

INFOWATCH

InfoWatch Traffic Monitor SDK. Uploading DB unloadings and sample documents (REST API SDK)

26/03/2021

© JSC “InfoWatch”

Tel/Fax +7 (495) 229-00-22

http://www.infowatch.com

Table of Contents

1 Purpose of REST API programming interface....4

2 General description of REST API programming interface....5

2.1 Creating sample unloading....5

2.2 Updating REST API sample unloading....5

2.3 Adding contents to REST API sample unloading....5

2.4 Getting state of REST API sample unloading....5

2.5 Initiating deployment of uploaded sample documents' new content to analysis modules....6

2.6 Getting interface version supported by Traffic Monitor....6

2.7 Creating sample documents catalogue....6

2.8 Deleting sample documents catalog....6

2.9 Creating sample document....7

2.10 Deleting sample document....7

2.11 Adding sample document content....7

2.12 Getting state of sample document....7

3 Format of sample unloading and REST API sample document....8

4 Registering REST API external component....9

4.1 REST API third-party component registry file format....9

5 Configuring REST API programming interface....11

6 Typical scenarios of using API....12

6.1 Typical scenario of creating a sample unloading....12

6.2 Typical scenario of updating a sample unloading....13

6.3 Typical scenario of adding data to unloading....13

6.4 Sample unloading loading process implementation....14

6.5 Typical scenario of sample documents catalog creation based on file system catalog....14

6.6 Typical scenario of synchroonization between sample documents catalog and file system....16

7 Functional description of REST API programming interface....18

7.1 Common headings for GET/POST requests....18

7.2 Working with configuration....18

7.2.1 Distribution of REST API configuration ... 18
7.3 Working with sample unloadings ... 18
7.3.1 Creating the unloading ... 19
7.3.2 Updating data of the unloading ... 20
7.3.3 Adding data for the unloading ... 21
7.3.4 Getting state of the unloading ... 21
7.3.5 Getting current version ... 22
7.4 Common errors ... 24
7.4.1 Successful response ... 24
7.4.2 Data/version validation error ... 24
7.4.2.1 Error format in case of incorrect version of API (JSON Scheme) ... 24
7.4.2.2 Response format in case of request validation error (JSON Scheme) ... 25
7.4.2.3 Example of error when incorrect version is specified ... 27
7.4.2.4 Example of error when checking uniqueness of a sample unloading name ... 27
7.4.3 Authorization is required ... 28
7.4.4 Insufficient permissions ... 28
7.4.5 Entity is not found ... 28
7.4.6 Request body is too large ... 28
7.4.7 Number of server requests is exceeded ... 28
7.4.8 System error ... 29
7.4.9 Server is temporarily unavailable ... 30

1 Purpose of REST API programming interface

This programming interface is designed to integrate third-party components with InfoWatch Traffic Monitor (Traffic Monitor). Third-party components can generate documents which represent "Database sample unloadings" and "Sample documents" and then pass them to Traffic Monitor using the provided interface. After that, the passed unloadings will be detected in data streams that are analyzed by Traffic Monitor. Rules for detection of unloadings are set in Traffic Monitor console. A third-party component can only upload and update contents of the uploading. The third-party component must be registered in the system according to the procedure that is described in "Registering REST API external component".

2 General description of REST API programming interface

The programming interface is a REST API, that is a sequence of requests to a web server. The programming interface provides the following features:

- Creating sample unloading
- Updating EST API sample unloading
- Adding contents to REST API sample unloading
- Getting state of REST API sample unloading
- Initiating deplyment of upoaded sample documents' new content to analysis modules
- Getting interface version supported by Traffic Monitor
- Creating sample documents catalogue
- Adding sample document content
- Getting state of sample document

2.1 Creating sample unloading

```
FingerPrintDsc Create(string name, string[] columns)
```

The function creates a sample unloading with a specified name and a list of column names of the sample unloading. Returns a description of the created sample object (FingerPrint).

2.2 Updating REST API sample unloading

```
FingerPrintDsc UpdateContent(string FingerPrintId,string[]content)
```

The function allows you to upload contents to an existing sample object (unloading). The sample object is determined by its identifier (FingerPrintDsc field of the structure). Returns a description of the sample object (FingerPrint). Once the contents are uploaded, the process of converting and saving the contents is initiated. This function completely replaces the contents of a sample object.

2.3 Adding contents to REST API sample unloading

```
FingerPrintDsc AddContent(string FingerPrintId, string[] content)
```

The function allows you to add extra contents to an existing sample object (unloading). The sample object is determined by its identifier (FingerPrintDsc field of the structure). Returns a description of the sample object (FingerPrint). Once the contents are uploaded, the process of converting and saving the contents is initiated. This function completely replaces the contents of a sample object.

2.4 Getting state of REST API sample unloading

```
FingerPrintDsc GetInfo(string FingerPrintId)
```

The function allows you to receive information about existing sample unloading. Sample object description (FingerPrint) is returning. The important result of calling this function is a field containing status of sample unloading.

2.5 Initiating deployment of uploaded sample documents' new content to analysis modules

```
Void Ditribute()
```

The function initiates the procedure of deploying new contents of sample unloadings to analisys modules. The function is synchronous. Once the function is executed, Traffic Monitor is ready to detect new contents in data streams it analyzes.

2.6 Getting interface version supported by Traffic Monitor

```
VersionInfo GetInterfaceVersion()
```

The function allows you to receive information about the interface supported by Traffic Monitor.

Each part of the unloading, which is uploaded to Traffic Monitor, passes stages of conversion to an internal format and saving to a database. After that, an external component initiates passing the new unloading to Traffic Monitor systems of analysis. Thus, a sufficiently large unloading can be uploaded in parts. That is, it takes several requests to upload the unloading and then a single command is used to deploy it to systems of analysis.

Each step of processing the uploaded sample unloading takes some time. At each step, an error may occur.

The programming interface allows you to track the steps of processing a sample unloading and respond to errors that may occur during processing.

2.7 Creating sample documents catalogue

`CatalogId CreateEtDocCatalog(CatalogId parentCatalog, string name);`

The function is intended to create a sample documents catalogue, returning the created catalogue's identifier, where:

- `parentCatalog` - parent directory, where the catalogue is created. (0 - to create in the root directory).
- `name` - name of sample documents catalogue. This name will be displayed in TM management console.

