



a division of ciena

Blue Planet BPO-Core Developer Guide

Blue Planet Release 17.06
August 2017

Table of Contents

Introduction	1
Feedback.....	1
Hello World Tutorial	2
Prerequisites.....	2
Steps	2
Main Concepts	14
Instance Catalog	15
Extensible Type Catalog	16
Domains and Resource Providers	17
External vs. Internal vs. Pure Composite Resources.....	18
External Resources	19
Built-in Resources	19
Pure Composite Resources	20
Service Templates	20
Products and Product Catalog	21
Resource Types vs. How They Are Realized.....	22
Resource Instance Lifecycle.....	22
Template Engine	23
Asset Manager.....	23
Summary.....	24
Reference Documentation	25
Definition Modules.....	26
File Encoding	26
File Structure.....	26
Processing	29
Example.....	30
Model Definitions Directory Structure	32
Resource Types	35
Service Templates	45
Service Template Directives	63
Imperative Plans	97
Capability Types.....	107
Relationship Types	109
Onboarding.....	111
Model Definitions.....	112
Onboarding Failures	121
Product Catalog	122
Product Instances	122
Instance Catalog	126
Resource Instances.....	126
Relationship Instances	140
Lifecycle Operations	144
Activate.....	147

Auto Clean	148
Update.....	149
Terminate	150
Failure	151
Mixing Imperative and Declarative Lifecycle Operations	152
Imperative Activate	153
Declarative Activate.....	154
Template Engine	155
Overview	155
Declarative and Imperative Lifecycle Operations.....	155
Lifecycle Plans.....	156
Update Concurrency	160
Resource Updates.....	161
Resource Configuration Update.....	161
Resource Observed State Update	162
REST Methods for Resource Updates - PUT versus PATCH.....	163
Updating Orchestration States.....	164
Updating Discovered Resources	164
Validation Operations	165
Built-in Validation.....	165
Custom Validation Operations.....	165
Resource Validation APIs	166
Custom Operations	169
Components of Custom Operations	169
Typical Workflow	170
Resource Operation Lifecycle.....	174
Resource State Constraints	176
Policy Enforcement.....	179
Table of Contents	179
Common Policy Structure.....	179
Policy System.....	180
Policy Condition Definition Language	181
Authorization Policies	183
Event Policies	188
Policy Database.....	200
Authorization Policy Fields	202
Tutorials	212
Requirements.....	212
Overview of the Tutorials	212
Getting Started.....	212
Beginner	212
Intermediate	212
Advanced	213
Setup Instructions for the Tutorials	214
Docker Installation.....	214
Git Setup	214

Python Setup	214
Bootstrap Steps for bpocore-dev and Environment.....	216
Running the frost-orchestrate-ui-dev Docker container.....	218
bpocore CLI Bootstrap Tutorial	220
Hello World Tutorial.....	222
Prerequisites	222
Steps	222
Web Callout Tutorial	234
Steps	234
Further Notes	240
Allocating Numbers in a Number Pool Tutorial	241
Steps	241
How to Reuse Property Types in Resource Type Definitions	244
Steps	244
Using bpocore CLI for fast onboarding.....	247
Steps	247
First OpenStack Tutorial	251
Steps	251
Further Notes	271
Hello RP Tutorial	272
Steps	272
Service Template Tutorial.....	278
Steps	279
Developing Remote Plans with script-dev	316
Running script-dev.....	316
Create Definition Modules	316
Onboard and Create a Resource	320
Debugging Scripts	322
Multi-Tenant Tutorial: Managing Customers as Separate Tenants.....	324
Overview	324
Prerequisites	325
Steps	325
Blue Planet Orchestrate REST APIs.....	335
Overview	335
API General Content Sections	336
Getting Started.....	337
Authentication	338
Filtering	342
Sorting.....	348
Pagination.....	349
SSL Certificates	352
REST Component Overview	353
Application	354
Asset Manager.....	355
Authentication	356
Component Registry	357

Diagnostics	358
Events	359
Import/Export	360
Market Component	367
Policy Manager	379
Resource Provider	380
Rest-Server	382

Introduction

The Blue Planet Orchestrator (BPO) is a modular, extensible, multi-domain orchestrator. This Developer Guide includes overviews, concepts, tutorials, and an overview of BPO's APIs:

Hello World Tutorial

A simple example for the impatient.

Main Concepts

An overview of how BPO models the world.

Reference Documentation

Detailed documentation of BPO concepts, components, and APIs.

Tutorials

For BPO APIs and workflows ranging from beginner to advanced.

REST APIs

- [Getting Started](#) – Overview and common information generally applicable to the APIs.
- [REST Component Overview](#) – Overview of each BPO REST component.

NOTE

For details of each API endpoint, refer to the Swagger API documentation, which can be found in the Blue Planet UI by selecting the **System** tab, then **Documentation > API**.

Docker Registry, Vendor, and Version Conventions

NOTE

- Throughout the documentation, solutions, applications, and instructions refer to fully-qualified names that include a specific docker registry. Normally, the registry *bpdr.io* is used and the vendor *blueplanet* is used. If running in a different environment, substitute the appropriate registry before executing the commands.
- Throughout the documentation, the software packages, solutions, applications, and instructions reference specific versions of software which are accurate at the time of writing. These versions may not be the most current or appropriate versions to use. It is often necessary to substitute current or appropriate versions when executing the commands or instructions in a tutorial.

Feedback

If you have questions, comments, or changes to contribute, you can contact us at blueplanet.com.

Hello World Tutorial

This tutorial serves as the "Hello World" example for bpocore. If you complete this tutorial you will know how to:

- run bpocore in "dev-ops" mode
- onboard and instantiate service templates
- instantiate, query, and teardown resource instances

Prerequisites

Some minimal setup is required to have an environment in which you can follow the steps below. See [Setup Instructions for the Tutorials](#).

Steps

1. If this is the very first time you have used bpdr.io

```
docker login --password="secret" --username="bp2user" \
--email="bp2@ciena.com" bpdr.io
```

2. Run bpocore in a terminal separate from the other steps

```
docker pull bpdr.io/blueplanet/bpocore-dev:1.7.0-8289-bcff141
docker run --name bpocore-dev -it --rm bpdr.io/blueplanet/bpocore-dev:1.7.0-
8289-bcff141
```

3. Set up constants, aliases, environment, etc.

```
BPOCORE_CID=$(docker ps | grep "bpocore-dev" | cut -f 1 -d ' ')
BPOCORE_IP=$(docker inspect --format '{{ .NetworkSettings.IPAddress }}' \
bpocore-dev)
MARKET_URL="http://$BPOCORE_IP:8181/bpocore/market/api/v1"
ASSETS_URL="http://$BPOCORE_IP:8181/bpocore/asset-manager/api/v1"
BRANCH_NAME="helloworld"
ssh-keygen -R $BPOCORE_IP
ssh-keyscan -H $BPOCORE_IP >> ~/.ssh/known_hosts
```

4. Upload your public key

```
PUBLIC_KEY=`cat ~/.ssh/id_rsa.pub`
curl -s -H "Content-Type: application/json" \
-d "{\"key\": \"$PUBLIC_KEY\"}" $ASSETS_URL/keys | python -m json.tool
```

5. Clone the definitions repository and checkout a private branch from the **production** branch

```
git clone ssh://git@$BPOCORE_IP:22/home/git/repos/model-definitions  
DEFINITIONS_DIR=$PWD/model-definitions  
cd $DEFINITIONS_DIR  
git checkout -b $BRANCH_NAME origin/production
```

6. Add your service template

```
cd $DEFINITIONS_DIR/types/tosca
mkdir -p example
cat <<EOF > example/hello_world.tosca
"\$schema"      = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-
module#"
title          = "A very simple service built from a template"
package        = example
version        = "1.0"
description    = """Defines a Hello World service template."""
authors        = ["Royal Joe (royal.joe@ciena.com)"]

resourceTypes {

    HelloWorld {
        title = Hello World
        description = "Hello World"
        derivedFrom = tosca.resourceTypes.Root
        properties {
            property1 {
                title = Property 1
                description = "Some property of the service"
                type = string
            }
            property2 {
                title = Property 2
                description = "Some other property of the service"
                type = string
            }
        }
    }
}

serviceTemplates {

    HelloWorld {

        title = Hello World
        description = "Hello World"
        implements = example.resourceTypes.HelloWorld

        resources {
            // various random strings modeled as subresources
            dbName.type = tosca.resourceTypes.RandomString
            dbUser.type = tosca.resourceTypes.RandomString
            dbUserPw.type = tosca.resourceTypes.RandomString
            dbRootPw.type = tosca.resourceTypes.RandomString
        }
    }
}
EOF
```

7. Commit and push your changes

```
git add .
git commit -m "Add Hello World definitions"
git push origin $BRANCH_NAME
```

8. Submit a pull request

```
cat <<EOF > pullrequest.json
{
  "branch": "$BRANCH_NAME",
  "title": "Add Hello World definitions",
  "comment": ""
}
EOF
curl -s -H "Content-Type: application/json" \
  -d @pullrequest.json $ASSETS_URL/areas/model-definitions/pullrequests \
  | python -m json.tool
```

Let's record the pull request ID for later use:

```
PULL_REQUEST_ID='<USE "requestId" FROM OUTPUT ABOVE>'
```

9. Wait for the pull request to be "accepted"

```
curl -s $ASSETS_URL/areas/model-definitions/pullrequests/$PULL_REQUEST_ID \
  | python -m json.tool
```

10. Create a corresponding product

```
cat <<EOF > product.json
{
  "resourceTypeId": "example.resourceTypes.HelloWorld",
  "title": "Hello World",
  "active": true,
  "domainId": "built-in",
  "providerData": {
    "template": "example.serviceTemplates.HelloWorld"
  }
}
EOF
curl -s -H "Content-Type: application/json" \
  -d @product.json $MARKET_URL/products | python -m json.tool
```

Let's record the product ID for later use:

```
PRODUCT_ID='<USE "id" FROM OUTPUT ABOVE>'
```

11. Instantiate a resource for the product

```
cat <<EOF > resource.json
{
    "productId": "$PRODUCT_ID",
    "label": "Hello World 0001",
    "properties": {
        "property1": "foo",
        "property2": "bar"
    }
}
EOF
curl -s -H "Content-Type: application/json" -d @resource.json
$MARKET_URL/resources | python -m json.tool
```

Let's record the resource ID for later use:

```
RESOURCE_ID='<USE "id" FROM OUTPUT ABOVE>'
```

12. Query for the resource and its dependencies

```
curl -s $MARKET_URL/resources/$RESOURCE_ID | python -m json.tool
curl -s $MARKET_URL/resources/$RESOURCE_ID/dependencies | python -m json.tool
```

13. Teardown the resource

```
curl -X DELETE -s $MARKET_URL/resources/$RESOURCE_ID
curl -s $MARKET_URL/resources/$RESOURCE_ID | python -m json.tool # THE RESOURCE
WILL BE GONE
```

