Integration Service for Developers

Version 1.8.1

Table of Contents

Chapter 1

Introduction to the Integration Service Tools

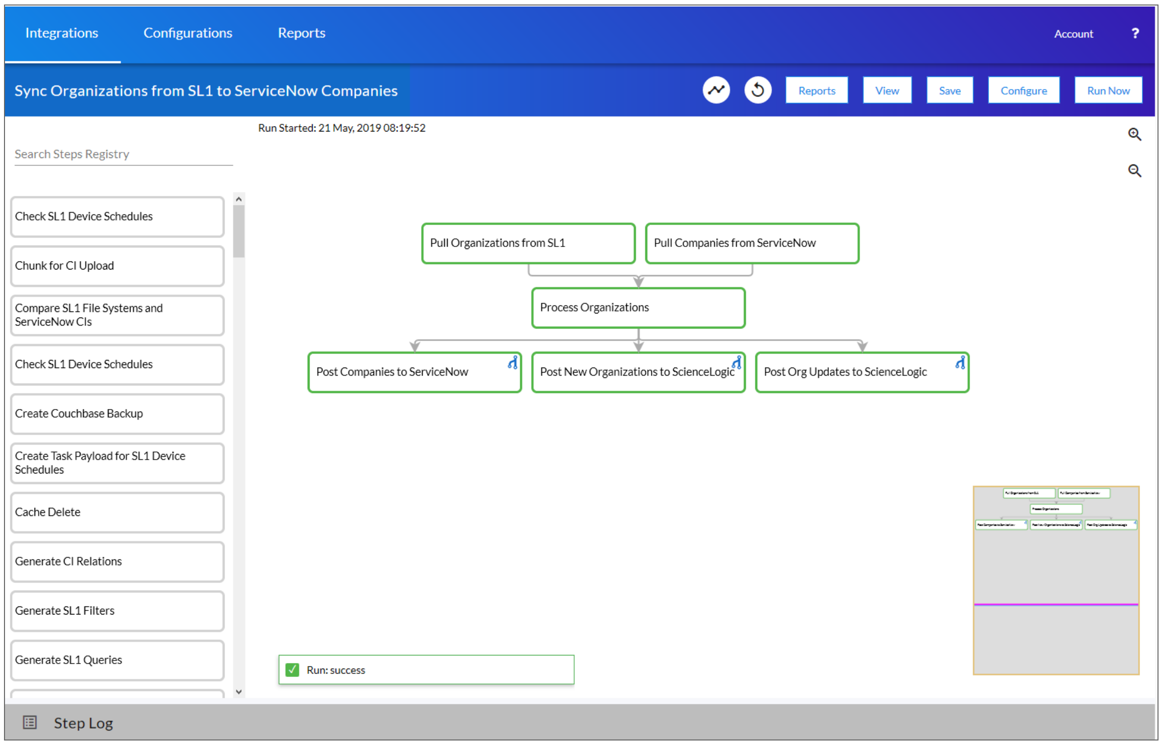
Overview

This manual describes how you can use the tools included in the Integration Service and your own tools to create your own integration applications, steps, and configurations that you can use with the Integration Service.

This chapter covers the following topics:

What is the Integration Service?

The Integration Service enables intelligent, bi-directional communication between the ScienceLogic data platform and external data platforms to promote a unified management ecosystem. The Integration Service allows users to translate and share data between SL1 and other platforms without the need for programming knowledge. The Integration Service is designed to provide high availability and scalability.


Integrations
Configurations
Reports
Account
?
Sync Organizations from SL1 to ServiceNow Companies
Reports
View
Save
Configure
Run Now
Run Started: 21 May, 2019 08:19:52
Search Steps Registry
Check SL1 Device Schedules
Chunk for CI Upload
Compare SL1 File Systems and ServiceNow CIs
Check SL1 Device Schedules
Create Couchbase Backup
Create Task Payload for SL1 Device Schedules
Cache Delete
Generate CI Relations
Generate SL1 Filters
Generate SL1 Queries
Pull Organizations from SL1
Pull Companies from ServiceNow
Process Organizations
Post Companies to ServiceNow
Post New Organizations to ScienceLogic
Post Org Updates to ScienceLogic
Run: success
Step Log

The key elements of the Integration Service user interface include the following:

- ***Steps***. A step is a generic Python class that performs an action. Steps accept arguments and can be re-used. The arguments tell the step which variables and values to use when executing. Steps can also pass results to a subsequent step.
- ***Integration Applications***. An integration application is a JSON object that includes all the information required for executing an integration on the Integration Service platform. An integration application includes a list of steps and metadata for those steps. Each step will execute a single action and pass the results to subsequent, dependent steps. The parameters for each step are also defined in the application and can be provided either directly in the step or in the parent integration application.
- ***Configurations***. A configuration object is a stand-alone JSON file that contains a set of configuration variables. Configurations live on the Integration Service system and can be accessed by *all* integration applications and their steps. Click the **[Configure]** button from an integration application in the Integration Service user interface to access the configuration object for that integration application.

The topics in this section describe how to create a step that is compatible with the Integration Service system. These topics also describe parameters, built-in parameters, and transferring data.

After you have created one or more steps, you can use them in one or more ***integration applications***.

Tools Included with Integration Service

The Integration Service includes the following tools for creating custom integrations:

- ***iscli***. The Integration Service includes a command line tool called ***iscli*** (Integration Service Command Line Interface). When you install the Integration Service, ***iscli*** is automatically installed. The iscli allows you to upload integration applications, steps, and configurations. For more information on the iscli, see the section on *iscli*.
- ***IS API***. The Integration Service includes an API. When you install the Integration Service, the API is available. For more information on the IS API, see the section on ***the IS API***.
- ***ipaascore.BaseStep class***. The Integration Service includes a Python class called ***ipaascore.BaseStep class***. This class includes multiple pre-defined functions that you can use when you are writing or editing a step. For more information, see the section on ***ipaascore.BaseStep class***.

Prerequisites for Creating Integration Applications

To create your own integration application, you must:

- Deploy an Integration Service system and ensure it is accessible. For details, see the ***Integration Service Platform*** manual.
- Have SSH or console access to the Integration Service system, so you can use the iscli (Integration Service Command Line Interface). You will use the iscli to upload steps, integration applications, and configurations to the Integration Service.
- Be comfortable with Python.
- Install a local copy of a Python IDE to use for development.
- Install a local copy of an API tool, like cURL, Insomnia, or Postman.
- Install a local copy of a source-code editor like Notepad++, vi, or TextEdit.

Chapter 2

Creating a Step

Overview

This chapter explains how to create your own custom step that you can then upload and run on an Integration Service system.

This chapter covers the following topics:

What is a Step?

In an Integration Service system, a ***step*** is a generic Python class that performs a single action, such as caching device data:

Steps accept arguments called *parameters*. These arguments specify the values, variables, and configurations to use when executing the step. Parameters allow steps to accept arguments and allow steps to be re-used in multiple integrations. For example, you can use the same QueryREST step to query both the local system and another remote system; only the arguments, (hostname, username, and password) change.

A step can pass the data it generates during execution to a subsequent step. A step can use the data generated by another step.

The Integration Service system analyzes the required parameters for each step and alerts you if any required parameters are missing before the Integration Service runs the step.

Steps are grouped into the following types:

- ***Standard***. Standard steps do not require any previously collected data to perform. Standard steps are generally used to generate data to perform a transformation or a database insert. These steps can be run independently and concurrently.
- ***Aggregated***. Aggregated steps require data that was generated by a previously run step. Aggregated steps are not executed by the Integration Service until all data required for the aggregation is available. These steps can be run independently and concurrently.
- ***Trigger***. Trigger steps are used to trigger other integration applications. These steps can be configured to be blocking or not.

Default Steps

The Integration Service system includes some already-defined steps:

- MicrosoftSqlDescribe
- MicrosoftSqlInsert
- MicrosoftSqlSelect

- MySqlDescribe
- MySqlInsert
- MySqlSelect
- QueryGQL
- QueryREST
- stepTemplate

To view the code for one of these steps:

1. Using an API tool like Postman or cURL, use the API `GET /steps` to download the steps:

   ```
   URL_for_your_Integration_Service_system/api/v1/steps/step_name
   ```

   where:

   - *URL_for_your_Integration_Service_system* is the IP address or URL for the Integration Service.
   - *step_name* is the name of the step you want to view.

Requirements

To create a custom step, you must perform the following tasks:

1. Download or copy the step template, called *stepTemplate*.
2. Set up the required classes and methods in the step.
3. Define logic for the step, including transferring data between steps.
4. Define parameters for the step.
5. Define logging for the step.
6. Define exceptions for the step.
7. Upload the step to the Integration Service system.
8. Validate and test the step.

Creating a Step from the Step Template

The easiest way to create a new step is to use the step template that is included with the Integration Service system. To copy this template to your desktop:

1. Using an API tool like Postman or cURL, use the API ***GET /steps/{step_name}***:

   ```
   GET URL_for_your_Integration_Service _system/api/v1/steps/stepTemplate
   ```

   where:

- *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.

2. Select and copy all the text in the stepTemplate.
3. Open a source-code editor and paste the content of the stepTemplate in the source-code editor.
4. Save the new file as *newfilename*.py

   where:

   - *newfilename**.py*** is the new name of the step and includes the .py suffix. The file name must be unique within your Integration Server system and cannot contains spaces. Note that the step name will also be the name of the Python class for the step.

Including the SubClass and Required Methods

To execute successfully on the Integration Service system, your step must be a subclass of the ipaascore.BaseStep class and include the init method and the execute method.

Subclass

The stepTemplate.py file is already configured to include the new step as a subclass of the ipaascore.BaseSetp class. To update your step:

1. Use a source-code editor to open the re-named file for editing.
2. Notice that the file includes these lines of text:

```
from ipaascore.BaseStep import BaseStep
from ipaascommon import ipaas_exceptions
```

3. Do not remove or alter these lines of text.
4. Search for the following:

```
class stepTemplate(BaseStep):
```

5. Replace ***stepTemplate*** with the new name of the file (without the .py suffix).
6. Save and close the file.

Required Methods

To execute successfully on the Integration Service system, your step must contain at least two methods:

- ***init method***. Allows you to define initialization options and parameters for the step.
- ***execute method***. The execute method includes the logic for the step and performs the action. After the Integration Service system evaluates all parameters and initialization settings and aligns the step with a worker process, the Integration Service system examines the execute method.

Without these methods, the Integration Service system will consider your step to be "incomplete" and will not execute the step.

The stepTemplate.py file includes these two methods and the syntax of some of the sub-methods you can use within the main methods.

Defining the Logic for the Step

Each step requires the init method and the execute method. Within those methods, you can specify parameters and logic for the step. The following sections describe how to do this.

The init Method

From the init method, you can define the friendly name, the step description, and the step version:

```
self.friendly_name = “friendly name of the step. This name appears in the user
interface”
self.description = “Description of the step”
self.version = “version number”
```

In the QueryREST step, the friendly name, description, and version number are defined like this:

```
def __init__(self):
self.friendly_name = “Query REST”
self.description = “Step facilitates REST interactions and will return the returned
data dictionary and specified headers as data to the next step”
self.version = “1.0.0”
```

From the init method, you also define the parameters for the step. The Integration Service system will examine the parameters and enforce the parameters when the step is run.

For example, if you specify a parameter as required, and the user does not specify the required parameter when calling the step, the Integration Service system will display an error message and will not execute the step.

To define a parameter, use the self.***new_step_parameter*** function.

```
self.new_step_parameter(name=parameter_name, description="description", sample_
value="sample value", default_value=None, required=False)
```

where:

- *name*. The name of the parameter. This value will be used to create a name:value tuple in the integration application file (in JSON).
- *description*. A description of the step parameter.
- *sample_value*. A sample value of the required data type or schema.
- *default_value*. If no value is specified for this parameter, use the default value. Can be any Python data structure. To prevent a default value, specify "None".
- *required*. Specifies whether this parameter is required by the step. The possible values are "True" or "False".

Here is an example from the QueryREST step (available on each Integration Service system):

```
self.new_step_parameter(name=PREFIX_URL, description="used with relative_url to
create the full URL.", sample_value="http://10.2.11.253", default_value=None,
required=True)
```

The execute Method

From the execute method you can:

- Use Python logic and functions
- retrieve the value of a parameter with self.*get_parameter*
- retrieve the value of a variable with self.*get_app_variables* (variables are defined in the integration application)
- *Retrieve data from a previous step* in the integration application
- *Save data for use by the next step* in the integration application
- Define logging for the step
- Define exceptions for the step

For details on all the functions you can use in the execute method, see the chapter on the *ipaascore.BaseStep class*.

You can also define additional methods in the step. For examples of this, see the QueryREST step provided with the Integration Service system.

For examples of the logic in a step, view one or more of the steps listed in *Default Steps*.

Transferring Data Between Steps

An essential part of integrations is passing data between tasks. The Integration Service system includes native support for saving and transferring Python objects between steps. The *ipaascore.BaseStep class* includes multiple functions for transferring data between steps.

Saving Data for the Next Step

The *save_data_for_next_step* function saves an object or other type of data and make the data available to another step. The object to be saved and made available must be able to be serialized with pickle:

```
save_data_for_next_step(data_to_save)
```

where:

- *data_to_save* is a variable that contains the data.

NOTE: The *data_to_save* object must be of a data type that can be pickled by Python: None, True and False, integers, long integers, floating point numbers, complex numbers, normal strings, unicode strings, tuples, lists, set, and dictionaries.

The following is an example of the ***save_data_for_next_step*** function:

```
save_data = {'key': 'value'}
self.save_data_for_next_step(save_data)
```

The integration application must then specify that the data from the current step should be passed to one or more subsequent step, using the **output_to** parameter. For details, see the section about ***using integration applications to transfer data between steps***.

