Connectors - SDK

Connectors - SDK

The **Purple Fabric's SDK** empowers the users, developers and subject-matter experts to build, integrate, and extend agentic AI solutions within the Purple Fabric ecosystem. It brings the platform's powerful building blocks directly into your applications.

Overview

The Connector module enables seamless integration between third-party tools and the Platform. This integration pattern facilitates data exchange from tools like Google Drive (GDrive), Amazon S3, Notion, and more. The Platform provides a suite of pre-built (Out-of-the-Box or OOTB) connectors, as well as the flexibility for clients to build custom connectors. Developers can leverage the Software Development Kit (SDK) provided by the Platform to create these connectors. The design documentation will guide developers through the SDK lifecycle, connector creation journey, and deployment options.

We currently offer support for two types of Connector journeys, as outlined in below table:

Connector Type	**Description**
OOTB Connectors	Connectors built and maintained by the PF team
Custom Connectors	Connectors built by the tenant, maintained by the respective tenant, and published to a specific workspace

Design Checklist

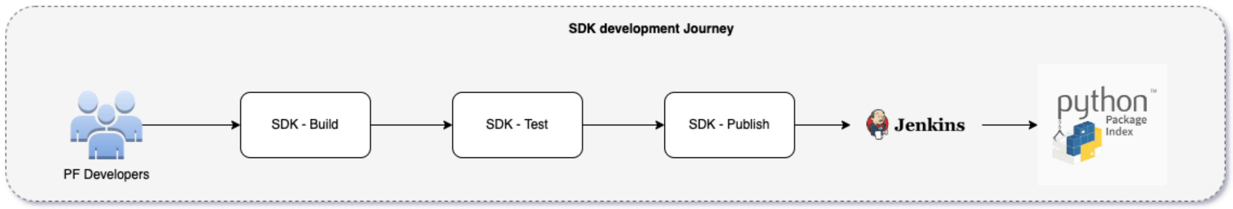
Feature	**Priority**	**Description**
SDK download	P1	Users should be able to download the SDK for integration and development
OOTB Connector Creation	P1	The Platform (PF) team must have the ability to create(build, test, publish) Out-Of-The-Box (OOTB) connectors for seamless integration
Custom Connector Creation	P1	Tenants or non-PF team members should be able to create and deploy their own custom connectors
NFR Metrics Visibility	P2	Users should have access to Non-Functional Requirement (NFR) metrics to monitor the

Feature	Priority	Description
		performance and reliability of published connectors

SDK Lifecycle

The SDK lifecycle defines the process of developing, testing, building, and publishing new versions of the SDK. The SDK provides developers with the necessary tools and libraries to create custom connectors, ensuring consistency and ease of use.

Note:
The connector SDK currently supports only Python, with plans to expand support to other languages in the future.



Note:

- SDK source code would be checked in iGIT
- Create PyPi account for PF/IntellectAI

Below is the typical Python SDK development flow:

- **SDK - Build**
  Developers write and organize the SDK code, which is stored in version control (i.e., iGit repositories). The SDK code is designed to be flexible, so developers can use it to create connectors for various third-party tools

- **SDK - Test**
  Once the code is built, it undergoes rigorous testing to ensure its functionality and integration capabilities. This phase involves verifying that the SDK interacts as expected with external services and handles data correctly

- **SDK - Publish**
  After passing testing, the SDK is packaged and published. The new SDK version is made available for integration with the platform and for client use. A PyPi account is created for the Platform (PF/IntellectAI) to distribute the SDK.

- **Jenkins Integration**
  Jenkins plays a key role in automating the SDK publishing process. Once the SDK passes

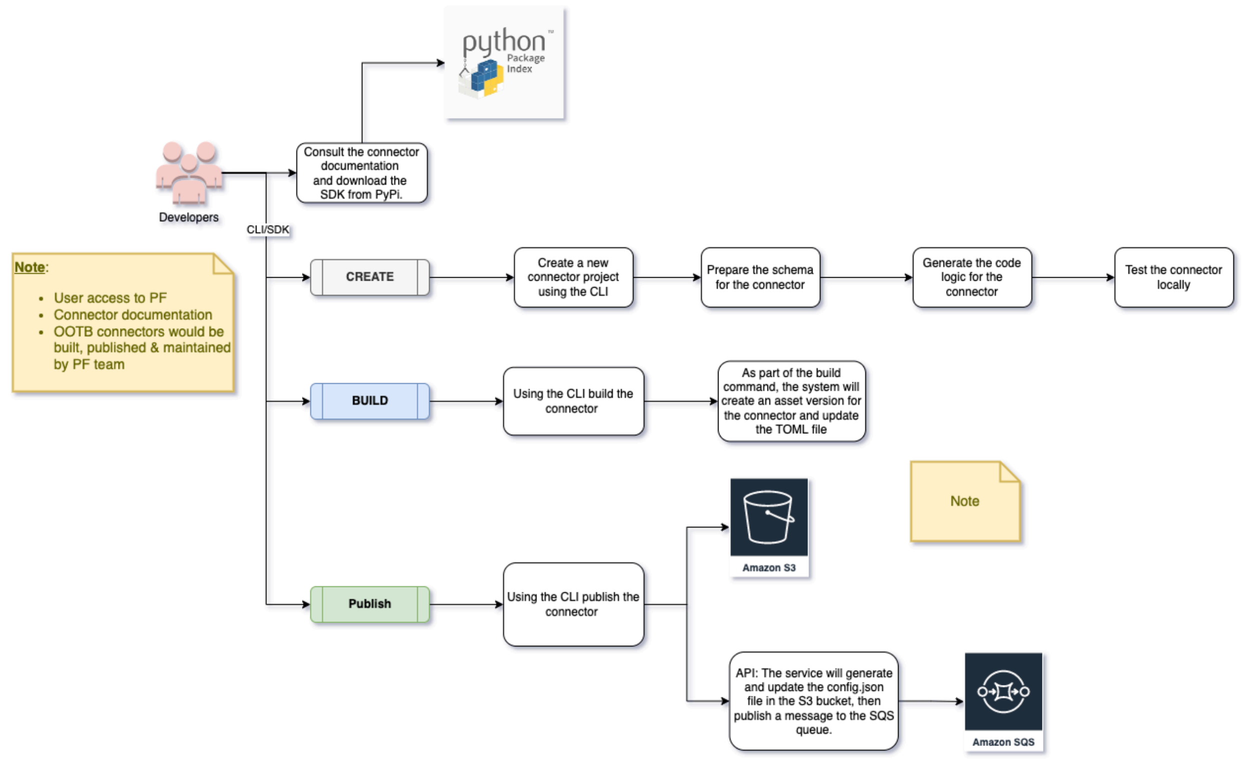
the build and test phases, Jenkins automates the publishing of the SDK package to PyPi, where it can be downloaded and integrated into new connectors.

- **Connector Deployment**
  Once the SDK is available, developers can use it to build custom connectors. After development, these connectors are deployed through various pipeline options, which are discussed in the next section.

Connector Build Journey Using SDK

The Platform provides an SDK with a Command Line Interface (CLI) to facilitate the development of custom connectors. Using the SDK, developers can create, build, test, and publish connectors that integrate third-party services (e.g., Google Drive, Amazon S3, Notion) with the Platform.

This section outlines the step-by-step process for creating a connector using the SDK and the CLI.


python Package Index
Developers
Consult the connector documentation and download the SDK from PyPi.
CLI/SDK
Note:
User access to PF
Connector documentation
OOTB connectors would be built, published & maintained by PF team
CREATE
Create a new connector project using the CLI
Prepare the schema for the connector
Generate the code logic for the connector
Test the connector locally
BUILD
Using the CLI build the connector
As part of the build command, the system will create an asset version for the connector and update the TOML file
Amazon S3
Note
Publish
Using the CLI publish the connector
API: The service will generate and update the config.json file in the S3 bucket, then publish a message to the SQS queue.
Amazon SQS

Note:

- **OOTB Connectors** are built, published, and maintained by the PF team
- Developers must have **user access** to the Platform
- The CLI supports additional commands such as **help**, **nfr-report**, and **deploy**

Step 1: Consult the Documentation & Download the SDK

Developers begin by referring to the **Connector** Documentation, which provides guidelines for building connectors. The SDK is available on **PyPi**, and developers can install it using:

pip install platform-sdk

Step 2: Create a New Connector Project
To create a new connector, developers use the CLI to generate a project structure.

pf-cli create [OPTIONS]

Actions Performed:
- A new connector project is initialized
- The required folder structure and configuration files are generated
- The developer can then define the schema for data ingestion and processing

Step 3: Define the Schema & Implement Code Logic
Once the project is created, developers:
1. Prepare the Schema: Define the data model, API endpoints, and authentication requirements
2. Generate the Code Logic: Implement the connector logic, handling API requests and data transformations
3. Test Locally: Before deployment, the connector should be tested locally

Step 4: Build the Connector
Once development is complete, the next step is to build the connector:

pf-cli build [OPTIONS]

Actions Performed:
- The CLI validates the connector code and dependencies
- A build artifact is generated

Step 5: Publish the Connector
To deploy the connector to the Platform, developers use:

pf-cli deploy [OPTIONS]

Actions Performed:
- The connector is uploaded to Amazon S3, where its configuration (config.json) is stored. Following is the structure in our storage repository:

```
Bucket Name/
└── Asset Name/
    └── Version/
```

```
├── code.zip
├── config.json
└── nfr_report.json
```

- A message is sent to Amazon SQS, triggering the connector registration process.

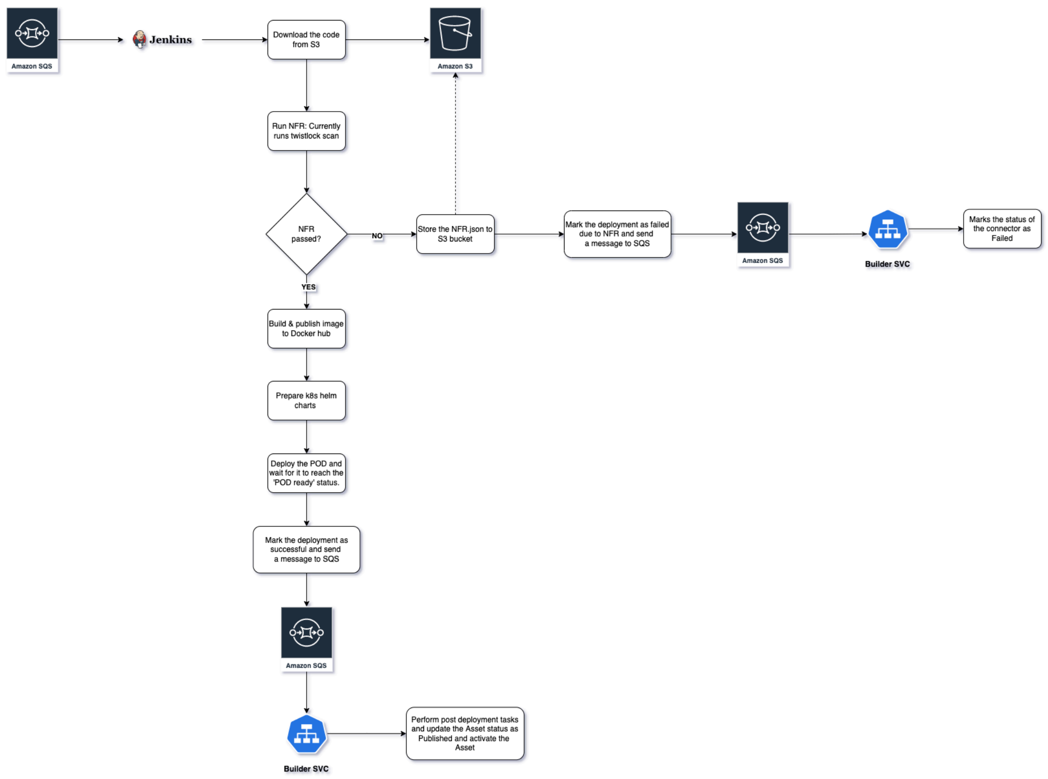
Step 6: Deployment & Integration
After publishing, the connector is available for use within the Platform. Developers can configure and monitor it through the UI.

