English

Asema IoT Central

Server Side Scripting manual

ASEMA

Table of Contents

1. Introduction

Server Side Scripts (or later just "SSS") are a way to extend and customize the Asema IoT Central API. An SSS can receive data over HTTP POST in basically any format, process it asynchronously and return the result in any format. This gives great flexibility in implementing system integration and creating web applications for the system.

An SSS uses the common request-response model and has the freedom to modify both the body and the headers of the call. Each script has a context which remains in memory throughout the time the script is loaded. This means that the script can also be state-dependent and the state values are retained over several calls.

Access to the Asema IoT system takes place through contexts and each SSS has the API of the internal object cache, which lets the SSS find an manipulate objects in the Asema IoT system.

2. Calling scripts

Each SSS is recognized by its path. Server Side Scripts are separated from the rest of the calls to the system by a common beginning of the path while an individual script gets called from the path defined when the script is created.

The default common path is `/script`. This means that if your Asema IoT system responds at localhost address 127.0.0.1 and the path of your script is `myscript`, that script gets called when an HTTP POST is sent to http://127.0.0.1:8080/script/myscript. Similarly, a call to the script `myotherscript` would take place at http://127.0.0.1:8080/script/myotherscript.

The common path is set in the Webserver settings. You can find this at Asema IoT web admin interface under System > Webserver.

A call to the path splits the incoming data into headers and body, just like a standard HTTP server would do. These are passed to the script for processing.

3. A Hello World sample

To illustrate how scripts work, let's take a look at a classical "Hello world" sample. This script responds to a request synchronously with a simple "Hello world" text.

```
import ScriptingEngine 1.0
import QtQuick 2.3

RequestHandlingScript {
 function onRequest(request, response) {
  response.setHeader('content-type', 'text/plain);
  response.write("Hello World!");
  response.end();
 }
}
```

Each call to an SSS goes to the `onRequest` method. This is a mandatory method your script must implement. The method must always accept the same parameters: a request object and a response object. The request contains the data you get from the call and the response has the data you want to return.

You can write to the response headers with the `setHeader()` method and to the body with the `write()` method. The response will accept input until you call the `end()` method. Until that the response remains unsent.

> **Important**
> Always remember to call `end()`for the response. Without this method call, your script will not send anything back.

4. Handling incoming data

Incoming data is contained in the headers and the request body. Headers are in the request map `.headers`. The body is an array of bytes in `request.body`. JavaScript has native methods for parsing these into a structured format. The most common option is handling JSON. It is parsed by calling `JSON.parse()`.

The opposite to parsing is serialization. JavaScript engine again has a native method for this: `JSON.serialize()`. Let's look at an example:

```
import ScriptingEngine 1.0
import QtQuick 2.3

RequestHandlingScript {
 function onRequest(request, response) {
  if (request.headers['content-type'] == 'application/json') {
   var json = JSON.parse(request.body);
   var respDict = { a: 10, b: 100, c: json }
   response.setHeader('content-type', 'application/json');
   response.write(JSON.stringify(respDict));
   response.end();
  }
 }
}
```

The example above takes some JSON structure from the incoming call, forms the `respDict` dictionary with a couple of sample values and adds the incoming data as one variable. The resulting structure is returned in a serialized format.

5. Synchronous and asynchronous responses

An SSS can respond to a request immediately (synchronously) or after some processing and returning a signal (asynchronously). Both methods allow data processing before responding but the asynchronous method lets other operations in the system run while the call results are processed.

To be able to respond by writing to the correct response object while waiting for processing to finish, you need to store that object. There is the `store()` method that does this. You can then fetch these objects from memory with `storedRequest()` and `storedResponse()` methods. So when you receive a call that needs an asynchronous response, (1) store the request and response, (2) make a processing call that returns a signal when done, (3) react to the signal by loading the request and response from memory and writing the data accordingly.

Let's take an example:

```
import ScriptingEngine 1.0
import QtQuick 2.3

RequestHandlingScript {

 function callback() {
  sendResponse("done");
 }

 function onRequest(request, response) {
  store(request, response);
  mydatahandler.dataReady.connect(callback)
  mydatahandler.process();
 }

 function sendResponse(result) {
  if (storedRequest().headers['content-type'] == 'application/json') {
   var resp = { result: result };
   storedResponse().setHeader('content-type', 'application/json');
   storedResponse().write(JSON.stringify(resp));
   storedResponse().end();
  }
 }
}
```

In this example there is some asynchronous data handler `mydatahandler` which sends a signal `dataReady` once it is done. So our `onRequest` method connects to that signal and tells the handler to process the call. Once processed, that signal causes the `callback()` method to be called. This in turn calls the `sendResponse()` method that fetches the stored values from memory and writes the response to them.

