RFP-172 Protocols

Draft

11 Pages

Abstract

Distributed OSGi is most frequently used with the default configurations. This works quite well inside a cluster or other local topologies. However, the specification also makes it possible to specify configurations on the service, allowing a service to mapped to for example WS-deathstar, REST or JSON RPC. It turns out that several protocols have a useful mapping to a Java service and in the remaining cases annotations can give hint. This RFP seeks a proposal for the distributed OSGi configuration schemes to support REST with JSON and JSON RPC in such a way that the service interface is used as the definition of the protocol.

Copyright © OSGi Alliance 2015.
All company, brand and product names contained within this document may be trademarks that are the sole property of the respective owners.
The above notice must be included on all copies of this document that are made.

0 Document Information

0.1 Table of Contents

Copyright © OSGi Alliance 2015 All Rights Reserved

0.2 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.
```

0.3 Revision History

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

Revision	Date	Comments
Initial	*04-12-14*	*Initial* *Peter.Kriens@aQute.biz*
	03-08-15	*Update from Cologne meeting*

1 Introduction

Distributed OSGi is most frequently used with the default configurations. This works quite well inside a cluster or other local topologies. However, the specification also makes it possible to specify configurations on the service, allowing a service to mapped to for example REST or JSON RPC. It turns out that several protocols can inspect the to be distributed service and provide a valid mapping to it. In some cases annotations can give hint. This RFP seeks a proposal for the distributed OSGi configuration schemes to support RESTwith JSON and JSON RPC in such a way that the service interface is used as the definition of the protocol.

This RFP originates from the OSGi enRoute work. In this project, a number of services were identified, designed and implemented based on their needs for web based applications. This document analyzes the application domain and defines the problem that need to be solved.

Copyright © OSGi Alliance 2015 All Rights Reserved

2 Application Domain

2.1 Distributed OSGi

Distributed OSGi defines a number of *service properties* that are picked up by the *distribution provider* to *export* the corresponding service. Exporting a service means it is mapped to an *endpoint*. An endpoint is a location on the internet, usually addressable by a URL.

The primary property for distributed OSGI s `service.exported.interfaces`. This property permits a distribution provider to *export* a service. In general, the *topology*, all the *imported* and exported services of all systems, are controlled via the Remote Service Admin service.

A Distribution Provider provides one or more *protocols* to export the service. A protocol defines the means how the information is conveyed between the service importing and exporting system. Most implementations support a proprietary protocol by default that is highly optimized for transferring Java method calls. This can be proprietary because generally the distribution provider in the participating systems is the same.

2.2 Standard Protocols

However, one of the key design goals of the Distributed OSGi specification was to allow the use of endpoints that communicate with standard protocols. The typical use case in the specification is to register a service that maps to a remote WS-* web service. Since these protocols in general require more information than is conveyed in the service itself, the service properties can contain *configuration* for a specific protocol. The configuration is identified by a reverse domain name like `net.rmi`. The *configuration type* is listed in the `service.exported.configs` service property. Any details are then listed as additional service properties; each of these property keys must start with the configuration type. For example `net.rmi.url`.

Protocols like SOAP, REST, JSON RPC, and others create a request that in the end must be handled by a method on an object so that the result can be calculated. However, these protocols only define the wire format. The *request processor*, e.g. a servlet, must map the protocol elements to an internal call. Distributed OSGi provides the concept of an endpoint where an endpoint provides one or more procedures to call.

The request processor can use the configuration provided by the service. For example, a WS-* endpoint could find the details for the XML handling from a resource in the exporting service's bundle through the configuration. However, in several cases it is possible to use the service interface, whose information is available through Java reflection, as the specification for the endpoint in a given protocol. This requires mapping rules from the Java syntax to the protocol syntax. Since Java provides very rich reflective information it is usually possible to create endpoints that work quite well.

2.3 Security

Current security practice is that services exported and imported with default configurations assume they are in the same cluster and do not perform explicit security checks. This is likely not a good practice unless guaranteed by the Distribution Provider. i.e. it should at least have authenticated communication between the systems and they should be in the same security realm.

Copyright © OSGi Alliance 2015 All Rights Reserved

2.4 Remote Service Admin and Multiple Distribution Providers

The Distribution Provider must only export services for which it can handle the specified configuration type. If a Distribution Provider cannot handle certain configuration types then it is necessary to install an additional distribution provider. This would then install an additional Remote Service Admin, not all topology managers can handle multiple instances of this service. During the development of Remote Service Admin it was considered to expose the Distribution Provider which so far had been anonymous. Time was to short to implement this which made implementations unify the Distribution Provider and Remote Service Admin resulting in multiple when no Distribution Provider can support all wanted configuration types.

2.5 OSGi enRoute REST and JSON RPC

In OSGi enRoute, this model was used to map a service to a REST endpoint and a JSON RPC endpoint.

2.5.1 REST

For REST, the methods were used to specify the verb (GET, PUT, etc.), the partial URI (foo), and the remaining segments of the URI. For example, the following method can be mapped to a URL

```
Person getPerson( RESTRequest rq, int id) → GET /rest/foo/person/<id>
```

The return object is transferred via JSON, requiring the return object to be DTO like. A body in a PUT or POST request can be defined by extending the `RESTRequest` interface and define a `_body()` method on it. The return type of the body method must be also DTO like; the REST subsystem will automatically convert the body in JSON to this Java type. The same interface is used to pass any query parameters on the call (the part after the ?). These parameters can be defined as methods on this interface and they are automatically converted to the given parameter method's return type.