`CatalogId GetEtDocCtalogIdByName(string name);`

The function is intended to receive the catalogue's ID by name, returning the created catalogue's identifier.

`CatalogList GetEtDocCtalogs();`

The function is intended to receive the catalogues, created by plugin, returning the list of catalogues.

2.8 Deleting sample documents catalog

```
void DeleteEtDocCatalog(CatalogId id)
```

The fuinction is intended to delete sample documents catalog if it is a part of protected object.

The function returns the deleted catalog's id.

2.9 Creating sample document

```
DocumentId CreateEtDoc(CatalogId parentCatalog, string name, Type type);
```

The function is intended to create a sample document, returning sample document ID, where:

- `parentCatalog` - the catalog ID, where sample document is created. Cannot be set to "0" since documents cannot be created in the root directory.
- `name` - name of sample document. Sample document cane be created in accordance with intercepted text or its binary representation.

2.10 Deleting sample document

```
void DeleteEtDoc(DocumentId doc);
```

The function is intended to delete a sample document if it is a part of protected object.

2.11 Adding sample document content

```
void AddEtDocContent(DocumentId doc, Buffer buf);
```

The function is intended to add sample document content, thus letting to download the content by parts, where:

- `doc` - sample document identifier, which will be completed by content;
- `buf` - data buffer, aded to document.

```
void UpdateEtDocContent(DocumentId doc, Buffer buf);
```

The function is intended to update or replace sample document content, where:

- `doc` - sample document identifier, where the content is changed;
- `buf` - data buffer, being a new content of sample document.

2.12 Getting state of sample document

```
DocState GetEtDocState(DocumentId doc);
```

The function is intended to get the state of sample document, where:

- `doc` - sample document identifier.

3 Format of sample unloading and REST API sample document

A sample unloading is a sequence of rows in UTF-8 encoding. The rows should be separated by CRLF or LF characters. Cell values in the row should be separated by TAB or ‘,’ (comma) characters. These limitations follow TSV (Tab separated values) CSV (Comma separated values) format. The number of cells in all rows must be the same. When you create an unloading, the number of cells must be specified. It is equal to the number of column names, which are passed to the *Create* function. It is only possible to upload contents to an existing sample unloading if the number of cells is equal to the number specified during the unloading creation. If a different number of cells is specified it will make the unloading invalid.

A sample document is a file of any format. If the file is not an encrypted archive, the System extracts the files from the archive, process them with separately from each other and creates a separate sample document basing on each extracted file (a sample document cannot be created based on the whole archive). All created sample documents are placed into the catalog, selected while creating a sample document. If the file is an encrypted archive, sample document cannot be created by System. If the file is an envelope, System will create one sample document basing on the file.

4 Registering REST API external component

To integrate an external data interception module with Traffic Monitor system, perform the following steps:

1. Register with InfoWatch company the identifier string of vendor company producing a component. This will link the vendor company with a license that will be installed in the Traffic Monitor system.
2. Get a developer license linked with the company identifier. This will enable loading sample unloadings from this vendor to Traffic Monitor. The developer license allows you to load any sample unloadings.
3. Create an external component registry file as per below. By this moment, the data source names for sample unloading should be already defined, which is to be generated by the external component. The license obtained as per step 2 is added to the registry file.
4. Load the external component registry file to Traffic Monitor system via Traffic Monitor console interface. For details, see the *Administration -> Plugins* section in "*InfoWathc Traffic Monitor. User Guide*";
5. Get the access token within the properties of the external component loaded.

Now the System is ready to receive generated sample unloadings from the specified vendor company with the specified source of the sample unloading.

4.1 REST API third-party component registry file format

The plugin is an archive in a ZIP format. The archive includes:

- **licenses** folder that contains license files;
- **icon** folder that contains files with pictograms for registered events;
- **manifest.json** file with information on a plugin.

> **Note:**
>
> Folders **licenses** and **icon** are optional. License files and files with pictograms can be located in the root folder.

For events source external systems, **manifest.json** file should contain the following information:

Content	Type of data	Optional	Description
`{`			
`"PLUGIN_ID": "",`	String	Yes	Unique plugin identifier (UUID). Up to 40 characters. **Example:** `"PLUGIN_ID": "189C38D390396EB6E0530100007F1CA200000001",`
`"DISPLAY_NAME": "",`	String	Yes	Plugin name displayed. **Example:** `"DISPLAY_NAME": "Plugin name",`
`"DESCRIPTION": "",`	String	No	Plugin name displayed. **Example:** `"DESCRIPTION": "Test plugin to demonstrate",`

Content	Type of data	Optional	Description
`"VERSION": "",`	String	Yes	Plugin version. **Example** `"VERSION": "1.0.0",`
`"VENDOR": "",`	String	Yes	Software company ID in accordance with the company name specified in the license. **Example** `"VENDOR": "infowatch",`
`"LICENSE": [ { "PATH":"" } ],`	Array String	Yes	License files. The paths to license files hould be specified in accordance with archive root folder. **Example:** `"LICENSE": [ { "PATH":"licenses/license file 1.license" }, { "PATH":"licenses/license file 2.license" }, ],`
`"PATTERN_SEARCH_LICENSE": "",`	String	No	Template to search previously downloaded licenses to be binded to plugin. It has the following format: `{operator:or|and,child[{name:value}]}`. **Example:** `"PATTERN_SEARCH_LICENSE": "{\"operator\":\"or\",\"conditions\":[{\"origin\":\"dm\"},{\"origin\":\"dmmobile\"}]}",`
`"DATATYPE": [ { "VALUE": "" } ]`	Array String	Yes	List of data types specifying the source systems. **Example:** `"DATATYPE": [ { "VALUE": "type1_s1", "VALUE": "type2_s2" } ],`
`}`			

Important!

The file contains an expired license. To use the sample file for registration of a plugin, you need to obtain a new license.

A full JSON scheme is a part of SDK samples (see "Description of PushAPI SDK samples"). A sample file of the module registration is included in the SDK.

