

FUSETM Services Framework

Developing RESTful Services

Version 2.1.x
May 2008

Developing RESTful Services

Version 2.1.x

Published 22 Jan 2009

Copyright © 2008 IONA Technologies PLC, a wholly-owned subsidiary of Progress Software Corporation.

Legal Notices

Progress Software Corporation and/or its subsidiaries may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this publication. Except as expressly provided in any written license agreement from Progress Software Corporation, the furnishing of this publication does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Any rights not expressly granted herein are reserved.

Progress, IONA, IONA Technologies, the IONA logo, Orbix, High Performance Integration, Artix, FUSE, and Making Software Work Together are trademarks or registered trademarks of Progress Software Corporation and/or its subsidiaries in the US and other countries.

Java and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the US and other countries. All other trademarks that appear herein are the property of their respective owners.

While the information in this publication is believed to be accurate Progress Software Corporation makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Progress Software Corporation shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

All products or services mentioned in this manual are covered by the trademarks, service marks, or product names as designated by the companies who market those products.

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of IONA Technologies PLC. No third-party intellectual property right liability is assumed with respect to the use of the information contained herein. IONA Technologies PLC assumes no responsibility for errors or omissions contained in this publication. This publication and features described herein are subject to change without notice. Portions of this document may include Apache Foundation documentation, all rights reserved.

Table of Contents

Introduction to RESTful Services	7
Using Automatic REST Mappings	11
Using Java REST Annotations	15
Publishing a RESTful Service	19
Index	23

List of Examples

1. Invalid REST Request	9
2. Wrapped REST Request	9
3. Widget Catalog CRUD Class	11
4. URI Template Syntax	16
5. Using a URI Template	16
6. SEI for a Widget Ordering Service	16
7. WidgetOrdering with REST Annotations	17
8. Setting a Server Factory's Service Class	19
9. Setting Wrapped Mode	19
10. Selecting the REST Binding	20
11. Setting the Base URI	20
12. Setting the Service Invoker	20
13. Publishing the WidgetCatalog Service as a RESTful Endpoint	20

Introduction to RESTful Services

Representational State Transfer (REST) is a software architecture style that centers around the transmission of data over HTTP, using only the four basic HTTP verbs. It also eschews the use of any additional wrappers such as a SOAP envelope and the use of any state data.

Overview

Representational State Transfer (REST) is an architectural style first described in a doctoral dissertation by a researcher named Roy Fielding. In REST, servers expose resources using a URI, and clients access these resources using the four HTTP verbs. As clients receive representations of a resource they are placed in a state. When they access a new resource, typically by following a link, they change, or transition, their state. In order to work, REST assumes that resources are capable of being represented using a pervasive standard grammar.

The World Wide Web is the most ubiquitous example of a system designed on REST principles. Web browsers act as clients accessing resources hosted on Web servers. The resources are represented using HTML or XML grammars that all Web browsers can consume. The browsers can also easily follow the links to new resources.

The advantages of REST style systems is that they are highly scalable and highly flexible. Because the resources are accessed and manipulated using the four HTTP verbs, the resources are exposed using a URI, and the resources are represented using standard grammars, clients are not as affected by changes to the servers. Also, REST style systems can take full advantage of the scalability features of HTTP such as caching and proxies.

Basic REST principles

RESTful architectures adhere to the following basic principles:

- Application state and functionality are divided into resources.
- Resources are addressable using standard URIs that can be used as hypermedia links.
- All resources use only the four HTTP verbs.
 - DELETE
 - GET
 - POST

- PUT
 - All resources provide information using the MIME types supported by HTTP.
 - The protocol is stateless.
 - The protocol is cacheable.
 - The protocol is layered.
-

Resources

Resources are central to REST. A resource is a source of information that can be addressed using a URI. In the early days of the Web, resources were largely static documents. In the modern Web, a resource can be any source of information. For example a Web service can be a resource if it can be accessed using a URI.

RESTful endpoints exchange *representations* of the resources they address. A representation is a document containing the data provided by the resource. For example, the method of a Web service that provides access to a customer record would be a resource, the copy of the customer record exchanged between the service and the consumer is a representation of the resource.

REST best practices

When designing RESTful services it is helpful to keep in mind the following:

- Provide a distinct URI for each resource you wish to expose.

For example, if you are building a system that deals with driving records, each record should have a unique URI. If the system also provides information on parking violations and speeding fines, each type of resource should also have a unique base. For example, speeding fines could be accessed through `/speeding/driverID` and parking violations could be accessed through `/parking/driverID`.

- Use nouns in your URIs.

Using nouns highlights the fact that resources are things and not actions. URIs such as `/ordering` imply an actions, whereas `/orders` implies a thing.

- Methods that map to `GET` should not change any data.
- Use links in your responses.