```
interface PutPersonRequest extends RESTRequest {
  Person _body();
  List<String> credentials();
}
boolean putPerson( PutPersonRequest rq )
```

The REST mapping also supports varargs, these methods get the remaining segments, this can be useful with DTOs:

```
Object getPerson( RESTRequest rq, int id, String ...path) {
  return dtos.get(persons.get(id),path);
}
// dtos → DTOs provides path based access into a DTO
```

One issue that needed to be handled was exception handling. REST APIs must translate any error to an HTTP response code. OSGi enRoute mapped common exceptions like File Not Found Exception being 404 and used 500 (Server Error) for the remaining.

The methods could access the HttpServletRequest and response via the RESTRequest argument.

2.5.2 JAX RS

JAX RS is a standard that defines a number of annotations to define the URIs for methods for a REST protocol. JAX RS is based on static class loading by default. However, a number of parties have made implementations that allow it to be useful with the OSGi service model. Instead of using a simple mapping from the method's parameters and the return type, the annotations provide a way to keep the URL completely different from the method's prototype, mostly by specifying crucial details in string arguments to the annotations. For example:

```
@Path(“/foo”)
```

Copyright © OSGi Alliance 2015 All Rights Reserved

```
public class Foo {

  @PUT
  @Path("bar/{duh}")
  public Response putFibreAttribs(
    @PathParam("duh") String fibreUrnStr,
    @QueryParam("data") String data) throws Exception {
        ...
    }
  }
}
```

Unlike enRoute, JAX RS provides alternative formats like XML for the input and out messages formats.

2.5.3 JSON RPC

JSON RPC is a protocol geared for communication between the server and the browser. It defines a message format based on JSON. The server and the browser exchange messages asynchronously. Either side can send a single message per web request or multiple. The protocol provides for exception handling, any thrown exceptions are forward to the caller.

In OSGi enRoute a JSON RPC mapping was defined to map a service interface. This is very straightforward since Java and Javascript are very similar. One of the largest differences is that Java can have multiple methods with the same name if the parameters differs and Javascript not. It turns out that the parameters and return values can be mapped well to each other. The JSON RPC runtime did provide a facility to return the names of the methods. This allowed the Javascript side to add those methods to an endpoint object. The application code then had the availability of a Javascript object with the same methods as the service interface without any extra work on the Javascript side, adding a method on the service is sufficent.

2.5.4 Javascript

Both REST and JSON RPC are heavily used from Javascript in the browser. This is an area where a lot of creativity has been applied in being malicious. This means there are a number of (semi) standards to protect against cross site scripting attacks and others. In the Java world, most of the defenses can be handled with Servlet Filters independent of the JSON RPC and REST engine.

2.6 Terminology + Abbreviations

3 Problem Description

Though there are a number of solutions to create REST and JSON RPC endpoints in OSGi it is currently not standardized nor are the current practices using the (extensive!) work in distributed OSGi leveraged widely. Amdatu and CXF use JSON but their internal communication, not for their standardized endpoints.

Copyright © OSGi Alliance 2015 All Rights Reserved

Since this would be the first time the OSGi Alliance will standardize a configuration type it should also consider the issue with multiple Remote Services Admin. Since the Distribution Provider does not have an API it requires Remote Service Admin implementations to embed all configuration types or use a proprietary plugin mechanism.

Concluding, this RFP seeks a solution to:

- A service based REST endpoint using JSON as data format based on distributed OSGi
- A service based JSON RPC endpoint based on distributed OSGi
- A plugin mechanism for configuration type providers

4 Use Cases

4.1 REST Usage

AI Bundle needs to provide a REST API that delivers the measurements of his management application. The objects that AI must return are:

```
public class Device extends DTO {
  public String name;
  public String type;
  public String serialnr;
  public Map<String,List<Measurement>> measurements;
}
public class Measurement extends DTO {
  public long time;
  public double value;
}
```

AI needs to find a REST API that provides the following URIs

```
../device                              All devices
../device/<name>                       One Device
../device/<name>/measurements          All the devices' msrmnts
../device/<name>/measurements/<mname>
../device/<name>/measurements/<mname>/time
../device/<name>/measurements/<mname>/value
```

AI therefore defines the following component

```
interface DeviceRESTAccess {
  Collection<Device> getDevice();

// Return Object because it can return different things depending on segments
```

Copyright © OSGi Alliance 2015 All Rights Reserved

```
  Object       getDevice(String name, String … segments );
}

@Component(
 property={
  “service.exported.configs=osgi.rest”,
  “service.exported.interfaces=com.example.DeviceRESTAccess”
 }
}
public class DeviceREST implements DeviceRESTAccess {
  Map<String,Device> devices= …;
  DTOs dtos = …;

