Subscribe-HR Platform Documentation

Subscribe-HR

Jun 23, 2020

Developer Documentation

Subscribe-HR is a leading cloud HR software provider. We offer a range of applications to help businesses to effectively manage their work force. Subscribe-HR is more than just a system. All our software is built on a platform that offers a range of comprehensive development and configuration tools. This documentation is designed to help technical resources to use Subscribe-HR development platform.

If you are looking for information about our RESTful API it can be found here.

CHAPTER 1

Getting Started

The easiest way to start building new functionality with the Subscribe-HR platform is to sign up for a free trial. Do let us know if you are developing an app for our marketplace. We will ensure your account stays active while the work is being completed. An account with Subscribe-HR platform will provide access to a range of technical automation tools to extend existing or add new functionality.

CHAPTER 2

Apps

Our developers are always trying to come up with clever code names for dev tools and SHaRpi is no exception. SHaRpi is an integration app that allows linking of two or more APIs together using only a JSON template. It also includes detailed logging and integration with common Subscribe-HR features like *SSQL*.

The definition is a combination of independent blocks each with its own purpose. Blocks are then combined into a pipeline which outlines where the data is coming from, how it should be transformed and where it should be saved. This documentation will provide full API reference for each block and include examples of real configurations.

2.1 Getting Started

2.1.1 Introduction

Welcome to Subscribe-HR development platform. Use this guide to extend Subscribe-HR functionality by building apps, tasks, or develop any other type of extensions. Subscribe-HR platform provides a wide range of tools like *SSQL* query language, search capabilities, built in RESTful API and much more to utilise when building your apps.

2.1.2 To App or not to App

Before venturing on the journey of extending Subscribe-HR functionality you need to make a decision whether this new functionaity is going to be an app available via app store or just an extension for a particular system or client. It's important to make that decision early on as naming convention for fields and entities will be different. Global apps will have an app code added to the names to maintain uniqueness across the platform.

If you decide to buld a global app then it's important to ensure that **Belongs to App** field is always linked to your app. Failing to set that field will result in app failure after installation.

Items that can be packaged into an app:

- Objects or Entities
- Elements or Fields

- Scripts
- Workflow Tasks
- System Tasks
- Labels
- Messages
- Components
- Authentication Records
- Integration Processes

2.1.3 Components

You will find a lot of references to components throughout this documentation. What are components? They are small (or sometimes not so small) pieces of functionality developed for a specific purpose. There are four types of components that can be created in Subscribe-HR.

- **Tool** - application that sits outside of a module. Usually a custom tool to extend existing functionality or provide additional capabilities.
- **Widget** - a tile on the dashboard. Dashboard can have up to ten tiles dropped to a single bucket at a time. Each tile can be an application with its own permissions and functionality.
- **Wizard** - a stepped wizard to enhance usability of the system. WIzards are available from Start option on the dashboard. Wizards can also be triggered from other places. For example widgets.

Under the hood each component uses the same core functionality therefore documentation will reference them as such unless there is a specific function that is designed for a particular component type.

2.1.4 Component Architecture

This provides a brief breakdown of component structure. More detailed examples will be provided in later chapters. There are six main parts:

- **API** - Allows creation of RESTful API. See *Server Side API Reference* for available functions.
- **Template** - Template that gets loaded when the component is first initiated. Generated on the server side.
- **Front End Javascript** - Heading says it all. *Front End API Reference* for available functions and libraries.
- **CSS** - Stylesheet.
- **Permissions** - JSON structure describing permissions required for this component.

2.2 Server API Reference

2.2.1 Classes

PermissionManager

Provides access to user defined permissions for a component.

`hasPermission`(*code*)
Returns true if current user has access to specified permission code.

Parameters **code** – (string) permission code

Return type boolean

2.2.2 Modules

Environment (Shr.Env)

The environment module provides access to a set of internal platform functions.

`Shr.Env.`**`log`**(*variable*)
Displays information about a variable in a readable way.

Parameters **variable** – (mixed) variable to format

Return type string

`Shr.Env.`**`getBaseUrl`**()
Returns system base URL (e.g. https://app.subscribe-hr.com).

Return type string

`Shr.Env.`**`getAppUrl`**()
Returns application URL (e.g. https://app.subscribe-hr.com/cb/app).

Return type string

`Shr.Env.`**`getModuleUrl`**()
Returns current module URL (e.g. https://app.subscribe-hr.com/cb/app/hr).

Return type string

`Shr.Env.`**`getComponentApiUrl`**(*id*, *function*, *options*)
Creates request URL for component API function.

Parameters

- **id** – (string) component id
- **function** – (string) function name
- **options** – (object) URL parameters

Return type string

`Shr.Env.`**`getComponentPermissions`**(*id*)
Returns component permission manager.

Parameters **id** – (string) component id

Return type *PermissionManager*

Request (Shr.Request)

Request module provides access to information about the client request. This information can be accessed via the following methods.

`Shr.Request.`**`getParameter`**(*name*)
Extracts client parameter from POST or GET request.

Parameters **name** – (string) parameter name

Return type mixed

UI / Template (Shr.UI)

UI module provides functions to help generate user interface.

`Shr.UI.`**`createField`**(*options*)
Generates form field in the template.

Parameters **`options`** – (arguments) series of arguments depending on the type of field being generated

Return type string

Util (Shr.Util.Base64)

Module to encode and decode base64 strings.

`Shr.Util.Base64.`**`encode`**(*content*)
Encodes string in base64 format.

Parameters **`content`** – (string) string to encode

Return type string

`Shr.Util.Base64.`**`decode`**(*content*)
Decodes a base64 string.

Parameters **`content`** – (string) string to decode

Return type string

Util (Shr.Util.File)

Module to work with files Subscribe-HR virtual storage.

`Shr.Util.File.`**`create`**(*name*, *content*, *isTemp*)
Creates file in virtual storage.

Parameters

- **`name`** – (string) file name
- **`content`** – (string) file content
- **`isTemp`** – (boolean) is file temporary (temporary files are removed after 24 hours if they are not attached to records)

Return type string - file id

`Shr.Util.File.`**`update`**(*id*, *name*, *content*)
Update file.

Parameters

- **`name`** – (string) file id
- **`name`** – (string) file name
- **`content`** – (string) file content

Return type string - file id

2.3 Front End API Reference

2.3.1 Classes

PermissionManager

Provides access to user defined permissions for a component.

hasPermission(*code*)
Returns true if current user has access to specified permission code.

Parameters **code** – (string) permission code

Return type boolean

2.3.2 Modules

Environment (Shr.Env)

The environment module provides access to a set of internal platform functions.

Shr.Env.**log**(*variable*)
Displays information about a variable in a readable way.

Parameters **variable** – (mixed) variable to format

Return type string

Shr.Env.**getBaseUrl**()
Returns system base URL (e.g. https://app.subscribe-hr.com).

Return type string

Shr.Env.**getAppUrl**()
Returns application URL (e.g. https://app.subscribe-hr.com/cb/app).

Return type string

Shr.Env.**getModuleUrl**()
Returns current module URL (e.g. https://app.subscribe-hr.com/cb/app/hr).

Return type string

Shr.Env.**getComponentApiUrl**(*id*, *function*, *options*)
Creates request URL for component API function.

