PROGRESS SOFTWARE

Artix® ESB

Developing Applications in JavaScript

Version 5.5
December 2008

Developing Applications in JavaScript

Version 5.5

Published 15 Dec 2008
Copyright © 2008 IONA Technologies PLC, a wholly-owned subsidiary of Progress Software Corporation.

Table of Contents

List of Tables

List of Examples

Preface

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 workf low 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].

Implementing a Service in JavaScript

JavaSript is a dynamic and lightweight language growing in popularity. It enables developers to quickly create functionality that runs on a large number of platforms.

Artix ESB supports services written in JavaScript. The pattern used to develop these services is similar to JAX-WS `Provider` implementations that handle

requests and responses (either SOAP messages or SOAP payloads) as DOM documents.

To write a service in JavaScript, you must:

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 must use ordinary JavaScript variables to specify metadata for JavaScript implementations. Artix ESB treats any JavaScript variable with a name that 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 and has no default.

Example

Example 1 on page 17 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

Overview

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 the output documents using the regular Java `DOMSource` class interface, just as a Java application would.

Example

Example 2 on page 18 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)

You can develop a service using ECMAScript for XML (E4X), much like you can do with JavaScript.

Overview

Developing a service using E4X is 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

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;
}
```

Developing Service Consumers

Artix ESB provides a number of tools for writing service consumers in JavaScript. These include generating code from existing applications and downloading JavaScript from Artix ESB-based services.

Artix ESB JavaScript client-side support allows you to create JavaScript service consumers that can communicate natively with SOAP/HTTP service providers. The code generators produce proxy code and support classes for communicating directly with a service provider. Using the generated code, you can use JavaScript to build Web applications that access the back-end services. The consumers use asynchronous communication to access the services, making interaction as smooth as possible.

To develop service consumers in JavaScript, you must:

1. Generate the proxy and support code

2. Implement the callback functions used by the client

3. Invoke the service's operations

Generating Consumer Code

Overview

Artix ESB provides three mechanisms for creating client-side JavaScript code for a service. If you have a running service that was built using either Artix ESB or Apache CXF, you can access the client-side JavaScript code dynamically. If you want to start from a WSDL document, you can use the **wsdl2js** tool. If you want to start from a Java SEI, you can use the **java2ws** command.

When dealing with the command line code generators, you are responsible for ensuring that the generated supported code and the Artix ESB JavaScript utility code are available to any Web application that uses it. The dynamically generated support code is available whenever the service is active.

Dynamic access

If the service you want to access is developed using Artix ESB, you can dynamically access the proxy and support code for making remote invocations using the `?js` URI handler. For example, if your service provider's address is `http://my.widgets.example/WidgetService` you would access the JavaScript proxy code using the URI `http://my.widgets.example/WidgetService?js`.

Example 4 on page 22 shows a fragment of HTML used to access the JavaScript proxy code for the service on which requests are made.

Example 4. HTML for Dynamically Accessing JavaScript Proxy Code

```
<script type="text/javascript" src="/WidgetService?js"></script>
```

When using the standard `cxf.xml` file or `cxf-servlet.xml` file to configure your service provider, the `?js` URI handler is automatically loaded by the Artix ESB's bus. If you provide your own Spring configuration file, include `META-INF/cxf/cxf-extension-javascript-client.xml` in your bean configuration.

For more information about configuring the Artix ESB bus, see Artix® ESB Deployment Guide.

Starting from WSDL

If you are developing a JavaScript consumer from a WSDL document and you want to store the support code on a machine that is local to the consumer, you can use the **wsdl2js** command. This command takes a WSDL document containing a valid interface definition and generates the JavaScript support

code needed to make invocations on a service provider implementing the interface. The support code is generated into a single file.

Example 5 on page 23 shows how to call the **wsdl2js** command.

Example 5. Calling the WSDL to JavaScript Tool