Connector Deployment Journey

Note: The default SDK option currently utilizes Docker and generates a Docker image as part of the deployment pipeline. A proof of concept (POC) is needed to evaluate the deployment strategy (Docker vs Lambda vs Runtime). The results of this POC will guide the redefinition of the available deployment options.

The deployment of a connector follows a structured pipeline that ensures security, compliance, and reliability. The process involves **Jenkins, Amazon S3, Docker Hub, Kubernetes (K8s) Helm Charts,** and **Amazon SQS** for messaging.

This section details how a connector is published and deployed, including Non-Functional Requirement (NFR) validation and failure handling mechanisms.


Amazon SQS
Jenkins
Download the code from S3
Amazon S3
Run NFR: Currently runs twistlock scan
NFR passed?
NO
Store the NFR.json to S3 bucket
Mark the deployment as failed due to NFR and send a message to SQS
Amazon SQS
Builder SVC
Marks the status of the connector as Failed
YES
Build & publish image to Docker hub
Prepare k8s helm charts
Deploy the POD and wait for it to reach the 'POD ready' status.
Mark the deployment as successful and send a message to SQS
Amazon SQS
Builder SVC
Perform post deployment tasks and update the Asset status as Published and activate the Asset

Step 1: Initiating Deployment

- A **message is sent to Amazon SQS** to trigger the deployment process
- **Jenkins** picks up the job and starts the deployment workflow

Step 2: Code Download & NFR Validation

- Jenkins **downloads the connector code from the platform's internal storage** (currently it is S3).

```
Bucket Name/
└── Asset Name/
    └── Version/
        ├── code.zip
        ├── config.json
        └── nfr_report.json
```

- It runs the **NFR validation process**, which currently includes a **Twistlock scan** to check for security vulnerabilities.

Step 3: NFR Validation Check

- The deployment pipeline executes **Twistlock** scans
- If **NFR fails**, the deployment is **halted**, and:
  - The **NFR report (JSON)** is stored in the **S3 bucket**
  - A failure message is sent to **Amazon SQS**, which triggers **Builder SVC** to update the connector's status as **Failed**
- If **NFR passes**, the report would be stored in S3 bucket and the deployment proceeds
- The user can run the NFR report through the CLI to retrieve the NFR metrics for the corresponding connector

Step 4: Building & Publishing the Connector

- Jenkins **builds and publishes a Docker image** of the connector to **Docker Hub**
- The Kubernetes Helm Charts are prepared for deployment

Step 5: Deploying the Connector

- The connector's **POD is deployed** in Kubernetes
- The system waits for the POD to reach the **"POD ready"** status

Step 6: Deployment Status Update

- If deployment **fails**, a failure message is sent to **Amazon SQS**, and the **Builder SVC** marks the status as **Failed**
- If deployment **succeeds**, the system:
  - Sends a success message to **Amazon SQS**
  - The **Builder SVC** updates the **Asset status to "Published"** and activates the connector

Tools & Technologies Used

Tool	**Purpose**
Amazon SQS	Message queue for triggering and tracking deployment statuses.
Jenkins	Orchestrates the deployment pipeline.
Amazon S3	Stores connector artifacts and NFR reports.
Twistlock	Scans for security vulnerabilities in the code.
Docker Hub	Stores and manages containerized connector images.
Kubernetes Helm Charts	Manages deployment configurations.
Builder SVC	Updates the connector's status post-deployment.

Failure Handling & Logging

- **NFR Failure:** The deployment stops, and failure details are logged in **S3** and sent to **SQS**
- **Deployment Failure:** The connector status is updated as **Failed** in **Builder SVC**

Final Outcome

Once successfully deployed, the connector is available for use within the Platform. If any failures occur, logs and reports are available for debugging.

Refer to the following pages to know more:

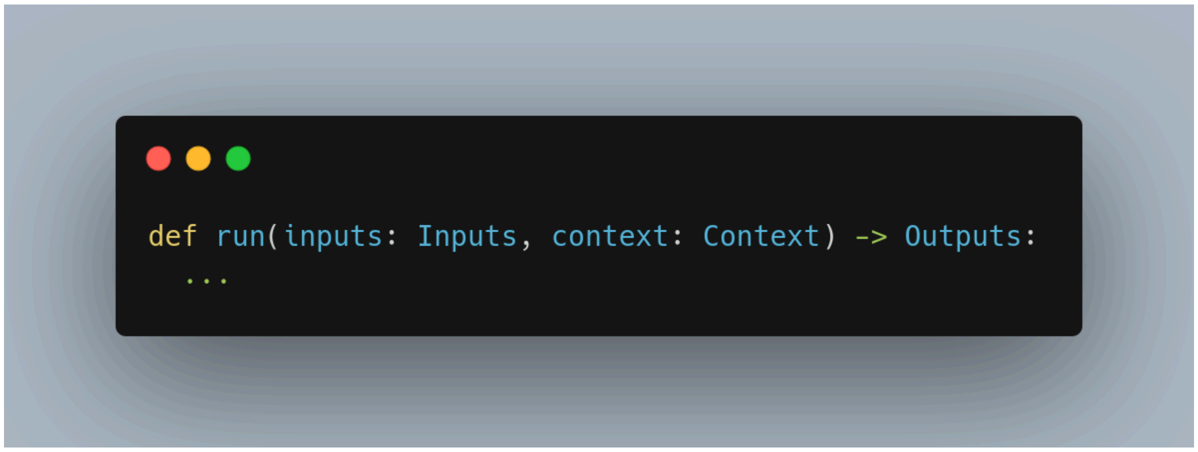
1. What Purple Fabric's SDK can do for you?
2. Powers of the CLI
3. SDK - Getting Started
4. The schema.json file
5. Examples of Production-ready Connectors
6. Frequently Asked Questions (FAQs)

What Purple Fabric’s SDK can do for you?

What Purple Fabric's SDK can do for you?

Overview

- The SDK expects you to implement the simple run() function to achieve your goals!
- SDK enables you to bring in custom dependencies at OS level as well as at project level
- SDK provides the Context object that can provide useful operations for the developer
- The developer can build the connector with type-safe Pydantic models generated from the schema



```python
def run(inputs: Inputs, context: Context) -> Outputs:
  ...
```

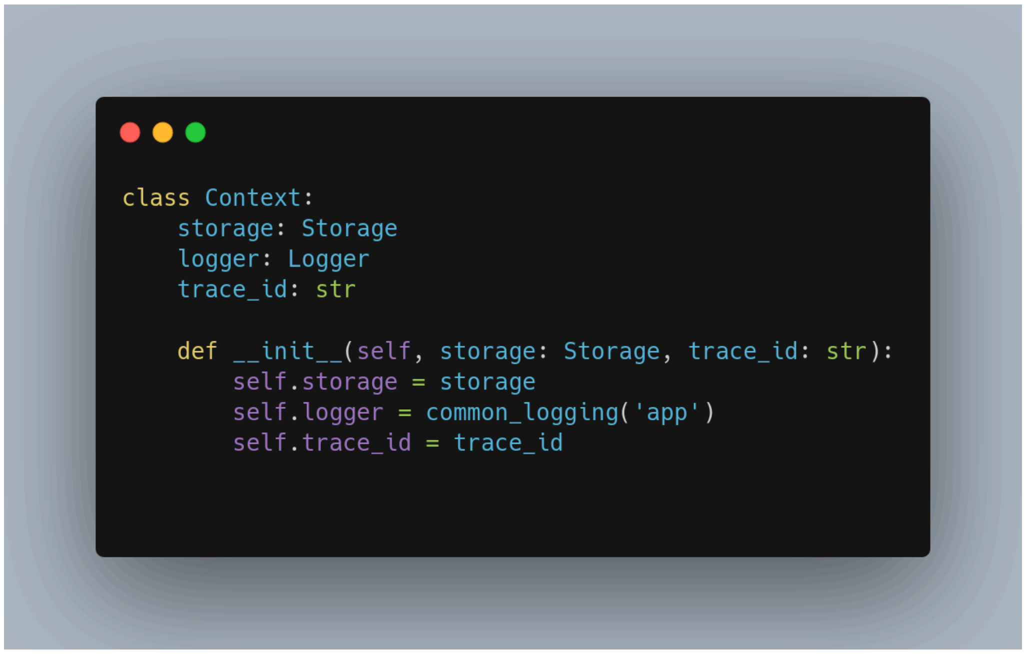
The run() function

The Context class

The **Context** class enables the developer to perform the following operations:

- **storage:** Upload and download files from Purple Fabric Platform

- **logger:** Provides a logging.Logger instance configured with sensible defaults**trace_id:** Provides the current trace_id used for tracking of requests across Purple Fabric Platform

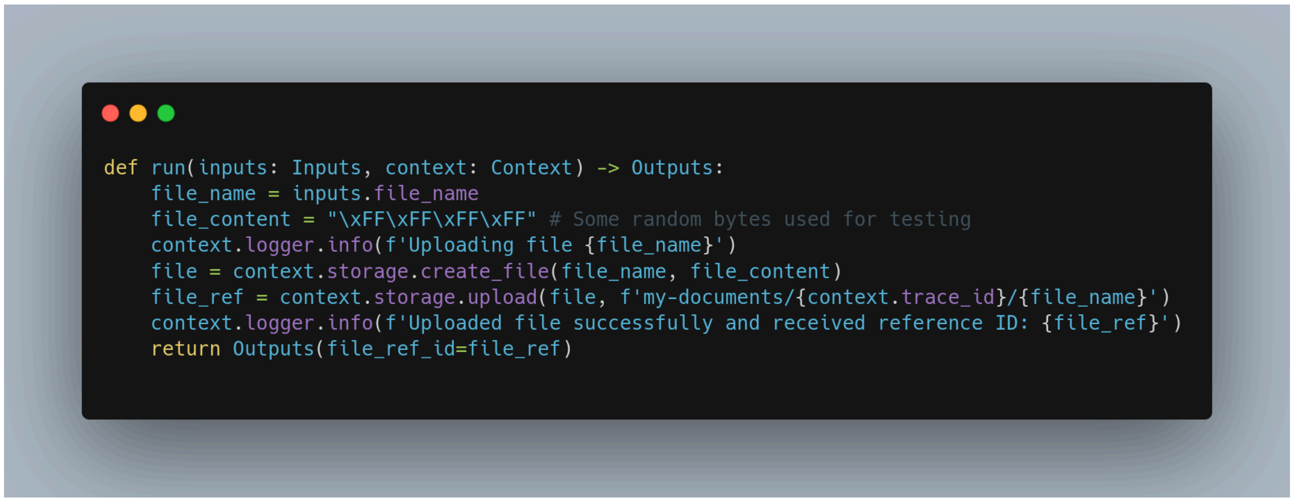


```python
class Context:
    storage: Storage
    logger: Logger
    trace_id: str

    def __init__(self, storage: Storage, trace_id: str):
        self.storage = storage
        self.logger = common_logging('app')
        self.trace_id = trace_id
```

Context Class

Uploading files to Purple Fabric Platform

- To upload a file to the Purple Fabric Platform, create an instance of **File**. It takes **file_name** and **file_content** as its arguments
- Use the **context.storage.upload()** function to upload the file. It provides a reference ID, that can be used to track the location of the file
- The **upload** function is capable of uploading a single file per call

