



RFC-217 JAX-RS Services

Draft

46 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 Document Information.....	2
0.1 License.....	2
0.2 Trademarks.....	3
0.3 Feedback.....	3
0.4 Table of Contents.....	3
0.5 Terminology and Document Conventions.....	4
0.6 Revision History.....	4
1 Introduction.....	5
2 Application Domain.....	6
2.1 Bootstrapping in a WAR file.....	6
2.1.1 Servlet 3.0 ServletContextInitializer.....	6
2.1.2 Custom JAX-RS Application classes.....	6
2.2 Bootstrapping in Java SE.....	6
2.3 Bootstrapping in OSGi.....	6
2.3.1 Deployment as a WAB.....	6
2.3.2 Deployment using the HttpService.....	6
2.4 Runtime behaviour.....	7
2.5 Locating JAX-RS endpoints.....	7
2.6 Terminology + Abbreviations.....	7
3 Problem Description.....	7
4 Requirements.....	8

5 Technical Solution.....	9
5.1 JAX-RS ServiceResource Endpoints.....	9
5.1.1 BeanResource mapping.....	9
5.1.2 Container base context paths.....	11
5.1.3 Container resource injection.....	12
5.2 Filter and Interceptor mapping.....	12
5.2.1 Named Filters and Interceptors.....	12
5.3 Service Lifecycle.....	12
5.3.1 Registering and Unregistering JAX-RS beansresources.....	12
5.3.2 JAX-RS bean lifecycle.....	12
5.4 JAX-RS Applications.....	13
5.5 JAX-RS endpoint advertisement.....	13
5.6 Error Handling.....	14
5.6.1 Failure to obtain a service instance.....	14
5.6.2 Invalid service objects.....	14
5.6.3 Overlapping resource mappings.....	14
5.6.4 Class-Space Compatibility.....	14
5.7 JaxRSServiceRuntime.....	14
5.7.1 JAX-RS runtime properties.....	14
5.7.2 JAX-RS whiteboard configuration.....	15
5.7.3 Runtime Introspection and DTOs.....	15
5.8 JAX-RS Clients.....	15
5.8.1 Locating specific implementations.....	15
5.8.2 Asynchronous Calls.....	15
5.9 Implementation Provided Capabilities.....	16
6 Data Transfer Objects.....	17
6.1 The Runtime DTO.....	17
6.2 The RequestInfo DTO.....	17
7 Javadoc.....	17
8 Considered Alternatives.....	41
9 Security Considerations.....	42
10 Document Support.....	42
10.1 References.....	42
10.2 Author's Address.....	42
10.3 Acronyms and Abbreviations.....	42
10.4 End of Document.....	42

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	13 Nov 2015	First draft of JAX-RS Services
0.1	18 Dec 2015	Updates after the Chicago F2F. Add a client API, and fill in the DTO section
0.2	29 Nov 2016	Add support for MessageBodyReader/MessageBodyWriter, clarify processing ordering, Add JaxRsContext
0.3	19 Jan 2017	Update support for JAX-RS extensions based on Feedback and bugs
0.4	26 Jan 2017	Updates from the Portland 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 services.

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. When used in JAX-RS these POJOs are known as JAX-RS resources.

Ideal JAX-RS resources 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 resources 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 using JAX-RS, and to support the use of JAX-RS resources within an OSGi framework without resorting to an HTTP call.

2 Application Domain

JAX-RS is a well-known standard for building RESTful web 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 resources, 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 resources 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 resources, or a JAX-RS application. Usually this requires the resource 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 `HttpService`.

2.4 Runtime behaviour

Once the JAX-RS container has bootstrapped, the container has located the various JAX-RS resources 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 for the resource is available for use. In order for clients to be able to use the resource at 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 resource. 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 resources 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 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 [7], 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 resources , for example a `javax.ws.rs.core.Application` object.

5 Technical Solution

5.1 JAX-RS Resource Endpoints

JAX RS resources (sometimes known as 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 resource 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 resources and Servlets this RFC proposes to reuse the whiteboard model defined by the Http Whiteboard defined in chapter 140 of the OSGi Compendium Specification [8].

5.1.1 Resource mapping

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

1. It serves as a marker to the JAX-RS whiteboard runtime that this OSGi service should be hosted as a JAX-RS resource
2. The value of the property defines a base request URI path to which this resource should be mapped. This value is prepended to the path defined by the JAX-RS resource in its annotations. 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 resources 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 parameter:

```
@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 parameter, 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 resource 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 {

    private final List<String> entries = Arrays.asList("fizz", "buzz",
        "fizzbuzz");

    @GET
    public List<String> getFoos() {
        return Collections.unmodifiableList(entries);
    }

    @GET
    @PathParam("{name: [a-zA-Z]+}")
    public String getFoo(@PathParam("name") String name) {
        if(entries.contains(name)) {
            return getFooInfo(name);
        }
    }
}
```

```

        throw new IllegalArgumentException("No foo called " + name);
    }

}

```

As a result of the way in which URI paths are mapped by JAX-RS resources, one resource 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.resource.base` property value of “bar”

5.1.1.1 Path matching example

```

@Path("foo")
public class Foo {

    private final List<String> entries = Arrays.asList("fizz", "buzz",
        "fizzbuzz");

    @GET
    public List<String> getFoos() {
        return Collections.unmodifiableList(entries);
    }

    @GET
    @PathParam("{name: [a-zA-Z]+}")
    public String getFoo(@PathParam("name") String name) {
        if(entries.contains(name)) {
            return "A foo called " + name;
        }
        throw new IllegalArgumentException("No foo called " + name);
    }

}

```

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

URI path	Result
/	No match (404)
/foo	No match (404)
/bar	No match (404)
/bar/foo	“[fizz, buzz, fizzbuzz]”
/bar/foo/fizz	“A foo called fizz”
/bar/foo/buzz	“A foo called buzz”
/bar/foo/foobar	Error 500 (IllegalArgumentException)

/bar/foo/fizz/buzz	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 resource when accessing the HTTP endpoint.