Putting links to other resources in your responses makes it easier for clients to follow a chain of data. For example, if your service returns a collection of resources, it would be easier for a client to access each of the individual resources using the provided links. If links are not included, a client needs to have additional logic to follow the chain to a specific node.

- Make your service stateless.

Requiring the client or the service to maintain state information forces a tight coupling between the two. Tight couplings make upgrading and migrating more difficult. Maintaining state can also make recovery from communication errors more difficult.

Wrapped mode vs. unwrapped mode

RESTful services can only send or receive one XML element. To enable the mapping of methods that use more than one parameter, FUSE Services Framework can use *wrapped mode*. In wrapped mode, FUSE Services Framework wraps the parameters with a root element derived from the operation name. For example, the operation `Car findCar(String make, String model)` could not be mapped to an XML `POST` request like the one shown in [Example 1 on page 9](#).

Example 1. Invalid REST Request

```
<name>Dodge</name>
<model>Daytona</company>
```

[Example 1 on page 9](#) is invalid because it has two root XML elements, which is not allowed. Instead, the parameters would have to be wrapped with the operation name to make the `POST` valid. The resulting request is shown in [Example 2 on page 9](#).

Example 2. Wrapped REST Request

```
<findCar>
  <make>Dodge</make>
  <model>Daytona</model>
</findCar>
```

By default, FUSE Services Framework uses unwrapped mode, because, for cases where operations use a single parameter, it creates prettier XML. Using unwrapped mode, however, requires that you constrain your service interfaces to sending and receiving single elements. If your operation needs to take multiple parameters, you must combine them in an object. With the

`findCar()` example above, you would want to create a `FindCar` class that holds the make and model data.

Implementing REST with FUSE Services Framework

FUSE Services Framework uses an HTTP binding to map Java interfaces into RESTful services. There are two ways to map the methods of the Java interface into resources:

- Convention based mapping (see [Using Automatic REST Mappings on page 11](#))
- Java REST annotations (see [Using Java REST Annotations on page 15](#))

Using Automatic REST Mappings

To simplify the creation of RESTful services, FUSE Services Framework can automatically map a Java service to a RESTful interface. The mapping requires that the Java service is defined as a CRUD based class.

Overview

To simplify the creation of RESTful service endpoints, FUSE Services Framework can map the methods of a CRUD (Create, Read, Update, and Destroy) based Java bean class to URIs automatically. The mapping looks for keywords in the method names of the bean, such as get, add, update, or remove, and maps them onto HTTP verbs. It then uses the remainder of the method name to create a URI by pluralizing the field name and appending it to the base URI at which the endpoint is published.



Note

For more information about publishing RESTful endpoints, see [Publishing a RESTful Service on page 19](#).

Typical CRUD class

[Example 3 on page 11](#) shows a CRUD based class for updating a catalog of widgets.

Example 3. Widget Catalog CRUD Class

```
import javax.jws.WebService;

@WebService
public interface WidgetCatalog
{
    Collection<Widget> getWidgets();
    Widget getWidget(long id);
    void addWidget(Widget widget);
    void updateWidget(Widget widget);
    void removeWidget(String type, long num);
    void deleteWidget(Widget widget);
}
```



Important

You must use the `@WebService` annotation on any class or interface that you wish to expose as a RESTful endpoint.

The class has six operations that are mapped to a URI/verb pair:

- `getWidgets()` is mapped to a GET at `baseURI/widgets`.
 - `getWidget()` is mapped to a GET at `baseURI/widgets/id`.
 - `addWidget()` is mapped to a POST at `baseURI/widgets`.
 - `updateWidget()` is mapped to a PUT at `baseURI/widgets`.
 - `removeWidget()` is mapped to a DELETE at `baseURI/widgets/type/num`.
 - `deleteWidget()` is mapped to a DELETE at `baseURI/widgets`.
-

Mapping to GET

When FUSE Services Framework sees a method name in the form of `getResource()`, it maps the method to a GET. The URI is generated by appending the plural form of `Resource` to the base URI at which the endpoint is published. If `Resource` is already plural, it is not pluralized. For example, `getCustomer()` is mapped to a GET on `/customers`. The method `getCustomers()` would result in the same mapping.

Any method parameters are appended to the URI. For example, `getWidget(long id)` is mapped to `/widgets/id` and `getCar(String make, String model)` would be mapped to `/cars/make/model`. A call to `getCar(plymouth, roadrunner)` would be executed by a GET to `/cars/plymouth/roadrunner`.



Important

FUSE Services Framework only supports get methods that use XML primitives in their parameter list.

Mapping to POST

Methods of the form `addResource()` or `createResource()` are mapped to POST. The URI is generated by pluralizing *Resource*. For example `createCar(Car car)` would be mapped to a POST at `/cars`.