5 Configuring REST API programming interface

The following parameters participate in configuring the programming interface:

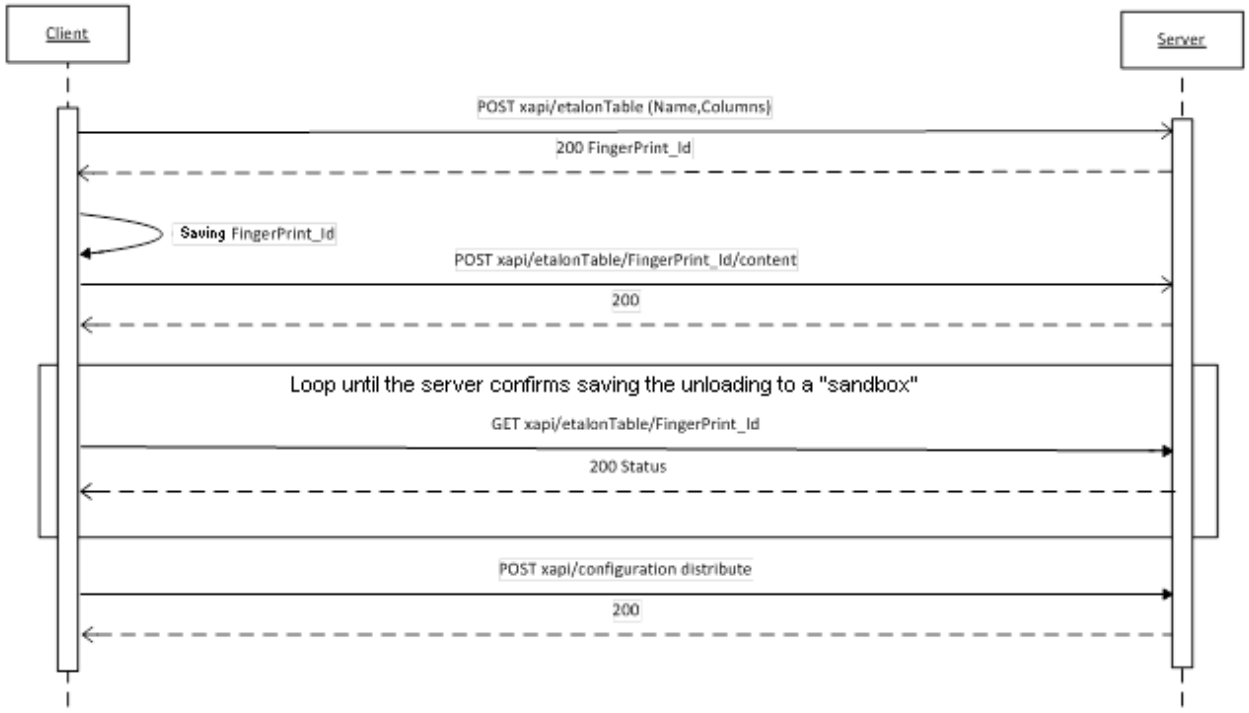
- Address for connection to the SDK service (network address, port);
- "Authorization" token. A string of characters, which allows you to authorize components that use the SDK. This string can be obtained in the control panel by external automated systems in Traffic Monitor management console;
- Identifier of transmitted data. A string received by a vendor of the external component from InfoWatch. This string describes the type of transmitted data and is used in the licensing policy;
- Identifier of a vendor of the external component. A string received by a vendor of the external component from InfoWatch. This string describes the vendor and is used in the licensing policy.

6 Typical scenarios of using API

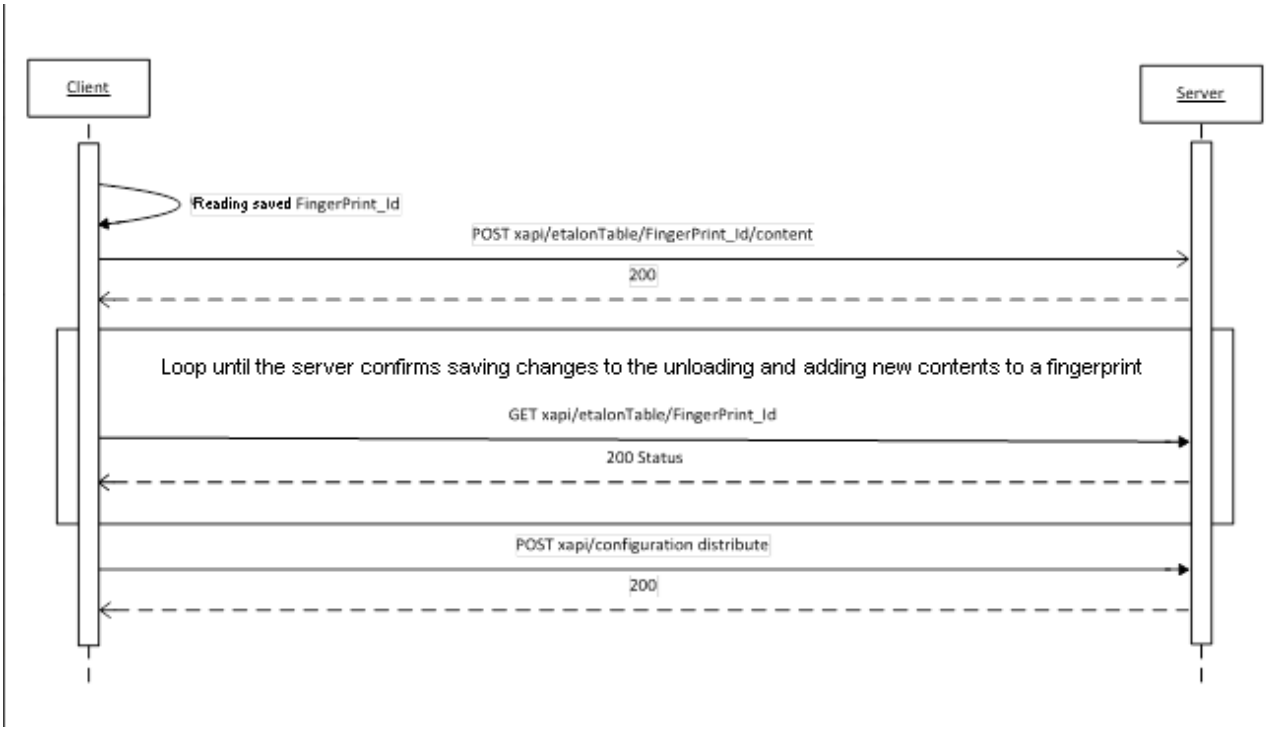
Below you can find sequence diagrams which represent typical scenarios of using REST API:

