D2000 REST API

The REST API interface implemented by the SmartWeb platform can be divided into the following parts:

- interface for authentication
- interface for access to data and services of the D2000 system
- administrative interface for monitoring calls into D2000 and a state of the SmartWeb server

These parts of the REST API interface are described in the following chapters.

Authentication

As we have already mentioned in the chapter Other Functions of the SmartWeb Platform, the supported way of authentication for the REST API is HTTP-BASIC. This type of authentication sends the user name and password directly in the header of every HTTP request. This means that every REST API request also automatically authenticates the user. In the case of an unsuccessful authentication, the server returns the status 404 in the header of the HTTP response. That is why it is not necessary to have an explicit login into the REST API interface via a special URL. Despite that, it is optimal to extract the function because of applications in which users login explicitly and thus the application needs to verify the given name and password.

Verifying of successful authentication is thus possible by sending an empty GET request with the HTTP-BASIC authentication to the address:

```
GET https://<domain.sk>/<application name>/api/rest/v0/d2/auth/login
```

Logout is realized by sending an empty GET request with the HTTP-BASIC authentication to the address:

```
GET https://<domain.sk>/<application name>/api/rest/v0/d2/auth/logout
```

> Calling of an explicit logout is recommended because the SmartWeb server keeps the session of a logged in user of the REST service (identified by their login name) until the expiration of the session configurable in the part Configuration of Authentication of the SmartWeb Platform.

Reading Values from Archive

Reading values from an archive is possible via the GET request with the HTTP header `Content-Type: application/json` to the address:

```
POST https://<domain.sk>/<application name>/api/rest/v0/d2/archive/<archive object name>?beginTime=<integer>&endTime=<integer>&oversampleSeconds=<integer>&limitDataLength=<integer>returnFields=<text>
```

Meanings of individual parameters are as follows:

Parameter name	Type	Mandatory	Description
`beginTime`	integer (number of milliseconds from the epoch)	yes	the beginning of a time interval of requested historical values
`endTime`	integer (number of milliseconds from the epoch)	yes	the end of a time interval of requested historical values
`oversampleSeconds`	integer (number of seconds)	no	interval length in seconds for data oversampling, if it is not defined, original data will return
`limitDataLength`	integer (number of values)	no	maximal number of returned values, if it is not defined, all values from the interval will return
`returnFields`	text (defined Unival attributes separated by a comma)	no	requested Unival attributes which will be returned together with a time stamp and a value, e.g. `"Status,Flags"`

For example, for the following HTTP GET calls

GET http://localhost/smartWeb/api/rest/v0/d2/archive/H.AdeunisRF1External?beginTime=504501912000&endTime=1504601912000,

we will get the response with an array of historical values from the server:

```
[
        [
                1503492475090,
                23.2
        ],
        [
                1503642560209,
                23.2
        ],
        [
                1503643165774,
                23.3
        ],
        ...
]
```

Every historical value is represented by an individual array due to the size of the transmitted message, and the first array element is always a time stamp and the second is always a value. In the case of defining other returning arrays by the parameter `returnFields`, these parameters are returned in other elements of an array according to the order in which they were defined.

Calling of D2000 RPC Methods

Via the REST interface, it is possible to call D2000 RPC procedures written in ESL and also in Java. Calling of ESL RPC procedures happens by sending a POST request with the HTTP header `Content-Type: application/json` to the address:

```
POST https://<domain.sk>/<application name>/api/rest/v0/d2/rpc/<event name>/<RPC method name>
```

in the case of Java RPC calling is URL as follows:

```
POST https://<domain.sk>/<application name>/api/rest/v0/d2/rpc/java/<event name>/<RPC method name>
```

The body of the sent message is the JSON array with parameters of calling RPC. The output of such request are values of output RPC parameters stored in the JSON object, attributes of which are required names of output parameters defined by the attributes `returnAs`. Details of serialization of RPC method parameters were described in the previous chapter. The example of calling RPC with the name `TestInOut` on the event `E.E.SmartWebApiTutorial` with five parameters, while the first, the third and the fourth parameter is an input-output one and defines a logical name for returned parameters' values. Input parameters (the second and the fifth) also use implicit conversion on the Unival object from simple JSON types.

```
POST http://localhost/smartWeb/api/rest/v0/d2/rpc/E.E.SmartWebApiTutorial/TestInOut
```

