Welcome to our webinar!

- This webinar starts in a moment - please stay tuned
- This webinar will be recorded
- You will get the slides, recording and SQL snippets
- During the webinar, you may ask questions using the Q&A button - you may ask questions anonymously
- You can hop on and off on all the sessions

Day 1	14:00 - 15:00 Automate tasks using SYS and SYSADMIN schemas 15:15 - 16:00 The DV REST API and how to use it 16:15 - 17:00 Security
Day 2	15:00 - 16:00 Synchronization Methods 16:15 - 17:15 Working with Connectors
Day 3	15:00 - 16:00 Self-service Troubleshooting 16:15 - 17:15 Job Management Outlook

DATA VIRTUALITY

Working with Data Virtuality's REST API

Agenda

- Motivation
- How to access the REST API
- Overview of endpoints
- Endpoint /status
- Endpoint /source
- Endpoint /query
- Summary

Motivation - Why this course?

- Learn about the API's very existence
- Integrate Data Virtuality into websites, front-ends, self-service portals
- Trigger processes from outside of Data Virtuality's native tools
- Learn about REST APIs in general and learn by example from Data Virtuality

Organization

Learning objectives

- Learn the capabilities of the built-in API
- See actual requests to perform various types of tasks

Supplementary files

- Every participant will receive a link to GitHub after the masterclasses are over
- All slides can be found in the repository
- There will be a Postman collection with all the sample requests and templates
- Additional files like enabling HTTPS for on-premise installations can be found there

Basic information

How to access the API - Samples used in the class

Sample Requests

- Requests will be HTTP (unsecure) so everyone can test them (SaaS and on-premise customers)
- The Postman collection has the Header pre-defined in the collection settings
- All Postman requests ‘inherit’ the authentication from the collection settings
- The .rest sample file(s) in GitHub will explicitly show all manually set headers, including the Authorization header
- The requests are sorted with increasing complexity, i.e. easiest ones first
- All requests are tested and shown on 2.3.15 release

How to access the API - Endpoint

SaaS customers

- Unsecure requests go to http://<HostnameOrIP>/rest/api
- Secure requests go to https://<Hostname>/rest/api

On-premise customers

- Unsecure requests go to http://<HostnameOrIP>:8080/rest/api
- Secure requests go to https://<Hostname>:8443/rest/api

  Secure requests must be enabled first (see document enable-secure-web-server.pdf)

How to access the API - Authentication

Authentication

- All requests require **Basic Authentication** to be performed
- Method one - put the credentials in the request URL: http://<user>:<pass>@<HostnameOrIP>/rest/api
- Method two (preferred) - add an **Authorization header** to your requests

Authorization

- API requests are all subject to permissions.
- Only operations that the calling user can performed will succeed
- calling user = the account whose credentials were used for the Basic Authentication

How to access the API – Auth Header

Get the authentication token

- `select to_chars(to_bytes('<username>:<password>', 'UTF-8'), 'BASE64') as BasicToken;`
- `Replace the variabes with your actual user + pass and you will get the authentication token`

Apply the header

- The requests need the following header
  Authorization: Basic <BasicToken>

Available endpoints - /status

Endpoint /status

Get API Status

- GET operation only

Sample request

```
GET http://localhost:8080/rest/api/status HTTP/1.1
Authorization: Basic <BasicToken>
```

Sample Response

```
{
  "status": "OK"
}
```

Available endpoints - /source

Endpoint /source - Operations

Supported operations

- List available data sources and virtual schemas
- List tables, views and procedures within a data source or virtual schema
- List the contents of a table or view
- Call a stored procedure
- Bulk-calling a stored procedure
- Add a new data source
- Drop an existing data source

Endpoint /source - Operations

Unsupported operations

- Create a new virtual schema

  Can be done via *SYSADMIN.createVirtualSchema* procedure
- Drop an existing virtual schema

  Can be done via *SYSADMIN.dropVirtualSchema* procedure (requires the schema id!)
- Update an existing data source

  Can be done via deleting the data source and recreating it.

  This is also not possible via SQL. Uses the same approach.

