RFC-217 JAX-RS Services

Draft

16 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

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	13 Nov 2015	First draft of JAX-RS Services

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 Error: Reference source not found 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 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. 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.

5 Technical Solution

5.1 JAX-RS Service Endpoints

JAX RS beans (also known as endpoints or services) are objects that are bound to a particular URI and used to service HTTP requests. The JAX-RS beans will return/update/delete data as a result of the request. In effect a JAX-RS bean behaves a lot like a Servlet, but with additional mapping of request data to methods/method parameters.

Due to the similarity in behaviour between JAX-RS beans and Servlets this RFC proposes to reuse the whiteboard model defined by the Http Whiteboard defined in chapter 140 of the OSGi Compendium Specification [7]..

5.1.1 Bean mapping

To contribute a JAX-RS service to the JAX-RS container a bundle must register the JAX-RS bean as an OSGi service. Furthermore the bundle must register the service with the `osgi.jaxrs.bean.base` property. This property has two purposes:

1. It serves as a marker to the JAX-RS whiteboard runtime that this service should be hosted as an endpoint

2. The value of the property defines a base request URI path to which this bean should be mapped. This value is prepended to the path defined by the JAX-RS bean. Values follow the same syntax rules as the JAX-RS path annotation, but do not permit parameter templates.

When mapping servlets the URI path is either a fixed value or a glob wildcard pattern. If the request URI matches the pattern then the service method of the servlet is called. For JAX-RS beans the behaviour is different. The URI

path defined is not a wildcard, but a defined set of path segments. Some of the path segments may be “variable” and used as method parameters.

For example the following bean maps HTTP GET requests for the path “foo” to the getFoo() method:

```
@Path(“foo”)
public class Foo {
    @GET
    public String  getFoo() {
        return “foo”;
    }
}
```

This bean maps HTTP GET requests for the path “foo/xxx” to the getFoo(String) method, taking the next URI segment as a paremeter:

```
@Path(“foo/{name}”)
public class Foo {
    @GET
    public String  getFoo(@PathParam(“name”) String name) {
        return “foo” + name;
    }
}
```

This bean maps HTTP GET requests for the path “foo/xxx” to the getFoo(String) method, taking the next URI segment as a paremeter, but only if the next segment matches the supplied regex:

```
@Path(“foo/{name: [a-zA-Z]+}”)
public class Foo {
    @GET
    public String  getFoo(@PathParam(“name”) String name) {
        return “foo” + name;
    }
}
```

In addition to defining a path, a JAX-RS bean may define “sub-resources” with child paths.

This bean maps HTTP GET requests for the path “foo” to the getFoos() method, which returns the list of known foos. The sub-resource method, getFoo(String) is called if the next URI segment matches the supplied regex:

```
@Path(“foo”)
public class Foo {
    @GET
    public List<String>  getFoos() {
        return Arrays.asList(“foo”, “bar”, “baz”);
```

```
    }

    @GET
    @PathParam(“{name: [a-zA-Z]+}”)
    public String  getFoo(@PathParam(“name”) String name) {
        String fooInfo = getFooInfo(name);
        return fooInfo;
    }

}
```

As a result of the way in which URI paths are mapped by JAX-RS beans, one bean may map several URI paths, but each path is limited to a fixed number of URI segments.

In the following example the bean is registered with a `osgi.jaxrs.bean.base` property value of “fizzbuzz”

5.1.1.1 Path matching example

```
@Path(“foo”)
public class Foo {
    @GET
    public List<String>  getFoos() {
        return Arrays.asList(“foo”, “bar”, “baz”);
    }

    @GET
    @PathParam(“{name: [a-zA-Z]+}”)
    public String  getFoo(@PathParam(“name”) String name) {
        return “foo” + name;
    }

}
```

This JAX-RS bean will be mapped to a base URI of “fizzbuzz” with a resource URI of “foo” and a single capturing URI segment.

