ScienceLogic

SL1 PowerFlow for Developers

Version 2.6.0, rev 3

Table of Contents

Chapter

1

Introduction to SL1 PowerFlow for Developers

Overview

This manual describes how you can use the tools included in SL1 PowerFlow and your own tools to create custom PowerFlow applications, steps, and configuration objects that you can use with PowerFlow.

This chapter covers the following topics:

What is SL1 PowerFlow?

SL1 PowerFlow enables intelligent, bi-directional integration between SL1 and third-party applications to promote a unified management ecosystem. PowerFlow contains default workflows that let users translate and share data between SL1 and third-party applications, and it also allows the development of standardized, reusable snippets called "steps" that non-developers can use to create integration workflows without writing code. In addition, PowerFlow is designed to provide high availability and scalability.

The following image shows an example of a PowerFlow application workflow and its steps in the PowerFlow user interface:
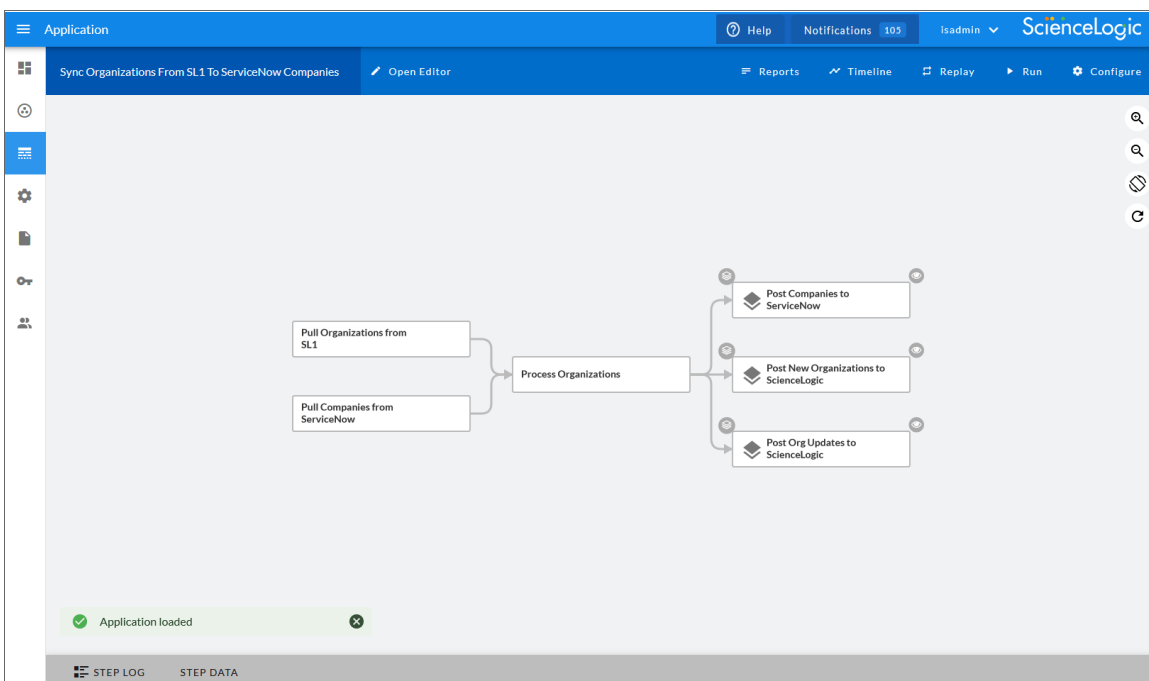
Tools Included with PowerFlow

PowerFlow includes the following tools for creating custom integrations:

- **iscli**. PowerFlow includes a command-line tool called **iscli** (PowerFlow Command Line Interface). When you install PowerFlow, **iscli** is automatically installed. The iscli allows you to upload PowerFlow applications, steps, and configuration objects. For more information, see the section on *iscli*.

- **IS API**. PowerFlow includes an API. When you install PowerFlow, the API is available. For more information, see the section on *the API*.

- **ipaascore.BaseStep class**. This Python class is included with PowerFlow, and it contains 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*.

- **Base Steps SyncPack**. The *Base Steps* SyncPack includes a number of basic steps that perform common API calls. For more information, see *Default Steps*.

In addition, you can contact your ScienceLogic Customer Success Manager (CSM) to get access to the following tool, which is not included in a PowerFlow system:

- **PowerFlow SyncPack Cookiecutter**. Provides a template that you can use to create SyncPacks in a repeatable manner. For more information see *Generating the SyncPack Structure*.

Prerequisites for Creating PowerFlow Steps and Applications

To create your own PowerFlow steps and applications, you must:

- Deploy a PowerFlow system and ensure it is accessible. For details, see the **SL1 PowerFlow Platform** manual.

- Have SSH or console access to the PowerFlow system, so you can use the **iscli** (PowerFlow Command Line Interface). You will use the iscli to upload steps, applications, and configurations to PowerFlow.

- 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 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 one or more steps that you can then upload and use in your PowerFlow system.

After you create steps, you can use them with other steps in one or more PowerFlow *applications*.

> **TIP:** You can also use the PowerFlow user interface to perform many of the actions in this chapter. For more information, see *Managing SL1 PowerFlow Applications*.

> **NOTE:** All Python step code should be Python 3.7 or later.

This chapter covers the following topics:

What is a Step?

In PowerFlow, a **step** is a generic Python class that performs a single action, such as gathering data about an organization.

Steps can accept zero or many input parameters or data from previous steps, and steps can specify output to be used by other steps. The input parameters are configurable variables and values used during execution.

You can use existing steps to create your own workflows, and you can re-use steps in more than one workflow. When these steps are combined in an application, they provide a workflow that satisfies a business requirement. All Python step code should be Python 3.7 or later.

The following image shows a step from the PowerFlow user interface:
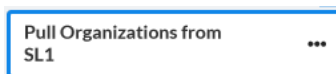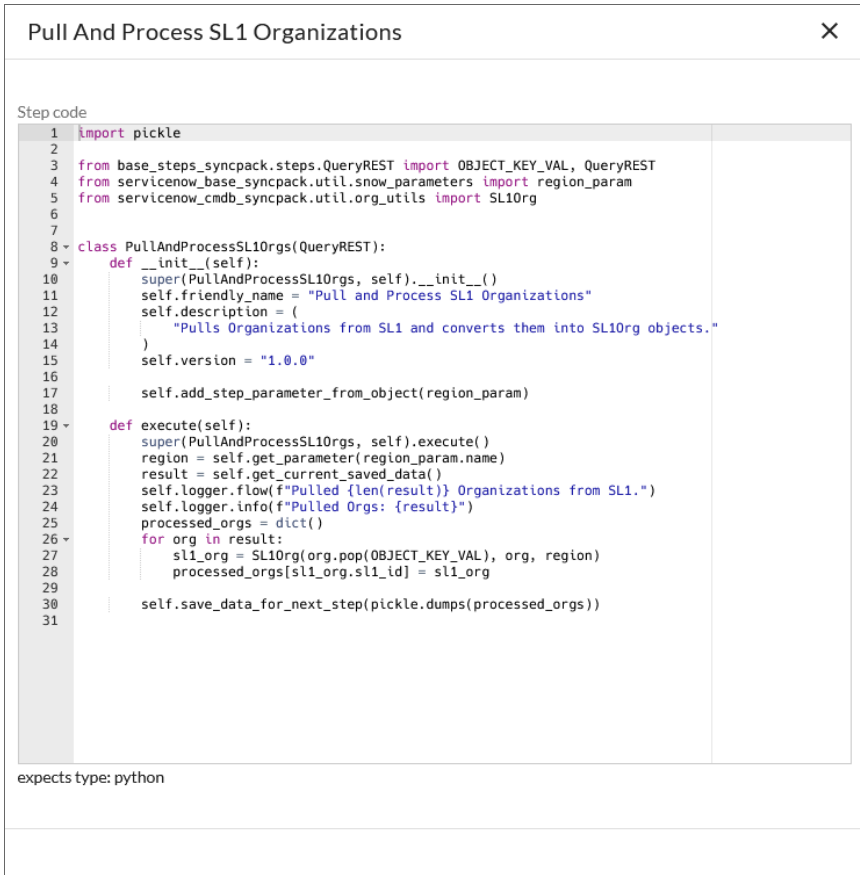
Using Steps in a PowerFlow Application

You can create new steps or use existing steps to create your own workflows, and you can re-use steps in more than one workflow. When these steps are combined as part of a PowerFlow application, they provide a workflow that satisfies a business requirement.

For example, the set of steps below in the "Sync Organizations from SL1 to ServiceNow Companies" application in the PowerFlow user interface gathers data about SL1 organizations and ServiceNow companies, processes that data based on the configuration settings specified for that set of steps, and posts that data to SL1 and ServiceNow to keep the organization and company data in sync in both places:

What is a Step?
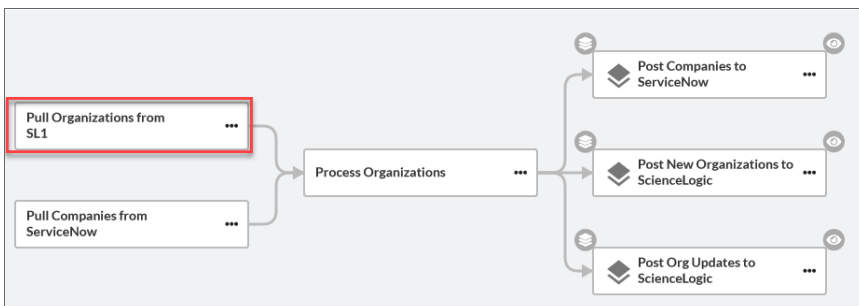
In the PowerFlow builder user interface, if you click the ellipsis icon (**⋯**) on a step, you can select *View step code* to view the Python code for that step:

Using Input Parameters to Configure a Step

You can configure how a step works by adjusting a set of arguments called ***input parameters***. The parameters 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 step to query both the local system and another remote system; only the arguments, such as hostname, username, and password change.