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/bar/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 Additional JAX-RS types

In addition to JAX-RS resources, a JAX RS application may define a number of different types which provide extensions to the JAX-RS container. For Example the JAX-RS 2.0 specification defines the following:

- JAX-RS Container Request and Container Response filters are used to alter request and response parameters,
- JAX-RS Reader and Writer Interceptors are used to alter the incoming or outgoing request/response entities
- JAX-RSMessage body readers/writers are used to deserialize/serialize objects to the wire for a given media type (e.g. application/json).
- JAX-RS Context Resolvers are used to provide objects for injection into the JAX-RS resources by the JAX-RS runtime
- JAX-RS Exception Mappers are used to map exceptions thrown by JAX-RS resources into responses
- JAX-RS Parameter Converter Providers are used to map rich parameter types to and from String values
- JAX-RS Features and Dynamic Features are used as a way to register multiple extension types with the JAX-RS container. Dynamic Features further allow the extensions to be targeted to specific resources within the JAX-RS container

All of these extensions may be provided to the JAX-RS whiteboard by registering them as services with the appropriate advertised interface(s) and the `osgi.jaxrs.extension.name` property. Due to the way in which JAX-RS Extensions are registered with the container it is not possible to restrict the set of types provided. This means that if the extension service object implements multiple extension interface then all of them will be used by the JAX-RS container, regardless of which types are advertised by the service. It is therefore recommended that the objectClass provided by the service is unimportant as the extension types service will be registered with the JAX-RS container and introspected to determine its type.

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.

5.2.2 Pre-matching filters

JAX-RS filters can be annotated with `@PreMatching` to indicate that they should be applied *before* the container works out which resource should be called by the incoming request. These filters can therefore change the request such that it maps to a different resource than it would have before the filter's operation. `@PreMatching` filters cannot be Named (as no corresponding named resource is available to the runtime yet).

Pre matching filters should be handled by the container, honouring their relative position in the filter list

5.2.3 Message Body Reader/Writer Content Types

Message Body Reader types can be annotated with `@Consumes` to define the media type that they accept. Similarly Message Body Writer types can be annotated with `@Produces` to define the media type that they produce. If provided to the whiteboard then these annotations must be taken into account when determining whether a reader/writer should be called. Sections of the media type may be wildcarded, e.g. "application/*" or "*/*". If the `@Consumes/@Provides` annotation is not present then the value defaults to `*/*`.

For incoming requests the `@Consumes` must be matched against the incoming Content-Type header of the request. For outgoing responses the `@Provides` annotation must be matched against the outgoing Content-Type header.

5.2.4 Overall ordering of calls

The overall ordering of request/response mapping mimics the standard behaviours described by the JAX-RS RI [6].

Where more than one filter/interceptor/reader/writer is available then they are ordered according to their `javax.annotation.Priority`. If two types have the same priority then they are ordered according to the natural ordering of their service references

The processing flow is as follows:

- 1) Server receives a request
- 2) Pre-matching `ContainerRequestFilters` are executed. Changes made here can affect which resource method is chosen
- 3) The Server matches the request to a resource method
- 4) Post-matching `ContainerRequestFilters` are executed. This includes execution of all filters which match the incoming path and any name-bound filters.
- 5) `ReaderInterceptors` which match the incoming path are applied to the incoming request body. If the request has no body then the `ReaderInterceptors` are not called.

- 6) The list of `MessageBodyReaders` applicable to the path and incoming content type are tried according to the standard ordering rules. The first `MessageBodyReader` which states that it can deserialise the entity “wins” and is used to create the entity object. If the incoming request has no body then no `MessageBodyReaders` are called.
- 7) If the resource is request scoped then it is instantiated and injected with relevant types from any defined `ContextResolver` instance. The Server resource method is executed.
- 8) `ContainerResponseFilters` are executed. This includes execution of all filters which match the incoming path and any name-bound filters.
- 9) `WriterInterceptors` which match the incoming path are applied to the outgoing request stream. If the response has no body then the `WriterInterceptors` are not called.
- 10) The list of `MessageBodyWriters` applicable to the path and outgoing content type are tried according to the standard ordering rules. The first `MessageBodyWriter` which states that it can serialise the entity “wins” and is used to write out the entity object. If there is no response body then no `MessageBodyWriters` are called
- 11) Server response is flushed/committed

5.2.5 Depending on Extension Services

When writing and configuring a JAX-RS resource or extension it is possible for one extension to depend on another. For example a JAX-RS resource may have a dependency upon a `MessageBodyWriter` to provide JSON serialization, or a `ContainerRequestFilter` may depend on a `ContextResolver` to provide injected configuration.

In these cases it is necessary to express a dependency on the existence of an extension within the JAX-RS container. This can be expressed on any JAX-RS whiteboard service using the `osgi.jaxrs.extension.select` property. This property has type `String+` and contains a list of LDAP filters. These filters will be run against the service properties of each extension service registered with the container. If all of the supplied filters pass then the whiteboard service will be registered.

If at some point later a necessary extension service dependency becomes unavailable then the whiteboard service will become ineligible for inclusion in the JAX-RS container until a suitable replacement is found.

5.3 Service Lifecycle

5.3.1 Registering and Unregistering JAX-RS resources

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

If a registered JAX-RS resource'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 resources, 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 resources 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.

