CloudBlue
https://cloudblue.com

Extension Project

This article has been generated from the online version of the documentation and might be out of date. Please, make sure to always refer to the online version of the documentation for the up-to-date information.

Auto-generated at May 17, 2022

The following article provides instructions on how to bootstrap and deploy your DevOps extension project. This article showcases required CloudBlue Connect Command Line Interface (CLI) commands and demonstrates how to successfully configure your extension. Furthermore, the following outlines all tools and programs that are required to start working with your extension project.

Requirements

Before starting to configure your extension project, make sure to install the following tools and programs:

- **Docker**
  Install Docker to run Python (recommended version 3.8 or later) and your extension projects.
- **Poetry**
  Deploy the Poetry package manager and get familiar with its documentation.
- **Connect CLI**
  Connect CLI is an essential tool that allows you to bootstrap, test, and validate your extension project.
- **Your IDE app**
  Make sure to get your favorite Integrated Development Environment software application up and running for the following operations.

In addition, it is strongly recommended to familiarize yourself with general operations on the CloudBlue Connect platform and Connect Open API client documentation.

It is also recommended to get familiar with GitHub actions and streamline your test operations with PyTest.

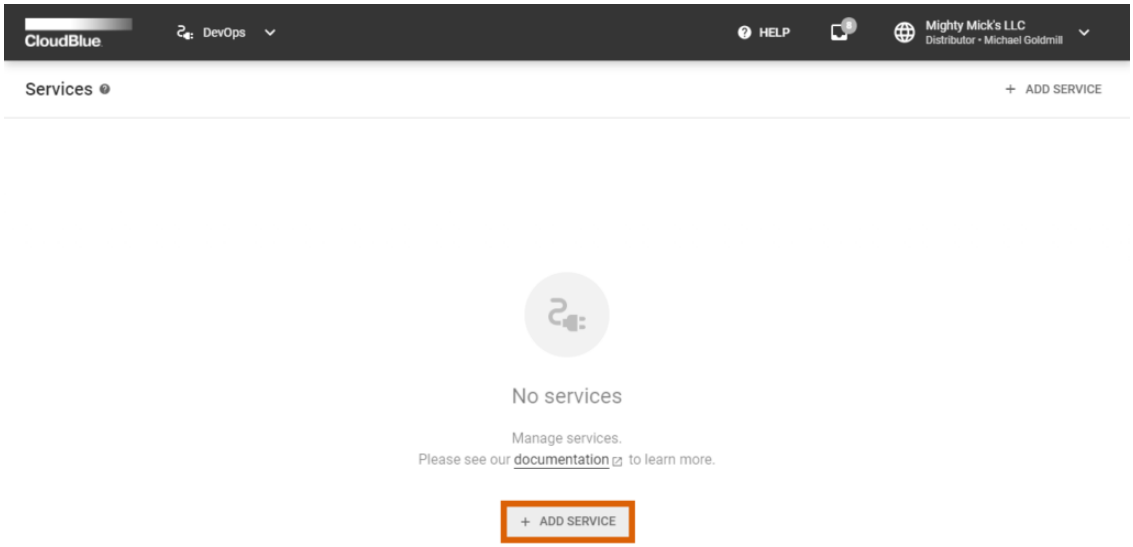
Getting Started

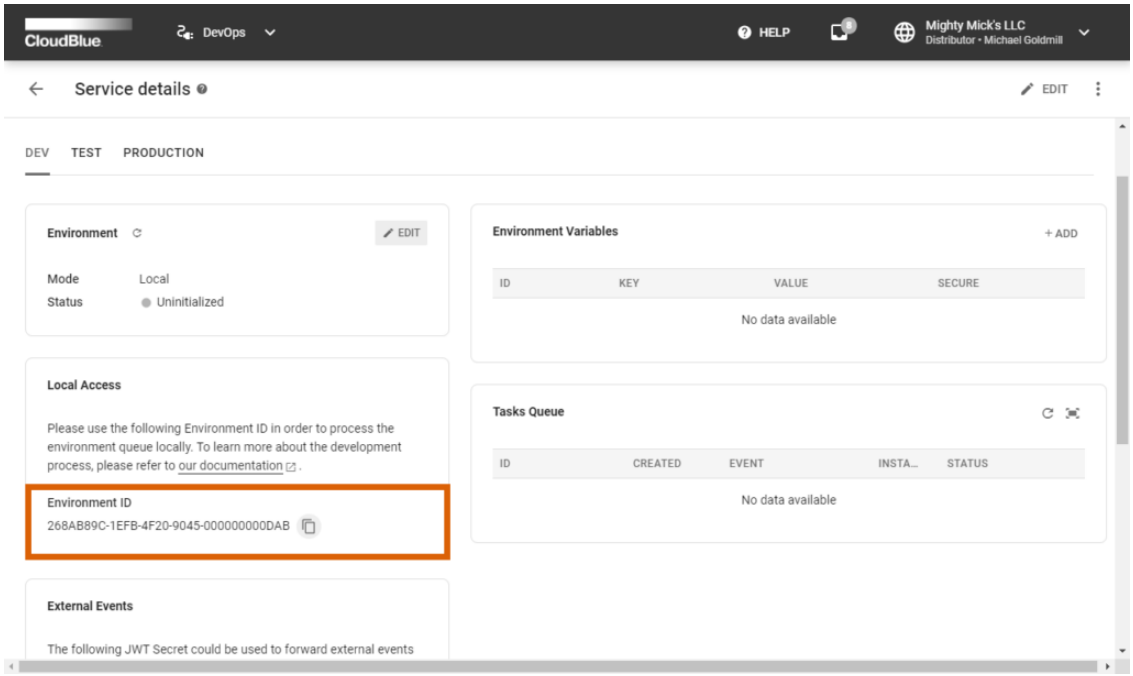
The following guidelines showcase how to make necessary preparations before the extension project configuration. Namely, the guidelines below showcase how to define a service and access required environment identifier on the DevOps module. Furthermore, use the following to learn how to activate your account via the Connect CLI tool.