To view and edit the input parameters for a step in the PowerFlow builder:

1. Go to the **Applications** page of the PowerFlow user interface and click the name of a PowerFlow application.
2. Click the **[Open Editor]** button.

3. Click the ellipsis icon (⋯) on the step and select *Configure*. The **Configuration** pane for that step appears:
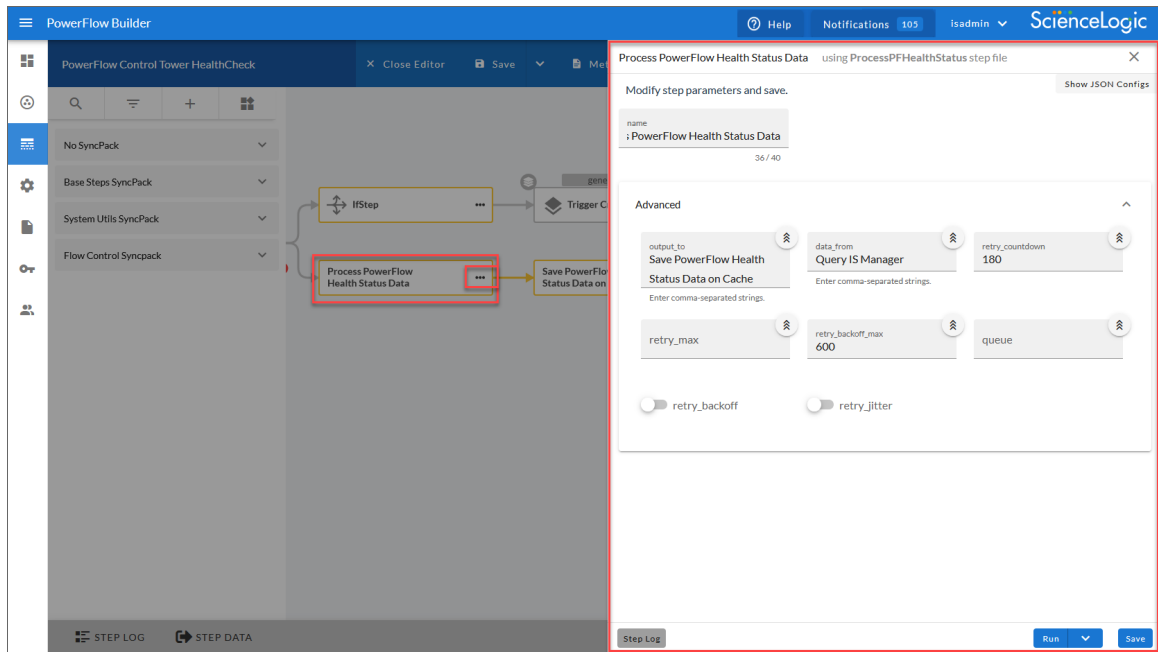
Sharing Data Between Steps

A step can pass the data it generates during execution to a subsequent step. A step can use the data generated by another step. Also, you can run test data for that step by hovering over the **[Run]** button and selecting *Custom Run*.

PowerFlow analyzes the required parameters for each step and alerts you if any required parameters are missing before running the step.

Types of Steps

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 PowerFlow 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 PowerFlow applications. These steps can be configured to be blocking or not (in other words, if the step is set to be blocking and it fails to trigger the application, the application will fail).

A variety of generic steps are available from ScienceLogic, and you can access a list of steps by sending a GET request using the *API /steps* endpoint.

Workflow for Creating a Step

To create a custom step without using the PowerFlow user interface:

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 PowerFlow*.
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 PowerFlow. 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_PowerFlow>/api/v1/steps/stepTemplate
   ```

   where `<URL_for_PowerFlow>` is the IP address or URL for PowerFlow.

   For example:

   ```
   https://10.1.1.111/api/v1/steps/stepTemplate
   ```

2. Select and copy all the text from the **"data"** field 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 PowerFlow system and cannot contain spaces. Note that the step name will also be the name of the Python class for the step.

Example Code: stepTemplate

You can also copy the following code from the **stepTemplate.py** file and paste it into a source-code editor to create your own template file:

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


"""
Name of the class must match the name of the file (without the .py)
"""


DESCRIPTION = "A brief description of what this step will do. This will be
visibile from the GUI"


class stepTemplate(BaseStep):

    def __init__(self):
        # Define step parameters here that are expected as input for this
step. The values for the step parameters will
        # be defined as json data when making requests to execute this
step.
        #
        # Any parameter values set in the step execution json will be
retrievable at runtime from the class.
        #
        # If a param is defined as required, the parser will
        # throw an exception if a value is not set. Add as many parameters
as you like without repeating them
        self.new_step_parameter(name="parameter_name", descrip-
tion="parameter description", sample_value="sample value",
                                default_value=None, required=False)


    def execute(self):
        """
        All logic main logic for executing the step happens here
        :return:
        """

        # You can retrieve parameter value set in the json file (defined
in init) like this:
        step_param_value = self.get_parameter("parameter_name")

        # You can retrieve application wide variable values like this:
```

```
            app_variable_value = self.get_app_variable("app-var-name")


            # You can log with the built in python logging levels like this.
All logging for this step will be written to
            # stdout and its own file
            self.logger.info("Loaded parameter value: {val} for parameter:
{name}".format(val=step_param_value,

                name="parameter_name"))


            # You can retrieve data saved from a previous step by name like
this:
            other_step_data = self.get_data_from_step_by_name('previous_step_
name')


            # If you just want to get all data from all previous steps out-
puting to this one at once, you can
            # use the data merge helper:
            all_step_data_combined = self.join_previous_step_data()


            # Perform whatever logic you want here


            # In the event of a failure, you can raise any exception to exit.
We recommend using the StepFailedException but
            # this is not required
            # raise ipaas_exceptions.StepFailedException("Error occurred")


            # You can save data generated from this step which will auto-
matically be available in subsequent steps like so:
            save_data = {'key': 'value'}
            self.save_data_for_next_step(save_data)
```

Including the Subclass and Required Methods

To execute successfully on PowerFlow, your step must be a subclass of the *ipaascore.BaseStep* class. The **ipaascore.BaseStep** class is a Python class that is included with PowerFlow, and it contains multiple predefined functions that you can use when you are writing or editing a step.

Your new step must include the **init** method and the **execute** method. These methods are explained in the *Required Methods* section, below.

Subclass

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

1. Use a source-code editor to open the new **.py** file for editing.

2. Notice that the file includes these lines of text:

    ```
    from ipaascore.BaseStep import BaseStep
    ```

    ```
    from ipaascommon import ipaas_exceptions
    ```

    Do not remove or alter these lines of text.

3. Search for the following:

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

4. Replace `stepTemplate` with the new name of the file (without the **.py** suffix).

5. Save and close the file.

Required Methods

To execute successfully on PowerFlow, your step must contain at least these two methods:

*   ***init method***. This method lets you define initialization options and parameters for the step.
*   ***execute method***. This method includes the logic for the step and performs the action. After PowerFlow evaluates all parameters and initialization settings and aligns the step with a worker process, PowerFlow examines the **execute** method.

Without these methods, PowerFlow 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:

Example Code: Subclass and Required Methods

For example, the "GetREST" step from the "Base Steps" SyncPack contains the following code (the **init** and **execute** methods are bolded in the code below):

```
from ipaascommon.ipaas_exceptions import StepFailedException
from ipaascommon.ipaas_utils import str_to_bool

from base_steps_syncpack.steps.HTTPBaseStep import HTTPBaseStep
from base_steps_syncpack.util.request_params import (
    chunk_size_param,
    chunk_name_for_pos_in_url_param,
    chunk_name_for_max_in_url_param,
    chunk_name_for_total_param,
    chunk_name_for_returned_param,
    missing_return_total_param,
)


class GetREST(HTTPBaseStep):
```

```
    def __init__(self):
        super(GetREST, self).__init__()
        self.friendly_name = "GetREST"
        self.description = "Step facilitates REST GET interactions and
will return the returned data dictionary and specified headers as data to
the next step"
        self.version = "1.0.0"


        self.add_step_parameter_from_object(chunk_size_param)
        self.add_step_parameter_from_object(chunk_name_for_pos_in_url_
param)
        self.add_step_parameter_from_object(chunk_name_for_max_in_url_
param)
        self.add_step_parameter_from_object(chunk_name_for_total_param)
        self.add_step_parameter_from_object(chunk_name_for_returned_param)
        self.add_step_parameter_from_object(missing_return_total_param)


        self.enable_paging = True
        self.method = "GET"


    def execute(self):
        if not self.chunk_size:
            self.chunk_size = int(self.get_parameter(chunk_size_para-
m.name))
        self.chunk_name_for_pos_in_url = self.get_parameter(
            chunk_name_for_pos_in_url_param.name
        )
        self.chunk_name_for_max_in_url = self.get_parameter(
            chunk_name_for_max_in_url_param.name
        )
        self.chunk_name_for_total = self.get_parameter(chunk_name_for_
total_param.name)
        self.chunk_name_for_returned = self.get_parameter(
            chunk_name_for_returned_param.name
        )
        if self.missing_return_total is None:
            self.missing_return_total = str_to_bool(
                self.get_parameter(missing_return_total_param.name)
            )
```

```
        try:
            self.execute_request()
            response = self.get_current_saved_data()
            self.save_data_for_next_step(response)
        except StepFailedException as err:
            if len(err.args) > 1:
                self.save_data_for_next_step(err.args[1])
            raise StepFailedException(err.args[0])
```

