https://cloudblue.com

Subscription Activation

Auto-generated at November 2, 2025

Overview

Once a customer creates a new fulfillment request, its corresponding subscription is generated on the CloudBlue Connect platform. The initial status of a generated subscription is *Processing*, while the initial status of a created fulfillment request is *Pending*. Note that subscriptions and fulfillment requests remain in their corresponding states until they are processed via the Connect platform or via your deployed middleware. In case a fulfillment request is successfully *approved*, the Connect platform also switches its associated subscription to the *Active* state.

Vendors can deploy the Processor SDK to approve pending fulfillment request and consequently activate a required subscription via their systems.

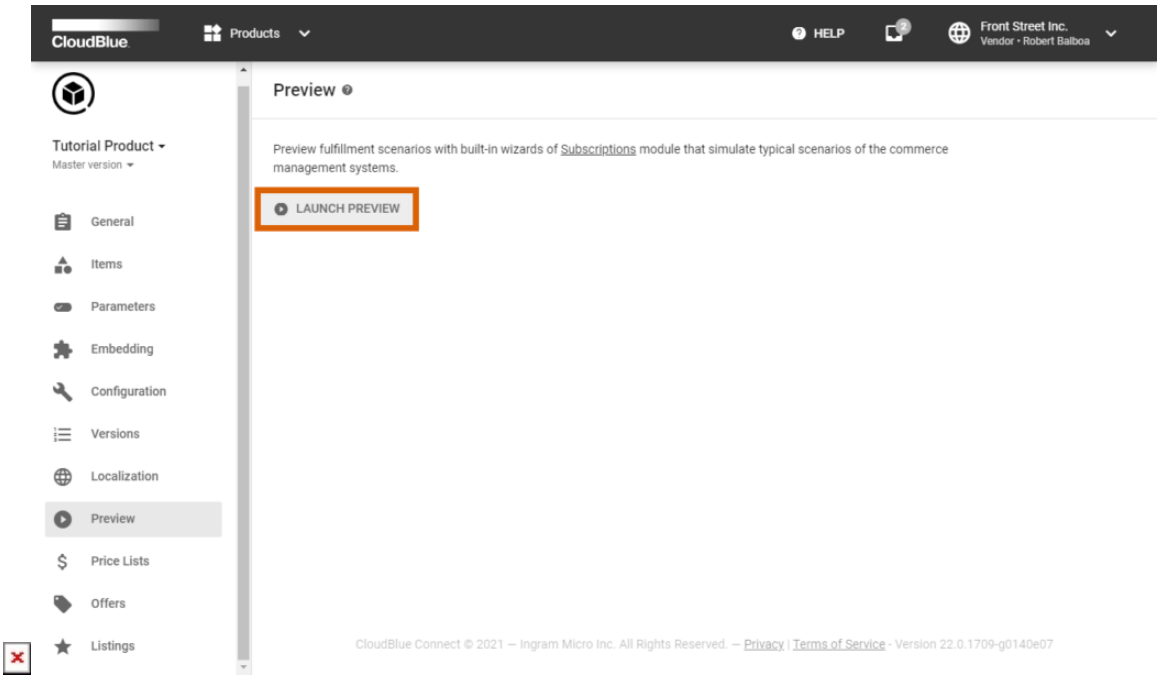
Prerequisites

Before following the provided use case, make sure that all of these prerequisites are met:

- Your product is defined on the Connect platform.
- The Processor SDK Template is successfully deployed.
- Your deployed Processor is properly configured.
- Vendor API configuration is presented.