RPC method call example

```
[
  {
    "type": "bool",
    "value": "vTrue",
    "returnAs": "boolParam" // the output value will be named boolParam
  },
  123,
  {
    "type": "real",
    "value": 10.9,
    "returnAs": "realParam", // the output value will be named realParam
    "returnFields": ["ValueTime", "Status"] // attributes ValueTime and Status are required to the output value
too
  },
  {
    "type": "time",
    "returnAs": "timeParam" // the output value will be named timeParam
  },
  "hello D2000"
]
```

The code of the called RPC method can be for example:

```
RPC PROCEDURE TestInOut(BOOL _bool, IN INT _int, REAL _real, TIME _time, IN TEXT _text)
  _bool := !_bool
  _real := _real / 2
  _time := SysTime
END TestInOut
```

The output of the RCP calling is in the JSON object which has attributes according to the required names of output parameters:

RPC method call output

```
{
  "realParam": {
    "type": "real",
    "value": 5.45,
    "status": [
      "Valid"
    ],
    "valueTime": 1498213794522
  },
  "boolParam": {
    "type": "bool",
    "value": "vFalse"
  },
  "timeParam": {
    "type": "time",
    "value": 1498213794012
  }
}
```

Calling of D2000 SBA RPC Methods

Because D2000 does not know binary data type, RPC procedures are not suitable for binary data transfer. The Simple Byte Array (SBA) methods written in the D2000 Java serve for this purpose.

> SBA RPC is, in fact, a Java RPC method which has one input parameter of the byte array type (`byte[]`) and the same output parameter.

Downloading of Binary Data from D2000 via HTTP GET (URL Link)

To download binary data from D2000 to a client, we use the GET command with the HTTP header `Content-Type: application/octet-stream on the address`:

```
GET https://<domain.sk>/<application name>/api/rest/v0/d2/sba/<event name>/<SBA RPC method name>?fileName=file.dat
[&parameterX=valueX&parameterY=valueY&...]
```

Parameters of the query (the part after the question mark) are automatically sent to the SBA method in the same way as they were inputted into the address. The parameter fileName defines the name of the downloaded file under which the web browser will store the file on the disc. If this parameter is not set, the name of the downloaded file will be implicitly "file.dat". Besides that, the Smart Web server adds for SBA RPC also the parameter sessionUserName for identification of a user who calls SBA RPC. Other parameters depend on the particular implementation of SBA RPC method calling. The SBA RPC method gets all the inputted parameters in the input byte array (`byte[]`).

The following example illustrates the calling and implementation of the SBA RPC method for downloading a particular file of the pdf report from the local file system. The following example, however, is not suitable at all for a real usage and it is only for illustration of SBA RPC calling. Calling of SBA RPC method `reportContract_PDF` in the event `E.E.SmartWebApiTutorial` with the parameter `id` which identifies the report number and thus the downloaded file.

```
GET http://localhost/smartWeb/api/rest/v0/d2/sba/E.E.SmartWebApiTutorial/reportContract_PDF?fileName=report.
pdf&id=10
```

The implementation of the SBA RPC method is as follows:

```
package app.runnables;

import app.wrappers.E$E.SmartWebApiTutorial__$WRAPPER$__;

import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;


public class E$E.SmartWebApiTutorial extends E$E.SmartWebApiTutorial__$WRAPPER$__ {

    public byte[] reportContract_PDF(byte[] urlParamsBytes) throws IOException {
                // Parsing of sent URL paramaters into hash-map
        final HashMap<String, String> parameters = getParametersFromUrlQueryString(urlParamsBytes);
                // Parsing parameters id into variable
        final long contractId = Long.parseLong(parameters.get("id"));
                // Acquiring the name of current user calling SBA RPC
        final String userName = parameters.get("sessionUserName");
        System.out.println("DEBUG: Downloading contract with id " + contractId);
        // Creating a path to the file with the given report
        Path path = Paths.get("D:/D2000/Contract" + contractId + ".pdf");
                // reading file content and its return as a byte array
        return Files.readAllBytes(path);
    }

        /**
          * Utility method, it returns parsed URL parameters from the input array byte[]
          **/
    private HashMap<String, String> getParametersFromUrlQueryString(byte[] urlQueryStringBytes) throws
UnsupportedEncodingException {
        final String urlParamsString = new String(urlQueryStringBytes, "UTF-8");
        final List<String> paramPartsList = Arrays.asList(urlParamsString.split("&"));
        final HashMap<String, String> parameters = new HashMap<String, String>();
        for (String paramPartsString : paramPartsList) {
            final String[] paramParts = paramPartsString.split("=");
            final String paramName = paramParts[0];
            final String paramValue = paramParts.length > 1 ? paramParts[1] : null;
            parameters.put(paramName, paramValue);
        }
        return parameters;
    }
};
```