Parameters

- **id** – (string) component id
- **function** – (string) function name
- **options** – (object) URL parameters

Return type string

Shr.Env.**getComponentPermissions**(*id*)
Returns component permission manager.

Parameters **id** – (string) component id

Return type *PermissionManager*

Request (Shr.Request)

Request module provides access to information about the client request. This information can be accessed via the following methods.

`Shr.Request.`**`getParameter`**(*name*)
Extracts client parameter from POST or GET request.

Parameters **`name`** – (string) parameter name

Return type mixed

UI / Template (Shr.UI)

UI module provides functions to help generate user interface.

`Shr.UI.`**`createField`**(*options*)
Generates form field in the template.

Parameters **`options`** – (arguments) series of arguments depending on the type of field being generated

Return type string

Util (Shr.Util.Base64)

Module to encode and decode base64 strings.

`Shr.Util.Base64.`**`encode`**(*content*)
Encodes string in base64 format.

Parameters **`content`** – (string) string to encode

Return type string

`Shr.Util.Base64.`**`decode`**(*content*)
Decodes a base64 string.

Parameters **`content`** – (string) string to decode

Return type string

Util (Shr.Util.File)

Module to work with files Subscribe-HR virtual storage.

`Shr.Util.File.`**`create`**(*name*, *content*, *isTemp*)
Creates file in virtual storage.

Parameters

- **`name`** – (string) file name
- **`content`** – (string) file content
- **`isTemp`** – (boolean) is file temporary (temporary files are removed after 24 hours if they are not attached to records)

Return type string - file id

Shr.Util.File.**update**(*id*, *name*, *content*)
Update file.

Parameters

- **name** – (string) file id
- **name** – (string) file name
- **content** – (string) file content

Return type string - file id

CHAPTER 3

Integration Tool (SHaRpi)

Our developers are always trying to come up with clever code names for dev tools and SHaRpi is no exception. SHaRpi is an integration app that allows linking of two or more APIs together using only a JSON template. It also includes detailed logging and integration with common Subscribe-HR features like *SSQL*.

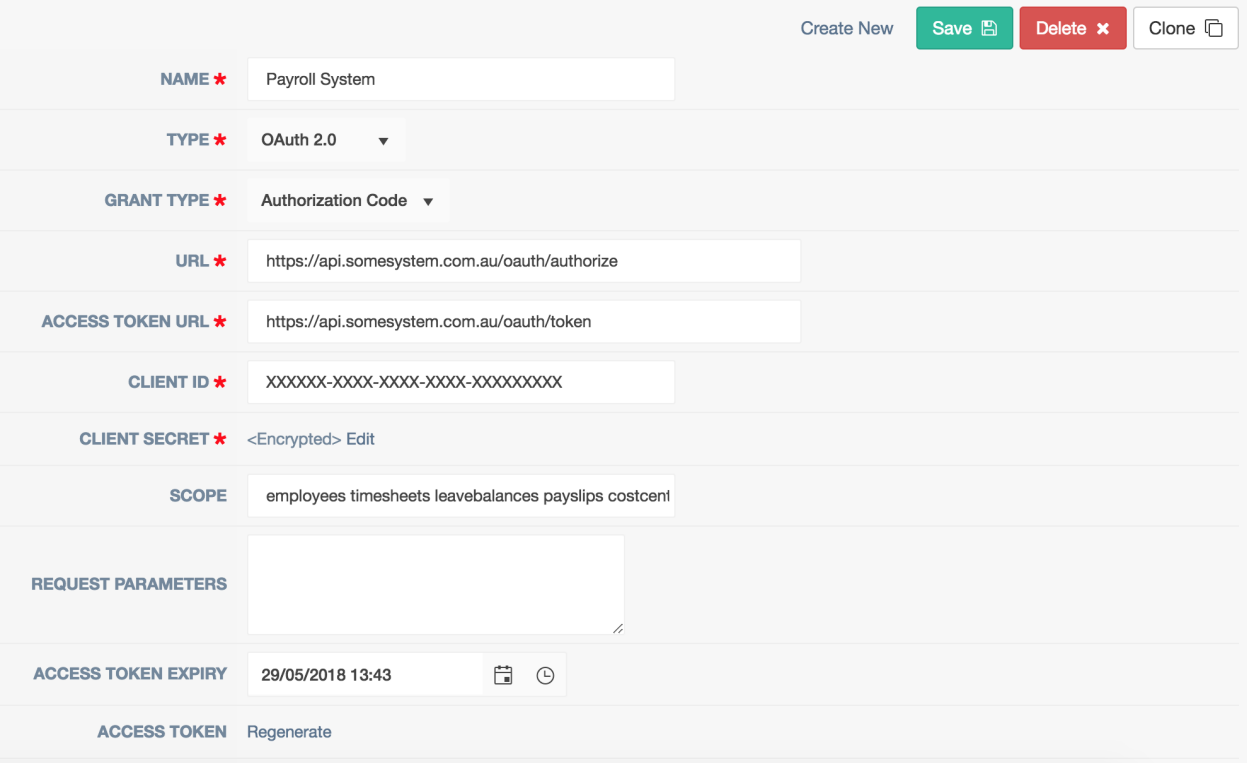
The definition is a combination of independent blocks each with its own purpose. Blocks are then combined into a pipeline which outlines where the data is coming from, how it should be transformed and where it should be saved. This documentation will provide full API reference for each block and include examples of real configurations.

3.1 Authentication

Most connections will require some sort of authentication to occur to enable access to remote resources. Authentication configuration is performed through Integration > Authentication screen and then linked to configuration file using record Id. A number of authentication types are supported. Additional types will be added as required.

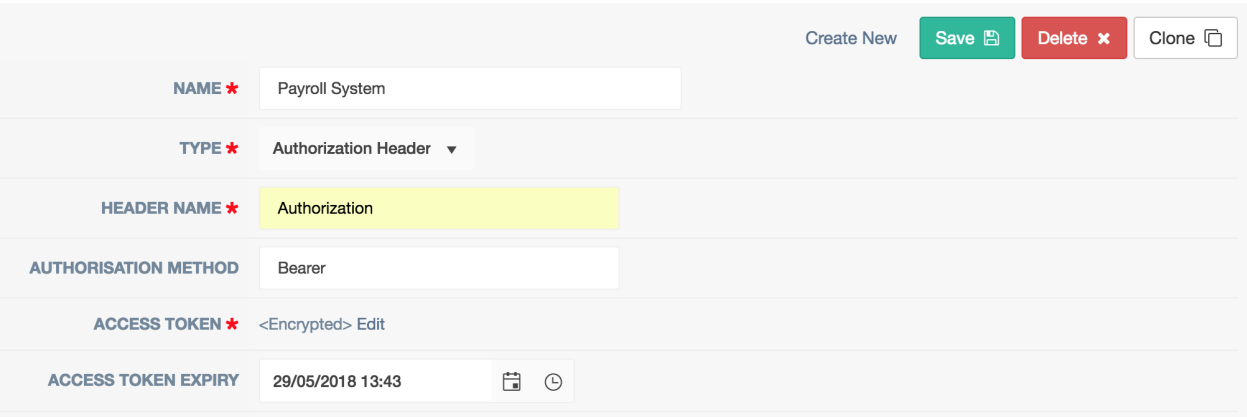
3.1.1 OAuth 2

This is probably the most common authentication type used with RESTful APIs. Multiple Grant Types are supported including `Authorization Code`. OAuth dance can be performed through this page by clicking on `Generate` link. SHaRpi will then keep track of access and refresh tokens (if available) and request new tokens as required.



