OS2sync

API Documentation

Version: 2.1.0
Date: 18.05.2020
Author: BSG

Content

1 Introduction 3
2 API overview 4
2.1 Service API 4
2.2 SDK API 4
2.2.1 Security 5
2.2.2 Cvr 5
2.3 SQL API 5
2.4 Registration objects 5
2.4.1 UserRegistration 6
2.4.2 OrgUnitRegistration 7
2.5 The operations 11
2.6 Error handling 11
2.6.1 Information about KOMBIT status codes 12
3 Prerequisites and configuration 14
3.1 SQL and Service API 14
3.2 SDK API 14
4 API Usage 15
4.1 Service API 15
4.1.1 Maintaining users 15
4.1.2 Maintaining organizational units 16
4.2 SDK API 17
4.2.1 Initialization 17
4.2.2 Maintaining users 17
4.2.3 Maintaining organizational units 18
4.3 SQL API 18
5 Success / Error tables 19

1 Introduction

The purpose of this document is it describe the API offered by the OS2sync solution in technical details, so the reader is fully equipped to use the OS2sync APIs.

This document consists of 4 chapters (besides this), which contains the following information

Chapter 2. This chapter gives a logical overview of the API, and describes all the input fields. Once development has started, this is the reference chapter to lookup when in doubt about data and behavior.

Chapter 3. This chapter describes the steps necessary to setup the development environment.

Chapter 4. This chapter describes how to use the 3 different physical APIs, showing example code on the various operations.

Chapter 5. This chapter describes the two tools that can be used during development to test that data is synchronized correctly.

2 API overview

The API is a logical API, with three physical implementations (called Service, SQL and SDK). The document focuses on the logical API, but covers the concrete physical implementations where relevant, for instance if a given physical implementation deviates from the other.

2.1 Service API

The Service API is deployed as a REST/JSON web service, with the following endpoints (<server> is the FQDN of the server where the service is deployed)

http://<server>:5000/api/user

http://<server>:5000/api/orgUnit

These are for managing individual objects.

The HTTP methods POST and DELETE are used for synchronizing objects of the corresponding types. E.g. to create a user, perform a HTTP POST with UserRegistration object in the body to

http://<server>:5000/api/user

And to delete a user with uuid 315a4cdc-f77c-4037-ac3e-b2ecaa7f2f95, perform a HTTP DELETE with an empty object to

http://<server>:5000/api/user/315a4cdc-f77c-4037-ac3e-b2ecaa7f2f95

Finally, it is possible to read data using the GET operation like this

http://<server>:5000/api/user/315a4cdc-f77c-4037-ac3e-b2ecaa7f2f95

The format of the returned object in the GET operation is identical to the format used for calling POST.

2.2 SDK API

The SDK API is a .NET interface, written in C# (but accessible from any language on the .NET platform). The API has similar endpoints to the Service API, but they are implemented as service classes, and can be used like this

```
// the service for performing user operations
UserService userService = new UserService();

// create/update a user object
userService.Update(registration);

// delete a user object
userService.Delete("315a4cdc-f77c-4037-ac3e-b2ecaa7f2f95");
```

As with the Service API, it is possible to use the SDK for reading objects using this operation

```
// read details on a user
var user = userService.Read("315a4cdc-f77c-4037-ac3e-b2ecaa7f2f95");
```

The class instance returned by Read() is of the same class used for calling Update().

2.2.1 Security

It is possible to configure the REST API to require an ApiKey. If that is configured on the installation, a HTTP header called ApiKey must be supplied, containing the configured value.

2.2.2 Cvr

A Cvr number is required when using the API. This can either be configured globally in the installation, or it can be supplied as a HTTP header (The header is called "Cvr").

If no header is supplied, the configured value is used. It is possible to supply a different value than the configured value, for instance when deployed for multi-tenancy purposes.

2.3 SQL API

The SQL API is a set of tables, in which "requests" can be written. Unlike the SDK and Service API's, there is no immediate feedback, but as long as the request conforms to the table constraints, it is considered a valid request, and will be handled by the OS2sync component at some point.

