RFP-190: Resolution of JPMS modules

Draft

10 Pages

Abstract

Since Java 9 the fixed set of system packages was replaced by a mutable set of JPMS modules. This set can be defined at build time via static linking or introspected at runtime through reflection. This opens opportunities for OSGi to customize the JRE. This also means some application logic may come as JPMS modules not under the control of the OSGi developer. This RFP describes the requirements needed to use the Resolver service to discover the set of modules that are needed to satisfy a given set of OSGi bundles in order to enable further tool support for build-time interoperation between bundles and modules.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

0.2 Trademarks

0.3 Feedback

This document can be downloaded from the OSGi Alliance design repository at https://github.com/osgi/design The public can provide feedback about this document by opening a bug at https://www.osgi.org/bugzilla/.

0.4 Table of Contents

0.5 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 6.1.

`Source code is shown in this typeface.`

0.6 Revision History

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

Revision	Date	Comments
0.1	Sep 14 2018	Initial version based on rough notes and discussions within CPEG Todor Boev, Software AG, todor.boev@softwareag.com

1 Introduction

In today's world it has become increasingly important to create self-contained Java applications with minimal footprint. Even though we can resolve the minimal required set of bundles to include in an application often the total footprint of the application is dominated by the JRE itself.

With the introduction of the Java module system it became possible to statically link a JRE that reflects the needs of a concrete application. One problem that still hinders the creation of minimal OSGi runtimes is the lack of tools that can discover which JPMS modules are required by which OSGi bundles as well as the dependencies between the JPMS modules themselves.

This RFP introduces the requirements for a standard way to use the OSGi Resolver service to discover the dependencies between OSGi bundles and JPMS modules. In this way a complete description of the Java runtime can be obtained automatically. This should serve as the basis for tools that make the building of minimal OSGi applications practical.

This RFP is concerned primarily with the build and/or provisioning of OSGi applications. It assumes JPMS modules and OSGi bundles will remain separated at runtime as defined by OSGi R7.

2 Application Domain

In order to compete in the modern world web applications face unprecedented requirements for high availability and responsiveness at internet scale. For various reasons, outside of the scope of this RFP, this has lead to architectures where the application is decomposed into large number if independent simple processes or services. These architectures demand certain liquidity from their building blocks. Their binary images must be easy to move, their instances must be able to saturate a dynamically changing computational capacity, it must be possible to rapidly create and destroy large number of instances. This trend has progressed all the way to the "Function As A Service" (FAAS) architecture where instances must be used to service a single request and then be destroyed. In order to achieve this liquidity it is important for the services to have the smallest possible footprint both in storage and memory. The Java based services face increased competition from language runtimes like Node.js and Go which were built with these requirements in mind and often perform better in this respect.

On the opposite end of the spectrum when users deploy applications on their devices they expect a hassle-free experience. One way to achieve this is to deploy self-contained images. Again it is important for these images to be as small as possible in order to have quick download times and small footprint on the user device. Java applications however often require the users to first install a JRE, which must be of considerable size in order to support all future Java applications, rather than just the one being installed.

To address these needs Java SE 9 introduced the Java Platform Modular System (JPMS) which broke the JRE down into modules and provided a static linker tool which can build a smaller JRE from a subset of these modules. The modular system can also be used by application developers. The JPMS is inherently incompatible with OSGi and what's more both systems have independent dependency models which do not correlate by default. This leads to OSGi bundles being hosted in their own space within the JRE. These bundles however must still be supported by one or more modules from the JRE, which means they have hidden dependencies to those modules.

2.1 Terminology + Abbreviations

- **Bundle**
  For brevity the OSGi bundles will be referred to simply as "bundles".

- **Resource**
  A resource according to the Core OSGi specification *[needs reference]*. Resources are generic and can

for example represent any component of an application. The bundles are a kind of Resource specific to OSGi applications.

- **Resolver**
  The Resolver service as defined by the Core OSGi specification *[needs reference]*. Given a pool of Resources the resolver can discover a consistent sub-set where all Resource requirements are satisfied.

- **Module**
  For brevity the JPMS modules will be referred to simply as "modules".

- **Boot layer**
  The set of modules with which the JRE was started. The boot layer replaces the boot classpath in Java 9+ JREs.

- **Package**
  A java package or just "package".
- **Bundle packages**
  Bundles provide packages to other bundles through the `Export-Package` manifest header. Bundles consume individual packages from other bundles through the `Import-Package` manifest header. This RFP is not concerned with bundles consuming packages through the `Require-Bundle` manifest header.
- **Module packages**
  Modules provide packages to other modules through the `exports` declaration in `module-info.java.` In contrast to bundles, modules consume packages by requiring entire modules through the `requires` declaration in `module-info.java`.
- **System packages**
  Bundles can consume packages provided by modules from the boot layer if those packages are exported by the system bundle.

- **Module Service**
  Modules can register classes with the `java.util.ServiceLoader` utility through the `provides` declaration in `module-info.java`.
  Modules can declare they wish to use `java.util.ServiceLoader` to obtain all classes registered under a given interface through the `uses` declaration in `module-info.java`.
  Bundles can use `java.util.ServiceLoader` to load classes provided by modules from the boot layer.