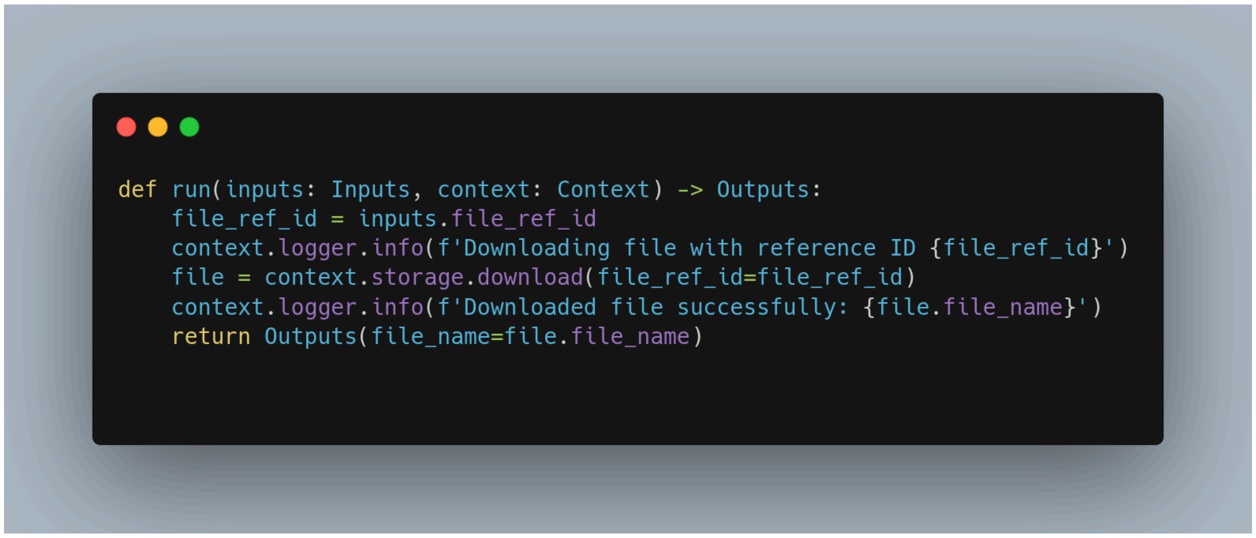


```python
def run(inputs: Inputs, context: Context) -> Outputs:
    file_name = inputs.file_name
    file_content = "\xFF\xFF\xFF\xFF" # Some random bytes used for testing
    context.logger.info(f'Uploading file {file_name}')
    file = context.storage.create_file(file_name, file_content)
    file_ref = context.storage.upload(file, f'my-documents/{context.trace_id}/{file_name}')
    context.logger.info(f'Uploaded file successfully and received reference ID: {file_ref}')
    return Outputs(file_ref_id=file_ref)
```

Sample code for uploading files

Downloading files from Purple Fabric Platform

- To download a file from Purple Fabric Platform, the user has to provide the **file_ref_id** parameter
- The **context.storage.download()** function is capable of downloading a single file per call
- The downloaded file is an instance of **File** model



```
def run(inputs: Inputs, context: Context) -> Outputs:
    file_ref_id = inputs.file_ref_id
    context.logger.info(f'Downloading file with reference ID {file_ref_id}')
    file = context.storage.download(file_ref_id=file_ref_id)
    context.logger.info(f'Downloaded file successfully: {file.file_name}')
    return Outputs(file_name=file.file_name)
```

Powers of the CLI

Powers of the CLI

The SDK's CLI provides the following commands:

1. **create** command

Command:

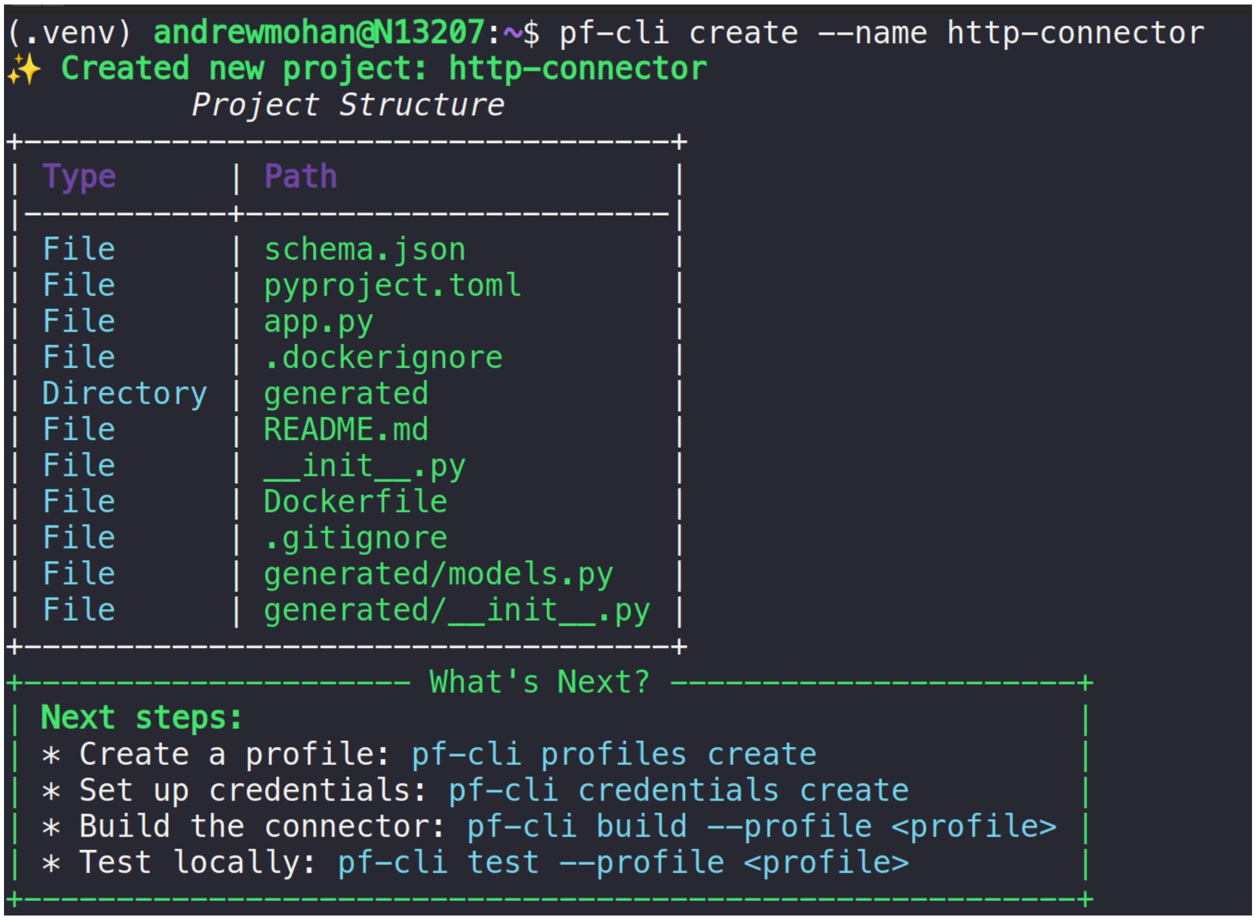
```
Usage: pf-cli create [OPTIONS]

  Create a new project with the specified name.

Options:
  --name TEXT                   Name of the project
  --current-dir                 Create the project in the current directory
  --asset-category [INTEGRATOR|TOOLS]
                                Category of the asset (default: INTEGRATOR)
  --asset-description TEXT      Description of the asset
  --help                        Show this message and exit
```

Description

- Creates a new project with the provided options
- Below is the project structure of the generated project

```
.
└── hello-world-connector/
    ├── generated/
    │   ├── __init__.py
    │   └── models.py
    ├── app.py
    ├── __init__.py
    ├── pyproject.toml
    ├── schema.json
    ├── README.md
    ├── Dockerfile
├── .dockerignore
└── .gitignore
```

Screenshot



```
(.venv) andrewmohan@N13207:~$ pf-cli create --name http-connector
✨ Created new project: http-connector
          Project Structure
+-----------------------------------+
| Type      | Path                  |
|-----------+-----------------------|
| File      | schema.json           |
| File      | pyproject.toml        |
| File      | app.py                |
| File      | .dockerignore         |
| Directory | generated             |
| File      | README.md             |
| File      | __init__.py           |
| File      | Dockerfile            |
| File      | .gitignore            |
| File      | generated/models.py   |
| File      | generated/__init__.py |
+-----------------------------------+
+---------------------- What's Next? -----------------------+
| Next steps:                                               |
| * Create a profile: pf-cli profiles create                |
| * Set up credentials: pf-cli credentials create           |
| * Build the connector: pf-cli build --profile <profile>   |
| * Test locally: pf-cli test --profile <profile>           |
+-----------------------------------------------------------+
```

2. **run** command

Command

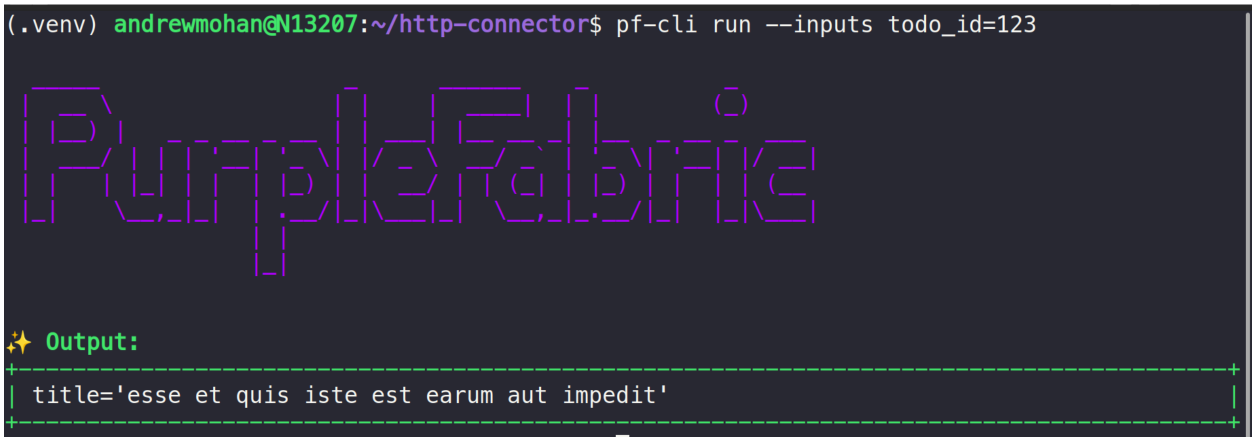
```
Usage: pf-cli run [OPTIONS]

  Run the project in local using the specified inputs.

Options:
  --inputs TEXT       Input key-value pairs (e.g., --inputs name="Purple
                      Fabric" --inputs secrets="credentials-123")
  --inputs-file PATH  Path to a JSON file containing inputs (e.g., --inputs-
                      file inputs.json)
  --help              Show this message and exit.
```

Description

Executes the connector locally by invoking the run function inside app.py.

User can pass arguments as key-value pairs via the CLI, or they can pass the arguments inside a JSON file and pass the JSON file as input.