A sample insert for creating or updating a user would be something like this

```
INSERT INTO queue_users (
  user_uuid,
  ... attributes ...,
  operation) VALUES (
  '315a4cdc-f77c-4037-ac3e-b2ecaa7f2f95',
  ... attribute values ...,
  'UPDATE'
);
```

While a delete is simply this

```
INSERT INTO queue_users (
  user_uuid,
  operation) VALUES (
  '315a4cdc-f77c-4037-ac3e-b2ecaa7f2f95',
  'DELETE'
);
```

Note that it is not possible to read data from Organisation using the SQL API

2.4 Registration objects

There are 2 different registration objects, one for each of the API endpoints. Note that the registration objects are only used for update/create operations, for delete and read operations only the UUID of the object is required, and the list operation takes no arguments.

The attributes in the registration objects are identical in the three physical APIs, though the SQL tables are partially denormalized, where the Service/SDK API's has a more normalized structure.

The SDK version of the registration object is shown below, and the fields (which are identical in the three APIs) are described in details

2.4.1 UserRegistration

The registration of a user contains the following attributes

```
public class UserRegistration
{
    // attributes for User object
    public string Uuid;
    public string ShortKey;
    public string UserId;

    // address information for the user
    public string PhoneNumber;
    public string Email;
    public string Location;
    public string RacfID;

    // relevant information about the Users positions in the municipality
    public List<Position> Positions;

    // attributes for Person object
    public Person Person;

    // registration timestamp
    public DateTime Timestamp;
}


// Position structure
public class Position
{
    public string Name;
    public string OrgUnitUuid;
}

// Person structure
public class Person
{
    public string Name;
    public string Cpr;
}
```

The attributes must conform to the following rules

Attribute	Mandatory	Description
Uuid	Yes	This must be a UUID v4 value, and is used to uniquely identify the object. This value cannot be modified later on.
ShortKey	No	This value is a unique, but short (max 50 characters) identifier. If not supplied in the registration, OS2sync will generate one, as Organisation requires this field to be filled out.

UserId	Yes	This is the userId of the user (e.g. the SAMAccountName from Active Directory).
PhoneNumber	No	The phone number of the user.
Email	No	The email address of the user.
Location	No	The physical location of the user (e.g. office number)
RacfID	No	The userId of the user in CICS (for legacy KMD support)
Position	Yes	The list of positions that the user holds within the municipality. The Name attribute must be filled out, and contains the 'title' of the position. There is no functionality tied to the value of this field, and it is used for presentation purposes only. The OrgUnitUuid attribute must be filled out, and contains the UUID of the OrgUnit that this position relates to. A User must have at least 1 position.
Person	Yes	The person object contains the name and potentially the cpr number of the user. The name must be filled out, but the cpr number can be left empty.
Timestamp	No	This is the timestamp of the registration. It is recommended to leave it empty, as it will then default to today. It is not currently possible to register data into the future, as these will be rejected by Organisation. This might be possible in later releases of Organisation.

2.4.2 OrgUnitRegistration

The registration of an OrgUnit contains the following attributes

```
public class OrgUnitRegistration
{
    public string Uuid;
    public string ShortKey;
    public string Name;
    public string ParentOrgUnitUuid;
    public string PayoutUnitUuid;
    public string ManagerUuid;
    public DateTime Timestamp;
    public string PhoneNumber;
    public string Email;
    public string Location;
    public string LOSShortName;
    public string LOSId;
```

```
        public string ContactOpenHours;
        public string EmailRemarks;
        public string Contact;
        public string PostReturn;
        public string PhoneOpenHours;
        public string Ean;
        public string Url;
        public string Landline;
        public string Post;
        public OrgUnitType Type;
        public List<string> Tasks;
        public List<string> ContactForTasks;
    }

    public enum OrgUnitType { DEPARTMENT, TEAM }
```

The attributes must conform to the following rules

Attribute	Mandatory	Description
Uuid	Yes	This must be a UUID v4 value, and is used to uniquely identify the OrgUnit. This value cannot be modified later on.
ShortKey	No	This value is a unique, but short (max 50 characters) identifier for the OrgUnit. If not supplied in the registration, OS2sync will generate one, as Organisation requires this field to be filled out.
Name	Yes	This is the name of the OrgUnit. The value is used for presentation purposes.
ParentOrgUnitUuid	No*	This is the UUID of the OrgUnit that is the parent of this OrgUnit. While it is allowed (technically) to not have a parent, it is required by KOMBIT that only the top-level OrgUnit is parent-less, to ensure that the set of registered OrgUnits are in fact a hierarchy. STSOrgSync does not in any way validate, that the set of OrgUnits that are registered, follow this rule.
PayoutUnitUuid	No	This is a UUID that references an OrgUnit that is a PayoutUnit (udbetalingsenhed). When implementing the it-systems KY and KSD, they require that the municipalities PayoutUnits (Udbetalingsenheder) are known (see the LOSAddress attribte), and that any team that performs payouts on behalf of these units, have a reference to these units. If the OrgUnit being registered, is a team that performs payouts, then this field must be filled