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 init Method

Defining the Step Name, Description, and Version

From the **init** method, you can define the friendly name, the step description, and the step version. The following code examples contain the definitions for each value:

```
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 "GetREST" step, the friendly name, description, and version number are defined like this:

```
def __init__(self):
```

```
    self.friendly_name = "GetREST"
```

```
    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"
```

Defining Parameters for the Step

From the **init** method, you also define the input parameters for the step. The input parameters 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. PowerFlow will examine the parameters for the step and enforce the parameters when the step is run.

> **TIP:** Parameters display as editable fields on the **Configuration** pane for that step in the PowerFlow user interface.

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

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

```
self.new_step_parameter(
        name=<parameter_name>,
        description="<description>",
        sample_value="<sample_value>",
        default_value=<default_value>,
        required=<True/False>,
        param_type=parameter_types.<Number/String/Boolean>Parameter(),
    )
```

where:

- `name`. The name of the parameter. This value will be used to create a name:value tuple in the PowerFlow 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`.
- `param_type`. Specifies the type of parameter. Options include Number, String, Boolean. This setting is optional.

The following is an example from the "Cache Save" step from the "Base Steps" SyncPack:

```
self.new_step_parameter(
        name=SAVE_KEY,
        description="The key for which to save this data with",
        sample_value="keyA",
        default_value=None,
        required=True,
        param_type=parameter_types.StringParameterShort(),
    )
```

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 data from a previous step* in the application
- *Save data for use by the next step* in the 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 and other examples of the logic in a step, see the any of the steps in the "*Base Steps*" SyncPack provided with PowerFlow.

Transferring Data Between Steps

An essential part of integrations is passing data between tasks. PowerFlow 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. For more information about pickle, see https://docs.python.org/3/library/pickle.html.

For example:

```
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 PowerFlow 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 more information, see *Transferring 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*

> **IMPORTANT:** To retrieve data from a previous step:
>
> 1. That previous step must save the data with the *save_data_for_next_step* function.
> 2. The PowerFlow 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 **join_previous_step_data** or **get_data_from_step_by_order** functions allow you to create a more generic, reusable step.

For example:

```
get_data_from_step_by_name('<step_name>')
```

where *<step_name>* is the name of a previous step in the PowerFlow application. Use the system name of the step, not the "friendly name" with spaces that appears in the PowerFlow user interface.

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 application.

For example:

```
get_data_from_step_by_order(<position>)
```

where: *<position>* is the position of the step (the order that the step was run) in the PowerFlow application. Position starts at 0 (zero).

For example:

- Suppose your 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 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 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.

For example:

```
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 the following:

```
self.join_previous_step_data(["stepA", "stepD"]),
```

The following is an example of the *join_previous_step_data* function in the "SaveToCache" step (included in each PowerFlow system):

```
def execute(self):
        data_from = self.get_parameter(DATA_FROM_PARAM, {})
        if data_from:
            data_to_cache = self.join_previous_step_data(data_from)
        else:
            data_to_cache = self.join_previous_step_data()
        ...
```

Step Parameters

Steps accept arguments, called *input parameters*. Users can configure these parameters to specify the values, variables, and configurations to use when executing.

Base Parameters Available in All Steps

The BaseStep class 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 applications and use different names for the step in each 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 can define one ore more input parameters for the step. The PowerFlow 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 PowerFlow system will display an error message and will not execute the step.

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

```
self.new_step_parameter(
            name=<parameter_name>,
            description="<description>",
            sample_value="<sample_value>",
            default_value=<default_value>,
            required=<True/False>,
            param_type=parameter_types.<Number/String/Boolean>Parameter(),
        )
```

where:

- `name`. The name of the parameter. This value will be used to create a name:value tuple in the PowerFlow 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`.
- `param_type`. Specifies the type of parameter. Options include Number, String, Boolean. This setting is optional.

The following is an example from the GetREST 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>*. 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.

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

```
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 PowerFlow application that calls "GETgoogle", we specified:

```
"steps": [

  {

    "file": "GetREST",

    "name": "GETgoogle",

    "output_to": ["next_step"],

    "prefix_url": "http://google.com"

  }

],
```

Suppose we use the **get_parameter** function in the step "GETgoogle" 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

PowerFlow lets you define variables so that input parameters can be populated dynamically.

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

```
${<exampleVariable>}
```

PowerFlow includes the following types of variables that you can use in parameters:

- `${object_from_previous_step}`. PowerFlow will search the data from the previous steps for `object_from_previous_step`. If found, PowerFlow substitutes the value of the object for the variable.

- `${config.<exampleVariable>}`. Configuration variables are defined in a stand-alone file that lives on PowerFlow and can be accessed by all applications and their steps. Including the **config.** prefix with a variable tells PowerFlow 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 PowerFlow application. These variables can be accessed only by steps in the application. Including the **appvar.** prefix with a variable tells PowerFlow to look in the 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 *Defining Variables for an Application*.

Defining Logging for the Step

PowerFlow includes a logger for steps. The BaseStep 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

PowerFlow 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 in *Viewing Logs*.

Uploading Your Step

When you create a new step or edit an existing step, you must upload the step to the PowerFlow system.

There are two ways to upload a step to the PowerFlow system:

- *At the command line with the iscli utility*
- *With the API*

Uploading a Step with iscli

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

To upload a step to PowerFlow using iscli:

1. Either go to the console of PowerFlow 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_powerflow> -P <port_number_of_http_on_powerflow> -U <user_
   name> -p password
   ```

   where:

   - `<path_and_name_of_step_file>` is the full pathname for the step.
   - `<hostname_or_IP_address_of_powerflow>` is the hostname or IP address of PowerFlow.
   - `<port_number_of_http_on_powerflow>` is the port number to access PowerFlow. The default value is 443.

- *<user_name>* is the user name you use to log in to PowerFlow.
- *password* is the password you use to log in to PowerFlow.

Uploading a Step with the API

PowerFlow includes an *API* that you can use to upload steps.

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 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, PowerFlow 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, PowerFlow returns a 200 return code, logs, output, and the result of the step.

> **IMPORTANT:** If the step does not complete within the wait time, PowerFlow 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 PowerFlow 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 application. You can then use the information in step 3-6 below to view step logs.

To view the log information for a step:

1. Run a PowerFlow application.

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

    ```
    GET <URL_for_PowerFlow>/api/v1/applications/<application_name>/logs
    ```

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_PowerFlow>/<href_value>
```

    where `<href_value>` is the href value you can copy from the log file for the application. The href value is another version of the step name.

> **NOTE**: To view logs for subsequent runs of the 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_PowerFlow>/<task_id>
```

    where `<task_id>` is the ID value you can copy from the log file for the 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.

Default Steps

The "Base Steps" SyncPack contains a default set of steps that are used in a variety of different SyncPacks. You must install and activate this SyncPack before you can run any of the other SyncPacks.

This SyncPack is included with the most recent release of the PowerFlowPlatform.

Starting with version 1.5.0 of the "Base Steps" SyncPack, the "QueryREST" has been deprecated. ScienceLogic recommends that you use REST steps included in version 1.5.0 instead: "GetREST", "PostREST", "DeleteREST", and "PutREST".

TIP: To view the latest releases of this SyncPack, see *SL1 PowerFlow SyncPack Release Notes*.

NOTE: You can download this SyncPack from the [PowerPacks](#) page of the Support Site.

Chapter

3

Creating an SL1 PowerFlow Application

Overview

This chapter explains how to create your own application that can run on a PowerFlow system.

> **TIP:** You can also use the PowerFlow user interface to perform many of the actions in this chapter. For more information, see *Managing SL1 PowerFlow Applications*.

This chapter covers the following topics:

What is a PowerFlow Application?

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

An application combines a set of PowerFlow steps that execute a workflow. The input parameters for each step are also defined in the application and can be provided either directly in the step or in the parent application.

The following is an example of a PowerFlow application:
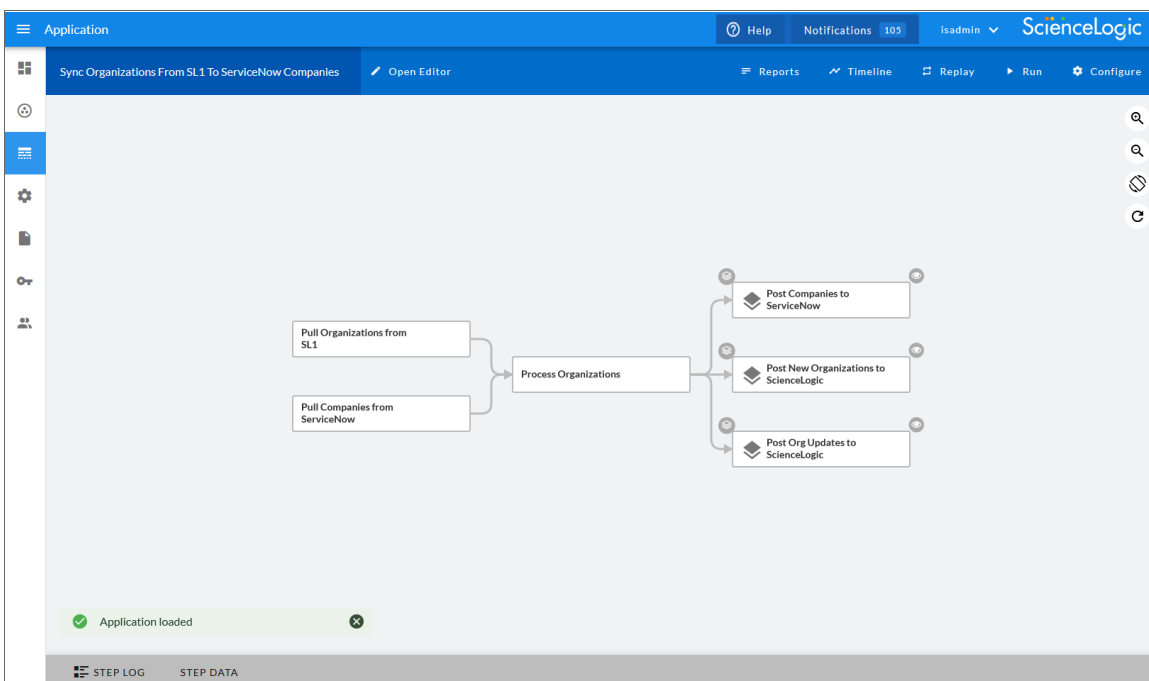
PowerFlow application JSON objects are defined by configuration settings, steps that make up the application, and application-wide variables used as parameters for each step. The parameters of each step can be configured dynamically, and each step can be named uniquely while still sharing the same underlying class, allowing for maximum re-use of code.

You can run an application in the PowerFlow user interface. You can also execute an application through the REST API, and PowerFlow will process the application as an asynchronous task. Executing an application from the REST API lets you dynamically set parameter values for the variables defined in the application.

During processing, PowerFlow generates a unique task ID for the application and each of its tasks. Using the task IDs, you can poll for the status of the application and the status of each individual running step in the application.

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

Workflow for Creating an Application

To create a PowerFlow application, you must perform these tasks:

1. *Download a copy of the application template.*
2. *Define the required fields for the application.*
3. *Create the list of steps and step parameters.*
4. *Define application variables.*
5. *Upload the application to PowerFlow.*
6. *Validate and test the application.*

View the "Template App" PowerFlow Application

PowerFlow includes a pre-defined application called *integration_template*. This default application appears in the PowerFlow user interface as the *Template App* application.

To view the *Template App* application:

1. Log into the PowerFlow user interface and go to the **Applications** page.
2. In the **Search Applications** field or the **Application Name** filter, type "Template App."
3. Select **Template App** from the list. The **Template App** detail page appears.

Creating an Application from the Application Template

The easiest way to create a new application is to use the application template that is included with the PowerFlow 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_PowerFlow>/api/v1/applications/integration_template
   ```