Add Your Service

Access the **DevOps** module to add a new service on the CloudBlue Connect platform.

Click the **Add Service** button to launch a wizard. Follow the wizard steps to successfully add your service.





Copy the *Environment ID* under **Local Access** for each of your local environments. In case all of your environments should be deployed in the Cloud, such identifiers are not necessary.

If more detailed instructions on service objects are required, navigate to the User Interface sub-article of the DevOps module documentation.

Activate Your Account

Next, make sure that your Connect account (that is required for the DevOps operations) should be added and activated via the **CloudBlue Connect CLI** tool. This streamlines and accelerates the following procedures, such as acquiring your token, copy-pasting your Environment ID, and so on.

Use the following command to open your account list via the CLI tool:

```
ccli account list
```

Activate your required account by using the corresponding command:

```
ccli account activate VA-123-123
```

Note that *VA-123-123* should be replaced with your actual account ID.

Bootstrap Your Extension

Once your required account is activated and your service object is created on the Connect platform, you can bootstrap your extension project. Use the following Connect CLI command to start your project configuration:

```
ccli project extension bootstrap
```

Therefore, you will be prompted to configure your extension project. Follow the guidelines below to successfully complete this configuration.

General Configuration

Specify your general extension project details as follows:

- **Project name**: Enter your project name in this field.
- **Description**: Provide your project description.
- **Package name**: Specify your project package name.
- **Author**: Enter the extension project author.
- **Version**: Specify your extension project version.
- **License**: Your extension should be compatible with one of the provided software licenses (*Apache Software License 2.0, MIT*, or *BSD*). Otherwise, the Connect platform will not work with your project.
- **GitHub actions**: Enable or disable GitHub actions for your extension project.
- **Asynchronous libraries**: Switch on or off asynchronous libraries for your project. Note that it is possible to use only either *asynchronous* library or *synchronous* library. Your project will not be deployed in case it uses both libraries.

Press **Next** to continue with your extension project configuration.

API, EnvID and Hostname

Provide the following values to the appeared form:

- **API Key**: If a local environment is required, enter your *EaaS API Key* that's generated via the Integrations module on the Connect platform. In case the Cloud environment is selected, the system automatically generates and injects a token to your required environment; therefore, it is necessary to leave this field intact.
- **Environment ID**: Fill out this field in case the local environment is required. Otherwise, leave this form intact. Access your Environment ID via the *Local Access* section of the Service Details screen.
- **Connect API hostname**: Specify your API hostname in this field if a local environment is required. This hostname is specified within the General section of the Integrations module.

Press **Next** to continue with your extension project configuration.

Asset Processing Capabilities

Specify required asset (subscription) processing capabilities that should be supported by your extension project. Include or exclude the following fulfillment request types:

- **Purchase**: Once a new subscription is created, the system generates a new purchase request to process this subscription.
- **Change**: These request are used to change purchased items or their quantity. Note that change requests can also be used to update ordering parameter data.
- **Suspend**: Such requests, as the name implies, are used to suspend a subscription.
- **Resume**: These requests are used to reactivate suspended subscriptions.
- **Cancel**: In case your customers terminate their subscription, the system generates a cancel request.
- **Adjustment**: Such requests are created by Vendors to adjust parameter information (*Fulfillment* and *Ordering* phases) and change *Activation Template*.

Note that this list is used to specify if your extension project supports such request, regardless of their statuses. In case more detailed information on fulfillment requests or their types is required, refer to the Fulfillment Requests sub-article of the Subscription module documentation.

Asset Validation Capabilities

Fill out the following form in case your extension should support real-time subscription validation:

- **Purchase Request**: Enable this capability if your project include the real-time validation of Purchase Fulfillment Requests.
- **Change Request**: In case your project features the real-time validation of Change Subscription Requests, turn this capability on.

Press **Next** to continue with your extension project configuration.

Tier Configuration Capabilities

Use the following form to select tier configuration capabilities that are supported by your extension:

- **Setup Request Processing**: Enable this capability if your extension can process Setup Tier Requests.
- **Change Request Processing**: In case your extension features the Change Tier Request processing, switch this capability on.
- **Adjustment Request Processing**: Turn on this capability if your extension supports Adjustment Tier Request processing.
- **Setup Request Validation**: Enable this capability in case your extension features the Setup Tier Request validation.
- **Change Request Validation**: If your extension project supports Change Request validation, turn this capability on.

Press **Next** to continue with your extension project configuration.

Product Capabilities

Specify the following product capabilities in case they are supported by your extension:

- **Product Action Execution**: If your extension project should include product actions, switch this capability on.
- **Product Custom (External) Event**: Enable this capability in case certain 3rd party systems should interact with your extension. For example, you can use an external system that is webhook-based and uses asynchronous libraries. Thus, by using this capability, you can wake an endpoint that should handle these webhooks.