3.1.2 Authorization Header

A simple header authentication method where access token is passed as a header with each request. This type allows users to specify header name, method (e.g. Bearer) and access code.



3.2 Architecture

SHaRpi is architected to be highly configurable to ensure that interfaces to third party products can be easily setup and adjusted based on client requirements. Usually customers have unique needs in relation to data flow directions, field mappings, scheduling and accessibility of end points. By using SHaRpi these modifications are easily implemented in a configuration file rather than by performing complex customisations.

The architecture of the tool consists of a number of predefined blocks (or as we call them actions) each with its own

purpose. Actions then get connected together to form a pipeline. Each action performs a unique task like extracting data from data source or transforming it into another format. Each action is logged into the Subscribe-HR logging platform including input and output making it easy to debug.

The data format that is used to pass information between blocks is JSON. Some operations may not always be capable of returning JSON in which case conversion is performed. Below you will find examples of input and output data from different action types.

3.2.1 Actions

The following table outlines all possible actions that can be defined within a pipeline. For detailed reference refer to *Pipeline Action*.

Action	Description
Operation	Performs CRUD operation on a given connection
Iterator	Iterates over a dataset
Map	Transforms data from one format into another
Function	Executes a javascript function
Pipeline	Triggers a pipeline

3.2.2 Operation

Operations can be of different types and require different parameters to be passed into them. Return data can also vary. For example with RESTful APIs it may be important to be able to access response headers as well as set request headers based on information that was received from another operation. For a Datum connection however headers are not required and it would not make sense to pass them in. Examples below show different formats for input and output for each action type.

RESTful Input

Parameters.Url

Will be merged into operation path which makes it possible to add parameters directly to URL. If path is set to `/api/v1/employee/:Id` the following input will change it to `/api/v1/employee/100`.

Parameters.Form

Will be added as variables to request.

Headers

Will be appended to request headers.

Data

Sent in request body and encoded as JSON or supplied Content Type.

```
{
    "Parameters": {
        "Url": {
            "Id": "100"
        }
        "Form": {
            "Name": "Alex"
        }
```

(continues on next page)

(continued from previous page)


```
    },
    "Headers": {
        "Content-Type": "application\/json"
    },
    "Data": [
        {
            "Id": 1,
            "Company": 1,
            "FirstName": "Alex"
        }
    ]
}
```


RESTful Output

Headers

Response headers.

StatusCode

Response status code.

Status

Response status text.

Data

Based on return format specified in operation.


```
{
    "Headers": {
        "Cache-Control": [
            "no-cache, must-revalidate"
        ],
        "Content-Type": [
            "application\/json"
        ],
        "Date": [
            "Sat, 26 May 2018 07:28:18 GMT"
        ],
        "Expires": [
            "0"
        ],
        "Server": [
            "Apache"
        ]
    },
    "StatusCode": 200,
    "Status": "OK",
    "Data": [
        {
            "Id": 1,
            "Company": 1,
            "FirstName": "Alex",
            "LastName": "Agafonov"
        }
```


(continues on next page)

(continued from previous page)

```
    ]
}
```

Datum Input

Parameters

Parameters to merge into query. For example if query is set to `SELECT e FROM Employee e WHERE e.Id = :Id` the following input will change it to `SELECT e FROM Employee e WHERE e.Id = 71`.

Data

Data to write into entity.

```
{
    "Parameters": {
        "Id": 300
    },
    "Data": [
        {
            "Surname": "Brounders",
            "FirstName": "Maria",
            "DateOfBirth": "1970-11-05",
            "EyeTest": "2013-01-08",
            "DateLeft": null,
            "EmploymentType": {
                "Value": "fulltime"
            }
        }
    ]
}
```

Datum Output

Data

Returns output of a query. Data for each entity is divided using entity name e.g. "Employee". If SSQL query is executed across multiple entities then multiple entity names are returned.

```
{
    "Data": [
        {
            "Employee": {
                "Id": 71,
                "CreatedBy": 4,
                "CreatedDate": "2009-06-22T13:26:21+10:00",
                "LastModifiedBy": 1,
                "LastModifiedDate": "2018-05-18T09:07:03+10:00",
                "Surname": "Brounders",
                "FirstName": "Maria",
                "DateOfBirth": "1970-11-05",
                "EyeTest": "2013-01-08",
                "DateLeft": null,
                "EmploymentType": {
```

(continues on next page)

(continued from previous page)

```
                    "Value": "fulltime",
                    "Text": "Full Time"
                }
            },
            "EmployeeAddress": [
                {
                    "Type": {
                        "Value": "residential",
                        "Text": "Residential"
                    },
                    "Address1": "..."
                },
                {
                    "Type": {
                        "Value": "postal",
                        "Text": "Postal"
                    },
                    "Address1": "..."
                }
            ]
        }
    ]
}
```

3.2.3 Iterator

Iterator action will loop through the data. It can be thought of as standard for loop in programming. Selector attribute will determine what data needs to be iterated over.

Conside the following example which is an output from Datum operation.

```
{
    "Data": [
        {
            "Employee": {
                "Id": 71
            }
        },
        {
            "Employee": {
                "Id": 72
            }
        },
        {
            "Employee": {
                "Id": 73
            }
        }
    ]
}
```

Assuming that iterator selector is set to `$.Data[*]` the first entry that will be returned is

```
{
    "Employee": {
```

(continues on next page)

(continued from previous page)

```
        "Id": 71
    }
}
```

3.2.4 Map

Map action will receive an input perform data transformation based on mappings specified and return transformed data structure.

Consider the following example which is a sample output from another operation.

```
{
    "Data": {
        "Employee": {
            "Id": 71
        }
    }
}
```

The following transformation will then be applied.

```
{
    "SampleMappings": [
        {
            "FromField": "$.Data.Employee.Id",
            "ToField": "$.Result.EmployeeCode",
        }
    ]
}
```

Which will result in the following output.

```
{
    "Result": {
        "EmployeeCode": 71
    }
}
```

3.2.5 Function

Functions add ability to perform complex mapping logic or make routing decisions based on output that is returned from previous operation. There are two different function types that can be used. Logical for routing and mapping for transforming data. Functions can be created inline or predefined and then called inside a pipeline. Function language is javascript. Engine that we use behind the scenes is v8 which supports most of ECMAScript 2015 (ES6). More information on each function type is available below.

Logical Function

Logical functions are designed to make complex routing decisions based on input data.

Consider the following example of predefined function

```
{
    "DecisionFunction1": {
        "Type": "Logical",
        "Code": function(input) {
                    if (input.Data.length == 0) {
                        return "Pipeline2";
                    }
                    return "Pipeline3";
                }
    }
}
```

If input is

```
{
    "Data": []
}
```

Then function will return "Pipeline2" otherwise "Pipeline3". Putting it into simple terms, if Data element is empty then execution will move on to Pipeline2 otherwise it will trigger Pipeline3. It is also possible to return array with multiple pipelines e.g. `["Pipeline4", "Pipeline5"]` which will then execute two pipelines.