- Typical scenario of creating a sample unloading
- Typical scenario of updating a sample unloading
- Typical scenario of adding data to unloading
- Sample unloading loading process implementation
- Typical scenario of sample documents catalog creation based on file system catalog
- Typical scenario of synchroonization between sample documents catalog and file system

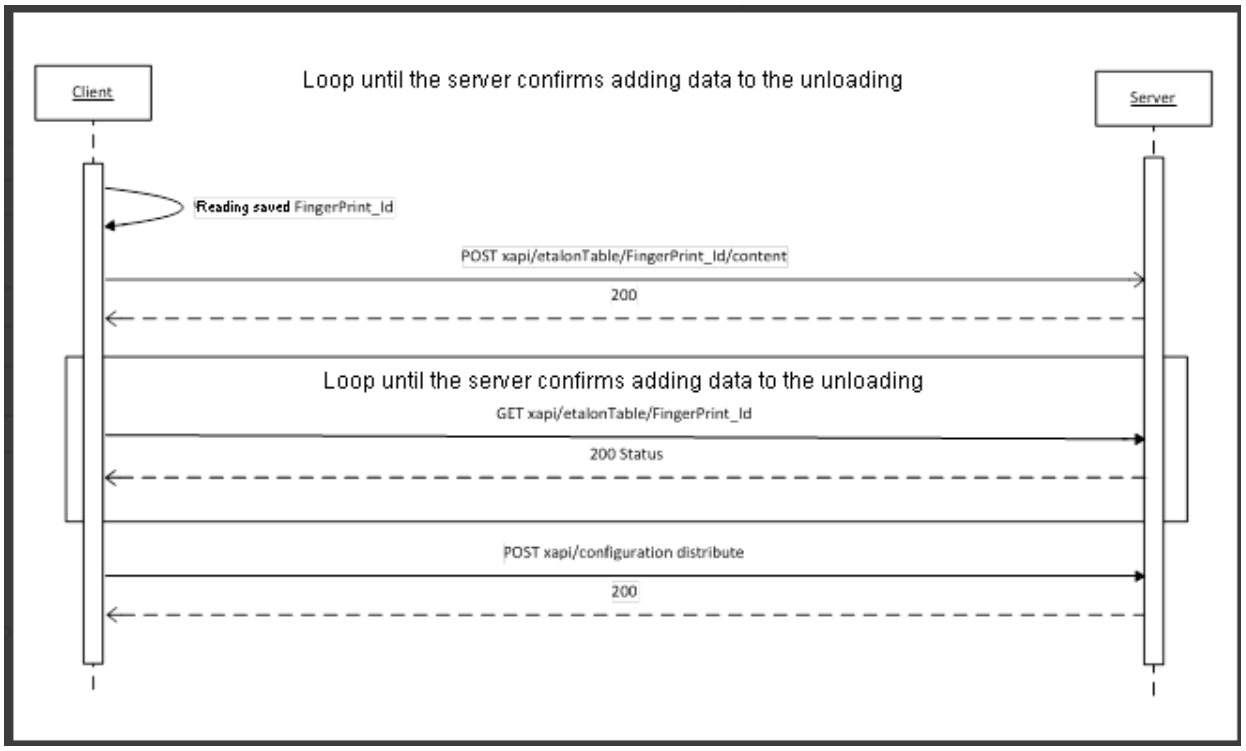
6.1 Typical scenario of creating a sample unloading



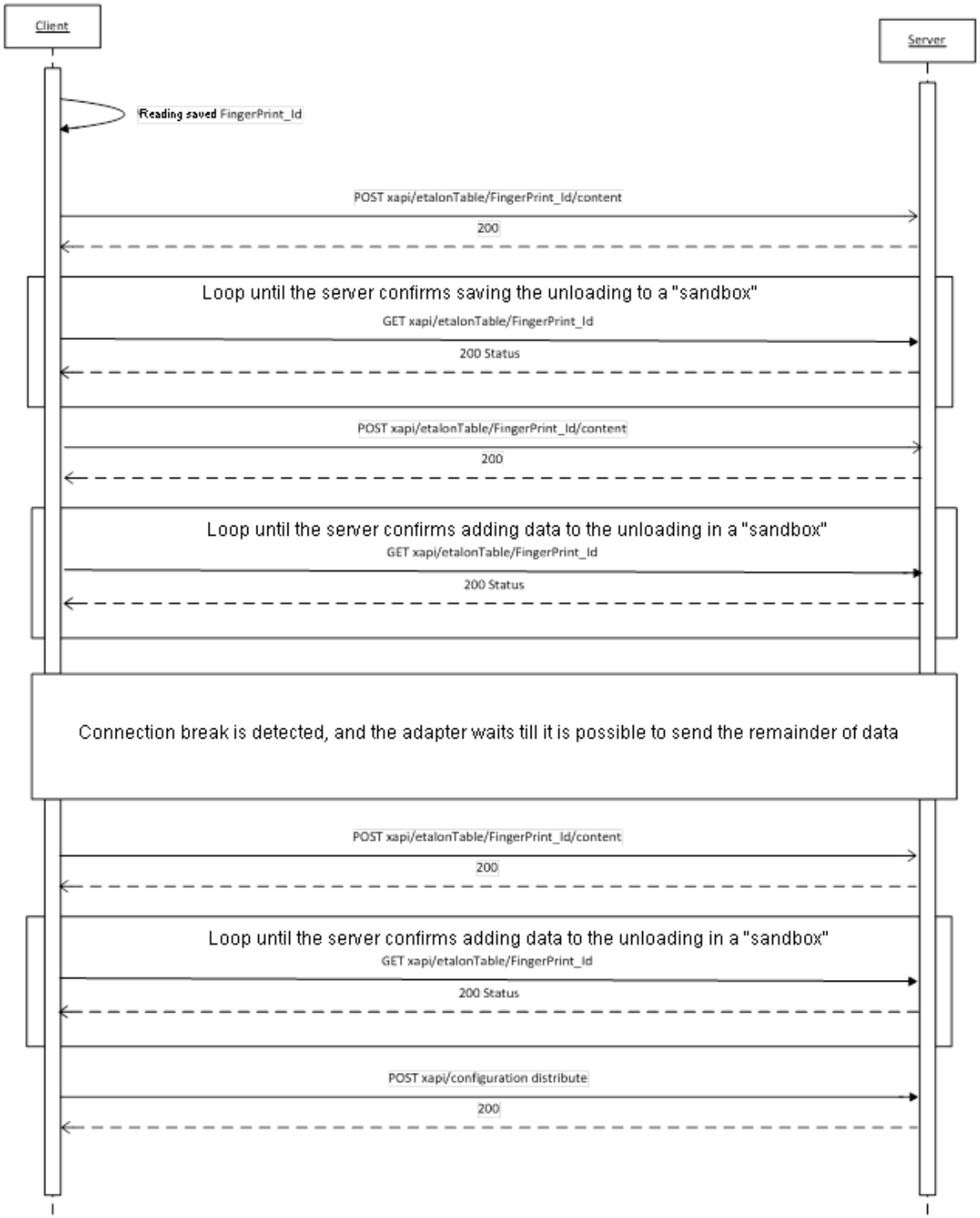
6.2 Typical scenario of updating a sample unloading