Endpoint /source - List available data sources/schemas

Input

- none

Return parameters

- **Name**: data source or virtual schema name
- Returns a list of system schemas (UTILS, SYS, SYSADMIN, SYSLOG, ...) too
- Only returns what the user is allowed to access in terms of permissions

Endpoint /source - List data source/schema contents

Sample Request

```
GET http://localhost:8080/rest/api/source HTTP/1.1
Authorization: Basic <BasicToken>
```

Sample Response

```
[{
    "Name": "SYS"
  },
   {
    "Name": "views"
  },  ...
  ,{
    "Name": "dwh"
}]
```

Endpoint /source - List data source/schema contents

Input

- Name of the data source or virtual schema in endpoint URL.

  Input is case insensitive

Return parameters

- **Name**: object name
- **Type**: one of {Table, View, Procedure)
- **Description**: the object description, if there is any. *null* otherwise

Endpoint /source - List data source/schema contents

Sample Request for ‘dwh’

```
GET http://localhost:8080/rest/api/source/dwh HTTP/1.1
Authorization: Basic <BasicToken>
```

Sample Response

```
[ {
    "Name": "mytable",
    "Type": "Table",
    "Description": null
  },
  {
    "Name": "native",
    "Type": "Procedure",
    "Description": "Invokes translator with a native query that returns results in array of values"
  }]
```

Endpoint /source - Get a table's/view's records

Input

- Name of the data source or virtual schema and the object in endpoint URL

  Input is case insensitive

Return parameters

- Returns a list of JSON documents that are the actual data of the table or view
- Always returns the entire data in one go. No pagination.
- For larger tables/views the /query endpoint should be used and the data can be retrieved in chunks

Endpoint /source - List data source/schema contents

Sample Request for “views.v1”

```
GET http://localhost:8080/rest/api/source/views/v1 HTTP/1.1
Authorization: Basic <BasicToken>
```

Sample Response

```
[{
    "id": 1,
    "message": "hello",
    "createdTime": "2020-09-07T11:13:07.663+02:00",
    "isAvailable": true
  }, {
    "id": 2,
    "message": "world",
    "createdTime": "2020-09-07T11:13:07.663+02:00",
    "isAvailable": false
}]
```

Demo - List sources and their contents

Endpoint /source - Execute a stored procedure

<u>Overview</u>

- Executing procedures require **POST** operations
- Every API call requires the header **Content-Type: application/json**
- Every API call, even for a procedure without input parameters, requires a request body
- The request body is
  - one JSON document for a single procedure execution ('*{"param1": val1, "param2": "val2"}*')
  - an array of JSON documents for bulk execution
  - an empty JSON for a procedure without input parameters ('{}')
- The response is always an array of JSON documents but bulk execution adds a property name to each document and it follows the pattern <procedureName>_1, <procedureName>_2, ...

Endpoint /source - Execute a procedure without input

Sample Request for “SYSADMIN.getCurrentDWH”

```
POST http://localhost:8080/rest/api/source/SYSADMIN/getCurrentDWH HTTP/1.1
Authorization: Basic <BasicToken>
Content-Type: application/json

{}
```

Sample Response

```
[
  {
    "nameInDv": "dwh",
    "nameInSource": "public"
  }
]
```

Notice the additional header and the empty request body (‘{}’) in the call above!

Endpoint /source - Execute a procedure with input

Sample Request for “views.solveQE”

```
POST http://localhost:8080/rest/api/source/views/solveQE HTTP/1.1
Authorization: Basic <BasicToken>
Content-Type: application/json

{"a": 1, "b": -1, "c": -4}
```

Sample Response

```
[
    {
        "solution1":2.562
        ,"solution2":-1.562
    }
]
```

Endpoint /source - Bulk-execute a procedure with input

Sample Request for “views.solveQE”

```
POST http://localhost:8080/rest/api/source/views/solveQE HTTP/1.1
Authorization: Basic <BasicToken>
Content-Type: application/json

[{"a": 1, "b": -1, "c": -4}
,{"a": 1, "b": -2, "c": -4}]
```

Sample Response

```
[
    {"solveQE_1":
        [{"solution1":2.562,"solution2":-1.562}]
    }
    ,{"solveQE_2":
        [{"solution1":3.236,"solution2":-1.236}]
    }
]
```

Demo - Execute stored procedures

Endpoint /source - Drop an existing data source

Overview

- An existing data source can be dropped
- This is permanent so be careful!
- Does not work with virtual schemas
- The API request uses a **DELETE** operation
- Request body is not needed