Screenshot



```
(.venv) andrewmohan@N13207:~/http-connector$ pf-cli run --inputs todo_id=123

  ____                   _      _____     _          _
 |  _ \ _   _ _ __ _ __ | | ___|  ___|_ _| |__  _ __(_) ___
 | |_) | | | | '__| '_ \| |/ _ \ |_ / _` | '_ \| '__| |/ __|
 |  __/| |_| | |  | |_) | |  __/  _| (_| | |_) | |  | | (__
 |_|    \__,_|_|  | .__/|_|\___|_|  \__,_|_.__/|_|  |_|\___|
                  |_|

✨ Output:
+------------------------------------------------------------------------------------------+
| title='esse et quis iste est earum aut impedit'                                          |
+------------------------------------------------------------------------------------------+
```

3. **profiles** command

Command

Usage: pf-cli profiles [OPTIONS] COMMAND [ARGS]...

Manage user profiles.

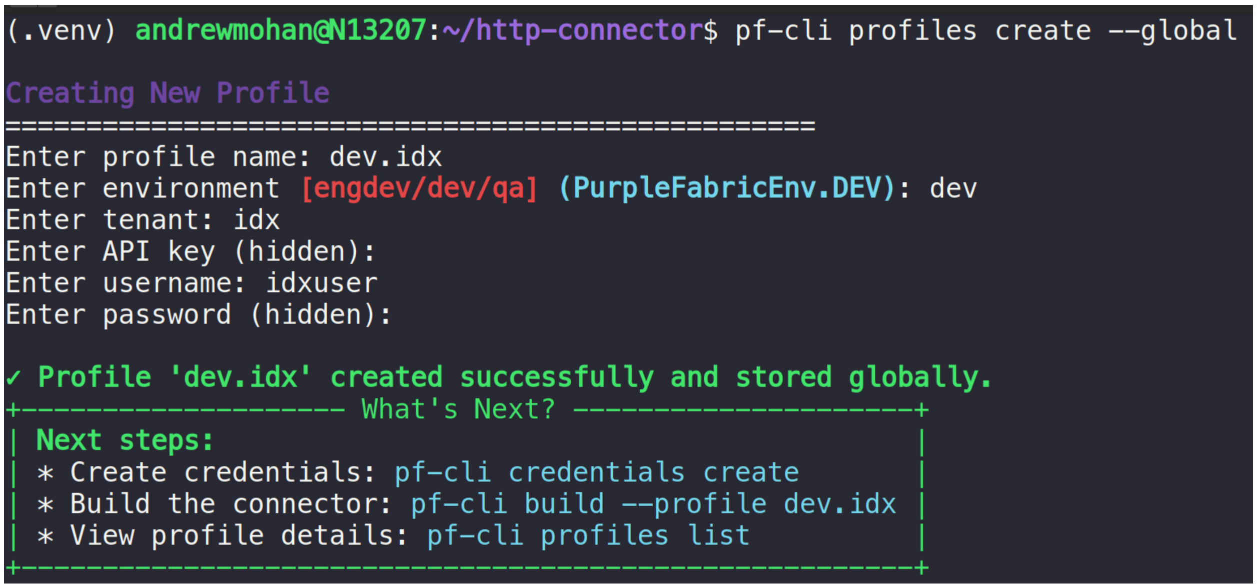
Options:
--help Show this message and exit.

Commands:
create Create a new profile.
delete Delete a profile.
list   List profiles.
update Update an existing profile.

Description

- A profile consists of the Purple Fabric Platform’s login credentials used by the developer
- Profiles can be created in global scope or in local scope
  - Global profiles can be used by all projects
  - Local profiles can only be used in the project it was created
- Profiles play an integral role in promoting the connector to the Purple Fabric Platform

Screenshot



```
(.venv) andrewmohan@N13207:~/http-connector$ pf-cli profiles create --global

Creating New Profile
==================================================
Enter profile name: dev.idx
Enter environment [engdev/dev/qa] (PurpleFabricEnv.DEV): dev
Enter tenant: idx
Enter API key (hidden):
Enter username: idxuser
Enter password (hidden):

✓ Profile 'dev.idx' created successfully and stored globally.
+-------------------- What's Next? ---------------------+
| Next steps:                                          |
| * Create credentials: pf-cli credentials create      |
| * Build the connector: pf-cli build --profile dev.idx |
| * View profile details: pf-cli profiles list         |
+------------------------------------------------------+
```

4. **credentials** command

Command

Usage: pf-cli credentials [OPTIONS] COMMAND [ARGS]...

Manage user credentials.

When retrieving credentials, local credentials are checked first. If not found locally, global credentials will be checked as a fallback.

Options:
--help Show this message and exit.

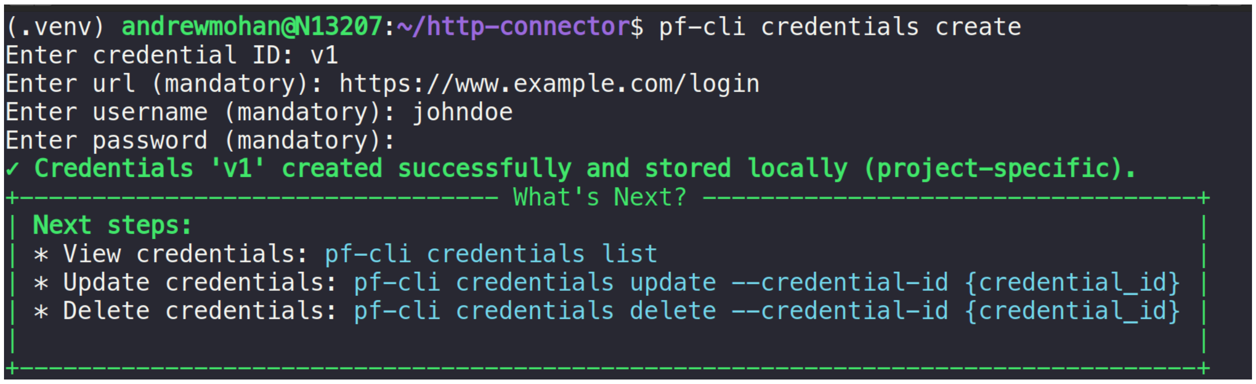
Commands:
create Create new credentials.
delete Delete credentials.
list   List all stored credential IDs.
update Update existing credentials.

Description

- If a connector requires usage of sensitive information like credentials, the SDK supports managing the credentials locally
- Credentials are defined by the **schema.json**
- Credentials are always locally encrypted
- Credentials can be created in local scope or in global scope
  - Global credentials can be used by any projects, provided that they have exactly identical fields

    ○ Local credentials can only be used in the current project. This is the recommended way to use credentials.

Screenshot



```
(.venv) andrewmohan@N13207:~/http-connector$ pf-cli credentials create
Enter credential ID: v1
Enter url (mandatory): https://www.example.com/login
Enter username (mandatory): johndoe
Enter password (mandatory):
✓ Credentials 'v1' created successfully and stored locally (project-specific).
+---------------------------------- What's Next? -----------------------------------+
| Next steps:                                                                       |
| * View credentials: pf-cli credentials list                                       |
| * Update credentials: pf-cli credentials update --credential-id {credential_id}  |
| * Delete credentials: pf-cli credentials delete --credential-id {credential_id}  |
|                                                                                   |
+-----------------------------------------------------------------------------------+
```

5. **build** command

Command

Usage: pf-cli build [OPTIONS]

Build the project using the specified profile.

The profile will be searched for in the local scope first, then in the global scope if not found locally.

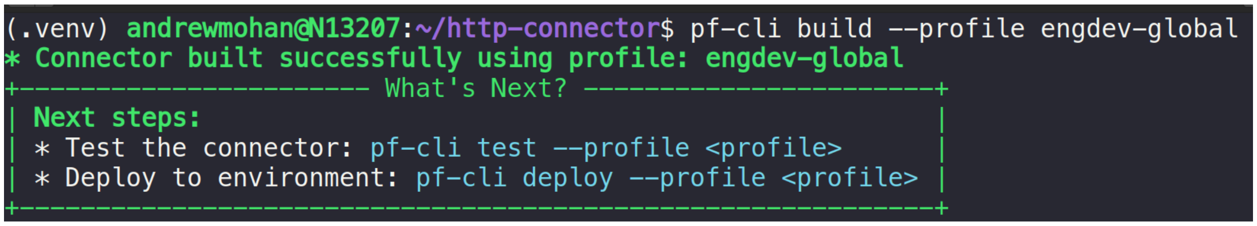
Options:

```
  --profile TEXT  Name of the profile to use for building the connector (e.g.,
                  --profile my-profile)  [required]
  --help          Show this message and exit.
```

Description

- This is the first stage of 2 stages to promote your connector to Purple Fabric Platform
- In this stage, the Platform will allocate the necessary identifiers, schemas and performs validations to ensure that your connector is developed with the proper schema
- The profile determines the destination where the connector will be later published
- It is imperative that **build** operation must be executed at-least once.

Screenshot



```
(.venv) andrewmohan@N13207:~/http-connector$ pf-cli build --profile engdev-global
* Connector built successfully using profile: engdev-global
+------------------------- What's Next? -------------------------+
| Next steps:                                                    |
| * Test the connector: pf-cli test --profile <profile>         |
| * Deploy to environment: pf-cli deploy --profile <profile>    |
+----------------------------------------------------------------+
```

6. **deploy** command

Command

Usage: pf-cli deploy [OPTIONS]

Deploy the project using the specified profile.

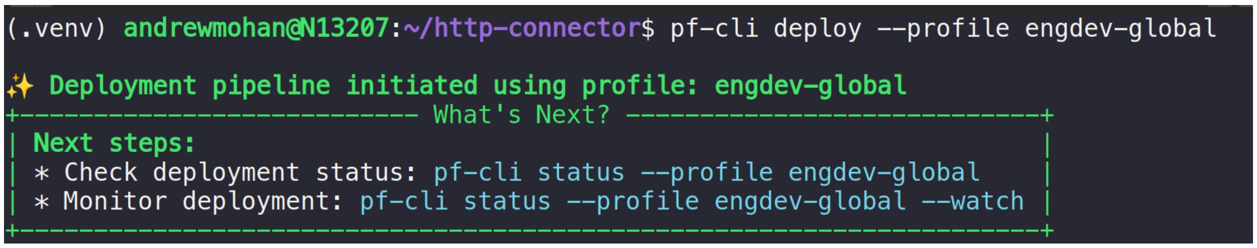
This command initiates the deployment pipeline. Use 'status' to check deployment progress.

Options:
```
  --profile TEXT  Name of the profile to use for deployment  [required]
  --help          Show this message and exit.
```

Description

- This is the second stage of 2 stages to promote your connector to Purple Fabric Platform
- In this stage, your connector project will be converted to an artifact and then sent for deployment to the Purple Fabric Platform
- The profile determines the destination where the connector will be deployed
- It is imperative that the **build** operation must be executed before running this command
- Once the deployment is initiated, you can monitor the deployment status using the **status** command

Screenshot



```
(.venv) andrewmohan@N13207:~/http-connector$ pf-cli deploy --profile engdev-global

✨ Deployment pipeline initiated using profile: engdev-global
+--------------------------- What's Next? ----------------------------+
| Next steps:                                                          |
| * Check deployment status: pf-cli status --profile engdev-global     |
| * Monitor deployment: pf-cli status --profile engdev-global --watch  |
+----------------------------------------------------------------------+
```

7. **status** command

Command

Usage: pf-cli status [OPTIONS]

Check the status of the deployment.