Retrieving Data from a Previous Step

The *ipaascore.BaseStep class* includes multiple functions that retrieve data from a previous step:

- *get_data_from_step_by_name*
- *get_data_from_step_by_order*
- *join_previous_step_data*

NOTE: To retrieve data from a previous step, that previous step must save the data with the ***save_data_for_next_step*** function, and the integration application must specify that the data from the previous step should be passed to the current step using the ***output_to*** parameter.

get_data_from_step_by_name

The ***get_data_from_step_by_name*** function retrieves data saved by a previous step.

NOTE: Although the ***get_data_from_step_by_name*** function is simple to use, it does not allow you to write a generic, reusable step, because the step name will be hard-coded in the function. The functions ***join_previous_step_data*** or **get_data_from_step_by_order** allow you to create a more generic, reusable step.

```
get_data_from_step_by_name(step_name)
```

where:

- *step_name* is the name of a previous step in the integration application.

The following is an example of the ***get_data_from_step_by_name*** function:

```
em7_data = self.get_data_from_step_by_name('FetchDevicesFromEM7')
```

```
snow_data = self.get_data_from_step_by_name('FetchDevicesFromSnow')
```

get_data_from_step_by_order

The ***get_data_from_step_by_order*** function retrieves data from a step based on the position of the step in the integration application.

```
get_data_from_step_by_order(position)
```

where:

- *position* is the position of the step (the order that the step was run) in the integration application. Position starts at "0" (zero).

For example:

- Suppose your integration application has four steps: stepA, stepB, stepC, and stepD
- Suppose stepA was run first (position "0") and includes the parameter "output_to":["stepD"]
- Suppose stepB was run second (position "1") and includes the parameter "output_to":["stepD"]
- Suppose stepC was run third (position "2") and includes the parameter "output_to":["stepD"]
- Suppose stepD was run fourth
- If the current step is stepD, and stepD needs the data from stepC, you could use the following:

```
data_from_stepC = self.get_data_from_step_by_order(2)
```

join_previous_step_data

The *join_previous_step_data* function is the easiest and most generic way of retrieving data from one or more previous steps in the integration application.

If you are expecting similar data from multiple steps, or expecting data from only a single step, the ***join_previous_step_data*** function is the best choice.

The ***join_previous_step_data*** function gathers all data from all steps that included the **save_data_for_next_step** function and also include the ***output_to*** parameter in the integration application. By default, this function returns the joined set of all data that is passed to the current step. You can also specify a list of previous steps from which to join data.

The retrieved data must be of the same type. The data is then combined into a list in a dictionary.

If the data types are not the same, then the function will raise an exception.

```
join_previous_step_data([step_name])
```

where:

- *step_name* is an optional argument that specifies the steps. For example, if you wanted to join only the data from stepA and stepD, you could specify

```
self.join_previous_step_data([“stepA”, “stepD”]),
```

The following is an example of the ***join_previous_step_data*** function in the QueryREST step (included in each Integration Service system):

```
def query_with_url_generated_from_input(self):
"""
Iterates over data from previous steps and generates a relative url for each. Then
executes that command
:return:
"""
count = 0
input_data = self.join_previous_step_data()
payload = self.get_parameter(PAYLOAD)
if type(input_data) is list:
  for input_d in input_data:
    relative_url = self.get_parameter(RELATIVE_URL, input_d)
    self.process_REST_command(payload, relative_url)
    count += 1
  elif type(input_data) is dict:
    relative_url = self.get_parameter(RELATIVE_URL, input_data)
    self.process_REST_command(payload, relative_url)
    count += 1
else:
  raise NotImplementedError("Data type: {} is not currently supported for generating
  relative urls form data".format(type(input_data)))
```

Step Parameters

Steps accept arguments, called ***parameters***. These arguments specify the values, variables, and configurations to use when executing.

Base Parameters Available in All Steps

The Integration services BaseClass has a few base parameters that are automatically inherited by all steps and cannot be overwritten. You do not need to define these parameters before using them in steps:

- ***name***. The Application-unique name for this step. That parameter can be used by other steps to refer to a step.
- ***file***. The name of the file that will be executed by the step. For example, you could write step logic in a single file, but use that step logic with different integration applications and use different names for the step in each integration application.
- ***output_to***. A list indicating that the data retrieved from this step should be output to another step. Setting this parameter links the steps, and the subsequent step will be able to retrieve data from the current step. The format is:

  ```
  "output_to":["stepA", "stepB"]
  ```

Defining a Parameter

From the init method, you define the parameters for the step. The Integration Service system will examine the parameters and enforce the parameters when the step is run.

For example, if you specify a parameter as required, and the user does not specify the required parameter when calling the step, the Integration Service system will display an error message and will not execute the step.

To define a parameter, use the ***new_step_parameter*** function.

```
new_step_parameter(name='', description='', sample_value='', default_value=None,
required=False)
```

where:

- *name*. The name of the parameter. This value will be used to create a name:value tuple in the integration application file (in JSON).
- *description*. A description of the step parameter.
- *sample_value*. A sample value of the required data type or schema.
- *default_value*. If no value is specified for this parameter, use the default value. Can be any combination of alphanumeric characters. To prevent a default value, specify "None".
- *required*. Specifies whether this parameter is required by the step. The possible values are "True" or "False".

Here is an example from the QueryREST step:

```
self.new_step_parameter(name=PREFIX_URL, description="used with relative_url to
create the full URL.", sample_value="http://10.2.11.253", default_value=None,
required=True)
```

Retrieving Parameter Values

To retrieve the latest value of a parameter, use the ***get_parameter*** function.

```
get_parameter(param_name, lookup_data=None)
```

where:

- *param_name* is the name of the parameter that you want to retrieve the value for.
- *lookup_data* is an optional dictionary that can provide a reference for additional variable substitutions.

For example, suppose we defined this parameter in the step named "GETgoogle.com":

```
self.new_step_parameter(name=prefix_url, description="used with relative_url to create
the full URL.", sample_value="http://10.2.11.253", default_value=None, required=True)
```

Suppose in the integration application that calls "GETgoogle.com", we specified:

```
"steps": [
  {
    "file": "QueryREST",
    "method": "GET",
    "name": "GETgoogle.com",
    "output_to": ["next_step"],
    "prefix_url": "http://google.com"
  }
],
```

Suppose we use the ***get_parameter*** function in the step "GETgoogle.com" to retrieve the value of the "prefix_url" parameter:

```
    build_url_1 = self.get_parameter("prefix_url")
```

The value of ***build_url_1*** would be "http://google.com".

Variable Substitution in Parameters

The Integration Service system allows users to define variables, so that parameters can be populated dynamically.

To include a variable in a parameter, use the following syntax:

```
${exampleVariable}
```

The Integration Service system includes the following types of variables for use in parameters:

- ${*object_from_previous_step*}. The Integration Service system will search the data from the previous steps for *object_from_previous_step*. If found, the Integration Service system will substitute the value of the object for the variable
- $**{config**.*exampleVariable*}. Configuration variables are defined in a stand-alone file that lives on the Integration Service system and can be accessed by all integration applications and their steps. Including the **config.** prefix with a variable tells the Integration Service system to look in a configuration file to resolve the variable. If you want to re-use the same settings (like hostname and credentials) between applications, define configuration variables.
- ${**appvar**.*exampleVariable*}. Application variables are defined in the integration application. These variables can be accessed only by steps in the integration application. Including the **appvar.** prefix with a variable tells the Integration Service system to look in the integration application to resolve the variable.
- ${**stepfunc**.*exampleFunctionargs*}. The variable value will be the output from the user-defined function, specified in *exampleFunction*, with the arguments specified in *args*. The *exampleFunction* must exist in the current step. Additional parameters can be specified as *args* with a space delimiter. You can also specify additional variable substitution values as the arguments. This allows you to dynamically set the value of a variable using a proprietary function, with dynamically generated arguments. For example:

  ```
  “param”: “${step_func.add_numbers 1 2}”
  ```

  will call a function (defined in the current step) called “add_numbers” and pass it the arguments "1" and "2". The value of "param" will be "3".

For details on defining configuration variables and application variables, see ***the chapter on defining an integration***.

Retrieving Variable Values

You can use the ***get_app_variable*** function to retrieve the latest value of a variable.

```
get_app_variable (variable_name)
```

where:

- *variable_name* is the name of the application variable that you want to retrieve the value for.

For example, suppose we defined this application variable in the integration application:

```
"app_variables": [
  {
    "name": "sl1_hostname",
    "description": "The SL1 hostname to participate in the sync",
    "sample_value": "10.2.253.115",
    "default_value": null,
    "required": true,
    "value": 10.64.68.25
  },
]
```

Suppose this integration application calls the step "sync_SL1_data".

In the step "sync_SL1_data" , we could use the following function to resolve the value of "sl1hostname":

```
hostname = self.get_app_variable("sl1_hostname")
```

The value of ***hostname*** would be "10.64.68.25".

Defining Logging for the Step

The Integration Service system includes a logger for steps. The BasedStep class initializes the logger, so it is ready for use by each step.

To define logging in a step, use the following syntax:

```
self.logger.logging_level ("log_message")
```

where:

- *logging_level* is one of the following Python logging levels:
  - critical
  - error
  - info
  - warning
- *log_message* is the message that will appear in the step log.

For example :

```
self.logger.info("informational message")
```

Raising Exceptions

The Integration Service platform natively handles exceptions raised from custom steps. You can include a user-defined exception or any standard Python exception.

If an exception is raised at runtime, the step will immediately be marked as a failure and be discarded.

To view the exception and the complete stack trace, use the steps from the section on ***Viewing Logs***.

Uploading Your Step

When you create a new step or edit an existing step, you must upload the step to the Integration Service system. There are two ways to upload a step to the Integration Service system:

- at the command line with the iscli tool
- with the Integration Service API

Uploading a Step with iscli

The Integration Service system includes a command line tool called ***iscli***. When you install Integration Service system, ***iscli*** is automatically installed.

To upload a step to the Integration Service system using iscli:

1. Either go to the console of the Integration Service system or use SSH to access the server.
2. Log in as ***isadmin*** with the appropriate password.
3. Enter the following at the command line:

```
iscli -u -s -f path_and_name_of_step_file.py -H hostname_or_IP_address_of_
integration_service_system -P port_number_of_http_on_integration_service_system -u
user_name -p password
```

   where:

   - *name_of_step_file* is the full pathname for the step.
   - *hostname_or_IP_address_of_integration_service_system* is the hostname or IP address of the Integration Service system.
   - *port_number_of_http_on_integration_service_system* is the port number to access the Integration Service system. The default value is 443.
   - *user_name* is the user name you use to log in to the Integration Service system.
   - *password* is password you use to log in to the Integration Service system.

Uploading a Step with the API

The Integration Service system includes an API. When you install the Integration Service system, the API is available.

To upload a step with the API ***POST /steps***:

```
POST /steps
  {
    "name": "name_of_step",
    "data": "string"
  }
```

where:

- *data* is all the information included in the step.

Validating Your Step

After uploading a step, you can use the API ***POST /steps run*** to run the step individually without running an integration application. This allows you to validate that the step works as designed.

To run a step from the IS API:

```
POST /steps/run
  {
    "name": "name_of_step",
    all other data from the .json file for the step
  }
```

After the POST request is made, the Integration Service system will dispatch the step to a remote worker process for execution. By default, the POST request will wait five seconds for the step to complete. To override the default wait period, you can specify wait time as a parameter in the POST request. For example, to specify that the wait time should be 10 seconds :

```
POST /steps/run?wait=10
  {
    "name": "example_step",
  }
```

If the step completes within the wait time, the Integration Service system returns a 200 return code, logs, output, and the result of the step.

If the step does not complete within the wait time, the Integration Service system returns a task ID. You can use this task ID to view the logs for the step.

The API returns one of the following codes:

- 200. Step executed and completed within the timeout period.
- 202. Step executed but did not complete with timeout period or user did not specify wait. Returned data includes task to query for the status of the step
- 400. Required parameter for the step is missing.
- 404. Step not found.
- 500. Internal error. Database connection might be lost.

Viewing Logs

After running a step, you can view the log information for a step. Log information for a step is saved for the duration of the ***result_expires*** setting in the Integration Service system. The ***result_expires*** setting is defined in the file opt/iservices/scripts/docker-compose.yml. The default value for log expiration is 24 hours.

NOTE: To view the log information for a step before running an integration, you can use the API *POST /steps run* to run the step individually without running an integration application. You can then use the information in step #3 - step #6 below to view step logs.

To view the log information for a step:

1. Run an integration application.
2. Using an API tool like Postman or cURL, use the API *GET /applications/{appName}/logs*:

```
GET URL_for_your_Integration_Service _system/api/v1/applications/integration_
application_name/logs
```

   where:

   - *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.

3. You should see something like this:

```
{
  "app_name": "example_integration",
  "app_vars": {},
"href": "/api/v1/tasks/isapp-af7d3824-c147-4d44-b72a-72d9eae2ce9f",
  "id": "isapp-af7d3824-c147-4d44-b72a-72d9eae2ce9f",
  "start_time": 1527429570,
  "state": "SUCCESS",
  "steps": [
    {
      "href": "/api/v1/tasks/2df5e7d5-c680-4d9d-860c-e1ceccd1b189",
      "id": "2df5e7d5-c680-4d9d-860c-e1ceccd1b189",
      "name": "First EM7 Query",
      "state": "SUCCESS",
      "traceback": null
    },
    {
      "href": "/api/v1/tasks/49e1212b-b512-4fa7-b099-ea6b27acf128",
      "id": "49e1212b-b512-4fa7-b099-ea6b27acf128",
            "name": "second EM7 Query",
            "state": "SUCCESS",
            "traceback": null
          }
        ],
        "triggered_by": [
          {
            "application_id": "isapp-af7d3824-c147-4d44-b72a-72d9eae2ce9f",
            "triggered_by": "USER"
          }
        ]
      }
```

4. In the `"steps"` section, Notice the lines that start with ***href*** and **id**. You can use these lines to view the logs for the application and the steps.
5. To use the **href** value to get details about a step, use an API tool like Postman or cURL and then use the API *GET /steps{step_name}*:

   ```
   GET URL_for_your_Integration_Service _system/href_value
   ```

   where:

   - *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.
   - *href_value* is the href value you can copy from the log file for the integration application. The href value is another version of the step name.