To control whether a whiteboard service provides a singleton or per-request JAX-RS resource the registering bundle should set the scope of the OSGi service. If the service is registered as a prototype service factory then the JAX-RS whiteboard runtime must request a new service object for every request. Otherwise the JAX-RS resource is a singleton, and the same instance must be used to service each request.

Service scope	JAX-RS behaviour
singleton/bundle	Singleton resource looked up and used for all requests
prototype	A request scoped resource. 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`. Note that the base path of the application is applied in addition to any defined `ApplicationPath` annotation present on the `Application` Object.

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 and `Extension Services`.

5.5 JAX-RS endpoint advertisement

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

	RS service
<code>osgi.jaxrs.service.id</code>	Set to the id of the JAX-RS service that this endpoint represents

5.6

5.7 Error Handling

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

5.7.1 Failure to obtain a service instance

In the case where a published service is unable to be 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.

5.7.2 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 a relevant JAX-RS extension type, or to provide any resource methods 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.

5.7.3 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.

5.7.4 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 indicating that the JAX-RS object is unable to be registered due to an incompatible class-space.

5.7.5 Missing Required Extensions

If a JAX-RS resource or extension requires one or more extensions using a `osgi.jaxrs.extension.select` filter then at any given time it is possible that the JAX-RS container will not be able to host the resource. At this time a failure DTO must be created for the relevant resource or extension service.

5.8 JaxRSServiceRuntime

The JAX-RS Runtime Service provides introspection the current state of the JAX-RS Whiteboard Service, including information about the used whiteboard services, failed whiteboard services, and shadowed whiteboard services. The JAX-RS Runtime service also provides a mechanism to communicate information about the JAX-RS runtime implementation and configuration.

5.8.1 JAX-RS runtime properties

The JAX-RS Runtime Service implementation must define a set of runtime properties which can be used by whiteboard services to select a specific implementation. JAX-RS whiteboard services may register with a `osgi.jaxrs.whiteboard.target` service property, the value of which is an LDAP filter used to select applicable runtime instances.

The runtime attributes can be examined as service properties of the `HttpServiceRuntime` service registration. The runtime attributes may include providers specific properties, and must include the `osgi.jaxrs.endpoint` property. The value of this property is a `String+` containing the base URI(s) of the JAX-RS container. The URIs may either be absolute, or relative URIs if the root URI of the HTTP server is unknown (for example in a bridged framework).

5.8.2 JAX-RS whiteboard configuration

The level of configurability of the JAX-RS Whiteboard Service may vary between implementations. Some implementations may allow users to configure the HTTP interface and port (for example an implementation that creates its own HTTP server) while others will not have any control over the HTTP runtime (for example a runtime which builds on top of the `Http Service Whiteboard`).

If an implementation supports configuration, such configuration should be supplied via the Configuration Admin Service. The PID used is implementation specific as different implementations may choose to use different patterns, for example factory configurations. Importantly the JAX-RS service runtime must follow the expected practice that public Configuration properties (ones that do not start with a '.') are registered as service properties that can then be used when selecting a target runtime.

5.8.3 Runtime Introspection and DTOs

The `JaxRSServiceRuntime` provides two methods.

The first method on the Runtime Service is used to retrieve a snapshot of the current runtime state in a DTO. Further information about this DTO is available in section 6.1.

The second method on the Runtime Service is used to determine which JAX-RS resources, filters and interceptors would be used to handle an incoming call with a given URI path. Further information about this DTO is available in section 6.2

5.9 JAX-RS Clients

The JAX-RS 2.0 specification includes a client API for making REST requests. The normal mechanism for obtaining the Client is to use a `ClientBuilder`, which is instantiated using a static factory method.

As static factory methods require the reflective loading of classes they rarely work well in OSGi. There are also significant lifecycle issues, as there is no way to force indirectly wired objects to be discarded if the implementation bundle is stopped or uninstalled.

The correct way to obtain implementations in OSGi is to use the Service Registry. JAX-RS implementations must therefore register their `ClientBuilder` implementations as OSGi services.

5.9.1 Locating specific implementations

Some Client implementations provide additional methods and features on provider-specific interfaces. To provide these the JAX-RS provider should register the `ClientBuilder` advertising both the `javax.ws.rs.client.ClientBuilder` and as the provider-specific interface.

The `ClientBuilder` must be registered as a prototype scoped service. That will allow bundles to configure multiple separate Client instances.

5.9.2 Asynchronous Calls

The JAX-RS client API supports the concept of asynchronous calls which return a `java.util.concurrent.Future`. `Future` is a type with very limited capabilities, and it is impossible to use in a truly asynchronous way. As a result the JAX-RS client API provides a mechanism to register a callback on asynchronous calls, called the `InvocationCallback`. As OSGi promises provide a much better asynchronous programming model than Java `Futures` the JAX-RS whiteboard should offer a simple mechanism to obtain a promise from the JAX-RS client.

Due to the way in which JAX-RS infers return types, the `InvocationCallback` must have a concrete generic type at compilation time, and cannot be a wildcard. Therefore the proposed mechanism for obtaining a Promise is:

```
Client client = clientBuilder.build();

PromiseHandler<String> handler = new PromiseHandler<String>{};

client.target(REST_SERVICE_URL)
    .path("/foo")
    .path("/{name}")
    .resolveTemplate("name", buzz)
    .request()
    .async()
    .get(handler);

Promise<String> result = handler.getPromise();
```

5.9.3 Client Filters, Interceptors, Readers and Writers

While Container (Server) filters, interceptors, readers and writers can be made available as whiteboard components, the same is not true for Clients. There are two main reasons for this:

- There is no way to scope the filters and interceptors that would be applied to a given client. In a multi-tenant environment this could lead to unexpected behaviours.
- Clients are not, in general, expected to be extended by third parties. The Client model is designed to be used by a bundle when making requests from a REST API. If further requests need to be made by a different bundle then it should create and configure a separate client. This is different from the container, where one container port may host several distinct resources.