Example

The output below show the tutorial steps being run on the command line along with their output. This is only an example; the identifiers you see will be different in some cases.

```
#terminal 1

$ docker pull bpdr.io/blueplanet/bpocore-dev:1.7.0-8289-bcff141
Pulling repository bpdr.io/blueplanet/bpocore-dev
f83c82570fa8: Download complete
511136ea3c5a: Download complete
27d47432a69b: Download complete
5f92234dcf1e: Download complete
51a9c7c1f8bb: Download complete
5ba9dab47459: Download complete
c02320a6f792: Download complete
83a4c3577cb8: Download complete
bcd874cca324: Download complete
```

```

d3595ab7f891: Download complete
5f9d73ebfdcb: Download complete
0b53bd122b09: Download complete
c961e9577e85: Download complete
b7caebcbce04: Download complete
7df33a28b8d8: Download complete
2d0e812fc8cb: Download complete
cd06b7d4bc9b: Download complete
31386af49137: Download complete
79783377caf9: Download complete
e95e24a8914a: Download complete
c6caale22c60: Download complete
b0466c5c44db: Download complete
caf82d1bal6f: Download complete
f08d5b6bc355: Download complete
cfb8c63dfa83: Download complete
9b2a6886d8fc: Download complete
994c055178b0: Download complete
4a649e137d9a: Download complete
5324bfe7ae9a: Download complete
c32826eabe3d: Download complete
786e482e13c0: Download complete
Status: Downloaded newer image for bpdr.io/blueplanet/bpocore-dev:1.7.0-8289-
bcff141

$ docker run --name bpocore-dev -it --rm bpdr.io/blueplanet/bpocore-dev:1.7.0-8289-
bcff141

# terminal 2

$ BPOCORE_CID=$(docker ps | grep "bpocore-dev" | cut -f 1 -d ' ')
$ BPOCORE_IP=$(docker inspect --format '{{ .NetworkSettings.IPAddress }}' bpocore-
dev)
$ MARKET_URL="http://$BPOCORE_IP:8181/bpocore/market/api/v1"
$ ASSETS_URL="http://$BPOCORE_IP:8181/bpocore/asset-manager/api/v1"
$ BRANCH_NAME="helloworld"

$ ssh-keygen -R $BPOCORE_IP
/home/gbolt/.ssh/known_hosts updated.
Original contents retained as /home/gbolt/.ssh/known_hosts.old
$ ssh-keyscan -H $BPOCORE_IP >> ~/.ssh/known_hosts
# 172.16.0.119 SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2
# 172.16.0.119 SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2

$ PUBLIC_KEY=`cat ~/.ssh/id_rsa.pub`
$ curl -s -H "Content-Type: application/json" -d "{\"key\": \"$PUBLIC_KEY\"}"
$ ASSETS_URL/keys | python -m json.tool
{
    "id": "c573acf773a6affed69df2ff4c2ef2df",
    "key": "ssh-rsa"
AAAAB3NzaC1yc2EAAAQABAAQDAREvQTFujLmXDZu2H8p1ZKKaLS3JyOMiljV5A3rqWpi21I0PHxjt
4BoieKXnyQMeTRLnMk6J+pb3o3YM3ZzVt0EQbGK06Zk3zRgcdxNvZmi/+/_DbcxquimMGPv4zVoy+rk0Pyu3
abxSWVrrlyX2jxKXyNDrj0zLRhlgsM4GO7PAhrTfAOwjm8yscme6ziC1p3Iadt2fcXcC7qKt174PpnNN3Ez
x/2KYVJro3NNQW0EBpcKu9ns01Qmq7Lng36W1dco/QgwyVdNdsBFMLjDJK9cPMZe5DJocsVsQpFfIdAfvt
UPuA4804K2VV1PxhhtZhUxjCkk6/yiKus1B+etWP gbolt@GBOLT-6530-Ubuntu"
}

$ git clone ssh://git@$BPOCORE_IP:22/home/git/repos/model-definitions
Cloning into 'model-definitions'...

```

```
remote: Counting objects: 303, done.
remote: Compressing objects: 100% (171/171), done.
remote: Total 303 (delta 128), reused 301 (delta 127)
Receiving objects: 100% (303/303), 598.39 KiB | 0 bytes/s, done.
Resolving deltas: 100% (128/128), done.
Checking connectivity... done.

$ DEFINITIONS_DIR=$PWD/model-definitions
$ cd $DEFINITIONS_DIR
$ git checkout -b $BRANCH_NAME origin/production
Branch helloworld set up to track remote branch production from origin.
Switched to a new branch 'helloworld'

$ cd $DEFINITIONS_DIR/types/tosca
gbolt@GBOLT-6530-Ubuntu:/tmp/hello/model-definitions/types/tosca$ mkdir -p example
$ cat <<EOF > example/hello_world.tosca
> "\$schema"      = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-module#"
> title          = "A very simple service built from a template"
> package         = example
> version         = "1.0"
> description     = """Defines a Hello World service template."""
> authors         = ["Royal Joe (royal.joe@ciena.com)"]
>
> resourceTypes {
>
>   HelloWorld {
>     title = Hello World
>     description = "Hello World"
>     derivedFrom = tosca.resourceTypes.Root
>     properties {
>       property1 {
>         title = Property 1
>         description = "Some property of the service"
>         type = string
>       }
>       property2 {
>         title = Property 2
>         description = "Some other property of the service"
>         type = string
>       }
>     }
>   }
> }
>
> serviceTemplates {
>
>   HelloWorld {
>
>     title = Hello World
>     description = "Hello World"
>     implements = example.resourceTypes.HelloWorld
>
>     resources {
>       // various random strings modeled as subresources
>       dbName.type = tosca.resourceTypes.RandomString
>       dbUser.type = tosca.resourceTypes.RandomString
>       dbUserPw.type = tosca.resourceTypes.RandomString
>       dbRootPw.type = tosca.resourceTypes.RandomString
>
```

```
>      }
>    }
>  }
> EOF

$ git add .

$ git commit -m "Add Hello World definitions"
[helloworld 354abec] Add Hello World definitions
 1 file changed, 46 insertions(+)
 create mode 100644 types/tosca/example/hello_world.tosca

$ git push origin $BRANCH_NAME
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 888 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To ssh://git@172.16.0.119:22/home/git/repos/model-definitions
 * [new branch]      helloworld -> helloworld

$ cat <<EOF > pullrequest.json
> {
>   "branch": "$BRANCH_NAME",
>   "title": "Add Hello World definitions",
>   "comment": ""
> }
> EOF

$ curl -s -H "Content-Type: application/json" -d @pullrequest.json
$ASSETS_URL/areas/model-definitions/pullrequests | python -m json.tool
{
  "comment": "",
  "commitDate": "2016-01-28T01:52:58.000Z",
  "commitHash": "354abec5c6285bb1a76cdfac9828f508be929b21",
  "details": [],
  "email": "royal.joe@ciena.com",
  "reason": "",
  "requestDate": "2016-01-28T01:53:35.776Z",
  "requestId": "f3cb359c-0691-4309-822c-06e3d0567be0",
  "status": "pending",
  "title": "Add Hello World definitions"
}

$ PULL_REQUEST_ID=f3cb359c-0691-4309-822c-06e3d0567be0

$ curl -s $ASSETS_URL/areas/model-definitions/pullrequests/$PULL_REQUEST_ID |
python -m json.tool
{
  "comment": "",
  "commitDate": "2016-01-28T01:52:58.000Z",
  "commitHash": "354abec5c6285bb1a76cdfac9828f508be929b21",
  "details": [],
  "email": "royal.joe@ciena.com",
  "productionCommitHash": "354abec5c6285bb1a76cdfac9828f508be929b21",
  "reason": "",
  "requestDate": "2016-01-28T01:53:35.776Z",
```

```
"requestId": "f3cb359c-0691-4309-822c-06e3d0567be0",
  "status": "accepted",
  "title": "Add Hello World definitions"
}

$ cat <<EOF > product.json
> {
>   "resourceTypeId": "example.resourceTypes.HelloWorld",
>   "title": "Hello World",
>   "active": true,
>   "domainId": "built-in",
>   "providerData": {
>     "template": "example.serviceTemplates.HelloWorld"
>   }
> }
> EOF

$ curl -s -H "Content-Type: application/json" -d @product.json $MARKET_URL/products
| python -m json.tool
{
  "active": true,
  "constraints": {},
  "domainId": "built-in",
  "id": "56a974f3-f657-449f-899a-87eb2e9241ed",
  "providerData": {
    "template": "example.serviceTemplates.HelloWorld"
  },
  "resourceTypeId": "example.resourceTypes.HelloWorld",
  "title": "Hello World"
}

$ PRODUCT_ID=56a974f3-f657-449f-899a-87eb2e9241ed

$ cat <<EOF > resource.json
> {
>   "productId": "$PRODUCT_ID",
>   "label": "Hello World 0001",
>   "properties": {
>     "property1": "foo",
>     "property2": "bar"
>   }
> }
> EOF

$ curl -s -H "Content-Type: application/json" -d @resource.json
$MARKET_URL/resources | python -m json.tool
{
  "autoClean": false,
  "createdAt": "2016-01-28T01:55:52.144Z",
  "desiredOrchState": "active",
  "differences": [],
  "discovered": false,
  "id": "56a97528-49e4-4665-ac31-96ff0600081c",
  "label": "Hello World 0001",
  "orchState": "requested",
  "productId": "56a974f3-f657-449f-899a-87eb2e9241ed",
  "properties": {
    "property1": "foo",
    "property2": "bar"
  }
}
```

```
        },
        "providerData": {},
        "reason": "",
        "resourceTypeId": "example.resourceTypes.HelloWorld",
        "shared": false,
        "tags": {},
        "tenantId": "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738",
        "updatedAt": "2016-01-28T01:55:56.144Z"
    }

$ RESOURCE_ID=56a97528-49e4-4665-ac31-96ff0600081c

$ curl -s $MARKET_URL/resources/$RESOURCE_ID | python -m json.tool
{
    "autoClean": false,
    "createdAt": "2016-01-28T01:55:52.144Z",
    "desiredOrchState": "active",
    "differences": [],
    "discovered": false,
    "id": "56a97528-49e4-4665-ac31-96ff0600081c",
    "label": "Hello World 0001",
    "orchState": "active",
    "productId": "56a974f3-f657-449f-899a-87eb2e9241ed",
    "properties": {
        "property1": "foo",
        "property2": "bar"
    },
    "providerData": {},
    "reason": "",
    "resourceTypeId": "example.resourceTypes.HelloWorld",
    "shared": false,
    "tags": {},
    "tenantId": "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738",
    "updatedAt": "2016-01-28T01:55:56.200Z"
}

$ curl -s $MARKET_URL/resources/$RESOURCE_ID/dependencies | python -m json.tool
{
    "items": [
        {
            "autoClean": false,
            "createdAt": "2016-01-28T01:55:52.407Z",
            "desiredOrchState": "active",
            "differences": [],
            "discovered": false,
            "id": "56a97528-a449-412a-897f-a221d868d586",
            "label": "Hello World 0001.dbName",
            "orchState": "active",
            "productId": "c9bfce3c-149e-4495-b861-94fe801bdf6e",
            "properties": {
                "format": "hexadecimal",
                "length": 32,
                "value": "a67af51988c1a75a4a5f93298a388741"
            },
            "providerData": {},
            "reason": "",
            "resourceTypeId": "tosca.resourceTypes.RandomString",
            "shared": false,
            "tags": {}
        }
    ]
}
```

```
        "tenantId": "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738",
        "updatedAt": "2016-01-28T01:55:53.070Z"
    },
    {
        "autoClean": false,
        "createdAt": "2016-01-28T01:55:52.433Z",
        "desiredOrchState": "active",
        "differences": [],
        "discovered": false,
        "id": "56a97528-a897-4ebf-98df-487033998629",
        "label": "Hello World 0001.dbUserPw",
        "orchState": "active",
        "productId": "c9bfce3c-149e-4495-b861-94fe801bdf6e",
        "properties": {
            "format": "hexadecimal",
            "length": 32,
            "value": "504b1d528d77d7e94025cccaab7b5cdc"
        },
        "providerData": {},
        "reason": "",
        "resourceTypeId": "tosca.resourceTypes.RandomString",
        "shared": false,
        "tags": {},
        "tenantId": "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738",
        "updatedAt": "2016-01-28T01:55:53.271Z"
    },
    {
        "autoClean": false,
        "createdAt": "2016-01-28T01:55:52.476Z",
        "desiredOrchState": "active",
        "differences": [],
        "discovered": false,
        "id": "56a97528-dd2f-4b2c-9d46-75d1186e59a5",
        "label": "Hello World 0001.dbRootPw",
        "orchState": "active",
        "productId": "c9bfce3c-149e-4495-b861-94fe801bdf6e",
        "properties": {
            "format": "hexadecimal",
            "length": 32,
            "value": "caac32c11565e2a5ae24c8eebacd0e2b"
        },
        "providerData": {},
        "reason": "",
        "resourceTypeId": "tosca.resourceTypes.RandomString",
        "shared": false,
        "tags": {},
        "tenantId": "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738",
        "updatedAt": "2016-01-28T01:55:53.275Z"
    },
    {
        "autoClean": false,
        "createdAt": "2016-01-28T01:55:52.440Z",
        "desiredOrchState": "active",
        "differences": [],
        "discovered": false,
        "id": "56a97528-2743-488b-906f-68fa32cf9f8e",
        "label": "Hello World 0001.dbUser",
        "orchState": "active",
        "productId": "c9bfce3c-149e-4495-b861-94fe801bdf6e",
```

```
        "properties": {
            "format": "hexadecimal",
            "length": 32,
            "value": "1210d637b50df56cd5e96731ce828973"
        },
        "providerData": {},
        "reason": "",
        "resourceTypeId": "tosca.resourceTypes.RandomString",
        "shared": false,
        "tags": {},
        "tenantId": "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738",
        "updatedAt": "2016-01-28T01:55:53.239Z"
    }
],
"offset": 0,
"total": 4
}

$ curl -X DELETE -s $MARKET_URL/resources/$RESOURCE_ID

$ curl -s $MARKET_URL/resources/$RESOURCE_ID | python -m json.tool
{
    "autoClean": false,
    "createdAt": "2016-01-28T01:55:52.144Z",
    "desiredOrchState": "terminated",
    "differences": [],
    "discovered": false,
    "id": "56a97528-49e4-4665-ac31-96ff0600081c",
    "label": "Hello World 0001",
    "orchState": "terminating",
    "productId": "56a974f3-f657-449f-899a-87eb2e9241ed",
    "properties": {
        "property1": "foo",
        "property2": "bar"
    },
    "providerData": {},
    "reason": "",
    "resourceTypeId": "example.resourceTypes.HelloWorld",
    "shared": false,
    "tags": {},
    "tenantId": "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738",
    "updatedAt": "2016-01-28T01:57:01.250Z"
}

$ curl -s $MARKET_URL/resources/$RESOURCE_ID | python -m json.tool
{
    "failureInfo": {
        "detail": "",
        "reason": "Resource '56a97528-49e4-4665-ac31-96ff0600081c' not found"
    },
    "statusCode": 404
}
```

Main Concepts

BPO has a (logically) centralized model of the orchestrated (controlled) "world". This model describes all relevant aspects of the orchestrated domains, and all control is initiated via interacting with the model. This concept is very similar to traditional network management, except now the scope of orchestration is wider and can go beyond pure networking.