Mapping Function

Mapping functions are designed to make complex transformations where it may not be sufficient to use JsonPath.

Consider the following example of inline function within mapping definition

```
{
    "SampleMappings": [
        {
            "FromField": "$.Data",
            "ToField": "$.Result.EmplType",
            "Code": function(input) {
                if (input.EmploymentType === "f") {
                    return "FullTime";
                }
                return "PartTime";
            }
        }
    ]
}
```

We will then pass in the following input

```
{
    "Data": {
        "EmploymentType": "f"
    }
}
```

And get the following output

```
{
    "Result": {
        "EmplType": "FullTime"
```

(continues on next page)

(continued from previous page)

```
    }
}
```

3.2.6 Pipeline

Pipeline actions are used to split execution into multiple streams either to make configuration files easier to read or to create reusable pieces of logic. Pipeline take input from previous operation and continue executing actions defined within them.

Note: Pipeline action must be the last action in the sequence. It is not possible to return output from pipeline and continue executing another action. This design ensures that there is no retrace within execution plan to minimise errors and keep pipelines linear.

Example of pipeline action definition

```
{
    "Type": "Pipeline",
    "Id": ["Pipeline43", "Pipeline2"],
}
```

3.3 JsonPath

JsonPath is an XPath-like expression language for filtering, flattening and extracting data. JsonPath uses special notation to represent nodes and their connections to adjacent nodes in a JsonPath. Tables below outline syntax and provide some examples on how to apply it in transformations. JsonPath is used in iterators and mappings to help find and transform information. Most of the time it is sufficient to use JsonPath without applying functions to manipulate data because of its reach functionality.

3.3.1 Expression Syntax

Symbol	Description
$	The root object/element (not strictly necessary)
@	The current object/element
. or []	Child operator
..	Recursive descent
*	Wildcard. All child elements regardless their index.
[,]	Array indices as a set
[start:end:step]	Array slice operator borrowed from ES4/Python.
?() | Filters a result set by a script expression	
() | Uses the result of a script expression as the index	

3.3.2 Example

```
{
    "store":
    {
        "book":
        [
            {
                "category": "reference",
                "author": "Nigel Rees",
                "title": "Sayings of the Century",
                "price": 8.95,
                "available": true
            },
            {
                "category": "fiction",
                "author": "Evelyn Waugh",
                "title": "Sword of Honour",
                "price": 12.99,
                "available": false
            },
            {
                "category": "fiction",
                "author": "Herman Melville",
                "title": "Moby Dick",
                "isbn": "0-553-21311-3",
                "price": 8.99,
                "available": true
            },
            {
                "category": "fiction",
                "author": "J. R. R. Tolkien",
                "title": "The Lord of the Rings",
                "isbn": "0-395-19395-8",
                "price": 22.99,
                "available": false
            }
        ],
        "bicycle":
        {
            "color": "red",
            "price": 19.95,
            "available": true
        }
    },
    "authors":
    [
        "Nigel Rees",
        "Evelyn Waugh",
        "Herman Melville",
        "J. R. R. Tolkien"
    ]
}
```

JsonPath	Result
$.store.bicycle.price	All books.
$.store.book[*]	The current object/element
$.store.book[1,3]	The second and fourth book.
$.store.book[1:3]	From the second book to the fourth.
$.store.book[:2]	From the first book to the third.
$.store.book[x:y:z]	Books from x to y with a step of z.
[start:end:step]	Array slice operator borrowed from ES4/Python.
$..book[?(@.category == 'fiction')]	All books with category == 'fiction'.
$..*[?(@.available == true)].price	All prices of available products.
$..book[?(@.price < 10)].title	The title of all books with price lower than 10.
$..book[?(@.author==$.authors[3])]	All books by "J. R. R. Tolkien"
$[store]	The store.
$..book[*][title, 'category', "author"]	title, category and author of all books.

3.4 Date Formatting

There is no specific enforced date format that must be used. Mappings provide two attributes DateFormatFrom and DateFormatTo. Syntax for those attributes provided in the table below. A wide range of symbols is supported.

3.4.1 Syntax

Format Character	Description	Example
Day		
d and j	Day of the month, 2 digits with or without leading zeros	01 to 31 or 1 to 31
D and l	A textual representation of a day	Mon through Sun or Sunday through Saturday
S	English ordinal suffix for the day of the month, 2 characters. It's ignored while processing.	st, nd, rd or th.
z	The day of the year (starting from 0)	0 through 365
Month		
F and M	A textual representation of a month, such as January or Sept	January through December or Jan through Dec
m and n	Numeric representation of a month, with or without leading zeros	01 through 12 or 1 through 12
Year		
Y	A full numeric representation of a year, 4 digits	1999 or 2003
y	A two digit representation of a year (which is assumed to be in the range 1970-2069, inclusive)	99 or 03 (which will be interpreted as 1999 and 2003, respectively)
Time		
a and A	Ante meridiem and Post meridiem	am or pm
g and h	12-hour format of an hour with or without leading zero	1 through 12 or 01 through 12
G and H	24-hour format of an hour with or without leading zeros	0 through 23 or 00 through 23
i	Minutes with leading zeros	00 to 59
s	Seconds, with leading zeros	00 through 59
u	Microseconds (up to six digits)	45, 654321
Timezone		
e, O, P and T	Timezone identifier, or difference to UTC in hours, or difference to UTC with colon between hours and minutes, or timezone abbreviation	UTC, GMT, Atlantic/Azores or+0200 or +02:00 or EST, MDT
Full Date/Time		
U	Seconds since the Unix Epoch (January 1 1970 00:00:00 GMT)	1292177455
Whitespace and Separators		
(space)	One space or one tab	
#	One of the following separation symbols: ; : / . , - ( )	/
; : / . , - ( )	The specified character.	-

3.5 Reference

3.5.1 Runtime Settings

Runtime settings define common configuration parameters on how it should be executed.

Parameters

Version

Type: String
Required: No
Default: 1.0
Description: Engine version number.

Name

Type: String
Required: No
Example: "My Integration Config"
Description: Configuration file name.

LogPayload

Type: Boolean
Required: No
Default: false
Example: true
Description: Determines whether payload (input and output) should be logged. It is recommended to do so to help with debugging. Some integrations may use up a lot of disk however.

EntryPipelineId

Type: String
Required: Yes
Example: Pipeline1
Description: Starting point of execution.

3.5.2 Connection

Connection represents a physical connection to a local or a remote resource. Following sections describe supported connection types in more details.

RESTful

Connection definition for HTTP RESTful API. A RESTful API is an application program interface (API) that uses HTTP requests to GET, PUT, POST and DELETE data.

Parameters

Type

Type: String
Required: Yes
Accepts: `Restful`, `Datum`
Description: Connection type.

Name

Type: String
Required: Yes
Example: My connection name
Description: Connection name. It will be referenced in logs.

Url

Type: String
Required: Yes
Validation: Must be valid URL
Example: https://api.my-company.com/v1
Description: Connection base URL

Authentication

Type: Integer
Required: Yes
Example: 5
Description: Id of Authentication entity. Must be created in Integration > Authentication form.