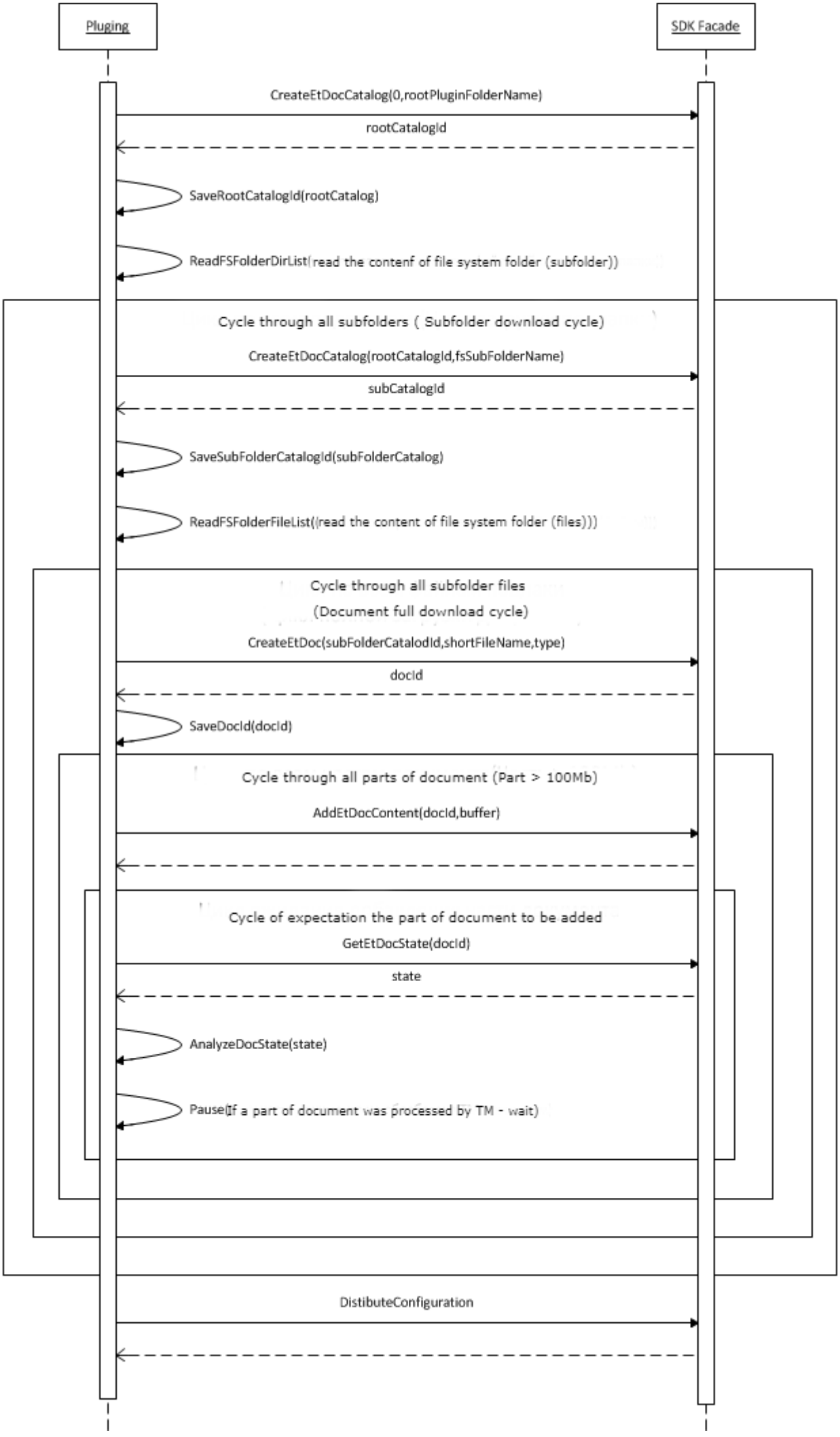
6.3 Typical scenario of adding data to unloading