Args: profile (str): Name of the profile to use for checking deployment status   watch (bool): Whether to continuously watch the deployment status

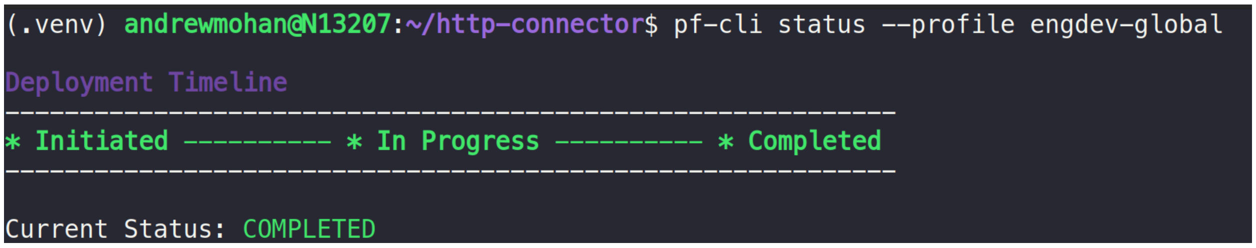
Options:
```
  --profile TEXT  Name of the profile to use for checking deployment status
```

```
              [required]
  --watch        Watch the deployment status by polling every 10 seconds
  --help         Show this message and exit.
```

Description

- This command provides the real-time status of the deployment pipeline
- If the deployment fails due to any reason, the timeline will contain a “Failed” state
- You can use the **watch** option to periodically check the deployment status

Screenshot



```
(.venv) andrewmohan@N13207:~/http-connector$ pf-cli status --profile engdev-global

Deployment Timeline
----------------------------------------------------------
* Initiated ---------- * In Progress ---------- * Completed
----------------------------------------------------------

Current Status: COMPLETED
```

8. **generate** command

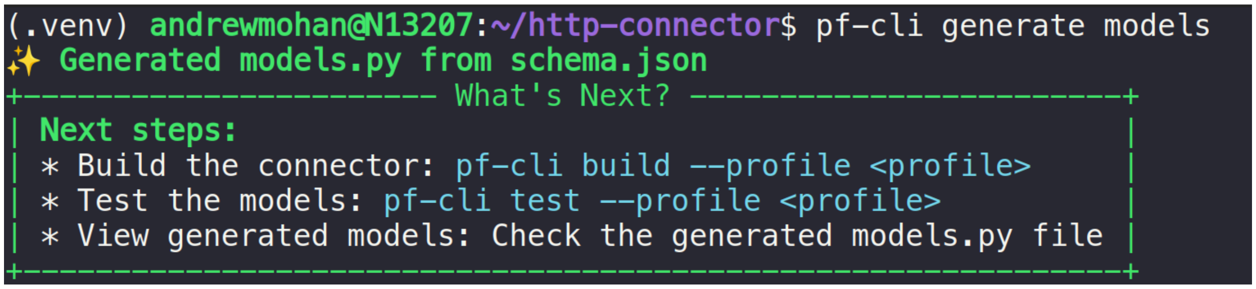
Command

```
Usage: pf-cli generate [OPTIONS] COMMAND [ARGS]...

  Generate project artifacts.

Options:
  --help  Show this message and exit.

Commands:
  models  Generate Pydantic models from schema.json.
```

Description

- This command contains sub-commands that can generate project artifacts on the fly
- Today, it contains a single sub-command called **models** that generates Pydantic models from the definition provided in **schema.json**
- These models can be used in the **app.py** to develop the connector with type-safeness

Screenshot



```
(.venv) andrewmohan@N13207:~/http-connector$ pf-cli generate models
✨ Generated models.py from schema.json
+------------------------ What's Next? ------------------------+
| Next steps:                                                   |
| * Build the connector: pf-cli build --profile <profile>       |
| * Test the models: pf-cli test --profile <profile>            |
| * View generated models: Check the generated models.py file   |
+---------------------------------------------------------------+
```

9. **version** command

Command

Usage: pf-cli version [OPTIONS]

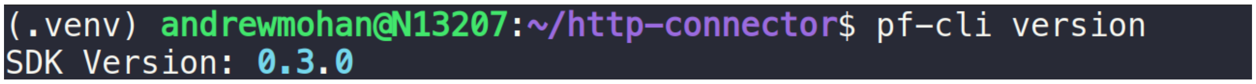
Output the SDK version.

Options:
--help Show this message and exit.

Description

- Provides the current SDK version used by the project

Screenshot



```
(.venv) andrewmohan@N13207:~/http-connector$ pf-cli version
SDK Version: 0.3.0
```

Note: The above output is only for illustration. The version can change based on the installed SDK version.

10. **help** command

Command

Usage: pf-cli help [OPTIONS]

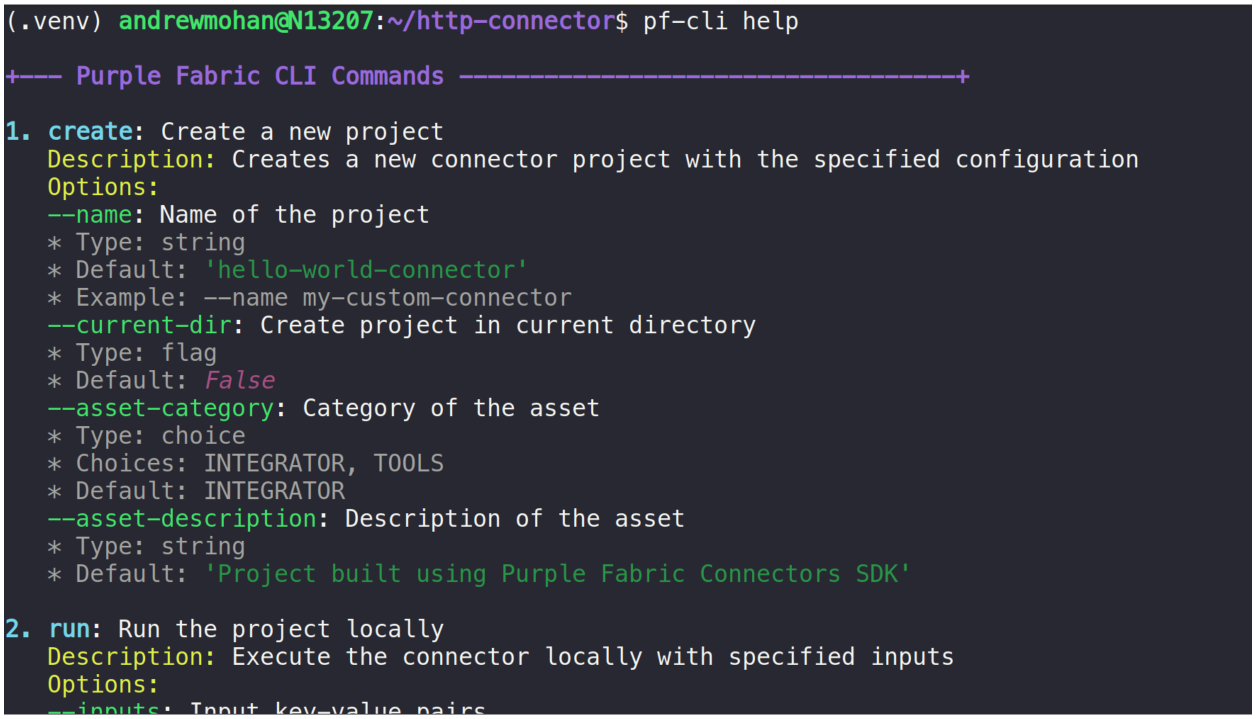
Show help information for available commands.

Options:
--help Show this message and exit.

Description

- Provides helpful information about all the available commands in the SDK.

Screenshot



```
(.venv) andrewmohan@N13207:~/http-connector$ pf-cli help

+---- Purple Fabric CLI Commands ----------------------------------------+

1. create: Create a new project
   Description: Creates a new connector project with the specified configuration
   Options:
   --name: Name of the project
   * Type: string
   * Default: 'hello-world-connector'
   * Example: --name my-custom-connector
   --current-dir: Create project in current directory
   * Type: flag
   * Default: False
   --asset-category: Category of the asset
   * Type: choice
   * Choices: INTEGRATOR, TOOLS
   * Default: INTEGRATOR
   --asset-description: Description of the asset
   * Type: string
   * Default: 'Project built using Purple Fabric Connectors SDK'

2. run: Run the project locally
   Description: Execute the connector locally with specified inputs
   Options:
   --inputs: Input key-value pairs
```

SDK - Getting Started

SDK - Getting Started

Perform the following steps to get started with SDK:

Setup SDK in your system

Here are the requirements for SDK installation and connector development:

- Python version 3.12 and above
- A valid, active user account registered with Purple Fabric platform. You must have the following policies assigned to your account:
    - GenAI_User
- Access for the **CONNECTOR_REPO_USERNAME** and **CONNECTOR_REPO_PASSWORD** credentials in order to download the SDK
    - To request access for these credentials, please fill the below Google form. The credentials will be sent via e-mail https://docs.google.com/forms/d/e/1FAIpQLSfnAfI1ICpZa3AkfWkRs91W0JfC-Go5BvVn7I1jfn-3IznUVg/viewform?usp=sharing
- A minimum disk space of 150 MB.

Follow the below steps to install the SDK:

- Download the installer file for your respective Operating System and Shell:
    - Linux / Mac: install.sh
    - Windows (Command Prompt): install.cmd
    - Windows (PowerShell): install.ps1
- Execute the installer file according to your Operating System and the shell:
    - Linux / Mac:
        - Make the installer executable using the command **chmod +x install.sh**
        - Setup the environment variables using the commands:
          export CONNECTOR_REPO_USERNAME=<provide username here>
          export CONNECTOR_REPO_PASSWORD=<provide password here>
        - Run the executable using **bash install.sh** (if you are using bash)
        - For any other shell, execute the **install.sh** as required by the shell
        - You may have to restart the terminal in order for changes to take effect
    - Windows (Command Prompt):
        - Open a Command Prompt in the directory where **install.cmd** is located
        - Setup the environment variables using the commands:
          set CONNECTOR_REPO_USERNAME=<provide username here>
          set CONNECTOR_REPO_PASSWORD=<provide password here>

        - Execute the command: **.\install.cmd**
    - Windows (PowerShell):
        - Open a PowerShell window in the directory where install.ps1 is located
        - Setup the environment variables using the commands:
          set CONNECTOR_REPO_USERNAME=<provide username here>
          set CONNECTOR_REPO_PASSWORD=<provide password here>
        - Execute the command **.\install.ps1**
- Finally, verify whether the command pf-cli is now available by executing it:
  pf-cli

Follow the below steps to uninstall the SDK:

- Uninstalling SDK can be helpful in order to re-install the SDK with the proper configuration.
- Execute the installer file according to your Operating System and the shell:
    - Linux / Mac: uninstall.sh
    - Windows (Command Prompt): uninstall.cmd
    - Windows (PowerShell): uninstall.ps1
- Follow the steps used in the previous section to run the executable file according to your Operating System and shell

Note:

- The uninstaller scripts only uninstalls **pf-cli** command and all installed SDK versions
- It does not remove any connector projects created by the **pf-cli** command
- Nor does it remove any Purple Fabric profiles and credentials stored in the system

Create a new Connector Project

- Run the following command to create a new connector project:
  pf-cli create --name <connector_name>
- Alternatively, create the connector project in the current directory
  pf-cli create —-current-dir

Note:
If **pf-cli** is being shown as an invalid command / not found, ensure that you have properly installed the SDK in the previous steps

- A sample project will be generated as per the folder structure as shown in the right side
- Once the project is created, install the dependencies in a new virtual environment, by following the below steps:

```
cd <connector_name>
python3.12 -m venv .venv
source .venv/bin/activate
pip install -e .
--extra-index-url=https://${CONNECTOR_REPO_USERNAME}:${CONNECTOR_REPO_PASSW
```

ORD}@iainexus.intellectseeccloud.com/repository/idxplatform-connectors-py-hosted/simple --verbose

Project Structure of a sample connector / tool

```
└── hello-world-connector/
    ├── .config/
    ├── generated/
    │   ├── __init__.py
    │   └── models.py
    ├── app.py
    ├── __init__.py
    ├── pyproject.toml
    ├── schema.json
    ├── README.md
    ├── Dockerfile
    ├── .dockerignore
    └── .gitignore
```

Connectors and Tools

- During project creation, you can pass an additional argument called **--asset-category** that accepts either **INTEGRATOR** or **TOOLS**
- Default asset category is **INTEGRATOR**
- **INTEGRATOR** asset category represents a family of data connectors in Purple Fabric, while **TOOLS** asset category represents a family of data transformers in Purple Fabric
- Please choose the asset category that best represents the nature of the project
- Agent category can be updated in **pyproject.toml** before the first **build** operation is executed

Details of the files and folders

Name	**File/Directory**	**Description**
.config	Directory	profiles and credentials created locally (project level). It will be empty by default.
generated	Directory	Contains files that are generated by the SDK using commands, like models.py Note: Do not edit any files in this directory
app.py	File	The main file that contains the implementations of the features mentioned in **schema.json**
__init__.py	File	Standard Python file that

Name	File/Directory	Description
		denotes the current directory is a Python module.
pyproject.toml	File	Standard Python file that defines the project's metadata and dependencies.
schema.json	File	A JSON file that contains the specifications of the project's features and schemas.
Dockerfile	File	Specifies the project's operating system level dependencies.
.gitignore, .dockerignore	File	Standard .ignore files used in respective contexts.
README.md	File	Contains the project's documentation.

