RESTful Web Services

The RESTful Web Service plugin allows exposing business logic as a RESTful Web Service.

In This Chapter

About RESTful Web Services

RESTful Web Services utilize the features of the HTTP Protocol to provide the API of the Web Service. For example, it uses the HTTP Request Types to indicate the type of operation:

Operation	HTTP Request Type
Retrieving of existing records	GET
Creating new records	POST
Removing records	DELETE
Updating existing records	PUT

Using these 4 HTTP Request Types a RESTful API mimics the CRUD operations (Create, Read, Update & Delete) common in transactional systems.

A defining feature of REST is that it is stateless: each call the to a RESTful Web Service is completely stand-alone: it has no knowledge of previous requests.

Implementing a RESTful Web Service in Servoy

A RESTful Web Service endpoint is created by defining a Form in a Solution and implementing one or more of the following methods to the Form:

HTTP Request Type	Method name	Description
GET	ws_read	To retrieve data
POST	ws_create	To create new data
DELETE	ws_delete	To remove data

PUT	ws_update	To update data

The RESTful Web Service endpoint is then available through the URL `{serverUrl}/servoy-service/rest_ws/{solutionName}/{formName}`. Multiple endpoints can be created by implementing the relevant `ws_*` methods on multiple forms.

If the method for one of the HTTP Request Types is not implemented, that operation is not available on the endpoint and when attempted to be called a METHOD_NOT_ALLOWED (HTTP 405) error will be thrown.

The RESTful Web Service plugin does not have any client side functions or properties: all its magic happens by implementing the methods with the predefined `ws_*` names on a form.

> **Service Solution**
>
> Servoy solution that has the web service implementation should be a Service type solution, this solution type should only be used for web service provider and batch processor.

Keeping It Stateless

RESTful Web Services are to be stateless. The RESTful Web Service plugin internally uses a pool of Headless Clients to handle multiple concurrent requests to the REStful Web Service API. As subsequent requests to the RESTful Web Service API might be handled by different headless clients in the pool of clients configured for the plugin, no state should be preserved in the headless client at the end of the call to the API. Therefore, make sure any state is cleared at the end of logic. This means at least the following:

- Do not use global or form variables (or set them to default values at the end of the logic)
- Do not use the solution model API
- Do not alter the state of the form's UI
- Do save or rollback any unsaved changes before the end of the method

Supporting GET Requests

GET Requests are supported on the endpoint if the `ws_read():Object` method is implemented. The method can return either a JavaScript primitive/array/object or a byte array.

If the return value is a JavaScript object, it will be serialized and then placed in the body of the HTTP Response. See Returning Binary Data for more info on returning binary content.

If the return value of the method is null, a NOT_FOUND response (HTTP 404) will be generated.

Supporting POST Requests

POST Requests are supported on the endpoint if the `ws_create(content):Object` method is implemented. POST Requests require that data is supplied in the body of the request.

The body content of the HTTP Request is passed into the `ws_create()` method as first argument. If the body Content-Type is not application /binary, it will be automatically converted to a JavaScript object / string.

If the body content cannot be converted to a JavaScript object / string based on the Content-Type, an INTERNAL_SERVER_ERROR response (HTTP 500) will be generated. If the content in the body of the HTTP Request is missing, a NO_CONTENT response will be generated (HTTP 204).

The `ws_create()` method supports consumption of multipart post requests. If the POST Request is a multipart POST, the `content` parameter will receive an Array of Objects, with each entry in the Array corresponding to one part of the multipart post, as per the example below:

Multipart post message

```
------WebKitFormBoundaryX6nBO7q27yQ1JNbb
Content-Disposition: form-data; name="myFileDescription"

My sample file content.
------WebKitFormBoundaryX6nBO7q27yQ1JNbb
Content-Disposition: form-data; name="myFile";
        filename="SomeRandomFile.bin"
Content-Type: application/octet-stream

..... BINARY CONTENT.....
------WebKitFormBoundaryX6nBO7q27yQ1JNbb--
```

content parameter value

```
[
   {
      name:"myFileDescription",
          // bytes content is "My sample file."
           value:byte[],
      contentType:"text/plain"
   },
   {
      name:"myFile",
      value:byte[],
      fileName:"SomeRandomFile.bin",
      contentType:"application/octet-stream"
   }
]
```

The method can optionally return a JavaScript primitive/array/object or a byte array.