In order to add filters, interceptors, readers and writers to the JAX-RS client users should use the `register()` method when building their client.

5.10 Implementation Provided Capabilities

The JAX-RS whiteboard implementation must define the following capabilities, including any additional static attributes:

- `osgi.service;objectClass:List<String>="org.osgi.service.jaxrs.runtime.JaxRSServiceRuntime"`
- `osgi.implementation;osgi.implementation="osgi.jaxrs";version:Version="1.0"`

Also, if the whiteboard implementation includes the JAX-RS API then it must also advertise the JavaJAXRS contract

`osgi.contract;osgi.contract=JavaJAXRS;version:List<Version>="1.1,2.0"`

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.

6.1 The Runtime DTO

The `RuntimeDTO` is the root DTO representing the state of the JAX RS Whiteboard. The DTO provides a snapshot of the system at the point where the DTO is generated. It can be used to list the available JAX-RS resources, filters and interceptors. In addition the DTO provides information about resources, filters and interceptors that have failed in some way.

6.2 The RequestInfo DTO

The `RequestInfo` DTO provides information about which resource will be called for a given URI, and which filters and interceptors will participate in the invocation chain. This can be used to diagnose issues, and to help with configuring additional parts in an active system.

7 Javadoc

OSGi Javadoc

26/01/17 13:11

Package Summary		Page
org.osgi.service.jaxrs.runtime	JAX-RS Runtime Package Version 1.0.	22
org.osgi.service.jaxrs.runtime.dto	JAX-RS Runtime DTO Package Version 1.0.	25
org.osgi.service.jaxrs.whiteboard	JAX-RS Whiteboard Package Version 1.0.	41

Package org.osgi.service.jaxrs.runtime

@org.osgi.annotation.versioning.Version(value="1.0")

JAX-RS Runtime Package Version 1.0.

See:

[Description](#)

Interface Summary		Page
JaxRSServiceRuntime	The JaxRSServiceRuntime service represents the runtime information of a JAX-RS Whiteboard implementation.	23

Class Summary		Page
JaxRSServiceRuntimeConstants	Defines standard names for Http Runtime Service constants.	24

Package org.osgi.service.jaxrs.runtime Description

JAX-RS Runtime Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.jaxrs.runtime; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.jaxrs.runtime; version="[1.0,1.1)"
```


Interface JaxRSServiceRuntime

org.osgi.service.jaxrs.runtime

```
@org.osgi.annotation.versioning.ProviderType
public interface JaxRSServiceRuntime
```

The JaxRSServiceRuntime service represents the runtime information of a JAX-RS Whiteboard implementation.

It provides access to DTOs representing the current state of the service.

The JaxRSServiceRuntime service must be registered with the [JaxRSServiceRuntimeConstants.JAX_RS_SERVICE_ENDPOINT](#) service property.

ThreadSafe

Method Summary		Page
RequestInfoDTO	calculateRequestInfoDTO (String path) Return a request info DTO containing the services involved with processing a request for the specified path.	23
RuntimeDTO	getRuntimeDTO () Return the runtime DTO representing the current state.	23

Method Detail

getRuntimeDTO

[RuntimeDTO](#) [getRuntimeDTO](#)()

Return the runtime DTO representing the current state.

Returns:
The runtime DTO.

calculateRequestInfoDTO

[RequestInfoDTO](#) [calculateRequestInfoDTO](#)(String path)

Return a request info DTO containing the services involved with processing a request for the specified path.

Parameters:
path - The request path, relative to the root of the JAX-RS Whiteboard implementation.

Returns:
The request info DTO for the specified path.

Class JaxRSServiceRuntimeConstants

[org.osgi.service.jaxrs.runtime](#)

```
java.lang.Object
└─org.osgi.service.jaxrs.runtime.JaxRSServiceRuntimeConstants
```

```
final public class JaxRSServiceRuntimeConstants
extends Object
```

Defines standard names for Http Runtime Service constants.

Field Summary		Page
static String	JAX_RS_SERVICE_ENDPOINT JAX-RS Runtime Service service property specifying the endpoints upon which the JAX-RS implementation is available.	24

Field Detail

JAX_RS_SERVICE_ENDPOINT

```
public static final String JAX_RS_SERVICE_ENDPOINT = "osgi.jaxrs.endpoint"
```

JAX-RS Runtime Service service property specifying the endpoints upon which the JAX-RS implementation is available.

An endpoint value is a URL or a relative path, to which the JAX-RS Whiteboard implementation is listening. For example, `http://192.168.1.10:8080/` or `/myapp/`. A relative path may be used if the scheme and authority parts of the URL are not known, e.g. if a bridged Http Whiteboard implementation is used. If the JAX-RS Whiteboard implementation is serving the root context and neither scheme nor authority is known, the value of the property is `"/"`. Both, a URL and a relative path, must end with a slash.

A JAX-RS Whiteboard implementation can be listening on multiple endpoints.

The value of this service property must be of type `String`, `String[]`, or `Collection<String>`.

Package org.osgi.service.jaxrs.runtime.dto

@org.osgi.annotation.versioning.Version(value="1.0")

JAX-RS Runtime DTO Package Version 1.0.

See:

[Description](#)

Class Summary		Page
DTOConstants	Defines standard constants for the DTOs.	26
ExtensionDTO	Represents a JAX-RS Filter service currently being hosted by the JaxRSServiceRuntime	28
FailedExtensionDTO	Represents a JAX-RS Filter service which is currently not being used due to a problem.	30
FailedResourceDTO	Represents a JAX-RS service which is currently not being used due to a problem.	32
RequestInfoDTO	Represents the services used to process a specific request.	34
ResourceDTO	Represents common information about a JAX-RS resource service.	36
ResourceMethodInfoDTO	Represents information about a JAX-RS resource method.	38
RuntimeDTO	Represents the state of a Http Service Runtime.	39

Package org.osgi.service.jaxrs.runtime.dto Description

JAX-RS Runtime DTO Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.jaxrs.runtime.dto; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.jaxrs.runtime.dto; version="[1.0,1.1)"
```