URI path	Result
/	No match (404)
/foo	No match (404)
/fizzbuzz	No match (404)
/fizzbuzz/foo	“[foo, bar, baz]”
/fizzbuzz/foo/bar	“foobar”
/fizzbuzz/foo/baz	“foobaz”
/fizzbuzz/foo/bar/baz	No match (404)

5.1.2 Container base context paths

This RFC does not define how the JAX-RS container is registered to process HTTP requests. The container may be directly listening on a port and processing requests, or it may be registered as a Servlet in an HTTP container, or registered in some other way. In all of these cases there may be a base URI required to access the container. This base URI must be used in addition to any base URI defined in service properties or the JAX-RS bean when accessing the resource.

For example, if the JAX-RS container is registered as a whiteboard servlet then it may have a root URI of “rest”. In this case the resource defined in example 5.1.1.1 would be available at /rest/fizzbuzz/foo.

5.1.3 Container resource injection

The JAX-RS container may inject container resources, either into fields or method parameters, using the `@Context` annotation. The JAX-RS container must honour these injection sites if they are present in the instances provided by the OSGi service.

5.2 Filter and Interceptor mapping

In addition to JAX-RS beans, a JAX RS application may define filters and interceptors. JAX-RS filters are used to alter request and response parameters, whereas JAX-RS interceptors are used to alter the incoming or outgoing request entities.

Typical JAX-RS filters and interceptors match all resources beneath their base URI, which means that they implictly behave as if they are wildcard matchers. Filters and Interceptors can be registered with the JAX-RS container by registering them as OSGi services with mapping properties. The property names are `osgi.jaxrs.filter.base` and `osgi.jaxrs.interceptor.base` respectively. These base URIs are written in the same way as for JAX-RS beans, however rather than matching an exact number of URI segments they match all sub-paths, regardless of length.

5.2.1 Named Filters and Interceptors

Sometimes filters and interceptors should not be run for all resources in the container. In this case custom annotations can be used to link together particular resource methods and the relevant filters/interceptors. If the Filter or Interceptor is annotated with one of these custom annotations then it will only be run when the resource method is also annotated with the same custom annotation.

The JAX-RS whiteboard container must honour the behaviour of the Named Filter and Interceptors, running them as appropriate.

5.3 Service Lifecycle

5.3.1 Registering and Unregistering JAX-RS beans

As this RFC uses a whiteboard model it is clear how the JAX-RS registration lifecycle should function. JAX-RS beans are registered with the JAX-RS container automatically as the OSGi service is registered, and they are unregistered from the container when the service is unregistered.

If a registered JAX-RS bean's OSGi service is updated such that its properties make it ineligible for registration (for example deletion of the base mapping property) then the JAX-RS bean must be unregistered. Similarly any change to the properties that adds or alters a JAX-RS bean mapping must be dynamically reflected in the JAX-RS container mappings.

5.3.2 JAX-RS bean lifecycle

JAX-RS beans, filters and interceptors are intended to be stateless objects, and are typically instantiated from a `Class` instance dynamically by the container on a per-request basis. This model ensures that no state persists

between requests, and that there is no risk of concurrent access. JAX-RS objects may, however, be singletons. In this case the same object is used for every request.

In OSGi a programmer will typically want to refer to OSGi services from within their bean, filter or interceptor. This means that reflective instantiation from a class is not appropriate. This specification therefore defines the `osgi.jaxrs.scope` property. This property determines the lifecycle scope of the JAX-RS object. Valid values for the property are "`REQUEST`" and "`SINGLETON`". The value of this property interacts with the scope of the OSGi service to determine the lifecycle of the bean

Service scope	**`osgi.jaxrs.scope`**	**JAX-RS behaviour**
singleton/bundle	Unset or SINGLETON	Singleton bean looked up and used for all requests
singleton/bundle	REQUEST	A configuration error. No bean registered
prototype	SINGLETON	Singleton bean looked up and used for all requests
prototype	Unset or REQUEST	A request scoped bean. A new service instance requested for each request, and released after the request.

5.4 JAX-RS Applications

Sometimes a JAX-RS application consists of a number of related resources which must collaborate with one another. In this case it is normal to wrap the resources inside an `Application` object.