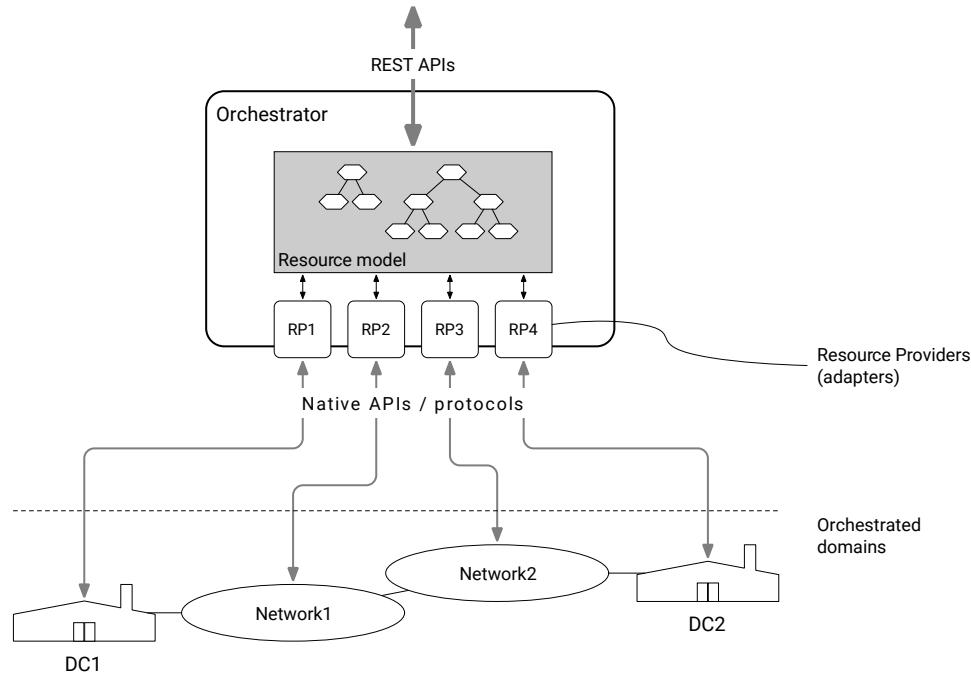


Figure 1. Simplified conceptual diagram on role of resource model

Of course the above illustration is a grand simplification. So below is a somewhat more accurate diagram, showing just enough details to illustrate the text that follows it.

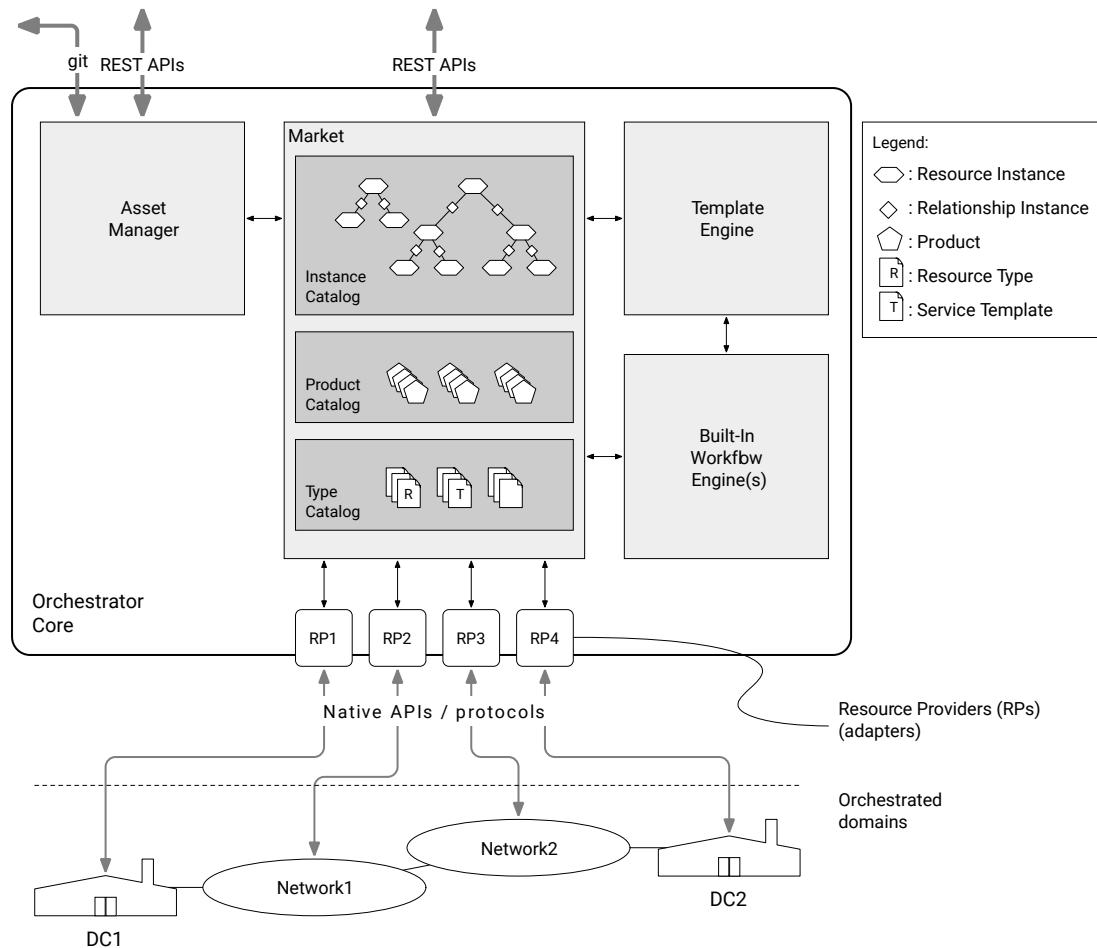


Figure 2. Key BPO components

Instance Catalog

A central component in BPO's model is a *Resource Instance*, or simply *Resource*. A Resource is the same concept as the managed object in traditional network management. Resources are uniquely identified by their universally unique ID (UUID). Resources have attributes; some are generic, some are specific to the resource type. The generic attributes always exist no matter what the resource type is. Type specific attributes are called *properties* and defined for each resource type.

Unlike many other systems, in BPO's model relationships between resources (containment, dependency, etc.) are expressed by another first class object, the *Relationship Instance*, or simply *Relationship*. A Relationship describes how a stated *Requirement* of one Resource is connected to the stated *Capability* of another Resource. Therefore, a Relationship always contains:

- a source descriptor tuple (Resource ID + requirement of the source)
- a target descriptor tuple (Resource ID and capability of the target).
- a relationship type identifier

In addition to source and target, a Relationship can have type specific properties based on the

relationship type.

Resources and Relationships are both stored and governed by BPO's *Instance Catalog*. The Instance Catalog is part of *Market*, which holds a few additional interlinked catalogs as described later.

The Instance Catalog can be regarded as a graph database. It provides extensive APIs to query and manipulate Resources and Relationships. The Instance Catalog is designed to model and track a very large number of Resources and Relationships. Both storage and processing can be scaled horizontally over arbitrary number of servers, taking advantage of commodity compute and storage.

In the following sub-sections we cover:

- introducing new types of Resources and Relationships into the [type catalog](#)
- the relationships between a Resource, the underlying [Product](#) and the provider [Domain](#)
- using Resources to directly represent external entities in the respective domain
- using Resources to act as of other Resources

Extensible Type Catalog

In most traditional management and orchestration solutions, the system is designed to deal with a fixed set of domain types and functionality. These systems typically have a fix set of built-in resource types, and it is often cumbersome to expand functionality to cover new types of domains and/or features.

BPO has a run-time extensible *Type Catalog*. The Type Catalog contains, among other things, the definition of all *Resource Types* and *Relationship Types*. Here is the full list of artifact types defined in the Type Catalog:

ARTIFACT TYPE	PURPOSE
Resource Type	Defines the properties, capabilities, requirements, custom operations, etc.
Relationship Type	Defines properties and connectivity constraints for a Relationship Type
Property Type	These property definitions can be re-used across many Resource and Relationship Types
Capability Type	Defines abstract Capabilities and Requirements
Alarm Type	Properties of an alarm type
KPI Type	Meta-data for a key performance indicator (KPI) type

Most of the above artifact types support a simple form of multiple inheritance. Inheritance provides the following important practical advantages:

- Composition: types can be cleanly composed from well-defined cohesive base types.
- Re-use: inheritance allows the re-use of base behaviors in a multitude of other types, avoiding meta-data duplication.

- Abstract, generalized filtering: instances can be searched by their types, including any of their base types.

The catalog can be expanded, modified, evolved "on-the-fly": no software upgrade, or stopping/restarting of any Blue Planet component is necessary. When updating the Type Catalog with additive or backward compatible changes, all existing system services continue to operate without any interruption or degradation.

Later we cover:

- How the content of the Type Catalog comes from the [Asset Manager](#).
- The process of changing the Type Catalog is often referred as part of [Onboarding](#).

Domains and Resource Providers

The main intent of the Orchestrator is to orchestrate the life-cycle, configuration, and behavior of one or more external (orchestrated) domains. Some examples of external domains:

- A cloud instance such as an OpenStack or VMware cloud
- A WAN NMS system or controller
- An SDN controller
- An individual or a group of physical or virtual network elements (these can be of the same family, or modeled similarly)
- Virtually anything that has an API and needs to be controlled as part of service orchestration

The granularity of domains typically aligns with administrative and authorization boundaries: a given API endpoint through which the domain can be accessed and managed with provided access credentials defines the domain boundary. For instance, a VMware cloud with a distinct API end-point and credentials is regarded as a Domain.

We mentioned it earlier that the Instance Catalog serves as the sole model of the orchestrated "world" and all control is exerted via interacting with this model via APIs. The behavioral coupling between Resource/Relationship instances and the external domains are implemented by domain-appropriate adapters, which we call *Resource Providers (RPs)*.

RPs are responsible for:

- Protocol adaptation
- Model adaptation: the native model of the external device vs. the Resource and Relationship model of BPO.

RPs are analogous to device drivers in an operating system (also to Element Adapters in an Element/Network Management System):

- An RP is a piece of code and/or execution unit which is purposely written to couple some external domains with its domain-specific model and domain-specific access protocol to the universal Resource model of BPO.
- They participate in the flow of information in both top-down and bottom-up directions:
 - In the top-down direction, they propagate life-cycle and configuration events.
 - In the bottom-up direction, they discover resources, detect state changes and discrepancies between intended and actual configuration, propagate performance indicators, etc.

A single RP can interface with one or more domains, typically of the same kind. This is analogous with a device driver which can attach multiple network interfaces or storage devices of the same device family to the OS.

A crucial feature of BPO is that new RPs can be added (on-boarded) to BPO at any time without stopping the system or interrupting its services. The protocol between bpocore and the RP is described in the RP-SDK documentation.

Ultimately, each Resource in the Instance Catalog is traced to exactly one domain and one RP. Relationship instances are indirectly tied to domains and RPs via the source and target Resources.

In order to track such linkages, both domain and RP entities have a simple object representation in the Instance Catalog of Market (refer to the illustration above):

- The Resource Provider objects keep track of existing RP instances. This is based on registration: when a new RP "shows up", it registers itself with the system, and as a result a new Resource Provider objects gets added to Market.
- A Domain object represents a given external domain. (We use capitalized Domain to refer to such object, while lower-case "domain" refers to the actual external domain).

