility beyond the 4.2 capabilities.The dependency resolution SHOULD be deterministic.

6. The solution MUST be able to perform resolution on a subset of the repositories known from the system.

7. The solution MUST NOT preclude repositories from being tiered to express client preferences on resolution location.

8. The solution MUST be able to perform resolution taking into account several repositories.

9. The end product of the resolver MUST BE more than just a configuration (a set of resources); it MUST also include references to the constraints and capabilities that were chosen.

10. The solution MUST NOT PRECLUDE the ability to provide enough access to the metadata to perform resolution external to the solution (the advanced resolution use case).

11.

5.5 Resources

1. The solution MUST allow for the metadata of the resource and the bytes of the resource to be stored on different physical servers.

2. The solution MUST allow for the bytes of the resources to be obtained from different servers.

3. The solution MUST NOT PREVENT a resource from being stored in multiple formats.

4. The solution MUST define an external format for the metadata of the repository.

5. The solution MUST allow for resource descriptors to not have an associated resource.

6. The solution MUST allow arbitrary opaque data to be associated with the resources, constraints and capabilities.

2 This sounds dangerously like requiring a specific resolution algorithm. Perhaps an OBR client wants to provision the smallest number of resources, and thus wants to explicitly ignore optional dependencies?

5.6 Non Functional

1. A repository conforming to the solution MUST NOT preclude the ability for the implementation to scale to hundreds of thousands bundles.

2. The solution MUST be possible to implement a repository with a simple file or set of files – i.e, a server must not be required to implement a repository or federated set of repositories.

3. The solution MUST define how to map OSGi metadata to OBR metadata.

6 Security Considerations

Security is an important part of the consideration of using the OBR, but is considered to be orthogonal to the solution. Issues such as mutual authentication between the client and the bundle repositories the client is using can be handled, for example, in much the same way that TLS is handled for secure access to web pages.

7 Document Support

7.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

[3]. The OSCAR Bundle Repository, Richard Hall. http://is.gd/novc

7.1 Author's Address

Name	Hal Hildebrand
Company	Oracle Corporation
Address	500 Oracle Pkwy, M/S 2op946, Redwood City, CA 94065
Voice	650 506 2055
e-mail	hal.hildebrand@oracle.com

Name	Tim Diekmann, Eric Johnson, David Kemper
Company	TIBCO Software Inc.
Address	3303 Hillview Ave, Palo Alto, CA 94304
Voice	650 846 1000
e-mail	tdiekman@tibco.com, eric@tibco.com, djk@tibco.com

7.2 End of Document