   where `<URL_for_PowerFlow>` is the IP address or URL for the PowerFlow 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 application includes the .json suffix. The file name must be unique within your PowerFlow system and cannot contain spaces.

Defining Required Fields for the Application

A PowerFlow application must include the following `key:value` pairs:

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

The **integration_template** includes the required fields listed above. 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_PowerFlow/api/v1/applications/<application_name>`

   where `<application_name>` is the application you want to edit.

2. Copy the contents of the 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 application template.

   - ***configuration*** (if applicable). Configuration variables are defined in a stand-alone file called a configuration object that lives on the PowerFlow system and can be accessed by all PowerFlow applications and their steps. If your application or the steps in the application reference configuration variables, you must specify the name of the configuration object in this value. For more details on creating a configuration object, see the section on *Defining a Configuration Object*.

   - ***description***. This description will be displayed in the PowerFlow user interface when viewing the available applications.

   - ***friendly_name***. The name of the application as it will appear in the user interface.

   - ***name***. Name of the file, without the .py suffix.

   - ***Version***. Version number of the application.

5. Save your changes. Save the file to the same name (**<application_name>.json**.)

6. To upload the application to the PowerFlow system, see the section on *Uploading an Application.*

Example Code: integration_template

```json
{
    "app_variables": [
        {
            "created_by_user": true,
            "default_value": "DefaultValue",
            "description": "Variables defined here will be available in
all steps of an application. Note that setting app variables manually is
not recommended. Whenever an app is saved with steps referencing an
appvar, one will automatically be created",
            "name": "exampleVariable",
            "required": true,
            "sample_value": "SampleValue",
            "type": {
                "args": {
                    "length": "short",
                    "password": false
                },
                "id": "string"
            },
            "value": "exampleValue"
        }
    ],
    "author": "ScienceLogic Inc.",
    "configuration": null,
    "content_type": "app",
    "description": "Application template for creating apps",
    "friendly_name": "Template App",
    "generate_report": false,
    "href": "/api/v1/applications/integration_template",
    "id": "integration_template",
    "last_modified": 1680277982,
    "last_run": null,
    "meta": {
        "hidden": false
    },
    "name": "integration_template",
    "queue": null,
```

```
    "steps": [
        {
            "file": "GetREST",
            "method": "GET",
            "name": "Get REST Test",
            "output_to": [
                "Another GET Run"
            ],
            "prefix_url": "https://ht-
tpbin.org/get?somevar=${appvar.exampleVariable}",
            "step_type": "step",
            "syncpack": "base_steps_syncpack"
        },
        {
            "data_from": [
                "GET REST Test"
            ],
            "file": "GetREST",
            "method": "GET",
            "name": "Another GET Run",
            "prefix_url": "https://ht-
tpbin.org/get?somevar=${appvar.exampleVariable}",
            "step_type": "step",
            "syncpack": "base_steps_syncpack"
        }
    ],
    "syncpack": "base_steps_syncpack",
    "version": "1.2.0"
}
```

Creating the List of Steps and Step Parameters

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

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

To edit the steps section of the application:

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

    ```
    GET <URL_for_PowerFlow>/api/v1/applications/<application_name>
    ```

    where `<application_name>` is the application you want to edit.

2. Copy the contents of the application to a source-code editor.

3. Search for the section of the file that contains the text `"steps"`.

4. The section of the application that specifies steps should look like the following:

    ```
    "steps":[

        {

            "name": "GETgoogle",

            "file": "GetREST",

            "prefix_url": "http://google.com",

            "output_to": ["next_step"]

        },

        {

            "name": "next_step",

            "file": "someOtherStep"

        }

    ]
    ```

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 application your downloaded.

6. To upload the application to the PowerFlow system, see the section on *Uploading an 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 application can use this name to refer to the step. This value can include spaces. This name must be unique to the 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 applications and use different names for the step in each application.

Specifying the Parameters

When you add a step to an 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 "GetREST"):

1. Using an API tool like Postman or cURL, use the API **GET /steps/{step_name}**:
   ```
   GET <URL_for_PowerFlow>/api/v1/steps/GetREST
   ```

2. The step "GETgoogle" (and its parent file, "GetREST") includes the following parameter, which is defined as "required":
   ```
   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. As a result, when the application includes the step "GETgoogle", the application must supply values for the required parameters:
   ```
   "prefix_url": "http://google.com",
   ```

Transferring Data Between Steps

An essential part of PowerFlow applications is passing data between tasks. PowerFlow 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 a PowerFlow 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 application:

1. Using an API tool like Postman or cURL, use the API **GET /applications/{application_name}**:
   ```
   GET ><URL_for_PowerFlow/api/v1/applications/<application_name>
   ```

   where `<application_name>` is the application you want to edit.

2. Copy the contents of the application to a source-code editor.

3. If you wanted the step "GETgoogle" to pass its output to the step "next-step", the application would include the following:

```
"steps":[

    {

        "name": "GETgoogle.com",

        "file": "GetREST",

        "prefix_url": "http://google.com",

        "method": "GET",

        "output_to": ["next_step"]

    },

    {

        "name": "next_step",

        "file": "someOtherStep"

    }

]
```

> **NOTE**: If steps have dependencies , PowerFlow will first analyze all steps in an 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 save your file as a **.JSON** file with the same name as the application you downloaded.

5. To upload the application to PowerFlow, see the section on *Uploading an 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 PowerFlow 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.

> **WARNING:** As a best practice, you should only edit the **retry_max** parameter and avoid editing any of the other retry parameters. Only advanced users who understand how the retries work and their side effects when they are not set correctly should change the other retry parameters.

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

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

- **retry_backoff**. Instead of using a defined interval between retries, the PowerFlow system will incrementally increase the interval between retries. Possible values are True or False. The default value is False.

- **retry_jitter**. Instead of using a defined interval between retries, the PowerFlow system will retry the step execution at random intervals. Possible values are True or 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. The default value is 600 seconds.

- **retry_countdown**. The interval between retries, in seconds. If you enabled **retry_backup**, the PowerFlow system will incrementally increase this interval. The default value is 180.

> **WARNING:** Use caution when editing the **retry_countdown** option. If you set it to a value smaller than the default of 180 seconds, PowerFlow might experience collisions between task executions, and PowerFlow might stop unexpectedly. If you set this option to a value larger than the default, you might have to wait longer for a task to execute.

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

1. Using an API tool like Postman or cURL, use the API GET **/applications/{application_name}**:
   ```
   GET <URL_for_PowerFlow>/api/v1/applications/<application_name>
   ```

   where `<application_name>` is the application you want to edit.

2. Copy the contents of the application to a source-code editor.

3. If you wanted to add retry options to the step "GETgoogle", you could include the following:

```
"steps":[

    {

      "name": "GETgoogle",

      "file": "GetREST",

      "prefix_url": "http://google.com",

      "retry_max": 5

      "retry_backoff": True

      "retry_backoff_max": 600

      "retry_countdown": 120

      "output_to": ["next_step"]

    },

    {

      "name": "next_step",

      "file": "someOtherStep"

    }

]
```

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

Defining Variables for an Application

Application variables are defined in the PowerFlow application. These variables can be accessed only by steps in the application. In a step, including the **appvar.** prefix in a variable tells the PowerFlow system to look in the 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_PowerFlow>/api/v1/applications/<application_name>
```

where *<application_name>* is the application you want to edit.