Each Domain object links to its Resource Provider. Domain objects are actively managed via Market's REST API. By adding a new Domain object and linking it to an RP object, the respective RP is instructed to connect to the domain described in the Domain object, and start its synchronization process.

In summary:

- Each Resource is linked to one Domain (this linkage is indirect, and will be described below in the [Products](#) section).
- Each Domain is linked to one Resource Provider.
- Each Relationship is linked to one or two Domains, via the source and target Resources it points to.

External vs. Internal vs. Pure Composite Resources

Depending on how a Resource relates to the external world (by external we mean the world outside the Orchestrator, in the managed domains), we can distinguish between three categories of Resources:

- External Resources
- Built-in Resources
- Pure Composite Resources

These are described below.

External Resources

These are Resources that have a one-to-one correspondence to some logical or physical entity in one of the orchestrated domains. The following are obvious examples:

- Network element
- Shelf
- Physical port
- Termination point
- Virtual machine
- Data center
- Virtual network in a data center
- A Heat Stack
- An end-to-end carrier grade Ethernet connection (E-Line)

External Resources are always tied to external Domains and mediated via Resource Providers (RPs).

Built-in Resources

These are special Resources that BPO can manage without relying on an actual RP. Built-in Resources serve as useful building blocks in writing composite (templated) Resources, albeit they can be used in other situations.

Currently BPO provides the following built-in Resource Types:

RESOURCE TYPE	PURPOSE
Noop	A No-Operation Resource Instance, which can be used to store arbitrary data without any implied behavior.
CatalogSearch	Provides direct query access to the Catalogs to implement complex queries and allow sharing the result with other Resources.
WebCallout	Supports "stateful" HTTP Web call-outs, i.e., a distinct call-out on each life-cycle events: create, update, terminate.
RandomString	Can be used to generate random strings of customizable style and length.
NumberPool	To manage a pool of continuous range of integer numbers. Useful to manage VLAN pools, GRE/VXLAN id pools, etc.

RESOURCE TYPE	PURPOSE
PooledNumber	Represents a number that is "checked out" from a given NumberPool.
IpV4AddressPool	To manage a pool of IP v4 addresses (defined by a CIDR prefix)
PooledIpV4Address	Represents an IP address or prefix checked out from a given IpV4AddressPool
Monitor	Monitors the Kafka message bus for specific events

Pure Composite Resources

A composite Resource is a resource that is:

- an aggregation of other external, built-in, and/or composite resources, and
- has no direct one-to-one relationship to any external entities, thus
- it solely represents a logical construct that exists only in the Orchestrator.

A pure composite Resource is loosely the equivalent of a Heat "Stack" in OpenStack.

Composite Resources can represent logical construct that span across multiple external domains.

They are the main form of abstraction, aggregation and/or decomposition:

- *Abstraction*: Composite Resources can represent abstract services, such as end-to-end connectivity across multiple domains, high-touch network services with L3-L7 processing (implemented in VNFs or PNFs), etc.
- *Aggregation*: Composite Resources can aggregate small parts into bigger constructs with fewer details, thus hide some of the complexities of the small building blocks.
- *Decomposition*: The opposite direction of aggregation is decomposition: when designing a complex service, it can often be decomposed into its natural components, which in turn can be decomposed to smaller sub-components, etc. This process can be repeated until we deal with Resources that are simple enough to be handled as external or built-in resources.

In the case of a composite Resource built from parts, the parts are often referred as its Sub-Resources.

Service Templates

One way of defining composite Resources is via *Service Templates*. A Template is a data-driven recipe from which a generic template engine can create and manage the Sub-Resources automatically. Templating allows rapid and robust definition of composite Resources.

BPO has an easy to understand *declarative* templating language, inspired by both TOSCA and Cloud Formation/Heat. It allows the service designer to directly define what Sub-Resources should make up the Service (host Resource), and how the Sub-Resources should be constructed (input attributes, build order, etc.).

In addition to declarative templating, BPO supports imperative (scripted) operations. This style can be used as an alternative to declarative templating, but it can also be used to implement custom operations.

BPO's Template Engine is regarded as a built-in RP managing all templated Resource Instances.

The [Definition Modules](#) reference and [Service Template](#) reference contain extensive text on templating.

The [Tutorials](#) section also provides many examples of templating.

Products and Product Catalog

As we described above, for every Resource Instance, there is exactly one Domain (and one RP) responsible for implementing behavior. This is BPO's "built-in domain" for the templated and built-in Resources, while external Resources are backed by their respective domain-facing RP.

The linkage between Resources and Domains is recorded in an intermediary object, called *Product*:

- Each Resource is linked to exactly one Product.
- Each Product is linked to exactly one Domain.

Products are stored in the *Product Catalog*, which has its own set of APIs. The Product Catalog also is part of Market.

Why the name? In real life, products tie together providers and consumers: Providers advertise products in a market, where consumers can see them, browse among them, and select which product they want to use. There can be many providers offering similar or even the same products. Some markets keep track of what the consumer bought and even allow returning the goods later on. Think about Amazon or any other on-line retailers.

This analogy applies closely to the orchestration world:

- The RPs are the *providers*.
- They populate the Product Catalog, i.e., *advertise*.
- Each Product is linked to a Resource Type, which is the basis of the *product specification* for the advertised product. Products can imply/express additional constraints on the Resource Type that they can instantiate.
- The API users of Market are the *consumers*.
- The Instance Catalog can be viewed as the catalog of *sold goods*.
- Instantiating and terminating Resources can be viewed as *buying/ordering and returning goods*.

Therefore, in addition to tying Resources to Domains, Products have a few additional key roles:

- Products tie Resources to their Resource Types.
- Products represent the system's ability to instantiate new Resources/services. In fact, RPs populate the Product Catalog to announce to the rest of the system that they are able to offer such and such

Resource Types for instantiation.

- Each product is identified by a UUID.

Think about it as the SKU of the cyberworld of orchestration. "Consumers" consume (instantiate) resources by referring to specific products. Hence, the Product ID is one of the mandatory input arguments in a Resource creation operation. Once the Resource instance is created, it is permanently tied to the Product from which it was created.

Products also serve the primary basis of access control for resource creation: authorization policies can be tied to products, hence controlling who can create what types of resources over what domains.

In addition to Products being auto-populated by the RPs, the admin/operator of BPO can create Products for Composite Resources. This is done via the Product Catalog APIs (see examples in the tutorials). For a Product that is created to manage templated Resources, the Product is linked to the "built-in" domain, and the Product contains the unique ID of the Service Template to be used.

Note: In future releases of BPO, the Market will have the ability to express the quantitative availability of a Product within a Domain which will impact the ability to "sell" (instantiate) new Resource Instances from the given Product.

Resource Types vs. How They Are Realized

It is now timely to point out that a given Resource Type does not in any way constrain how it is realized. Resource Types are analogous to C/C++ declarations or Java interface classes. Just as an interface class can be realized by many concrete implementations, the same Resource Type could be realized (offered) by many Products, tied to different providers (RPs), or even to different recipes in case of templated composite Resources.

So a Resource Type is not inherently composite or externally managed: the same Resource Type can be implemented by an RP as an external Resource, and can be implemented by a template as a composite resource. The difference is made by the respective Product.

Resource Instance Lifecycle

When a new Resource is requested (via a POST HTTP call, see the [API reference](#)), the Product ID must be passed in. The new instance is added to the catalog. Its `desiredOrchState` (the desired orchestration state) is set to `active` by default. This indicates the caller's intent for the resource to be activated right away. At the instant of creation, the `orchState` (this is the actual orchestration state) is unspecified. However, the database event triggers an asynchronous notification to the RP behind the Product. The RP is responsible to "wake up" and fulfill the desired state, i.e., activate the resource. In case of an external domain this typically involves creating things or configuring things in the external domain. For built-in resource types, internal handlers fulfill the request. In the templated case, the [Template Engine](#) is notified.

Irrespective of what the RP is, it shall attempt to activate the Resource. If and when this is accomplished, it marks the Resource's `orchState` as `active`. In case of failing to activate, the `orchState` is marked

as failed and an explanation is provided in the `reason` attribute of the Resource.

The [Lifecycle Operations](#) section explains in more detail how the lifecycle of a resource is managed.

Template Engine

The Template Engine acts as the RP for the "built-in" Domain for Products that reference a [service template](#) in the `providerData` attribute. [Lifecycle Operations](#) of resources derived from such products are executed according to the referenced service template, which may specify *declarative* or *imperative* implementations. Declarative templates specify all three [activate](#), [update](#), and [terminate](#) operations using the [template DSL](#), while [imperative plans](#) provide an external script (executable).

Either way, the Template Engine monitors the outcome of the operation and eventually updates the Resource `orchState`:

- A successful operation is reflected in the Resource's attributes and the `orchState` as appropriate (`active` or `terminated`); or
- A failed operation sets `orchState` to `failed` and an explanation is stored in the `reason` attribute of the Resource.

See the service template [tutorial](#) and [reference documentation](#) for more on this topic.

Asset Manager

The purpose of Asset Manager is to allow bringing files (digital assets) into BPO in an organized, controlled, and audited way. The open-source version control system [Git](#) is used internally in BPO to achieve this. Git allows:

- Version control of all assets
- Audit trail of who changed what
- Diffing various versions
- Rollback to older versions
- Branching/forking for exploratory or concurrent work
- Many many other forms of collaboration among various actors who need to tailor BPO.

Such digital assets can be organized in so-called Asset Areas (or just Areas). Each Area is managed independently, i.e., has its own `git` repository.

The list of Asset Areas is currently managed by a system configuration file. (In a future release an API may be added.) The only Asset Area defined by default is `model_definitions`, which holds all type definitions and is the primary input to the Type Layer of Market:

Internally, components can be stake-holders of a given Asset Area: Market is a stake-holder to the Area `model_definitions`.

Git allows branching, and branches can have very different content. One branch must be regarded as the official version of the files to be consumed by all stakeholders. This branch, by default, is called *production*. All other branches can be freely manipulated via the external git APIs.

Currently the (only) way to alter the content is:

1. Clone a given asset area from the Asset Manager into the workspace (local filesystem) of the user
2. Checkout a new private branch from the latest version on the *production* branch
3. Make changes locally
4. Commit the changes locally
5. Push the changes on the private branch back to Asset Manager
6. Submit a *Pull Request (PR)* to the Asset Manager from the pushed branch.

The processing done as a consequence of a new pull request is described in the [Onboarding](#) section.

Summary

To summarize all entities introduced here we provide their entity diagram:

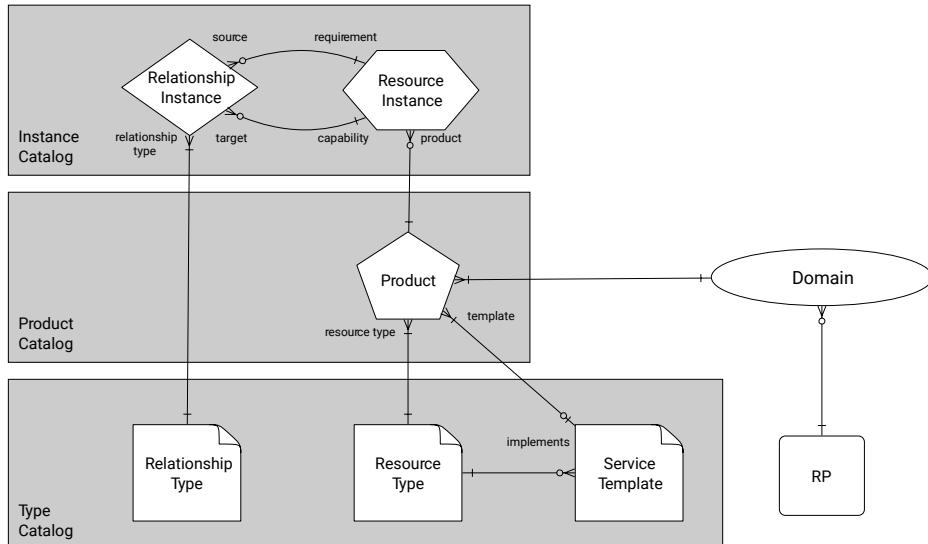


Figure 3. Entity diagram

Reference Documentation

This section provides information on BPO concepts, sub-components, APIs, etc.