Endpoint /source - Drop an existing data source

Sample Request for “file_src”

```
DELETE http://localhost:8080/rest/api/source/file_src HTTP/1.1
Authorization: Basic <BasicToken>
```

Sample Response header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Thu, 01 Jan 1970 01:00:00 CET
Content-Length: 0
Date: Mon, 07 Sep 2020 10:18:32 GMT
Connection: close
```

Endpoint /source - Create data source

Overview

- A new data source can be added
- Executing procedures require **POST** operations
- Every API call requires the header **Content-Type: application/json**
- A successful request gets an empty response with 200 code
- Usually, this operation is done by *SYSADMIN.createConnection()* and *SYSADMIN.createDataSource()* procedures but only one API request is needed
- When in doubt, export an existing data source via Studio exporter to know which information goes where in the input parameters

Endpoint /source - Create data source

Input

- **name**: target name for the new data source
- **template**: cli template that will add a connection to the underlying application server
- **translator**: which component to use when mapping SQL commands to the data source's native system
- **connectionProps**: the main connection information like host, database, username, ..
  also contains the list of added JDBC properties, if there are any
- **modelProps**: which objects to read from the source
- **translatorProps**: translator-specific properties like *supportsNativeQueries*

Endpoint /source - Create data source sample 1

Sample values for a local MS SQL Server

- name: "ds_ms_advworks",
- template: "mssql",
- translator: "sqlserver",
- connectionProps: "host=localhost,port=1433,db=AdventureWorks,user-name=sa,password=<Password>",
- modelProps: "importer.tableTypes=\"TABLE,VIEW\",importer.schemaPattern=\"Sales,Person\",importer.importIndexes=TRUE,importer.useFullSchemaName=FALSE",
- translatorProps: "SupportsOrderByString=false"

Endpoint /source - Create data source sample 2

Sample values for a local directory as ‘file’ data source

- name: "ds_local_file",
- template: "ufile",
- translator: "ufile",
- connectionProps: "ParentDirectory=D:/FileDataSource",
- modelProps: "importer.useFullSchemaName=false",
- translatorProps: ""

Endpoint /source - Create data source sample 2

Sample request

```
POST http://localhost:8080/rest/api/source/
Authorization: Basic <BasicToken>
Content-Type: application/json

{
    "name":"ds_local_file",
    "template":"ufile",
    "translator":"ufile",
    "connectionProps":"ParentDirectory=C:/FileDataSource",
    "modelProps":"importer.useFullSchemaName=false",
    "translatorProps":""
}
```

Demo - Dropping and creating data sources

Available endpoints - /query

Endpoint /query - Operations

Supported operations

- Run queries via POST (preferred): query is part of the request body
- Run queries via GET: query is part of endpoint URL
- Use pagination for large result sets
- Limit and offset for pagination
- Result set can be an array of objects or a two-dimensional array of single values
- Optional header can be returned alongside the result set
- Allow to workaround missing endpoints, operations and enable more sophisticated processes

Endpoint /query - Input parameters

Parameter definition

- **array** : boolean (optional, default : true) - if true, returns a 2D array, otherwise returns the array of objects. See samples below.
- **headers** : boolean (optional, default : false) - works only with **array=true** and controls whether the column headers will be included in response or not.
- **pagination** : boolean (optional, default : false) - activates pagination mode.
- **requestId** : String (optional, default : "") - allows to use a cursor with the provided identifier which was buffered in a previous request.
- **limit** : long (optional, default : -1) - restricts the number of results returned from a SQL statement.
- **offset** : long (optional, default : -1) - excludes the number of results returned from a SQL statement.

Endpoint /query - SQL with POST

Overview

- Running queries requires a **POST** operation
- Every API call requires the header **Content-Type: application/json**
- Input parameter **sql** gets the request to be executed
- Double quotes inside the SQL command must be escaped with \”
- all parameters can be used as defined

Endpoint /query - SQL with POST as 2D Array with headers

Sample Request

```
POST http://localhost:8080/rest/api/query?headers=true HTTP/1.1
Authorization: Basic YWRtaW46YWRtaW4=
Content-Type: application/json

{ "sql": "select * from \"SYS.Schemas\";" }
```

- Uses POST operation
- Note the *headers=true* parameter in the URL
- Also note the escaped double quotes in the SQL statment

Endpoint /query - SQL with POST as 2D Array with headers