NOTE: To view logs for subsequent runs of the integration application, you can include the href specified in the ***last_run*** field.

6. To use the task **id** value to views details about a step, use an API tool like Postman or cURL and then use the API *GET /tasks/{task_ID}*:

   ```
   GET URL_for_your_Integration_Service _system/task_id
   ```

   where:

   - *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.
   - *task_id* is the id value you can copy from the log file for the integration application. The task ID specifies the latest execution of the step.

NOTE: After you find the href and task ID for a step, you can use those values to retrieve the most recent logs and status of the step.

Chapter

3

Creating an Integration Application

Overview

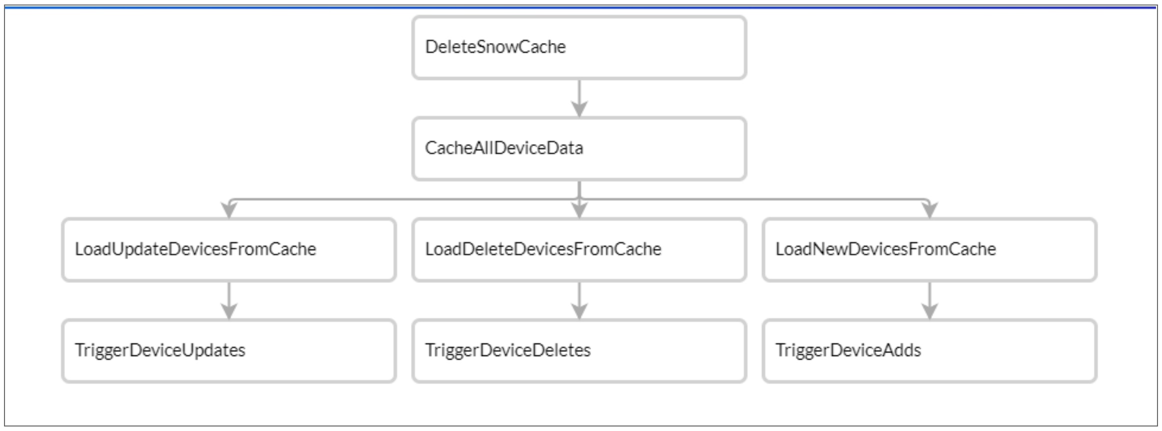
This chapter explains how to create your own integration application that can run on an Integration Service system.

This chapter covers the following topics:

What is an Integration Application?

In the Integration Service, an ***integration application*** is a JSON file that specifies which steps to execute and the order in which to execute those steps. An integration application also defines variables and provides arguments for each step.

The following is an example of an integration application:



Integration application JSON objects are defined by configuration settings, steps that make up the integration, and application-wide variables to use as parameters for each step. The parameters of each step can be configured dynamically, and each step can be named uniquely while still sharing the same underlying class, allowing for maximum re-use of code.

Integration applications can be executed through the REST API and are processed as an asynchronous task in the Integration Service. During processing the user is provided a unique task ID for the application and each of its tasks. Using the task IDs, the user can poll for the status of the integration application and the status of each individual running step in the integration application.

Executing an integration application from the REST API allows the user to dynamically set one-time parameter values for the variables defined in the integration.

The required parameters of integration applications are strictly enforced, and the Integration Service will refuse to execute the integration application if all required variables are not provided.

Default Integration Application

The Integration Service system includes a pre-defined integration application called ***integration_template***. This default integration appears in the Integration Service user interface as the ***Template App*** integration application.

To view this integration application:

1. Using an API tool like Postman or cURL, use the API ***GET /applications/{application_name}***:

   ```
   GET URL_for_your_Integration_Service _system/api/v1/applications/integration_
   template
   ```

   where:

   - *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.

Requirements

To create an integration application, you must perform these tasks:

1. Download or copy the application template, called ***integration_template***.
2. Define the required fields for the integration application.
3. Create the list of steps and step parameters.
4. Define application variables.
5. Upload the integration applecart to the Integration Service.
6. Validate and test the integration applications step.

Creating an Integration Application from the Integration Template

The easiest way to create a new integration application is to use the application template that is included with the Integration Service system. To copy this template to your desktop:

1. Using an API tool like Postman or cURL, use the API ***GET /applications/{application_name }***:

   ```
   GET URL_for_your_Integration_Service _system/api/v1/applications/integration_
   template
   ```

   where:

   - *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.

2. Select and copy all the text in the application.
3. Open a source-code editor and paste the application in the source-code editor.
4. Save the new file as *newfilename*.json

   where:

   - *newfilename****.json*** is the new name of the integration application includes the .json suffix. The file name must be unique within your Integration Server system and cannot contains spaces.

Defining Required Fields for the Integration Application

Each integration application should include the following key:value pairs

```
"author": "name of the author",
"configuration": "if this integration application uses configuration variables,
specify the name of the configuration",
"description": "description of the integration application",
"friendly_name": "name that appears in the user interface",
"name": "file name without the py suffix".
"version": "version number",
```

The integration_template includes these required fields. You can edit the value for each key. To do so:

1. Using an API tool like Postman or cURL, use the API ***GET /applications/{application_name}***:

   ```
   GET URL_for_your_Integration_Service _system/api/v1/applications/application_name
   ```

   where:

   - *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.
   - *application_name* is the integration application you want to edit.

2. Copy the contents of the integration application to a source-code editor.
3. Search for the section of the file that contains the text "description".
4. Supply new values for the following keys:

   - ***author***. User who created the integration template.
   - ***configuration*** (if applicable). Configuration variables are defined in a stand-alone file (called a configuration) that lives on the Integration Service system and can be accessed by all integration applications and their steps. If your integration application or the steps in the application reference configuration variables, you must specify the name of the configuration in this value. For more details on creating a configuration, see the section on ***Defining a Configuration***.
   - ***description***. This description will be displayed in the user interface when viewing the available Integrations.
   - ***friendly_name***. The name of the integration application as it will appear in the user interface.
   - ***name***. Name of the file, without the .py suffix.
   - ***Version***. Version number of the integration application.

5. Save your changes. Save the file to the same name (*application_name*.json.)
6. To upload the integration application to the Integration Service system, see the section on ***Uploading an Integration Application.***

Creating the List of Steps and Step Parameters

In the integration application, you must specify the steps to execute and the order in which they should be executed.

If steps do not have dependencies, the Integration Service system will execute steps in parallel. If steps have dependencies (meaning one of the steps requires data from another step), the Integration Service system will execute the step that provides data and then execute the step that consumes that data.

To edit the steps section of the integration application:

1. Using an API tool like Postman or cURL, use the API ***GET /applications/{application_name}***:

   ```
   GET URL_for_your_Integration_Service _system/api/v1/applications/application_name
   ```

   where:

   - *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.
   - *application_name* is the integration application you want to edit.

2. Copy the contents of the integration application to a source-code editor.
3. Search for the section of the file that contains the text "steps".
4. The section of the integration application that specifies steps should look like this:

   ```
   "steps":[
     {
       "name": "GETgoogle.com",
       "file": "QueryREST",
       "prefix_url": "http://google.com",
       "method": "GET",
       "output_to": ["next_step"]
     },
     {
       "name": "next_step",
       "file": "someOtherTask"
     }
   ]
   ```

5. Edit the "steps" section as needed and save your changes. Be sure to your file as a .JSON file with the same name as the integration application your downloaded.
6. To upload the integration application to the Integration Service system, see the section on ***Uploading an Integration Application.***

Specifying Values for name and file

For each step, you must specify its name and the file it executes. These two keys can have the same value.

- ***name***. The name for the step. Other steps in the integration application can use this name to refer to the step. This value can include spaces. This name must be unique to the integration application.
- ***file***. The name of the file that will be executed by the step. You could write step logic in a single file but use that step logic with different integration applications and use different names for the step in each integration application.

Specifying the Parameters

When you add a step to an integration application, you must view the step and determine if it includes any required parameters.

To view details about a step (in our example, the step file is named QueryREST):

1. Using an API tool like Postman or cURL, use the API ***GET /steps/{step_name}***:

```
GET URL_for_your_Integration_Service _system/api/v1/steps/QueryREST
```

where:

- *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.

2. The step GETgoogle.com (and its parent file, QueryREST) includes the following parameters that are defined as "required":

```
self.new_step_parameter(name=METHOD, sample_value="GET", description="HTTP method
to perform", required=True)

self.new_step_parameter(name=PREFIX_URL, description="used with relative_url to
create the full URL.",sample_value="http://10.2.11.253", default_value=None,
required=True)
```

3. Therefore, when the integration application includes the step GETgoogle.com, the integration application must supply values for the required parameters.

```
"method": "GET",
"prefix_url": "http://google.com",
```

Transferring Data Between Steps

An essential part of integrations is passing data between tasks. The Integration Service system includes native support for saving and transferring Python objects between steps. Within a step, you can use one of the functions in the included in the ipaascore.BaseStep class.

In an integration application, you can use the ***ouput_to*** key to specify that the results of a step should be piped to one or more specified steps. The output_to key uses the following syntax:

```
"output_to": ["step_name1", "step name2"]
```

To pass data from step to step, you must include the output_to key along with the parameters for a step.

To edit the step parameters in the integration application:

1. Using an API tool like Postman or cURL, use the API ***GET /applications/{application_name}***:

   ```
   GET URL_for_your_Integration_Service _system/api/v1/applications/application_name
   ```

   where:

   - *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.
   - *application_name* is the integration application you want to edit.

2. Copy the contents of the integration application to a source-code editor.
3. If you wanted the step "GETgoogle.com" to pass its output to the step "next-step",the integration application would include the following :

```
"steps":[
  {
    "name": "GETgoogle.com",
    "file": "QueryREST",
    "prefix_url": "http://google.com",
    "method": "GET",
    "output_to": ["next_step"]
  },
  {
    "name": "next_step",
    "file": "someOtherTask"
  }
]
```

   - If steps have dependencies , the Integration Service system will first analyze all steps in an integration application and ensure that data-gathering steps are performed before steps that require that data.

4. Edit the "steps" section as needed and save your changes. Be sure to your file as a .JSON file with the same name as the integration application your downloaded.
5. To upload the integration application to the Integration Service system, see the section on ***Uploading an Integration Application.***

Defining Retry Options for a Step

The following parameters allows you to define multiple retry options for a step. You can specify that the Integration Service system try to re-run a step if that step fails. Retries work following the rules of exponential backoff: the first retry will have a delay of 1 second, the second retry will have a delay of 2 seconds, the third retry will delay 4 seconds, the fourth retry will delay 8 seconds, and so on. You can edit the ***retry_max*** and ***retry_backoff_max*** values, below to further refine the retry settings.

You can include the following retry options in the integration application file, where you define parameters for each step:

- ***retry_max*** . The maximum number of times the Integration Service system will retry to execute the step before it stops retrying and logs a step failure. For example, if ***retry_max*** is 3, the Integration Service will retry after 1 second, then 2 seconds, then 4 seconds, and stop if the last retry fails. Possible values are:

- Integer that specifies number of retries.
- The default value is "0" (zero).

- ***retry_backoff***. Instead of using a defined interval between retries, the Integration Service system will incrementally increase the interval between retries. Possible values are:
  - True
  - False
  - The default value is "False".
- ***retry_jitter***. Instead of using a defined interval between retries, the Integration Service system will retry the step execution at random intervals. Possible values are:
  - True
  - False
  - The default value is "False".
- ***retry_backoff_max***. The maximum time interval for the ***retry_backoff*** option, in seconds. For example, This means, if you have *retry_max* set to 15, the delays will be 1, 2, 4, 8, 16, 32, 64, 120, 240, 480, 600, 600, 600, 600, and 600. Possible values are:
  - Integer that specifies number of seconds
  - The default value is "600"
- ***retry_countdown***. The interval between retries, in seconds. If you enabled ***retry_backup***, the Integration Service system will incrementally increase this interval:
  - Integer that specifies number of seconds
  - The default value is "180"

To define the retry options for a step in the integration application file:

1. Using an API tool like Postman or cURL, use the API GET **/applications/{application_name}**:

   ```
   GET URL_for_your_Integration_Service _system/api/v1/applications/application_name
   ```

   where:

   - *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.
   - *application_name* is the integration application you want to edit.
2. Copy the contents of the integration application to a source-code editor.
3. If you wanted to add retry options to the step "GETgoogle.com", you could include the following:

```
"steps":[
  {
    "name": "GETgoogle.com",
    "file": "QueryREST",
    "prefix_url": "http://google.com",
```

```
      "method": "GET",
      "retry_max": 5
      "retry_backoff": True
      "retry_backoff_max": 600
      "retry_countdown": 120
      "output_to": ["next_step"]
    },
    {
      "name": "next_step",
      "file": "someOtherTask"
    }
  ]
```

4. Edit the "steps" section as needed and save your changes. Be sure to your file as a .JSON file with the same name as the integration application your downloaded.

Defining Variables for an Integration Application

Application variables are defined in the integration application. These variables can be accessed only by steps in the integration application. In a step, including the **appvar.** prefix in a variable tells the Integration Service system to look in the integration application to resolve the variable. In a step, application variables are used in step parameters.

To define an application variable:

1. Using an API tool like Postman or cURL, use the API ***GET /applications/{application_name}***:

   ```
   GET URL_for_your_Integration_Service _system/api/v1/applications/application_name
   ```

   where:

   - *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.
   - *application_name* is the integration application you want to edit.