2. Copy the contents of the 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. This is 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 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 application your downloaded.

8. To upload the application to PowerFlow system, see the section on *Uploading an Application.*

Uploading the Application to PowerFlow

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

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

Uploading an Application with iscli

The PowerFlow system includes a command-line utility called **iscli**. When you install PowerFlow system, **iscli** is automatically installed.

To upload an application to the PowerFlow system using iscli:

1. Either go to the console of the PowerFlow 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_application_file>.py -H <hostname_
or_IP_address_of_powerflow> -P <port_number_of_http_on_powerflow> -U
<user_name> -p password
```

   where:

   - `<path_and_name_of_application_file>` is the full pathname for the application.
   - `<hostname_or_IP_address_of_powerflow>` is the hostname or IP address of PowerFlow.
   - `<port_number_of_http_on_powerflow>` is the port number to access PowerFlow. The default value is 443.
   - `<user_name>` is the user name you use to log in to PowerFlow.
   - `password` is the password you use to log in to PowerFlow.

Uploading an Application with the API

To specify that an application always uses a custom queue, edit the **.JSON** file for the application::

1. Using an API tool like Postman or cURL, use the API **POST/applications/{application_name}/run**:
```
POST https://<URL_for_PowerFlow>/api/v1/applications/<application_
name>/run
```

   where `<application_name>` is the application you want to edit.

2. Copy the contents of the 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. The application will now always use the specified queue.

7. To upload your changes to the PowerFlow system, see *Uploading an application*.

Running the Application

After uploading an application to PowerFlow, you can run it to ensure that it works as designed.

To run an application from the API:

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

```
POST https://<URL_for_your_PowerFlow_
system>/api/v1/applications/<application_name>/run
```

where `<application_name>` is the application you want to edit.

2. The body of the post should contain:

```
{

"name": "name_of_application",

"params": {}

}
```

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

   - parameters use the following syntax:

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

   - `wait`. Number of seconds to wait for application to complete.

   - `configuration`. The configuration object to use with the application. For details, see *configuration objects*.

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 Application with a Custom Queue

If your PowerFlow system runs multiple applications, you might want to ensure that one of those applications runs before other 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 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 application to use the custom queue

Defining a Custom Queue

To create a custom queue:

1. SSH to the PowerFlow 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/pf-worker:latest

       environment:
   ```

```
  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 volumes:

- "/var/log/iservices:/var/log/iservices"

- /var/log/iservices:/var/log/contentapi:rw

- read_only: true

  source: syncpacks_virtualenvs

  target: /var/syncpacks_virtualenvs

type: volume
```

steprunner_1:

```
image: sciencelogic/pf-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

volumes:

- "/var/log/iservices:/var/log/iservices"

- /var/log/iservices:/var/log/contentapi:rw

- read_only: true

  source: syncpacks_virtualenvs

  target: /var/syncpacks_virtualenvs

  type: volume

steprunner_2:

image: sciencelogic/pf-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

volumes:

- "/var/log/iservices:/var/log/iservices"

- /var/log/iservices:/var/log/contentapi:rw

- read_only: true

  source: syncpacks_virtualenvs

  target: /var/syncpacks_virtualenvs

  type: volume
```

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

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

Running the Application with a Custom Queue

- To add additional services to your PowerFlow 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>* are the names 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 PowerFlow to pick up the changes to the **docker-compose** file. To do this, SSH to PowerFlow and execute the following command:

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

Configuring an Application to Use a Custom Queue

There are two ways to specify that a PowerFlow application should use a custom queue:

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

Configuring an Application to Always Use a Custom Queue

To specify that an application always uses a custom queue, edit the .**JSON** file for the application:

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

```
POST https://<URL_for_PowerFlow>/api/v1/applications/<application_
name>/run
```

   where *<application_name>* is the application you want to edit.

2. Copy the contents of the 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, add 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.

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",
```

5. Save your changes. Save the file to the same name: **<application_name>.json**. The application will now always use the specified queue.

6. To upload your changes to the PowerFlow system, see *Uploading an application*.

Configuring an Application to Use a Custom Queue at Run Time

After uploading an application, you can run it and specify a queue.

To run an application from the PowerFlow API:

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

   ```
   POST https://<URL_for_PowerFlow>/api/v1/applications/<application_
   name>/run
   ```

   where *<application_name>* is the application you want to edit.

2. The body of the post should contain:

   ```
   {

   "name": "name_of_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 Object

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

In a step, including the **config.** prefix with a variable tells PowerFlow 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",
```

Defining a Configuration Object

```
                    "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 PowerFlow 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 PowerFlow 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 PowerFlow 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 5-6 for each configuration variable.

8. Save the new file as **<config_file>.json**.

   where **<config_file>** 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 PowerFlow system and cannot contain spaces.

Uploading the Configuration Object to PowerFlow

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

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

Uploading a Configuration with iscli

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

To upload a configuration file to the PowerFlow system using iscli:

1. Either go to the console of the PowerFlow 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_PowerFlow> -P <port_number_of_http_on_PowerFlow> -U
<user_name> -p <password>
```

   where:

   - `<path_and_name_of_configuration_file>` is the full pathname for the step.
   - `<hostname_or_IP_address_of_PowerFlow>` is the hostname or IP address of the PowerFlow system.
   - `<port_number_of_http_on_PowerFlow>` is the port number to access the PowerFlow system. The default value is 443.
   - `<user_name>` is the user name you use to log in to the PowerFlow system.
   - `<password>` is the password you use to log in to the PowerFlow system.

Uploading a Configuration Object Using the API

PowerFlow includes an *API*. When you install the PowerFlow system, the API is available.

To upload a configuration with the 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.

View Logs for an Application

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

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

```
GET <URL_for_PowerFlow>/api/v1/applications/<application_name>/logs
```

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_PowerFlow>/<href_value>
```

where *<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 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_PowerFlow>/<task_id>
```

where *<task_id>* is the ID value you can copy from the log file for the application. The task ID specifies the latest execution of the application.

Chapter

4

Creating a SyncPack

Overview

This chapter describes how to package your PowerFlow applications, steps, configuration objects, and utilities into a SyncPack.

This chapter covers the following topics:

What is a SyncPack?

A **SyncPack** is a Python **.whl** file that contains PowerFlow applications, steps, configuration objects, and utilities that can be used in PowerFlow. SyncPacks are versioned and can define dependencies on other SyncPacks or on external Python packages.

When applications from a SyncPack are executed, they run within a Python Virtual Environment that contains the SyncPack itself and its dependencies.

Generating the SyncPack Structure

ScienceLogic uses a **Cookiecutter** repository for generating the standard SyncPack structure. This structure includes all necessary files and the default directory structure that is generated based on your response to the prompts.

For more information about Cookiecutter, see https://cookiecutter.readthedocs.io/en/latest/.

The GitHub repository for the Cookiecutter project can be found at https://github.com/ScienceLogic/is_syncpack_cookiecutter.

To use Cookiecutter to generate the SyncPack structure:

1. On a system with Python 3.6 or later installed, run the following commands to install Cookiecutter (use a virtual environment if possible):

    ```
    pip install cookiecutter
    ```

    ```
    cookiecutter https://github.com/ScienceLogic/is_syncpack_
    cookiecutter.git
    ```

    > **IMPORTANT:** To use Cookiecutter, you will need to install Visual Studio Code (VSCode) or PyCharm. You will also need to install Docker.

2. Log in to the SL container registry:

    ```
    docker login registry.scilo.tools
    ```

    > **TIP:** You can find the credentials for this step by logging in to the Harbor user interface, clicking your user name in the top right, and selecting *User Profile*. From the **User Profile** dialog, copy the username and CLI secret (key) you will need to authenticate with when running the `docker login` command.

3. Create a root directory where you can start building the SyncPack. For example: **/home/username/pf_syncpacks_workspace**.

4. In the root directory from step 3, run the Cookiecutter tool to create a SyncPack directory structure.

> **TIP:** For more information about the requirements for the Cookiecutter tool, see the [Syncpack Cookie Cutter Readme file](#).

5. The Cookiecutter tool will ask you for the following information:

    - *author*. Name of the person or organization that created this SyncPack.

    - *url_project*. URL of the repository where this SyncPack is hosted. You can use a placeholder or fake URL in this field if you do not currently plan to push code to a live PowerFlow system.

    - *syncpack_name*. Developer name of the SyncPack, without spaces. For example: *test_syncpack*.

    - *syncpack_friendly_name*. Friendly name of the SyncPack, which will display in the PowerFlow user interface. For example: *Test SyncPack*.

    - *syncpack_description*. Short description of the SyncPack, which will display in the PowerFlow user interface and in the **readme** file.

    - *version*. Number of the SyncPack version.

    - *requires_minimum_pf_version*. The minimum version of the PowerFlow platform required for this SyncPack.

    - *dev_container_source*. Use "SL External" if you are not a ScienceLogic employee.

After you complete this process, Cookiecutter creates a new directory in your current working directory named with the *syncpack_name* you specified, above. For more information about the contents of the new directory, see *SyncPack Structure*.

To continue the process of creating the SyncPack, see *Building a SyncPack*.

SyncPack Structure

After you use Cookiecutter to generate the SyncPack structure, you can access the following files and folders in the folder created by the previous process.

- *MANIFEST.in*. Allows the Python **.whl** file to contain non-Python files.

- *README.md*. The **readme** file for this SyncPack, pre-populated with basic information.

- *setup.py*. Python setup file for building the SyncPack. You should not need to modify this file.

- *docker*. This directory contains the following Docker information:

    ○ *docker-compose.yml*. Docker Compose file that uses the above container.

    ○ *Dockerfile*. Sample Dockerfile that can be used for building and testing the SyncPack in a continuous integration and continuous delivery (CI/CD) environment.

- **test_syncpack**. This directory contains all SyncPack content, including the following:
    - **__init__.py**. Python init file.
    - **apps**. This directory contains SyncPack application .**json** files.
    - **configs**. This directory contains SyncPack configuration object .**json** files.
    - **meta.json**. This file defines the *SyncPack properties*.
    - **steps**. This directory contains SyncPack Step Python files.
    - **util**. This directory contains Python classes or modules that can be used by multiple steps in the SyncPack.
    - **tox.ini**. Basic tox configuration for running unit tests.

SyncPack Properties

The **meta.json** file contains all configurable properties for a SyncPack. This file is where you define the name, friendly name, version, description, default schedules, tags, and supported PowerFlow version of the SyncPack.

`requires_dist` is where you define the Python packages on which this SyncPack depends. For example, if you would like to use a step from the "Base Steps" SyncPack and also use Amazon's boto library, you would use the following:

```
"requires_dist": ["boto", "base_steps_syncpack"],
```

Example Code: meta.json

```
{

    "name": "test_syncpack",

    "version": "0.0.1",

    "summary":"Testing Syncpack",

    "description":"PF Syncpack for testing",

    "author":"John Smith",

    "home_page":"https://example.com/test_syncpack",

    "requires_dist":[],

    "schedules": [

      {

        "application_id": "dummy_app",
```

```
        "entry_id": "dummy app",

          "params": {},

          "schedule": {

            "schedule_info": {

                "run_every": 3600

            },

            "schedule_type": "frequency"

          }

      }

    ],

    "tags":["test_syncpack"],

    "critical": [],

    "requires_is_version": ">=2.0.0"

  }
```

Building a SyncPack

After you complete your SyncPack structure and place your applications, steps, and configuration objects in the right directories, you can build the SyncPack **.whl** file (wheel).

You can build a SyncPack with Python3 or with the *iscli utility* that comes with PowerFlow.

> **NOTE**: Not included is how to build a SyncPack automatically with CI/CD platforms.

Building a SyncPack with Python3

To build the SyncPack wheel with Python3, run the following command from the top-level SyncPack directory:

```
python3 setup.py sdist bdist_wheel
```

This command creates **build** and **dist** folders within your SyncPack directory. In the **dist** folder, you will now have a Python wheel that you can upload to PowerFlow.

Building a SyncPack with the iscli Utility

If you build your SyncPack with the iscli utility, the utility builds and uploads the SyncPack to PowerFlow in one step.

To build and upload your SyncPack with the iscli utility, run the following command:

```
iscli -H <IP_PowerFlow_host> -U <PowerFlow_user> -p <PowerFlow_password> -
ukf </path/to/SyncPack/Directory>
```

where `</path/to/SyncPack/Directory>` is the SyncPack root that contains the **setup.py** file.

Extending an Existing SyncPack

Because the content that is included in a SyncPack cannot be overwritten inside PowerFlow, you will need to extend existing SyncPacks with your own SyncPack to add customizations.

Restrictions

- The names for applications, steps, and configuration objects must be unique within your entire PowerFlow platform. For example, you cannot add a custom step named "GetREST", because that step is already included in the "Base Steps" SyncPack. You will need to change the name of your custom step.

- Your new SyncPack must include the SyncPack that you are extending in the **requires_dist** list in your **meta.json**. This ensures that your SyncPack will have access to the applications and steps of the parent SyncPack in its execution environment.

- After you install the SyncPack, its environment will never be modified, even if the parent SyncPacks are updated. If you wish to include changes from the parent SyncPack, you will need to re-install your customized SyncPack.

Advanced Method: Extracting and Updating an Existing SyncPack

This method of modifying a SyncPack includes the following steps:

1. Download an existing SyncPack from a PowerFlow system

2. Modify the contents of the SyncPack and save as a new SyncPack

3. Re-upload and install the new SyncPack.

Although this advanced method involves more manual steps, this method is recommended over the basic method described in later sections.

To extract and update an existing SyncPack:

1. Identify the SyncPack that you want to extend and download it from the PowerFlow system or download it from ScienceLogic Support site at https://support.sciencelogic.com/s/powerpacks.

> **TIP:** You can download any SyncPack on a PF system from the devpi interface:
> **https://10.2.11.232:3141/isadmin/syncpacks**.

2. You can access the contents of the **.whl** file for the SyncPack by unzipping the **.whl** file. Run the following command to unzip the contents into the current working directory:
   ```
   unzip <syncpack_whl_file>
   ```

   where `<syncpack_whl_file>` is the name of the **.whl** file.

   For example:

   ```
   unzip base_steps_syncpack-1.0.0-py3-none-any.whl
   ```

3. Update the SyncPack directory name to make it easier to recognize than the default version of the SyncPack you are modifying.

4. Update the following elements in the *meta.json file*:

   - Change the SyncPack name to match the updated directory name chosen in step 3.

   - Update the `requires_dist` parameter to include the original SyncPack which is being extending. This ensures the modified SyncPack can use the existing code as necessary.
     For example, if you are modifying **base_steps_syncpack** version 1.0, add the following to the **requires_dist** parameter:
     ```
     "requires_dist":[ "base_steps_syncpack==1.0" ]
     ```

5. Delete the contents of the directories for the applications and steps that you are not going to be customizing. As a best practice, start with a single application or step and add required steps or applications as necessary.

6. Modify the existing applications and steps as needed:

   - Be sure to change the file name of the Python file containing the step you are editing, as well as the name of the class contained in the file. The name of the Python file and the class name should match.

   - Be sure to change the file name of the application configuration file in the apps directory, or PowerFlow will reject the installation. Additionally, update the *friendly_name* parameter within the application JSON file.

   - Make the code changes you need to make to customize the applications and steps. If additional Python packages are required, be sure to add them to the `required_dist` list mentioned in step 4.

7. Run the following command to build the SyncPack with the new name and version contained in the **meta.json** file:
   ```
   python3 setup.py bdist_wheel
   ```

8. To upload the SyncPack using the iscli tool, run the following command:

```
iscli -ukf <new_syncpack_whl_file>
```

   where `<new_syncpack_whl_file>` is the name from the .**whl** file created from the previous step.

> **TIP:** To forcefully overwrite an existing version of this SyncPack, use the following command: `iscli -ukFf <new_syncpack_whl_file>`

9. To upload the SyncPack using the PowerFlow user interface, go to the **SyncPacks** page, click **[Import SyncPack]**, and select the .**whl** file.

Basic Method: Modifying Individual Steps and Applications

This basic method of modifying a SyncPack is less involved than the advanced method, and it is effective for making quick, minor changes to a SyncPack. This approach is limited, however, because the custom steps will not run inside a SyncPack.

Modifying Application Definitions

To change the API endpoint or query used in an existing PowerFlow application:

1. Download the existing application contents from PowerFlow by copying the content of this URL from your browser:
   <URL_for_PowerFlow>/api/v1/applications/<application_name>
   where **<application_name>** is the name of the application that you want to edit.

> **TIP:** To locate the **<application_name>** for a PowerFlow application, do a **GET /api/v1/applications** and use the "id" value for the application in the results.

2. Create a JSON file on your local system and give it a unique name for the new application. The name of this file cannot match the name of the existing application file.

3. Copy the contents of the result of step 1 into the new JSON file.

4. In the JSON code, change the `friendly_name`, `author`, `description`, and `version` as required.

5. Make any other changes to the application as needed, such as changing step names, adding new steps, and adding or changing parameters.

6. Upload the new json file as a new application to the PowerFlow system using the iscli tool:

```
iscli -uaf <my_new_app_name.json> -H <host>
```

Modifying a Step

You have two methods for modifying a step:

1. Change the code in a step itself.
2. Extend the step class and execute your own code before or after the existing step.

To change the logic performed within a step:

1. Download the existing application contents from PowerFlow by copying the content of this URL from your browser:
   <URL_for_PowerFlow>*/api/v1/steps/*<step_name>
   where ***<step_name>*** is the name of the step that you want to edit.

   > **TIP:** To locate the ***<step_name>*** for a PowerFlow application, do a **GET /api/v1/steps** and use the "id" value for the step in the results.

2. Create a Python file on your local system and give it a unique name for the new step. The name of this file cannot match the name of the existing step file.
3. Copy the contents of the result of step 1 into the new Python file.

   > **TIP:** `echo -e` can help with the newlines and make the indents correct when writing to a file.

4. In the step file, change the `class` name to match the name of the step file (without a **.py** extension).
5. In the `__init__()` section of the code, change the `friendly_name`, `description`, and `version` as required.
6. Make any other changes to the step as desired.

To "extend" the behavior of an existing step by performing work before or after the existing logic:

1. *Create a new step* and give it a unique name.
2. Instead of importing and extending **BaseStep**, import and extend the step class you would like to modify. For example:
   ```
   from base_steps_syncpack.steps.QueryGQL import QueryGQL

   class OptionalQueryGQL(QueryGQL):
   ```
3. Complete the `__init__()` as normal, but make sure to call init of the `super` class to include all of its parameters. For example:
   ```
   super(OptionalQueryGQL, self).__init__()
   ```

4. Create your additional logic in the `execute()` function. This logic can be performed either before or after you call the execute of the parent step. For example:

```
super(OptionalQueryGQL, self).execute()
```

> **NOTE**: If you would like to perform additional logic after the execute of the parent step runs, you can access the data from that step via `self.get_current_saved_data()`.

5. Upload the new Python file to the PowerFlow system using the iscli tool:

```
iscli -usf <new_step.py> -H <host>
```

Example Code: Full Extended Step

```python
from base_steps_syncpack.steps.QueryGQL import QueryGQL

from ipaascore import parameter_types

from ipaascommon.ipaas_utils import str_to_bool

class OptionalQueryGQL(QueryGQL):

  def __init__(self):

    super(OptionalQueryGQL, self).__init__()

    self.friendly_name = "Optional QueryGQL Call"

    self.description = "Optionally Runs GQL call."

    self.version = "1.0.0"

    self.new_step_parameter(

      name="enable",

      description="Should this step run.",

      default_value=True,

      required=False,

      param_type=parameter_types.BoolParameterToggle()

    )

  def execute(self):
```

```
enabled = str_to_bool(self.get_parameter("enable", None))

if enabled:

    super(OptionalQueryGQL, self).execute()

else:

    self.logger.flow(

        "Step is Disabled. Returning empty dict."