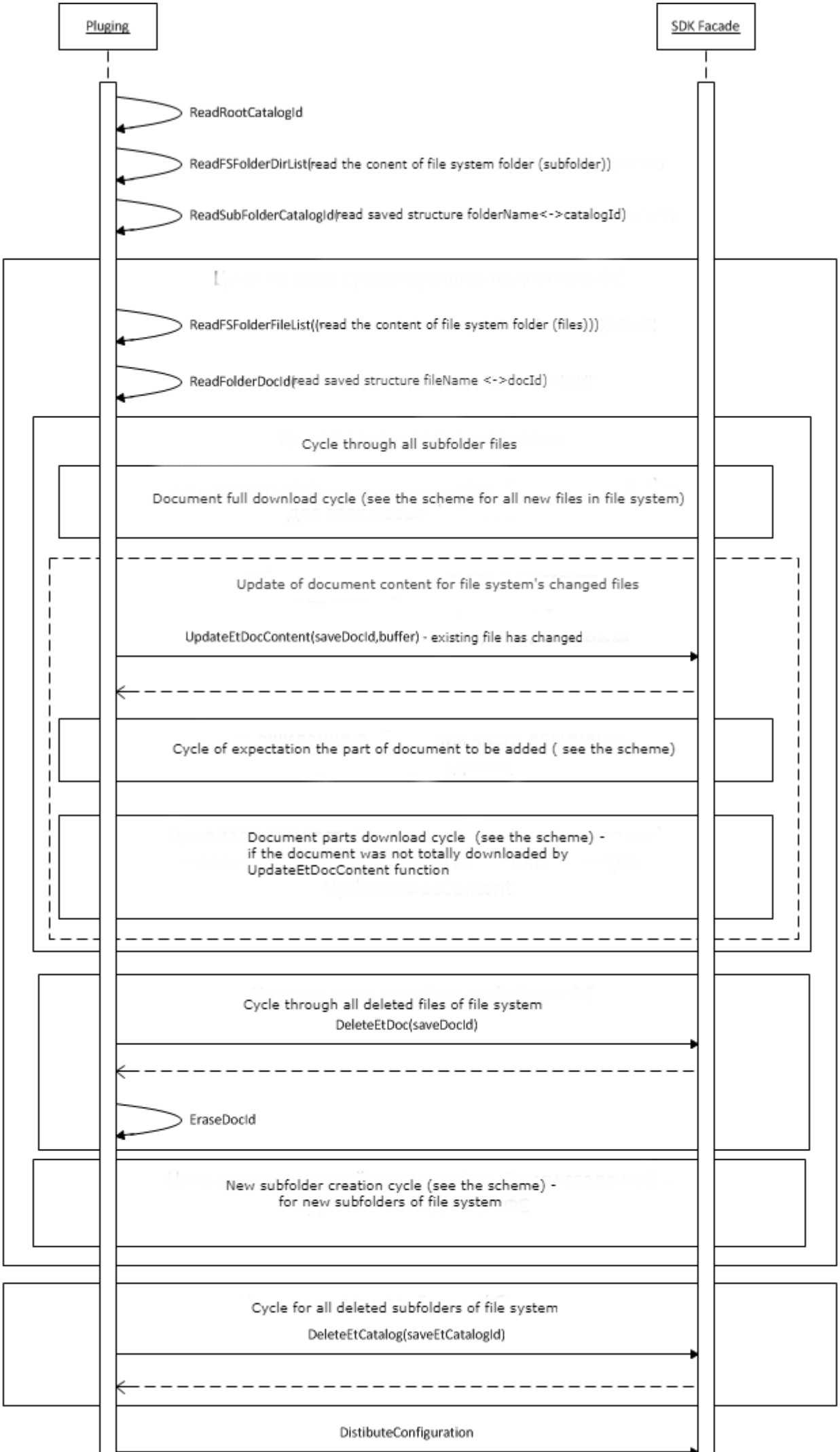
6.4 Sample unloading loading process implementation



6.5 Typical scenario of sample documents catalog creation based on file system catalog


Pluging
SDK Facade
CreateEtDocCatalog(0,rootPluginFolderName)
rootCatalogId
SaveRootCatalogId(rootCatalog)
ReadFSFolderDirList(read the contenf of file system folder (subfolder))
Cycle through all subfolders (Subfolder download cycle)
CreateEtDocCatalog(rootCatalogId,fsSubFolderName)
subCatalogId
SaveSubFolderCatalogId(subFolderCatalog)
ReadFSFolderFileList((read the content of file system folder (files)))
Cycle through all subfolder files
(Document full download cycle)
CreateEtDoc(subFolderCatalodId,shortFileName,type)
docId
SaveDocId(docId)
Cycle through all parts of document (Part > 100Mb)
AddEtDocContent(docId,buffer)
Cycle of expectation the part of document to be added
GetEtDocState(docId)
state
AnalyzeDocState(state)
Pause(If a part of document was processed by TM - wait)
DistibuteConfiguration

6.6 Typical scenario of synchroonization between sample documents catalog and file system


Pluging
SDK Facade
ReadRootCatalogId
ReadFSFolderDirList(read the conent of file system folder (subfolder))
ReadSubFolderCatalogId(read saved structure folderName<->catalogId)
ReadFSFolderFileList((read the content of file system folder (files)))
ReadFolderDocId(read saved structure fileName <->docId)
Cycle through all subfolder files
Document full download cycle (see the scheme for all new files in file system)
Update of document content for file system's changed files
UpdateEtDocContent(saveDocId,buffer) - existing file has changed
Cycle of expectation the part of document to be added (see the scheme)
Document parts download cycle (see the scheme) -
if the document was not totally downloaded by
UpdateEtDocContent function
Cycle through all deleted files of file system
DeleteEtDoc(saveDocId)
EraseDocId
New subfolder creation cycle (see the scheme) -
for new subfolders of file system
Cycle for all deleted subfolders of file system
DeleteEtCatalog(saveEtCatalogId)
DistibuteConfiguration

7 Functional description of REST API programming interface

The programming interface is a REST API, that is a sequence of GET/POST requests to a specific resource of a web server and responses to them. The request and response formats are described below. Data protection during the transfer is carried out through the SSL/TLS protocol without the client's authorization. Transferring unencrypted data is not supported. For data transfer, a UTF-8 encoding is used.

7.1 Common headings for GET/POST requests

Each request to the server must contain the following obligatory headings:

- X-API-Auth-Token – the authorization token is specified;
- X-API-DataType – an identifier of the transferred data type is specified;
- X-API-CompanyId – an identifier of the vendor is specified;
- X-API-Version – a version of the programming interface is specified. For now, we use number 1.

7.2 Working with configuration

After the content of the sample unloading has been sent to the SDK and successfully compiled, you can initiate distribution of the modified configuration to the analysis systems.

7.2.1 Distribution of REST API configuration

Description: the method is required to distribute configuration to analysis subsystems.

Method features: POST, synchronous.

Resource: xapi/configuration.

JSON Scheme request body:

```
{
    "STATUS": "distribute"
}
```

In case of success – the return code is "200" and the response body is as follows:

```
{
    "STATUS": "distributed"
}
```

In case of an error – the return code is "500". Possible values of the `code` field:

- error – cannot apply the configuration.