Press **Create** to finalize your extension project configuration. The CLI tool informs you about the successful extension project creation and provides your project location.

Therefore, you can open your IDE software application and start working with your extension. Note that it is required to deploy Docker to run your created project.

Project File Structure

The following outlines the general project file structure and provides guidelines on how to work with your created extension project.

Environment Files

Your required local environments are specified within the *.ENV* files. The environment files incorporate the following structure:

```
# Required API Key is generated via the Integrations module of the Connect platform
export API_KEY="ApiKey XXXXXXX:xxxxxxxxxxxxxxxxxxxxxxxxxxxx"
```

```
# Environment ID is displayed under Local Access within the Service Details screen on Connect
export ENVIRONMENT_ID=xxxxxxx-xxxx-xxxx-xxxx


# Connect API hostname can be accessed via the Integrations module of the Connect platform
export SERVER_ADDRESS=api.connect.cloudblue.com
```

Once all of your required environment files are presented, customize the docker compose file and specify your environments within this file as described below.

Docker-Compose File

Access the **docker-compose.yml** file to define your docker containers. It is recommended to use individual containers for each of your deployed environments (i.e., a container for development environment, one for testing environment, and so on). The following showcases examples of defined docker containers:

```
version: '3'

services:
  project_dev:
  # Define your required environment
    container_name: project_dev
    image: cloudblueconnect/connect-extension-runner:01.1
    # Pushes images to their respective registry/repository
    command: cextrun -d # Use this command to select the debug mode
    volumes:
      # Declare your volume for persisting data generated by and used by your docker container
      - .:/extension
    env_file:
      # Your specified environment; it is used only for the local mode
      - project_dev.env

  # Bash containers are especially helpful for local environments, since it
  # allows using your commands and displays your command output
  project_bash:
    container_name: project_bash
    image: cloudblueconnect/connect-extension-runner:01.1
    command: /bin/bash
    volumes:
      - .:/extension
    env_file:
      - project_bash.env
```

```
  # This container is instantiated for your test operations
  # note that it is especially helpful in case you use GitHub actions
  project_test:
    container_name: project_test
    image: cloudblueconnect/connect-extension-runner:01.1
    command: /bin/bash -c "poetry install && pytest --verbose --cov=project --cov-report=html --cov-
report=term-missing:skip-covered"
    volumes:
      - .:/extension
    env_file:
      - project_test.env
```

PyProject File

Note that the Poetry package manager is used to install required dependencies. Therefore, access the **pyproject.toml** file to familiarize yourself with provided Poetry dependencies. This file displays versions of each provided dependency, incorporated required notations, and more.

```
[tool.poetry]
name = "eaas-extension-e2e"
version = "1.0.0"
description = "Extension capabilities description"
authors = ["Globex Corporation"]
license = "Apache Software License 2.0"
packages = [
    { include = "connectext" }
]
readme = "./README.md"

[tool.poetry.plugins."connect.eaas.ext"]
"extension" = "connectext.extension:E2EExtension"

[tool.poetry.dependencies]
python = "^3.8"


[tool.poetry.dev-dependencies]
pytest = "^6.1.2"
pytest-cov = "^2.10.1"
pytest-mock = "^3.3.1"
coverage = {extras = ["toml"], version = "^5.3"}
responses = "^0.12.0"
flake8 = "~3.8"
flake8-bugbear = "~20"
flake8-cognitive-complexity = "^0.1"
flake8-commas = "~2.0"
flake8-future-import = "~0.4"
flake8-import-order = "~0.18"
flake8-broken-line = "~0.3"
flake8-comprehensions = "^3.3.1"
flake8-debugger = "^4.0.0"
flake8-eradicate = "^1.0.0"
flake8-string-format = "^0.3.0"

[build-system]
requires = ["poetry-core>=1.0.0"]
```

The pyproject file is especially helpful to fix any appeared issues with your project. Furthermore, note that you can use this file to declare your own dependencies if necessary.

Once your work with all of the aforementioned files is complete, you can launch your containers and your required environments as described below.