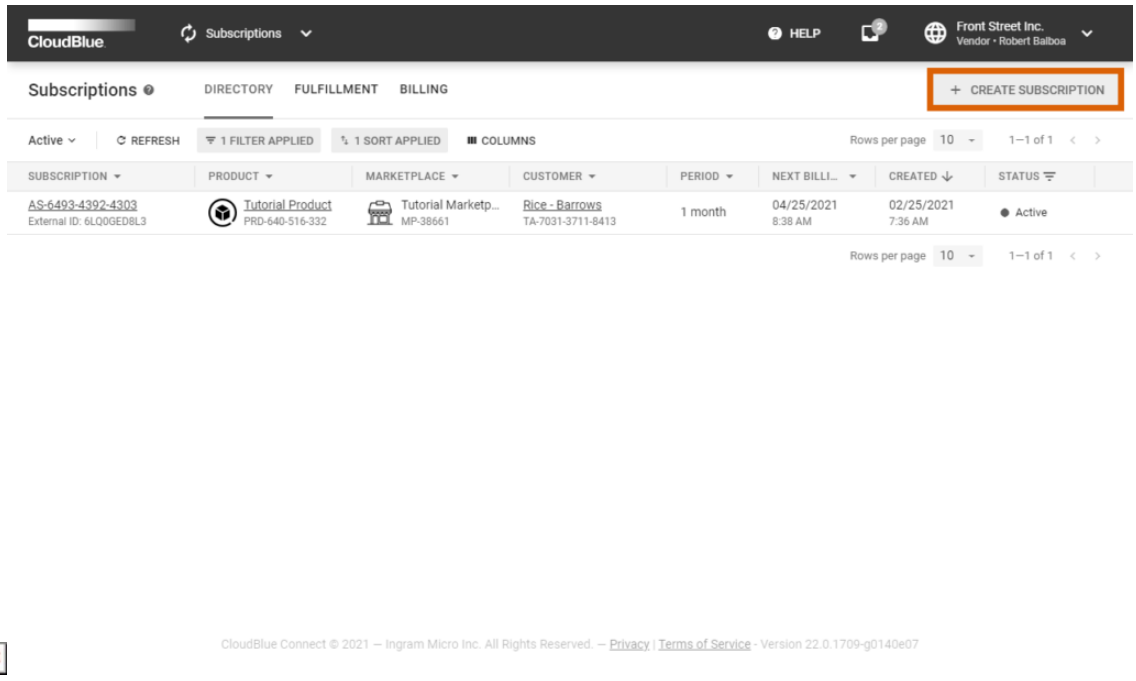
Create a Subscription

Fulfillment requests are created via specified Marketplaces in case your product is listed there. However, Vendors can create a preview subscription from the **Preview** section of the product profile page.



Alternatively, Vendors can create a preview subscription from the **Subscriptions** module. Follow the Subscription Creation

documentation for more details.



Code Snippets

The following code snippets represent reusable source codes with their detailed descriptions and customization instructions. Follow the provided code snippets to configure your deployed Processor SDK Template.

processor.py

Before starting to work with the **processor.py** file, make sure that your Connect Open API client and the **config.json** file are properly configured. Specify your API key and the API endpoint to establish connection with the platform. Note that the Connect Open API Client is used to fetch required requests from the Connect platform.

```
from cnct import ConnectClient
from datetime import datetime
from connect_processor.app.utils.globals import Globals
from connect_processor.app.utils.utils import Utils
from cnct import R

project_manager = __import__('connect_processor.app', globals(), locals(),
                             ['purchase', 'change', 'cancel', 'suspend', 'resume', 'tier_fulfillment',
'report_usage'],
                             0)


# The processor.py serves as an entry point for the Processor.
```

```
if __name__ == '__main__':

    # config_file represents the config.json file from the Processor SDK.
    config_file = Utils.get_config_file()
    # apiEndpoint is the API End-point of the Connect platform.
    connect_api_url = config_file['connectApiEndpoint'],
    # apiKey represents your API key that is created from the Integrations module of the Connect platform.
    connect_key = config_file['connectApiKey'],
    # Products that should be processed by this middleware are presented via a list of product IDs.
    client = ConnectClient(api_key=connect_key[0], endpoint=connect_api_url[0])

    # If your Product has a parameter of any 'Tier' scope, a Tier Configuration Request should be
processed
    # before a Fulfillment Request. Refer to the Tier Configuration Use Case from
    # the Connect Community page for more information.
    if bool(project_manager.tier_fulfillment):
        # Use the filter below to fetch a Tier-Config-Request (TCR) for your product.
        query_tcr = R()
        query_tcr &= R().configuration.product.id.oneof(Globals.PRODUCTS)
        # Note that the Processor works only with TCRs in 'pending' status.
        query_tcr &= R().status.oneof(['pending', 'inquiring'])
        # IMPORTANT: Remove 'inquiring' from the filter query. It's added to simplify debugs and unit
tests.
        tcrs = client.ns('tier').collection('config-requests').filter(query_tcr)
        # Process each TCR
        for tcr in tcrs:
            project_manager.tier_fulfillment.TierConfiguration.process_request(tcr, client)


    # Use this filter to fetch pending Fulfillment requests from Connect:
    query = R()
    query &= R().asset.product.id.oneof(Globals.PRODUCTS)
    query &= R().status.oneof(['pending'])

    # The following applies the filter.
    requests = client.collection('requests').filter(query)

    # Process each request:
    for request in requests:
        request_id = Utils.get_basic_value(request, 'id')
        request_status = Utils.get_basic_value(request, 'status')
        # Process all Fulfillment Requests in Pending state.
        if request_status == 'pending':
            # The following checks Fulfillment Request types.
            type = Utils.get_basic_value(request, 'type')
```

```
            if type == 'purchase':
                # The PURCHASE type refers to a new subscription creation on the Connect platform.
                if bool(project_manager.purchase):
                    project_manager.purchase.Purchase.process_request(request, client)

            if type == 'change':
                # The CHANGE type represents requests to change the subscribed Items or their quantity.
                if bool(project_manager.change):
                    project_manager.change.Change.process_request(request, client)

            if type == 'cancel':
                # The CANCEL type represents a request to terminate an existing subscription.
                if bool(project_manager.cancel):
                    project_manager.cancel.Cancel.process_request(request, client)

            if type == 'suspend':
                # The SUSPEND type refers to the subscription suspension requests.
                if bool(project_manager.suspend):
                    project_manager.suspend.Suspend.process_request(request, client)

            if type == 'resume':
                # The RESUME type represents requests to resume suspended subscriptions.
                if bool(project_manager.resume):
                    project_manager.resume.Resume.process_request(request, client)


    # The following checks if your product includes Pay-as-you-go items.
    # Refer to Usage Reporting Use Case from the Connect Community portal for more information.
    if bool(project_manager.report_usage):
        # Customize: usage reports are submitted to the Connect platform with your specified frequency.
        if datetime.today().day == Globals.DAY_TO_REPORT_USAGE:
            project_manager.report_usage.Usage(client).process_usage()
```

