



FUSE Services Framework

Developing Applications with Dynamic Languages

Version 2.0
December 2007

Making Software Work Together™

Developing Applications with Dynamic Languages

IONA Technologies

Version 2.0

Published 19 Mar 2008

Copyright © 2001-2008 IONA Technologies PLC

Trademark and Disclaimer Notice

IONA Technologies PLC 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 IONA Technologies PLC, 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.

IONA, IONA Technologies, the IONA logo, Orbix, High Performance Integration, Artix, FUSE, and Making Software Work Together are trademarks or registered trademarks of IONA Technologies PLC and/or its subsidiaries.

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 United States 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, IONA Technologies PLC 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. IONA 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.

Copyright Notice

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

Using ECMAScript to Implement Services	9
Implementing a Service in JavaScript	10
Defining the Metadata	11
Implementing the Service Logic	13
Implementing a Service in ECMAScript for XML (E4X)	14
Publishing Services Developed in a Dynamic Language	15
Deploying JavaScript Services	16
Index	19

List of Tables

1. Optional Arguments to <code>ServerApp</code>	16
---	----

List of Examples

1. JavaScript Web Service Metadata	12
2. JavaScript Service Implementation	13
3. E4X Service Implementation	14
4. Deploying a Service at a Specified Address	17
5. Deploying a Group of Services to a Base Address	17
6. Combining the Command Line Arguments	17

Using ECMAScript to Implement Services

Summary

JavaScript, also known by its formal name ECMAScript, is one of the many dynamic languages that are growing in prevalence in development environments. It provides a quick and lightweight means of creating functionality that can be run on a number of platforms. Another strength of JavaScript is that applications can be quickly rewritten.

Table of Contents

Implementing a Service in JavaScript	10
Defining the Metadata	11
Implementing the Service Logic	13
Implementing a Service in ECMAScript for XML (E4X)	14

FUSE Services Framework provides support for developing services using JavaScript and ECMAScript for XML(E4X). The pattern used to develop these services are similar to JAX-WS `Provider` implementations that handle their requests and responses (either SOAP messages or SOAP payloads) as DOM documents.

Implementing a Service in JavaScript

Table of Contents

Defining the Metadata	11
Implementing the Service Logic	13

Writing a service in JavaScript is a two step process:

1. Define the JAX-WS style metadata.
2. Implement the service's business logic.

Defining the Metadata

Java based service providers typically use annotations to specify JAX-WS metadata. Since JavaScript does not support annotations, you use ordinary JavaScript variables to specify metadata for JavaScript implementations. FUSE Services Framework treats any JavaScript variable in your code whose name equals or begins with `WebServiceProvider` as a JAX-WS metadata variable.

Required properties

Properties of the variable are expected to specify the same metadata that the JAX-WS `WebServiceProvider` annotation specifies, including:

- `wSDLLocation` specifies a URL for the WSDL document that defines the service.
 - `serviceName` specifies the name of the service.
 - `portName` specifies the service's port/interface name.
 - `targetNamespace` specifies the target namespace of the service.
-

Optional properties

The JavaScript `WebServiceProvider` can also specify the following optional properties:

- `ServiceMode` indicates whether the specified service handles SOAP payload documents or full SOAP message documents. This property mimics the JAX-WS `ServiceMode` annotation. The default value is `PAYLOAD`.
 - `BindingMode` indicates the service binding ID URL. The default is the SOAP 1.1/HTTP binding.
 - `EndpointAddress` indicates the URL consumer applications use to communicate with this service. The property is optional but has no default.
-

Example

Example 1, “JavaScript Web Service Metadata” shows a metadata description for a JavaScript service implementation.

Example 1. JavaScript Web Service Metadata

```
var WebServiceProvider1 = {  
  'wsdlLocation': 'file:./wsdl/hello_world.wsdl',  
  'serviceName': 'SOAPService1',  
  'portName': 'SoapPort1',  
  'targetNamespace': 'http://object  
web.org/hello_world_soap_http',  
};
```

Implementing the Service Logic

You implement the service's logic using the required `invoke` property of the `WebServiceProvider` variable. This property is a function that accepts one input argument, a `javax.xml.transform.dom.DOMSource` node, and returns a document of the same type. The `invoke` function can manipulate either the input or output documents using the regular Java `DOMSource` class interface just as a Java application would.

Example

Example 2, “JavaScript Service Implementation” shows an `invoke` function for a simple JavaScript service implementation.

Example 2. JavaScript Service Implementation

```
WebServiceProvider.invoke = function(document) {
    var ns4 = "http://apache.org/hello_world_soap_http/types";
    var list = document.getElementsByTagNameNS(ns4, "requestType");
    var name = list.item(0).getFirstChild().getNodeValue();
    var newDoc = document.getImplementation().createDocument(ns4, "ns4:greetMeResponse",
null);
    var el = newDoc.createElementNS(ns4, "ns4:responseType");
    var txt = newDoc.createTextNode("Hi " + name);
    el.insertBefore(txt, null);
    newDoc.getDocumentElement().insertBefore(el, null);
    return newDoc;
}
```

Implementing a Service in ECMAScript for XML (E4X)

Developing a service using E4X is very similar to developing a service using JavaScript. You define the JAX-WS metadata using the same `WebServiceProvider` variable in JavaScript. You also implement the service's logic in the `WebServiceProvider` variable's `invoke` property.

The only difference between the two approaches is the type of document the implementation manipulates. When working with E4X, the implementation receives requests as an E4X XML document and returns a document of the same type. These documents are manipulated using built-in E4X XML features.

Example

Example 3, “E4X Service Implementation” shows an `invoke` function for a simple E4X service implementation.

