Resource Encoding For Java Modles

Draft

15 Pages

Text in Red is here to help you. Delete it when you have followed the instructions.
The <RFC Title> can be set from the File>Properties:User Defined menu. To update it onscreen, press F9. To update all of the fields in the document Select All (CTRL-A), then hit F9. Set the release level by selecting one from: Draft, Final Draft, Release. The date is set automatically when the document is saved.

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 and to represent accurately the JRE within the OSGi runtime. This also means some application logic may come as JPMS modules not under the control of the OSGi developer. This RFC describes the requirements needed to represent the JPMS modules are OSGi Resources. This will make the OSGi technology applicable to them. One prominent use case is the application of the OSGi Resolver in the construction of complete OSGi plus JPMS runtimes.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

0.2 Trademarks

OSGi™ is a trademark, registered trademark, or service mark of the OSGi Alliance in the US and other countries. Java is a trademark, registered trademark, or service mark of Oracle Corporation in the US and other countries. All other trademarks, registered trademarks, or service marks used in this document are the property of their respective owners and are hereby recognized.

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 10.1.

`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
Initial	16 Sep 2019	Todor Boev, initial content copied from RFP

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 RFC introduces the requirements for a standard way to encode JPMS modules as OSGi Resources. Among other things this will enable the use of 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. At runtime the accurate representation of the JRE as Resources can enable new kinds of application logic such as the application of extenders to modules.

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 RFC defines the encoding of JPMS modules as OSGi Resources by mapping some of the JPMS module 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 RFC. Also making the JRE visible at runtime as a set of Resources enables new kinds of application logic that can introspect the modules and the connections they have to bundles and act accordingly

4 Requirements

4.1 General

- G-1: **Must not** preclude the use of the Resolver as the sole mechanism to build a consistent set of modules and bundles. Note that this does not mean the Resolver specification must not be extended.

- G-2: **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-3: **Should** handle the case where a bundle is also a module. For example publish it to a repository as two separate resources pointing to the same artifact.

- G-4: **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.

4.2 Resolving Dependencies

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

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

- D-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.

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

- D-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.

4.3 Module Services

- S-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.

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

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

4.4 Third Party Modules

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

4.5 Module Validation

- V-1: **Must not** allow two modules with the same name to simultaneously participate in the resolution result.
- V-2: **May not** allow for two modules to export the same package and be part of the resolution.
- V-3: **May not** allow modules participating in the solution to form a dependency cycle.

4.6 Module Metadata Enhancement

- E-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)`.
- E-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`

4.7 Modules at Runtime

- R-1: **Should** model the boot layer at runtime as Resources. In other words the `org.osgi.framework.system.packages` and `org.osgi.framework.system.packages.extra` should be removed or set to empty and their content be distributed between the Resources that represent the modules from the boot layer. Note that this does not require that the OSGi bundles are treated as modules.

4.8 Compatibility with Other Specifications

- C-1: **Must** agree with the technical solution of *RFC-243: Connect* *[needs reference]*
  - *NOTE: It will force the encoding to be that of Bundles, so maybe re-write all requirements with hits in mind?*
  - *NOTE: It will cover things like lifecycle operations on JMPS modules at runtime.*

5 Technical Solution

First give an architectural overview of the solution so the reader is gently introduced in the solution (Javadoc is not considered gently). What are the different modules? How do the modules relate? How do they interact? Where do they come from? This section should contain a class diagram. Then describe the different modules in detail. This should contain descriptions, Java code, UML class diagrams, state diagrams and interaction diagrams. This section should be sufficient to implement the solution assuming a skilled person.

Strictly use the terminology a defined in the Problem Context.

On each level, list the limitations of the solutions and any rationales for design decisions. Almost every decision is a trade off so explain what those trade offs are and why a specific trade off is made.