2. Copy the contents of the integration application to a source-code editor.
3. Search for the section of the file that contains the text "app_variables".
4. In the integration_template file, the section looks like this:

   ```
   "app_variables": [
     {
       "name": "exampleVariable", "value": "exampleValue","description": "Variables
       defined here will be available in all steps of an application","required":
       true,"sample_value": "SampleValue", "default_value": "DefaultValue"
     }
   ],
   ```

5. You can copy and paste the example section above for each application variable you want to define.
6. For each application variable, supply values for the following keys:

- ***name***. This key requires a value. The name of the variable.
- ***value***. This key requires a value. This is the value that the variable will resolve to at runtime.
- ***description***. A description of this variable. This description appears in the user interface. The default value is null.
- ***required***. Specifies whether the variable is required. Possible values are "True" or "False". The default value is "False". If the variable is required, but a value is not specified and a default value is not specified, the Integration Application will fail.
- ***sample_value***. A sample value of the required data type or schema. The default value is null. For example, if the parameter is "port" and the expected value is an integer, a good ***sample_value*** would be "443".
- ***default_value***. If no value is specified for this variable (in the ***value*** key), use the value of this key. The default value of this key is null.

7. Edit the "app_variables" section as needed and save your changes. Be sure to your file as a .JSON file with the same name as the integration application your downloaded.
8. To upload the integration application to the Integration Service system, see the section on ***Uploading an Integration Application.***

Uploading the Integration Application to the Integration Service System

When you create a new integration application or edit an existing integration application, you must upload the integration application to the Integration Service system. There are two ways to upload an integration application to the Integration Service system:

- at the command line with the iscli tool
- with the Integration Service API

Uploading an Integration Application with iscli

The Integration Service system includes a command line tool called ***iscli***. When you install Integration Service system, ***iscli*** is automatically installed.

To upload an integration application to the Integration Service system using iscli:

1. Either go to the console of the Integration Service system or use SSH to access the server.
2. Log in as ***isadmin*** with the appropriate password.
3. Enter the following at the command line:

```
iscli -u -a -f path_and_name_of_integration application_file.py -H hostname_or_IP_
address_of_integration_service_system -P port_number_of_http_on_integration_
service_system -U user_name -p password
```

   where:

- *name_of_integration application_file* is the full pathname for the integration application.
- *hostname_or_IP_address_of_integration_service_system* is the hostname or IP address of the Integration Service system.
- *port_number_of_http_on_integration_service_system* is the port number to access the Integration Service system. The default value is 443.
- *user_name* is the user name you use to log in to the Integration Service system.
- *password* is password you use to log in to the Integration Service system.

Uploading an Integration Application with the API

The Integration Service system includes an API. When you install the Integration Service system, the API is available.

To upload an integration application with the IS API:

```
POST /applications
  {
    "name": "name_of_integration application",
    contents of the .json file for the integration application
  }
```

The API returns one of the following:

- 200. Application successfully updated or added.
- 400. Name or data parameter is missing.
- 500. Internal error. Database connection might be lost.

Running the Integration Application

After uploading an integration application, you can run it to ensure that it works as designed. To run an integration application from the IS API:

1. Using an API tool like Postman or cURL, use the API ***POST/applications/{application_name}/run***:

   ```
   POST https"//URL_for_your_Integration_Service _system/api/v1/applications/run
   ```

   where:

   - *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.
   - *application_name* is the integration application you want to edit.

2. The body of the post should contain:

   ```
   {
     "name": "name_of_integration application",
     "params": {}
   }
   ```

3. You can include the following parameters with the POST request:

   - parameters use the syntax:

     ```
     "parameter name":"parameter value"
     ```

   - ***wait***. Number of seconds to wait for integration application to complete.
   - ***configuration***. The configuration file to use with the integration application. For details, see the section on *configurations*.

4. The API returns one of the following:

   - 200. Application started successfully.
   - 400. Name or data parameter is missing.
   - 404. Application not found.
   - 500. Internal error. Database connection might be lost.

Running the Integration Application with a Custom Queue

If your Integration Service system runs multiple integration applications and you might want to ensure that one of those integration applications runs before other integration applications in the processing queues. To do this, you can align a queue with a worker process; the worker process will be dedicated to only that queue, will listen only to that queue, and process only jobs from that queue. You can then configure the high-priority integration application to use the queue that you aligned with its own worker process.

There are two steps to using a custom queue:

- Defining the queue and aligning it with a worker
- Configuring the Integration Application to use the custom queue

Defining a Custom Queue

To create a custom queue:

1. SSH to the Integration Service system
2. Use a text editor like vi to edit the file **/opt/iservices/scripts/docker-compose.yml**.
3. The docker-compose.yml file contains definitions for worker processes. For example, you might see something like this:

```
services:
  steprunner:
    image: sciencelogic/is-worker:latest
    environment:
    LOGLEVEL: 10
    celery_log_level: 10
    logdir: /var/log/iservices
    broker_url: 'pyamqp://guest@rabbit//'
    result_backend: 'redis://redis:6379/0'
    db_host: 'couchbase,localhost'
```

```
  secrets:
    - is_pass
    - encryption_key
  deploy:
  replicas: 2
  networks:
    - isnet
  depends_on:
    - redis
    - rabbitmq
    - couchbase
  volumes:
    - "/var/log/iservices:/var/log/iservices"
    - "statedb:/var/run/celery/states/"

steprunner_1:
  image: sciencelogic/is-worker:latest
  environment:
  LOGLEVEL: 10
  celery_log_level: 10
  logdir: /var/log/iservices
  broker_url: 'pyamqp://guest@rabbit//'
  result_backend: 'redis://redis:6379/0'
  db_host: 'couchbase,localhost'
  user_queues: 'test_queue'
  secrets:
    - is_pass
    - encryption_key
  deploy:
  replicas: 2
  networks:
    - isnet
  depends_on:
    - redis
    - rabbitmq
    - couchbase
  volumes:
    - "/var/log/iservices:/var/log/iservices"
    - "statedb:/var/run/celery/states/"

steprunner_2:
  image: sciencelogic/is-worker:latest
  environment:
  LOGLEVEL: 10
  celery_log_level: 10
  logdir: /var/log/iservices
  broker_url: 'pyamqp://guest@rabbit//'
  result_backend: 'redis://redis:6379/0'
  db_host: 'couchbase,localhost'
  user_queues: 'critical_queue'
  secrets:
    - is_pass
    - encryption_key
  deploy:
  replicas: 2
  networks:
    - isnet
  depends_on:
    - redis
    - rabbitmq
    - couchbase
```

```
volumes:
  - "/var/log/iservices:/var/log/iservices"
  - "statedb:/var/run/celery/states/"
```

4. The services with names that start with "steprunner" are the workers for the Integration Service system.

   - Notice that the service named "steprunner" does not include any queues. This means that the worker "steprunner" listens to the default queues in the Integration Service system.

NOTE: ScienceLogic recommends that you allot at least one worker to handle the default queues.

   - To add additional services to your Integration Service system. copy all the lines included in the service definition, paste it into the file, and rename the service.
   - For example, you could copy all the lines from "steprunner_2", paste the text in the "services" section, and rename the service "steprunner_3".

5. To create one or more queues and dedicate a worker to them, enter the following line in definition of the worker, under the ***environment*** section:

   ```
   user_queues: 'queue_name1, queue_name2'
   ```

   where:

   - *queue_name1, queue_name2*. The name of the new queues. The worker will monitor only these queues and execute tasks only from these queues.

6. After you have updated the docker-compose file, you can update and re-deploy the Integration Service system to pick up the changes to the docker-compose.yml file. To do this, SSH to the Integration Service system and execute the following command:

   ```
   docker stack deploy -c /opt/iservices/scripts/docker-compose.yml is4
   ```

Configuring an Integration Application to Use a Custom Queue

There are two ways to specify that an integration application should use a custome queue:

- In the .JSON file for an integration applicaiton, you can specify that that integration application should always use a custom queue.
- At run time, you can specify that an integration application shoudl use a custom queue only for that single execution of the integration application.

Configuring an Integration Application to Always Use a Custom Queue.

To specify that an integration application should always use a custom queue, edit the .json file for the integration application. To do this:

1. Using an API tool like Postman or cURL, use the API ***GET /applications/{application_name}***.

   ```
   GET URL_for_your_Integration_Service _system/api/v1/applications/application_name
   ```

where:

- *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.
- *application_name* is the integration application you want to edit.

2. Copy the contents of the integration application to a source-code editor.
3. Search for the section of the file that contains the lines "author", "configuration", "description", "friendly_ name", and "name".
4. In that section of the file, added the bolded line:

   "queue": "*name_of_queue*",

   where:

   - *name_of_queue* is one of the custom queues you defined in the docker-compose.yml file.

5. For example:

```
"author": "ScienceLogic, Inc.",
"configuration": "",
"description": "Read SL1 and ServiceNow devices and write them to a cache.",
"friendly_name": "Cache SL1 Devices using GraphQL",
"name": "cache_em7_device_data",
"generate_report": true,
"queue": "test_queue",
```

6. Save your changes. Save the file to the same name (*application_name*.json.)
7. The integration application will now always use the specified queue.
8. To upload your changes to the Integration Service system, see the section on ***Uploading an Integration Application***.

Configuring an Integration Application to Use a Custom Queue at Run Time

After uploading an integration application, you can run it and specify a queue. To run an integration application from the IS API:

1. Using an API tool like Postman or cURL, use the API *POST/applications/{application_name}/run*:

   ```
   POST https://URL_for_your_Integration_Service _system/api/v1/applications/run
   ```

   where:

   - *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.
   - *application_name* is the integration application you want to edit.

2. The body of the post should contain:

   ```
   {
     "name": "name_of_integration application",
     "params": {
     "queue": "custom_queue"
     }
   }
   ```

3. You can include the following parameter with the POST request:

   ```
   "queue": "name_of_queue"
   ```

   where:

   - *name_of_queue* is one of the custom queues you defined in the docker-compose.yml file.

4. The API returns one of the following:

   - 200. Application started successfully.
   - 400. Name or data parameter is missing.
   - 404. Application not found.
   - 500. Internal error. Database connection might be lost.

Defining a Configuration

Configuration variables are defined in a stand-alone JSON file called a configuration that lives on the Integration Service system and can be accessed by all integration applications and their steps.

In a step, including the **config.** prefix with a variable tells the Integration Service system to look in a configuration file to resolve the variable.

If you want to re-use the same settings (like hostname and credentials) between applications, define configuration variables.

To define a configuration file:

1. Use a source-code editor to open a new file.
2. Copy and paste the following example text into the new file:

```
{
  "author": "ScienceLogic, Inc.",
  "description": "Fsun's config for demo.",
  "name": "fsun-demo-settings",
  "friendly_name": "Demo Settings",
  "configuration_data": [
    {
      "encrypted": false,
      "name": "em7_host",
      "value": "10.2.11.42"
    },
    {
      "encrypted": false,
      "name": "em7_user",
      "value": "em7admin"
    },
    {
      "encrypted": true,
      "name": "em7_password",
      "value": "+dqGJe1NwTyvdaO2EizTWjJ2uj2C1wzBzgNqVhpdTHA="
    },
  ],
  }
```

3. To create your own configuration file, edit the following keys:

   - ***author***. Name of the author of the configuration file. This field is optional.
   - ***description***. Description of the configuration file. This field is optional.
   - ***name***. Name of the configuration file. The name cannot contain any spaces and should be unique in your Integration Service system
   - ***friendly_name***. User-friendly name for the configuration.

4. The section under "configuration_data" defines each configuration variable. You can use these variables as examples.
5. To create your own configuration variables, note the syntax:

   - The variable definition is surrounded by curly braces
   - Each key:value pair in the definition is separated with a comma.

6. To create your own configuration variable, define the following keys:

   - ***encrypted***. Specifies whether the value will appear in plain text or encrypted in this .json file. If set to "true", specifies that when the value is uploaded, the Integration Service system will encrypt value of the variable. The plain text value will never again be retrievable by an end user. The encryption key is unique to each Integration Service system.
   - ***name***. Name of the configuration file (without the .json suffix). This value appears in the user interface.
   - ***value***. The value to assign to the variable.

7. Repeat steps 5 and 6 for each configuration variable.
8. Save the new file as *config_file*.json

   where:

   - *config_file**.json*** is the name of the configuration file. This value must match the ***name*** key in the file. The file name must be unique within your Integration Server system and cannot contains spaces.

Uploading the Configuration File to the Integration Service System

When you create a new configuration file or edit an existing configuration file, you must upload the step to the Integration Service system. There are two ways to upload a configuration file to the Integration Service system:

- at the command line with the iscli tool
- with the Integration Service API

Uploading a Configuration with iscli

The Integration Service system includes a command line tool called ***iscli***. When you install Integration Service system, ***iscli*** is automatically installed.

To upload a configuration file to the Integration Service system using iscli:

1. Either go to the console of the Integration Service system or use SSH to access the server.
2. Log in as ***isadmin*** with the appropriate password.

3. Type the following at the command line:

   ```
   iscli -u -c -f path_and_name_of_configuration__file.py -H hostname_or_IP_address_
   of_integration_service_system -P port_number_of_http_on_integration_service_system
   -U user_name -p password
   ```

   where:

   - *path_and_name_of_configuration__file* is the full pathname for the step.
   - *hostname_or_IP_address_of_integration_service_system* is the hostname or IP address of the Integration Service system.
   - *port_number_of_http_on_integration_service_system* is the port number to access the Integration Service system. The default value is 443.
   - *user_name* is the user name you use to log in to the Integration Service system.
   - *password* is password you use to log in to the Integration Service system.

Uploading a Configuration with the API

The Integration Service system includes an ***API***. When you install the Integration Service system, the API is available.

