

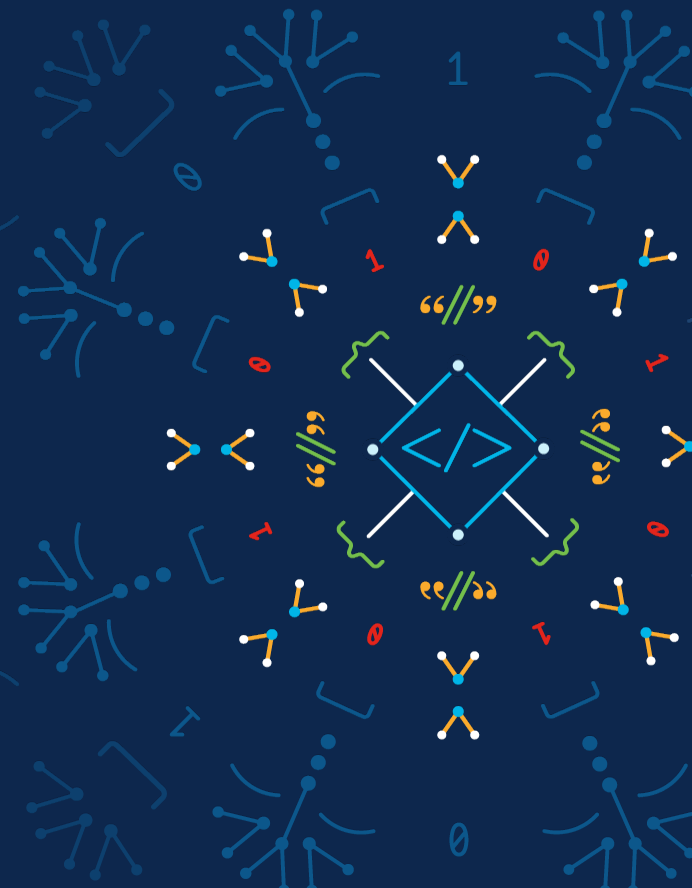
Batch Executor

NSO subscriber based batch code execution

Lenin Khaidem

Software Consulting Engineer

10th May 2022



Agenda

- What is it?
- It's applications
- Architecture – how does it work?
- Understand the different callbacks of the framework
- Development Steps
- Execution Operations
- Demo

What is it?

- ✓ Executes user defined NSO code for N items batches of size X
- ✓ Batch size is user defined or programtically identified
- ✓ Developer focuses on logic to execute on smaller scale
- ✓ Plug and play – NSO subscriber based package
- ✓ Calendaring – Execution in a specified slot on specific days

Some features include but not limited to

- Application specific user defined inputs
- Batch execution logs are stored against a request id
- Execution could be terminated or redeployed for a specific request id
- Global execution lock
- Success / failures items identification
- Re-try failures
- Package generator (be-make-package) included