Datum

Datum is the name of the internal Subscribe-HR data layer. It is used to perform direct read and write operations from and to Subscribe-HR. Unlike API connection Datum is fast and does not require additional parameters or authentication to be defined.

Parameters

Type

Type: String
Required: Yes
Accepts: `Restful`, `Datum`
Description: Connection type.

Name

Type: String
Required: Yes
Example: My connection name
Description: Connection name. It will be referenced in logs.

3.5.3 Operation

Operation represents an action that can be performed through a given connection. For example a GET requests through RESTful API or query operation through a database connection.

RESTful

Operation definition for HTTP RESTful API. Must reference RESTful connection.

Parameters

Name

Type: String

Required: Yes
Example: My read operation
Description: Operation name. It will be referenced in logs.

Connection

Type: String
Required: Yes
Example: MyRestfulConnection
Description: Connection reference key that was used in the definition.

Method

Type: String
Required: Yes
Accepts: `GET`, `POST`, `PUT`, `PATCH`, `DELETE`
Example: GET
Description: HTTP request method.

Path

Type: String
Required: Yes
Example: /api/v1/employees/:Id
Description: HTTP end point path. Supports merge parameters e.g. :Id. Will be merged from Parameters.Url when passed in input.

ErrorHandlingStyle

Type: String
Required: No
Accepts: `warn`, `halt`
Default: halt
Example: warn
Description: Indicates how to handle errors when they are encountered. For example if operation received an error when trying to write data `warn` will produce a warning and continue execution of next record. Halt will completely terminate the process.

ErrorStatusCodes

Type: Array
Required: No
Default: ['4xx', '5xx']
Example: [400, '5xx']
Description: HTTP status codes that constitue errors. Supports wild card declaration e.g. `5xx` will include all 500 error codes.

ErrorStatusCodeExceptions

Type: Array
Required: No
Default: [404]
Example: [403, 404]
Description: HTTP status codes that should be ignored even if they are defined in ErrorStatusCodes field. Supports wild card declaration e.g. `5xx` will include all 500 error codes.

OutputType

Type: String
Required: No
Accepts: `json`, `raw`
Default: json
Example: json
Description: Output format to expect operation to return.

Datum

Operation definition for Datum connection type. Must reference Datum connection.

Parameters

Name

Type: String
Required: Yes
Example: My read operation
Description: Operation name. It will be referenced in logs.

Connection

Type: String
Required: Yes
Example: MyDatumConnection
Description: Connection reference key that was used in the definition.

Action

Type: String
Required: Yes
Accepts: `Create`, `Update`, `Get`, `Delete`, `Query`
Example: Get
Description: Type of operation to perform.

Entity

Type: String
Required: *Yes unless Action is Query*
Validation: Valid Subscribe-HR entity name
Example: Employee
Description: Subscribe-HR entity name. Can be found in Development > Objects > System Name.

Query

Type: String
Required: *Yes only if Action is Query*
Validation: Valid SSQL query
Example: SELECT e FROM Employee e
Description: Must provide full SSQL query if `Action is Query`. Otherwise only where clause condition e.g. `Id = :Id`.

ErrorHandlingStyle

Type: String
Required: No
Accepts: `warn`, `halt`
Default: halt
Example: warn
Description: Indicates how to handle errors when they are encountered. For example if operation received an error when trying to write data `warn` will produce a warning and continue execution of next record. Halt will completely terminate the process.

Pagination

Type: Object (*Pagination*)
Required: No
Example: `{ MaxItemsPerPage: 10 }`
Description: Pagination parameters allow splitting requests into batches. There may be cases where a lot of data needs to be processed at once. Due to resource allocation limits it is best practice to paginate your operations to return 100 or less records at a time. The limit is not enforced. This may change in the future. Pagination parameter can only be used with Action of type `Get` and `Query`.

OutputType

Type: String
Required: No
Accepts: `json`, `raw`
Default: json
Example: json
Description: Output format to expect operation to return.

3.5.4 Pagination

Pagination defines how operations paginate requests.

Datum

Pagination definition for Datum operations.

Parameters

MaxItemsPerPage

Type: Integer
Required: Yes
Example: 100
Description: Number of records to return per page.

3.5.5 Mapping

Mapping defines how source field is transformed into destination field. Mappings utilise JsonPath syntax to perform transformations making it very flexible to manipulate data. Mappings only deal with JSON format. Any operation that returns any other type of data will need to be converted into JSON first.

Parameters

FromField

Type: String
Required: *Yes if no Default specified*
Example: $.id
Description: JsonPath pattern to extract source field value.

ToField

Type: String
Required: Yes
Example: $.Data.Surname
Description: JsonPath pattern for destination field.

Tag

Type: String
Required: No
Example: MyEarlierTag
Description: Perform mapping from a data tag instead of input data.

Default

Type: String
Required: *Yes if no FromField specified*
Example: Some string
Description: Default value to assign to a field if from field is NULL or an empty string. Alternatively if FromField is not specified default value will be written into destination field.

Translations

Type: Object
Required: No
Example: `{"m": "male"}`
Description: Transformation object if field needs to be transformed to a different value. For example source value may be set to `m` which should be translated into `male` in destination field.

DateFormatFrom

Type: String
Required: *Yes if DateFormatTo is specified*
Example: d-m-Y
Description: For date fields indicates what format to expect dates in.

DateFormatTo

Type: String
Required: No
Example: Y-m-d
Description: For date fields indicates what format to output dates into.

Note: If only DateFormatFrom attribute is specified the default output format will be set to `Y-m-d H:i:s`.

FunctionId

Type: Object (*Function*)
Required: No
Example: MyMappingFunction
Description: Reference to predefined function to use to perform the mapping.

Code

Type: Function
Required: No
Example: function(input) { const output = input; return output; }
Description: Inline function definition to perform the mapping.

3.5.6 Function

Users can define javascript functions to use during pipeline execution. The syntax is `function(input) { const output = input; return output; }`. Javascript engine that we use behind the scenes is v8 which has support for ECMAScript 2015 (ES6).

Logical

Logical functions control the flow of pipeline execution process and determine what pipelines should be executed next based on the input. One of the most common use cases will be to determine if API has returned a status code of 404 and then execute record creation pipeline otherwise perform an update. Expected output of the function is a string with a pipeline name or an array with multiple names to be executed next.

Parameters

Type

Type: String
Required: Yes
Accepts: `Logical`, `Mapping`
Example: Logical
Description: Defines function type.

Code

Type: Function
Required: Yes
Validation: Valid javascript function
Example: function(input) { if (input.StatusCode == “404”) { return “Pipeline2”; } return “Pipeline3”; }
Description: Function code.

Mapping

Mapping functions are used to assist with performing mapping operations. Input comes from a selector or a previous action. Function can then manipulate the input and produce an output that is then passed back to the execution action.

Parameters

Type

Type: String
Required: Yes
Accepts: `Logical`, `Mapping`
Example: Mapping
Description: Defines function type.

Code

Type: Function
Required: Yes
Validation: Valid javascript function
Example: function(input) { if (input.EmploymentType == "f") { return "fulltime"; } }
Description: Function code.