To upload a configuration with the IS API:

```
POST /configurations
  {
    contents of the .json file for the configuration
  }
```

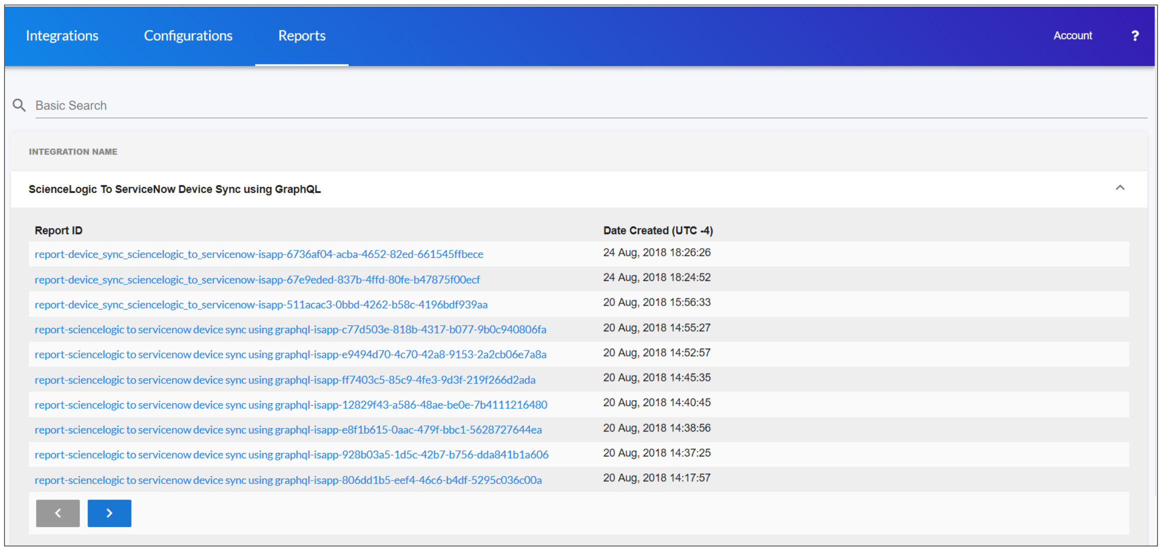
The API returns one of the following:

- 200. Configuration successfully updated or added.
- 400. Name or data parameter is missing.
- 500. Internal error. Database connection might be lost.

Viewing a Report for an Integration Application

In the Integration Service user interface, the **[Reports]** tab contains a list of reports associated with integration applications. If an integration application has the reporting feature enabled and the application supports reports, then the Integration Service will generate a report each time you run the integration application.

An individual report displays data only from the most recent run of the integration application; a report is not an aggregation of all previous runs.



ScienceLogic To ServiceNow Device Sync using GraphQL

Report ID	Date Created (UTC -4)
report-device_sync_sciencelogic_to_servicenow-isapp-6736af04-acba-4652-82ed-661545ffbece	24 Aug, 2018 18:26:26
report-device_sync_sciencelogic_to_servicenow-isapp-67e9eded-837b-4ffd-80fe-b47875f00ecf	24 Aug, 2018 18:24:52
report-device_sync_sciencelogic_to_servicenow-isapp-511acac3-0bbd-4262-b58c-4196bdf939aa	20 Aug, 2018 15:56:33
report-sciencelogic to servicenow device sync using graphql-isapp-c77d503e-818b-4317-b077-9b0c940806fa	20 Aug, 2018 14:55:27
report-sciencelogic to servicenow device sync using graphql-isapp-e9494d70-4c70-42a8-9153-2a2cb06e7a8a	20 Aug, 2018 14:52:57
report-sciencelogic to servicenow device sync using graphql-isapp-ff7403c5-85c9-4fe3-9d3f-219f266d2ada	20 Aug, 2018 14:45:35
report-sciencelogic to servicenow device sync using graphql-isapp-12829f43-a586-48ae-be0e-7b4111216480	20 Aug, 2018 14:40:45
report-sciencelogic to servicenow device sync using graphql-isapp-e8f1b615-0aac-479f-bbc1-5628727644ea	20 Aug, 2018 14:38:56
report-sciencelogic to servicenow device sync using graphql-isapp-928b03a5-1d5c-42b7-b756-dda841b1a606	20 Aug, 2018 14:37:25
report-sciencelogic to servicenow device sync using graphql-isapp-806dd1b5-eef4-46c6-b4df-5295c036c00a	20 Aug, 2018 14:17:57

NOTE: Not all integration applications generate reports. Currently, only the "Sync Devices from SL1 to ServiceNow" and "Integration Service System Diagnostics" integration applications support the generation of reports.

An integration application report includes the following fields:

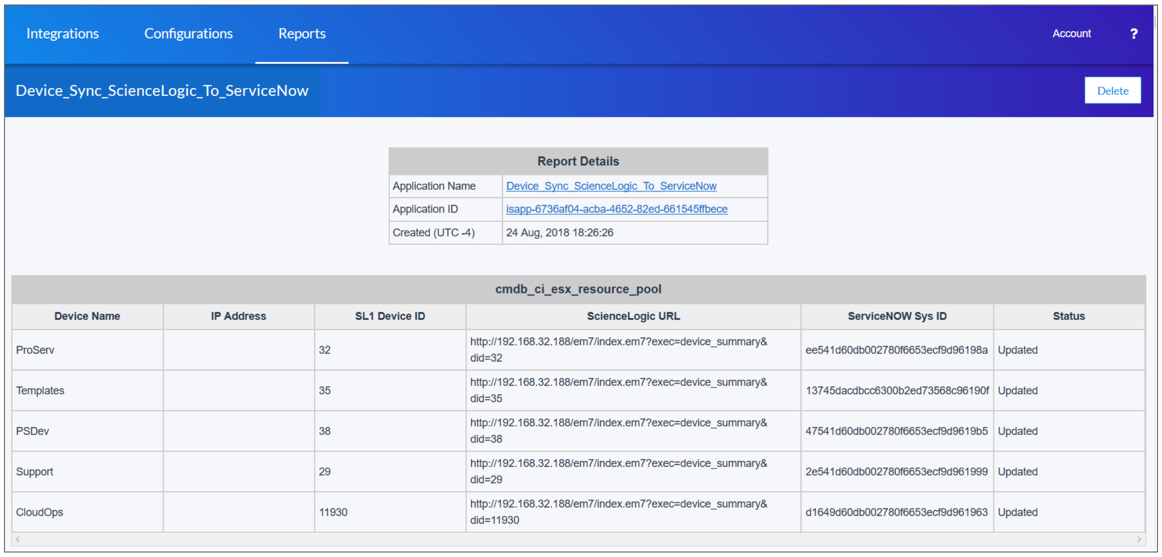
- ***Device Name***
- ***IP Address***
- ***SL1 Device ID***
- ***ScienceLogic URL***
- ***ServiceNow Sys ID***
- ***Status***. The current state of the synced item, which can include *New*, *Removed*, *Updated*, or *Unchanged*.

To view details for an integration application report:

1. On the **[Reports]** tab, click the name of the integration application to expand the list of reports for that application.

TIP: Click the arrow buttons to scroll forward and back through the list of reports.

2. Click a report name in the **Report ID** column. The **Report Details** page appears:



Report Details	
Application Name	Device_Sync_ScienceLogic_To_ServiceNow
Application ID	isapp-6736af04-acba-4652-82ed-661545ffbece
Created (UTC -4)	24 Aug, 2018 18:26:26

cmdb_ci_esx_resource_pool

Device Name	IP Address	SL1 Device ID	ScienceLogic URL	ServiceNOW Sys ID	Status
ProServ		32	http://192.168.32.188/em7/index.em7?exec=device_summary&did=32	ee541d60db002780f6653ecf9d96198a	Updated
Templates		35	http://192.168.32.188/em7/index.em7?exec=device_summary&did=35	13745dacdbcc6300b2ed73568c96190f	Updated
PSDev		38	http://192.168.32.188/em7/index.em7?exec=device_summary&did=38	47541d60db002780f6653ecf9d9619b5	Updated
Support		29	http://192.168.32.188/em7/index.em7?exec=device_summary&did=29	2e541d60db002780f6653ecf9d961999	Updated
CloudOps		11930	http://192.168.32.188/em7/index.em7?exec=device_summary&did=11930	d1649d60db002780f6653ecf9d961963	Updated

3. To view the detail page for the integration application on the **[Integrations]** tab, click the **Application Name** link.

TIP: From the detail page for the integration application, click the **[Reports]** button to return to the **[Reports]** tab.

4. To view the specific run-time instance for the integration application that generated the report, click the **Application ID** link.
5. To delete a report, click the **[Delete]** button. Click **[OK]** to delete the report.

Viewing Logs for an Integration Application

After running an integration application, you can view its log information. To view the log information for an integration application:

1. Run an integration application.
2. Using an API tool like Postman or cURL, use the API *GET /applications/{appName}/logs*:

```
GET URL_for_your_Integration_Service _system/api/v1/applications/integration_
application_name/logs
```

where:

- *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.

3. You should see something like this:

```
{
    "app_name": "example_integration",
    "app_vars": {},
    "href": "/api/v1/tasks/isapp-af7d3824-c147-4d44-b72a-72d9eae2ce9f",
    "id": "isapp-af7d3824-c147-4d44-b72a-72d9eae2ce9f",
    "start_time": 1527429570,
    "state": "SUCCESS",
    "steps": [
    {
      "href": "/api/v1/tasks/2df5e7d5-c680-4d9d-860c-e1ceccd1b189",
      "id": "2df5e7d5-c680-4d9d-860c-e1ceccd1b189",
      "name": "First EM7 Query",
      "state": "SUCCESS",
      "traceback": null
    },
    {
      "href": "/api/v1/tasks/49e1212b-b512-4fa7-b099-ea6b27acf128",
      "49e1212b-b512-4fa7-b099-ea6b27acf128",
      "name": "second EM7 Query",
      "state": "SUCCESS",
      "traceback": null
    }
  ],
  "triggered_by": [
    {
      "application_id": "isapp-af7d3824-c147-4d44-b72a-72d9eae2ce9f",
      "triggered_by": "USER"
    }
  ]
}
```

4. Notice the bolded lines that start with ***href*** and **id**. You can use these lines to view the logs for the application.

5. To use the **href** value to get details about an application, use an API tool like Postman or cURL and use the API *GET /applications/{appName}*:

```
GET URL_for_your_Integration_Service _system/href_value
```

where:

- *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.
- *href_value* is the href value you can copy from the log file. The href value is another version of the application name.

NOTE: To view logs for subsequent runs of the integration application, you can include the href specified in the ***last_run*** field.

6. To use the task **id** value to views details about an application, use an API tool like Postman or cURL and use the API *GET /tasks/{task_ID}*:

```
GET URL_for_your_Integration_Service _system/task_id
```

   where:

   - *URL_for_your_Integration_Service _system* is the IP address or URL for the Integration Service system.
   - *task_id* is the id value you can copy from the log file for the integration application. The task ID specifies the latest execution of the application.

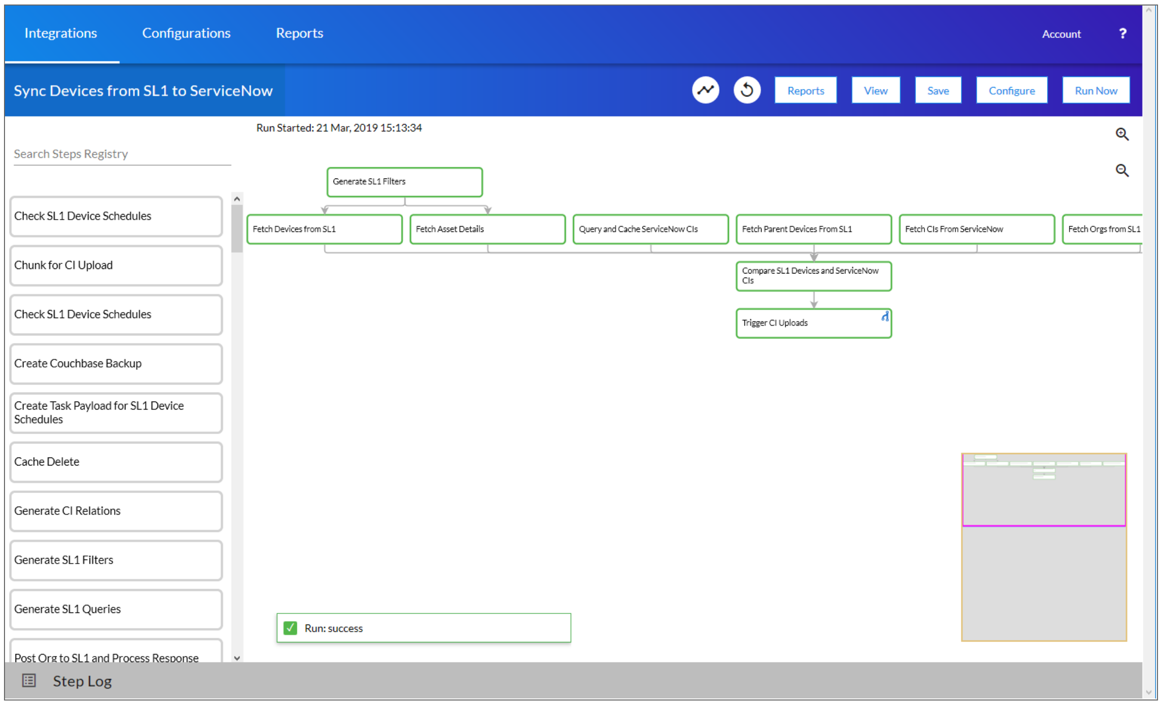
Working with Integration Applications in the Integration Service User Interface

You can log into the user interface for the Integration Service to edit, configure, and run an integration application.

Editing an Integration Application in the User Interface

To work with an integration application:

1. From the **Integration Application** page, click the **[Edit]** button. The **Search Steps Registry** pane appears, with a list of all of the steps that are available for that integration:



2. Scroll through the **Search Steps Registry** pane or use the **Search** field at the top of the pane to find the step you want to add.
3. Click the step you want to add, drag it to the main pane of the **Integration Application** page, and drop it into the integration application .
4. To adjust the position of any step in the integration application, click the step you want to move and drag it to its new location.
5. To redirect the arrows connecting the steps, click an arrow and drag it to reposition it.
6. To remove a step, click the step to select it and press the **[Delete]** key on your keyboard.
7. To save the changes you made to the integration application, click the **[Save]** button.
8. To stop editing and close the **Search Steps Registry** panel, click the **[View]** button.