7.3 Working with sample unloadings

The chapter contains the following information:

- Creating the unloading
- Updating data of the unloading
- Adding data for the unloading
- Getting state of the unloading
- Getting current version

7.3.1 Creating the unloading

Description: the method is required to add a new unloading without filling it with data. The data will be added with a separate method.

Method features: POST, synchronous.

Resource: xapi/etalonTable.

JSON Scheme request body:

```
{
    "properties": {
        "DISPLAY_NAME": {
            "description": "Name of unloading",
            "type": "string",
            "errors": ["required", "not_match", "not_unique_field"]
        },
        "CONDITION_COLUMNS": {
            "description": "a string, an encoded JSON array with column names, e.g. [\"first_column\", \"second_column\"]",
            "type": "string",
            "errors": ["required", "not_valid_json"]
        },
        "NOTE": {
            "description": "Arbitrary description of the unloading",
            "type": "string"
        }
    },
    "required": [
        "DISPLAY_NAME",
        "CONDITION_COLUMNS"
    ]
}
```

In case of success – the return code is "200" and the response body is as follows:

```
{
    "FINGERPRINT_ID":    "string",
    "TYPE":              "string",
    "SOURCE":            "string",
    "DISPLAY_NAME":      "string",
    "NOTE":              "string",
    "CONDITION_COLUMNS": "string",
```

```
    "CREATE_DATE":      "string",
    "CHANGE_DATE":      "string",
    "STATUS":           "string"
}
```

> **Important!**
>
> The number of columns that was specified when creating the unloading cannot be changed. When loading the contents, the number of columns in the rows being loaded should match the value that was specified when creating the sample unloading.

7.3.2 Updating data of the unloading

Description: the method is required to update the unloading contents. All entries of the sample unloading will be deleted, and then, a new entry will be added. This method can be used for the initial filling of the unloading.

> **Important!**
>
> The filling should have an integer number of rows.
> The minimum number of row for filling is 2.
> The minimum number of columns is 2.
> The minimum byte size of loadable content is 128 byte.
> The minimum byte size of loadable content is 2GB.

Methods features: POST, asynchronous.

Resource: xapi/etalonTable/{id}/content/replace.

Parameters:

- id – fingerprint_id of the object being edited.

Example: enatalonTable/0123456789ABCDEF/content/replce

Content-type of the request body should be not `application/json`, but `multipart/form-data` (for more details, see: https://tools.ietf.org/html/rfc2388).

Content-Disposition that will include the file should have the `name` field with CONTENT value and `filename` attribute with the file name and TSV or CSV extension depending on the file format, separated with Tab or commas respectively.

Possible values of errors for the CONTENT field:

- not_valid – the file has an invalid signature;
- not_allowed_file_extension – an unauthorized file extension, for now. only 2 formats are supported: TSV and CSV;
- too_long_size – the file size is too large;
- duplicate – this file already exists in the system.

In case of success – the return code is "200" and the response body is as follows:

```
{
    "FINGERPRINT_ID":    "string",
```

```
    "TYPE":              "string",
    "SOURCE":            "string",
    "DISPLAY_NAME":      "string",
    "NOTE":              "string",
    "CONDITION_COLUMNS": "string",
    "CREATE_DATE":       "string",
    "CHANGE_DATE":       "string",
    "STATUS":            "string"
}
```

After getting the "200" return code, you should take the FINGERPRINT_ID field and use it for the checking the unloading state, until the state changes to "ready".

> **Important!**
>
> The number of cells in loadable rows should match the number of columns that was specified when creating the unloading.
> When you delete the unloading through Traffic Monitor console, it is considered existing until you distribute the configuration, i.e. you can update this unloading and add contents to it.
> When loading "erroneous" content, i.e. the content which duplicates an already loaded unloading (or its part) or the content whose rows contain an incorrect number of cells, the whole sample object gets an invalid status. The external subsystem should stop loading this content and go to the next one. Loading the correct content changes the sample object status to the correct one. Incorrect content never reaches the analysis modules.

7.3.3 Adding data for the unloading

Description: the method is used for adding content to the unloading. All content entries will be saved, and a new entry will be added.

Method features: POST.

Resource: xapi/etalonTable/{id}/content.

For details, see "Updating data of the unloading".

7.3.4 Getting state of the unloading

Description: the method is required to get information about the state of a single unloading.

Method features: GET.

Ресурс: xapi/etalonTable/{id}.

Parameters:

- id – fingerprint_id of the object being edited.

Example: etalonTable/0123456789ABCDEF

Response body:

```
{
```

```
    "FINGERPRINT_ID":    "string",
    "TYPE":              "string",
    "SOURCE":            "string",
    "DISPLAY_NAME":      "string",
    "NOTE":              "string",
    "CONDITION_COLUMNS": "string",
    "CREATE_DATE":       "string",
    "CHANGE_DATE":       "string",
    "STATUS":            "string"
    "content":           [
        {
            "CONTENT_ID":   "string",
            "CHECKSUM":     "string",
            "CONTENT_SIZE": "integer",
            "CONTENT_MIME": "string"
        }
    ]
}
```

The `status` field can have the following values:

- sending – sending data to the compiler;
- sent – data has been sent to the compiler, and compilation has been started;
- compiling – the document is being compiled;
- compiled – the document is ready;
- saving – the document is being saved;
- ready – the document has been saved, it is ready for further changes and does not block applying the configuration;
- fatal_error – an error occurred when saving the document, the operation cannot be repeated or continued;
- duplicate – the content data is duplicated (loading the content can be continued);
- not_match_delimiter – rows in the unloading do not contain delimiters (COMMA or TAB);
- wrong_column_count – the number of columns in the row does not match the number of columns that was specified when creating the unloading;
- error – a temporary error, you can try the operation again in some time (better in an hour);
- max_words - the maximum number of words is exceeded;
- max_columns - the maximum number of columns is exceeded.

7.3.5 Getting current version

Description: the method is required for getting supported versions.

If the return code is "200", you can continue working with the system.

You can use this method to check a license, a version or the authorization. In case of success, the return code is "200". In case of an error, the error code is returned.

Method features: GET.

Resource: xapi/version.