  Collection<Device> getDevices() { return devices.values(); }
  Object       getDevice(String name, String … segments ) {
    Device d = devices.get(name);
    if ( d == null)
      throw new FileNotFoundException(“No such device ” + name);

    return dtos.get(d, segments );
}
```

And voila, Al is ready to go home time for supper.

4.2 Receiving a Measurement with a PUT

Extends 4.1.

Al also needs to receive measurements. He designs the `../measurement/<device>?type=<type>` URI. To provide this URI he adds the following method:

```
interface PutMeasurement extends RESTRequest {
  Measurement _body();
  String type();
}
public void putMeasurement(PutMeasurement  pm, String device) {
  Device d = devices.get(name);
  if ( d == null)
    throw new FileNotFoundException(“No such device ” + name);
  String type = pm.type();
  if ( type == null)
    throw new IllegalArgumentException(“No type specified”);

  List<Measurement> l = d.measurements.get(type);
  if ( l == null ) d.measurements.put( l = new ArrayList<Measurement>());
  l.add( pm._body() );
}
```

Thanks to OSGi, Al has again dinner with his kids!

4.3 JSON RPC

Initially Al used the REST API to get his device information from the server in his Javascript code but he finds that he needs more and more complex options. He also is getting fed up by having to design a URI, implement a XMLHttpRequest for that URI, then write the (although very small and nice of course) server side method. Lots of

Copyright © OSGi Alliance 2015 All Rights Reserved

work and easy to get one part wrong. So he reads about the JSON RPC option. So he writes an additional component (he keeps the REST, it is good for access by external tools) to implement a JSON RPC endpoint:

```
@Component(
 property={
  "service.exported.configs=osgi.jsonrpc",
  "service.exported.interfaces=*"
 },
 service= DeviceManagerFacade.class
}
public class DeviceManagerFacade  {
  Map<String,Device> devices= …;

  public Collection<Device> getDevices() { return devices.values(); }
  public Device getDevice(String name) { return devices.get(name); }
  public void addMeasurement(String device, String type,
     Measurement … msrmnts ) {
    List<Measurement> l = getDevice(device)
      .computeIfAbsent( device, k-> new ArrayList<>());
    Arrays.stream(msrmnts).forEach( (m) → l.add(m) );
  }
}
```

And another day at the beach for AI …

5 Requirements

5.1 General

- G0010 – Ensure compatibility with the Authority API
- G0020 – Ensure that configuration types can be used at resolve time. E.g. a capability namespace.
- G0030 – Define a way for an exported service to detect that it is inside the same security realm or that it must protect itself.

5.2 REST

- R0010 – Provide a Distributed OSGi configuration type that defines the mapping from a Java interface to a REST endpoint that uses JSON as data format.
- R0030 – Provide a type safe way for parameters on the REST URI to be made available to the implementation methods
- R0040 – Define how the URI segments are mapped and converted to the method's parameters

Copyright © OSGi Alliance 2015 All Rights Reserved

- R0050 – Define how the JSON body of a REST request is passed to the method
- R0060 – Define how the returned object is converted to JSON. (For example RFP 169 Object Conversion)
- R0070 – Define on what endpoint the URI will rest.
- R0080 – Allow the service to optionally define the actual endpoint URI
- R0100 – Define an Exception mapping to HTTP response codes
- R0110 – Provide a way to set the return code, return body, and content type.
- R0120 – Allow the REST method access to the underlying Http(s)ServletRequest and response. This will allow the service to handle special cases like returning a non-JSON result.
- R0120 – Provide a mechanism to discover metadata about REST endpoints

5.3 JSON RPC

- J0010 – Provide a Distributed OSGi configuration type that defines the mapping from a Java interface to a JSON RPC endpoint.
- J0030 – Define the mappings from the request's JSON RPC arguments to the method's call parameters. Minimize restriction on usable types.
- J0040 – Define the mapping from the method's returned type to the JSON RPC return value.
- J0050 – Define how exceptions are handled
- J0060 – Define on what endpoint the URI will rest.
- J0070 – Allow the service to define the actual endpoint URI
- J0080 – Provide a mechanism to discover the actual methods available on an endpoint
- J0090 – Provide a mechanism to carry additional information about specific service information during the initial handshake
- J0100 – Allow the service to require a specific protocol version
- J0110 – Provide a Javascript service API to use JSON RPC

5.4 Pluggable Distribution Providers

- P0010 – Provide a service API so that a Distribution Provider can add additional configuration type driver services.
- P0020 – Define how conflicts between configuration types are handled

Copyright © OSGi Alliance 2015 All Rights Reserved

6 Document Support

6.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

6.2 Author's Address

Name	Peter Kriens
Company	aQute
Address	9c, Avenue St. Drezery
Voice	+33 467542167
e-mail	Peter.kriens@aQute.biz

6.3 End of Document

Copyright © OSGi Alliance 2015 All Rights Reserved