TIP: If the main pane has too many steps to see without scrolling, you can zoom in or out by clicking and holding the wheel on your mouse. You can also use the pane in the bottom right-hand corner to click on a part of the integration application that you want to see, and it will move the screen to focus on that part.

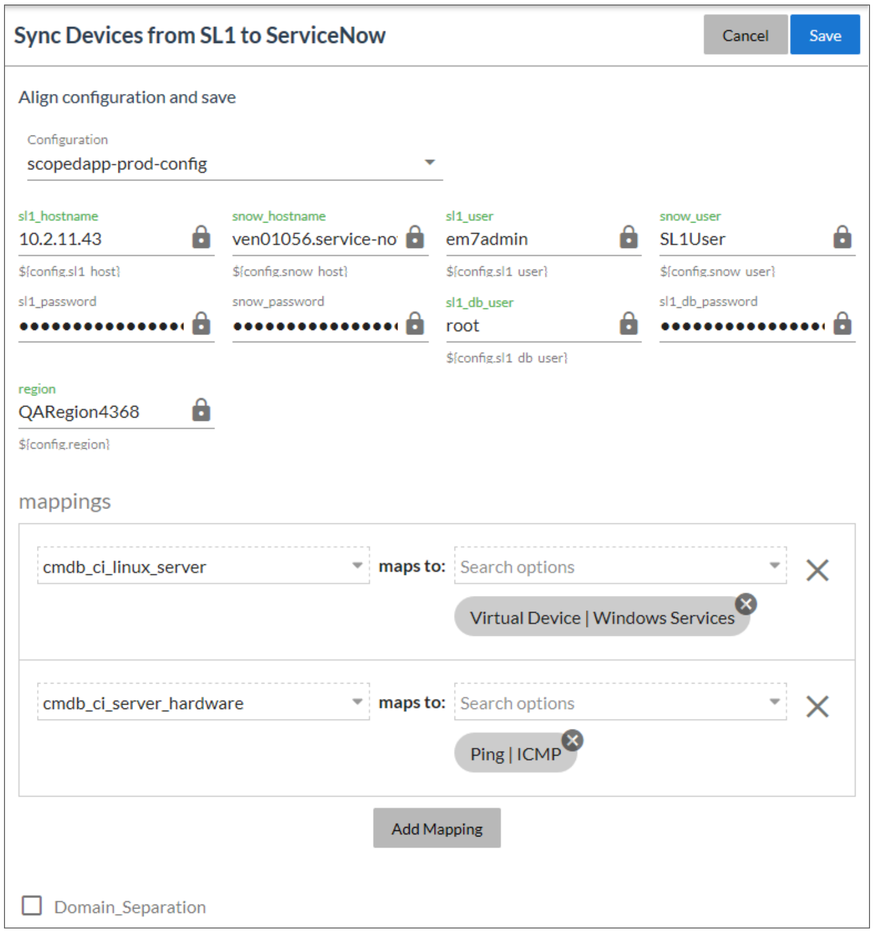
Configuring an Integration Application in the User Interface

Before you can run an integration application, you must align the application with a configuration from the **[Configurations]** tab. A ***configuration*** defines global variables, such as endpoints and credentials, that can be used by multiple steps and integration applications.

You can "align" the configuration you want to use with an integration application from the **Configuration** pane for that application integration. Where relevant, you can also edit the sections for the **additional_attributes** and **mappings** parameters to update or add new mappings between SL1 and another application.

To configure an integration application:

1. On the **[Integrations]** tab, select the integration application you want to configure.
2. On the **Integration Application Editor** page, click the **[Configure]** button. The **Configuration** pane opens on the right side of the window:



Sync Devices from SL1 to ServiceNow

Align configuration and save

Configuration: scopedapp-prod-config

sl1_hostname: 10.2.11.43 ${config.sl1 host}	snow_hostname: ven01056.service-no ${config.snow host}	sl1_user: em7admin ${config.sl1 user}	snow_user: SL1User ${config.snow user}
sl1_password: ●●●●●●●●●●●●●●●●	snow_password: ●●●●●●●●●●●●●●●●	sl1_db_user: root ${config.sl1 db user}	sl1_db_password: ●●●●●●●●●●●●●●●●
region: QARegion4368 ${config.region}			

mappings

cmdb_ci_linux_server maps to: Search options — Virtual Device | Windows Services

cmdb_ci_server_hardware maps to: Search options — Ping | ICMP

Domain_Separation

3. Select a configuration from the **Configuration** drop-down list to "align" to this integration application. This step is required.

4. To update the device attribute mappings for this integration application, scroll down to the section for the **additional_attributes** parameter and click the **[Add Mapping]** button to add a custom attribute, or edit an existing attribute that you want to map between SL1 and another application. Press **[Enter]** after editing each attribute mapping to make sure your changes are saved.

TIP: Use the **[Tab]** button to move down through the list of options in a **Mapping** dropdown list, and press **[Shift]**+**[Tab]** to move up.

5. To update the device class and asset mappings, scroll down to the section for the **mappings** parameter and click the **[Add Mapping]** button to add a custom class or asset, or edit an existing class or asset that you want to map between SL1 and another application. Press **[Enter]** after editing an item to make sure your changes are saved.
6. As needed, edit the other configuration values for the application. Press **[Enter]** after editing an item to make sure your changes are saved.

NOTE: To prevent potential issues with security and configuration, the fields related to configuration and any fields that are encrypted on the **Configuration** pane for an integration application cannot be edited.

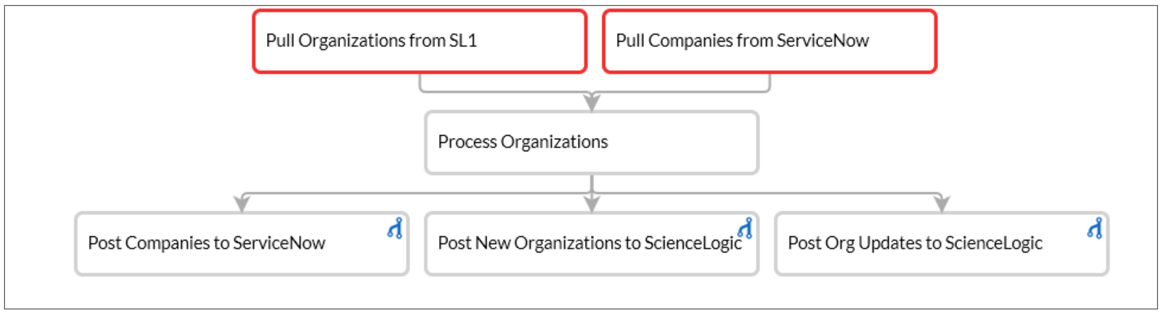
7. When you are finished, click the **[Save]** button. If needed, click anywhere off of the **Configuration** pane to close the pane.

NOTE: If you have created a new configuration in your own text editor, you must upload it to your Integration Service instance using the command line tool. After you upload it to the instance, it appears in the **Configuration** drop-down field on the **Configuration** pane.

Running or Stopping an Integration Application in the User Interface

To run an integration application:

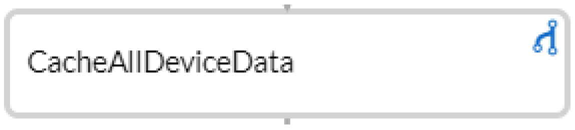
1. Click the **[Run Now]** button from the **Integrations** window or the individual application's window.
2. As the application runs, the color of the border around each step represents whether it is running, successful, or has failed:


Pull Organizations from SL1
Pull Companies from ServiceNow
Process Organizations
Post Companies to ServiceNow
Post New Organizations to ScienceLogic
Post Org Updates to ScienceLogic

Step Color	State
Blue	Running
Green	Successful
Red	Failed

NOTE: Pop-up status messages also appear in the bottom left-hand corner of the **Integration Application Editor** page to update you on the progress of the application status and alert you of any errors.

3. If a step triggers a child application, a branch icon ( ) appears in the upper right-hand corner of the step:



4. Double-click the branch icon to open the child application. Click the branch icon once to display the triggered application's run ID as a link in a pop-up window. If no run ID is present, the branch icon displays "NONE".
5. To stop the integration while it is running, click the **[Stop Run]** button. The **[Stop Run]** button is the same as the **[Run Now]** button, but toggles to **[Stop Run]** when you run the integration.

Chapter 4

ipaascore.BaseStep class

Overview

This chapter describes functions included in the ipaascore.BaseStep class. You can use these functions to define the logic in a step.

This chapter covers the following topics:

get_app_variable

Description

Retrieve the value of an application variable.

Syntax

get_app_variable*(name)*

Parameters

variable_name. The name of the application variable that you want to retrieve the value for.

Return

The value of the application variable.

Example

Suppose we defined this application variable in the integration application:

```
 "app_variables": [
   {
     "name": "sl1_hostname",
     "description": "The SL1 hostname to participate in the sync",
     "sample_value": "10.2.253.115",
     "default_value": null,
     "required": true,
     "value": 10.64.68.25
   },
]
```

Suppose this integration application calls the step "sync_SL1_data".

In the step "sync_SL1_data", we could use the following function to resolve the value of "sl1hostname":

```
hostname = self.get_app_variable("sl1_hostname")
```

The value of ***hostname*** would be "10.64.68.25".

get_available_previous_step_input_positions

Description

Retrieves the list of steps in the integration application along with the position of the step (the order that the step was run in the integration application. Position "0" (zero) is reserved for the current step's arguments.

Syntax

```
get_available_previous_step_input_positions()
```

Return

Returns a list of tuples. Each tuple includes a step name and the step's position.

get_data_from_step_by_name

Description

Retrieves data saved from a previous step.

NOTE: To retrieve data from a previous step, that previous step must save the data use the ***save_data_for_next_step*** function and the integration application must specify that the data from the previous step should be passed to the current step using the ***output_to*** parameter.

NOTE: Although the ***get_data_from_step_by_name*** function is simple to use, it does not allow you to write a generic, reusable step, because the step name will be hardcoded in the function. The functions ***join_previous_step_data*** or **get_data_from_step_by_order** allow you to create a more generic, reusable step.

Syntax

get_data_from_step_by_name*(step_name)*

Parameters

step_name. The name of a previous step in the integration application.

Return

The data that was saved by the previous step.

Example

The following is an example of the ***get_data_from_step_by_name*** function:

```
em7_data = self.get_data_from_step_by_name('FetchDevicesFromEM7')
snow_data = self.get_data_from_step_by_name('FetchDevicesFromSnow')
```

get_data_from_step_by_order

Description

This function retrieves data from a step based on the position of the step in the integration application.

NOTE: To retrieve data from a previous step, that previous step must save the data use the ***save_data_for_next_step*** function and the integration application must specify that the data from the previous step should be passed to the current step using the ***output_to*** parameter.

Syntax

get_data_from_step_by_order(*pos)*

Parameters

position. The position of the step (order that the step was run) in the integration application. Position values start at "0" (zero) .

Return

The data that was saved by the previous step.

Exception

DataNotAvailableException

Example

- Suppose your integration application has four steps: stepA, stepB, stepC, and stepD
- Suppose stepA was run first (position "0") and includes the parameter "output_to":["stepD"]
- Suppose stepB was run second (position "1") and includes the parameter "output_to":["stepD"]
- Suppose stepC was run third (position "2") and includes the parameter "output_to":["stepD"]
- Suppose stepD was run fourth
- If the current step is stepD, and stepD needs the data from stepC, you could use the following:

```
data_from_stepC = self.get_data_from_step_by_order(2)
```

get_name

Description

Returns the name of the current step.

Syntax

```
get_name()
```

Return

The name of the current step.

get_parameter

Description

Retrieves a parameter value.

Syntax

get_parameter(*param_name, lookup_data=None)*

Parameters

param_name. The name of the parameter that you want to retrieve the value for.

lookup_data. An optional dictionary that can provide a reference for additional variable substitutions.

Return

Value of the requested parameter.

Example

For example, suppose we defined this parameter in the step named "GETgoogle.com":

```
self.new_step_parameter(name=prefix_url, description="used with relative_url to create
the full URL.",sample_value="http://10.2.11.253", default_value=None, required=True)
```

Suppose in the integration application that calls "GETgoogle.com", we specified:

```
"steps": [
  {
    "file": "QueryREST",
    "method": "GET",
    "name": "GETgoogle.com",
    "output_to": ["next_step"],
    "prefix_url": "http://google.com"
  }
],
```

Suppose we use the ***get_parameter*** function in the step "GETgoogle.com" to retrieve the value of the "prefix_url" parameter:

```
build_url_1 = self.get_parameter("prefix_url")
```

The value of ***build_url_1*** would be "http://google.com".

get_parameter_from_previous_step

Description

Retrieves a parameter value from a previous step.

Syntax

get_parameter_from_previous_step*(parameter_name, step_name)*

Parameters

param_name. The name of the parameter that you want to retrieve the value for.

step_name. the step from which you want to retrieve a parameter value.

Return

The value of the parameter.

join_previous_step_data

Description

This function retrieves data from one or more previous steps in the integration application.

NOTE: To retrieve data from a previous step, that previous step must save the data use the ***save_data_for_next_step*** function and the integration application must specify that the data from the previous step should be passed to the current step using the ***output_to*** parameter.

If you are expecting similar data from multiple steps, or expecting data from only a single step, the ***join_previous_step_data*** function is the best choice.

The ***join_previous_step_data*** function gathers all data from all steps that included the **save_data_for_next_step** function and also include the ***output_to*** parameter in the integration application. By default, this function returns the joined set of all data that is passed to the current step. You can also specify a list of previous steps from which to join data.

The retrieved data must be of the same type. The data is then combined into a list or a dictionary. If the data types are not the same, then the function will raise an exception

Syntax

join_previous_step_data(*data_from)*

Parameters

data_from. An optional argument that specifies the steps. For example, if you wanted to join only the data from stepA and stepD, you could specify

```
self.join_previous_step_data([“stepA”, “stepD”]),
```

Return

Combined data structure, either a list or a dictionary, of all the retrieved step data.