If the return value is a JavaScript object, it will be serialized and then placed in the body of the HTTP Response. See Returning Binary Data for more info on returning binary content.

Supporting DELETE Requests

DELETE Requests are supported on the endpoint if the `ws_delete():Boolean` method is implemented. The method has to return a Boolean value:

- true: to indicate successful deletion. This result will generate an OK response (HTTP 200)
- false: to indicate delete failure. This response will generate a NOT_FOUND response (HTTP 404)

Supporting PUT Requests

PUT Requests are supported on the endpoint if the `ws_update(content):Boolean` method is implemented. Data has to be supplied in the body of the HTTP request and the method has to return a Boolean value:

- true: to indicate successful update. This result will generate an OK response (HTTP 200)
- false: to indicate update failure. This response will generate a NOT_FOUND response (HTTP 404)

The body content of the HTTP Request is passed into the `ws_update()` method as first argument. If the body Content-Type is not application /binary, it will be automatically converted to a JavaScript object / string.

If the body content cannot be converted to a JavaScript object / string based on the Content-Type an INTERNAL_SERVER_ERROR response (HTTP 500) will be generated. If the content in the body of the HTTP Request is missing, a NO_CONTENT response will be generated (HTTP 204).

Adding Extra Data to the Request URL

Additional data can be added in the URL of the HTTP Requests. There are two variations and how they are made available in the `ws_*` methods differs.

Additional URL Fragments
The base URL for each operation is `{serverUrl}/servoy-service/rest_ws/{solutionName}/{formName}`.

Additional URL fragments can be added to the URL like:

`{serverUrl}/servoy-service/rest_ws/{solutionName}/{formName}/{someValue}/{anotherValue}`

The additional URL fragments `{someValue}` and `{anotherValue}` will be passed into the `ws_*` method as additional arguments. This means that for `ws_read()` and `ws_delete()` they will be the first and second argument and for `ws_create()` and `ws_update()` they will be the 2nd and 3rd argument, as these methods already have by default the content of the request as first argument.

Query Parameters
The request URLs can also be extended with Query parameters like: `{serverUrl}/servoy-service/rest_ws/{solutionName}/{formName}?{someKey}={someValue}&{anotherKey}={anotherValue}&&{anotherKey}={anotherValue2}`

If the URL contains Query parameters, these will be passed into the `ws_*` method as an additional last argument. This last argument is a JavaScript object containing all keys as properties with the values associated with the key in an Array: Object<Array<String>>

Note that if custom endpoint restrictions are implemented and the `ws_authenticate()` method returns any value but null or false, the RESTful Web Services plugin adds a query parameter with the key `ws_authenticate` and as value the returned value of the `ws_authenticate()` method.

Combining Additional URL Fragments and Query Parameters
Additional URL path elements and Query parameters can be combined in the URL (the query parameters should come after the additional URL fragments to make it a valid URL):

`{serverUrl}/servoy-service/rest_ws/{solutionName}/{formName}/{someValue}/{anotherValue}?{someKey}={someValue}&{someKey}={anotherValue}&&{anotherKey}={anotherValue2}`

Example

A HTTP Get Request on URL `{serverUrl}/servoy-service/rest_ws/myRestAPISolution/APIv1/foo/bar?name=John&age=30&pet=Cat&pet=Dog` would result in invoking the `ws_read` method on form `APIv1` of solution `myRestAPISolution`.

The `ws_read` function will be invoked with three parameters: `'foo'`, `'bar'`, `{name: ['John'] , age: [30] , pet: ['Cat', 'Dog'] }`

```
function ws_read()
{
  for (var i = 0; arguments.length, i++) {
      if (typeof arguments[i] == 'String') { //The URL path additions are passed
in as Strings
           application.output('URL Path addition: ' + arguments[i])
      } else {
           for each (var key in arguments[i]) {
                application.output('Query Parameter "' + key + '", values:  "' +
arguments[i][key].join(', ') + '"')
           }
      }
  }
}

//outputs:
//URL Path addition: foo
//URL Path addition: bar
//Query Parameter "name", values:  "John"
//Query Parameter "age", values:  "30"
//Query Parameter "pet", values:  "Cat, Dog"
```

Supported Content Types & Encoding

The plugin supports the following Content Types:

- JSON - application/json
- XML - application/xml
- binary - application/binary

Request Content-Type and Encoding

The Content-Type of the HTTP Request can be explicitly set by the caller using the `Content-Type` header in the HTTP Request. Optionally, the charset can be included in the Content-Type header to specify the encoding used. If the charset is not specified, `utf-8` will be assumed.