    )

    self.save_data_for_next_step(dict())
```

Uploading a SyncPack with the PowerFlow User Interface

After you create your SyncPack wheel, you can upload it to PowerFlow. You can upload this SyncPack to PowerFlow using the PowerFlow user interface or with the iscli utility.

Uploading a SyncPack with the User Interface

To import a SyncPack in the PowerFlow user interface:

1. On the **SyncPacks** page (⚙) of the PowerFlow user interface, click **[Import SyncPack]**. The **Import SyncPack** page appears.

2. Click **[Browse]** and select the **.whl** file for the SyncPack you want to install. You can also drag and drop a .whl file to the **Import SyncPack** page.

3. Click **[Import]**. PowerFlow registers and uploads the SyncPack. The SyncPack is added to the **SyncPacks** page.

4. You will need to activate and install the SyncPack in PowerFlow. For more information, see *Activating and Installing a SyncPack*.

NOTE: You cannot edit the content package in a SyncPack published by ScienceLogic. You must make a copy of a ScienceLogic SyncPack and save your changes to the new SyncPack to prevent overwriting any information in the original SyncPack when upgrading.

Uploading a SyncPack with the iscli Utility

To upload a SyncPack with the iscli utility:

1. Using a tool like secure copy (SCP), copy the SyncPack from your local computer to your PowerFlow host.

2. Start an SSH session with your PowerFlow host.

3. Run the following command:

```
iscli -ukf /path/to/test_syncpack-0.0.1-py2.py3-none-any.whl
```

> **NOTE:** If you have already uploaded this SyncPack to your PowerFlow system and you want to re-upload with the same version, add the `-F` flag to the above command.

4. You can now activate the SyncPack in PowerFlow. See the following topic for more information.

Activating and Installing a SyncPack

After you upload a SyncPack to PowerFlow, you need to activate and install the SyncPack to access its applications, steps, and configuration objects.

For the "Activate & Install Syncpacks" application, the **retry_max** parameter for the "Activate Syncpack" and "Install Syncpack" steps is set to 3 by default. The time between those retries is calculated randomly based on the number of retries. These settings prevent the steps from colliding with each other when the steps are run in a environment with a large number of syncpack_steprunners that are trying to install a SyncPack at the same time in their respective volumes.

> **WARNING:** As a best practice, you should only edit the **retry_max** parameter and avoid editing any of the other retry parameters. Only advanced users who understand how the retries work and their side effects when they are not set correctly should change the other retry parameters. For more information, see *Defining Retry Options for a Step*.

Activating and Installing a SyncPack with the User Interface

To activate and install a SyncPack in the PowerFlow user interface:

1. On the **SyncPacks** page of the PowerFlow user interface, click the **[Actions]** button (⋮) for the SyncPack you want to install and select *Activate & Install*. The **Activate & Install SyncPack** modal appears.

> **NOTE:** If you try to activate and install a SyncPack that is already activated and installed, you can choose to "force" installation across all the nodes in the PowerFlow system.

> **TIP:** If you do not see the PowerPack that you want to install, click the Filter icon (≡) on the **SyncPacks** page and select *Toggle Inactive SyncPacks* to see a list of the imported PowerPacks.

2. Click **[Yes]** to confirm the activation and installation. When the SyncPack is activated, the **SyncPacks** page displays a green check mark icon (✓) for that SyncPack. If the activation or installation failed, then a red exclamation mark icon (❗) appears.

3. For more information about the activation and installation process, click the check mark icon (✓) or the exclamation mark icon (❗) in the **Activated** column for that SyncPack. For a successful installation, the "Activate & Install SyncPack" application appears, and you can view the Step Log for the steps. For a failed installation, the **Error Logs** window appears.

4. If you have other versions of the same SyncPack on your PowerFlow system, you can click the **[Actions]** button (⋮) for that SyncPack and select *Change active version* to activate a different version other than the version that is currently running.

Activating and Installing with the iscli Utility

To activate and install an uploaded SyncPack using the iscli utility, run the following command:

```
iscli -AIkn <syncpack_name> -V <syncpack version>
```

where `<syncpack_name>` is the name from the **meta.json** file, and `<syncpack_version>` is the version number of the SyncPack.

For example:

```
iscli -AIkn test_syncpack -V 0.0.1
```

> **NOTE:** To force a re-activation of the SyncPack, add `-F` to the above command.

Chapter

5

ipaascore.BaseStep class

Overview

This chapter describes functions included in the **ipaascore.BaseStep** class. The **ipaascore.BaseStep** class is a Python class that is included with PowerFlow, and it contains multiple predefined functions that you can use when you are writing or editing a step.

This chapter covers the following topics:

get_app_variable

Description

Retrieve the value of an application variable.

Syntax

```
get_app_variable("<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 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 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 application, along with the position of the step (the order that the step was run in the 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.

> **IMPORTANT:** To retrieve data from a previous step:
>
> 1. That previous step must save the data with the *save_data_for_next_step* function.
>
> 2. The PowerFlow 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 hard-coded in the function. The **join_previous_step_data** or **get_data_from_step_by_order** functions 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 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 application.

> **IMPORTANT:** To retrieve data from a previous step:
>
> 1. That previous step must save the data with the *save_data_for_next_step* function.
> 2. The PowerFlow 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(<position>)
```

Parameters

`<position>`. The position of the step (order that the step was run) in the PowerFlowapplication. Position values start at 0 (zero) .

Return

The data that was saved by the previous step.

Exception

DataNotAvailableException

Example

- Suppose your 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":

```
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 application that calls "GETgoogle.com", we specified:

```
"steps": [

  {

    "file": "GetREST",

    "name": "GETgoogle",

    "output_to": ["next_step"],

    "prefix_url": "http://google.com"

  }

],
```

Suppose we use the *get_parameter* function in the step "GETgoogle" 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

`<parameter_name>`. The name of the parameter that you want to retrieve the value for.

`<step_name>`. The name of 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 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 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.

> **IMPORTANT:** To retrieve data from a previous step:
>
> 1. That previous step must save the data with the *save_data_for_next_step* function.
>
> 2. The PowerFlow application must specify that the data from the previous step should be passed to the current step using the *output_to* parameter.

Syntax

```
join_previous_step_data(<step_name>)
```

Parameters

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

```
self.join_previous_step_data(["stepA", "stepD"]),
```

Return

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

Example

The following is an example of the *join_previous_step_data* function in the "SavetoCache" step (included in each PowerFlow system):

```
def execute(self):
    data_from = self.get_parameter(DATA_FROM_PARAM, {})
    if data_from:
       data_to_cache = self.join_previous_step_data(data_from)
    else:
       data_to_cache = self.join_previous_step_data()
    ...
```

new_step_parameter

Description

Defines an input parameter for the step. The PowerFlow application 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, PowerFlow will display an error message and will not execute the step.

Syntax

```
self.new_step_parameter(
            name=<parameter_name>,
            description="<description>",
            sample_value="<sample_value>",
            default_value=<default_value>,
            required=<True/False>,
            param_type=parameter_types.<Number/String/Boolean>Parameter(),
        )
```

Parameters

- `name`. The name of the parameter. This value will be used to create a name:value tuple in the PowerFlow 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`.

- `param_type`. Specifies the type of parameter. Options include Number, String, Boolean. This setting is optional.

Example

The following is an example from the "Cache Save" step from the "*Base Steps*" SyncPack:

```
self.new_step_parameter(
        name=SAVE_KEY,
        description="The key for which to save this data with",
        sample_value="keyA",
        default_value=None,
        required=True,
        param_type=parameter_types.StringParameterShort(),
    )
```

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. For more information about pickle, see https://docs.python.org/3/library/pickle.html.

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 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 more information, see *Transferring Data Between Steps*.

validate_parameter_values

Description

Validates the parameter values provided for an 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

6

Reference: Tools for SL1 PowerFlow

Overview

PowerFlow includes the following developer tools to help you manage content:

- **iscli**. PowerFlow includes a command line tool called **iscli**. When you install PowerFlow, **iscli** is automatically installed. The iscli tool allows you to upload PowerFlow applications, steps, and configurations.

- **PowerFlow API**. PowerFlow includes an API. When you install PowerFlow, the API is available.

In addition, you can contact your ScienceLogic Customer Success Manager (CSM) to get access to the following tool, which is not included in a PowerFlow system:

- **PowerFlow SyncPack Cookiecutter**. Provides a template that you can use to create SyncPacks in a repeatable manner. For more information see *Generating the SyncPack Structure*.