Class DTOConstants

org.osgi.service.jaxrs.runtime.dto

```
java.lang.Object
└─ org.osgi.service.jaxrs.runtime.dto.DTOConstants
```

```
final public class DTOConstants
extends Object
```

Defines standard constants for the DTOs. The error codes are defined to take the same values as used by the Http Service Whiteboard

Field Summary		Page
static int	FAILURE_REASON_NOT_AN_EXTENSION_TYPE The extension service is registered in the service registry but the service is not registered using a recognised extension type	27
static int	FAILURE_REASON_REQUIRED_EXTENSIONS_UNAVAILABLE The service is registered in the service registry with the JaxRSWhiteboardConstants.JAX_RS_EXTENSION_SELECT property and one or more of the filters is not matched.	27
static int	FAILURE_REASON_SERVICE_NOT_GETTABLE The service is registered in the service registry but getting the service fails as it returns null.	26
static int	FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE Service is shadowed by another service.	26
static int	FAILURE_REASON_UNKNOWN Failure reason is unknown.	26
static int	FAILURE_REASON_VALIDATION_FAILED The service is registered in the service registry but the service properties are invalid.	27

Field Detail

FAILURE_REASON_UNKNOWN

```
public static final int FAILURE_REASON_UNKNOWN = 0
```

Failure reason is unknown.

FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE

```
public static final int FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE = 1
```

Service is shadowed by another service.

For example, a service with the same service properties but a higher service ranking.

FAILURE_REASON_SERVICE_NOT_GETTABLE

```
public static final int FAILURE_REASON_SERVICE_NOT_GETTABLE = 2
```

The service is registered in the service registry but getting the service fails as it returns null.

FAILURE_REASON_VALIDATION_FAILED

```
public static final int FAILURE_REASON_VALIDATION_FAILED = 3
```

The service is registered in the service registry but the service properties are invalid.

FAILURE_REASON_NOT_AN_EXTENSION_TYPE

```
public static final int FAILURE_REASON_NOT_AN_EXTENSION_TYPE = 4
```

The extension service is registered in the service registry but the service is not registered using a recognised extension type

FAILURE_REASON_REQUIRED_EXTENSIONS_UNAVAILABLE

```
public static final int FAILURE_REASON_REQUIRED_EXTENSIONS_UNAVAILABLE = 5
```

The service is registered in the service registry with the [JaxRSWhiteboardConstants.JAX_RS_EXTENSION_SELECT](#) property and one or more of the filters is not matched.

Class ExtensionDTO

org.osgi.service.jaxrs.runtime.dto

```
java.lang.Object
├── org.osgi.dto.DTO
│   └── org.osgi.service.jaxrs.runtime.dto.ExtensionDTO
```

Direct Known Subclasses:

[FailedExtensionDTO](#)

```
public class ExtensionDTO
extends org.osgi.dto.DTO
```

Represents a JAX-RS Filter service currently being hosted by the [JaxRSServiceRuntime](#)

NotThreadSafe

Field Summary		Page
String[]	consumes The media types consumed by this service, if provided in an <code>Consumes</code> annotation	29
String[]	extensionTypes The extension types recognized for this service.	29
ResourceDTO []	filteredByName The resourceDTOs that are mapped to this extension using a <code>NameBinding</code> annotation	29
String	name The JaxRSWhiteboardConstants.JAX_RS_EXTENSION_NAME for this extension	28
String[]	produces The media types produced by this service, if provided in an <code>Produces</code> annotation	29
long	serviceId Service property identifying the JAX-RS extension service.	29

Constructor Summary	Page
ExtensionDTO ()	29

Methods inherited from class org.osgi.dto.DTO

toString

Field Detail

name

```
public String name
```

The [JaxRSWhiteboardConstants.JAX_RS_EXTENSION_NAME](#) for this extension

extensionTypes

```
public String[] extensionTypes
```

The extension types recognized for this service.

produces

```
public String[] produces
```

The media types produced by this service, if provided in an `Produces` annotation

consumes

```
public String[] consumes
```

The media types consumed by this service, if provided in an `Consumes` annotation

filteredByName

```
public ResourceDTO[] filteredByName
```

The resourceDTOs that are mapped to this extension using a `NameBinding` annotation

serviceId

```
public long serviceId
```

Service property identifying the JAX-RS extension service.

Constructor Detail

ExtensionDTO

```
public ExtensionDTO()
```


Class `FailedExtensionDTO`

[org.osgi.service.jaxrs.runtime.dto](#)

```
java.lang.Object
├── org.osgi.dto.DTO
│   └── org.osgi.service.jaxrs.runtime.dto.ExtensionDTO
│       └── org.osgi.service.jaxrs.runtime.dto.FailedExtensionDTO
```

```
public class FailedExtensionDTO
    extends ExtensionDTO
```

Represents a JAX-RS Filter service which is currently not being used due to a problem.