Develop the Connector

Note:

- To proceed with the below steps, a profile is required
- Profiles can be created using the pf-cli

Specification → Schema

- Update **schema.json** to define the interface for your connector. Specify the input schema and output schema for your connector inside it
- Generate Pydantic models for your **schema.json** by executing the **pf-cli generate models --profile <profile_name>** command. Models will be generated inside the **generated** directory
- Use your models inside **app.py** and build your business logic inside the **run_<feature_name>() function**

Custom Dependencies

- To install custom dependencies:
  - Update the **dependencies** section inside **pyproject.toml**
  - Ensure that you are using the project's virtual environment and the virtual environment is activated
  - Run the below command to install the custom dependencies
    pip install -e . --extra-index-url=https://${CONNECTOR_REPO_USERNAME}:${CONNECTOR_REPO_PASSWORD}@iainexus.intellectseeccloud.com/repository/idxplatform-connectors-py-hosted/simple --verbose

Implementation → Run function

Definition

- In the app.py file, you will find a run function with the following signature:
  def run_<FEATURE_NAME>(inputs: <FEATURE_NAME>Inputs, context: Context) -> <FEATURE_NAME>Outputs: …
- <FEATURE_NAME> should match at least one feature mentioned in **schema.json** file
- Add your business logic inside the run function
- You can use custom libraries inside the run function as long as you have mentioned them in **pyproject.toml** and installed them in your virtual environment
- At least one run function is required to execute the project
- In case there is a feature defined in schema.json, but not implemented in app.py, the SDK will raise an exception during build phase

Create a new feature

To generate a new feature and its implementation, execute the below command:

pf-cli generate feature --name <feature_name> --profile <profile_name>

This command does the following operations:

- Adds the boilerplate feature definition, input schema and output schema components in schema.json for the feature <feature_name>
- Generates models for the new input schema and output schema of the feature <feature_name>
- Imports these models in app.py
- Adds a new run function for the <feature_name> feature in app.py

Relationship of a feature between schema.json and app.py

- The **features** section in **schema.json** represents the specification of an operation and its input and output schemas
- The specified **features** are implemented as run functions inside **app.py**. The feature is mapped to the function name in the form **run_<FEATURE_NAME>**
  - Example, if the feature **download_files** is defined in **schema.json**, the corresponding implementing function's name should equal to **run_download_files**
- The input schema and output schema defined for the **feature** in the **schema.json** will be mapped to the respective **<FEATURE_NAME>Input** and **<FEATURE_NAME>Output** Pydantic models.
  - Example, if the feature **download_files** is defined in **schema.json**, the corresponding input model name should equal to **DownloadFilesInput** and the corresponding output model name should equal to **DownloadFilesOutput**

Multi-feature support

- A project can support multiple features and its respective implementations in the form of run functions
- This helps in reducing code duplication as well as improving code re-usability of shared logic

Logging

- You can add log statements to your application by using the context.logger() function

```
def run_<FEATURE_NAME>(inputs: <FEATURE_NAME>Inputs, context: Context) ->
<FEATURE_NAME>Outputs:
   context.logger.info('Hello, World!')
   ...
```

Outputs

- You must always return an object of type **<FEATURE_NAME>Outputs**
- You must use the **<FEATURE_NAME>Outputs** Pydantic model available in **generated/models.py** module

```
def run_<FEATURE_NAME>(inputs: <FEATURE_NAME>Inputs, context: Context) ->
<FEATURE_NAME>Outputs:
   ...
   return <FEATURE_NAME>Outputs(name='Hello, World!')
```

Exceptions

- You must use **AssetException** to raise any exception from your connector
- When building custom exceptions, always ensure that **AssetException** is inherited

```
from pf_connectors_py_sdk.core.exceptions import AssetException
# other imports...
class InvalidNameException(AssetException):
   def __init__(self, message):
       super().__init__(message)
def run_<FEATURE_NAME>(inputs: <FEATURE_NAME>Inputs, context: Context) ->
<FEATURE_NAME>Outputs:
   if inputs.name is None or inputs.name.startswith('123'):
       raise InvalidNameException('Name is invalid')
   return <FEATURE_NAME>Outputs(name='Hello, World!')
```

Credentials

<u>Adding Credentials to the Connector</u>

- This step is only needed for the connectors that require access to credentials
- In the **schema.json** file, add the new input section for connection and update the data type as "connection"

```
{
  "datatype": "connection",
  "mandatory": true,
    "hidden": false,
    "displayname": "Connection",
    "properties": {
      "label": "Connection",
      "options": [],
      "placeholder_text": "Select Connection",
      "component": "connectionSelect",
      "help_text": "Choose a connection",
      "parameter": true,
      "value_key": "connection",
      "display_order": 1,
      "option_label_key": "name",
      "option_value_key": "connection_id",
      "provider_name": "Service Name"
    }
  }
```

- Generate the models:
  ```
  pf-cli generate models --profile <profile>
  ```

Credential Providers

- Purple Fabric Platform provides support to create credentials for specific service providers like AWS, Google Cloud, Atlassian etc
- In order to consume a specific credential provider in your connector, please provide the name of the credential provider in the **provider_name** field

```
  {
    "datatype": "connection",
    "mandatory": true,
    "hidden": false,
    "displayname": "Connection",
    "properties": {
      // ...other properties
      "provider_name": "Amazon Web Service"
    }
  }
```

- After this, please generate the models to make use of the provider specific Pydantic model
  ```
  pf-cli generate models --profile <profile>
  ```

Supported Credential Providers

The list of supported credential providers is available here - Flow Designer: Asset Schemas

Note: You can use Generic Service provider in case no other credential provider suits your requirements.

Manage Profiles for the Connector

- Profiles are used for communication with Purple Fabric Platform to execute various commands.
- Keep the below details available to create a profile:
  - Environment
  - Tenant ID
  - API Key
  - Username
  - Password
- Create a profile using pf-cli profiles create
- Profiles can be re-used across projects, if it is created in the global scope
  Note:
  - The profile details will be validated before its created to ensure the provided details are valid
  - Profile validation will be performed for all the commands that require a profile argument
  - If your profile details are valid, and yet face issues with the SDK commands, you might need to check the policies assigned to your profile by your administrator.

Manage Credentials for the Connector

- Credentials can be created for any credential provider supported by Purple Fabric Platform
- To create a credential, use the below command:
  pf-cli credentials create --profile <profile>
- You will be prompted to choose a credential provider. You can use a **Generic Service** provider in case no other credential provider suits your requirements
  Note:
  - Credentials will be created at workspace level
  - You will be prompted to select a workspace from the available workspaces in your account
  - When creating a credential via CLI, it will be created in your local system as well as in the Purple Fabric Platform
- Once a credential is created, you will receive a **connection_id**
- Use this **connection_id** when executing the connector that uses a connection field
  pf-cli run --feature <feature> --inputs connection=<connection_id>

Running the Connector

Run the connector locally using the following commands:

- Use **run** to verify **app.py** is running as expected
- You can only run a single feature at a time

Build and Publish the Connector

To build the connector, run the below command:

pf-cli build --profile <profile>

Note:

- When building the connector, it is bound to a specific workspace
- You will be prompted to select a workspace from the available workspaces in your account
- Once a workspace is selected, any future commands executed will be done on the chosen workspace

During build, the following validations will be performed:

- Validation of **schema.json**
- Validation of the asset details provided in **pyproject.toml**
- Validate whether **run** function with the proper signature is present in **app.py** for all the features specified in **schema.json**

To publish the connector, run the below command:

pf-cli deploy --profile <profile>

- During publish, your code will be converted into a deployment artifact and sent to the deployment pipeline
- To check the status of the deployment, run the below command:

  pf-cli status --profile <profile>

schema.json file

schema.json file

Refer to the following steps:

1. Anatomy of schema.json file
2. Building features and schemas
3. Supported Data Types

Anatomy of schema.json file

Top-level Structure

The schema.json file is a structured configuration file that defines the behavior and interface of a connector or tool. It consists of the following main sections:

```
{
  "id": "unique.identifier",
  "name": "Display Name",
  "version": 1,
  "tags": ["category", "subcategory"],
  "features": [...],
  "configuration": {},
  "exceptions": [...],
  "schema": [...],
  "deployment": {...}
}
```

Key Components

- **Id:** Unique identifier for the connector/tool (e.g., "connectors.send-email-connector")
- **Name:** Display name of the connector/tool
- **Version:** Version number of the schema
- **Tags:** Categorization labels for the connector/tool
- **Features:** Defines available operations and their schemas
- **Configuration:** Global configuration settings
- **Exceptions:** List of possible exceptions
- **Schema:** Detailed schema definitions for inputs and outputs
- **Deployment:** Deployment configuration for the container

Building features and schemas

Feature Definition

Features are defined in the features array of the schema.json file. Each feature represents a specific operation or functionality of the connector/tool.

```
"features": [
  {
    "feature_name": {
      "input_schema": "request_schema_name",
      "output_schema": "response_schema_name",
      "configuration": null,
      "exceptions": ["namespace.error_type"],
      "displayname": "User-Friendly Name",
      "hidden": false
    }
  }
]
```

Feature Components

- **Feature Name:** Unique identifier for the feature
- **Input Schema:** Reference to the request schema definition
- **Output Schema:** Reference to the response schema definition
- **Configuration:** Feature-specific configuration (if any)
- **Exceptions:** List of possible exceptions
- **Display Name:** User-friendly name for the feature
- **Hidden:** Visibility flag for the feature

Building request schemas

Request schemas define the input parameters for a feature:

```
{
  "request_schema_name": {
    "entities": [
      {
        "field_name": {
          "displayname": "Field Display Name",
          "datatype": "string|object|file|connection",
          "is_array": false,
          "mandatory": true,
          "parameters": false,
          "properties": {
            "value_key": "field_name",
            "component": "input|connectionSelect",
            "display_order": 1,
            "label": "Field Label",
            "placeholder": "Placeholder Text",
            "parameter": true,
            "help_text": "Help text for the field"
          }
        }
      }
    ]
```

```
 }
}
```

Building response schemas

Response schemas define the output structure of a feature:

```
{
  "response_schema_name": {
    "entities": [
      {
        "result_field": {
          "displayname": "Result Field Name",
          "datatype": "string|object|file",
          "is_array": false,
          "mandatory": true,
          "properties": {}
        }
      }
    ]
  }
}
```