Boot Up Your Containers

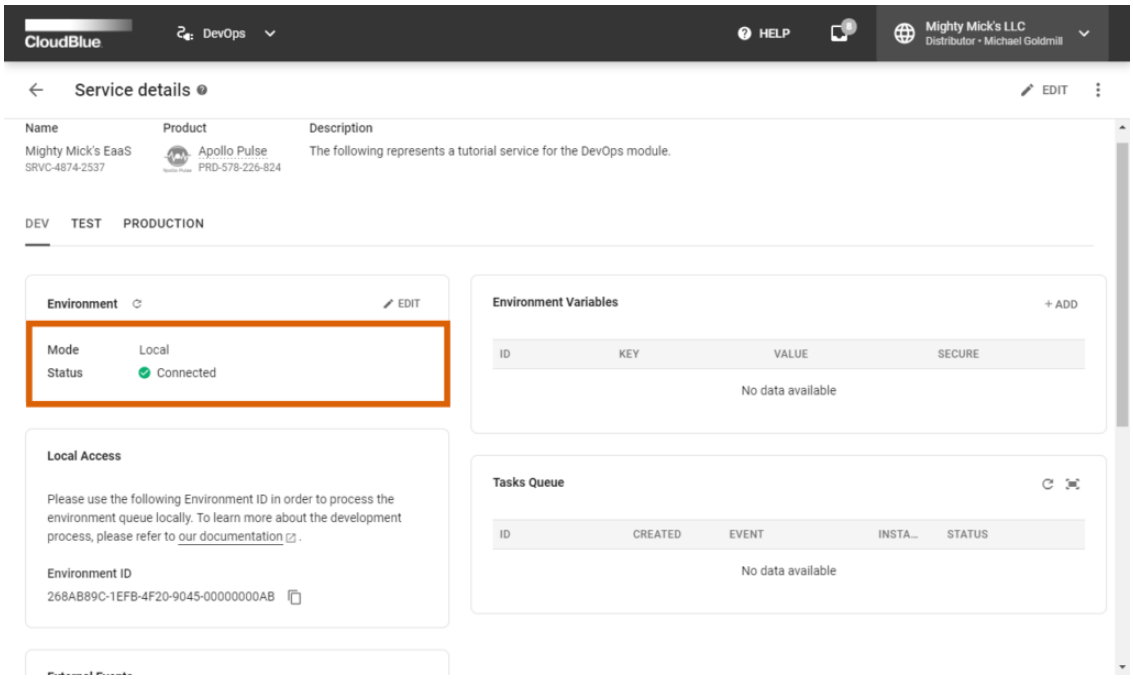
It is required to use Docker to build your extension project container and boot up required environments. Make sure to select your project via `cd {your_project_path}`. Thereafter, execute the following command:

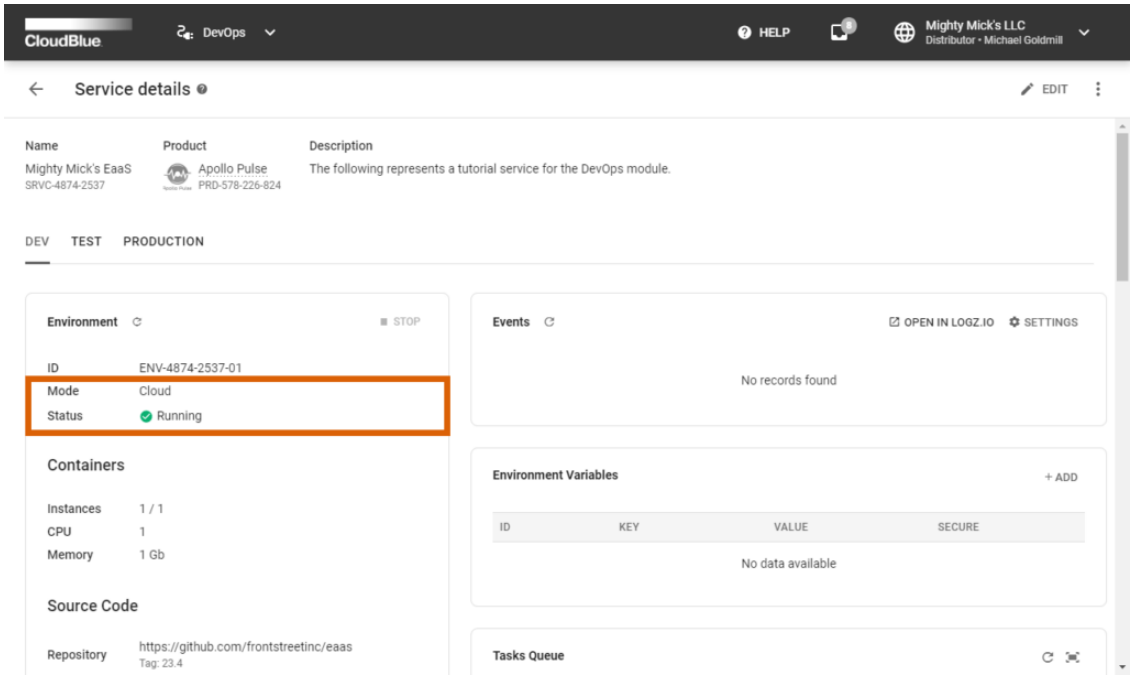
```
docker compose run project_bash
```

Note that *project_bash* should be replaced with your specified container name. In case a container with `bin/bash` command is deployed, you can also:

- Install required dependencies with the `poetry install` command.
- Start your testing operations by using the `pytest` command.
- Initialize selected environment in the debug mode with the `cextrun -d` command.

Therefore, your specified environment should be successfully deployed on the CloudBlue Connect platform. Make sure that the Connect interface displays the *Connected* status next to your selected environment. In case the Cloud mode is selected, provide a link to your Git repository with your extension project and ensure that the environment is switched to the *Running* state.





Once your environment is initialized successfully, the system starts the SSL connection. Note the Connect platform acts both as an endpoint and as a router for your custom events, product actions, configurations, etc. Any supported event or action within the initialized environment will be processed by your extension.

Project Customization

The following explains how to change the workflow and capabilities of your extension project. Note that you can also get your environment up and running for the following extension project customization. The Connect platform can apply your changes while your environments are in the *Connected* or *Running* states.