		out, and reference the OrgUnit that corresponds to that specific PayoutUnit. This registration pattern will be described in details as part of the KY and KSD implementations, and these values can be left out until then.
ManagerUuid	No	UUID that references the User object that is the manager for this OrgUnit.
Timestamp	No	This is the timestamp of the registration. It is recommended to leave it empty, as it will then default to today. It is not currently possible to register data into the future, as these will be rejected by Organisation. This might be possible in later releases of Organisation.
PhoneNumber	No	The phone number of the OrgUnit.
Email	No	The email address of the OrgUnit.
Location	No	The physical location of the OrgUnit (e.g. office space).
LOSShortName	No	If this OrgUnit is a PayoutUnit (Udbetalingsenhed), it must have a reference to the corresponding unit in LOS. The value to give here is the KaldenavnKort value from LOS. Please note that as a side-effect of registering this information on the OrgUnit, it will be created as a PayoutUnit, so do not put this value on ordinary OrgUnits.
LOSId	No	This is the ID of the OrgUnit if it comes from LOS. This is used to support certain legacy KMD systems.
ContactOpenHours	No	For contact purposes, it is possible to register at which days and which hours of the day, that this OrgUnit is open for business. KOMBIT will at some point document the exact format that this value must be written in, so for now leave this field empty.
EmailRemarks	No	Some textual message about email contacts. E.g. "It can take up to 2 days before you get a response"
Contact	No	The contact address for this OrgUnit, in case it differs from the actual post address of the OrgUnit.
PostReturn	No	The return address for physical mail that is returned to the OrgUnit.
PhoneOpenHours	No	For contact purposes, it is possible to register at which days and which hours of the day, that this OrgUnit is open for (phone) business.
Ean	No	The EAN number of this OrgUnit (if it has one).
Url	No	This is the website-address of this OrgUnit.
Landline	No	This is the landline phone number of this OrgUnit.
Post	No	This is the post-address of this OrgUnit.

Type	Yes	Must be filled out with one of the values in the enumeration: DEPARTMENT or TEAM. DEPARTMENT is the "normal" value, whereas TEAM is currently only used in DUBU.
Tasks	No	A list of UUIDs referencing the KLE objects in STS Klassifikation. Used for describing which tasks are solved in this OrgUnit.
ContactForTasks	No	A list of UUIDs referencing the KLE objects in STS Organisation. Used for describing which tasks this OrgUnit is the contactplace (Henvendelssted) for.

2.5 The operations

On each of the mentioned objects, four operations can be performed

- **Update**. This operation will either create or update an object of the given type, with the data supplied in the call to the operation
- **Delete**. This operation will (for users and organizational units) perform a "soft-delete" on the object, which will cause the object to change its state to inactive. The object will still exist, but will have the state 'inactive' when read from Organisation. A deleted object is undeleted if an Update operation is called on it.
- **Read**. This operation reads a single object, using the UUID of the object as the key for reading the object. It will return a structure identical to the one used for calling Update().

2.6 Error handling

Error handling depends on the chosen API, but toughly the errors falls into two categories

- Temporary errors (e.g. the Organisation service is down), which should be handled by trying again
- Permanent errors (e.g. invalid input data), which should be handled by fixing the data or the implementation before trying again

SDK API error handling

In case of a temporary error, the API will throw an exception of the following type

```
Organisation.BusinessLayer.TemporaryFailureException
```

The exception will contain both an inner exception, as well as an unstructured textual message, indicating the cause of the temporary error. The caller should wait for a period of time, and then try again (pause all calls to the OS2sync API for maybe 5 minutes, then try again)

It is highly recommended to monitor TemporaryFailureExceptions, and if they occur often, check the logs and see what is wrong. Likely it is the Organisation service that is unavailable, but it could also be an expired certificate, an expired service agreement or something similar which requires human action.