Supported Data Types

Below are the supported data types in **schema.json**

- Data Type - **String**
  Description - Represents textual information
  Usage:

```
{
  "datatype": "string",
  "is_array": false,
  "mandatory": true,
  "properties": {
    "value_key": "field_name",
    "component": "input",
    "display_order": 1,
    "label": "Field Label",
    "parameter": true,
    "placeholder": "Enter value",
    "help_text": "Help text for the field"
  }
}
```

- <u>Data Type</u> - **Objects**
  <u>Description</u> - Represents key-value pairs
  <u>Usage:</u>
  ```
  {
    "datatype": "object",
    "is_array": false,
    "mandatory": true,
    "properties": {
      "value_key": "field_name",
      "display_order": 1,
      "label": "Field Label",
      "parameter": true,
      "placeholder": "Select JSON",
      "help_text": "Help text for the field"
    }
  }
  ```

- <u>Data Type</u> - **File**
  <u>Description</u> - Represents documents in Purple Fabric
  <u>Usage:</u>
  ```
  {
    "datatype": "file",
    "is_array": true,
    "min_count": 1,
    "max_count": -1,
    "mandatory": true,
    "values": ["png", "jpg", "jpeg", "pdf", "tif", "tiff"],
    "properties": {
      "value_key": "files",
      "parameter": true,
      "display_order": 4,
      "label": "Input",
      "help_text": "Select the files to be uploaded to the specified S3 folder."
    },
    "entities": {
      "file_ref_id": {
        "datatype": "string",
        "mandatory": true,
        "hidden": false
      },
      "mime_type": {
        "datatype": "string",
        "mandatory": false,
        "hidden": false
  ```

```
      },
      "file_name": {
        "datatype": "string",
        "mandatory": true,
        "hidden": false
      },
      "file_id": {
        "datatype": "string",
        "mandatory": false,
        "hidden": false
      }
    }
  }
```

- Data Type - **Connection**
  Description - Represents a credential in Purple Fabric
  Usage:
  ```
   {
     "datatype": "connection",
     "mandatory": true,
     "hidden": false,
     "displayname": "Connection",
     "properties": {
       "label": "Connection",
       "options": [],
       "placeholder_text": "Select Connection",
       "component": "connectionSelect",
       "help_text": "Choose a connection",
       "parameter": true,
       "value_key": "connection",
       "display_order": 1,
       "option_label_key": "name",
       "option_value_key": "connection_id",
       "provider_name": "Service Name"
     }
   }
  ```

- Data Type - **Number**
  Description - Represents numerical data
  Usage:
  ```
            {
              "datatype": "number",
              "mandatory": true,
              "hidden": false,
  ```

```
  "properties": {
    "value_key": "field_name",
    "component": "input",
    "parameter": true,
    "display_order": 1,
    "label": "Field Label",
    "help_text": "Help text for the field"
  }
}
```

Examples of Production-ready Connectors

Examples of Production-ready Connectors

S3 Download Connector

1. File - schema.json
2. File - Pyproject.toml
3. File - app.py
4. File - models.py

1. File - schema.json

```
{
  "id": "connectors.s3_download",
  "name": "S3 Download",
  "version": 1,
  "tags": [
    "connectors",
    "storage",
    "download_files"
  ],
  "features": [
    {
      "download_files": {
        "input_schema": "download_files_request_schema",
        "output_schema": "download_files_response_schema",
        "configuration": null,
        "exceptions": [
          "connectors.s3_connector.S3FileDownloadError"
        ],
        "displayname": "Read from S3",
        "hidden": false
      }
    }
  ],
  "configuration": {},
  "exceptions": [
    "connectors.s3_connector.InvalidConfigurationError",
    "connectors.s3_connector.ConfigurationLoadError"
  ],
  "schema": [
    {
      "download_files_request_schema": {
        "entities": [
          {
```

```
  "connection": {
    "datatype": "connection",
    "mandatory": true,
    "hidden": false,
    "displayname": "Connection",
    "properties": {
      "label": "Connection",
      "options": [],
      "placeholder_text": "Select Connection",
      "component": "connectionSelect",
      "help_text": "Choose an S3 connection from the available list.",
      "parameter": true,
      "value_key": "connection",
      "display_order": 1,
      "option_label_key": "name",
      "option_value_key": "connection_id",
      "provider_name": "Amazon Web Service"
    }
  }
},
{
  "bucket_name": {
    "datatype": "string",
    "mandatory": true,
    "hidden": false,
    "displayname": "Bucket Name",
    "properties": {
      "label": "Bucket Name",
      "component": "input",
      "help_text": "Enter the name of the S3 bucket to download files from.",
      "parameter": true,
      "value_key": "bucket_name",
      "placeholder": "Provide Bucket Name",
      "display_order": 2
    }
  }
},
{
  "path": {
    "datatype": "string",
    "mandatory": true,
    "hidden": false,
    "displayname": "Path",
    "properties": {
            "label": "Path",
            "component": "input",
            "help_text": "Specify the path within the bucket to locate the files.",
            "parameter": true,
            "value_key": "path",
            "placeholder": "Provide Path",
            "display_order": 3
          }
        }
      }
    ]
  }
},
{
  "download_files_response_schema": {
    "entities": [
      {
        "files": {
          "datatype": "file",
          "displayname": "Files",
          "mandatory": true,
          "hidden": false,
          "values": [
            "png",
            "jpg",
            "jpeg",
            "pdf",
            "tif",
            "tiff"
          ],
          "is_array": true,
          "min_count": 1,
          "max_count": -1,
          "entities": {
            "file_name": {
              "datatype": "string",
              "mandatory": true,
              "hidden": false
            },
            "file_ref_id": {
              "datatype": "string",
              "mandatory": true,
              "hidden": false
            },
              "file_id": {
                "datatype": "string",
                "mandatory": false,
                "hidden": false
              },
              "mime_type": {
                "datatype": "string",
                "mandatory": false,
                "hidden": false
              },
              "total_pages": {
                "datatype": "number",
                "mandatory": true,
                "hidden": false
              }
            }
          }
        }
      ]
    }
  }
],
"deployment": {
  "type": "container",
  "shared": true,
  "properties": {
    "nodeAffinity": "",
    "autoScale": true
  },
  "workers": 2,
  "resources": {
    "cpu": 2,
    "mem": 2
  }
}
}
```

2. File - pyproject.toml

```
[project]
name = "read-from-s3-connector"
version = "0.0.1"
description = "Project built using the Purple Fabric Connectors Python SDK for
integration with the Purple Fabric Platform."
```

```
readme = "README.md"
requires-python = ">=3.12"
dependencies = [ "pf-connectors-py-sdk==0.1.111", "boto3",]

[tool.pf-connectors-py-sdk]
asset_name = "S3 Download"
asset_category = "INTEGRATOR"
asset_description = "Retrieve files from an Amazon S3 bucket."
```

3. File - app.py

```
from typing import List
import os
import boto3
from botocore.exceptions import ClientError
from generated.models import DownloadFilesInputs, DownloadFilesOutputs
from pf_connectors_py_sdk import Context
from pf_connectors_py_sdk.core.models import FileReference

from exceptions import (
    AWSAccessException,
    InvalidBucketNameException,
    InvalidPathException,
    FileDownloadException,
    FileProcessingException,
    InvalidAuthenticationException
)


def list_s3_objects(s3_client, bucket: str, prefix: str, recursive: bool = False) -> List[str]:
    """List objects in S3 bucket with given prefix. Can handle both file and directory
paths."""
    try:
        # Make a single list_objects_v2 call with appropriate parameters
        params = {
            'Bucket': bucket,
            # Remove trailing slash to handle both file and directory cases
            'Prefix': prefix.rstrip('/'),
        }

        # First try without delimiter to check if there are any files
        response = s3_client.list_objects_v2(**params)

        # If no files found with direct prefix, try with trailing slash
        if 'Contents' not in response:
            params['Prefix'] = f"{prefix.rstrip('/')}/"
            response = s3_client.list_objects_v2(**params)

        # If still no files and not recursive, try with delimiter
        if not recursive and 'Contents' not in response:
            params['Delimiter'] = '/'
            response = s3_client.list_objects_v2(**params)

        files = []

        # Process all contents (files) in the response
        for obj in response.get('Contents', []):
            key = obj['Key']
            if not key.endswith('/'):  # Skip directory markers
                files.append(key)

        # If we're not in recursive mode and found no direct files,
        # but have CommonPrefixes, switch to recursive mode for this case
        if not recursive and not files and 'CommonPrefixes' in response:
            # Try one more time with recursive listing
            return list_s3_objects(s3_client, bucket, prefix, recursive=True)

        return sorted(files)  # Return sorted list for consistent results

    except ClientError as e:
        if e.response['Error']['Code'] == 'NoSuchBucket':
            raise InvalidBucketNameException(
                f"Bucket '{bucket}' does not exist")
        raise AWSAccessException(f"Failed to list objects: {str(e)}")


def run_download_files(inputs: DownloadFilesInputs, context: Context) ->
DownloadFilesOutputs:
    # Validate authentication configuration
    if not inputs.connection.authentication:
        raise InvalidAuthenticationException(
            "Authentication type must be specified in the connection configuration")

    # Initialize S3 client with appropriate authentication
    try:
        if inputs.connection.authentication == "IAM ROLE":
            # Use IAM role-based authentication
            if not inputs.connection.IAM_role_name:
                raise InvalidAuthenticationException(
                "IAM role name must be provided for IAM Role authentication")

        # First, assume the outbound role in the current account
        outbound_role_arn = os.environ.get('AWS_OUTBOUND_IAM_ROLE')
        if not outbound_role_arn:
            raise InvalidAuthenticationException(
                "AWS_OUTBOUND_IAM_ROLE environment variable must be set for
cross-account access")

        sts_client = boto3.client('sts')

        # Step 1: Assume the outbound role
        outbound_role = sts_client.assume_role(
            RoleArn=outbound_role_arn,
            RoleSessionName="S3ConnectorOutboundSession"
        )

        # Create STS client with outbound role credentials
        sts_client_outbound = boto3.client(
            'sts',
            aws_access_key_id=outbound_role['Credentials']['AccessKeyId'],
            aws_secret_access_key=outbound_role['Credentials']['SecretAccessKey'],
            aws_session_token=outbound_role['Credentials']['SessionToken'],
            region_name=inputs.connection.region
        )

        # Step 2: Use the outbound role to assume the destination role
        destination_role = sts_client_outbound.assume_role(
            RoleArn=inputs.connection.IAM_role_name,
            RoleSessionName="S3ConnectorDestinationSession"
        )

        # Get final credentials from destination role
        credentials = destination_role['Credentials']
        s3_client = boto3.client(
            's3',
            aws_access_key_id=credentials['AccessKeyId'],
            aws_secret_access_key=credentials['SecretAccessKey'],
            aws_session_token=credentials['SessionToken'],
            region_name=inputs.connection.region
        )
    elif inputs.connection.authentication == "Secrets":
        # Use access key based authentication
```

```
            if not inputs.connection.access_key_id or not
inputs.connection.secret_access_key:
                raise InvalidAuthenticationException(
                    "Access key ID and secret access key must be provided for Secrets
authentication")

            s3_client = boto3.client(
                's3',
                aws_access_key_id=inputs.connection.access_key_id,
                aws_secret_access_key=inputs.connection.secret_access_key,
                region_name=inputs.connection.region
            )
        else:
            raise InvalidAuthenticationException(
                f"Invalid authentication type: {inputs.connection.authentication}. Must be either
'IAM Role' or 'Secrets'")

    except ClientError as e:
        raise AWSAccessException(f"Failed to initialize S3 client: {str(e)}")
    except Exception as e:
        raise AWSAccessException(f"Failed to initialize S3 client: {str(e)}")

    # Normalize path (remove leading/trailing slashes)
    path = inputs.path.strip('/')

    context.logger.info(
        f"Searching for files in bucket '{inputs.bucket_name}' with path '{path}'")

    # List files in S3
    try:
        s3_files = list_s3_objects(s3_client, inputs.bucket_name, path)
        if not s3_files:
            # Try to list the prefix to see if it exists
            try:
                s3_client.list_objects_v2(
                    Bucket=inputs.bucket_name,
                    Prefix=path,
                    MaxKeys=1
                )
                context.logger.error(
                    f"Path '{path}' exists but no files were found")
            except ClientError:
                context.logger.error(f"Path '{path}' does not exist in bucket")
            raise FileDownloadException(
            f"No files found at path '{path}' in bucket '{inputs.bucket_name}'")

    context.logger.info(f"Found {len(s3_files)} files to process")

    # Download files from S3 and upload to storage
    uploaded_files: List[FileReference] = []

    for s3_key in s3_files:
        try:
            # Download file from S3
            context.logger.info(f"Downloading file: {s3_key} from S3")
            response = s3_client.get_object(
                Bucket=inputs.bucket_name, Key=s3_key)
            file_content = response['Body'].read()

            # Upload to storage
            file_name = s3_key.split('/')[-1]
            context.logger.info(f"Uploading file: {file_name} to storage")
            file = context.storage.create_file(file_name, file_content)
            file_ref = context.storage.upload(
                file, f's3-upload-connector/{context.trace_id}/{file_name}')

            if file_ref:
                uploaded_files.append(file_ref)
                context.logger.info(
                    f"Successfully processed file: {file_name}")
            else:
                raise FileProcessingException(
                    f"Failed to upload file {file_name} to storage")

        except ClientError as e:
            raise FileDownloadException(
                f"Failed to download file {s3_key}: {str(e)}")
        except Exception as e:
            raise FileProcessingException(
                f"Failed to process file {s3_key}: {str(e)}")

    if not uploaded_files:
        raise FileProcessingException(
            "No files were successfully processed")

    return DownloadFilesOutputs(files=uploaded_files)

except Exception as e:
```

```
            raise InvalidPathException(f"Failed to access path '{path}': {str(e)}")
```