Extension.JSON

The **extension.json** file within your project folder includes specified capabilities of your created extension project. This file also incorporates additional properties that are described in the example below:

```
{
  "name": "Extension Project", // Your specified project name
  "description": "Mighty Mick's Extension Project.", // Your project description
  "version": "1.0.0", // Change your extension version if necessary
  "readme_url": "https://github.com/fstreeteaas/README.md", // Link to your readme file
  "changelog_url": "https://github.com/fstreeteaas/changelog.md", // Link to your changelog file
  "capabilities": { // This provides a list with your enabled capabilities
    "asset_purchase_request_processing": [
      "draft" // Request states that are supported by your project
```

```
      "tiers_setup"
      "pending"
      "inquiring"
      "approved"
      "failed"
  ]
}
```

Customize your extension project by excluding request states that should not be supported and by including statuses that your project should support. For example, you can exclude the *draft* statuses for your purchase fulfillment requests in case your project features real-time validation of draft purchase requests.

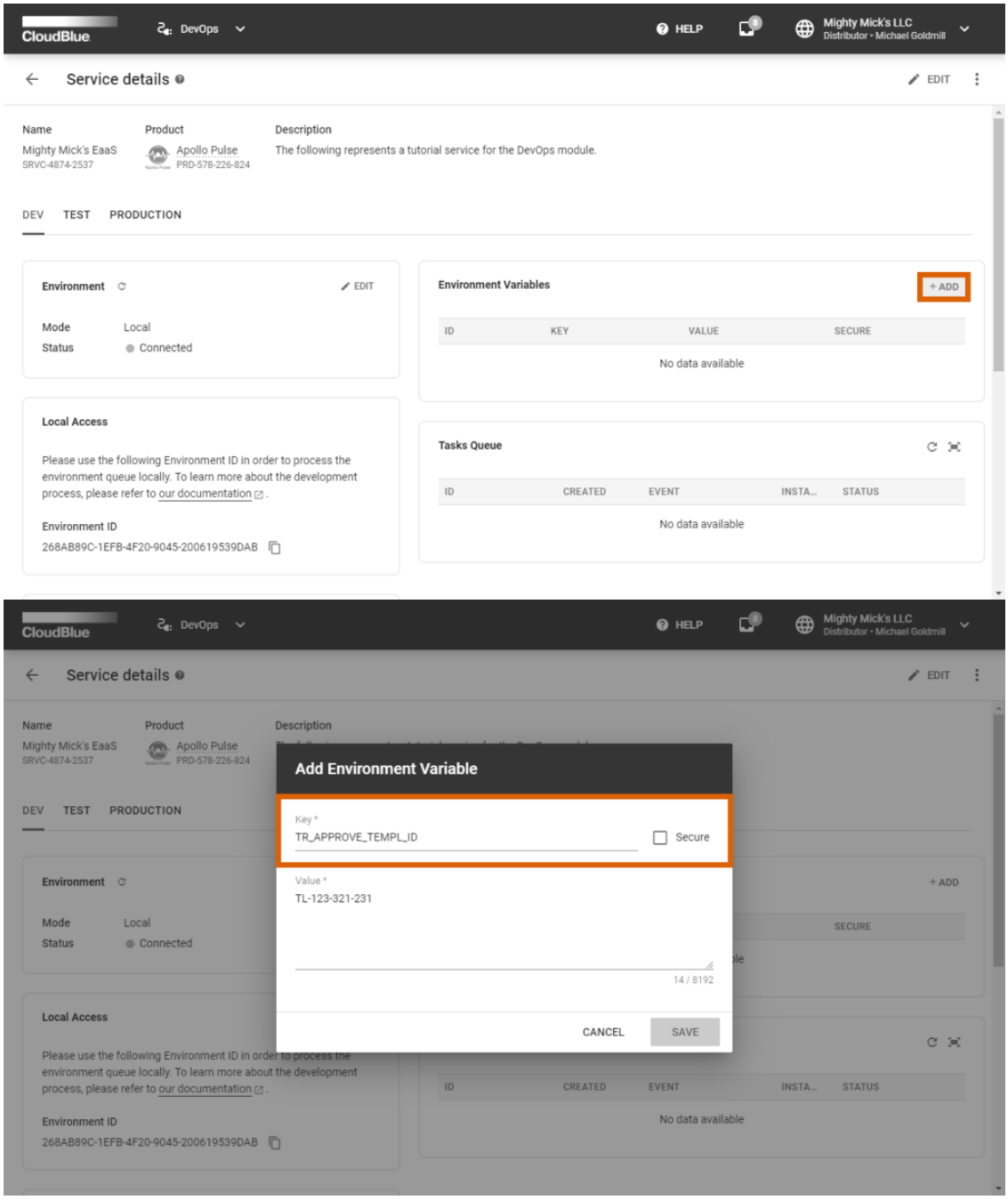
Extension.PY

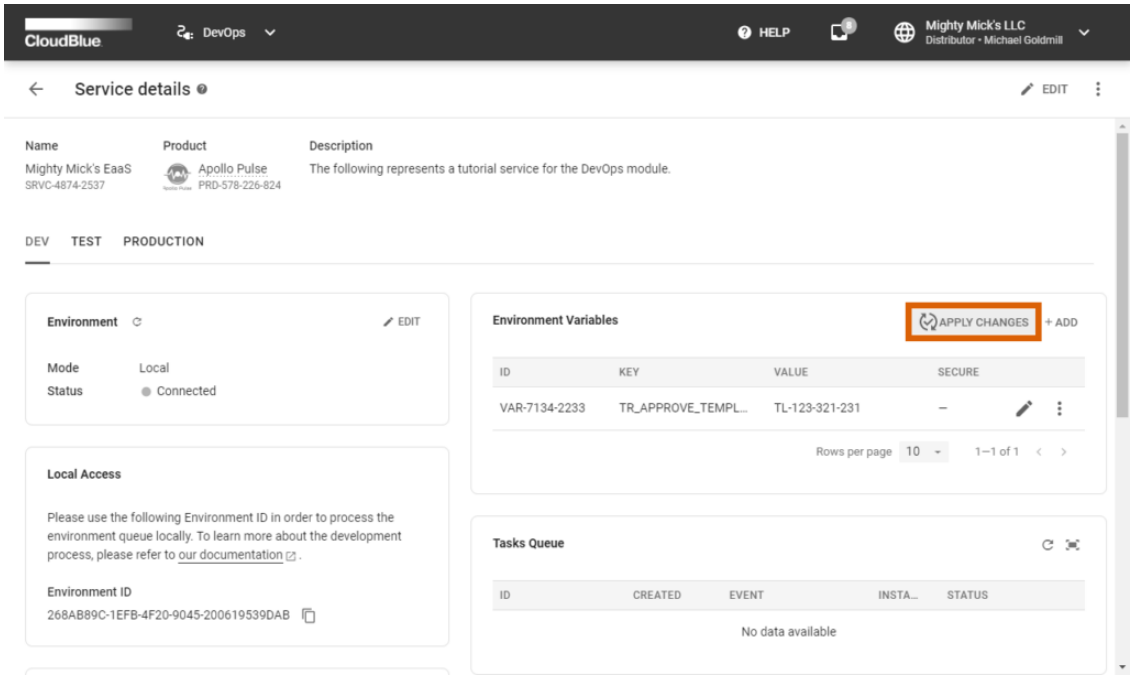
The **extension.py** file is used to define the workflow of your extension project. The structure of this file and the workflow logic are demonstrated in the following example:

```
# Imports Extension as a Service classes that are required for following operations

from connect.eaas.extension import (
    Extension,
    ProcessingResponse,
    ValidationResponse,
    ProductActionResponse,
    CustomEventResponse,
)

import random
import string


class MyProject(Extension):