As the service represented by this DTO is not used due to a failure, the field [ExtensionDTO.filteredByName](#) always returns {null}.

NotThreadSafe

Field Summary		Page
int	failureReason The reason why the servlet filter represented by this DTO is not used.	30

Fields inherited from class `org.osgi.service.jaxrs.runtime.dto` [ExtensionDTO](#)

[consumes](#), [extensionTypes](#), [filteredByName](#), [name](#), [produces](#), [serviceId](#)

Constructor Summary		Page
FailedExtensionDTO ()		31

Methods inherited from class `org.osgi.dto.DTO`

[toString](#)

Field Detail

`failureReason`

```
public int failureReason
```

The reason why the servlet filter represented by this DTO is not used.

See Also:

[DTOConstants.FAILURE_REASON_UNKNOWN](#),
[DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE](#),
[DTOConstants.FAILURE_REASON_VALIDATION_FAILED](#),
[DTOConstants.FAILURE_REASON_NOT_AN_EXTENSION_TYPE](#),
[DTOConstants.FAILURE_REASON_REQUIRED_EXTENSIONS_UNAVAILABLE](#)

Constructor Detail

FailedExtensionDTO

```
public FailedExtensionDTO()
```

Class `FailedResourceDTO`

[org.osgi.service.jaxrs.runtime.dto](#)

```
java.lang.Object
├── org.osgi.dto.DTO
│   └── org.osgi.service.jaxrs.runtime.dto.ResourceDTO
│       └── org.osgi.service.jaxrs.runtime.dto.FailedResourceDTO
```

```
public class FailedResourceDTO
    extends ResourceDTO
```

Represents a JAX-RS service which is currently not being used due to a problem.

As the service represented by this DTO is not used due to a failure, the field [ResourceDTO.resourceMethods](#) always returns null.

NotThreadSafe

Field Summary		Page
int	failureReason The reason why the resource represented by this DTO is not used.	32

Fields inherited from class `org.osgi.service.jaxrs.runtime.dto.ResourceDTO`

[base](#), [name](#), [resourceMethods](#), [serviceId](#)

Constructor Summary		Page
FailedResourceDTO ()		33

Methods inherited from class `org.osgi.dto.DTO`

[toString](#)

Field Detail

`failureReason`

```
public int failureReason
```

The reason why the resource represented by this DTO is not used.

See Also:

[DTOConstants.FAILURE_REASON_UNKNOWN](#),
[DTOConstants.FAILURE_REASON_SERVICE_NOT_GETTABLE](#),
[DTOConstants.FAILURE_REASON_VALIDATION_FAILED](#),
[DTOConstants.FAILURE_REASON_SHADOWED_BY_OTHER_SERVICE](#),
[DTOConstants.FAILURE_REASON_REQUIRED_EXTENSIONS_UNAVAILABLE](#)

Constructor Detail

FailedResourceDTO

public **FailedResourceDTO**()

Class RequestInfoDTO

[org.osgi.service.jaxrs.runtime.dto](#)

```
java.lang.Object
├── org.osgi.dto.DTO
│   └── org.osgi.service.jaxrs.runtime.dto.RequestInfoDTO
```

```
public class RequestInfoDTO
    extends org.osgi.dto.DTO
```

Represents the services used to process a specific request.

NotThreadSafe

Field Summary		Page
ExtensionDTO[]	extensionDTOs The JAX-RS filters processing this request.	34
String	path The path of the request relative to the root.	34
ResourceDTO	servletDTO The resource processing this request.	34

Constructor Summary	Page
RequestInfoDTO ()	35

Methods inherited from class org.osgi.dto.DTO

toString

Field Detail

path

```
public String path
```

The path of the request relative to the root.

extensionDTOs

```
public ExtensionDTO[] extensionDTOs
```

The JAX-RS filters processing this request. If no filters are called for processing this request, an empty array is returned.

servletDTO

```
public ResourceDTO servletDTO
```

The resource processing this request.

Constructor Detail

RequestInfoDTO

public **RequestInfoDTO**()

Class ResourceDTO

[org.osgi.service.jaxrs.runtime.dto](#)

```
java.lang.Object
├─ org.osgi.dto.DTO
│   └─ org.osgi.service.jaxrs.runtime.dto.ResourceDTO
```

Direct Known Subclasses:

[FailedResourceDTO](#)

```
abstract public class ResourceDTO
extends org.osgi.dto.DTO
```

Represents common information about a JAX-RS resource service.

NotThreadSafe

Field Summary		Page
String	base The base URI of the resource defined by JaxRSWhiteboardConstants.JAX_RS_RESOURCE_BASE .	36
String	name The name of the resource if it set one using JaxRSWhiteboardConstants.JAX_RS_RESOURCE_NAME , null otherwise.	36
ResourceMethodInfoDTO[]	resourceMethods The RequestPaths handled by this resource	37
long	serviceId Service property identifying the JAX-RS resource service	37

Constructor Summary	Page
ResourceDTO ()	37

Methods inherited from class org.osgi.dto.DTO
toString

Field Detail

name

```
public String name
```

The name of the resource if it set one using [JaxRSWhiteboardConstants.JAX_RS_RESOURCE_NAME](#), null otherwise.

base

```
public String base
```


The base URI of the resource defined by [JaxRSWhiteboardConstants.JAX_RS_RESOURCE_BASE](#).

resourceMethods

```
public ResourceMethodInfoDTO[] resourceMethods
```

The RequestPaths handled by this resource

serviceId

```
public long serviceId
```

Service property identifying the JAX-RS resource service

Constructor Detail

ResourceDTO

```
public ResourceDTO()
```

Class ResourceMethodInfoDTO

[org.osgi.service.jaxrs.runtime.dto](#)

```
java.lang.Object
└─ org.osgi.dto.DTO
    └─ org.osgi.service.jaxrs.runtime.dto.ResourceMethodInfoDTO
```

```
public class ResourceMethodInfoDTO
extends org.osgi.dto.DTO
```

Represents information about a JAX-RS resource method.