```
[
  [
    "VDBName",
    "Name",
    "IsPhysical",
    "UID",
    "Description",
    "PrimaryMetamodelURI",
    "OID"
  ],...
  [
    "datavirtuality",
    "SYS",
    true,
    "tid:2cb59cfd55db-000142ad-00000000",
    null,
    "http://www.metamatrix.com/metamodels/Relational",
    1
  ]
]
```

Endpoint /query - SQL with POST as JSON Documents

Sample Request

```
POST http://localhost:8080/rest/api/query?array=false HTTP/1.1
Authorization: Basic YWRtaW46YWRtaW4=
Content-Type: application/json

{ "sql": "select * from \"SYS.Schemas\";" }
```

- Note the *array=false* parameter in the URL
- This will generate a list of JSONs

Endpoint /query - SQL with POST as JSON Documents

```
[
  {
    "VDBName": "datavirtuality",
    "Name": "SYS",
    "IsPhysical": true,
    "UID": "tid:2cb59cfd55db-000142ad-00000000",
    "Description": null,
    "PrimaryMetamodelURI": "http://www.metamatrix.com/metamodels/Relational",
    "OID": 1
  }, ...
  {
    "VDBName": "datavirtuality",
    "Name": "file_src",
    "IsPhysical": true,
    "UID": "tid:d846c0077155-d42dc4a1-00000000",
    "Description": null,
    "PrimaryMetamodelURI": "http://www.metamatrix.com/metamodels/Relational",
    "OID": 15
  }
]
```

Endpoint /query - Paginated query (1st query)

Sample Request

```
POST http://localhost:8080/rest/api/query?array=false&pagination=true&limit=10&offset=0 HTTP/1.1
Authorization: Basic YWRtaW46YWRtaW4=
Content-Type: application/json

{ "sql": "select * from \"SYS.Schemas\";" }
```

- we keep *array=false*, as it is prettier
- Note that we used
  - pagination=true to enable pagination for this query
  - limit=10 to get always 10 records (also known as page size)
  - offset=0 so we don't skip any records

Endpoint /query - Paginated query (1st query)

Sample Response Header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
requestId: 87631
limit: 10
offset: 0
prevPage: http://localhost:8080/rest/api/query?requestId=87631&limit=10
nextPage: http://localhost:8080/rest/api/query?requestId=87631&limit=10&offset=10
Content-Type: application/json
Date: Mon, 07 Sep 2020 11:41:10 GMT
Connection: close
```

- the server assigned a requestId which we can use to iterate over the result set
- alternatively, we can read the header nextPage and use this URL for the next page of results

Endpoint /query - Paginated query (2nd query)

Sample Request with POST

```
POST http://localhost:8080/rest/api/query?requestId=87631&limit=10&offset=10 HTTP/1.1
Authorization: Basic YWRtaW46YWRtaW4=
Content-Type: application/json

{}
```

Sample Request with GET

```
GET http://localhost:8080/rest/api/query?requestId=87631&limit=10&offset=10 HTTP/1.1
Authorization: Basic YWRtaW46YWRtaW4=
```

Note that both requests use the **requestId** and no repetition of the original query is necessary

Demo - Run queries

Implement unsupported operations?

Missing an endpoint? Add it! (sort of)

Workarounds and customization

- The /source endpoint can be used to easily implement access to data and flows that are not provided by the REST API by default
- Remember that /source/<dataSourceOrSchema>/<object> can be used to read data or run a stored procedure

Examples

- Drop a schema based on its name (instead of its id)
- Refresh all ‘FAILED’ data sources (via LOOP or TABLE)
- Trigger a job run based on the job description

Demo - ‘Customization’

Summary

- How to call the API
- /status endpoint
- /source endpoint
- /query endpoint
- custom procedures to implement ‘desired’ functions

Questions?

Where to get help

Built-In documentation on the server
http://<yourHostOrIP>:8080/rest/ (on-premise)
http://<yourHostOrIP>/rest/ (DV hosted)

Help Center
https://support.datavirtuality.com/hc

Community
https://support.datavirtuality.com/hc/en-us/community/topics

Thank you!

Please feel free to contact us at:
info@datavirtuality.com

or

visit us at:
datavirtuality.com