RESTful Web Services

Using the RESTful Web Service plugin business logic can be exposed as a RESTful Web Service.

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 used 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

The RESTful Web Service plugin does not contain any client side functions or properties, it is a 100% server side operating plugin.

A RESTful Web Service can be created by creating a Form in a solution and implement one or more of the following methods to the Form:

Method name	HTTP Request Type	Description
ws_read	GET	Used for the retrieval of data
ws_create	POST	Used for the creation of new records
ws_delete	DELETE	Used for the removal of data
ws_update	PUT	Used for updating data
ws_authenticate	N/A	Used to authenticate the requesting client
ws_response_headers	N/A	Allows the retrieval in the HTTP Headers in the incoming Request and set the HTTP headers in the outgoing Response

Implement ws_read():Object to allow data retrieval
By performing an HTTP GET on the url {serverUrl}/servoy-service/rest_ws/{solutionName}/{formName}, the ws_read() method will be invoked.
The method **must return** a JavaScript object. The object will be serialized and placed in the body of the HTTP Response. If the return value of the method is null, a NOT_FOUND response (HTTP 404) will be generated

Implement ws_create(content):Object to allow adding data
By performing an HTTP POST on the url {serverUrl}/servoy-service/rest_ws/{solutionName}/{formName}, the ws_create() method will be invoked. **Data has to be supplied** in the body of the HTTP Request.
The method **can return** a JavaScript object. The object will be serialized and placed in the body of the HTTP Response.

Implement ws_delete():Boolean to allow data removal
By performing an HTTP DELETE on the url {serverUrl}/servoy-service/rest_ws/{solutionName}/{formName}, the ws_delete() method will be invoked.
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)

Implement ws_update(content):Boolean to allow updating existing data
By performing an HTTP PUT on the url {serverUrl}/servoy-service/rest_ws/{solutionName}/{formName}, the ws_update() method will be invoked. **Data has to be supplied** in the body of the HTTP request.
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)

In case the matching method for the specific HTTP operation (GET, POST, DELETE or PUT) does not exists on the form, an INTERNAL_SERVER_ERROR response (HTTP 500) will be generated.

Request parameters

Additional parameters can be specified in the URL of the HTTP Requests. There are two variations and how they are forwarded to the ws_* methods differs.

Additional URL path elements
The base URL for each operation is {serverUrl}/servoy-service/rest_ws/{solutionName}/{formName}. Additional arguments can be specified by adding to the URL path:

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

The additional URL path elements {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 method already have by default the content of the request as first argument

Query parameters
The request URLs can also be extended with Query parameters: {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 a Array: Object<Array<String>>

Example
Additional URL path elements and Query parameters can be combined in the URL (the query parameters should come after the additional URL path elements):

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

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"
```

Stateless

RESTful Web Services are to be stateless. 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, do not use any state in between calls to the API. This means at least the following:

- Do not use global or form variables
- Do not use the solution model API
- Do not alter the state of the a form's UI
- Do save or rollback any unsaved changes before the end of the method

HTTP Request

For the POST and PUT operation (resp. ws_create() and ws_update() methods), data has to be supplied in the body of the HTTP Request. If the content in the body of the HTTP Request is missing, a NO_CONTENT response will be generates (HTTP 204).

The supported Content-Types are JSON (application/json) and XML (application/xml). The Content-Type can be explicitly set in the header of the HTTP Request:

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

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

Note: the charset is optional. If not specified, utf-8 will be assumed.

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).

The supplied value in the body of the HTTP request will be applied as argument to the invocation of the method. The body content will be converted to a JavaScript object automatically. If the body content cannot be converted to a JavaScript object based on the Content-Type an INTERNAL_SERVER_ERROR response (HTTP 500) will be generated.

HTTP Response

By default, the plugin will respond with the same Content Type as was specified in the HTTP Request. This can be overruled by specifying a different response Content-Type in the Accept header of the HTTP Request:

```
Accept: application/json
```

By default, the response will be encoded with the UTF-8 charset. If the HTTP Request specified a different encoding, that will be used instead. 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: application/json; charset=UTF-16
```

Returning custom status codes

While some of the HTTP Response status codes are hardcoded in the RESTful Webservices plugin (see this documentation), 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 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;
}
```

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

See List_of_HTTP_status_codes on WikipediA for additional information on all status codes

Authentication/Authorization

The RESTful Web service plugin can be configured to check authentication/authorisation.
The plugins server property rest_ws_plugin_authorized_groups can be set with a comma separated list of groups defined in the built-in security system of Servoy.
When the property is filled with usergroups, HTTP Basic authentication is enabled for all webservice requests. 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 exists 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).

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:

```
http://domain:port/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).

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 untill a client becomes available
- fail: the request will fail. The API will generate a SERVICE_UNAVAILABLE response (HTTP 503)
- grow: allows the pool to temporarily grow, by starting additional clients. These will be automatically removed when not required anymore.

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.

Samples

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.

For more information on RESTful Web Services, see:

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

Server Property Summary

rest_ws_plugin_authorized_groups
rest_ws_plugin_client_pool_exhausted_action
rest_ws_plugin_client_pool_size

Server Property Details

rest_ws_plugin_authorized_groups
Only authenticated users in the listed groups (comma-separated) have access, when left empty unauthorised access is allowed

rest_ws_plugin_client_pool_exhausted_action
The following values are supported for this property:
block (default): requests will wait untill a client becomes available
fail: the request will fail. The API will generate a SERVICE_UNAVAILABLE response (HTTP 503)
grow: allows the pool to temporarily grow, by starting additional clients. These will be automatically removed when not required anymore.

rest_ws_plugin_client_pool_size
Max number of clients used (this defines the number of concurrent requests and licences used), default = 5