Artix ESB

Developing Applications in JavaScript

Version 5.1
December 2007

Making Software Work Together™

Developing Applications in JavaScript

IONA Technologies

Version 5.1

Published 28 Mar 2008

Trademark and Disclaimer Notice

Copyright Notice

Table of Contents

List of Tables

List of Examples

Preface

Table of Contents

What is Covered in This Book

This book describes how to use the Artix ESB APIs to develop applications.

Who Should Read This Book

This book is intended for developers using Artix ESB. It assumes that you have a good understanding of the following:

- general programming concepts.
- general SOA concepts.
- JavaScript.
- the runtime environment into which you are deploying services.

How to Use This Book

This book is organized so that it follows the general workflow for developing and deploying services with Artix ESB. It begins with a discussion of implementing your services, progresses through how to set up the physical details of how your service will be exposed as an endpoint, and concludes by discussing how to deploy endpoints into Artix ESB.

The Artix ESB Documentation Library

For information on the organization of the Artix ESB library, the document conventions used, and where to find additional resources, see Using the Artix ESB Library [http://www.iona.com/support/docs/artix/5.1/library_intro/index.htm].

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

Artix ESB 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

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. Artix ESB 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. Artix ESB provides a lightweight container for hosting services developed using dynamic languages.

Table of Contents

Deploying JavaScript Services

Artix ESB provides a lightweight container that allows you to deploy your JavaScript and E4X services and take advantage of Artix ESB'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 Artix ESB 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
`-a addressURL`	Specifies the address at which `ServerApp` publishes the service endpoint implementation found in the script file following the URL.
`-b baseAddressURL`	Specifies the base address used by `ServerApp` 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.

Argument	Description
-v	Specifies that ServerApp 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