Address what security mechanisms are implemented and how they should be used.

```
public class JmodBundle {
    /** Extends the {@link IdentityNamespace} with a new type of artifact */
    private static final String TYPE_JAVA_MODULE = "java.module";

    /**
     * Extends the {@link BundleNamespace} to make it possible to resolve only
     * modules to modules
     */
    public static final String CAPABILITY_BUNDLE_TYPE_ATTRIBUTE = "bundle-type";

    /**
     * Extends the {@link PackageNamespace} to denote friend modules
     *
     * TODO Should be handled by the {@link ResolveContext} during request matching
     *
     * TODO Seems wrong to add such specifics to the {@link PackageNamespace}, but
     * on the other hand it has meaning at resolution time.
     */
    public static final String CAPABILITY_TARGETS_DIRECTIVE = "java-module-targets";

    /**
     * Extends the {@link PackageNamespace} to denote reflection vs static access
     *
     * TODO Does this make sense at resolution time? It directs the JVM reflection
     * checks during execution. Perhaps for introspection/informative purposes?
     *
     * TODO Seems wrong to add such specifics to the {@link PackageNamespace}
     */
    public static final String CAPABILITY_ACCESS_DIRECTIVE = "java-module-access";
    public static final String ACCESS_STATIC = "static";
    public static final String ACCESS_REFLECTION = "reflection";

    /**
     * From the Service Loader spec. Useful to expand the java module set at
     * resolution time.
     *
     * TODO Can a module use a "osgi.serviceloader" capability from a bundle? If not
     * these must be limited like require-bundle so a "type" attribute must be
```

```
     * introduced here as well?
     */
    public static final String SERVICELOADER_NAMESPACE = "osgi.serviceloader";

    private final ModuleDescriptor module;
    private final ResourceBuilder resource;
    private final String name;
    private final Version ver;

    public JmodBundle(ModuleDescriptor module) {
        this.module = module;
        this.resource = resource();

        this.name = module.name();
        this.ver = getOsgiVersion(module.rawVersion().orElse("0.0.0"));

        // Identity
        addIdentityCap();

        // Exported packages
        module.exports().forEach(this::addPackageWiringCap);
        module.opens().forEach(this::addOpenPackageWiringCap);

        // Required modules
        addBundleWiringCap();
        module.requires().forEach(this::addBundleWiringReq);

        // Services
        module.provides().forEach(this::addServiceLoaderCap);
        module.uses().forEach(this::addServiceLoaderReq);

        // descr.mainClass().ifPresent(main -> {
        // System.out.printf("\tmain %s\n", main);
        // });
    }

    public Resource getResource() {
        return resource.build();
    }

    /**
     *
     */
    private void addIdentityCap() {
        resource.add(capability(IdentityNamespace.IDENTITY_NAMESPACE)
                .attribute(IdentityNamespace.IDENTITY_NAMESPACE, name)
                .attribute(IdentityNamespace.CAPABILITY_VERSION_ATTRIBUTE, ver)
                .attribute(IdentityNamespace.CAPABILITY_TYPE_ATTRIBUTE, TYPE_JAVA_MODULE)
                .attribute(IdentityNamespace.CAPABILITY_SINGLETON_DIRECTIVE, "true"));
    }

    /**
     *
     */
    private void addBundleWiringCap() {
        resource.add(capability(BundleNamespace.BUNDLE_NAMESPACE)
                .attribute(BundleNamespace.BUNDLE_NAMESPACE, name)
                .attribute(BundleNamespace.CAPABILITY_BUNDLE_VERSION_ATTRIBUTE, ver)
                .attribute(CAPABILITY_BUNDLE_TYPE_ATTRIBUTE, TYPE_JAVA_MODULE));
```



```
}

/**
 *
 * @param exp
 */
private void addPackageWiringCap(Exports exp) {
    String pack = exp.source();
    List<String> to = new ArrayList<>(exp.targets());

    CapabilityBuilder capability = capability(PackageNamespace.PACKAGE_NAMESPACE)
            .attribute(PackageNamespace.PACKAGE_NAMESPACE, pack)
            .attribute(PackageNamespace.CAPABILITY_VERSION_ATTRIBUTE, new Version("0.0.0"))
            .attribute(PackageNamespace.CAPABILITY_BUNDLE_SYMBOLICNAME_ATTRIBUTE, name)
            .attribute(PackageNamespace.CAPABILITY_BUNDLE_VERSION_ATTRIBUTE, ver);
    if (!to.isEmpty()) {
        capability.attribute(CAPABILITY_TARGETS_DIRECTIVE, to);
    }

    resource.add(capability);
}

/**
 *
 * @param exp
 */
private void addOpenPackageWiringCap(Opens exp) {
    String pack = exp.source();
    List<String> to = new ArrayList<>(exp.targets());

    CapabilityBuilder capability = capability(PackageNamespace.PACKAGE_NAMESPACE)
            .attribute(PackageNamespace.PACKAGE_NAMESPACE, pack)
            .attribute(PackageNamespace.CAPABILITY_VERSION_ATTRIBUTE, new Version("0.0.0"))
