Using Dynamic Languages to Implement Services

Table of Contents

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.

Celtix 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

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. Celtix 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.

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

Example 1: JavaScript WebServiceProvider Metadata

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.

```
WebServiceProvider.invoke = function(document) {
    var ns4 = "http://objectweb.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;
}
```

Example 2: JavaScript Service Implementation

Implementing a Service in ECMAScript for XML(E4X)

Writing a Celtix 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.

```
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://objectweb.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;
}
```

Example 3: E4X Service Implementation