purchase.py

Create a **pyrchase.py** file and write a script that triggers specific Vendors API requests from the Vendor API endpoint. Therefore, Vendors can create and approve subscriptions from their systems. In case of a successful subscription activation, the Processor should save specified fulfillment parameters and return your customized Subscription Activation Template.

```
from connect_processor.app.utils.utils import Utils
from cnct import ConnectClient
from typing import Dict


class Purchase:
```

```
    # This class is used for new subscriptions in Connect.

    @staticmethod
    def process_request(request, client):
        # type: (Dict, ConnectClient) -> Dict
        # This method processes the Fulfillment Requests in Pending status.

        # Create a subscription in the vendor system by using the Vendor API.
        # The following represents the Mock API and API client to create subscriptions:
        # api_client = APIClient(api_url='',
        #                        api_key='')
        # data = {}
        # subscription_info = api_client.create_subscription(data=data)

        # The response for the subscription creation API call may have information that should be saved in
        # the fulfillment parameter.
        Purchase._save_fulfillment_parameters(request, client)

        return Utils.approve_fulfillment_request(request, client)
        # Therefore, fulfillment request will be Approved and subscription will be activated.
        # Note that in the fulfillment request will not be approved in case any mandatory/required
fulfilment parameter remains empty.

    @staticmethod
    def _save_fulfillment_parameters(request, client):
        # type: (Dict, ConnectClient) -> None
        # This method saves the fulfillment parameter data. Customize the payload params IDs with your
fulfillment parameters id and
        # the params values with the values given by the Vendor API.

        request_id = Utils.get_basic_value(request, 'id')
        # Update the fulfilment parameters in Connect with your required value.
        # Prepare the update parameters payload with the response of APIClient.create_subscription method.
        payload = {
            "asset": {
                "params": [
                    {
                        # Provide the fulfillment parameter ID as it is specified on the Connect platform.
                        "id": "subscription_id",
                        "value": "value for subscription ID in Vendor System",
                        "value_error": "",
                        "structured_value": ""
                    },
                    {
                        # Add other fulfillment parameters if necessary:
                        "id": "param_b",
```

```
                    "value": "value for parameter b",
                    "value_error": "",
                    "structured_value": ""
                }
            ]
        }
    }

    # Update the fulfillment parameter value:
    Utils.update_subscription_parameters(request_id, payload, client)
```

