RFP-173 JAX-RS Services

Draft

10 Pages

Abstract

10 point Arial Centered.

The RESTful service model has existed for several years as a simple means of providing CRUD (Create, Read, Update, Delete) style services using existing HTTP standards, request types, and parameter passing. As REST services grew in popularity they were adopted into Java EE as the JAX-RS standard. This standard was designed to be standalone, with minimal dependencies on other Java EE specifications, and to provide a simple way to expose HTTP REST services producing JSON, XML, plain text or other response types, without resorting to a servlet container model. This RFP aims to enable JAX-RS components and applications as first-class OSGi citizens, making it easy to write RESTful services in a familiar way, whilst simultaneously having access to the benefits of the modular, service-based OSGi runtime.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

The OSGi Alliance hereby grants you a limited copyright license to copy and display this document (the “Distribution”) in any medium without fee or royalty. This Distribution license is exclusively for the purpose of reviewing and providing feedback to the OSGi Alliance. You agree not to modify the Distribution in any way and further agree to not participate in any way in the making of derivative works thereof, other than as a necessary result of reviewing and providing feedback to the Distribution. You also agree to cause this notice, along with the accompanying consent, to be included on all copies (or portions thereof) of the Distribution. The OSGi Alliance also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights, to create and/or distribute an implementation of the Distribution that: (i) fully implements the Distribution including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the OSGi Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the OSGi Name Space other than those required and authorized by the Distribution. An implementation that does not satisfy limitations (i)-(ii) is not considered an implementation of the Distribution, does not receive the benefits of this license, and must not be described as an implementation of the Distribution. "OSGi Name Space" shall mean the public class or interface declarations whose names begin with "org.osgi" or any recognized successors or replacements thereof. The OSGi Alliance expressly reserves all rights not granted pursuant to these limited copyright licenses including termination of the license at will at any time.

EXCEPT FOR THE LIMITED COPYRIGHT LICENSES GRANTED ABOVE, THE OSGi ALLIANCE DOES NOT GRANT, EITHER EXPRESSLY OR IMPLIEDLY, A LICENSE TO ANY INTELLECTUAL PROPERTY IT, OR ANY THIRD PARTIES, OWN OR CONTROL. Title to the copyright in the Distribution will at all times remain with the OSGi Alliance. The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted therein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

THE DISTRIBUTION IS PROVIDED "AS IS," AND THE OSGi ALLIANCE (INCLUDING ANY THIRD PARTIES THAT HAVE CONTRIBUTED TO THE DISTRIBUTION) MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DISTRIBUTION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
NEITHER THE OSGi ALLIANCE NOR ANY THIRD PARTY WILL BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE DISTRIBUTION.

Implementation of certain elements of this Distribution may be subject to third party intellectual property rights, including without limitation, patent rights (such a third party may or may not be a member of the OSGi Alliance). The OSGi Alliance is not responsible and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

The Distribution is a draft. As a result, the final product may change substantially by the time of final publication, and you are cautioned against relying on the content of this Distribution. You are encouraged to update any implementation of the Distribution if and when such Distribution becomes a final specification.

The OSGi Alliance is willing to receive input, suggestions and other feedback (“Feedback”) on the Distribution. By providing such Feedback to the OSGi Alliance, you grant to the OSGi Alliance and all its Members a non-exclusive, non-transferable,

worldwide, perpetual, irrevocable, royalty-free copyright license to copy, publish, license, modify, sublicense or otherwise distribute and exploit your Feedback for any purpose. Likewise, if incorporation of your Feedback would cause an implementation of the Distribution, including as it may be modified, amended, or published at any point in the future ("Future Specification"), to necessarily infringe a patent or patent application that you own or control, you hereby commit to grant to all implementers of such Distribution or Future Specification an irrevocable, worldwide, sublicenseable, royalty free license under such patent or patent application to make, have made, use, sell, offer for sale, import and export products or services that implement such Distribution or Future Specification. You warrant that (a) to the best of your knowledge you have the right to provide this Feedback, and if you are providing Feedback on behalf of a company, you have the rights to provide Feedback on behalf of your company; (b) the Feedback is not confidential to you and does not violate the copyright or trade secret interests of another; and (c) to the best of your knowledge, use of the Feedback would not cause an implementation of the Distribution or a Future Specification to necessarily infringe any third-party patent or patent application known to you. You also acknowledge that the OSGi Alliance is not required to incorporate your Feedback into any version of the Distribution or a Future Specification.

I HEREBY ACKNOWLEDGE AND AGREE TO THE TERMS AND CONDITIONS DELINEATED ABOVE.

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 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	Jan 19 2015	Tim Ward (tim.ward@paremus.com) First draft of the JAX-RS whiteboard.
0.1	May 20 2015	Updates following the Cologne F2F
0.2	July 02 2015	Updates following the Chicago F2F

1 Introduction

Over the last decade there has been a significant shift in the way that many computer programs are written. The focus has changed from building larger, more monolithic applications that provide a single high-level function, to composing these high-level behaviours from groups of smaller, distributed services. This is generally known as a "microservice" architecture, indicating that the services are smaller and lighter weight than typical web servies.