JAX-RS Applications may also be registered with the container by using the JAX-RS whiteboard. In this case the relevant service property is `osgi.jaxrs.application.base`

Once an Application is registered then all of its contained resources will be registered as children of its base URI. Effectively the application object behaves as a group registration for a set of JAX-RS objects.

5.5 JAX-RS endpoint advertisement

All JAX-RS beans and applications may be registered with an optional `osgi.jaxrs.name` property. If the registered service has this property then the JAX-RS container must register an `Endpoint` service identifying the URI that can be used to access the service.

The endpoint service must declare the following properties:

Name	Value
`osgi.jaxrs.name`	The name of the JAX-RS bean or application that has been registered
`osgi.jaxrs.uri`	The URI that can be used to access the JAX-RS resources
`service.exported.interfaces`	Set to export this endpoint for discovery in other remote frameworks
`osgi.jaxrs.bundle.symbolicname`	Set to the symbolic name of the bundle that provided the JAX-RS service
`osgi.jaxrs.bundle.id`	Set to the id of the bundle that provided the JAX-RS service

`osgi.jaxrs.bundle.version`	Set to the version of the bundle that provided the JAX-RS service
`osgi.jaxrs.service.id`	Set to the id of the JAX-RS service that this endpoint represents

5.6 Error Handling

There are a number of error cases where the JAX-RS container may be unable to correctly register a resource

5.6.1 Failure to obtain a service instance

In the case where a published service is unable to obtained by the JAX-RS container then the object is blacklisted by the container. A failure DTO is available from the JAXRSServiceRuntime representing the blacklisted service object.

5.6.2 Invalid service objects

Certain JAX-RS objects are required to implement certain interfaces, or to extend certain types. If a service advertises itself using a Jax-RS service property,but fails to implement the relevant JAX-RS type then this is an error and the service must be blacklisted by the container. A failure DTO is available from the JAXRSServiceRuntime representing the blacklisted service object.

5.6.3 Overlapping resource mappings

When multiple bundles collaborate it is possible that two JAX-RS beans will register for the same path. In this case the lower ranked service object is shadowed, and a failure DTO is available from the JAXRSServiceRuntime representing the shadowed object.

Note that determining when two JAX-RS endpoints overlap requires an analysis of the resource path and all of sub-resource paths. If **any** of these paths clash then the **entire** of the lower-ranked JAX-RS bean must be unregistered and marked as a failure. It is a container error for some sub-resource paths to be available while others are shadowed.

5.6.4 Class-Space Compatibility

Much of the JAX-RS mapping definition is handled using annotations with runtime visibility. As JAX-RS beans are POJOs there is no guarantee of class-space compatibility when the JAX-RS container searches for whiteboard services. The JAX-RS container must therefore confirm that the registered service shares the correct view of the JAX-RS packages. If the class space is not consistent then the JAX-RS whiteboard container must not register the services, but instead should create a failure DTO indiecating that the JAX-RS object is unable to be registered due to an incompatible class-space.

5.7 JaxRSServiceRuntime

TODO an equivalent of the HttpServiceRuntime + DTOs

6 Data Transfer Objects

This RFC defines an API to retrieve administrative information from the JAX-RS Whiteboard Service implementation. The JaxRSServiceRuntime service is introduced and can be called to obtain various DTOs.

The DTOs for the various services contain the field serviceId. In the case of whiteboard services this value is the value of the service.id property of the corresponding service registration. In the case of a clash, e.g. two services registered with the same path, only the service with the highest ranking is used. The service(s) with the lower ranking(s) are unused. The JAX-RS Service Runtime provides DTOs for those unused services as well as failures when using a service, for example like an exception thrown when obtaining the service, in order to find setup problems.

TODO define the DTOs and Runtime service – should be very similar to the Http Whiteboard

7 Javadoc

TODO add JavaDoc once the DTOs are defined

8 Considered Alternatives

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

[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

[7]. OSGi Compendium R6 - https://osgi.org/download/r6/osgi.cmpn-6.0.0.pdf

10.2 Author's Address

Name	
Company	
Address	
Voice	
e-mail	

10.3 Acronyms and Abbreviations

10.4 End of Document