Example 3. E4X Service Implementation

```
var SOAP_ENV = new Namespace('SOAP-ENV',
                             'http://schemas.xmlsoap.org/soap/envelope/');
var xs = new Namespace('xs', 'http://www.w3.org/2001/XMLSchema');
var xsi = new Namespace('xsi', 'http://www.w3.org/2001/XMLSchema-instance');
var ns = new Namespace('ns', 'http://apache.org/hello_world_soap_http/types');

WebServiceProvider1.invoke = function(req) {
    default xml namespace = ns;
    var name = (req.requestType)[0];
    default xml namespace = SOAP_ENV;
    var resp = <SOAP-ENV:Envelope xmlns:SOAP-ENV={SOAP_ENV} xmlns:xs={xs} xmlns:xsi={xsi}/>;

    resp.Body = <Body/>;
    resp.Body.ns::greetMeResponse = <ns:greetMeResponse xmlns:ns={ns}/>;
    resp.Body.ns::greetMeResponse.ns::responseType = 'Hi ' + name;
    return resp;
}
```

Publishing Services Developed in a Dynamic Language

Summary

Most dynamic languages require an interpreter to run. FUSE Services Framework provides a lightweight container for hosting services developed using dynamic languages.

Table of Contents

Deploying JavaScript Services 16

Exposing a scripted service through FUSE Services Framework's runtime is handled by a lightweight container. The container loads the required runtime interpreters for the service, runs the code, and connects the application's logic to the underlying runtime. The scripted services can take advantage of most of the features offered by the runtime through the container.

Deploying JavaScript Services

FUSE Services Framework provides a lightweight container that allows you to deploy your JavaScript and E4X services and take advantage of FUSE Services Framework's pluggable transport infrastructure.

Important

JavaScript based services work with SOAP messages. So, while they are multi-transport, they can only use the SOAP binding.

Deployment command

You deploy them into the container using the following command:

```
java org.apache.cxf.js.rhino.ServerApp [ -a addressURL ] [ -b  
baseAddressURL ] { file ...}
```

The `org.apache.cxf.js.rhino.ServerApp` class, shorted to `ServerApp` below, takes one or more JavaScript files, suffixed with a `.js`, or E4X files, suffixed with a `.jsx`, and loads them into the FUSE Services Framework runtime. If `ServerApp` locates JAX-WS metadata in the files it creates and registers a `JAX-WS Provider<DOMSource>` object for each service. The `Provider<DOMSource>` object delegates the processing of requests to the implementation stored in the associated file. `ServerApp` can also take the name of a directory containing JavaScript and E4X files. It will load all of the scripts that contain JAX-WS metadata, load them, and publish a service endpoint for each one.

`ServerApp` has three optional arguments:

Table 1. Optional Arguments to `serverApp`

Argument	Description
<code>-a <i>addressURL</i></code>	Specifies the address at which <code>ServerApp</code> publishes the service endpoint implementation found in the script file following the URL.

Argument	Description
<code>-b baseAddressURL</code>	Specifies the base address used by <code>ServerApp</code> when publishing the service endpoints defined by the script files. The full address for the service endpoints is formed by appending the service's port name to the base address.
<code>-v</code>	Specifies that <code>ServerApp</code> is to run in verbose mode.

The optional arguments take precedence over any addressing information provided in `EndpointAddress` properties that appear in the JAX-WS metadata.

Examples

For example, if you deployed a JavaScript service using the command shown in Example 4, “Deploying a Service at a Specified Address”, your service would be deployed at `http://cxf.apache.org/goodness`.

Example 4. Deploying a Service at a Specified Address

```
java org.apache.cxf.js.rhino.ServerApp -a http://cxf.apache.org/goodness hello_world.jsx
```

To deploy a number of services using a common base URL you could use the command shown in Example 5, “Deploying a Group of Services to a Base Address”. If the service defined by `hello_world.jsx` had port name of `helloWorld`, `ServerApp` would publish it at

`http://cxf.apache.org/helloWorld`. If the service defined by `goodbye_moon.js` had a port name of `blue`, `ServerApp` would be published at `http://cxf.apache.org/blue`.

Example 5. Deploying a Group of Services to a Base Address

```
java org.apache.cxf.js.rhino.ServerApp -b http://cxf.apache.org hello_world.jsx goodbye_moon.js
```

You can also combine the arguments as shown in Example 6, “Combining the Command Line Arguments”. Your service would be deployed at `http://cxf.apache.org/goodness`. `ServerApp` would publish three service endpoints:

Example 6. Combining the Command Line Arguments

```
java org.apache.cxf.js.rhino.ServerApp -b http://cxf.apache.org hello_world.jsx goodbye_moon.js -a http://cxf.apache.org/goodness chocolate.jsx
```

1. The service defined by `hello_world.jsx` at
`http://cxf.apache.org/helloWorld.`
2. The service defined by `goodbye_moon.js` at
`http://cxf.apache.org/blue.`
3. The service defined by `chocolate.jsx` at
`http://cxf.apache.org/goodness.`

Index

B

BindingMode property, 11

D

deploying, 16

DOMSource, 13

E

endpoint

 specifying the address, 11

EndpointAddress property, 11

I

invoke(), 13, 14

J

JAX-WS

 WebServiceProvider annotation, 11

M

message manipulation, 13, 14

P

portName property, 11

S

ServerApp, 16

service metadata, 11

 optional, 11

 required, 11

ServiceMode property, 11

serviceName property, 11

T

targetNamespace property, 11

W

WebServiceProvider variable, 11

wsdlLocation property, 11

X

XML documents, 14