6. Manipulating objects

To actually do some IoT with your scripts, you'll need to access the of Asema IoT object pool. In Server Side Scripts, you do this in exactly the same manner as you would do in any other QML-based scripting method in Asema IoT, such as screenlets.

The objects are accessed through the ObjectManager. To fetch an object, connect to the `objectsReady()` signal of ObjectManager and call one of its object-fetching methods. Once the objects are ready for processing, your callback will be called, you can do the processing and then respond to the call.

> **Note**
> For full documentation of the API of ObjectManager, please see the Asema IoT Central Context API documentation.

Let's take another example. The following shows how to fetch an object by its GID from the pool. It then extracts the value of a property given in the request and returns a disctionary with that value in the response.

```
import ScriptingEngine 1.0
import QtQuick 2.3

RequestHandlingScript {
 property variant sampleObject: null

 function loadedObjectsReady(propertyName) {
  if (propertyName == "sample") {
   var json = JSON.parse(storedRequest().body);
   sampleObject = context.objects.sample.list[0];
   sendResponse(sampleObject.properties[json.property]);
  }
 }

 function onRequest(request, response) {
  store(request, response);
  var json = JSON.parse(request.body);
  if (sampleObject == null) {
   context.objectsReady.connect(loadedObjectsReady);
   objectManager.findObjectByGid(context, "sample", json.gid);
  } else {
   sendResponse(sampleObject.properties[json.property]);
  }
 }

 function sendResponse(propertyValue) {
  if (storedRequest().headers['content-type'] == 'application/json') {
   var resp;
   if (propertyValue != undefined) {
    resp = { property: propertyValue };
   } else {
    resp = { property: 'unknown' };
   }
   storedResponse().setHeader('content-type', 'application/json');
   storedResponse().write(JSON.stringify(resp));
   storedResponse().end();
  }
 }
}
```

Similar to getting properties, you can set properties and in this way achieve automation. The process is very similar to the example above, simply use the `setProperty()` method on the object. The following example shows how to do this:

```
import ScriptingEngine 1.0
import QtQuick 2.3

RequestHandlingScript {
 property variant sampleObject: null

 function loadedObjectsReady(propertyName) {
  if (propertyName == "sample") {
   var json = JSON.parse(storedRequest().body);
   sampleObject = context.objects.sample.list[0];
   sampleObject.setProperty(json.property, json.value);
   sendResponse("OK");
```

```
  }
 }

 function onRequest(request, response) {
  store(request, response);
  var json = JSON.parse(request.body);
  if (sampleObject == null) {
   context.objectsReady.connect(loadedObjectsReady);
   objectManager.findObjectByGid(context, "sample", json.gid);
  } else {
   console.log("SET: " + json.property + " = " + json.value);
   sampleObject.setProperty(json.property, json.value);
   sendResponse("OK");
  }
 }

 function sendResponse(result) {
  if (storedRequest().headers['content-type'] == 'application/json') {
   var resp = { result: result };
   storedResponse().setHeader('content-type', 'application/json');
   storedResponse().write(JSON.stringify(resp));
   storedResponse().end();
  }
 }
}
```

7. Accessing the database

In addition to accessing data through objects, you can access the Asema IoT database from Server Side Scripts directly. This may be useful if you have some proprietary data stored in a database that is not covered by the object system itself. This is a probable scenario because the purpose of Server Side Scripts is to offer extensions to the API in operations that are not covered by the basic object system. So such extensions might also have extensions to the data needs.

The queries to the database are standard SQL queries which you generate in the scripts. You run these queries with the `rawQuery()` method. This method takes an SQL string and runs it against the database. The result is a list of dictionaries. Each item in the list is one row of the result. Each row contains key-value pairs of the query with column name as the key and column value as the value. So if your query is a SELECT for columns "id" and "name", a row entry may look like this: `{ id: 1, name: "My Object" }`.

Let's take an example of a database query inside the Server Side Script. This one returns a comma-separated list of all values in the "mytable" table.

```
import ScriptingEngine 1.0
import QtQuick 2.3

RequestHandlingScript {
 function onRequest(request, response) {

  var resultString = "Database values: ";
  var allVals = rawQuery("SELECT * from mytable");
  for (var rowIndex in allVals) {
   var row = allVals[rowIndex];
   for (var key in row) {
    resultString += "," + row[key];
   }
  }

  response.setHeader('content-type', 'text/plainjson');
  response.write(resultString);
  response.end();
 }
}
```