3.5.7 Pipeline

Pipeline is a list of actions that need to be performed in a sequence. Pipelines can be nested and trigger other pipelines. There are a number of action types that can be performed within pipeline which are outlined below.

Common Action Parameters

Type

Type: String
Required: Yes
Accepts: `Operation`, `Iterator`, `Map`, `Function`, `Pipeline`
Example: Operation
Description: Defines action type to execute.

InputTag

Type: String
Required: No
Example: MyEmployeeRecord
Description: Tags input data to reuse later on in the pipeline.

OutputTag

Type: String
Required: No
Example: MyEmployeeOutputRecord
Description: Tags output data to reuse later on in the pipeline.

Operation Action

Triggers an operation.

Parameters

Inherits all *Common Action Parameters*

Id

Type: String
Required: Yes
Example: MyApiGetOperation
Description: Operation Id to execute.

Iterator Action

Iterates over a data set.

Parameters

Inherits all *Common Action Parameters*

Selector

Type: String
Required: Yes
Example: $.Data.content[*]
Description: JsonPath expression to indicate which data to iterate over.

Map Action

Iterates over a data set.

Parameters

Inherits all *Common Action Parameters*

Id

Type: String
Required: Yes
Example: MyEmployeeMappings
Description: Mapping Id to execute.

Function Action

Executes a javascript function. Can either be an inline function or reference to a predefined function.

Parameters

Inherits all *Common Action Parameters*

Id

Type: String
Required: *Yes if using a predefined function*
Example: MyPredefinedFunction1
Description: Predefined function Id.

FunctionType

Type: String
Required: *Yes if it is an inline function*
Accepts: `Logical`, `Mapping`
Example: MyPredefinedFunction1
Description: Predefined function Id.

Code

Type: String
Required: *Yes if it is an inline function*
Example: function(input) { const output = input; return output; }
Description: Inline function definition.

Pipeline Action

Executes a pipeline or a series of pipelines.

Note: Pipeline action must be the last action in the sequence. It is not possible to return output from pipeline and continue executing another action. This design ensures that there is no retrace within execution plan to minimise errors and keep pipelines linear.

Parameters

Inherits all *Common Action Parameters*

Id

Type: String or Array
Required: Yes
Example: [“Pipeline2”, “Pipeline3”]
Description: One or multiple pipelines to execute.

3.6 Tutorial

This tutorial will outline how to extract data from Subscribe-HR and then send it to a RESTful API end point.

3.6.1 Template Structure

The following array represents bare-bones configuration file that will be used to run our export. If you tried to run it now it would not pass validation because our main pipeline is empty. Following sections will detail creation of components in the file to create a fully working configuration.

```
var configuration = {
    RuntimeSettings: {
        Version: "1.0",
        Name: "SampleExport",
        LogPayload: true,
        EntryPipelineId: "Pipeline1"
    },
    Connections: {},
    Operations: {},
    Mappings: {},
    Functions: {},
    Pipelines: {
        Pipeline1: []
    }
};
```

3.6.2 Setting Up Connections

The first thing that we need to do is to add some connections. Operations cannot be performed without connections. One way to describe connections is they are client libraries designed to handle specific protocols. They establish line of communication to local or remote resources and allow operations to perform actions.

Because we are trying to export data from Subscribe-HR the first connection that needs to be created is Datum. Datum is the internal name of data layer that is used in the Subscribe-HR platform to handle all underlying data actions. Think of it as our proprietary ORM library.

Datum connection definition is simple because all information about it is already available through the system. It simply looks like this.

```
ShrConnection: {
    Type: "Datum",
    Name: "Datum Connection"
}
```

Next we need to create a connection to RESTful API. It requires couple more parameters than Datum connection. We provide base URL for all requests and authentication record Id so that our connection knows how to handle authentication.

```
ApiConnection: {
    Type: "Restful",
    Name: "RESTful API Connection",
    Url: "https://api.somesystem.com.au",
    Authentication: 1
}
```

3.6.3 Adding Operations

Having connections is great but not very useful without being able to perform operations. So what we will do next is to create some operations that we can use in execution pipelines.

Because we are trying to export some employee data, first operation will query it through Datum connection. Key parameters are `Action` and `Query`. We are executing SSQL query. Pagination parameter is also defined with max items set to 10 which means that if there are more than 10 records this operation will run in batches. It will not happen in this case because we are querying 2 employees only.

```
GetShrEmployees: {
    Name: "Get Shr Employees",
    Connection: "ShrConnection",
    Action: "Query",
    Query: "SELECT e FROM Employee e WHERE e.Id IN (71, 72)",
    ErrorHandlingStyle: "halt",
    Pagination: {
        MaxItemsPerPage: 10
    }
}
```

Now that first operation has been defined we can update our configuration template. It now looks like this. Also note that `GetShrEmployees` operation was added to Pipeline1 as first action.

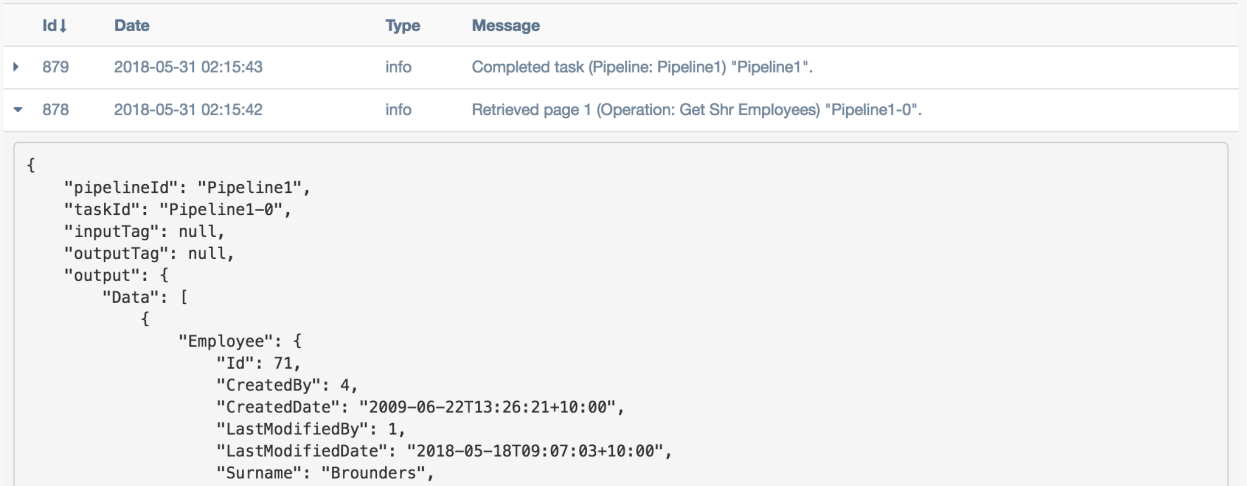
```
var configuration = {
    RuntimeSettings: {
        Version: "1.0",
        Name: "SampleExport",
        LogPayload: true,
        EntryPipelineId: "Pipeline1"
    },
    Connections: {
        ShrConnection: {
            Type: "Datum",
            Name: "Datum Connection"
        },
        ApiConnection: {
            Type: "Restful",
            Name: "RESTful API Connection",
            Url: "https://api.somesystem.com.au",
            Authentication: 1
        }
    },
    Operations: {
        GetShrEmployees: {
            Name: "Get Shr Employees",
            Connection: "ShrConnection",
            Action: "Query",
            Query: "SELECT e FROM Employee e WHERE e.Id IN (71, 72)",
            ErrorHandlingStyle: "halt",
            Pagination: {
                MaxItemsPerPage: 10
            }
        }
    },
    Mappings: {},
    Pipelines: {
        Pipeline1: [
            {
                Type: "Operation",
                Id: "GetShrEmployees"
            }
        ]
    }
};
```

Now let's run the above configuration to see what it does. To do that you will need to first create `Process` record in the system by going to `Integration > Processes` and clicking `Create` button. Enter process name and paste

configuration into code editor. Press `Save` button. No errors should be generated at this stage as our configuration meets minimal requirements. At this point `Run Process` button will appear. Once the button is pressed you will see a loading icon. At this stage a message has been sent to the worker in the background to let it know that the process needs to run immediately. It may take few minutes for it to complete depending on the volume of data being processed. To see what's going on with the process, go to Events tab. It will show all the actions that have been executed.