Content-Type Header

```
Content-Type: application/json; charset=utf-8
```

For requests that specify body content (POST and PUT Requests (resp. `ws_create()` and `ws_update()` methods), if no valid Content-Type is set, the plugin will try to establish the type of the content based on the first character of the content:

- '{': Content-Type application/json will be assumed
- '<': Content-Type application/xml will be assumed

When the Content-Type cannot be determined, an UNSUPPORTED_MEDIA_TYPE response will be generated (HTTP 415).

Response Content-Type and Encoding

By default, the plugin will respond with the same content type as used in the HTTP Request. The client calling the RESTful Web Service can request a different response content type by specifying the desired content type for the Response by setting the `Accept` header of the HTTP Request, for example: `Accept: application/json` or `Accept: application/binary`

The response will be encoded with the `utf-8` charset by default, if the response is not returning binary data.

If the encoding of the response needs to be different than the request encoding, this can be specified in the HTTP Request by setting the charset value in the Accept header:

Accept Header

```
Accept: application/json; charset=UTF-16
```

Returning Binary Data

The RESTful Web Services plugin supports returning binary data in GET operations. Binary data could for example be the content of a dataprovider of type MEDIA or the contents of a file read from disk. In this case, the body of the response will contain the actual stream of bytes.

Clients consuming the web service must be able to handle the binary response. When the consuming client specifies the `Accept` header, it must accept `application/binary`, otherwise an UNSUPPORTED_MEDIA_TYPE (HTTP 415) response will be generated. If the consuming client did not specify the Content Type of the request, the binary result is returned unconditionally.

Setting Custom Headers

Custom headers can be set in the response by implementing the `ws_response_headers()` method. This method can return:

- a **string** or an **array of strings** of type `"headerName=headerValue"`. For example `"Server=Servoy(Unix)"`.
- (starting with Servoy 8.2.2) an **object** or an **array of objects** containing headerName/headerValue pairs. For example `{ name : "Content-Disposition", value : 'attachment; filename="test.txt"' }`
  .

Each key/value pair will be added as Header to the HTTP Response.

Returning Custom Status Codes

While some of the HTTP Response status codes are hardcoded in the RESTful Webservices plugin, it is possible to control the HTTP Status codes from within the `ws_*` methods. Returning a custom HTTP Status Code can be done by throwing the specific value (a number) for the HTTP Status Code.

For example, when a `ws_update` call comes in for a non-existing resource, the HTTP Status Code to return would be a 'Not Found' status code, which is a HTTP 404. Returning the 404/Not Found HTTP Status code from within a `ws_*` method could be done the following way:

```
function ws_update() {
    //your logic here
    throw 404;
}
```

It is also possible to set the body content of the HTTP Response with a more elaborate message to give more context to the returned status code:

```
function ws_update(){
    //your logic here
    var code = plugins.http.HTTP_STATUS.SC_UNAUTHORIZED
    var message = "<?xml version=\"1.0\" encoding=\"UTF-8\"?
><error><reason>access denied</reason><message>access token is expired</message><
/error>"
        throw [code, message];
}
```

For convenience purposes, all available HTTP Status Codes are also listed under the HTTP Plugin, so instead of throwing the number 404 in the first example, a more readable way would be to throw plugins.http.HTTP_STATUS.SC_NOT_FOUND

For additional explanation of all the status codes, see List_of_HTTP_status_codes on Wikipedia.

Authentication/Authorization

By default access to any endpoint in unrestricted. The plugin supports 2 modes of implementing Authentication and authorization.

