# JavaScript

## Using Dynamic Languages to Implement Services

### Overview

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.

CXF provides support for developing services using JavaScript and ECMAScript for XML(E4X). The patterns 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

Writing a service in JavaScript is a two step process:

1. Define the JAX-WS style metadata.
2. Implement the services business logic.

#### Defining the Metadata

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

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 defining 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.

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 1 shows a metadata description for a JavaScript service implementation.

**Example 1:JavaScript Metadata**

```
var WebServiceProvider1 = {
    'wsdlLocation': 'file:./wsdl/hello_world.wsdl',
    'serviceName': 'SOAPService1',
    'portName': 'SoapPort1',
    'targetNamespace': 'http://apache.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 variable 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 2 shows an `invoke` property 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)

Writing a CXF service using E4X is very similar to writing 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 3 shows an `invoke` property 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;
}
```

## Deploying Scripted Services

CXF provides a lightweight container that allows you to deploy your Javascript and E4X services and take advantage of CXF's pluggable transport infrastructure.



**Note**

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

You deploy services into the container using the following command:

```
java org.apache.cxf.js.rhino.ServerApp [ -a addressURL ] [ -b baseAddressURL ] file.js [ file2.js file3.jsx ...
]
```

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 CXF 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:

| 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. |
| `-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.

For example, if you deployed a JavaScript service using the command shown in Example 4, 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. 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 publish 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 and your service would be deployed at `http://cxf.apache.org/goodness`. `ServerApp` would publish three service endpoints:

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

**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
```