NotThreadSafe

Constructor Summary	Page
ResourceMethodInfoDTO()	38

Methods inherited from class org.osgi.dto.DTO
toString

Constructor Detail

ResourceMethodInfoDTO

```
public ResourceMethodInfoDTO()
```

Class RuntimeDTO

org.osgi.service.jaxrs.runtime.dto

```
java.lang.Object
├── org.osgi.dto.DTO
│   └── org.osgi.service.jaxrs.runtime.dto.RuntimeDTO
```

```
public class RuntimeDTO
extends org.osgi.dto.DTO
```

Represents the state of a Http Service Runtime.

NotThreadSafe

Field Summary		Page
String	contextPath The root context path.	40
ExtensionDTO[]	extensionDTOs Returns the representations of the JAX-RS extension services associated with this Runtime.	40
FailedExtensionDTO[]	failedExtensionDTOs Returns the representations of the JAX-RS extension services associated with this runtime but currently not used due to some problem.	40
FailedResourceDTO[]	failedResourceDTOs Returns the representations of the JAX-RS resource services associated with this runtime but currently not used due to some problem.	40
org.osgi.resource.dto.ResourceDTO[]	resourceDTOs Returns the representations of the JAX-RS resource services associated with this Runtime.	40
org.osgi.framework.dto.ServiceReferenceDTO	serviceDTO The DTO for the corresponding JaxRSServiceRuntime.	39

Constructor Summary	Page
RuntimeDTO()	40

Methods inherited from class org.osgi.dto.DTO

toString

Field Detail

serviceDTO

```
public org.osgi.framework.dto.ServiceReferenceDTO serviceDTO
```

The DTO for the corresponding JaxRSServiceRuntime. This value is never null.

contextPath

```
public String contextPath
```

The root context path. This path may be "/", or it may be a sub path if this [JaxRSServiceRuntime](#) is running within an external Http container.

resourceDTOs

```
public org.osgi.resource.dto.ResourceDTO[] resourceDTOs
```

Returns the representations of the JAX-RS resource services associated with this Runtime. The returned array may be empty if this whiteboard is currently not associated with any JAX-RS Resource services.

extensionDTOs

```
public ExtensionDTO[] extensionDTOs
```

Returns the representations of the JAX-RS extension services associated with this Runtime. The returned array may be empty if this whiteboard is currently not associated with any JAX-RS extension services.

failedResourceDTOs

```
public FailedResourceDTO[] failedResourceDTOs
```

Returns the representations of the JAX-RS resource services associated with this runtime but currently not used due to some problem. The returned array may be empty.

failedExtensionDTOs

```
public FailedExtensionDTO[] failedExtensionDTOs
```

Returns the representations of the JAX-RS extension services associated with this runtime but currently not used due to some problem. The returned array may be empty.

Constructor Detail

RuntimeDTO

```
public RuntimeDTO()
```

Package org.osgi.service.jaxrs.whiteboard

```
@org.osgi.annotation.versioning.Version(value="1.0")
```

JAX-RS Whiteboard Package Version 1.0.

See:

[Description](#)

Class Summary		Page
JaxRSWhiteboardConstants	Defines standard constants for the JAX-RS Whiteboard services.	42

Package org.osgi.service.jaxrs.whiteboard Description

JAX-RS Whiteboard Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.jaxrs.whiteboard; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.jaxrs.whiteboard; version="[1.0,1.1)"
```

Class JaxRSWhiteboardConstants

org.osgi.service.jaxrs.whiteboard

java.lang.Object

└─ org.osgi.service.jaxrs.whiteboard.JaxRSWhiteboardConstants

```
final public class JaxRSWhiteboardConstants
extends Object
```

Defines standard constants for the JAX-RS Whiteboard services.

Field Summary		Page
static String	JAX_RS_EXTENSION_NAME Service property specifying the name of a JAX-RS extension service.	43
static String	JAX_RS_EXTENSION_SELECT A Service property specifying one or more target filters used to select the set of JAX-RS extension services required to support this whiteboard service.	43
static String	JAX_RS_RESOURCE_BASE Service property specifying the base URI mapping for a JAX-RS resource service.	42
static String	JAX_RS_RESOURCE_NAME Service property specifying the name of a JAX-RS resource.	42
static String	JAX_RS_WHITEBOARD_TARGET Service property specifying the target filter to select the JAX-RS Whiteboard implementation to process the service.	43

Field Detail

JAX_RS_RESOURCE_NAME

```
public static final String JAX_RS_RESOURCE_NAME = "osgi.jaxrs.name"
```

Service property specifying the name of a JAX-RS resource.

This name is provided as a property on the registered Endpoint service so that the URI for a particular JAX-RS service can be identified. If this service property is not specified, then no Endpoint information will be registered for this resource.

Resource names should be unique among all resource service associated with a single Whiteboard implementation.

The value of this service property must be of type `String`.

JAX_RS_RESOURCE_BASE

```
public static final String JAX_RS_RESOURCE_BASE = "osgi.jaxrs.resource.base"
```

Service property specifying the base URI mapping for a JAX-RS resource service.

The specified uri is used to determine whether a request should be mapped to the resource. Services without this service property are ignored.

The value of this service property must be of type `String`, and will have a "/" prepended if no "/" exists.

JAX_RS_EXTENSION_NAME