    # Extension is a general EaaS class that provides an instantiated
    # Open API client, a logger, and configuration with your environment variables.

    def approve_asset_request(self, request, template_id):
        # This function is used to approve purchase request;
        # note that only the request parameter is necessary to obtain a required
        # request object

        self.logger.info(f'request_id={request["id"]} - template_id={template_id}')

        # This code instantiates the client,
```

```
        # specifies the collection operation, provides request IDs,
        # and defines the required action

        self.client.requests[request['id']]('approve').post(
            {
                'template_id': template_id,
            }
        )
        self.logger.warning(f"Approved request {request['id']}")

        # Note that your project can have different logging levels.

    def process_tier_config_change_request(self, request): # This function is used to process change tier
requests
        self.logger.info(
            f"Received event for request {request['id']}, type {request['type']} "
            f"in status {request['status']}"
        )
        if request['status'] == 'pending':
            template_id = self.config['TR_APPROVE_TEMPL_ID']

            # Define values for your environment variables (e.g.,TR_APPROVE_TEMPL_ID) via Service Details
on Connect

            self.approve_tier_request(request, template_id)
        return ProcessingResponse.done()
```

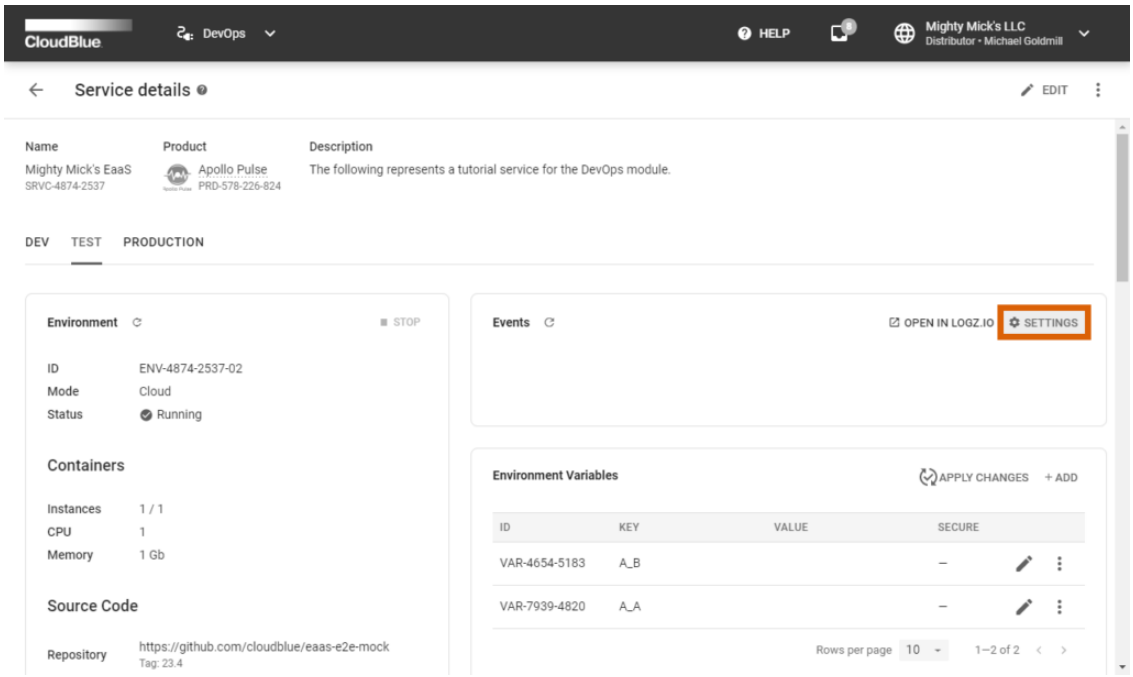
Note that environment variables that are created for the *Extension's config* parameter can include the **Secured** flag. In case the *Secured* checkbox is checked, the provided value will be encrypted. Therefore, this value can be used only by your extension project.