            .attribute(PackageNamespace.CAPABILITY_BUNDLE_SYMBOLICNAME_ATTRIBUTE, name)
            .attribute(PackageNamespace.CAPABILITY_BUNDLE_VERSION_ATTRIBUTE, ver);
    if (!to.isEmpty()) {
        capability.attribute(CAPABILITY_TARGETS_DIRECTIVE, to);
    }
    capability.directive(CAPABILITY_ACCESS_DIRECTIVE, ACCESS_REFLECTION);

    resource.add(capability);
}

/**
 *
 * @param req
 */
private void addBundleWiringReq(Requires req) {
    String reqName = req.name();
    Version reqVer = getOsgiVersion(module.rawVersion().orElse("0.0.0"));
    // TODO Is the OSGi re-export a good model for the JPMS transitive?
    boolean transitive = req.modifiers().contains(Requires.Modifier.TRANSITIVE);
    // TODO "static" in JPMS means "required at compile-time, optional at runtime" -
    // so which one should we use?
    boolean optional = req.modifiers().contains(Requires.Modifier.STATIC);

    RequirementBuilder requirement = requirement(BundleNamespace.BUNDLE_NAMESPACE);
    requirement.directive(REQUIREMENT_FILTER_DIRECTIVE,
            "(&" +
```

```
                "(" + BundleNamespace.BUNDLE_NAMESPACE + "=" + reqName + ")" +
                // TODO Or an inferred version range?
                "(" + BundleNamespace.CAPABILITY_BUNDLE_VERSION_ATTRIBUTE + "=" + reqVer + ")" +
                "(" + CAPABILITY_BUNDLE_TYPE_ATTRIBUTE + "=" + TYPE_JAVA_MODULE + ")" +
                ")");
        requirement.directive(REQUIREMENT_RESOLUTION_DIRECTIVE,
                optional ? RESOLUTION_OPTIONAL : RESOLUTION_MANDATORY);
        if (transitive) {
            requirement.directive(
                    BundleNamespace.REQUIREMENT_VISIBILITY_DIRECTIVE,
BundleNamespace.VISIBILITY_REEXPORT);
        }

        resource.add(requirement);
    }

    /**
     *
     * @param provided
     */
    private void addServiceLoaderCap(Provides provided) {
        String service = provided.service();
        // TODO Handle internal classes. E.g. stop at $
        String pack = service.substring(0, service.lastIndexOf('.') - 1);

        resource.add(capability(SERVICELOADER_NAMESPACE, service)
                .directive(CAPABILITY_USES_DIRECTIVE, pack));
    }

    /**
     *
     * @param used
     */
    private void addServiceLoaderReq(String used) {
        resource.add(requirement(SERVICELOADER_NAMESPACE, used)
                .directive(REQUIREMENT_RESOLUTION_DIRECTIVE, RESOLUTION_OPTIONAL)
                // TODO Remove this?
                .directive(REQUIREMENT_CARDINALITY_DIRECTIVE, CARDINALITY_MULTIPLE));
    }

    /**
     *
     * @param jmodVersion
     * @return
     */
    private static Version getOsgiVersion(String jmodVersion) {
        Pattern number = Pattern.compile("(\\d+)\\.?");
        Matcher match = number.matcher(jmodVersion);

        int major = Integer.parseInt(matchOrDefault(match, 1, "0"));
        int minor = Integer.parseInt(matchOrDefault(match, 1, "0"));
        int micro = Integer.parseInt(matchOrDefault(match, 1, "0"));

        return new Version(major, minor, micro);
    }

    /**
     *
     * @param match
```



```
     * @param group
     * @param def
     * @return
     */
    private static String matchOrDefault(Matcher match, int group, String def) {
        return match.find() ? match.group(group) : def;
    }
}
```

6 Data Transfer Objects

RFC 185 defines Data Transfer Objects as a generic means for management solutions to interact with runtime entities in an OSGi Framework. DTOs provides a common, easily serializable representation of the technology.

For all new functionality added to the OSGi Framework the question should be asked: would this feature benefit from a DTO? The expectation is that in most cases it would.

The DTOs for the design in this RFC should be described here and if there are no DTOs being defined an explanation should be given explaining why this is not applicable in this case.

This section is optional and could also be provided in a separate RFC.

7 Javadoc

Please include Javadoc of any new APIs here, once the design has matured. Instructions on how to export Javadoc for inclusion in the RFC can be found here: https://www.osgi.org/members/RFC/Javadoc

8 Considered Alternatives

For posterity, record the design alternatives that were considered but rejected along with the reason for rejection. This is especially important for external/earlier solutions that were deemed not applicable.

9 Security Considerations

Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.

10 Document Support

10.1 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.***

10.2 Author's Address

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

10.3 Acronyms and Abbreviations

10.4 End of Document