Sending Binary Data to D2000 via HTTP POST

Sending binary data to D2000 is possible via HTTP POST method to an identical URL link when downloading binary data with the same HTTP header `Content-Type: application/octet-stream`.

```
POST https://<domain.sk>/<application name>/api/rest/v0/d2/sba/<event name>/<SBA RPC method name>
```

When sending binary data, it is not possible to send specific URL parameters directly to the SBA RPC method. The value of the input parameters of the SBA RPC method will be in this case a binary content sent in the body of the POST request. However, clients are in no way limited by the content type which they send via the HTTP POST request. The only condition is for the binary content to be able to decode the relevant implementation of the SBA RPC method, analogically as in the previous case when we illustrated decoding of URL parameters with a special utility by the method `getParametersFromUrlQueryString()`. If the client needs to send with the binary content also additional input parameters (e.g. identifying the content), encoding and decoding of such content in SBA RPC will be completely in his competency. The following chapter contains the recommended way of such coding.

Recommended way of coding input parameters together with binary content

In the case that we need to send with the binary content itself also other input parameters within calling of the SBA RPC method, we recommend encoding parameters also with the attached content into a zip stream. The advantage of such solution is the universality and simplicity of usage in most of the programming languages (ZIP compression is mostly well-supported either in the standard library of the given language or in some other freely available version of a library).

We show an example of compression of more parameters into one ZIP byte stream in Java on the client's side:

```
/**
  * Proposal method how to serialize multiple key-value pairs to byte array
  */
  @SuppressWarnings("unused")
  private byte[] writeParameters(Map<String, byte[]> params) throws IOException {
      try (ByteArrayOutputStream baos = new ByteArrayOutputStream()) {
          try (ZipOutputStream zos = new ZipOutputStream(baos)) {
              for (Map.Entry<String, byte[]> entry : params.entrySet()) {
                  final ZipEntry zipEntry = new ZipEntry(entry.getKey());
                  zipEntry.setSize(entry.getValue().length);
                  zos.putNextEntry(zipEntry);
                  zos.write(entry.getValue());
                  zos.closeEntry();
              }
          }
          return baos.toByteArray();
      }
 }
```

Then an example of decoding the ZIP byte stream on the SBA RPC side will be:

```
/**
  * Proposal method how to deserialize multiple key-value pairs from byte array
  */
@SuppressWarnings("unused")
private Map<String, byte[]> loadParameters(byte[] inputBytes) throws IOException {
    final Map<String, byte[]> dataParts = new HashMap<String, byte[]>();
    try (ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(inputBytes)) {
        try (ZipInputStream zipInputStream = new ZipInputStream(new ByteArrayInputStream(inputBytes))) {
            ZipEntry zipEntry = zipInputStream.getNextEntry();
            byte[] buffer = new byte[4096];
            while (zipEntry != null) {
                final ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
                int len;
                while ((len = zipInputStream.read(buffer)) > 0) {
                    outputStream.write(buffer, 0, len);
                }
                final byte[] data = outputStream.toByteArray();
                dataParts.put(zipEntry.getName(), data);
                zipEntry = zipInputStream.getNextEntry();
            }
        }
    }
    return dataParts;
}
```

Automatic coding of input parameters together with binary content

As an alternative to a previous way of coding input parameters together with the binary content on the client's side is calling SBA RPC method via HTTP POST with another value of the HTTP header `Content-Type: multipart/form-data.` In this case, the body of the POST calling is coded according to the defined standard for sending forms also with binary files from a browser. SmartWeb supports also this format for calling SBA RPC methods. The input parameter when calling the SBA RPC method will contain in this case content of values of individual form arrays zipped in a way described in the previous chapter. This means that for decoding parameters, it is possible to use already mentioned Java method `loadParameters`.

> The Smart Web server, in this case, same as with downloading binary data from D2000 via HTTP GET, automatically adds for SBA RPC also the parameter sessionUserName for identification of a user that calls SBA RPC.