3 Problem Description

At present bundles only depend on the boot layer as a flat list of system packages. OSGi R7 specifies that this list is discovered dynamically at runtime by the OSGi framework by introspection of the boot layer. Even if this discovery is made at build time this will at best allow the build to make sure the set of bundles fit the boot layer. This does not fix the reverse problem of building the boot layer to fit a set of bundles. To solve this problem the modules must be visible to the build tools. Without automated help the developers must perform the reverse discovery manually through trial and error. This makes the creation of self-contained Java applications hard and sometimes impractical. For example when a large set of small self-contained services has to be maintained or when it must be easy to merge and split bundles into multiple runtimes.

This RFP defines the requirements for encoding the JPMS modules as OSGi resources by mapping some of their metadata to OSGi metadata. Having this mapping will make the hidden dependencies bundles have on modules visible.

Standardizing the encoding should enable open collaboration between tools that handle modules and bundles at different stages in the software production pipeline. For example one tool may publish the module metadata while another may read it to build a Java runtime, while yet another may read it to present to a developer. Such tooling will enable developers to achieve the use cases described in this RFP.

4 Use Cases

4.1 Build a compact docker image

Systems that run their components in docker invariably need the container images to be as small as possible. The operational cost of the system increases with image size and performance such as startup time can be negatively impacted by image size. Java containers have to include a JRE of considerable size. Since the JRE is dedicated to running only one well defined Java application it is desirable to shrink it only to the parts needed to support that application.

4.2 Build a self-contained tool

Sometimes the user experience is affected by the ability to download and run a tool quickly when their environment is not setup with a JRE. The users of the tool must not be burdened with the need to first provide a JRE that is compatible with the tool. Examples of such cases are bootstrap tools like installers, diagnostic tools, tools for management of remote cloud resources.

4.3 Introspect the JRE of an OSGi application

In a case where dependency resolution is done against predefined JRE a developer needs to extract the dependencies between bundles and modules to gain insight into the “module footprint” of their bundles – the modules each bundle has dragged into the JRE.

4.4 Build a hybrid OSGi-JPMS application

A Java application may have some of it’s functionality implemented as bundles and another part implemented as modules. It should be possible to automatically resolve the complete Java runtime consisting of the required JRE modules, application modules, and bundles.

5 Requirements

The solution to this RFP has to fulfill the following requirements.

5.1 General

- G-1: **Must not** force bundles to explicitly reference modules in order for their requirements to resolve to modules. For example bundles may choose to, but must not be forced to, require a module by ID or include the module ID in the filter of another requirement.

- G-2: **Should** handle the case where a bundle is also a module. For example publish it as two separate resources pointing to the same artifact.

- G-3: **Should** handle the case of the OSGi framework, which is the only bundle that can also be a module **at runtime.** For example allow the framework to have both module and bundle metadata.

5.2 Resolving Dependencies

- RD-1: **Must** define how the identity capability of modules is derived from the module metadata. This should include best effort to map the module version to bundle version.

- RD-2: **Must** be able to resolve a bundle's package import to a package exported by a module.

- RD-3: **Must not** allow a bundle's package import to be resolved to a module's package export when that module exports the package only to a restricted set of other modules. For example such exports may be omitted from the module encoding into a resource.

- RD-4: **Must not** allow a module's requirement for another module to be resolved to a bundle.

- RD-5: **Must** be able to resolve a module's requirement to another module. For example this helps to understand the complete footprint of an application from the output of the Resolver alone since it will include not only the modules that are directly required by bundles, but also their transitive dependencies.

5.3 Module Services

- MS-1: **May** represent a module's provided services as a resource capabilities and a module's used services as resource requirements. This for example will enable developers to discover missing implementations they should include in their Java runtime.

- MS-2: **May resolve** a bundle requirement in the `osgi.serviceloader` namespace to a module service.

- MS-3: **Must not** resolve a module's service requirement to bundle capability in the `osgi.serviceloader` namespace should MS-2 be implemented.

5.4 Third Party Modules

- TPM-1: **Must** allow modules not provided by the JRE to participate in the resolution process as equal peers to the JRE modules.

- TPM-2: **Must** allow packages of modules which are not part of the JRE to be automatically exposed **at runtime** within the OSGi framework. For example without having to enumerate them beforehand in a configuration file.

5.5 Module Validation

- MV-1: **Must not** allow two modules with the same name to simultaneously participate in the resolution result.

- MV-2: **May not** allow for two modules to export the same package and be part of the resolution.

- MV-3: **May not** allow modules participating in the solution to form a dependency cycle.

5.6 Module Metadata Enhancement

- ME-1: **Should** allow module requirements for other modules to be enriched with version ranges. For example if the metadata of module `A` says it was build against module `B version V` this can map to a resource requirement with version range `[V, infinity)`.

- ME-2: **May** allow bundle annotations *[needs reference]* to be used on Java modules to handle dependencies that can not be captured by `module-info.java`

6 Document Support

6.1 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

Add references simply by adding new items. You can then cross-refer to them by chosing <Insert><Cross Reference><Numbered Item> and then selecting the paragraph. ***STATIC REFERENCES (I.E. BODGED) ARE NOT ACCEPTABLE, SOMEONE WILL HAVE TO UPDATE THEM LATER, SO DO IT PROPERLY NOW.***

6.2 Author's Address

Name	Todor Boev
Company	Software AG
Address	
Voice	
e-mail	todor.boev@softwareag.com

6.3 End of Document