```
wsdl2js widgets.wsdl
```

⚠ **Important**

> The generated code does not contain the Artix ESB JavaScript utility code. For more information see Artix ESB utility code on page 24.

For more information on using the **wsdl2js** command see artix wsdl2js in the *Artix® ESB Command Reference*.

Starting from a Java SEI

If you are developing a JavaScript consumer from a Java SEI and you want to store the support code on a machine that is local to the consumer, you can use the **java2js** command. This command takes a compiled Java SEI, and all of its supporting type classes, and generates the JavaScript support code needed to make invocations on a service provider implementing the interface. The support code will be generated into a single file.

Example 6 on page 23 shows how to call the **java2js** command.

Example 6. Calling the Java to JavaScript Tool

```
java2js org.widgets.examples.WidgetService
```

⚠ **Important**

> By default, the tool does not add the Artix ESB JavaScript utility code to the generated code. However, you can use the -jsutils option
>
> to force the tool to add the utility code. For more information see Artix ESB utility code on page 24.

For more information on using the **java2js** command see artix java2js in the *Artix® ESB Command Reference*.

Understanding the Generated Code

Overview

The code used to support the Artix ESB JavaScript consumers falls into three catagories:

Artix ESB utility code

The utility code provides the hooks needed to run inside a browser. It also provides some XML management functions.

Schema generated type objects

The type objects serialize and deserialize the SOAP messages used by the service.

Service proxy code

The proxy code is used to make requests on the remote service. It includes methods for each operation defined by the interface.

Artix ESB utility code

JavaScript consumers require a set of JavaScript utility functions to run in a Web browser. They also require a few XML management functions. These utility functions are supplied by Artix ESB.

When you access the dynamically generated proxy code using the `?js` URI handler, the utility code is automatically included in the response. If you do not want to incur the penalty for downloading the additional JavaScript, you can access the proxy code by appending `?nojsutils`.

The command line code generators do not include the utility code. However, the **java2js** command's `-jsutils` does include the utility code in the generated proxy code.

If the utility code is not included in the JavaScript proxy code you are using to develop your consumer, then you must include the utility code. The utility functions are included in the `InstallDir/etc/cxf-utils.js` file. Package this file with the other JavaScript code used to implement your consumer.

Generated types

The JavaScript code generators create objects to support all of the schema types used by your service. The generated objects have names derived from their QName. For example, the schema type `http://widgets.examples.com/types/WidgetOrder` is named `widgets_example_com_types_WidgetOrder`.

Each generated type object has a constructor. It also has a getter and setter for each property defined by the schema type. If the service's Java implementation of the schema type has private properties, they are identified by placing an underscore(_) in front of their names.

The code generator creates the JavaScript object shown in Example 8 on page 25 for the schema type shown in Example 7 on page 25.

Example 7. Schema for JavaScript Type Object

```
<definitions ...>
  <types>
    <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
            xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
            ...>
        <complexType name="numInventory">
          <sequence>
            <element name="numLeft" type="xsd:int" />
            <element name="size" type="xsd:string" />
          </sequence>
        </complexType>
    </schema>
  </types>
  ...
</definitions>
```

The schema type in Example 7 on page 25 is a basic sequence complex type with two fields: numLeft and size. The generated object contains getters and setters for both fields. The name generated for the object is also shortened to use the namespace prefix xsd1 as the root.

Example 8. JavaScript Type Object

```
//
// Constructor for XML Schema item {http://widgetVendor.com/types/widgetTypes}numInventory
//
function XSD1_numInventory () { ❶
    this.typeMarker = 'XSD1_numInventory';
    this._numLeft = 0;
    this._size = '';
}


//
// accessor is XSD1_numInventory.prototype.getNumLeft
// element get for numLeft
// - element type is {http://www.w3.org/2001/XMLSchema}int
// - required element
//
```

```
// element set for numLeft
// setter function is is XSD1_numInventory.prototype.setNumLeft
//
function XSD1_numInventory_getNumLeft() { return this._numLeft;} ❷