Any other exception type is considered a permanent error, and attempting to call again with the same input will result in the same error.

SQL API error handling

The SQL API rely on input validation (though schema constraints) to ensure that data is well-structured before the data is accepted. The SchedulingLayer will automatically deal with temporary errors, and attempt retries at regular intervals.

As some temporary errors cannot resolve themselves automatically (e.g. an expired certificate), it is important to monitor the logs (or use the monitoring service) for OS2sync, so these errors can be resolved.

Permanent errors are logged as such in the log file.

Service API error handling

The Service API uses an input validation approach similar to the SQL API, and will return HTTP 400 on invalid input. If the input is valid, it will return HTTP 200, and place the request on the queue. The service might return HTTP 500 if it is badly configured, or if some technical issue prevents it from processing the request (look in the log file for details).

As the request processing is dealt with by the SchedulingLayer, the same recommendations for monitoring the log file (or using the monitoring service) goes here.

2.6.1 Information about KOMBIT status codes

The API validates all input data, with the intend of catching as many errors in data as possible, but it does not catch all errors, and sometimes it might be helpful to inspect the status codes that KOMBIT returns to OS2sync (the error codes are logged to the logfile, as well as embedded in the error message when using the SDK API).

Note that the status codes are not passed through the API in any structured way, and it is not recommended that the software using the API should attempt processing of these codes – instead use the error handling outlined in the previous chapter.

The Organisation services returns a status code 20 on a successful operation, but if it returns a non-20 status code, and it is not a status code that STSOrgSync knows how to deal with, an error is thrown, which should be dealt with by the user of the API.

The list of possible status codes is shown below, together with a description on how STSOrgSync deals with these codes, and what possible actions that the user of the API could do to mitigate the error

Status Code	Description
40	This is probably the most common error returned by Organisation, and it means that the input is inconsistent. OS2sync, through input validation, attempts to ensure that this error never occurs, but if it is returned, then the input is incorrect according to the validation rules of Organisation. Look at the request (i.e. enable Request/Response logging) and see if something obvious is wrong with the request – usually this is the case (empty fields, incorrect timestamps, etc)
41	Authorization error – the service agreement is setup incorrectly.
45	Timestamp issue – unfortunately the Import() operation on Organisation requires a timestamp (registration-time), and if this timestamp is _after_ system-time on the Organisation server, this error is thrown. Ensure that the server running OS2sync has the correct time set, and is synchronized with a time-server so time does not drift.
47	Invalid validity period. This happens if two updates on the same object occur in the "wrong order" – the logic inside OS2sync always assumes that the current registration is the latest registration. The easy fix is to not supply a timestamp when using the API, then OS2sync will use current time for each registration, and

	this error will not happen. Otherwise make sure to call in timestamp order if there are multiple updates on the same object.
49	The object has been deleted or passivated, and cannot be updated. As OS2sync does not delete or passivate objects, this means that the object has been deleted or passivated through some other means. If the object needs to be updated, use the tool that deleted or passivated the object to restore it first.

There are other errors that can occur, but they are technical errors on Organisation, and besides waiting for them to be fixed (please report the error to KOMBIT), there is nothing much that can be done on the OS2sync side.

3 Prerequisites and configuration

Installation and configuration is covered in detail in the installation guide, but depending on the specific API being used, there are additional setup steps. These are outlined below

3.1 SQL and Service API

There are no additional setup steps required when using the SQL or Service API. All the configuration is done on the server where the service is deployed.

Simply start calling the REST endpoints to use the Service API or perform SQL Inserts when using the SQL API.

3.2 SDK API

The SDK API requires Visual Studio 2017 or later, and works with .NET Core 2.2 and later.

As the SDK API is used by importing the BusinessLayer (and IntegrationLayer) into a Visual Studio project, there are several configuration steps that must be performed before development can commence.

Note that the registry, certificate and service-agreement steps are also required when deploying the end-product outside the development machine.

1. A FOCES certificate (including private keys) must be installed on the development machine. This certificate is used when calling Organisation. The certificates used by KOMBIT must also be loaded onto the machine, so trust can be established during communication.
2. A set of service-agreements must be in place – these service-agreements must be based on the certificate from the previous step (otherwise access is denied when calling Organisation).
3. OS2sync has all its configuration stored in appsettings.json.