Note: While testing your configuration ensure that volume of data that is being sent or received is limit to few records only. It will make it easier to debug and save a lot of waiting time.

The following image shows entries in my Events tab after executing above configuration. Returned data can be seen in detailed view `output` field.



3.6.4 Adding Iterator

What are iterators? They help us to run through multiple records. Above example returns two employee records. If we were working with operation that supports importing multiple employees then at this stage we can just perform data transformation and call the operation. It however is not the case with a lot of APIs. From our experience the standard data flow is Get Data => Loop => Transform => Check If New / Existing => Create / Update. So let's create an iterator for our two records.

```
{
    Type: "Iterator",
    Selector: "$.Data",
    OutputTag: "ShrEmployee"
}
```

Above example will iterate over `$.Data[*]`. You will also notice that the record gets tagged at this point. This is to ensure that if we need to access original data later on in the transformation process that it can easily be done without performing additional actions.

3.6.5 Adding Another Operation

OK so now we have two employee records that we loop over. As mentioned above at this point we probably want to check if this employee already exists in the destination system before trying to create it. There is a number of ways to do this. One, we can create a flag that tracks whether employee has already been exported or not. Two, we can try to always create it and just let it fail. If it fails we then trigger an update pipeline. Three, and this is the method I

personally prefer as it is pretty fail safe, we check whether record already exists in destination system and then trigger appropriate pipeline. So let's add an operation that checks if record exists or not.

```
LookupApiEmployee: {
    Name: "API Lookup Single Employee",
    Connection: "ApiConnection",
    Method: "GET",
    Path: "/api/v1/employees/:EmployeeId"
}
```

It can be seen in the definition above that this operation will require a URL parameter `:EmployeeId` to be passed in. This can be done using mappings or a function. I prefer mappings as relying on functions can make it harder to maintain configuration files.

3.6.6 Adding Mappings

Above operation requires `EmployeeId` parameter to be passed in for it to work correctly. We already know from *RESTful Input* that URL parameters can be passed using `Parameters.Url` attribute so we define some mappings to create this structure.

```
LookupApiEmployeeMappings: [
    {
        FromField: "$.Employee.EmployeeCode",
        ToField: "$.Parameters.Url.EmployeeId"
    }
]
```

Now let's update our configuration file and add the latest changes.

```
var configuration = {
    RuntimeSettings: {
        Version: "1.0",
        Name: "SampleExport",
        LogPayload: true,
        EntryPipelineId: "Pipeline1"
    },
    Connections: {
        ShrConnection: {
            Type: "Datum",
            Name: "Datum Connection"
        },
        ApiConnection: {
            Type: "Restful",
            Name: "RESTful API Connection",
            Url: "https://api.somesystem.com.au",
            Authentication: 1
        }
    },
    Operations: {
        GetShrEmployees: {
            Name: "Get Shr Employees",
            Connection: "ShrConnection",
            Action: "Query",
            Query: "SELECT e FROM Employee e WHERE e.Id IN (71, 72)",
            ErrorHandlingStyle: "halt",
            Pagination: {
```

(continues on next page)

(continued from previous page)

```
                MaxItemsPerPage: 10
            }
        },
        LookupApiEmployee: {
            Name: "API Lookup Single Employee",
            Connection: "ApiConnection",
            Method: "GET",
            Path: "/api/v1/employees/:EmployeeId"
        }
    },
    Mappings: {
        LookupApiEmployeeMappings: [
            {
                FromField: "$.Employee.EmployeeCode",
                ToField: "$.Parameters.Url.EmployeeId"
            }
        ]
    },
    Pipelines: {
        Pipeline1: [
            {
                Type: "Operation",
                Id: "GetShrEmployees"
            },
            {
                Type: "Iterator",
                Selector: "$.Data",
                OutputTag: "ShrEmployee"
            },
            {
                Type: "Map",
                Id: "LookupApiEmployeeMappings"
            },
            {
                Type: "Operation",
                Id: "LookupApiEmployee"
            }
        ]
    }
};
```

Quick summary of the changes in the pipeline:

- Call `GetShrEmployees` operation which will return two employee records
- Iterate over results
- Map record to produce URL parameter
- Call `LookupApiEmployee` to see if record already exists in the destination system

So far so good. Now how do we actually test result of the last operation. This is where logical functions can be very useful.

3.6.7 Adding Function

Because last operation is of type RESTful API the response will contain headers, status codes and response body. Refer to *RESTful Output* for more details. If the API end point is implemented correctly then we should receive status

code 404 if record does not exist. Let's define action of type function with inline function to test for it.

```
{
    Type: "Function",
    FunctionType: "Logical",
    Code: function(input) {
        if (input.StatusCode == "404") {
            return "Pipeline2";
        }
        return "Pipeline3";
    }
}
```

Deciphering the above. If response code is 404 then trigger `Pipeline2` (creation of new record) otherwise go to `Pipeline3` (update existing record).

3.6.8 Adding New Pipeline

Now we can add new pipeline to handle record creation. It only requires two actions, Map and Operation. Let's create another operation and mappings to use in the new pipeline.

Note: Each subsequent action will inherit output of previous action. Tags can be used to work around this issue.