1. Restricting access to all endpoints based on HTTP Basic Authentication, validated against Servoy's built-in security
2. Restricting access per endpoint based on HTTP Basic Authentication with custom validation

Global Restrictions Based on Servoy Security

By setting the plugins server property `rest_ws_plugin_authorized_groups` to a comma separated list of groups defined in the built-in security system of Servoy, access to all endpoints is automatically restricted to users that are part of the groups in the list.

The endpoints must be called with HTTP Basic authentication to provide a username and password. The username/password supplied in the HTTP Request is validated against the users registered in Servoy's built-in security system and additionally against group membership. Access is denied if the user does not exist or the supplied password is incorrect, or the user doesn't belong to one of the specified groups.

When access is denied, an UNAUTHORIZED response is generated (HTTP 401).

Custom Endpoint Restrictions

For each endpoint custom authentication and authorization can be added by implementing the `ws_authenticate(userName, password): Object` method. If present, it will be called before performing any rest operation (ws_read, ws_create, ws_delete, ws_update) and the calls to the endpoint must be made with HTTP Basic Authentication to supply a username and password.

If the method returns false or null an UNAUTHORIZED response is generated (HTTP 401).

Any other return value will be added as an extra query parameter with the key `ws_authenticate` to the incoming request.

JSONP support

The plugin supports so-called JSONP: a variation of JSON that allows cross domain data retrieval. The JSONP variation can be invoked by added a 'callback' parameter to the HTTP Request URL:

```
{serverUrl}/servoy-service/rest_ws/{solutionName}/{formName}?callback=
{callbackFunctionName}
```

When invoked with the value "myCallback" for the 'callback' parameter, the returned JSON value will be wrapped in a function call to the `myCallback` function:

```
myCallback({ "hello" : "Hi, I'm JSON. Who are you?"})
```

Pool of Clients

To service the requests to the RESTful Web service API, the plugin creates a pool of (headless) clients. The maximum number of clients allowed can be set using the `rest_ws_plugin_client_pool_size` property of the plugin (default value = 5). If you have multiply rest solutions a the property: "rest_ws_plugin_client_pool_size_per_solution" can be used to set the pool size per solution (Servoy 8.3.2). This latter property will set the "`rest_ws_plugin_client_pool_size` " to -1. Because the max number will then be number of rest solutions multiplied by the "per solution".

When there are more concurrent requests than the number of clients in the pool, by default the requests will wait until a client becomes available in the pool. This behavior can be altered by setting the `rest_ws_plugin_client_pool_exhausted_action` property of the plugin. The following values are supported for this property:

- block (default): requests will wait until a client becomes available
- fail: the request will fail. The API will generate a SERVICE_UNAVAILABLE response (HTTP 503)
- grow: allows the pool to grow, by starting additional clients. This will set the 2 pool size properties to -1 (so voiding both of them). And if the "rest_ws_plugin_client_pool_size_per_solution" is set then that will be used as the number of idle clients to keep per solution.

> **Servoy Cluster**
>
> The RESTful Web service plugin uses a pool of headless clients to service the requests. When operated within a Servoy Cluster, note that poolsize is set per Servoy Application Server.

Client plugin

Code running inside a rest-ws request can access the request and the response using the rest_ws client plugin.

For example, retrieve a custom header sent by the client:

```
var request = plugins.rest_ws.getRequest();
var myheader = request.getHeader('My-Special-Header');
```

Note that the client-plugin can only be used within a request handler, the function `plugins.rest_ws.isRunningRequest()` can be used to check whether the code is running as a request hander.

See API documentation for the plugin.

Sample Solution

A sample solution is included in the Servoy distribution (servoy_sample_rest_ws.servoy), detailing how to retrieve data from the HTTP Request and to return a response.

More information on RESTful Web Services

See the following links for more information on RESTful Web Services:

- http://en.wikipedia.org/wiki/Representational_State_Transfer
- http://www.infoq.com/articles/rest-introduction
- http://www.ibm.com/developerworks/webservices/library/ws-restful/
- http://home.ccil.org/~cowan/restws.pdf

API Docs

Server Property Summary

Server Property Details

rest_ws_plugin_authorized_groups

rest_ws_plugin_client_pool_exhausted_action

rest_ws_plugin_client_pool_size