Many of these microservices are used to provide access to data from a data store. This may be a traditional relational database, or it may use some other mechanism, such as a Document store, or a key-value store. These sorts of service frequently offer a limited set of operations that fit the CRUD (Create, Read, Update, Delete) model, and produce a representation of the data in a simple text-based format, which may be XML, JSON, or plain text. By using the various methods defined in the HTTP 1.1 specification [3]. it is relatively simple to map these operations into standard HTTP requests. As native HTTP support is widely available across programming languages, and also because almost all client systems are equipped with a web browser, HTTP is the obvious choice for accessing these services. Implementing services in this way has become such a common pattern that it is now seen as distinct from “Web Services” and instead these services are known as REST (Representational State Transfer) or RESTful services.

REST services in Java can be implemented in many ways. Simple services can be implemented relatively easily using Servlets, but there are numerous frameworks, such as Jersey, Restlet, and CXF which provide their own APIs for implementing RESTful services. The ideas from these frameworks were then used in the JCP to produce a standard for REST in Java, known as JAX-RS.

JAX-RS provides a simple annotation-based model in which POJOs can have their methods mapped to RESTful service invocations. There is automatic mapping of HTTP parameters, and of the HTTP response, based on the annotations, and the incoming HTTP Headers. JAX-RS also includes support for grouping these POJOs into a single Application artifact. This allows the POJOs to interact with one another, as well as to share configuration and runtime state.

Ideal JAX-RS services are stateless, and are usually instantiated by the container from a supplied class name or `Class` object. The use of class names is obviously a problem in OSGi, but otherwise JAX-RS services share many features with OSGi services. In that they provide a way for machines (or processes within a machine) to interact with one another through a defined contract. It would be advantageous to allow OSGi services to be directly exposed as JAX-RS beans, and to support the use of JAX-RS services within an OSGi framework without resorting to an HTTP call.

2 Application Domain

JAX-RS is a well-known standard for building RESTful services, and a number of popular open source implementations exist. JAX-RS applications all make use of annotations and/or XML for configuration but can bootstrap themselves in a variety of different ways.

2.1 Bootstrapping in a WAR file

In a WAR file there can be a web.xml descriptor that is used to configure the application, or (from servlet 3.0) annotation-scanning can be used to locate items

2.1.1 Servlet 3.0 ServletContextInitializer

In an annotation scanning Servlet Container the JAX-RS implementation provides an annotated ServletContextInitializer, which is called back when the web application starts. This callback is used to scan the application for JAX-RS managed beans, and to register the JAX-RS container with the servlet container.

2.1.2 Custom JAX-RS Application classes

It is possible to customise the `javax.ws.rs.core.Application` used to represent the set of JAX-RS beans in the application. This is supplied as a servlet initialization parameter, providing the name of the custom subclass which will be instantiated by the JAX-RS container.

2.2 Bootstrapping in Java SE

Most JAX-RS libraries provide their own HTTP server implementations for use in Java SE. These require implementation specific code to bootstrap the server, and can then be supplied with individual JAX-RS beans, or a JAX-RS application. Usually this requires the bean or application to be wrapped in an implementation-specific type.

2.3 Bootstrapping in OSGi

Most of the popular JAX-RS frameworks describe how to run JAX-RS applications deployed in an OSGi framework. Most of the static configuration options for JAX-RS do not work well in OSGi as they exchange String class names.

2.3.1 Deployment as a WAB

The simplest way to deploy JAX-RS applications in OSGi is to package them in a WAB. WABs run in the same way as WAR files do in a standard Servlet Container, and therefore the JAX-RS implementations work as if they were in a non-OSGi environment. Note that this model either requires the JAX-RS runtime to be packaged inside the WAR file, or for the Thread Context ClassLoader to be set to the WAB ClassLoader on initialization.

2.3.2 Deployment using the HttpService

Most JAX-RS frameworks offer an implementation-specific “wrapper servlet” which adapts the Servlet API into the JAX-RS API, and delegates to the JAX-RS beans. This wrapper servlet can be configured in code and registered with the Http Service.

2.4 Runtime behaviour

Once the JAX-RS container has bootstrapped, the container has located the various JAX-RS beans and validated any declared metadata and injection sites. Incoming HTTP requests are routed to the beans based on this metadata, and behaviour is unaffected by the underlying container. This means that at runtime JAX-RS behaves the same way in Java EE, Java SE and OSGi.

2.5 Locating JAX-RS endpoints

Once a JAX-RS application has been started then the HTTP endpoint is available for use. In order for clients to be able to use this endpoint they must be notified of where it is. In general there is no standard way to discover this information, however a number of approaches can be used.

- Static configuration – Typically this is achieved using a properties file which statically defines the URI. The URI must be manually updated everywhere if the service is ever moved to a different host or path.
- Central registry – This may be static (i.e. a fixed list) or dynamic (i.e. the application registers itself). The client contacts a central registry, and queries for the location of the JAX-RS endpoint. The registry returns the location for the client to use.
- Dynamic discovery – A configuration discovery layer (e.g. ZeroConf, mDNS etc) can be used to dynamically discover local endpoints.