Mapping to PUT

Methods of the form `updateResource()` are mapped to PUT. The URI is generated by pluralizing *Resource* and appending any parameters except the resource to be updated. For example `updateHitter(long number, long rotation, Hitter hitter)` would be mapped to a PUT at `/hitters/number/rotation`.



Important

FUSE Services Framework only supports get methods that use XML primitives in their parameter list.

Mapping to DELETE

Methods of the form `deleteResource()` or `removeResource()` are mapped to DELETE. The URI is generated by pluralizing *Resource* and appending any parameters. For example `removeCar(String make, long num)` would be mapped to a DELETE at `/cars/make/num`.



Important

FUSE Services Framework only supports get methods that use XML primitives in their parameter list.

Using Java REST Annotations

FUSE Services Framework recognizes a set of annotations that allow you to dictate the mappings of Java operations to a RESTful interface.

Overview

While the convention-based REST mappings provide an easy way to create a service that maintains a collection of data, or looks like it does, it does not provide the flexibility to create a full range of RESTful services that require operations whose names don't fit into the CRUD format. FUSE Services Framework provides a collection of annotations that allows you to define the mapping of an operation to an HTTP verb/URI combination. The REST annotations allow you to specify which verb to use for an operation and to specify a template for creating a URI for the exposed resource.

Specifying the HTTP verb

FUSE Services Framework uses four annotations for specifying the HTTP verb that will be used for a method:

- `org.codehaus.jra.Delete` specifies that the method maps to a `DELETE`.
- `org.codehaus.jra.Get` specifies that the method maps to a `GET`.
- `org.codehaus.jra.Post` specifies that the method maps to a `POST`.
- `org.codehaus.jra.Put` specifies that the method maps to a `PUT`.

When you map your methods to HTTP verbs, you must ensure that the mapping makes sense. For example, if you map a method that is intended to submit a purchase order, you would map it to a `PUT` or a `POST`. Mapping it to a `GET` or a `DELETE` would result in unpredictable behavior.

Specifying the URI

You specify the URI of the resource using the `org.codehaus.jra.HttpResource` annotation. `HttpResource` has one required attribute, `location`, that specifies the location of the resource in relationship to the base URI specified when publishing the service (see [Publishing a RESTful Service on page 19](#)). For example, if you specify `carts` as the location of the resource and the base URI is

`http://myexample.iona.org`, the full URI for the resource will be
`http://myexample.iona.org/carts`.

Using URI templates

In addition to specifying hard coded resource locations, FUSE Services Framework provides a facility for creating URIs on the fly using either the method's parameters or a field from the JAXB bean in the parameter list. When providing a value for the `HttpResource` annotation's `location` parameter you provide a URI template using the syntax in [Example 4 on page 16](#).

Example 4. URI Template Syntax

```
@HttpResource(location="resourceName/{param1}/../{paramN}")
```

`resourceName` can be any valid string, and forms the base of the location. Each `param` is the name of either a method parameter or a field in the JAXB bean in the parameter list. To create the URI, FUSE Services Framework replaces `param` with the value of the associated parameter. For example, if you have the method shown in [Example 5 on page 16](#) and wanted to access the record at id 42, you would perform a GET at

```
http://myexample.iona.com/records/42.
```

Example 5. Using a URI Template

```
@Get
@HttpResource(location="\records\{id}")
Record fetchRecord(long id);
```



Important

FUSE Services Framework only supports XML primitives in URI templates.

Example

If you wanted to implement a system for ordering widgets out of the catalog defined by [Example 3 on page 11](#) you may use an SEI like the one shown in [Example 6 on page 16](#).

Example 6. SEI for a Widget Ordering Service

```
@WebService
public interface WidgetOrdering
{
```

```
void placeOrder(WidgetOrder order);
OrderStatus checkOrder(long orderNum);
void changeOrder(WidgetOrder order, long orderNum);
void cancelOrder(long orderNum);
}
```

`WidgetOrdering` does not match any of the naming conventions outlined in [Using Automatic REST Mappings on page 11](#) so the RESTful binding cannot automatically map the methods to verb/URI combinations. You will need to provide the mappings using the Java REST annotations. To do this, you need to consider what each method in the interface does and how it correlates to one of the HTTP verbs:

- `placeOrder()` creates a new order on the system. Resource creation correlates with `POST`.
- `checkOrder()` looks up an order's status and returns it to the user. Returning resources correlates with `GET`.
- `changeOrder()` updates an order that has already been placed. Updating an existing record correlates with `PUT`.
- `cancelOrder()` removes an order from the system. Removing a resource correlates with `DELETE`.

For the URI, you would use a resource name that hinted at the purpose of the resource. For this example, the resource name used is `orders` because it is assumed that the base URI at which the endpoint is published provides information about what is being ordered. For the methods that use `orderNum` to identify a particular order, URI templating is used to append the value of the parameter to the end of the URI.