The three steps above are covered in some detail in the installation guide, as they are required for any communication with Organisation.

Inside Visual Studio, open (or create) the project where OS2sync is to be used, and perform the following steps

4. Right-click on "References" and pick "Add Reference…". Add the following two DLL files (part of the OS2sync distribution)
   - BusinessLayer.dll
   - IntegrationLayer.dll
5. Create (or copy from OS2sync) a log.config file for log4net

Before using the API, remember to call Initializer.Init(), and initialize the log4net API (otherwise the log will be empty).

4 API Usage

4.1 Service API

The Service API requires that the Service has been deployed on some server, which can be done either by installing the pre-compiled Windows Service, or by building a custom service program.

The pre-compiled version accepts request on port 5000, and the correct API usage is shown below.

4.1.1 Maintaining users

The /api/user endpoint accepts both POST, GET and DELETE requests. The POST request is used both for creating and updating users. The GET operation returns a structure identical to the one supplied when using POST.

Creating (or updating) a user object is done by POST'ing the a JSON payload against the /api/user endpoint. The example below shows the full data structure

```
POST /api/user HTTP/1.1
content-type: application/json

{
  "Uuid": "8e8f07d9-8261-446c-83f3-6b2edb121162",
  "ShortKey": null,
  "UserId": "bsg",
  "PhoneNumber": null,
  "Email": "bsg@digital-identity.dk",
  "Location": "Kontor 15",
  "Positions": [
    {
      "OrgUnitUuid": "3094b893-157c-4f20-91ef-bd2e95ee26fe",
      "Name": "Udvikler"
    }
  ],
  "Person": {
    "Name": "Brian Storm Graversen",
    "Cpr": null
  }
}
```

It is only required to supply the fields that are mandatory according to the API specification, so the following request is also valid, and does the same as the above request

```
{
  "Uuid": "8e8f07d9-8261-446c-83f3-6b2edb121162",
  "UserId": "bsg",
  "Email": "bsg@digital-identity.dk",
  "Location": "Kontor 15",
  "Positions": [
    {
      "OrgUnitUuid": "3094b893-157c-4f20-91ef-bd2e95ee26fe",
      "Name": "Udvikler"
    }
  ],
```

```
  "Person": {
    "Name": "Brian Storm Graversen"
  }
}
```

Deleting a user is done by performing a HTTP DELETE against the following endpoint (the UUID is the UUID of the user to delete).

http://<server>:5000/api/user/8e8f07d9-8261-446c-83f3-6b2edb121162

Reading a user is done by performing a HTTP GET against the following endpoint (the UUID is the UUID of the user to read).

http://<server>:5000/api/user/8e8f07d9-8261-446c-83f3-6b2edb121162

4.1.2 Maintaining organizational units

Maintenance of units is done in the same way as users, and the full JSON payload for creating or updating a unit, looks like this (and is the same structure that GET returns)

```
POST /api/v1_1/orgunit HTTP/1.1
content-type: application/json

{
  "Uuid": "3094b893-157c-4f20-91ef-bd2e95ee26fe",
  "ShortKey": "DEV",
  "Name": "Development",
  "ParentOrgUnitUuid": "e2f45c88-0d20-4b0b-80cd-f923fd175757",
  "PayoutUnitUuid": null,
  "PhoneNumber": "30 34 05 76",
  "Email": "kontakt@digital-identity.dk",
  "Location": null,
  "LOSShortName": null,
  "ContactOpenHours": null,
  "PhoneOpenHours": null,
  "PostReturn": null,
  "EmailRemarks": null,
  "Contact": null,
  "Ean": null,
  "Post": null,
  "Url": null,
  "Landline": null,
  "Type": "DEPARTMENT",
  "Tasks": [
     "13946fcc-2ac0-4c75-a35b-e3431efbed29",
     "98274f19-3827-4910-abbb-e294719bc290"
  ],
  "ContactForTasks": [
     "839183dd-2bb1-4811-a35b-ba431efbed55",
  ]
}
```

Deleting a unit is done by performing a HTTP DELETE against the following endpoint (the UUID is the UUID of the unit to delete).

http://<server>:5000/api/orgUnit/3094b893-157c-4f20-91ef-bd2e95ee26fe

Reading a unit is done by performing a HTTP GET against the following endpoint (the UUID is the UUID of the unit to read).

http://<server>:5000/api/orgUnit/3094b893-157c-4f20-91ef-bd2e95ee26fe

4.2 SDK API

The SDK API has a single initialization method, that must be called before the API is ready for use, and then it exposes three Service classes that can be used for maintaining Users, OrgUnits and reading data objects from STS Organisation.