XSD1_numInventory.prototype.getNumLeft = XSD1_numInventory_getNumLeft;

function XSD1_numInventory_setNumLeft(value) { this._numLeft = value;} ❸

XSD1_numInventory.prototype.setNumLeft = XSD1_numInventory_setNumLeft;
//
// accessor is XSD1_numInventory.prototype.getSize
// element get for size
// - element type is {http://www.w3.org/2001/XMLSchema}string
// - required element
//
// element set for size
// setter function is is XSD1_numInventory.prototype.setSize
//
function XSD1_numInventory_getSize() { return this._size;} ❹

XSD1_numInventory.prototype.getSize = XSD1_numInventory_getSize;

function XSD1_numInventory_setSize(value) { this._size = value;} ❺

XSD1_numInventory.prototype.setSize = XSD1_numInventory_setSize;
```

The generated code in Example 8 on page 25 has the following features:

❶   A constructor for the `numInventory` object

In addition to fields for the two defined properties, the generated object
has a typeMarker field. The typeMarker field is used to assist in serializing
and deserializing the SOAP messages.

❷   A getter method for the numLeft property

❸   A setter method for the numLeft property

❹   A getter method for the size property

❺   A setter method for the size property

Service proxy code

The code generators create a service proxy for the interface defined by the
service. Example 9 on page 27 shows a generated constructor for a service
proxy.

Example 9. JavaScript Service Proxy Constructor

```
function widgetVendor_com_widgetOrderForm_orderWidgets () {
    this.jsutils = new CxfApacheOrgUtil();
    this.jsutils.interfaceObject = this;
    this.synchronous = false;
    this.url = null;
    this.client = null;
    this.response = null;
    this._onsuccess = null;
    this._onerror = null;
    this.globalElementSerializers = [];
    this.globalElementDeserializers = [];
    this.globalElementSerializers['{http://widgetVendor.com/types/widgetTypes}restrictedAd
dress'] = XSD1_restrictedAddress_serialize;
    this.globalElementDeserializers['{http://widgetVendor.com/types/widgetTypes}restricte
dAddress'] = XSD1_restrictedAddress_deserialize;
    this.globalElementSerializers['{http://widgetVendor.com/types/widgetTypes}numInventory']
 = XSD1_numInventory_serialize;
    this.globalElementDeserializers['{http://widgetVendor.com/types/widgetTypes}numInventory']
 = XSD1_numInventory_deserialize;
    this.globalElementSerializers['{http://widgetVendor.com/types/widgetTypes}Address'] =
XSD1_Address_serialize;
    this.globalElementDeserializers['{http://widgetVendor.com/types/widgetTypes}Address']
= XSD1_Address_deserialize;
    this.globalElementSerializers['{http://widgetVendor.com/types/widgetTypes}widgetOrder
BillInfo'] = XSD1_widgetOrderBillInfo_serialize;
    this.globalElementDeserializers['{http://widgetVendor.com/types/widgetTypes}widgetOrder
BillInfo'] = XSD1_widgetOrderBillInfo_deserialize;
    this.globalElementSerializers['{http://widgetVendor.com/types/widgetTypes}widgetOrder
Info'] = XSD1_widgetOrderInfo_serialize;
    this.globalElementDeserializers['{http://widgetVendor.com/types/widgetTypes}widgetOrder
Info'] = XSD1_widgetOrderInfo_deserialize;
}
```

Most of the generated code is responsible for setting up the utility functions needed by the proxy, and for setting up the type support used for serializing and deserializing the data used by the service.

However, there are two important properties that you must use when setting up your proxy:

synchronous

> The synchronous property determines if the client communicates with the provider synchronously or asynchronously. The proxy is designed to use asynchronous communication because all operation stubs use callback methods to receive responses. However, setting this property

to true causes the client to block while it waits for a response from the service provider.

url

The url property specifies the URL for contacting the service provider.

> ⚠ **Important**
>
> Artix ESB does not support cross-scripting.

The code generator also produces stubs for each of the operations exposed by the interface. For one-way operations, the parameter list of the generated stub is identical to the parameter list defined in the interface. As shown in Example 10 on page 28, the generated stubs for two-way operations take two special parameters: *successCallback* and *errorCallback*. These parameters are user implemented callback methods that are called by the service provider when the response is ready.

Example 10. Two-Way Operation Stub