4

Example

The following is an example of the ***join_previous_step_data*** function in the QueryREST step:

```
def query_with_url_generated_from_input(self):
"""
Iterates over data from previous steps and generates a relative url for each. Then
executes that command
:return:
"""
count = 0
input_data = self.join_previous_step_data()
payload = self.get_parameter(PAYLOAD)
if type(input_data) is list:
  for input_d in input_data:
    relative_url = self.get_parameter(RELATIVE_URL, input_d)
    self.process_REST_command(payload, relative_url)
    count += 1
elif type(input_data) is dict:
  relative_url = self.get_parameter(RELATIVE_URL, input_data)
  self.process_REST_command(payload, relative_url)
  count += 1
else:
  raise NotImplementedError("Data type: {} is not currently supported for generating
  relative urls form data".format(type(input_data)))
```

new_step_parameter

Description

Define a parameter for the step. The integration application will provide the parameter values during runtime. If a parameter is defined as required, the step will fail with an exception if the parameter is not provided.

Syntax

new_step_parameter(*name*='', *description*='', *sample_value*='', *default_value*=None, *required*=False)

Parameters

name. The name of the parameter. This value will be used to create a name:value tuple in the integration application file (in JSON).

description. A description of the step parameter.

sample_value. A sample value of the required data type or schema.

default_value. If no value is specified for this parameter, use the default value. Can be any combination of alphanumeric characters. To prevent a default value, specify "None".

required. Specifies whether this parameter is required by the step. The possible values are "True" or "False".

Example

Here is an example from the QueryREST step:

```
self.new_step_parameter(name=PREFIX_URL, description="used with relative_url to create
the full URL.", sample_value="http://10.2.11.253", default_value=None, required=True)
```

save_data_for_next_step

Description

This function saves an object (usually a variable) and makes the data available to another step. The object must be of a data type that can be pickled by Python.

Syntax

save_data_for_next_step(*data_to_save)*

Parameters

data_to_save. A variable that contains the data.

NOTE: The *data_to_save* object must be of a data type that can be pickled by Python: None, True and False, integers, long integers, floating point numbers, complex numbers, normal strings, unicode strings, tuples, lists, set, and dictionaries.

Example

The following is an example of the ***save_data_for_next_step*** function:

```
save_data = {'key': 'value'}
self.save_data_for_next_step(save_data)
```

The integration application must then specify that the data from the current step should be passed to one or more subsequent step, using the **output_to** parameter. For details, see the section about ***using integration applications to transfer data between steps***.

validate_parameter_values

Description

Validates the parameter values provided for an integration application. For example, the validate_parameter_ value function will raise an error if the user failed to provide a required parameter.

Syntax

```
validate_parameter_values()
```

Chapter 5

Customizing the Execution Environments

Overview

Users can create their own custom content on the Integration Service system, such as their own Python wheels, libraries, or other source files. This chapter explains how you can customize the Python execution environment for the Integration Service.

This chapter covers the following topics:

Creating Dockerfiles

The Integration Service system is composed of microservices running in Docker containers. Users can update the container environments to include additional code, environments, and files.

To manually supplement the Integration Service system, you must update two containers:

- ***sciencelogic/is-worker***. This container reads step tasks from the queue and executes them.
- ***sciencelogic/is-api***. This container accepts requests and dispatches tasks for execution.

These two containers instantiate steps and applications, therefore the containers must be informed of any additional libraries that are called from your custom steps. You must make all changes to both containers.

NOTE: To customize the Python execution environments, you must have a working knowledge of Docker containers. An excellent resource for getting started with Docker is available from Docker: https://docs.docker.com/get-started/.

To update the existing containers, you must create two Dockerfiles.

Each new Dockerfile includes the existing ScienceLogic container and any additional layers you want to add.

Dockerfile for is-worker

To customize a Dockerfile for the is-worker container:

1. Either go to the console of the Integration Service system or use SSH to access the server.
2. Log in as ***isadmin*** with the appropriate password.
3. View the file **/opt/iservices/scripts/docker-compose.yml**.
4. Search for the line that contains "steprunner". Find the line below it that begins with "image". Copy the image tag. It will look something like this:

```
steprunner
  image: sciencelogic/is-worker:latest
```

6. Create a new empty directory.
7. In that new directory, create a file called "Dockerfile_is-worker":

```
vi Dockerfile_is-worker
```

8. The first line of the Dockerfile must be:

```
From image-id
```

where:

image-id is the tag of the image that you identified in step 4. This is similar to an "include"; everything from the original image will be included in the new image.

An example line would be:

```
FROM sciencelogic/is-worker:latest
```

9. Add any additional Docker run commands after the FROM line. For example:

```
RUN pip install pymysql
```

10. Save the new Dockerfile.

Dockerfile for is-api

To customize a Dockerfile for the is-api container:

1. Either go to the console of the Integration Service system or use SSH to access the server.
2. Log in as ***isadmin*** with the appropriate password.
3. View the file **/opt/iservices/scripts/docker-compose.yml**.
4. Search for the line that contains "steprunner". Find the line below it that begins with "image". Copy the image tag. It will look something like this:

```
contentapi
  image: sciencelogic/is-api:1.6.0
```

6. Go to the directory that contains Dockerfile_is-worker. In that new directory, create a file called "Dockerfile_is-api":

   ```
   vi Dockerfile_is-api
   ```

7. The first line of the Dockerfile must be:

   ```
   FROM image-id
   ```

   where:

   image-id is the tag of the image that you identified in the earlier step. This is similar to an "include"; everything from the original image will be included in the new image.

   An example line would be:

   ```
   FROM sciencelogic/is-api:1.6.0
   ```

8. Add any additional Docker run commands after the FROM line. For example:

   ```
   RUN pip install pymysql
   ```

9. Save the new Dockerfile.

Building the Docker Images

After you have created the custom Dockerfiles, you can build the Docker images.

To build the Docker image:

1. Either go to the console of the Integration Service system or use SSH to access the server.
2. Log in as ***isadmin*** with the appropriate password.
3. Navigate to the directory that contains the custom Dockerfiles.
4. To build Dockerfile_is-api, run the following command:
   ```
   docker build -f Dockerfile_is-api
   ```
5. If the build is successful, Docker will create a new image and assign it a unique image tag. You can edit the image tag to be a more readable name, for example id-api:1.6.0.
6. To build Dockerfile_is-worker, run the following command:
   ```
   docker build -f Dockerfile_is-worker
   ```
7. If the build is successful, Docker will create a new image and assign it a unique image tag. You can edit the image tag to be a more readable name, for example, is-worker:1.6.0.
8. Ensure you have a new docker image tag for both the is-worker container and the is-api container before continuing.

NOTE: Refer to the Docker documentation for more information on building containers.

Using Your New Image in the Integration Service System

After you have created a new image and generated the image tags, you can install the image on the Integration Service system.

To replace the default images with your new images:

1. Either go to the console of the Integration Service system or use SSH to access the server.
2. Log in as ***isadmin*** with the appropriate password.
3. Using a text editor like "vi", edit the file **/opt/iservices/scripts/docker-compose.yml**.
4. Search for the line that contains "steprunner". Find the line below it that begins with "image. Edit the value of the image tag to match the new tag you generated in the section ***Building the Docker Image***.

```
steprunner:
  image: sciencelogic/is-worker:1.6.0
```

5. Search for the line that contains "contentapi". Find the line below it that begins with "image". Edit the value of the image tag to match the new tag you generated in the section ***Building the Docker Image***.

```
contentapi:
  image: sciencelogic/is-api:1.6.0
```

5. Save your changes (:wq).
6. After you have updated the docker-compose file, you can update and re-deploy the Integration Service system with your new images. To do this, run the following command:

```
docker stack deploy -c /opt/iservices/scripts/docker-compose.yml is4
```

7. The Integration Service system should now include the customizations added in the images, and steps can now reference those new classes or libraries.

Special Considerations for Multi-Host Clusters

For multi-host clusters, ScienceLogic recommends that you:

1. On one of the servers in the cluster, perform the steps in ***Creating Dockerfiles***.
2. On one of the servers in the cluster, perform the steps in ***Building the Docker Image***.
3. Upload your images to an internal Docker registry. This includes tagging the image with the hostname and port of the registry. For details, see https://docs.docker.com/registry/#alternatives.
4. On each server in the cluster, edit the the ***docker-compose.yml*** file as described in ***Using Your New Image in the Integration Service System***. Replace the entire line after "image:" with the image tag from the registry.
5. On each server in the cluster, re-deploy the Integration Service system with your new images, as described in ***Using Your New Image in the Integration Service System***.

If you do not have a Docker registry, you must perform the following on each host in the Integration Services cluster:

1. On each server in the cluster, perform the steps in *Creating Dockerfiles*.
2. On each server in the cluster, perform the steps in *Building the Docker Image*.
3. Use the same name for the images across the cluster. For example, each server in the cluster would contain the images "sciencelogic/custom_is-worker_11" and "sciencelogic/custom_is-ap_11".
4. On each server in the cluster, edit the ***docker-compose.yml*** file as described in *Using Your New Image in the Integration Service System*.
5. On each server in the cluster, re-deploy the Integration Service system with your new images, as described in *Using Your New Image in the Integration Service System*.

Both approaches will ensure that each host in the cluster references the same image.

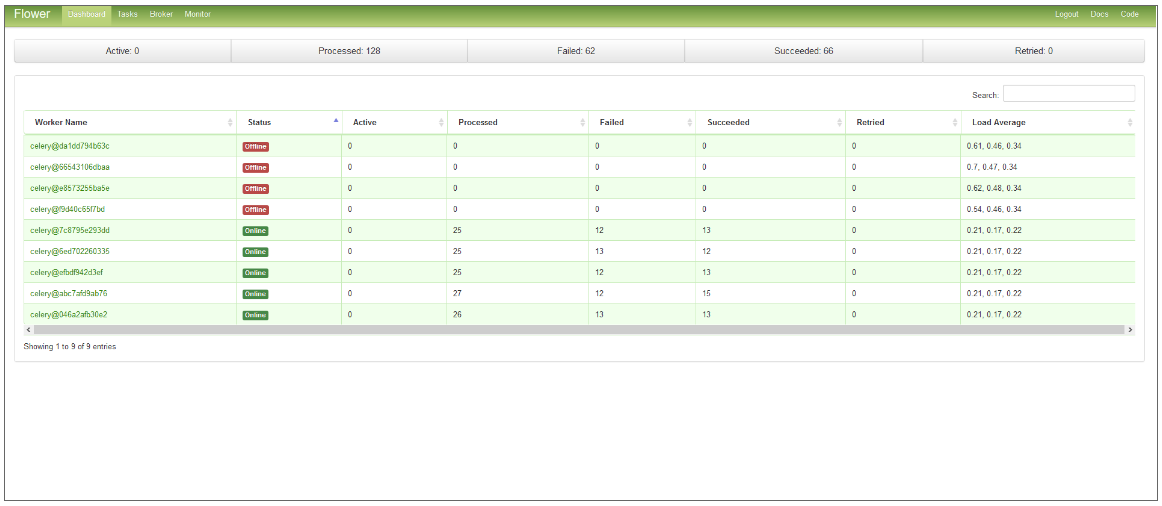
Identifying and Fixing a Deadlocked State

If the Integration Service appears to be running, but the Integration Service is not processing any new integrations or tasks, then the Integration Service could be in a ***deadlocked*** state.

A deadlocked state occurs when one or more integration applications include steps that are either ordered improperly or that contain syntax errors. In this situation, tasks are waiting on subsequent tasks to finish executing, but the worker pool is exhausted. As a result, the Integration Service is not able to execute those subsequent tasks.

To identify a deadlocked state with an Integration Service system:

1. Navigate to the Celery Flower interface for the Integration Service by typing the URL or IP address for your Integration Service and adding **/flower** at the end of the URL, such as **https://192.0.2.0/flower/**.
2. Click the **[Dashboard]** tab for Flower. A list of workers appears:



Active: 0 | Processed: 128 | Failed: 62 | Succeeded: 66 | Retried: 0

Worker Name	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@da1dd794b63c	Offline	0	0	0	0	0	0.61, 0.46, 0.34
celery@66543106dbaa	Offline	0	0	0	0	0	0.7, 0.47, 0.34
celery@e8573255ba5e	Offline	0	0	0	0	0	0.62, 0.48, 0.34
celery@f9d40c65f7bd	Offline	0	0	0	0	0	0.54, 0.46, 0.34
celery@7c8795e293dd	Online	0	25	12	13	0	0.21, 0.17, 0.22
celery@6ed702260335	Online	0	25	13	12	0	0.21, 0.17, 0.22
celery@efbdf942d3ef	Online	0	25	12	13	0	0.21, 0.17, 0.22
celery@abc7afd9ab76	Online	0	27	12	15	0	0.21, 0.17, 0.22
celery@046a2afb30e2	Online	0	26	13	13	0	0.21, 0.17, 0.22

Showing 1 to 9 of 9 entries

3. Review the number of active or running tasks for all workers. If all workers have the maximum number of tasks, and no new tasks are being consumed, then you might have a deadlock state.

5

To fix a deadlocked state in an Integration Service system, perform one of the following steps:

1. Go to the console of the Integration Service system or use SSH to access the server.
2. Log in as ***isadmin*** with the appropriate password.
3. Increase the number of workers by either:

   - Executing the following at the shell prompt:

   ```
   docker service scale iservices_steprunner=x
   ```

   where:

     - *x* is the number of workers.

   - Using a text editor like vi to edit the file **/opt/iservices/scripts/docker-compose.yml**.