The Reference section has the following chapters:

- **Main Concepts** – The [Main Concepts](#) reference is the best place to start.
- **Concepts and Components** – After you have an understanding of the main concepts, see the following sections for more detailed descriptions of the concepts and components:
 - [Product Catalog](#) – the Product Catalog within Market.
 - [Instance Catalog](#) – the Instance Catalog within Market.
 - [Lifecycle Operations](#) – activate, update and terminate lifecycle operations on Resource Instances.
 - [Validation Operations](#) – how validation of inputs to operations on Resource Instances works.
 - [Custom Operations](#) – defining non-lifecycle operations on Resource Instances.
 - [Policy Enforcement](#) – management of Policies and their enforcement.
 - [Asset Manager](#) – stores all Definition Modules.
 - [Onboarding](#) – how to introduce Definition Modules into the system.
- **Definition Modules** – describes the files used to define Resource Types and Service Templates.

If you are interested in specific definition types, see:

- [Resource Types](#) – schema definitions for resource and service types.
- [Service Templates](#) – implementation templates for services (including [Imperative Plans](#)).
- [Capability Types](#) – schema definitions for capabilities offered by resource types.
- [Relationship Types](#) – schema definitions for types of resource relationships.

Definition Modules

Definition Module files (just "Definition Modules" for short) contain artifacts which define data models (e.g, Resource Types) or behaviour (e.g, Service Templates) in BPO. This information model is based on the [Topology and Orchestration Specification for Cloud Applications \(TOSCA\) OASIS Standard](#).

Since the TOSCA standard is still under development, the BPO model is not completely aligned with TOSCA. In the future the BPO model will align as closely as possible with TOSCA. At this time, the main difference between the models is only in naming: instead of "Node Types", the BPO model uses "Resource Types". Other concepts such as Relationship Types and Capability Types are the same.

Some familiarity with JSON-Schema is useful (but not required) to understand the structure of Definition Modules. For more information on JSON-Schema, see <http://json-schema.org>.

Definition Modules are stored in the Model Definitions area in [Asset Manager](#). The directory layout of the Model Definitions area is described in [Model Definitions Directory Structure](#). For more information on how they arrive there, see [Onboarding](#).

File Encoding

Definition Modules can be encoded using JSON or HOCON.

HOCON is a superset of JSON and their data models are the same, e.g, you can express basic types like strings and integers as well as lists and objects. Both lists and objects can be nested. For more information on HOCON, see the official [HOCON Reference](#).

We recommend you author Definition Modules in HOCON as it is more readable than JSON. Also, HOCON supports a few useful features over JSON, e.g:

- Comments
- Multi-line/block strings with triple quotes
- Substitution and merging of content inside the same file

File Structure

Definition Modules are essentially JSON objects conforming to a certain schema. The schema can be found in the `json-schemas/tosca-lite-v1/definition-module.hocon` file in the Model Definitions area. When in doubt, consult this file.

The following table describes the attributes of Definition Modules.

ATTRIBUTE	TYPE	REQUIRED	PURPOSE
<code>"\$schema"</code>	string	yes	Identifies the schema to which the Definition Module conforms.

ATTRIBUTE	TYPE	REQUIRED	PURPOSE
title	string	yes	Used for display purposes in UIs.
description	string	yes	Used for help text and other places where detailed information is displayed on UIs.
package	string	yes	Provides namespace for the artifacts in the Definition Module.
version	string	yes	Provides versioning for the artifacts in the Definition Module.
authors	string	no	Provides list of authors (e.g, names and e-mail addresses) of the Definition Module.
imports	object	no	Provides aliases for artifacts defined in other Definition Modules.
artifact definition	object	no	Contains the main content of the Definition Module. See Artifact Definitions below.

The "\$schema" Attribute

The "\$schema" attribute must always be set to "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-module#". This uniquely identifies the supported Definition Module schema. The schema definition can be found in the `definition-module.hocon` file in the `json-schemas/tosca-lite-v1/` sub-directory of the Model Definitions area.

The version Attribute

The `version` attribute defines the version of the artifacts inside the Definition Module. The `version` attribute format should be `<major>.<minor>`. This version identifier is assigned by the Definition Module's author and is independent of the BPO software and release version.

The package Attribute (and FQNs)

The `package` attribute provides a namespace for all of the artifacts defined in the Definition Module. The same `package` may be re-used across multiple Definition Modules. This concept is similar to what you find in programming languages like Java.

By namespacing artifacts in a Definition Module, the `package` value is used to form the fully qualified name (FQN) of the artifacts. This means the `package` value must conform to the format described in the [Artifact References](#) section.

The FQN of an artifact has the following structure: `<package>.<artifactType>.<artifactName>` and is case sensitive. The These components are described in the table below.

COMPONENT	DESCRIPTION
<code><package></code>	The value of the <code>package</code> attribute in the Definition Module containing the artifact.

COMPONENT	DESCRIPTION
<artifactType>	The type of artifact. This will be one of the artifact types referenced in the Artifact Definitions section.
<artifactName>	The name of the artifact given by the author.

BPO's model is based on [TOSCA](#) so the built-in types are contained in the `tosca` package. These types are shipped with BPO and can be found in the Model Definitions area under the `types/tosca` directory.

While the package name does not have to align with the directory layout, by convention all Definition Modules from a particular package are stored under a sub-directory of `types/tosca`, e.g., `types/tosca/openstack`.

Imports

Imports allows you to avoid specifying the entire FQN each time you reference the artifact. If you find yourself referencing the same artifact many times in a single Definition Module, consider adding an import for it.

You may reference artifacts from other Definition Modules without using imports. If the artifact is not imported, it must be referenced with the FQN (e.g. `tosca.resourceTypes.Root`).

Artifact Definitions

Artifact Definitions are the main content of Definition Modules. You can define multiple types of artifacts and multiple instances of each artifact type inside the same file. The ordering of the artifacts does not matter.

ATTRIBUTE NAME	PURPOSE
<code>resourceTypes</code>	Contains Resource Types definitions.
<code>propertyTypes</code>	Contains Property Types definitions.
<code>capabilityTypes</code>	Contains Capability Types definitions.
<code>relationshipTypes</code>	Contains Relationship Types definitions.
<code>serviceTemplates</code>	Contains Service Templates definitions.

By convention, all artifacts should be named using CamelCase format, e.g, a Resource Type representing a Virtual Network Function be named `VirtualNetworkFunction`.

Title and Description

All Artifact Definitions include `title` and `description` attributes. These attributes are used solely for display purposes in UIs.

By convention we use Title Case for the `title` attribute with spaces to separate words. The `title` value

length is limited to 32 characters. This is to ensure the value can be clearly displayed in UIs. For the `description` attribute, we recommend meaningful phrases describing the object being modeled or defined. The `description` may be used as help text in UIs. If you cannot provide a useful description, prefer an field empty (e.g, "") instead of re-supplying the `title` (or some derivative of it).

Custom Property Naming

In many cases, Artifact Definitions contain sections for custom property definitions. Typically these sections are defined as an attribute called `properties`. Custom property names must conform to the following regular expression: `^ [a-zA-Z] [-_a-zA-Z0-9]* $`, i.e, they must:

- Start with a letter.
- Only contain alpha-numeric characters or "-" or "_" .

The reason for this restriction is that property definitions may be used by tooling to generate code in programming languages which place restrictions on member names, function names, etc.

Artifact References

References to artifacts in other Definition Modules may be written using either FQNs or aliases (described in the [Imports](#) section). FQNs must conform to the following regular expression: `^ [a-zA-Z] (\.\. [a-zA-Z] [-_a-zA-Z0-9]*)*`, i.e, they may consist of one or more (dot-separated) names which must:

- Start with a letter.
- Only contain alpha-numeric characters or "-" or "_" .

References to artifacts in the same Definition Module must be written using the FQN.

Processing

Definition Modules are processed by BP components when they are on-boarded. All Definition Modules undergo the same validation process initially. First they are checked to ensure valid JSON/HOCON syntax. Then they are validated against the `definition-module.hocon` schema. At this point the artifacts inside the Definition Modules are "compiled" into intermediate representations (IRs). Further artifact type specific validation is then performed against the IRs.

The compilation step which converts artifacts from the "raw" Definition Module format into the IR involves:

- Detecting duplicate artifacts
- Checking and normalizing artifact references
- Checking cyclic references
- Checking artifact inheritance

Example

```
"$schema"      = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-module#"
title         = Example Definition Module
package        = com.acmetel
version        = "1.0"
description    = This document defines the Physical abstract resource type.
authors        = [ "Joe Big (jbig@someinc.com)" ]

imports {
    Root = tosca.resourceTypes.Root
}

resourceTypes {

    Physical {
        derivedFrom = tosca.resourceTypes.Root
        title = Physical
        abstract = true
        description = """
                    This resource type is to be used as base type for physical resources.
                    """
    }

    properties {

        address {
            title      = "Address"
            description = "Street address of the resource"
            type       = string
            optional   = true
            updatable  = true
        }

        locator {
            title      = "Locator Info"
            description = "Additional information to locate the resource (within the
address)"
            type       = string
            optional   = true
            updatable  = true
        }

        latitude {
            title      = "Latitude"
            description = "GPS latitude coordinate of the resource"
            type       = number
            optional   = true
            updatable  = true
        }

        longitude {
            title      = "Longitude"
            description = "GPS longitude coordinate of the resource"
            type       = number
            optional   = true
            updatable  = true
        }
    }
}
```

Model Definitions Directory Structure

This reference describes the layout of the Model Definitions area. The Model Definitions area is managed by the [Asset Manager](#) component.

Here is the high-level directory structure without files:

```
---- json-schemas
  -   ---- market-v1
  -   ---- tosca-lite-v1
  ---- types
    -   ---- tosca
    -   ---- ddui
      -       ---- views
  ---- domain-types
    -   ---- ddui
      -       ---- views
  ---- ui-schema
    -   ---- domain-types
      -       ---- icons
    -       ---- icons
```

Typical development involves adding, removing, and modifying files under the `types/tosca/` and `ui-schema/` sub-directories. The `json-schemas/` sub-directory contains files which define the schema of the [Definition Modules](#) and related artifacts.

Unless specified otherwise, you can add new files or sub-directories under the root. They will be ignored.

json-schemas / (**read-only**)

You should regard the entire content of this directory (and all sub-directories) as "for your information". The files here are used by BPO's schema validators and related toolchains. This is where BPO's implementation looks for these files. You can use these files as reference.

You find two files on the top:

- `jsonschema.metaschema.draft-04.json` - the "official" root draft-04 JSON-Schema.
- `jsonschema.metaschema.draft-04-strict.json` - a stricter version of draft-04, rejecting all foreign constructs not recognized as JSON-Schema. We use this schema to validate all other JSON-Schema snippets.

json-schemas/market-v1 / (**read-only**)

This directory contains the JSON-Schema files that define Market specific objects. Currently this is limited to [Product Constraints](#) objects.

json-schemas/tosca-lite-v1/ (**read-only**)

The files in this directory are referred to as "meta-schema" in BPO. These are the files which define the schema of all of our TOSCA-based artifacts contained in the types/tosca/ directory tree (see below). These files conform to the schema defined in the jsonschema.metaschema.draft-04-strict.json document referenced earlier.

You can regard these as the authoritative definition and documentation for our TOSCA file and data formats.

The OASIS TOSCA standard is still a work in progress so we cannot claim full compliance. As a result, we refer to the model as "TOSCA Lite". Once the standard has settled we plan to release compliant schemas and artifacts.

types/tosca/

Here are all the files that BPO's Market Type Catalog is built and maintained from.

Any and all files with the .tosca extension in this directory and in all of its sub-directories are regarded as [Definition Modules](#).

You can create arbitrary sub-directories, or even rename the existing ones. The actual names of the files and sub-directories are ignored except for processing purposes. At the same time, we include the filenames in error messages for debugging purposes.

You can freely (re-)arrange your TOSCA files in arbitrary sub-directories within this directory. BPO's Market Type Catalog only cares about the content of the files, not their name or location in the directory structure.

In addition to the TOSCA files, you should store here any external TOSCA "plan" files (currently * .py Python scripts) referred from your Service Templates. The reference in the Service Template is a relative path from the root of the Model Definitions area. E.g, types/tosca/myservice/activate.py.

The .tosca files shipped in BPO are primarily examples or legacy definitions. Normally RAs and onboarding applications onboard the types, templates, plans, and schemas that are needed for their specific purposes.

types/ddui/views

The *ddui/views* (Data-driven User Interface views) directory contains UI Schema 2 (UIS2) files that describe how resource types should be rendered in the UI for specific views and actions.