Adding create employee operation.

```
{
    CreateApiEmployee: {
        Name: "API Lookup Single Employee",
        Connection: "ApiConnection",
        Method: "POST",
        Path: "/api/v1/employees"
    }
}
```

Adding mappings.

```
MapShrEmployeeToApi: [
    {
        FromField: "$.Employee.Id",
        ToField: "$.Data[0].id"
    },
    {
        FromField: "$.Employee.Surname",
        ToField: "$.Data[0].surname"
    },
    {
        FromField: "$.Employee.FirstName",
        ToField: "$.Data[0].firstNames"
    },
    {
        FromField: "$.Employee.StartDate",
        ToField: "$.Data[0].startDate",
        DateFormatFrom: "Y-m-d",
        DateFormatTo: "d-M-Y"
    },
```

(continues on next page)

(continued from previous page)

```
    {
        FromField: "$.Employee.Gender.Value",
        ToField: "$.Data[0].gender",
        Translations: {
            male: "Male",
            female: "Female"
        }
    }
]
```

3.6.9 Putting It All Together

The following configuration can now be used as a template for all integration processes. It should be expanded to add `Pipeline3` which should looks very similar to `Pipeline2` with small difference in mappings and operator call.

```
var configuration = {
    RuntimeSettings: {
        Version: "1.0",
        Name: "SampleExport",
        LogPayload: true,
        EntryPipelineId: "Pipeline1"
    },
    Connections: {
        ShrConnection: {
            Type: "Datum",
            Name: "Datum Connection"
        },
        ApiConnection: {
            Type: "Restful",
            Name: "RESTful API Connection",
            Url: "https://api.somesystem.com.au",
            Authentication: 1
        }
    },
    Operations: {
        GetShrEmployees: {
            Name: "Get Shr Employees",
            Connection: "ShrConnection",
            Action: "Query",
            Query: "SELECT e FROM Employee e WHERE e.Id IN (71, 72)",
            ErrorHandlingStyle: "halt",
            Pagination: {
                MaxItemsPerPage: 10
            }
        },
        LookupApiEmployee: {
            Name: "API Lookup Single Employee",
            Connection: "ApiConnection",
            Method: "GET",
            Path: "/api/v1/employees/:EmployeeId"
        },
        CreateApiEmployee: {
            Name: "API Lookup Single Employee",
            Connection: "ApiConnection",
            Method: "POST",
```

(continues on next page)

(continued from previous page)

```
            Path: "/api/v1/employees"
        }
    },
    Mappings: {
        LookupApiEmployeeMappings: [
            {
                FromField: "$.Employee.EmployeeCode",
                ToField: "$.Parameters.Url.EmployeeId"
            }
        ],
        MapShrEmployeeToApi: [
            {
                FromField: "$.Employee.Id",
                ToField: "$.Data[0].id"
            },
            {
                FromField: "$.Employee.Surname",
                ToField: "$.Data[0].surname"
            },
            {
                FromField: "$.Employee.FirstName",
                ToField: "$.Data[0].firstNames"
            },
            {
                FromField: "$.Employee.StartDate",
                ToField: "$.Data[0].startDate",
                DateFormatFrom: "Y-m-d",
                DateFormatTo: "d-M-Y"
            },
            {
                FromField: "$.Employee.Gender.Value",
                ToField: "$.Data[0].gender",
                Translations: {
                    male: "Male",
                    female: "Female"
                }
            }
        ]
    },
    Pipelines: {
        Pipeline1: [
            {
                Type: "Operation",
                Id: "GetShrEmployees"
            },
            {
                Type: "Iterator",
                Selector: "$.Data",
                OutputTag: "ShrEmployee"
            },
            {
                Type: "Map",
                Id: "LookupApiEmployeeMappings"
            },
            {
                Type: "Operation",
                Id: "LookupApiEmployee"
```

(continues on next page)

(continued from previous page)

```
            },
            {
                Type: "Function",
                FunctionType: "Logical",
                Code: function(input) {
                    if (input.StatusCode == "404") {
                        return "Pipeline2";
                    }
                    return "Pipeline3";
                }
            }
        ],
        Pipeline2: [
            {
                Type: "Map",
                InputTag: "ShrEmployee",
                Id: "MapShrEmployeeToApi"
            },
            {
                Type: "Operation",
                Id: "CreateApiEmployee"
            }
        ]
    }
};
```

CHAPTER 4

SSQL

4.1 Overview

Subscribe-HR Structured Query Language is a subset of SQL language which is used to query data within Subscribe-HR object relational model. Queries can be performed using either server side data access module or RESTful API query end point.

4.2 SSQL Terminology

The following terminology will be used when referencing SSQL statements. SSQL left, SQL right.

- Entity / Object - table
- Field - column

4.3 Supported Expressions

4.3.1 Statement Types

- SELECT
- INSERT
- UPDATE
- DELETE

4.3.2 Joins

- JOIN / INNER JOIN

- LEFT JOIN
- RIGHT JOIN
- CROSS JOIN

4.3.3 Operators

Operator	Description
=	Comparison equal operator
<=>	NULL-safe equal. This operator performs an equality comparison like the = operator, but returns 1 rather than NULL if both operands are NULL, and 0 rather than NULL if one operand is NULL
>	Greater than
>=	Greater than or equals to
<	Less than
<=	Less than or equals to
<>	Not equal
LIKE	Uses wildcard operators to compare a value to similar values.
NOT	Reverses the meaning of the logical operator e.g. NOT IN
AND	It allows the existence of multiple conditions
OR	It is used to combine multiple conditions
IN	It is used to compare a value in a list of literal values
IS	Tests a value against a boolean value e.g. IS NULL
+	Addition
-	Subtraction
/	Division
*	Multiplication

4.3.4 Functions

Function	Description	Example
ABS	Returns the absolute value of a number	ABS(-5) Result: 5
CEIL	Returns the smallest integer value not less than the number specified as an argument	CEIL(1.2) Result: 2
FLOOR	Returns the largest integer value not greater than the number specified as an argument	FLOOR(1.2) Result: 1
SQRT	Square root	SQRT(25) Result: 5
MOD	Returns the remainder of dividend divided by divisor	MOD(17,5) Result: 2
LENGTH	Returns the length of a string	LENGTH('hi') Result: 2
SUBSTRING	Extract a substring from a string	SUBSTRING('Subscribe', 2, 5) Result: ubscr
LOWER	Convert the text to lower-case	LOWER('HI') Result: hi
UPPER	Convert the text to upper-case	UPPER('hi') Result: HI
CONCAT	Adds several strings together	CONCAT('h', 'i') Result: hi
COALESCE	Returns the first non-null value in a list	COALESCE(NULL, 1) Result: 1

4.3.5 Aggregate Function

Function	Description
AVG	Average
COUNT	Count a number of records in a group
MAX	Maximum value in a set
MIN	Minimum value in a set
STD	Standard deviation
SUM	Sum of values in a group
VARIANCE	Population standard variance
GROUP_CONCAT	Returns a string with concatenated non-NULL value from a group

4.4 Object Relationships

Subscribe-HR objects and fields can be created dynamically using development tool. These objects and fields can then be queried using SSQL. Objects can also have parent child relationships. One parent object can have many child objects related to it. At the same time, a child can only have one parent. For performance reasons no nested relationships are allowed.

Note: To find object names go to Development > Objects. You will see Object System Name column.

4.5 Common Fields

The following fields are common for every object.

Field Name	Description
Id	Unique record Id
CreatedBy	Creator user Id
CreatedDate	Date when record was first created
LastModifiedBy	User id that last modified the record
LastModifiedDate	Date when record was last modified
__ParentId	Foreign key (only child objects)

4.6 Learning By Example

4.6.1 Simple Statement

```
SELECT e FROM Employee e WHERE e.FirstName = 'Maria';
```

Return all employees with the first name Maria.

Note: Select * (star) expression is not supported. You must specify list of aliases or field names.

Note: All entities in from clause must have an alias.

4.6.2 Join Statement

```
SELECT e, ea FROM Employee e LEFT JOIN EmployeeAddress ea ON (e.Id = ea.__ParentId)␣
↪WHERE e.FirstName = 'Maria';
```

Return all employees with the first name Maria and their addresses.

Index

H

S