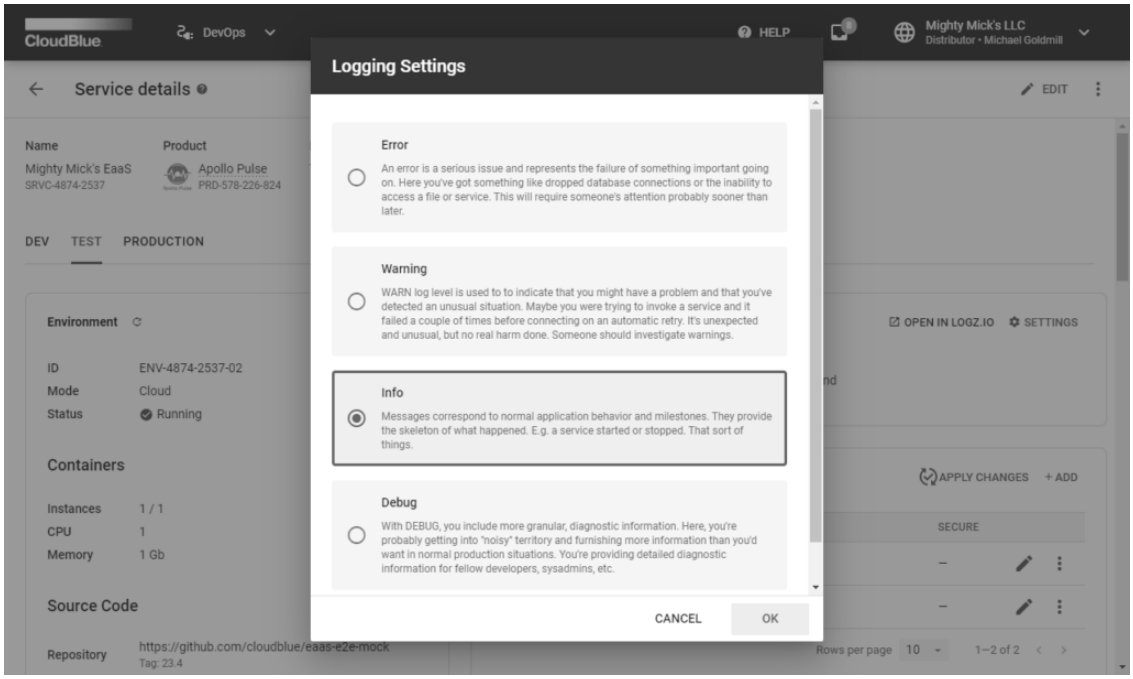




The system allows you to instantly implement your variables to your connected environment via the **Apply Changes** button.

Furthermore, note that *Extension's logger* generates required output locally if the local environment is initialized. In case the *Cloud* environment is selected, the logger uses the logz.io service with a Kibana UI to generate logs, create alerts, etc. Click the **Settings** button under your *Events* via the **Service Details** screen to access different logging levels.





Before changing the logic of the provided functions within the the **extension.py** file, it is increasingly important to outline the key difference between two categories of request tasks that your extension project can work with. In addition, it is also especially important to understand how your extension works with product actions and custom events.

Background Tasks

The first category is called *background tasks*, since such tasks can be processed in the background automatically. Such tasks are used to assign required statuses to certain objects. Note, however, that tasks and business objects on the Connect platform (e.g., requests) are different entities. Failing a background task does not mean that your request or any other object will be automatically assigned to the *Failed* state. Therefore, background tasks can be relaunched in case of an error. Each failed background task will be automatically restarted up to 10 times.

Access the **extension.py** file to customize the background task processing logic as described below:

```
def process_asset_resume_request(self, request): # This example function processes resume subscription
requests
        self.logger.info(
            f"Received event for request {request['id']}, type {request['type']} "
            f"in status {request['status']}"
        )
        if request['status'] == 'pending':
            template_id = self.config['ASSET_REQUEST_APPROVE_TEMPLATE_ID']
            self.approve_asset_request(request, template_id)
        return ProcessingResponse.done()
```

```
# You can add logic if the task is failed. For example, use the following code to provide a message for
failed tasks:
# return ProcessingResponse.fail(output= "Task is Failed!")
# You can skip a task by using this code: return ProcessingResponse.skip
# Note that you can also reschedule a task; there are two ways to perform this operation:
# 1) return ProcessingResponse.reschedule(countdown=30) – minimum value for countdown is 30 seconds
# 2) return ProcessingResponse.slow_process_reschedule(countdown=300) - minimum value for countdown is 300
seconds
```

Interactive Tasks

The second category of tasks can be referred to as *interactive tasks*, because such tasks require your attention to process business objects on the CloudBlue Connect platform. Namely, such tasks represent dynamic (real-time) validation of your specified request types. The following example demonstrates a dynamic validation function:

```
def validate_asset_purchase_request(self, request): # Real-time validation of purchase requests
    self.logger.info(f"Asset Validation with if {request['id']}")
    return ValidationResponse.done(request) # Returns request objects once the task is complete
```

Note that there is no need to provide codes for validation task failures. Failed validation tasks will not mark provided parameters as invalid. Furthermore, if the validation process fails (e.g., in case of the *500 error*), the system accepts your request either way.