utils.py

The **utils.py** file contains all the utility methods. Configure this file as follows:

```
import json
from typing import Any, Dict
from cnct import ConnectClient
from connect_processor.app.utils.globals import Globals


class Utils:
    # Class for all static processor methods.

    @staticmethod
    def get_config_file():
        # type: () -> Dict[str, Any]
        # Loads the config file.
        with open("./config.json") as file_handle:
            config = json.load(file_handle)
        return config

    @staticmethod
    def get_item_by_id(items, item_id):
        # type: (Dict, str) -> object
        # Locates the item object by using ID.
        for item in items:
            if item['id'] == item_id:
                return item
        raise Exception('Item {id} not found.'.format(id=item_id))

    @staticmethod
    def get_basic_value(base, name):
        # type: (Dict,str) -> Any
        # Locates value by using the provided name.
```

```
        if base and name in base:
            return base[name]
        return '-'

    @staticmethod
    def get_value(base, prop, value):
        # type: (Dict,str,str) -> Dict
        if prop in base:
            return Utils.get_basic_value(base[prop], value)
        return {}

    @staticmethod
    def update_subscription_parameters(request_id, payload, client):
        # type: (str,object,ConnectClient) -> object
        # Updates fulfillment parameters values in Connect:
        fulfillment_request = client.requests.resource(request_id).update(payload=payload)
        return fulfillment_request

    @staticmethod
    def get_param_value(request, param_phase_value, param_name):
        # type: (Dict, str, str) -> object
        # Get the request value for the product param name:
        params = Utils.get_value(request, 'asset', 'params')
        for param in params:
            param_phase = Utils.get_basic_value(param, 'phase')
            if param_phase == param_phase_value and \
                    Utils.get_basic_value(param, 'name') == param_name:
                return Utils.get_basic_value(param, 'value')

    @staticmethod
    def approve_fulfillment_request(request, client):
        # type: (Dict, ConnectClient) -> Dict
        # Approves the request with the Client
        # Get the subscription product_id value:
        product = Utils.get_value(request, 'asset', 'product')
        product_id = Utils.get_basic_value(product, 'id')
        # Get the activation template ID:
        template_id = Utils._get_template_by_product(product_id, Globals.ACTIVATION_TEMPLATE_NAME,
'asset', client)
        # Prepare the payload as follows:
        payload = {"template_id": template_id}
        request_id = Utils.get_basic_value(request, 'id')
        # Approve the fulfillment request with your payload.
        return Utils._approve_fulfillment_request(request_id, payload, client)

    @staticmethod
```

```
    def _approve_fulfillment_request(request_id, payload, client):
        # type: (str,object,ConnectClient) -> Dict
        result = client.requests[request_id].action(name="approve")
        approve_result = result.post(payload=payload)
        return approve_result

    @staticmethod
    def reject_fulfillment_request(request_id, reason, client):
        # type: (str,str,ConnectClient) -> object
        # Rejects the fulfillment request. Its status will be switched to Failed.
        # Prepare the reject payload as follows.
        payload = {
            "reason": reason
        }
        result = client.requests[request_id].action(name="fail")
        reject_result = result.post(payload=payload)
        return reject_result

    @staticmethod
    def _get_template_by_product(product_id, template_name, template_scope, client):
        # type: (str,str, str, ConnectClient) -> str
        # Locates the template for your product and its scope
        template = client.collection('products')[product_id].templates.filter(name=template_name,
scope=template_scope).first()
        template_id = Utils.get_basic_value(template, 'id')
        return template_id

    @staticmethod
    def update_tier1_parameters(tcr_id, payload, client):
        # type: (str, object, ConnectClient) -> object
        # Updates the tier 1 parameters in fulfillment requests:
        tier_request = client.ns('tier').collection('config-requests')[tcr_id].update(payload=payload)
        return tier_request

    @staticmethod
    def approve_tier_config_request(request, tcr_id, client):
        # type: (Dict, str, ConnectClient) -> Dict
        # Approves the tier config request with the Client. Its status will be switched to Approved.
        # Get the tier config product_id
        product = Utils.get_value(request, 'configuration', 'product')
        product_id = Utils.get_basic_value(product, 'id')
        template_id = Utils._get_template_by_product(product_id,
Globals.TIER_CONFIG_ACTIVATION_TEMPLATE_NAME, 'tier1',
                                                     client)
        payload = {
            "template": {
```

```
                "id": template_id
            }
        }
        return Utils._approve_tier_config_request(tcr_id, payload, client)

    @staticmethod
    def _approve_tier_config_request(tcr_id, payload, client):
        # type: (str,object, ConnectClient) -> Dict
        # Approves the Tier Request. Its status will be switched to Approved.
        # The status of the Tier Config will be updated to Active.
        # The status of the Subscription will switched to to Pending.
        result = client.ns('tier').collection('config-requests')[tcr_id].action(name='approve')
        approve_result = result.post(payload=payload)
        return approve_result

    @staticmethod
    def set_inquiring_tier_config_request(tcr_id, client):
        # type: (str, ConnectClient) -> Dict
        # Assigns the Inquiring status to the Tier Request.

        return client.ns('tier').collection('config-
requests')[tcr_id].action(name='inquire').post(payload={})
```

Summary

As a result, specified fulfillment requests will be approved by the Processor. The CloudBlue Connect platform will also assign the *Active* status to the corresponding subscription.

Note, however, that in case a tier configuration request should be processed, the middleware process this tier configuration request before your fulfillment request.