Response body:

```
{
    "data":       [
        {
            "NAME":    "string",
            "STATUS": "supported|deprecated|removed"
        }
    ]
}
```

An example of response if incorrect version is specified:

HTTP/1.1 400 Bad Request

Content-Type: application/json

```
{
    "error": {
        "code": "unsupported_version",
        "meta": {
            "versions": [
                {
                    "NAME":   "1.0.0",
                    "STATUS": "removed"
                },
                {
                    "NAME":   "2.0.0",
                    "STATUS": "deprecated"
                },
                {
                    "NAME":   "3.0.0",
                    "STATUS": "supported"
                }
            ]
        }
    }
}
```

7.4 Common errors

Description of possible errors:

- not_unique_field – not a unique value;
- unique – a unique value;
- too_short – too short value;
- too_long – too long value;
- wrong_length – the value has incorrect length;
- repeat_exactly – the value does not match the value in a different field;
- required – the field value is required;
- not_exist_in_list – the value is not in the list;
- invalid – invalid value;
- contains_child – the node has child elements;
- contains_items – the category node contains entities;
- parent_own_child – the parent contains child elements;
- parent_self_set_current – the object tries to set itself its parent;
- not_valid_json – invalid JSON;
- invalid_cron_format – incorrect cron format;
- read_only – the object is readonly;
- empty – the value cannot be empty.

7.4.1 Successful response

The return code is "200".

Response format:

```
{
    "data": "mixed",
    "meta": "mixed"
}
```

7.4.2 Data/version validation error

The return code is "400".

Error format in case of incorrect version of API (JSON Scheme)

```
{
    "properties": {
        "error": {
            "description": "Error container",
            "properties":  {
                "code":       {
                    "description": "General description of error",
                    "type":        "string",
```

```
                    "value":       "unsupported_version"
                },
                "meta": {
                    "description": "Additional information on error",
                    "type":        "object",
                    "properties":  {
                        "supported_versions": {
                            "description": "Array of supported versions",
                            "type":        "array",
                            "items": {
                                "description": "Supported version",
                                "type": "string"
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Response format in case of request validation error (JSON Scheme)

```
{
    "properties": {
        "error": {
            "description": "Error container",
            "properties":  {
                "code":       {
                    "description": "General description of error",
                    "type":        "string",
                    "value":       "validation"
                },
                "meta": {
                    "description": "Additional information on error",
                    "type":        "object",
                    "properties":  {
                        "validation": {
```

```
                        "description": "Names of model fields",
                        "type":        "object",
                        "properties":  {
                            "field":     {
                                "description": "Model field",
                                "type":        "array",
                                "items":       {
                                    "description": "Error",
                                    "type":        "string",
                                    "values": [
                                        "not_unique_field",
                                        "too_short",
                                        "too_long",
                                        "wrong_length",
                                        "repeat_exactly",
                                        "required",
                                        "not_exist_in_list",
                                        "invalid",
                                        "contains_child",
                                        "contains_items",
                                        "parent_own_child",
                                        "unique",
                                        "not_valid_json",
                                        "invalid_cron_format",
                                        "read_only",
                                        "parent_self_set_current",
                                        "empty"
                                    ]
                                }
                            },
                            "sub_model": {
                                "description": "Name of nested model",
                                "type":        "object",
                                "$ref":       "#/properties/error/properties/meta/
properties/validation"
                            }
                        }
```

```
                        }
                    }
                }
            }
        }
    }
}
```

Example of error when incorrect version is specified

```
{
    "error": {
        "code": "unsupported_version",
        "meta": {
            "supported_versions": ["2", "3"]
        }
    }
}
```

Example of error when checking uniqueness of a sample unloading name

```
{
    "error": {
        "code": "validation",
        "meta": {
            "validation":    {
                "DISPLAY_NAME": ["unique"]
            },
            "unique_models": [
                {
                    "FINGERPRINT_ID": "0123456789ABCDEF",
                    "DISPLAY_NAME":   "test"
                }
            ]
        }
    }
}
```

7.4.3 Authorization is required

The return code is "401".

Description: is returned in all requests which require authorization.

Response body: no.

7.4.4 Insufficient permissions

The return code is "403".

Description: is returned for all requests if the current user does not have sufficient permissions to perform a given operation.

Response body: no.

7.4.5 Entity is not found

The return code is "404".

Description: is returned if a particular model is not found in the system. For example, a sample unloading is not found.

Response body: no.

7.4.6 Request body is too large

The return code is "413".

Description: is returned if a too large request is transferred.

Response body: no.

7.4.7 Number of server requests is exceeded

The return code is "429".

Description: is returned if too many requests have been sent to the server over a certain period. May include Retry-After heading which specifies in how much time you can repeat the request.

Response body (JSON Scheme):

```
{
    "properties": {
        "error": {
            "description": "Error container",
            "properties":  {
                "code": {
                    "description": "General description of error",
                    "type":        "string",
                    "value":       "too_many_requests "
```

```
                },
                "meta": {
                    "description": "Additional information on error",
                    "type":        "object",
                    "properties":  {
                        "time": {
                            "description": "HTTP date or an integer in seconds, are which
you can repeat the request",
                            "type":        "string"
                        }
                    }
                }
            }
        }
    }
}
```

7.4.8 System error

The return code is "500".

Description: is returned if a logical error has occurred in the system, and the request execution cannot be continued.

Response body (JSON Scheme):

```
{
    "properties": {
        "error": {
            "description": "Error container",
            "properties":  {
                "code": {
                    "description": "General description of error",
                    "type":        "string"
                },
                "meta": {
                    "description": "Additional information on error",
                    "type":        "object"
                }
            }
        }
    }
```

```
}
```

7.4.9 Server is temporarily unavailable

The return code is "503".

Description: informs that currently is not possible to execute requests and the server is down for maintenance. The Retry-After heading contains estimated time of completion of maintenance (the heading value can be either HTTP-date or an integer in seconds).

Response body (JSON Scheme):

```
{
    "properties": {
        "error": {
            "description": "Error container",
            "properties":  {
                "code": {
                    "description": "General description of error",
                    "type":        "string",
                    "value":       "maintains"
                },
                "meta": {
                    "description": "Additional information on error",
                    "type":        "object",
                    "properties":  {
                        "time": {
                            "description": "HTTP-date or integer in seconds after which
you can repeat the request",
                            "type":        "string"
                        }
                    }
                }
            }
        }
    }
}
```