```
function widgetVendor_com_widgetOrderForm_placeWidgetOrder_op(successCallback, errorCallback,
 widgetOrderForm, intSize) {
...
}

widgetVendor_com_widgetOrderForm_orderWidgets.prototype.placeWidgetOrder = wid
getVendor_com_widgetOrderForm_placeWidgetOrder_op;
```

For more information on implementing the callbacks see Implementing the Callbacks on page 29.

Implementing the Callbacks

Overview

Artix ESB's JavaScript consumer support is designed to interact with service providers asynchronously. To accomplish this each two-way operation takes two callback objects when invoked:

• The response callback is invoked if the request is processed successfully

• The error callback is invoked if there is an error in processing the request

The response callback

The response callback is used by the service provider to return the response to the consumer. It has a single parameter that corresponds to the output message defined for the operation.

> 📄 **Note**
>
> Artix ESB JavaScript consumers only support document style exchanges.

Example 11 on page 29 shows a response callback for an operation that returns a response of the numInventory type shown in Example 8 on page 25.

Example 11. Response Callback

```
function inventoryUpdatetResponse(response) {
        sizeSpan = document.getElementById('sizeHolder');
        sizeSpan.firstChild.nodeValue = response.getSize(); ❶

        numSpan = document.getElementById('numHolder');
        numSpan.firstChild.nodeValue = response.getNumLeft(); ❷
}
```

The callback does the following:

❶ Retrieves the size property from the response and places it into a node for display

❷ Retrieves the numLeft property from the response and places it into a node for display

The error callback

The error callback is used by the service provider if there is an error in processing the request. It takes two parameters. The first parameter is the HTTP error code, and the second parameter is the HTTP error text.

shows a simple error callback.

Example 12. Error Callback

```
function onerror(errorNum, errorText) {
      alert('error ' + errorText);
}
```

Invoking Operations on a Service

Overview

Making requests on a remote service requires the following steps:

1. Instantiating a proxy object

2. Setting the proxy's url property to the appropriate URL

3. Invoking the proxy's operations

Setting up the proxy

Before you can make requests on a remote service you must instantiate and set-up the proxy object. As shown in Example 13 on page 31 this is a straightforward process.

Example 13. Setting up a JavaScript Service Proxy

```
var widgetProxy = new widgetVendor_com_widgetOrderForm_orderWidgets(); ❶

widgetProxy.url = "/widgetService"; ❷
```

The code in Example 13 on page 31 does the following:

❶ Instantiates a service proxy for the service

❷ Sets the proxy's url property to the URL for the service

> ⚠ **Important**
>
> Artix ESB does not support cross-scripting.

Invoking operations

What needs to be passed to an operation depends on whether the operation is one-way or two-way. One-way operations only need the parameters that are defined by the service interface. Two-way operations require two additional parameters that provide the service with the callback objects it uses to return the results of the operation.

Example 14 on page 31 shows code for invoking an operation on a service proxy.

Example 14. Invoking a Service Proxy's Operations