[Example 7 on page 17](#) shows `WidgetOrdering` with the required annotations.

Example 7. *WidgetOrdering* with REST Annotations

```
import org.codehaus.jra.*;

@WebService
public interface WidgetOrdering
{
```

```
@Post
@HttpResource(location="\orders")
void placeOrder(WidgetOrder order);

@Get
@HttpResource(location="\orders\{orderNum}")
OrderStatus checkOrder(long orderNum);

@Put
@HttpResource(location="\orders\{orderNum}")
void changeOrder(WidgetOrder order, long orderNum);

@Delete
@HttpResource(location="\orders\{orderNum}")
void cancelOrder(long orderNum);
}
```

To check the status of order number 236, you would perform a `GET` at `baseURI/orders/236`.

Publishing a RESTful Service

The FUSE Services Framework APIs provide a simple means of publishing a RESTful service using the `JaxWsServiceFactoryBean`.

Overview

You publish RESTful services using the `JaxWsServerFactoryBean` object. Using the `JaxWsServerFactoryBean` object, you specify the base URI for the resources implemented by the service and whether the resources use wrapped messages. You can then create a `Server` object to start listening for requests to access the service's resources.

Procedure

To publish your RESTful service, do the following:

1. Create a new `JaxWsServerFactoryBean`.
2. Set the server factory's service class to the class of your RESTful service's SEI using the factory's `setServiceClass()` method as shown in [Example 8 on page 19](#).

Example 8. Setting a Server Factory's Service Class

```
// Service factory sf obtained previously
sf.setServiceClass(widgetService.class);
```

3. If you want to use wrapped mode, set the factory's wrapped property to `true` using the `setWrapped()` method as shown in [Example 9 on page 19](#).

Example 9. Setting Wrapped Mode

```
sf.getServiceFactory().setWrapped(true);
```



Note

For more information about using wrapped mode or unwrapped mode, see [Wrapped mode vs. unwrapped mode on page 9](#).

4. Set the server factory's binding to the REST binding using the `setBindingId()` method.

The REST binding is selected using the constant `HttpBindingFactory.HTTP_BINDING_ID` as shown in

[Example 10 on page 20](#).

Example 10. Selecting the REST Binding

```
// Server factory sf obtained previously
sf.setBindingId(HttpBindingFactory.HTTP_BINDING_ID);
```

5. Set the base URI for the service's resources using the `setAddress()` method as shown in [Example 11 on page 20](#).

Example 11. Setting the Base URI

```
sf.setAddress("http://localhost:9000");
```

6. Set server factory's service invoker to an instance of your service's implementation class as shown in [Example 12 on page 20](#).

Example 12. Setting the Service Invoker

```
widgetService service = new widgetServiceImpl();
sf.getServiceFactory().setInvoker(new BeanInvoker(service));
```

7. Create a new `Server` object from the server factory using the factory's `create()` method.

Example

[Example 13 on page 20](#) shows the code for publishing a RESTful service at `http://jfu:9000`. All of the resources implemented by the service will use the published URI as the base address.

Example 13. Publishing the WidgetCatalog Service as a RESTful Endpoint

```
JaxWsServerFactoryBean sf = new JaxWsServerFactoryBean();
sf.setServiceClass(WidgetCatalog.class);

sf.setBindingId(HttpBindingFactory.HTTP_BINDING_ID);
sf.setAddress("http://jfu:9000");

widgetService service = new WidgetCatalogImpl();
sf.getServiceFactory().setInvoker(new BeanInvoker(service));
```

```
Server svr = sf.create();
```

If you used [Example 13 on page 20](#) to publish the service defined by [Example 3 on page 11](#), you would:

- Retrieve a list of all widgets in the catalog using a `GET` at `http://jfu:9000/widgets`.
- Retrieve information about widget 34 using a `GET` at `http://jfu:9000/widgets/34`.
- Modify a widget using a `PUT` at `http://jfu:9000/widgets` with an XML document describing the widget to modify.
- Delete 15 round widgets from the catalog using a `DELETE` at `http://jfu:9000/widgets/round/15`.

Index

Symbols

@Delete, 15
@Get, 15
@HttpResource, 15
@Post, 15
@Put, 15

A

annotations
 @Delete (see @Delete)
 @Get (see @Get)
 @HttpResource (see @HttpResource)
 @Post (see @Post)
 @Put (see @Post)

D

deploying
 RESTful service endpoint, 19

G

getResource(), 12

H

HTTP
 DELETE, 13, 15
 GET, 12, 15
 POST, 13, 15
 PUT, 13, 15

P

publishing
 RESTful service endpoint, 19

R

REST binding
 activating, 19

S

setAddress(), 20
setBindingId(), 19
setServiceClass(), 19
setWrapped(), 19

W

wrapped mode, 9
 activating, 19