The UIS2 files for a resource type are stored in the types/ddui/views/package.ResourceTypes.typeName subdirectory of the specified location, where package.ResourceTypes.typeName is the fully-qualified resource type ID for the type schema to describe.

See the **ui-schema-2-docs** on git.blueplanet.com for details on the syntax for the view description files.

domain-types/ddui/views

The UIS2 files describing how a domain type should be rendered in the Orchestrate UI. A domain type view description file should be stored in the `domain-types/ddui/views/<domain_type>` subdirectory of the specified location, where the `<domain_type>` is the domain type ID configured for the RA, with all colons (:) replaced with dots (.) (e.g., `urn:cyaninc:bp:domain:SAM5620` maps to `domain-types/ddui/views/urn.cyaninc.bp.domain.SAM5620`).

ui-schema / **(Deprecated)**

The original UI Schema files and icons used through BPO 16.06 and deprecated starting in 16.10.

This is a flat directory and only files with the `<resource-type-path>.json` format are considered by BPO, where the `<resource-type-path>` must match the fully qualified name (path) of one of the defined Resource Types.

For instance the file `tosca.resourceTypes.EthernetPort.json` is used as the UI schema file for the Resource Type `tosca.resourceTypes.EthernetPort`.

Resource Types

A Resource Type defines characteristics of Resource Instances that are specific to the Type. Resource Types are defined inside [Definition Modules](#).

Resource Types are conceptually similar to a C or C++ header file declaring a struct or a class, or interfaces in Java.

Important characteristics of Resource Instances found in the Resource Type definition include:

- The schema of the Resource Instance's `properties` attribute.
- The capabilities provided by the Resource Instance.
- The requirements of the Resource Instance.

Structure

Resource Types are represented as objects within Definition Modules. The schema of Resource Types can be found in the `json-schemas/tosca-lite-v1/resource-type.hocon` file the Model Definitions area.

The following table describes the attributes of Resource Types.

ATTRIBUTE	TYPE	REQUIRED	DEFAULT	PURPOSE
<code>title</code>	<code>string</code>	<code>yes</code>	<code>N/A</code>	Used for display purposes in UIs.
<code>description</code>	<code>string</code>	<code>yes</code>	<code>N/A</code>	Used for help text and other places where detailed information is displayed on UIs.
<code>abstract</code>	<code>boolean</code>	<code>no</code>	<code>false</code>	If <code>abstract = true</code> , the Resource Type cannot be instantiated directly. Only derived Resource Types may be instantiated. Conceptually this is similar to abstract classes/interfaces in many programming languages.
<code>final</code>	<code>boolean</code>	<code>no</code>	<code>false</code>	If <code>final = true</code> , the Resource Type cannot be derived from. Conceptually this is similar to the <code>final</code> keyword in many programming languages.
<code>derivedFrom</code>	<code>string or [string]</code>	<code>no</code>	<code>[]</code>	Specifies one or more Resource Types which this Resource Type inherits from. See Inheritance .
<code>properties</code>	<code>object</code>	<code>no</code>	<code>{ }</code>	Specifies the schema of the <code>properties</code> attribute of instances of this Resource Type. See Property Definitions .
<code>capabilities</code>	<code>object</code>	<code>no</code>	<code>{ }</code>	Specifies the capabilities provided by this Resource Type. See Capabilities and Requirements .

ATTRIBUTE	TYPE	REQUIRED	DEFAULT	PURPOSE
requirements	object	no	{ }	Specifies the requirements used by this Resource Type. See Capabilities and Requirements .
interfaces	object	no	{ }	Specifies the custom operation interfaces and the input/output schema for each. See Interface Definitions .

Inheritance

Resource Type definitions include three attributes which control the how Resource Type inheritance works: `abstract`, `final`, and `derivedFrom`. The first two are described above.

When one Resource Type derives from another, the derived Resource Type inherits the properties, capabilities, requirements, and interfaces of the parent(s). If the derived Resource Type re-defines any of the properties, capabilities, requirements, or interfaces of the parent, the parent's version is dropped in favor of the derived version.

Resource Types are not required to derive from a base Resource Type however by doing so instances of those types will not be automatically composable in [Service Templates](#). This functionality is provided by the `tosca.resourceTypes.Root` Resource Type. The definition of this type can be found in the `resource_type_basics.tosca` Definition Module inside the `types/tosca/` sub-directory of the Model Definitions area. We recommend you derive all Resource Types from `tosca.resourceTypes.Root` (directly or indirectly).

You should consider inheritance carefully when defining Resource Types.

It is better to derive from a domain specific abstract Resource Type than `tosca.resourceTypes.Root` even if the base does not include any properties. For example, if you are defining a new Virtual Network Function type, it should derive from `tosca.resourceTypes.VirtualNetworkFunction`. If there is no existing domain specific abstract Resource Type to derive from, you can define your own.

You should not rely on inheritance specifically for property definition re-use. Instead, rely on composition and define a [Property Type](#) to avoid the overhead of re-defining the property in each use case.

Property Definitions

Property Definitions specify the schema and access modifiers of Resource Instance "properties". The schema support is based on JSON-Schema so a subset of the standard JSON-Schema constraints are available.

The access modifiers allow authors to specify how the property can be used within BPO (e.g, configurable or not, obfuscated or not, etc.)

Table 1. Property Definition attributes

ATTRIBUTE	TYPE	REQUIRED	DEFAULT	PURPOSE
type	string	yes *	N/A	See JSON-Schema specification. This attribute is required unless a value is given for <code>propertyType</code> .
propertyType	string	no	N/A	See Property Types .
title	string	no	None	Used for display purposes in UIs. Limited to 32 characters.
description	string	no	None	Used for help text and other places where detailed information is displayed on UIs.
format	string	no	None	This can augment the <code>type</code> with additional format constraints. See Property Formats .
output	boolean	no	false	A value of <code>true</code> indicates that this property will be the output of the resource activation. The provider of the Resource shall guarantee that the property is set when the <code>orchState</code> is set to <code>active</code> . The property value will only be writable once, subsequent attempts to update the value will be rejected. **
config	boolean	no	true	A value of <code>true</code> indicates that this property is a configurable property of the resource. The pull request will be rejected if <code>output = true</code> . **
updatable	boolean	no	false	If this flag is set to <code>true</code> on a configurable property, it indicates that the property can be updated freely. If set to <code>false</code> , the property cannot be modified. The pull request will be rejected if <code>output == true</code> is set and is ignored if <code>config = false</code> is set. **
optional	boolean	no	false	If this flag is set to <code>true</code> on a configurable property, it indicates that providing a value for this property is optional. The pull request will be rejected if <code>output == true</code> is set and <code>config = false</code> is set. **
obfuscate	boolean	no	false	If this flag is set to <code>true</code> it will be stored in an encrypted form while "at rest". Useful for secret keys, passwords, etc. This flag cannot be mixed with <code>fulltext = true</code> . Attempts to do so will be rejected during onboarding. Future releases may obfuscate API responses and Log entries.
store	boolean	no	true	For a non-config (read-only) property this flag indicates if the value of the property will be stored in the database. If set to <code>false</code> , the values will not be stored in the database, and the simple GET/LIST operations will not show the property. Instead, a "deep" GET request must be issued, which will collect the current value from the backing Resource Provider. This is useful and should be used for volatile (frequently changing) properties, or properties that are resource-intensive to calculate. For any configurable property this flag is ignored.

ATTRIBUTE	TYPE	REQUIRED	DEFAULT	PURPOSE
history	boolean	no	true	If this flag is set to <code>false</code> , it indicates that historical values for the property do not need to be preserved. This flag does not disable historical values completely. Some historical values may still be available for limited period of time, including any values written while <code>history = true</code> was set. If historical values are not of interest (e.g, because the value changes too frequently) then this flag should be set to <code>false</code> . This flag is not applicable if <code>store</code> is set to <code>false</code> .
fulltext	boolean	no	false	If this flag is set to <code>true</code> , it indicates that an eventually consistent fulltext search index should be generated for the attribute. It defaults to <code>false</code> . The flag can only be set for string attributes and cannot be mixed with the <code>obfuscate</code> property.
units	string	no	None	An optional string designator for a numeric type defining its units (e.g., kg, mm, MB).
minimum	number	no	None	An optional minimum value for integers or numbers.
maximum	number	no	None	An optional maximum value for integers or numbers.
items	object	no	None	See JSON-Schema specification. This attribute is required if <code>type</code> is <code>array</code>
properties	object	no	None	See JSON-Schema specification.

(*) Encryption only applies to resource properties, relationship properties, and domain properties. This support was added in the 16.10 release.

(**) See [Resource Update Rules](#) for further information about how these fields affect update API usage. These changes were added in 17.02 release.

Property Formats

By default, properties only have a simple type such as `string`, `integer`, etc. In some cases, this is not enough to describe the semantics of the property to tooling and validation. These types are supplemented by various "format" specifiers which impose additional validation constraints.

The table below describes the suite of formats available in Property Definitions in addition to those described in the [JSON Schema specification](#).

FORMAT	APPLICABLE TYPE	PURPOSE
time	string	Represent time as "HH:mm:ss" (RFC 3339).
date	string	Represent date as "yyyy-MM-dd" (RFC 3339).

FORMAT	APPLICABLE TYPE	PURPOSE
uint64	integer	64-bit unsigned integer.
int64	integer	64-bit signed integer.
uint32	integer	32-bit unsigned integer.
int32	integer	32-bit signed integer.
uint16	integer	16-bit unsigned integer.
int16	integer	16-bit signed integer.
uint8	integer	8-bit unsigned integer.
int8	integer	8-bit signed integer.
hex-string	string	A string of hex digits separated by ':' characters or more formally, a string matching the following pattern: ^([0-9a-fA-F][0-9a-fA-F](:[0-9a-fA-F][0-9a-fA-F])*?)\$.
vlan-id	integer	A VLAN ID, i.e, an integer in the range 1 and 4094 (inclusive).
port-number	integer	A TCP/UDP port number, i.e, an integer between 1 and 65535 (inclusive).
bgp-as	integer	A BGP AS number, i.e, an integer between 1 and 4294967294 (inclusive) except 65535.
ipv4-address	string	IPv4 address in dotted-quad notation. Equivalent to JSON Schema's ipv4.
ipv4-prefix	string	IPv4 network address and netmask separated by '/' character, e.g, "192.168.0.0/24".
ipv4-interface	string	IPv4 address with netmask separated by '/' character, e.g, "192.168.1.1/24".
netmask	string	IPv4 netmask as a dotted quad, e.g, "255.255.255.0"
ipv6-address	string	Any IPv6 address, per RFC 2373 §2.2 . Equivalent to JSON Schema's ipv6.
ipv6-prefix	string	IPv6 network address and netmask separated by '/' character, per RFC 2373 §2.3 .
ipv6-interface	string	IPv6 address and netmask separated by '/' character, per RFC 2373 §2.3 .
ipv6-multicast	string	IPv6 multicast address (starts with FF).

Property Types

Property Types enable property definition re-use. The attributes of a Property Type are identical to those of [Property Definitions](#). When Property Types are referenced, attribute values from the referenced Property Type can be overridden by attribute values at the reference site. This is shown below with the `title` attribute.

To use a Property Type in another artifact, reference the Property Type using the `propertyType` attribute.

Example Property Type with Usage

```
"$schema" = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-module#"
title      = Property Type Example
package    = example
version    = "1.0"
description = This document shows how property definitions can be re-used.
authors     = [ "John Smith (jsmith@acmetel.com)" ]

propertyTypes {
    Port {
        title = "Device Port"
        description = "Configuration and operational state of a port."
        type = object
        properties {
            opState {
                title = "Operational State"
                type = string
                enum = [ up, down ]
            }
            macAddress {
                title = "MAC Address"
                type = string
                format = hex-string
            }
        }
    }
}

resourceTypes {
    Blade {
        title = "Blade"
        description = "A single blade in the device."
        properties {
            trunk {
                title = "Trunk Port"
                propertyType = "example.propertyTypes.Port"
            }

            trib {
                title = "Trib Port"
                propertyType = "example.propertyType.Port"
            }
        }
    }
}
```

Configurable, Non-Configurable, and Output Properties

Configurable Property

A **Configurable Property** (`config=true`) is a property that is specified as an input when the resource is created. The property may be required or optional (`optional = true`) and required properties may have default values so that they do not need to be explicitly specified. A configurable property of a managed resource (i.e., `discovered=false`) has both a **desired** state and an **observed** state. The **desired** state is specified as the intent given to the orchestrator. The **observed** state is reported by the resource provider as the actual state of the resource is the managed system. Discrepancies are represented as [Differences](#). See [Configurable Properties](#).