4.2.1 Initialization

All relevant code is placed in the Organisation.BusinessLayer namespace, which should be added the using-section of the code as shown below. The method Init() on the Initializer class is the first thing to call before using the API

```
using Organisation.BusinessLayer;

namespace DemoProgram
{
    class Program
    {
        static void Main(string[] args)
        {
            Initializer.Init();
        }
    }
}
```

4.2.2 Maintaining users

The class UserService exposes three relevant methods (Update, Read and Delete), the usage of which is shown below. The Update() method is used both for creating and updating users, and is idempotent, so it can be called multiple times with the same input, without causing any effect (besides some calls to the Organisation service).

```
using Organisation.BusinessLayer;

namespace DemoProgram
{
    class Program
    {
        static void Main(string[] args)
        {
            Initializer.Init();

            // create a UserRegistration object for supplying user information
            UserRegistration user = new UserRegistration();
            user.Uuid = "5713fb19-d46a-411b-96ad-0abc3f67689b";
            user.UserId = "JJ";
            user.Person.Name = "Jens Jensen";
            user.Positions.Add(new Position({
                Name = "Sagsbehandler",
                OrgUnitUuid = "bd9d43b8-748d-4889-9057-9d47ff7aed55"
            });
```

```
            user.Email = "jj@kommune.dk";

            // calling the service is just supplying the registration object
            UserService userService = new UserService();
            userService.Update(user);

            // reading the just stored object is just supplying the UUID
            user = userService.Read("5713fb19-d46a-411b-96ad-0abc3f67689b");

            // delete the user by supplying the UUID of the User to be deleted
            userService.Delete("5713fb19-d46a-411b-96ad-0abc3f67689b",
                               DateTime.Now);
        }
    }
}
```

4.2.3 Maintaining organizational units

Just like the UserService, the class OrgUnitService exposes three methods, the usage of which is shown below. The same idempotent capabilities are true for the OrgUnitService.

```
using Organisation.BusinessLayer;

namespace DemoProgram
{
    class Program
    {
        static void Main(string[] args)
        {
            Initializer.Init();

            // create a OrgUnitRegistration object for supplying information
            OrgUnitRegistration unit = new OrgUnitRegistration();
            unit.Uuid = "17a64b76-c805-43b1-8794-3ebc5c47bbe9";
            unit.Name = "Borgerservice";
            unit.ParentOrgUnitUuid = "96bf6a1c-c44e-4cb7-a627-0ec34a9f0fb6";
            unit.Ean = "12312312312";
            unit.Type = OrgUnitType.DEPARTMENT;

            // calling the service is just supplying the registration object
            OrgUnitService orgUnitService = new OrgUnitService();
            orgUnitService.Update(unit);

            // reading the just stored object is just supplying the UUID
            unit = orgUnitService.Read("17a64b76-c805-43b1-8794-3ebc5c47bbe9");

            // delete the unit by supplying the UUID of the unit to be deleted
            orgUnitService.Delete("17a64b76-c805-43b1-8794-3ebc5c47bbe9",
                                  DateTime.Now);
        }
    }
}
```

4.3 SQL API

The SQL based API's are just ordinary SQL tables, the schemas for these tables are located in the mssql and mysql folders (use the one matching your SQL database of choice. By

INSERT'ing into the queue_ prefixed tables, the OS2sync SchedulingLayer will be triggered, and it will ensure that the data is synchronized with STS Organisation.

Make sure to do all inserts using a transaction, otherwise the scheduler might pick up a partial object and synchronize it before all child tables have been inserted into.

5 Success / Error tables

When using either the Service API or the SQL API, the build-in queuing tables are used, and likewise the build-in success/failure tables are populated as data is send through OS2sync.

Whenever a data-row is successfully copied to STS Organisation, the data is moved into the matching success_ prefixed table, and likewise when an error occurs, the data is moved in the the failure_ prefixed tables.

In case of failures, it is possible to modify the data directly in the table, and copy it back into the queue, to ensure the data is retried against STS Organisation.