```
public static final String JAX_RS_EXTENSION_NAME = "osgi.jaxrs.extension.name"
```

Service property specifying the name of a JAX-RS extension service.

A service providing this property must be registered as one or more of the following types:

- `MessageBodyReader`
- `MessageBodyWriter`
- `ContainerRequestFilter`
- `ContainerResponseFilter`
- `ReaderInterceptor`
- `WriterInterceptor`
- `ContextResolver`
- `ExceptionMapper`
- `ParamConverterProvider`
- `Feature`
- `DynamicFeature`

If a service with this property does not match any of the defined types then it is registered as a failure DTO,

JAX_RS_EXTENSION_SELECT

```
public static final String JAX_RS_EXTENSION_SELECT = "osgi.jaxrs.extension.select"
```

A Service property specifying one or more target filters used to select the set of JAX-RS extension services required to support this whiteboard service.

A JAX-RS Whiteboard service may require one or more extensions to be available so that it can function. For example a resource which declares that it `@Produces("text/json")` requires a `MessageBodyWriter` which supports JSON to be available.

This service property provides a `String+` set of LDAP filters which will be applied to the service properties of all extensions available in the JAX-RS container. If all of the filters are satisfied then this service is eligible to be hosted by the JAX-RS container.

This service property may be declared by any JAX-RS whiteboard service, whether it is a resource, or an extension.

If this service property is not specified, then no extensions are required.

The value of this service property must be of type `String` and be a valid `filter string`.

JAX_RS_WHITEBOARD_TARGET

```
public static final String JAX_RS_WHITEBOARD_TARGET = "osgi.jaxrs.whiteboard.target"
```

Service property specifying the target filter to select the JAX-RS Whiteboard implementation to process the service.

A JAX-RS Whiteboard implementation can define any number of service properties which can be referenced by the target filter. The service properties should always include the [osgi.http.endpoint](#) service property if the endpoint information is known.

If this service property is not specified, then all JAX-RS Whiteboard implementations can process the service.

The value of this service property must be of type `String` and be a valid `filter string`.

DocFlex/Doclet is both a multi-format Javadoc doclet and a free edition of [DocFlex/Javadoc](#). If you need to customize your Javadoc without writing a full-blown doclet from scratch, DocFlex/Javadoc may be the only tool able to help you! Find out more at www.docflex.com

8 Considered Alternatives

8.1 JAX-RS Extension path restrictions

Typical JAX-RS filters and interceptors match all resources beneath their base URI, which means that they implicitly 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 resources, however rather than matching an exact number of URI segments they match all sub-paths, regardless of length.

Typical JAX-RS Message Body Readers and Writers are global in their scope, applied across all paths. This approach does not fit well with OSGi, as different context paths may wish to have differently configured readers and writers for a given media type. Therefore readers and writers are registered using the `osgi.jaxrs.reader.base` and `osgi.jaxrs.writer.base` respectively. These base URIs are written in the same way as for JAX-RS resources, however rather than matching an exact number of URI segments they match all sub-paths, regardless of length.

8.2 The JaxRSContext

JAX-RS Applications provide an excellent way to aggregate a number of resources, filters, interceptors, message body readers, and message body writers. This has the advantage that all of the additional handlers needed by a particular resource are guaranteed to be available before it handles its first request. For example a resource may require:

- A user context from Http Basic Authentication
- to consume an entity from a gzipped JSON input
- to produce a deflated YAML serialized response

Having all of these features automatically available solves an otherwise complex lifecycle issue that would require the JAX-RS resource to depend on a filter, some interceptors and some message body readers/writers. The main problem, however, is that the Application is relatively static, new JAX-RS resources cannot be easily added at runtime to extend an Application.

This use case is handled by the JaxRSContext service. The JaxRSContext provides a group of filters, interceptors, message body readers and message body writers as a named context registered against a particular context path. JAX-RS resources can then require the context using a filter in their service definition, preventing them from being served by the whiteboard until the context is available.

8.2.1 Providing a JaxRSContext

The JaxRSContext service may provide either singleton or request-scoped types, or a mixture of both. Request Scoped types are provided as a Set of Class<?>. These types must have a zero-argument constructor, and will be created for each request. Singleton resources are provided as a Set of Object. These objects will be used for each request. Finally the JaxRSContext provides a set of application-wide properties. This means that the JaxRSContext actually has the same methods as a JAX-RS Application object.

- Set<Class<?>> getClasses()
- Map<String, Object> getProperties()

- `Set<Object> getSingletons()`

8.2.2 Registering a JaxRSContext

JaxRSContexts are registered as OSGi services with two properties:

- `osgi.jaxrs.context.base` – this property defines the base URI that will be used by all resources and types provided by the context
- `osgi.jaxrs.context.name` – this property defines the name of the JaxRSContext. There is a default context called “default” which can be replaced using a service. If more than one JaxRSContext is defined with the same name then the natural ordering of the service references is used to determine the context that applies.

8.2.3 Requiring a JaxRSContext

JAX-RS whiteboard resources can require a JaxRSContext using the `osgi.jaxrs.context.select` property. The value of this property is a filter which is used to match the service properties of one or more JaxRSContext services. The JAX-RS resource will then be registered with each matching JaxRSContext.

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]. Jersey 2.x Documentation - <https://jersey.java.net/documentation/2.24.1/filters-and-interceptors.html#d0e10016>
- [7]. OSGi Portable Java Contracts - <http://www.osgi.org/Specifications/ReferenceContract>
- [8]. OSGi Compendium R6 - <https://osgi.org/download/r6/osgi.cmpn-6.0.0.pdf>

10.2 Author's Address

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

10.3 Acronyms and Abbreviations

10.4 End of Document