This chapter covers the following topics:

iscli

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

To access the iscli tool:

1. Either go to the console of the PowerFlow 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_powerflow> -P <port_number_of_http_on_powerflow> -U <user_name> -p
password
```

where

- `<path_and_name_of_step_file>` is the full pathname for the step.
- `<hostname_or_IP_address_of_powerflow>` is the hostname or IP address of PowerFlow.
- `<port_number_of_http_on_powerflow>` is the port number to access PowerFlow. The default value is 443.
- `<user_name>` is the user name you use to log in to PowerFlow.
- `password` is the password you use to log in to PowerFlow.

To upload an application:

```
iscli -u -a -f <path_and_name_of_application_file>.py -H <hostname_or_IP_
address_of_powerflow> -P <port_number_of_http_on_powerflow> -U <user_name>
-p password
```

where:

- `<path_and_name_of_application_file>` is the full pathname for the application.
- `<hostname_or_IP_address_of_powerflow>` is the hostname or IP address of PowerFlow.
- `<port_number_of_http_on_powerflow>` is the port number to access PowerFlow. The default value is 443.

- *<user_name>* is the user name you use to log in to PowerFlow.
- *password* is the password you use to log in to PowerFlow.

To upload a configuration object:

```
iscli -u -c -f <path_and_name_of_configuration_file>.py -H <hostname_
or_IP_address_of_PowerFlow> -P <port_number_of_http_on_PowerFlow> -U
<user_name> -p <password>
```

where:

- *<path_and_name_of_configuration_file>* is the full pathname for the step.
- *<hostname_or_IP_address_of_PowerFlow>* is the hostname or IP address of the PowerFlow system.
- *<port_number_of_http_on_PowerFlow>* is the port number to access the PowerFlow system. The default value is 443.
- *<user_name>* is the user name you use to log in to the PowerFlow system.
- *<password>* is the password you use to log in to the PowerFlow system.

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.** File path of SyncPacks, PowerFlow applications, steps, and configuration objects to upload. Note that while you can bulk upload applications, steps, and configuration objects, you cannot bulk upload SyncPacks.
- **-c or --configuration.**
- **-a or --application.** Use the iscli tool to interact with an application.
- **-s or --step.** Use the iscli tool to interact with a step.
- **-i or --insecure.** Upload with an insecure (http) connection.
- **-k or --syncpack.** Use the iscli tool to interact with a SyncPack.
- **-A or --activate**. Activate a SyncPack. Requires **-n, -v** tag. **-F** is optional to force the activation and the tag -I to install the SyncPack.
- **-I or --install**. Install a SyncPack. Requires **-n, -v** tag. **-F** is optional to force the installation.
- **-n or --name**. Name of the SyncPack that will be activated.

- **-V or --version**. Version of the SyncPack that will be activated.
- **-F or --force**. Force a SyncPack activation or installation.
- **-O or --overwrite**. Specify if non-SyncPack content will be overwritten.
- **-l or --license**. Use the CLI to license your IS system.
- **-e or --email**. Licensing: Customer Email.
- **-C or --customer**. Licensing: Customer Name.
- **-o or --output**. Licensing: Path to output license file.

You will use the following commands most often:

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

PowerFlow API

PowerFlow includes an API. When you install PowerFlow, the API is available.

Viewing API Documentation

To view the full documentation for the PowerFlow API:

1. From the PowerFlow system, copy the **/opt/iservices/scripts/swagger.yml** file 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 API documentation.

Available Endpoints

POST

/api/v1/apikeys/. Add a new API key.

/api/v1/applications. Add a new application or overwrite an existing application.

/api/v1/applications/{appName}/run. Run a single application by name with saved or provided configurations.

/api/v1/applications/run. Run a single application by name.

/api/v1/configurations. Add a new configuration or overwrite an existing configuration.

/api/v1/roles/owner. Add a new owner assigned a specific role.

/api/v1/schedule. Add a new scheduled PowerFlow application.

/api/v1/status. Runs the "PowerFlow Control Tower HealthCheck" application to generate health status data.

/api/v1/steps. Add a new step or overwrite an existing step.

/api/v1/steps/run. Run a single step by name.

/api/v1/syncpacks/{syncpackName}/install. Install a specific SyncPack version by name.

/api/v1/tasks/{taskId}/replay. Replay a specific PowerFlow application. Replayed applications run with the same application variables, configuration, and queue as the originally executed application.

/api/v1/tasks/{taskId}/revoke. Terminate a specific task or application. By default, this command will not terminate the current running task.

/api/v1/tasks/{appId}/revoke. Terminate all tasks associated with a specific application.

/api/v1/me/widgets/{widget_id}. Creates a new widget or updates a existing widget used on the **PowerFlow Control Tower** page.

GET

/api/v1/about. Retrieve version information about the packages used by this PowerFlow system, including the version of PowerFlow.

/api/v1/apikeys. Retrieve all available API keys saved in the PowerFlow system.

/api/v1/apikeys/{api_key}. Get details of a single API key.

/api/v1/applications. Retrieve a list of all available applications on this PowerFlow system.

/api/v1/applications/{appName}. Retrieve a specific application.

/api/v1/applications/{appName}/logs. Retrieve the logs for the specified application.

/api/v1/cache/{cache_id}. Retrieve a specific cache to gather information about the user interface and the PowerFlow applications.

/api/v1/cache/{cache_key}. Retrieve cache documents, but only if this cache document was explicitly saved to be exposed to the API. You will need to save the cache document using the latest version of the "SaveToCache" step in the *Base Steps* SyncPack. This step has a step parameter called "read_from_api" that lets you decide whether the cache document can be requested from the API.

/api/v1/configurations. Retrieve a list of all configurations on this PowerFlow system.

/api/v1/configurations/{configName}. Retrieve a specific configuration.

/api/v1/license?type=platform. Retrieve license data for this PowerFlow system.

/api/v1/reports. Retrieve a list of paginated reports.

/api/v1/reports/{reportId}. Retrieve a specific report by ID.

/api/v1/roles. Retrieve a list of available roles on this PowerFlow system.

/api/v1/roles/owner. Retrieve a list of roles assigned to owners on this PowerFlow system.

/api/v1/roles/owner/{owner}. Retrieve the role assigned to a specific owner.

/api/v1/sessions. Retrieve a list of sessions for this PowerFlow system.

/api/v1/sessions/status. Retrieve the Session Management status for this PowerFlow system.

/api/v1/sessions/username/{username}. Retrieve the session IDs for a specific user.

/api/v1/sessions/{session_id}. Retrieve a specific session from Session Management for this PowerFlow system.

/api/v1/schedule. Retrieve a list of all scheduled applications on this PowerFlow system.

/api/v1/status. Retrieve all the health status cache documents without running the "PowerFlow Control Tower HealthCheck" application.

/api/v1/status?all=true. Retrieve all health metrics for PowerFlow services and merge all the health status cache documents to return only one JSON response.

/api/v1/status/{service}. Retrieve all health metrics for a specific PowerFlow service, including the following services: **contentapi**, **couchbase**, **dexserver**, **iservices_syncpack_steprunner**, **iservices_syncpacks_ steprunner**, **pfctl_output**, **rabbitmq**, **redis**, **steprunner**.

Starting with PowerFlow version 2.6.0, you can run a `GET api/v1/status` operation that returns all the health status cache documents without running the HealthCheck application.

/api/v1/steps. Retrieve a list of all steps on this PowerFlow system.

/api/v1/steps/{stepName}. Retrieve a specific step.

/api/v1/syncpacks. Retrieve a list of all SyncPacks on this PowerFlow system.

/api/v1/syncpacks/{synpackName}. Retrieve the full details about a specific SyncPack.

/api/v1/syncpacks?only_installed=true. Retrieve a list of only the installed SyncPacks on this system.

/api/v1/syncpacks?only_activated=true. Retrieve a list of only the activated SyncPacks on this system.

/api/v1/tasks/{taskId}. Retrieve a specific task.

/api/v1/webhooks. Retrieve all available webhooks saved in the PowerFlow system.

/api/v1/me/widgets. Returns a list of all installed widgets used on the **PowerFlow Control Tower** page.

/api/v1/me/widgets/{widget_id}. Returns a specific widget using the specified widget ID.

DELETE

/api/v1/apikeys/{api_key}. Delete an API key.

/api/v1/applications/{appName}. Delete a PowerFlow application by name.

/api/v1/cache/{cache_id}. Delete a cache entry by name.

/api/v1/configurations/{configName}. Delete a configuration by name.

/api/v1/license?type=platform. Delete license data for this PowerFlow system.

/api/v1/me/widgets/{widget_id}. Delete the specified widget used on the **PowerFlow Control Tower** page.

/api/v1/roles/owner. Delete a specific owner role.

/api/v1/schedule. Delete a scheduled PowerFlow application by ID.

/api/v1/sessions. Delete a list of sessions for this PowerFlow system.

/api/v1/sessions?all=true. Delete all sessions for this PowerFlow system.

/api/v1/sessions/status. Delete the Session Management status for this PowerFlow system.

/api/v1/sessions/username/{username}. Delete the session IDs for a specific user.

/api/v1/sessions/{session_id}. Delete a specific session from Session Management for this PowerFlow system.

/api/v1/reports/{appName}. Delete a specific report by name.

/api/v1/reports/{reportId}. Delete a specific report by report ID.

/api/v1/steps/{stepName}. Delete a specific step by name.

/api/v1/syncpacks/{spName}. Delete a specific SyncPack by name.

ScienceLogic