     - In the **environment:** section at the top of the file, add the following:

   ```
   worker_threads: number_greater_than_3
   ```

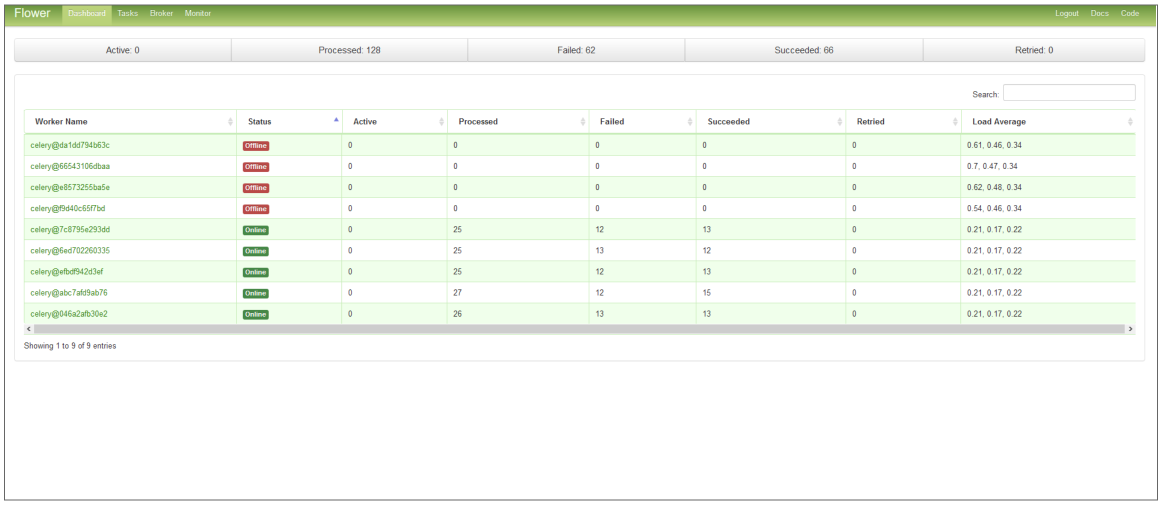
   where:

     - *number_greater_than_3* is an integer greater than 3.

4. After you have updated the docker-compose file, you can update and re-deploy the Integration Service system to pick up the changes in the docker-compose.yml file To do this, execute the following at the shell prompt:

   ```
   docker stack deploy -c /opt/iservices/scripts/docker-compose.yml is4
   ```

5. The Integration Service system should now include additional workers.
6. Navigate to the Celery Flower interface for the Integration Service by typing the URL or IP address for your Integration Service and adding **/flower** at the end of the URL, such as **https://192.0.2.0/flower/**.
7. Click the **[Dashboard]** tab for Flower. A list of workers appears:



Active: 0 | Processed: 128 | Failed: 62 | Succeeded: 66 | Retried: 0

Worker Name	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@da1dd794b63c	Offline	0	0	0	0	0	0.61, 0.46, 0.34
celery@66543106dbaa	Offline	0	0	0	0	0	0.7, 0.47, 0.34
celery@e8573255ba5e	Offline	0	0	0	0	0	0.62, 0.48, 0.34
celery@f9d40c65f7bd	Offline	0	0	0	0	0	0.54, 0.46, 0.34
celery@7c8795e293dd	Online	0	25	12	13	0	0.21, 0.17, 0.22
celery@6ed702260335	Online	0	25	13	12	0	0.21, 0.17, 0.22
celery@efbdf942d3ef	Online	0	25	12	13	0	0.21, 0.17, 0.22
celery@abc7afd9ab76	Online	0	27	12	15	0	0.21, 0.17, 0.22
celery@046a2afb30e2	Online	0	26	13	13	0	0.21, 0.17, 0.22

Showing 1 to 9 of 9 entries

8. Review the number of active or running tasks for all workers.

Optimizing Workers, Queues, and Tasks

The Integration Service system uses Celery to spawn and manage worker processes and queues. You can define environment variables to optimize these worker processes.

You can use the following environment variables to optimize worker processes:

- ***task_soft_time_limit***. Enforces global timeout for all tasks in the Integration Service system. If a task exceeds the specified timeout limit, the Integration Service system terminates the task so that the next task in the queue can be processed. Possible values are:
  - Integer that specifies the time in seconds.
  - Default value is "3600" (1 hour).
- ***optimization***. Determines how tasks are distributed to worker processes. Possible values are:
  - -*Ofair*. Celery distributes a task only to the worker process that is available for work.
  - " " (double-quotation mark, space, double-quotation mark). Distributes and queues all tasks to all available workers. Although this increases performance, tasks in queues might be delayed waiting for long-running tasks to complete.
  - Default value is -*Ofair*.
- ***task_acks_late***. Specifies that if a worker process crashes while executing a task, Celery will redistribute the task to another worker process. Possible values are:
  - *True*. Enables the environment variable.
  - *False*. disabled the environment variable.
  - Default value is "False"

NOTE: Because many integration applications run at regular intervals or are scheduled, the Integration Service system re-executes tasks even if the ***task_acks_late*** environment variable is disabled. in the event of a worker crash, if you want to ensure that tasks are completed, you can enable the ***task_ acks_late*** variable. However, be aware that if tasks are not idempotent, the ***task_acks_late*** variable can cause unpredictable results.

To define these environment variables:

1. Either go to the console of the Integration Service system or use SSH to access the server.
2. Log in as ***isadmin*** with the appropriate password.
3. Use a text editor like vi to edit the file **/opt/iservices/scripts/docker-compose.yml**.

4. You can define the environment variables for one or more worker processes. The docker-compose.yml file contains definitions for worker processes. For example, you might see something like this:

```
services:
  steprunner:
    image: sciencelogic/is-worker:latest
    environment:
    LOGLEVEL: 10
    celery_log_level: 10
    logdir: /var/log/iservices
    broker_url: 'pyamqp://guest@rabbit//'
    result_backend: 'redis://redis:6379/0'
    db_host: 'couchbase,localhost'
    secrets:
      - is_pass
      - encryption_key
    deploy:
    replicas: 2
    networks:
      - isnet
    depends_on:
      - redis
      - rabbitmq
      - couchbase
    volumes:
      - "/var/log/iservices:/var/log/iservices"
      - "statedb:/var/run/celery/states/"

  steprunner_1:
    image: : sciencelogic/is-worker:latest
    environment:
    LOGLEVEL: 10
    celery_log_level: 10
    task_soft_time_limit: 30
    optimization: ""
    task_acks_late: 'False'
    logdir: /var/log/iservices
    broker_url: 'pyamqp://guest@rabbit//'
    result_backend: 'redis://redis:6379/0'

    db_host: 'couchbase,localhost'
    secrets:
      - is_pass
      - encryption_key
    deploy:
    replicas: 2
    networks:
      - isnet
    depends_on:
      - redis
      - rabbitmq
      - couchbase
    volumes:
      - "/var/log/iservices:/var/log/iservices"
      - "statedb:/var/run/celery/states/"

  steprunner_2:
    image: : sciencelogic/is-worker:latest
    environment:
    LOGLEVEL: 10
    celery_log_level: 10
    task_soft_time_limit: 30
    optimization: '-Ofair'
    task_acks_late: 'False'
    logdir: /var/log/iservices
    broker_url: 'pyamqp://guest@rabbit//'
    result_backend: 'redis://redis:6379/0'
    db_host: 'couchbase,localhost'
    secrets:
      - is_pass
      - encryption_key
    deploy:
    replicas: 2
    networks:
      - isnet
    depends_on:
      - redis
      - rabbitmq
      - couchbase
    volumes:
      - "/var/log/iservices:/var/log/iservices"
      - "statedb:/var/run/celery/states/"
```

5. The services with names that start with "steprunner" are the workers for the Integration Service system. To define the optimization variables, enter the variables and values in the definition of the worker, under the ***environment*** section. See the example in step #3 to see the syntax for environment variables.
6. After you have updated the docker-compose file, you can update and re-deploy the Integration Service system to pick up your changes to the file. To do this, execute the following command:

```
docker stack deploy -c /opt/iservices/scripts/docker-compose.yml is4
```

Chapter 6

Tools for the Integration Service

Overview

The Integration Service includes the following tools to help you manage content:

- ***iscli***. The Integration Service includes a command line tool called ***iscli***. When you install Integration Service, ***iscli*** is automatically installed. The iscli allows you to upload integration applications, steps, and configurations.
- ***IS API***. The Integration Service includes an API. When you install the Integration Service, the API is available.

This chapter covers the following topics:

6

iscli

The Integration Service system includes a command line tool called ***iscli***. When you install the Integration Service system, ***iscli*** is automatically installed. The iscli allows you to upload integration applications, steps, and configurations.

To access the iscli tool:

1. Either go to the console of the Integration Service system or use SSH to access the server.
2. Log in as ***isadmin*** with the appropriate password.
3. Type the following at the command line:

   ```
   iscli -h
   ```

Syntax

- To upload a step:

```
iscli -u -s -f path_and_name_of_step_file.py -H hostname_or_IP_address_of_
integration_service_system -P port_number_of_http_on_integration_service_system -U
user_name -p password
```

- To upload an application:

```
iscli -u -a -f path_and_name_of_application_file.json -H hostname_or_IP_address_
of_integration_service_system -P port_number_of_http_on_integration_service_system
-U user_name -p password
```

- To upload a configuration:

```
iscli -u -c -f path_and_name_of_configuration_file.json -H hostname_or_IP_address_
of_integration_service_system -P port_number_of_http_on_integration_service_system
-U user_name -p password
```

List of Arguments

The following is a list of arguments you can use in the iscli tool:

- **-h or --help.** Help message.
- **-H or --host.** Hostname of content store. If left blank, default from config is used.
- **-P or --port.** Port of content store. If left blank, default from config is used.
- **-U or --username.** Username used to access content store. If left blank, default is used.
- **-p or --password.** Password used to access content store. If left blank, default is used.
- **-u or --upload.** Upload data to content store. Requires -f tag and a version to be used.
- **-f or --fileLocation.** Filepath of single file or directory containing python file(s) for new step(s).
- -**c or --configuration.** Use the CLI to interact with a configuration.
- **-a or --application.** Use the CLI to interact with an application.
- **-s or --step.** Use the CLI to interact with a step.

You will use the following commands most often:

- ***Upload an application file***. `iscli –uaf <app-file>`
- ***Upload a configuration file***. `iscli –ucf <config-fil>`
- ***Upload a step file***. `iscli –usf <step-file>`

Example

```
iscli -u -s -f path_and_name_of_step_file.py -H hostname_or_IP_address_of_
integration_service_system -P port_number_of_http_on_integration_service_system -U
user_name -p password
```

where:

- *name_of_step_file* is the full pathname for the step.
- *hostname_or_IP_address_of_integration_service_system* is the hostname or IP address of the Integration Service system.
- *port_number_of_http_on_integration_service_system* is the port number to access the Integration Service system.
- *user_name* is the user name you use to log in to the Integration Service system.
- *password* is password you use to log in to the Integration Service system.

IS API

The Integration Service includes an API. When you install the Integration Service, the API is available.

Viewing API Documentation

To view the full documentation for the IS API:

1. From the Integration Service system, copy the file **/opt/iservices/scripts/swagger.yml** to your local computer.
2. Open a browser session and go to editor.swagger.io.
3. In the Swagger Editor, open the **File** menu, select **Import File**, and import the file swagger.yml. The right pane in the Swagger Editor displays the IS API documentation.

Available Endpoints

POST

/applications. Add a new application or overwrite an existing application.

/applications/{appName}/run. Run a single application by name with saved or provided configurations.

/applications/run. Run a single application by name. For more information, see *Querying for the State of an Integration Application*.

/configurations. Add a new configuration or overwrite an existing configuration.

/steps. Add a new step or overwrite an existing step.

/steps/run. Run a single step by name.

/schedule. Add a new scheduled application integration.

/tasks/{taskId}/replay. Replay a specific integration application. Replayed integration applications run with the same application variables, configuration, and queue as the originally executed application.

/tasks/{taskId}/revoke. Revoke or terminate a specific task or integration application. If an application ID is provided, all tasks associated with that integration application are be revoked.

Querying for the State of an Integration Application

When triggering an integration application from the **applications/run** endpoint, you can query for the state of that integration application in two ways:

1. **Asynchronously**. When you POST a run of an integration application to **/applications/run**, the response is a integration status with a Task ID, such as: *isap-23233-df2f24-etc*. At any time, you can query for the current state of that task from the endpoint **/api/v1/tasks/isap-23233-df2f24-etc**. The response includes all of the steps run by the integration application, along with the status of the steps, and URL links to additional info, such as logs for each step.
2. **Synchronously**. When you POST a run of an integration application, you can tell the Integration Service to wait responding until the integration application is complete by adding the **wait** argument. For example, **/api/v1/applications/run?wait=20** will wait for 20 seconds before responding. The maximum wait time is 30 seconds. When the integration application completes, or 30 seconds has passed, the API returns the current status of the integration run. This process works the same as if you had manually queried **/api/v1/tasks/isapp-w2ef2f2f**. Please note that while the API is waiting for your integration application to complete, you are holding on to a thread. If you have multiple integration applications that run for a long period of time, do not use a synchronous query unless you have no other option. ScienceLogic recommends using an *asynchronous* query whenever possible.

GET

/applications. Retrieve a list of all available applications.

/applications/{appName}. Retrieve a specific application.

/applications/{appName}/logs. Retrieve the logs for the specified application.

/cache/{cache_id}. Retrieve a specific cache.

/configurations. Retrieve a list of all configurations available.

/configurations/{configName}. Retrieve a specific configuration.

/reports. Retrieve a list of paginated reports.

/reports/{reportId}. Retrieve a specific report by ID.

/schedule. Retrieve a list of all scheduled application integrations.

/steps. Retrieve a list of all steps.

/steps/{stepName}. Retrieve a specific step.

/tasks/{taskId}. Retrieve a specific task.

REST

/tasks. Terminate all running tasks.

/tasks/{taskId}. Terminate a specific running task.

DELETE

/applications/{appName}. Delete an integration application by name.

/cache/{cache_id}. Delete a cache entry by name.

/configurations/{configName}. Delete a configuration by name.

/schedule. Delete a scheduled application integration by ID.

/reports/{appName}. Delete a specific report by name.

/reports/{reportId}. Delete a specific report by report ID.

/steps/{stepName}. Delete a specific step by name.