```
function inventoryUpdatetResponse(response) {
...
```

```
}

function errorCallback(errorNum, errorText){
...
}

...
function invokeService() {
...
// Proxy widgetProxy instantiated previously
widgetProxy.getInventory(inventoryUpdatetResponse, errorCallback, size, color);
...
}
```

Deploying Applications

JavaScript applications require an interpreter to run. Artix ESB provides a lightweight container for deploying services. Client applications developed with Artix ESB can be deployed either directly in a browser or using the Rhino engine.

Publishing Services

Overview

Artix ESB provides a lightweight container that allows you to deploy your JavaScript and E4X services and take advantage of the Artix ESB's pluggable transport infrastructure.

> ⚠ **Important**
>
> JavaScript based services work with SOAP messages, and while they are multi-transport, they can only use the SOAP binding.

Deployment command

You deploy services 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. `ServerApp` loads all of the scripts that contain JAX-WS metadata and publishes a service endpoint for each one.

`ServerApp` has three optional arguments:

Table 1. Optional Arguments to `ServerApp`

Argument	Description
`-a addressURL`	Specifies the address where `ServerApp` publishes the service endpoint implementation found in the script file that follows 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` runs 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 15 on page 35, your service is deployed at `http://cxf.apache.org/goodness`.

Example 15. 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 invoke the command shown in Example 16 on page 35. If the service defined by `hello_world.jsx` has a port name of helloWorld, `ServerApp` publishes it to `http://cxf.apache.org/helloWorld`. If the service defined by `goodbye_moon.js` has a port name of `blue`, `ServerApp` publishes it to `http://cxf.apache.org/blue`.

Example 16. 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 17 on page 35. Your service is deployed to `http://cxf.apache.org/goodness`.

Example 17. 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
```

`ServerApp` publishes three service endpoints:

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`

Running Clients in a Browser

Overview

The main use case for JavaScript client support is to facilitate the creation of lightweight service consumers that can run inside of a browser. Artix ESB's JavaScript client support is designed to work with the `XMLHttpRequest` object.

Supported browsers

Consumer's developed using the Artix ESB JavaScript framework can be deployed into any browser that supports the `XMLHttpRequest` object.

Browsers that support the `XMLHttpRequest` object include:

- Camino

- Firefox

- Flock

- Internet Explorer 7.x

- Konqueror

- Mozilla

- Opera 8.0 and newer

- Safari 1.2 and newer

- SeaMonkey

> **Note**
>
> There is experimental support for browsers that support `ActiveXObject(MSXML2.XMLHTTP.6.0")`. This functionality is not tested and it might be unpredictable.

Example

shows an HTML page that uses the Artix ESB JavaScript client support to access a Web service.

Example 18. Web Page with a JavaScript Client

```html
<html>
  <head>
    <title>Hello World JavaScript Client Sample</title>

    <script type="text/javascript" src="/SoapContext/SoapPort?js"></script> ❶
    <script type="text/javascript">
      var Greeter = new apache_org_hello_world_soap_http_Greeter(); ❷

      Greeter.url = "/SoapContext/SoapPort"; ❸

      var responseSpan;

      function sayHiResponse(response) ❹
      {
        responseSpan.firstChild.nodeValue = response.getResponseType();
      }

      function sayHiError(error) ❺
      {
        alert('error ' + error);
      }

      function invokeSayHi() ❻
      {
        responseSpan = document.getElementById('sayHiResponse');
        responseSpan.firstChild.nodeValue = " - pending - ";
        Greeter.sayHi(sayHiResponse, sayHiError);
      }
    </script>
  </head>
  <body>
      ...
    <p>Run sayHi</p>
    <input type="button"
           value="invoke" name="sayHi"
           onClick="invokeSayHi()"> ❼
    <p>sayHi response</p>
    <p><span id='sayHiResponse'>- not yet invoked -</span></p>
  </body>
</html>
```

does the following:

❶   Downloads the client-side JavaScript from the service provider

❷   Creates a new proxy object to access the service provider

❸   Sets the proxy object's url property to the URL used to access the service

⚠ **Important**

Cross-scripting is not supported

❹   Creates the callback object for receiving a response from the service

❺   Creates the callback object for receiving an Error from the service

❻   Creates a function that can be used to invoke the service

❼   Displays a button that will invoke the service when clicked

Index