Product Actions

Product actions, as the name implies, are used to interact with your product. Refer to the Connect Open API specification in case more information on product action is required. The example below showcases a product action function within the **extension.py** file:

```
def execute_product_action(self, request): # The function requires request objects; i.e., the pilot of JWT
tokens
   self.logger.info(f'Product action: {request}')
   return ProductActionResponse.done(
      http_status=302, # Response for product action can include an http status; e.g., to perform
redirections
# Note, however, that this shouldn't be a permanent redirect; otherwise you will lose control of your
product action
      headers={'Location': 'https://google.com'}, # In addition, response can get headers
# Response can also include body and return html code if necessary
      )
```

There is also no need to provide codes for product action failures, since failing product actions actually means to fail your business flow and not to fail your specified task.

Custom Events

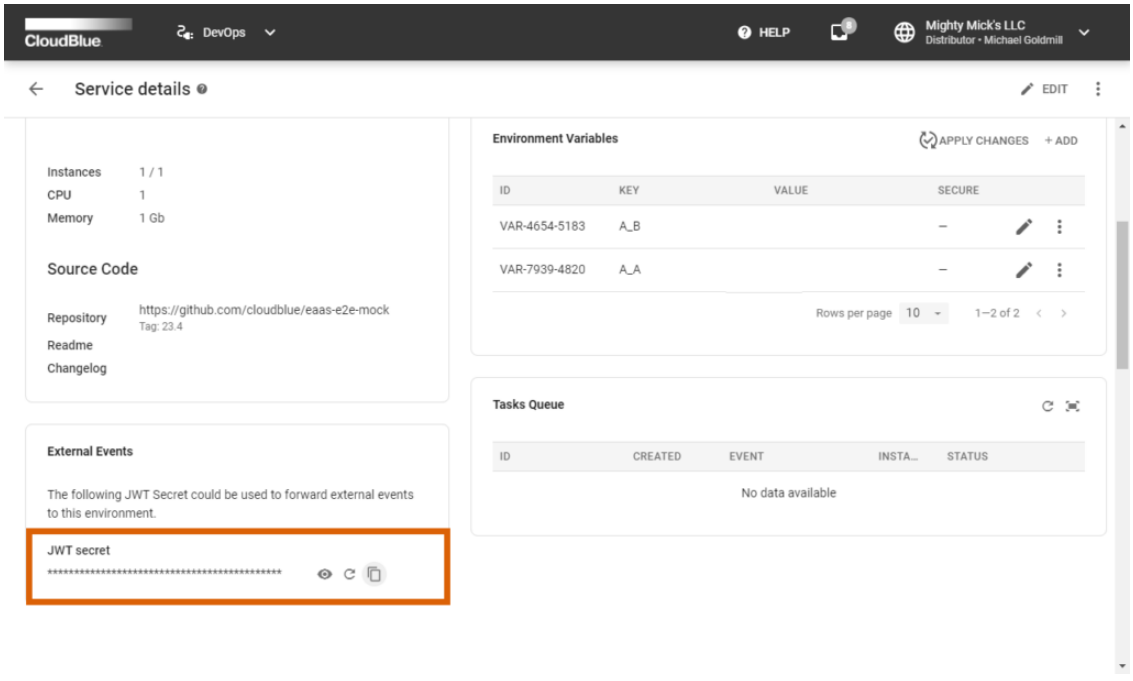
Custom (external) events are similar to product actions. The core difference, however, is that custom events require you to specify your product, select required environment, and define your operation.

Note that custom events have special endpoint in the Connect API. Thus, for example, you can configure a webhook for your custom event by using this endpoint. The endpoint URL looks like this: `https://{connect.domain}/public/v1/devops/hooks/{SRVC-XXXX-XXXX}/product/custom-event`. *Connect domain* represents your connect domain (e.g., portal.connect.com) and *SRVC* represents your added service identifier (e.g., SRVC-1234-4321).

An external system should return an authentication header that includes a bearer token. Namely, use the following code to include the required header: `headers['Authentication'] = 'Bearer {Your JSON Web Token}'`. Obtain the token by creating the following JWT claim:

```
{
     “product_id”: <your_product_id>,
     “environment”: (development|test|production),
     “exp”: <expiration (see jwt.io)>
}
```

Sign the claim with your JWT secret that is available within your environment on the Connect platform. Access the service details screen to locate your JWT secret:



The following example showcases a custom event function within the **extension.py** file:

```
def process_product_custom_event(self, request):
   self.logger.info(f'Custom event: {request}')
   sample_return_body = {
      "response": "OK"
      }
# There are two types of data for custom event functions: 1) JSON body (i.e.,the POST method) and 2)
formData object
# Required codes for both data types are presented below:
#1) request['body'] – contains the content of the POST method within the body
#2) request['form_data'] – includes required formData object
# Note that your function can include your specified external system header via this code:
request['headers']
# You can also access selected method by using this code: request['method']
# Furthermore, access the query string via the following code: request['querystring']
   return CustomEventResponse.done(body=sample_return_body)
```