4. File - Models.py

```
# Generated by pf-connectors-py-sdk
# Date: 2025-02-17 06:55:27 UTC
# Do not edit this file

from pf_connectors_py_sdk.core.models import FileReference
from pydantic import BaseModel, Field
from typing import List

class AmazonwebserviceCredentials(BaseModel):
    """Pydantic model for AmazonwebserviceCredentials.

    Generated automatically from schema definition.
    """
    model_config = {
        "frozen": True,
        "validate_assignment": True,
        "populate_by_name": True,
        "extra": "forbid"
    }

    authentication: str
    IAM_role_name: str
    access_key_id: str
    secret_access_key: str
    region: str

class DownloadFilesInputs(BaseModel):
    """Pydantic model for DownloadFilesInputs.

    Generated automatically from schema definition.
    """
    model_config = {
        "frozen": True,
        "validate_assignment": True,
        "populate_by_name": True,
        "extra": "forbid"
    }

    connection: AmazonwebserviceCredentials
    bucket_name: str
    path: str
```

```
class DownloadFilesOutputs(BaseModel):
    """Pydantic model for DownloadFilesOutputs.

    Generated automatically from schema definition.
    """
    model_config = {
        "frozen": True,
        "validate_assignment": True,
        "populate_by_name": True,
        "extra": "forbid"
    }

    files: List[FileReference]
```

Send Email Connector

1. schema.json
2. pyproject.toml
3. app.py
4. models.py

1. schema.json

```
{
  "id": "connectors.send-email-connector",
  "name": "send-email-connector",
  "version": 1,
  "tags": [
    "connectors",
    "send_gmail"
  ],
  "features": [
    {
      "send_email": {
        "input_schema": "send_email_request_schema",
        "output_schema": "send_email_response_schema",
        "configuration": null,
        "exceptions": [],
        "displayname": "Email Send Message",
        "hidden": false
      }
    }
  ],
  "configuration": {},
  "exceptions": [
    "connectors.send_email_connector.InvalidConfigurationError"
  ],
  "schema": [
    {
      "send_email_request_schema": {
        "entities": [
          {
            "connection": {
              "datatype": "connection",
              "mandatory": true,
              "hidden": false,
              "displayname": "Choose Connection",
              "properties": {
                "label": "Choose Connection",
                "options": [],
                "placeholder_text": "Select Connection",
                "component": "connectionSelect",
                "help_text": "Select a predefined Email connection.",
                "parameter": true,
                "value_key": "connection",
                "display_order": 1,
                "option_label_key": "name",
                "option_value_key": "connection_id",
                "provider_name": "SMTP Service"
              }
            }
          },
          {
            "to_addresses": {
              "datatype": "string",
              "mandatory": true,
              "hidden": false,
              "is_array": true,
              "min_count": 1,
              "max_count": -1,
              "displayname": "To (Recipient Email)",
              "properties": {
                "label": "To (Recipient Email)",
                "component": "input",
                "help_text": "Provide the primary recipient email addresses.",
                "parameter": true,
                "value_key": "to_addresses",
                "placeholder": "Provide Recipient Email addresses",
        "display_order": 2
      }
    }
  },
  {
    "subject": {
      "datatype": "string",
      "mandatory": true,
      "hidden": false,
      "displayname": "Subject",
      "properties": {
        "label": "Subject",
        "component": "input",
        "help_text": "Enter the email subject.",
        "parameter": true,
        "value_key": "subject",
        "placeholder": "Enter the email subject.",
        "display_order": 3
      }
    }
  },
  {
    "body": {
      "datatype": "string",
      "mandatory": true,
      "hidden": false,
      "displayname": "Email Body",
      "properties": {
        "label": "Email Body",
        "component": "input",
        "help_text": "Compose the email content (supports HTML and plain text).",
        "parameter": true,
        "value_key": "body",
        "placeholder": "Compose the email content (supports HTML and plain text).",
        "display_order": 4
      }
    }
  },
  {
    "attachments": {
      "displayname": "Attach files from previous workflow steps.",
      "datatype": "file",
      "is_array": true,
      "mandatory": false,
            "parameters": false,
            "properties": {
              "value_key": "attachments",
              "parameter": true,
              "display_order": 5,
              "label": "Attachments",
              "help_text": "Attach files from previous workflow steps."
            },
            "values": [
              "png",
              "jpg",
              "jpeg",
              "pdf",
              "tif",
              "tiff"
            ],
            "entities": {
              "file_ref_id": {
                "datatype": "string",
                "mandatory": true,
                "hidden": false
              },
              "mime_type": {
                "datatype": "string",
                "mandatory": false,
                "hidden": false
              },
              "file_name": {
                "datatype": "string",
                "mandatory": true,
                "hidden": false
              },
              "file_id": {
                "datatype": "string",
                "mandatory": false,
                "hidden": false
              }
            }
          }
        }
      ]
    }
  },
  {
        "send_email_response_schema": {
          "entities": [
            {
              "result": {
                "datatype": "string",
                "displayname": "Result",
                "mandatory": true,
                "hidden": false
              }
            }
          ]
        }
      }
    ],
    "deployment": {
      "type": "container",
      "shared": true,
      "properties": {
        "nodeAffinity": "",
        "autoScale": true
      },
      "workers": 2,
      "resources": {
        "cpu": 2,
        "mem": 2
      }
    }
  }
```

2. Pyproject.toml

```
[project]
name = "send-email-connector"
version = "0.0.1"
description = "Project built using the Purple Fabric Connectors Python SDK for
integration with the Purple Fabric Platform."
readme = "README.md"
requires-python = ">=3.12"
dependencies = ["pf-connectors-py-sdk==0.1.111", ]

[tool.pf-connectors-py-sdk]
asset_name = "Send Email"
asset_description = "Send emails with support for attachments."
asset_category = "INTEGRATOR"
```

3. app.py

```
import smtplib
from email.mime.application import MIMEApplication
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

from generated.models import SmtpserviceCredentials, SendEmailInputs,
SendEmailOutputs
from pf_connectors_py_sdk import Context
from pf_connectors_py_sdk.core.exceptions import AssetException


class SendEmailException(AssetException):
    def __init__(self, message):
        super().__init__(message)


def run_send_email(inputs: SendEmailInputs, context: Context) -> SendEmailOutputs:
    # Create message container
    msg = MIMEMultipart()
    msg['Subject'] = inputs.subject
    msg['From'] = inputs.connection.email_id
    msg['To'] = ', '.join(inputs.to_addresses)

    # Add body
    msg.attach(MIMEText(inputs.body, 'html'))

    # Add attachments if any
    for attachment in inputs.attachments:
        if attachment.file_ref_id and attachment.file_name:
            # Download file from storage using file_ref_id
            file_content = context.storage.download(file_ref_id=attachment.file_ref_id)

            # Create MIME attachment part directly from the downloaded content
            part = MIMEApplication(file_content.content, _subtype=attachment.mime_type)
            part.add_header('Content-Disposition', 'attachment',
filename=attachment.file_name)
            msg.attach(part)

    try:
        # Connect to SMTP server
        server = smtplib.SMTP(inputs.connection.server_url, inputs.connection.server_port)
        server.starttls()
        # Login
        server.login(inputs.connection.email_id, inputs.connection.password)

        # Send email
        server.send_message(msg)
        server.quit()

        return SendEmailOutputs(result="Email sent successfully")
    except Exception as e:
        raise SendEmailException(f"Failed to send email: {str(e)}")
```

4. models.py

```
# Generated by pf-connectors-py-sdk
# Date: 2025-03-20 09:56:25 UTC
# Do not edit this file

from typing import List

from pf_connectors_py_sdk.core.models import FileReference
from pydantic import BaseModel


class SmtpserviceCredentials(BaseModel):
    """Pydantic model for SmtpserviceCredentials.

    Generated automatically from schema definition.
    """

    model_config = {
        "frozen": True,
        "validate_assignment": True,
        "populate_by_name": True,
        "extra": "forbid",
    }

    password: str
    server_url: str
    server_port: int
    email_provider: str
    email_id: str


class SendEmailInputs(BaseModel):
    """Pydantic model for SendEmailInputs.

    Generated automatically from schema definition.
    """

    model_config = {
        "frozen": True,
        "validate_assignment": True,
        "populate_by_name": True,
        "extra": "forbid",
    }

    connection: SmtpserviceCredentials
    to_addresses: List[str]
    subject: str
    body: str
    attachments: List[FileReference]


class SendEmailOutputs(BaseModel):
    """Pydantic model for SendEmailOutputs.

    Generated automatically from schema definition.
    """

    model_config = {
        "frozen": True,
        "validate_assignment": True,
        "populate_by_name": True,
        "extra": "forbid",
    }

    result: str
```

SDK - Frequently Asked Questions (FAQs)

SDK - Frequently Asked Questions (FAQs)

User Accounts

No.	Question	Answer
1.	What are the policies needed by a user in Purple Fabric to work with the Connectors SDK?	At-least GenAI_User policy must be assigned to the user

Connector Development

No.	Question	Answer
2.	How to add dependencies that must be present at the Operating System during the runtime of the project?	Specify the operating system level dependencies in the **Dockerfile** of your project.
3.	Can I re-use virtual environments across multiple projects created by the SDK?	We strongly advise not to re-use virtual environments. We recommend one virtual environment per project to ensure isolation of dependencies and avoid version conflicts.