Non-Configurable Property

A **Non-Configurable Property** (`config=false`) is a property that is not specified as an input when the resource is created. A non-config property is expected to be set by the provider of the resource only. It doesn't have a **desired** state. The value is considered read-only from the resource API perspective. The value should be set by using the resource observed state API:

```
PUT/PATCH /resources/{resourceId}/observed
```

rather than modifying using the resource API:

```
PUT/PATCH /resources/{resourceId}
```

A non-configurable property may also be set via the `output` section of a service template.

Output Property

An **Output Property** (`output=true`) is a non-configurable property with the additional constraint that it is intended to be set only once on activation. It is required that the output value be set before the resource can become active.

Capabilities and Requirements

The Capabilities and Requirements defined on a Resource Type refer to the features provided and used by instances of that Resource Type (respectively). By modeling the features provided and used by instances of the Resource Type, BPO is able to validate Relationship creation and update requests to ensure that Relationship Instance, source Resource Instance, and target Resource Instance are compatible. This triplet is referred to as the "(relationship, source, target)" below.

Resource Types may include multiple Capability or Requirement Definitions. The schema of the `capabilities` and `requirements` attributes of a Resource Type definition can be found in `capability.hocon` and `requirement.hocon` (respectively) in the `json-schemas/tosca-lite-v1/` sub-directory of the Model Definitions area.

The table below describes the attributes of a Capability Definition:

ATTRIBUTE	TYPE	REQUIR ED	DEFAUL T	PURPOSE
type	string	yes	N/A	Specifies type of the Capability. See Capability Types and Type Compatibility for details.
title	string	no	None	Used for display purposes in UIs.
description	string	no	None	Used for help text and other places where detailed information is displayed on UIs.
resourceTypes	[string]	no	None	See Type Compatibility .
minimum	integer	no	0	See Cardinality .
maximum	integer	no	None	See Cardinality .

The table below describes the attributes of a Requirement Definition:

ATTRIBUTE	TYPE	REQUIR ED	DEFAUL T	PURPOSE
type	string	yes	N/A	Specifies type of the Requirement. See Capability Types and Type Compatibility for details.
title	string	no	None	Used for display purposes in UIs.
description	string	no	None	Used for help text and other places where detailed information is displayed on UIs.
resourceTypes	[string]	no	None	See Type Compatibility .
minimum	integer	no	1	See Cardinality .
maximum	integer	no	1	See Cardinality .

Type Compatibility

Capabilities and Requirements are used to perform validation on Relationship creation and update requests. Part of this validation involves checking for type compatibility between (relationship, source, target). The Capability and Requirement Definitions drive this part of the validation by constraining the valid:

1. Relationship Types via the `type` attribute.
2. Resource Types of instances on either side of the Relationship Instance via the `resourceTypes` attribute.

For Capability Definitions, the `resourceTypes` attribute specifies the valid set of Resource Types of the **source** Resource Instance.

For Requirement Definitions, the `resourceTypes` attribute specifies the valid set of Resource Types of the **target** Resource Instance.

Cardinality

The "cardinality" of a Capability or Requirement allows the author to control the number of Relationship Instances which may (or must) be associated with the Resource Instances on either side of the Relationship.

For Capability Definitions, the cardinality specifies the minimum and maximum number of times the feature may be provided. This is equivalent to the number of incoming Relationship Instances a Resource Instance may have for the associated Capability. By default this is unconstrained, i.e, the range is [0, undefined].

For Requirement Definitions, the cardinality specifies the minimum and maximum number of times the feature must be provided. This is equivalent to the number of outgoing Relationship Instances a Resource Instance may have for the associated Requirement. By default, requirements must be satisfied exactly once, i.e, the range is [1, 1].

Interface Definitions

The Interface Definitions specifies the "interfaces" for this Resource Type. When implemented by a Service Template, users can create "Operations" on a resource against a specific interface, which will cause the corresponding implementation to be run against the resource.

The table below describes the attributes of an interface definition

ATTRIBUTE	TYPE	REQUIR ED	DEFUALT	PURPOSE
<code>title</code>	string	no	None	Used for display purposes in UIs.
<code>description</code>	string	no	None	Used for help text and other places where detailed information is displayed on UIs.
<code>inputs</code>	object	no	{ }	Defines the schema for inputs of this interface using the Property Definitions format.
<code>outputs</code>	object	no	{ }	Defines the schema for outputs of this interface using the Property Definitions format.

See [Custom Operations](#) for more information on Interfaces and Operations.

Access Modifiers for `inputs` and `outputs`

The following access modifiers will be fixed for inputs/outputs definition for a resource interface. Such a modifier will be ignored if specified in the Resource Type definition file.

MODIFIER	FIXED VALUE FOR INPUTS	FIXED VALUE FOR OUTPUTS
config	true	false
output	false	true
updatable	false	false
store	true	true
history	true	true
fulltext	false	false

Example Interface Definitions

The following is an example of defining a **Sort** interface on a Sorter Resource Type

```

resourceTypes {
    Sorter {
        properties {
            ...
        }
        interfaces {
            sort {
                title = Sort an array of integers
                description = "Perform sorting of an integer array using the resource"
                inputs {
                    arrayToSort {
                        title = "Array To Sort"
                        description = "Array to be sorted"
                        type = array
                        items.type = integer
                    }
                    reverse {
                        title = "Reverse?"
                        description = "Whether to reverse the result"
                        type = boolean
                        default = false
                    }
                }
                outputs {
                    sortedArray {
                        title = "Sorted Array"
                        description = "Sorted array"
                        type = array
                        items.type = integer
                    }
                }
            }
        }
    }
}

```

Service Templates

A Service Template defines an implementation of [Resource Types](#). The implementation consists of **plans** defining the behavior for lifecycle events (e.g., activation, termination, update).

Drawing on the OO (Object-Oriented) analogy from the [Resource Types](#) reference, service templates are conceptually similar to the implementation of a class in C++ or Java.

The implementation describes the structure of a service offered through BPO. The structure specifies the Resources and Relationships of which the service is composed. Because of this composition, we typically refer to Resources implemented by a Service Template as **composite resources**. Furthermore, we refer to the Resources composed in Service Templates as **sub-resources**.

The implementation may be defined **declaratively** using a Domain Specific Language consisting of [Directives](#). Alternatively, the implementation may be defined **imperatively** by providing executables to handle the activate, terminate, and update events against associated Resources. The executables are included through [Plans](#) objects in the Service Template. The executable is responsible for realizing the desired structure through API calls to BPO.

In general, Service Template authors should prefer the declarative style when possible and fallback to the imperative style when necessary. This may occur if the DSL is insufficient. We encourage users to provide feedback or suggestions for directives.

Structure

Service Templates are represented as objects within Definition Modules. The schema of Service Templates can be found in the `json-schemas/tosca-lite-v1/service-template.hocon` file in the Model Definitions area.

The following table describes the attributes of Service Templates:

ATTRIBUTE	TYPE	REQUIRED	DEFAULT	PURPOSE
<code>title</code>	string	yes	N/A	Used for display purposes in UIs.
<code>description</code>	string	yes	N/A	Used for help text and other places where detailed information is displayed on UIs.
<code>implements</code>	string	yes	N/A	Specifies the Resource Type which this Service Template implements.
<code>resources</code>	object	no	{ }	Specifies sub-resources to compose. See Sub-Resources below.
<code>output</code>	object	no	{ }	Specifies how <code>output</code> properties should be populated. See Output below.
<code>trigger</code>	object	no	{ }	Specifies how <code>trigger</code> properties should be populated. See Trigger below.

ATTRIBUTE	TYPE	REQUIRED	DEFAULT	PURPOSE
plans	object	no	{ }	Includes scripts to run as part of activation, termination, and update of the composite resource. If this attribute is specified, the resources attribute is ignored. See Plans below.

Sub-Resources

Each value in the resources object specifies a sub-resource. The schema of the individual sub-resource objects can be found in the json-schemas/tosca-lite-v1/templated-resource.hocon file in the Model Definitions area.

The following table describes the attributes of sub-resources in the Service Template:

ATTRIBUTE	TYPE	REQUIRED	DEFAULT	PURPOSE
title	string	no	N/A	Used for display purposes in UIs.
description	string	no	N/A	Used for help text and other places where detailed information is displayed on UIs.
type	string	yes	N/A	Specifies the Resource Type of the sub-resource.
product	object or string	no	N/A	Specifies the exact Product to create the sub-resource from. See Product Locator below.
forEach	array	no	N/A	Used to declaratively create a variable number of sub-resources. Refer to forEach for more details.
createIf	object or boolean	no	N/A	Specifies that sub-resource should be created conditionally. Refer to createIf for more details.
autoClean	boolean	false	N/A	Specifies the value of the autoClean attribute on the created resource. See Instance Catalog for details.
activateAfter	string or strings	false	N/A	Specifies explicit ordering of activation for this sub-resource. By default, activation is ordered based on dependencies between sub-resources.
terminateAfter	string or strings	false	N/A	Specifies explicit ordering of termination for this sub-resource. By default, termination is ordered based on dependencies between sub-resources.
properties	object	no	{ }	Specifies how properties of the sub-resource should be populated. See Directives for details.

Product Locator

By default, BPO will automatically select the Product from which to create the sub-resource. If there is only one product in the catalog that provides the specified resource type, the choice is trivial and unambiguous. If there are multiple domains providing the a product for the specified resource type, a product in the same domain is preferred. If one doesn't exist in the same domain, a product is preferred in a domain with the same tenant.

In case there are multiple Products to choose from a `product` attribute should be specified on the sub-resource to provide information for the Product Locator to use when narrowing the set of candidates and automatically selecting the Product.

If the Product Locator is a `string`, the value is assumed to be the Product identifier. This should not be used if the Service Template will be re-used in multiple instances of BPO (since the Product identifier is unique per domain.)

```
product = 56fc3afb-fff0-4e38-b084-c05287946411
```

If the Product Locator is an `object` with a `domain` field, the domain is used to narrow the Product Locator search to available products for the resource type within the specified domain.

```
product {domain = {getDomain {getParam = privateNet}}}
```

Otherwise, if the Product Locator is an `object`, the value is interpreted as a directive. The only directive which currently can be used in the Product Locator is the `getParam` directive.

```
product = {getParam = flavorVmProduct}
```

If multiple products satisfy the criteria passed to the Product Locator, the first matching product is selected.

Resource Locator

The template engine has access to a powerful means of locating resources inside the orchestrator. This functionality is exposed to service templates via the `getResourceWith` and `getResourcesWith` directives and is operationally similar to the standard `GET /resources` API. The query parameters for the resource locator is a map of key-value pairs which must exist on a Resource.

Example:

```
value = { getResourceWith = {
    resourceId = "example.resourceTypes.ExampleType"
    orchState = "active"
    label = "example"
    "$.properties.myId" = "my-unique-identifier"
}}
```

The following example would try to locate a resource which is of the type "example.resourceTypes.ExampleType", has an `orchState` set to "active", has a Resource Attribute label set to "example", and has a properties field called "myId" set to "my-unique-identifier".

Usage Caveats

The Resource Locator supports searching with any number of normally filterable Resource Attributes, as well as top level and N level properties. It supports a mix of in-database and in-application filtering however it is recommended for performance reasons to only filter on top level primitive attributes and properties where possible.

Supported General Attributes

```
productId: String  
resourceTypeId: String  
exactResourceTypeId: String  
domainId: String  
tenantId: String
```

WARNING! For best performance, it is highly recommended that you **always** specify one (and only one) of `productId`, `resourceTypeId`, or `exactResourceTypeId`. This is required for database level filtering.

`domainId` will restrict the search to a given domain, while `tenantId` will restrict the search to a given subtenant. Only one of `productId`, `resourceTypeld` or `exactResourceTypeld` may be specified.

Supported Resource Attributes

The Resource Locator can filter resources based on the following attributes:

```
label: String  
description: String  
orderId: String  
shared: Boolean  
providerResourceId: String // Note: Requires a domainId to be specified  
discovered: Boolean  
desiredOrchState: String  
orchState: String  
reason: String  
autoClean: Boolean
```

Resource Properties

Top level primitive properties are supported for in-database filtering when one of `productId`, `resourceTypeld` or `exactResourceTypeld` is specified. More complex properties will require application level filtering and should be used sparingly. Properties filtering parameters are expected to be in the form of a JSONPath key followed by a corresponding value. Comparisons between complex types (i.e. JSON Array == JSON Array) are technically supported, yet not recommended. For best results, use paths which point to a primitive type inside the JSON Array or JSON Object, rather than the Object itself. Example: Given a JSON Object such as:

```
"properties": {
    "myObject": {
        "myString": "hello",
        "myBoolean": true
    }
}
```

One could construct a locator argument in multiple ways:

```
"$.properties.myObject" = { "myString": "hello", "myBoolean": true }
```

OR

```
"$.properties.myObject.myString" = "hello"
"$.properties.myObject.myBoolean" = true
```

The second form is highly recommended as the first relies on JSON implementation details to perform the comparison.