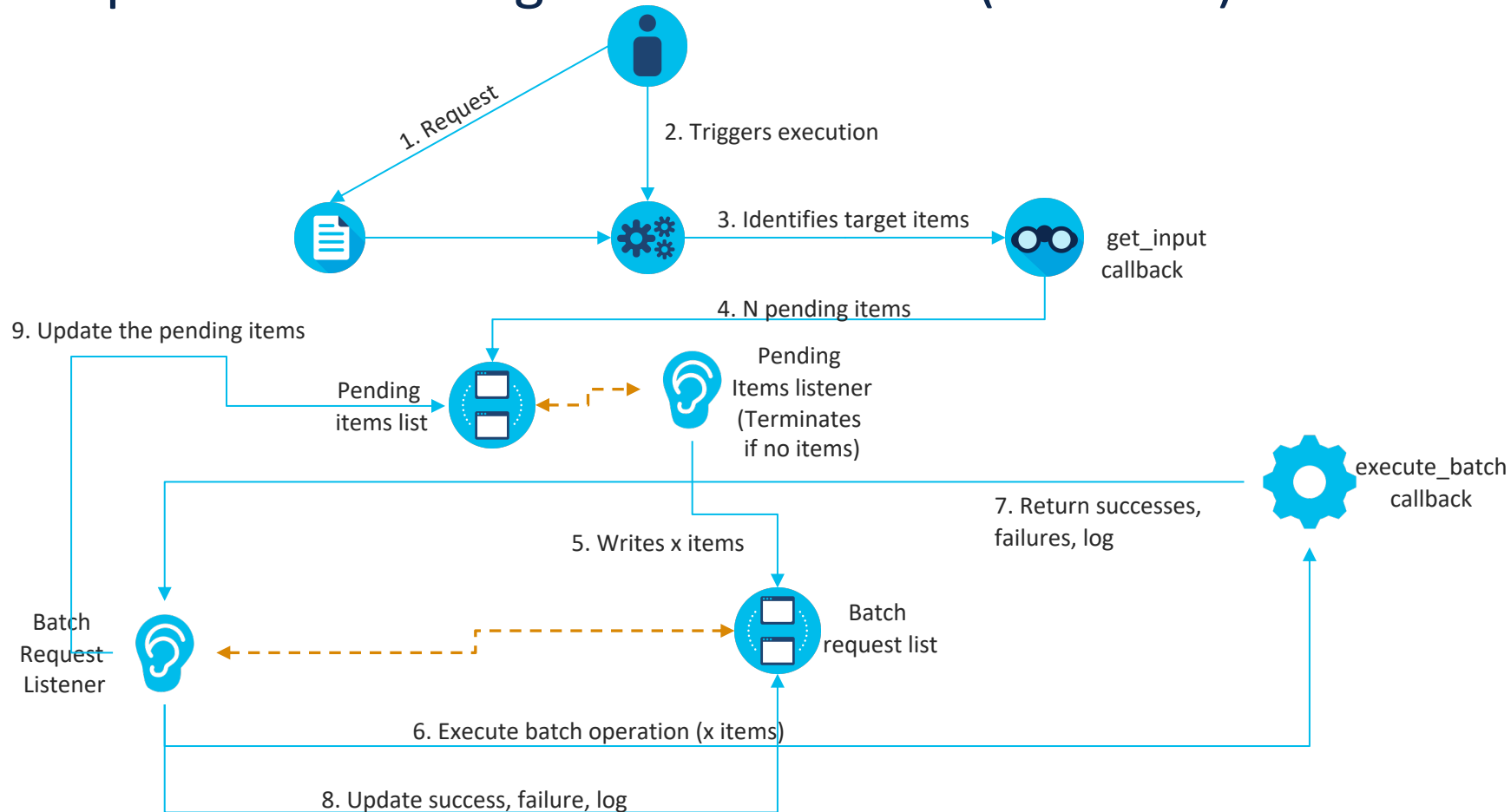
Batch executor can be used for

- Network wide topology discovery
- Service Inventory discovery
- NED migrate command execution
- Reconciliation code execution with no networking flag

Architecture



Implemented using NSO subscriber (listeners)



Understanding the callbacks defined in batch executor

get_input_items

```
7
8 class BeTestPkg(BatchExecutor):
9     def get_input_items(self, inputs) → list:
10         return []
11
```

- First callback
- Returns list of target items
- Could be devices or any attribute
- The execution ends when target items are processed
- User custom inputs are passed as argument

get_batch_items

```
class BeTestPkg(BatchExecutor):  
    def get_input_items(self, inputs) → list:  
        | return []  
  
    def get_batch_items(self, inputs, pending_items: list) → list:  
        | return []
```

- Overrides the default logic of getting batch items
- Programmatically identify batch items based on inputs / pending items
- Optional callback

execute_batch

```
class BeTestPkg(BatchExecutor):
    def get_input_items(self, inputs) → list:
        return []

    def get_batch_items(self, inputs, pending_items: list) → list:
        return []

    def execute_batch(self,
                     items: list,
                     inputs: dict) → Tuple[list, list, Union[str, dict,
                     list]]:

        logs = {}
        success_items = []
        failure_items = []
        return success_items, failure_items, logs
```

- Callback where user code is executed
- Items – batch items
- Could call any RFS / Action code
- User code identifies success & failure items
- Batch execution log could be captured
- Returns success, failures, log

Development Steps



Method 1- Generate a new package

```
[root@bru-dockerhost-1 packages]# be-make-package --help
version: 1.3
/root/.pyenv/versions/3.8.10/bin/be-make-package [-g | -a | -u ]
  [-g | --generate] Generate a package
  [-a | --augment] Augment to an existing package
  [-u | --update-lib] Update be Package Library
```

```
[root@bru-dockerhost-1 packages]# be-make-package -g
***** Batch Executor Package Generation Utility - 1.3 *****
Package Name: batch-ned-migrate
Package Description: batch ned migration
```

```
├── batch-executor
│   ├── load-dir
│   │   └── batch-executor.fxs
│   ├── package-meta-data.xml
│   ├── python
│   │   ├── batch_executor
│   │   │   ├── batch_executor.py
│   │   │   ├── __init__.py
│   │   │   ├── main.py
│   │   │   └── utils.py
│   ├── README
│   └── src
│       ├── Makefile
│       └── yang
│           └── batch-executor.yang
├── batch-ned-migrate
│   ├── package-meta-data.xml
│   ├── python
│   │   ├── batch_ned_migrate
│   │   │   ├── batch_ned_migrate.py
│   │   │   └── __init__.py
│   └── src
│       ├── Makefile
│       └── yang
│           └── batch-ned-migrate.yang
└── templates
    └── batch-ned-migrate.xml
```

12 directories, 15 files

Method 2- Augment to an existing nso package

```
[root@bru-dockerhost-1 packages]# be-make-package -a  
  
***** Batch Executor Package Generation Utility - 1.3 *****  
  
Package Name: batch-l3-inventory-discovery  
Package Description: sample package  
Parent Package Name (in current dir): simple-l3vpn-service
```

```
simple-l3vpn-service  
├── load-dir  
│   └── simple-l3vpn-service.fxs  
├── package-meta-data.xml  
├── python  
│   └── batch_l3_inventory_discovery  
│       ├── batch_l3_inventory_discovery.py  
│       └── __init__.py  
├── src  
│   ├── Makefile  
│   └── yang  
│       ├── batch-l3-inventory-discovery.yang  
│       └── simple-l3vpn-service.yang  
├── templates  
│   └── simple-l3vpn-service-template.xml  
└── test  
    ├── internal  
    │   ├── lux  
    │   │   ├── basic  
    │   │   │   ├── Makefile  
    │   │   │   └── run.lux  
    │   │   └── Makefile  
    │   └── Makefile  
    └── Makefile
```

10 directories, 13 files

How to model custom yang inputs

```
23     augment "/batch-executor:batch-executor" {
24         container batch-l3-inventory-discovery {
25             uses batch-executor:request-grouping {
26                 // augmenting is optional
27                 // all custom inputs should be augmented to request/inputs
28                 augment "request/inputs" {
29                     leaf some-input {
30                         type string;
31                     }
32                 }
33             }
34         }
35     }
```

Execution operations

A committed batch request accepts a few action based operations

- Operation is executed from the context of the request
- Execute – triggers the execution
- Terminate – terminate ongoing operation gracefully
- Re-try failures – re-execute the batch execution only for the failed items
- Redeploy – re-execute the batch operation for all the identified target items

Demo





The bridge to possible