The approaches above tie in very closely with the mechanisms available to OSGi's Remote Service Admin. Static endpoint information is available using the Endpoint XML extender, whereas dynamic discovery may use a central registry such as ZooKeeper, or a peer-to-peer discovery mechanism such as Bonjour. For OSGi environments it should be possible reuse RSA discovery, although there must not be a hard requirement on the presence of an RSA discovery provider for JAX-RS services to be hosted

2.6 Terminology + Abbreviations

3 Problem Description

As described in section 2.4 there is very little difference in behaviour between JAX-RS applications once they have been successfully bootstrapped. The bootstrapping process is, however, different in different environments.

In OSGi there are particular deployment problems where String class names are passed to the JAX-RS container, which is why WABs require special treatment. If the JAX-RS runtime is packaged into the WAB then the JAX-RS runtime cannot be changed easily, nor can that runtime be reused by other JAX-RS applications. When the HttpService is used there is a similar coupling to the JAX-RS implementation because an implementation-specific servlet must be created.

This RFP aims to address this issue by providing a loosely coupled, provider-independent mechanism for hosting JAX-RS applications and beans. This should fit with the modular, dynamic nature of the OSGi runtime. In addition, once the JAX-RS application has been registered it should be easy to identify the URI of the JAX-RS endpoint that has been created. Dynamic discovery in remote nodes must also be possible so that other OSGi containers can interact with the service.

Another issue encountered by many users of Java EE specifications in OSGi is that the versions of the specifications do not typically follow semantic versioning rules. JAX-RS is no different, and has two currently published versions JAX-RS 1.0[4]., and JAX-RS 2.0[5].. JAX-RS 2.0 is backward compatible with JAX-RS 1.0, but is exported using a higher major version. This problem is typically solved in OSGi using Portable Java Contracts. The JavaJAXRS contracts defined at [6]. can be used by clients to avoid version matching issues, and so any JAX-RS code in OSGi make use of them.

4 Use Cases

4.1 Single Resource

As an OSGi developer I wish to expose a single JAX-RS resource from my bundle in a provider-agnostic way. The JAX-RS resource depends on an OSGi service, and so must be able to be dynamically registered and unregistered from the JAX-RS container.

4.2 JAX-RS Application

As an OSGi developer I wish to expose a group of JAX-RS resources and providers packaged as a JAX-RS application. Several of the resources depend on configuration provided by Configuration Admin, and so need to be dynamically re-registered if the configuration changes.

4.3 Client Usage

As an OSGi developer I wish to discover and consume a JAX-RS service deployed in an OSGi framework so that I can call it using the JAX-RS client API. This requires a root URI for the JAX-RS resource endpoint. In the general case this endpoint may be in a separate OSGi framework.

4.4 Version usage

As an OSGi developer I wish to use the portable Java contract for JAX-RS so that my version 1.0 JAX-RS service can use the maximum possible number of provider versions, despite the lack of semantic versioning in the API.

5 Requirements

RS010 – The solution MUST provide a JAX-RS container independent mechanism for dynamically registering and unregistering an individual JAX-RS Resource

RS020 – The solution MUST provide a JAX-RS container independent mechanism for dynamically registering and unregistering a `javax.ws.rs.core.Application` with the container.

RS030 – The solution MUST provide a mechanism for locally discovering the URI at which the JAX-RS resource or application has become available. ~~and t~~This mechanism SHOULD be suitable for discovery in remote frameworks. Remote Discovery MAY require the use of Remote Service Admin, or some other OSGi specification.

RS050 – The solution SHOULD require implementations to provide a suitable contract capability so that clients can use backward compatible implementations that provide a higher version of the API.

RS060 – The solution SHOULD NOT require that the implementation use the HttpService or Http Whiteboard to provide a HTTP endpoint.

RS070 – The solution MUST NOT require the standardisation of another dependency injection container. JAX-RS services should be able to be provided as Declarative Service components, Blueprint beans or any other existing mechanism.

RS080 – The solution MUST NOT prevent the JAX-RS container from performing method parameter injection, for example an AsyncResponse object

RS090 – The solution MUST NOT prevent the JAX-RS container from injecting “Context” objects into fields or setters of the JAX-RS service, for example a `javax.ws.rs.core.Application` object.

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

[3]. HTTP 1.1 Specification RFC 2626 - http://www.w3.org/Protocols/rfc2616/rfc2616.html

[4]. JAX-RS 1.0 Specification - https://jcp.org/aboutJava/communityprocess/final/jsr311/index.html

[5]. JAX-RS 2.0 Specification - https://jcp.org/aboutJava/communityprocess/final/jsr339/index.html

[6]. OSGi Portable Java Contracts - http://www.osgi.org/Specifications/ReferenceContract

6.2 Author’s Address

Name	Tim Ward
Company	Paremus
Address	
Voice	
e-mail	tim.ward@paremus.com

6.3 End of Document