Special Arguments

For relationship usecases, custom filtering functions are also available in order to better locate candidate resources for use as the source or target in a relationship. All provided values must be strings or directives which resolve to strings.

targetFromRelationshipType

```
targetFromRelationshipType = <Desired Relationship Type URI>
sourceResourceId = <Resource Type URI of the Relationship Source>
requirementName = <Source Resource Type's Requirement name to use>
```

When `targetFromRelationshipType` is provided, the other two fields must also exist. This will locate a resource which provides a capability that is compatible with the given relationship type, source resource type, and source requirement name.

sourceFromRelationshipType

```
sourceFromRelationshipType = <Desired Relationship Type URI>
targetResourceId = <Resource Type URI of the Relationship Target>
capabilityName = <Target Resource Type's Capability name to use>
```

When `sourceFromRelationshipType` is provided, the other two fields must also exist. This will locate a resource which has a requirement that is compatible with the given relationship type, target resource

type, and target capability name.

Plans

The `plans` object in a Service Template definition includes scripts to be executed by BPO when certain lifecycle or custom operations occur.

The keys of the `plans` object specify the names of the plans. The schema of a plan object can be found in the `json-schemas/tosca-lite-v1/plan.hocon` file in the Model Definitions area, as illustrated in the following table:

ATTRIBUTE	TYPE	REQUIRED	DEFAULT	PURPOSE
<code>type</code>	string	yes	N/A	Specifies the kind of plan. Currently only "script" and "remote" are supported.
<code>language</code>	string	maybe	N/A	Specifies the language of the script. Currently only "python" is supported. Required if <code>type</code> is "script".
<code>path</code>	string	yes	N/A	Specifies the location of the script in the Model Definitions area. The path should be relative to the root of the area.
<code>autoClearDifferences</code>	boolean	no	true	Indicates if 'differences' should be cleared automatically when the plan has been successfully executed. This flag only applies to 'update' plans. See Differences for more information on differences.

For "script" plans, the script file is typically located in the same directory as the Service Template definition. See [Imperative Plans](#) for information on writing imperative plans as Python scripts.

Lifecycle Plans

The lifecycle plans defines the **imperative** style lifecycle operations, which consist of the following plans:

- `activate` - called when the Resource is activated.
- `update` - called when the Resource properties are updated.
- `terminate` - called when the Resource is terminated.

For more information on these operations see [Lifecycle Operations](#) for details.

NOTE

See [Mixing Imperative and Declarative Lifecycle Operations](#) for details about using both imperative and declarative plans in the same template.

Custom Plans

In addition to the lifecycle plans, custom plans can be defined in the `plans` object. The names of such

plans should correspond to the names defined in the `interfaces` section of the Resource Type definition. A custom plan will be called when a Resource Operation is created against the interface for a specific Resource Instance.

NOTE As opposed to the lifecycle plans, custom plans can co-exist with the `resources` section in the Service Template. This enables executing of resource operations on a declarative Service Template.

See [Custom Operations](#) for how to use custom plans, and [Imperative Plans](#) for how to implement them.

Examples

The following example defines the lifecycle plans and a custom plan called "provision":

```
serviceTemplates {
    IpVpnService {
        title = "IP VPN"
        description = "Implementation template for the service"
        implements = tosca.resourceTypes.IpVpnService
        plans {
            activate {
                type = script
                language = python
                path = "types/tosca/vpn/ip_vpn_activate.py"
            }
            terminate {
                type = script
                language = python
                path = "types/tosca/vpn/ip_vpn_terminate.py"
            }
            update {
                type = script
                language = python
                path = "types/tosca/vpn/ip_vpn_update.py"
            }
            provision {
                type = script
                language = python
                path = "types/tosca/vpn/ip_vpn_provision.py"
            }
        }
    }
}
```

The following example defines a custom plan called "provision" in a declarative Service Template:

```
serviceTemplates {
    IpVpnService {
        title = "IP VPN"
        description = "Implementation template for the service"
        implements = tosca.resourceTypes.IpVpnService
        resources {
            vrf {
                type = tosca.ResourceTypes.VirtualRouter
            }
        }
        plans {
            provision {
                type = script
                language = python
                path = "types/tosca/vpn/ip_vpn_provision.py"
            }
        }
    }
}
```

Validators

The `validators` object in a Service Template definition includes scripts to be executed by BPO for validating lifecycle operations.

The keys of the `validators` object specify the names of each plan. Only the following plans may be defined:

- `activate` – called when validating resource creation.
- `update` – called when validating resource update.
- `terminate` – called when validating resource termination.

The schema of each plan object is the same as that in the `plans` section.

For more information on resource validations, see [Validation Operations](#).

Example

The following example shows a Service Template with `activate` and `update` validators defined:

```
serviceTemplates {
    IpVpnService {
        title = "IP VPN"
        description = "Implementation template for the service"
        implements = tosca.resourceTypes.IpVpnService
        resources {
            vrf {
                type = tosca.ResourceTypes.VirtualRouter
            }
        }
        validators {
            activate {
                type = script
                language = python
                path = "types/tosca/vpn/vpn_activate_validate.py"
            }
            update {
                type = script
                language = python
                path = "types/tosca/vpn/vpn_update_validate.py"
            }
        }
    }
}
```

Output

The `output` object in a Service Template definition specifies how to populate the [Output Properties](#) of the Resource. Each value of the `output` object specifies how the property identified by the key should be populated. If the value contains any [Directives](#), they will be evaluated.

An `output` property is a `config=false` property that is expected to have its value set exactly once at activation time.

For example, consider the **ExampleService** resource type below that has an `output` property named **serialNumber**. The declarative service template defines a sub-resource named **RandStr1**. The `output` section of the service template declares that the value from the sub-resource is used to populate the **serialNumber** on the host resource. After the sub-resource has been activated, the template engine processes the directive in the `output` section to set the **serialNumber** `output` property on the host resource.

```

resourceTypes {
    ExampleService {
        title = "Example Service"
        description = "An example service"
        derivedFrom = "tosca.resourceTypes.Root"
        properties {
            serialNumber {
                title = "Serial Number"
                type = string
                output = true
            }
        }
    }
}

serviceTemplates {
    ExampleService {
        title = Example Service
        description = "An example service"
        implements = example.resourceTypes.ExampleService
        resources {
            RandStr1 {
                type = RandomString
            }
        }

        output {
            serialNumber = {getAttr = [RandStr1, value]}
        }
    }
}

```

Trigger

The `trigger` object in a Service Template definition specifies how to populate observed property values of the Resource based on observed updates to sub-resources. Each value of the `trigger` object specifies how the property identified by the key should be populated. If the value contains any [Directives](#), they will be evaluated.

A trigger fires at the end of activation after all sub-resources have become active and prior to the resource becoming active. The trigger also fires every time a child resource referenced in the trigger has a change to an observed value.

The difference between the `trigger` object and the `output` object is that the `output` is executed only once at activation (for set-once output properties) while the `trigger` is executed any time the observed state of a relevant sub-resource is updated.

For example, consider the **ExampleService** resource type below that has a `trigger` property named **serialNumber**. The declarative service template defines two sub-resources named **RandStr1** and **RandStr2**. The `trigger` section of the service template declares that the **joined** value from the sub-resources are used to populate the **serialNumber** on the host resource. After the sub-resources have been activated, the template engine processes the directive in the `trigger` section to set the **serialNumber**

trigger property on the host resource.

```

resourceTypes {
    ExampleService {
        title = "Example Service"
        description = "An example service"
        derivedFrom = "tosca.resourceTypes.Root"
        properties {
            serialNumber {
                title = "Serial Number"
                type = string
                config = false
            }
        }
    }
}

serviceTemplates {
    ExampleService {
        title = Example Service
        description = "An example service"
        implements = example.resourceTypes.ExampleService
        resources {
            RandStr1 {
                type = RandomString
            }
            RandStr2 {
                type = RandomString
            }
        }
        trigger {
            serialNumber = { join = ["/", {getAttr = [RandStr1, value]}, {getAttr = [RandStr2, value]}] }
        }
    }
}

```

During activation, a `MadeOf` relationship is established from the host resource to child resources. At the end of activation, the **serialNumber** value is set based on the initial values for **RandStr1** and **RandStr2**. If either (or both) or the sub-resources are observed updated after activation, then the parent trigger section will be reevaluated and the **serialNumber** value set based on the updated value(s) of the sub-resources.

Requirements

The `requirements` object in a Service Template `resources` section describes what **Relationship Instances** a resource must have before becoming **active**. The template engine will then use this information to dynamically generate such relationships. Should any requirement fail to be met, the resource will be left in a **failed** state.

The schema for this section is described below:

ATTRIBUTE	TYPE	REQUIRED	DEFAULT	PURPOSE
title	string	no	None	Used for display purposes in UIs.
description	string	no	None	Used for help text and other places where detailed information is displayed on UIs.
resource	string	yes	None	Specifies a Resource ID to use as the target satisfying this requirement.
relationship	string	yes	None	Defines a Relationship Type FQN to use when creating a Relationship satisfying this requirement.
capability	string	no	None	Specifies an optional Capability name to use which must be present on the specified resource. If this is not provided, the orchestrator will attempt to locate one instead.

The following example shows the general usage of the requirements section of a service template. This example would create a **ExampleService** with three resources, **ExampleResourceA**, **ExampleResourceB** and **ExampleResourceC**. **ExampleResourceB** and **ExampleResourceC** will create Relationships to **ExampleResourceA** using its **myCapability** capability. **ExampleResourceB** does not provide an explicit capability name to use, so it is instead discovered, while **ExampleResourceC** specifies **myCapability** explicitly. If the target resource (**ExampleResourceA** in this example) has multiple compatible capabilities, the first one will be chosen. If the target resource does not have any valid capabilities for the relationship, the creation of **ExampleResourceB** and **ExampleResourceC** would then fail.

```
resourceTypes {  
    ExampleResource {  
        title = "Example Resource"  
        description = "An example Resource"  
        derivedFrom = "tosca.resourceTypes.Root"  
        capabilities {  
            myCapability {  
                title = "An example Capability"  
                description = "An example Capability"  
                type = Endpoint  
            }  
        }  
        requirements {  
            myRequirement {  
                title = "An example Requirement"  
                description = "An example Requirement"  
                type = Endpoint  
            }  
        }  
    }  
}  
  
serviceTemplates {  
    ExampleService {  
        title = Example Service  
        implements = "tosca.resourceTypes.Root"  
        resources {  
            ExampleResourceA {  
                type = "example.resourceTypes.ExampleResource"  
            }  
            ExampleResourceB {  
                type = "example.resourceTypes.ExampleResource"  
                requirements {  
                    myRequirement {  
                        resource = { getResourceId = ExampleResourceA }  
                        relationship = "tosca.relationshipTypes.ConnectsTo"  
                    }  
                }  
            }  
            ExampleResourceC {  
                type = "example.resourceTypes.ExampleResource"  
                requirements {  
                    myRequirement {  
                        resource = { getResourceId = ExampleResourceA }  
                        relationship = "tosca.relationshipTypes.ConnectsTo"  
                        capability = "myCapability"  
                    }  
                }  
            }  
        }  
    }  
}
```

NOTE

The requirement name in the service template requirements section must match a corresponding requirement in the resource type definition. If it does not, the template will be rejected during onboarding.

Each field in the requirements section (**resource**, **relationship**, **capability**) will be evaluated by the template engine, so [Directives](#) may be used as long as they resolve to a valid Resource UUID, valid Relationship Type name and valid Capability name on the target resource respectively.

Activation Ordering

Resource Requirements will implicitly determine the ordering of resource creation and termination. The above example would result in **ExampleResourceA** *always* being created first, followed by both **ExampleResourceB** and **ExampleResourceC** in parallel. Furthermore, **ExampleResourceB** and **ExampleResourceC** will not become **active** until the relationship is built. This means that if these resources were backed by their own templates, those templates will not be executed until the requirement was fulfilled. The following diagram visually shows this behavior on a three resource template, where the resource **VNF/2** is creating a relationship with **VNF/3** based on some requirement information:

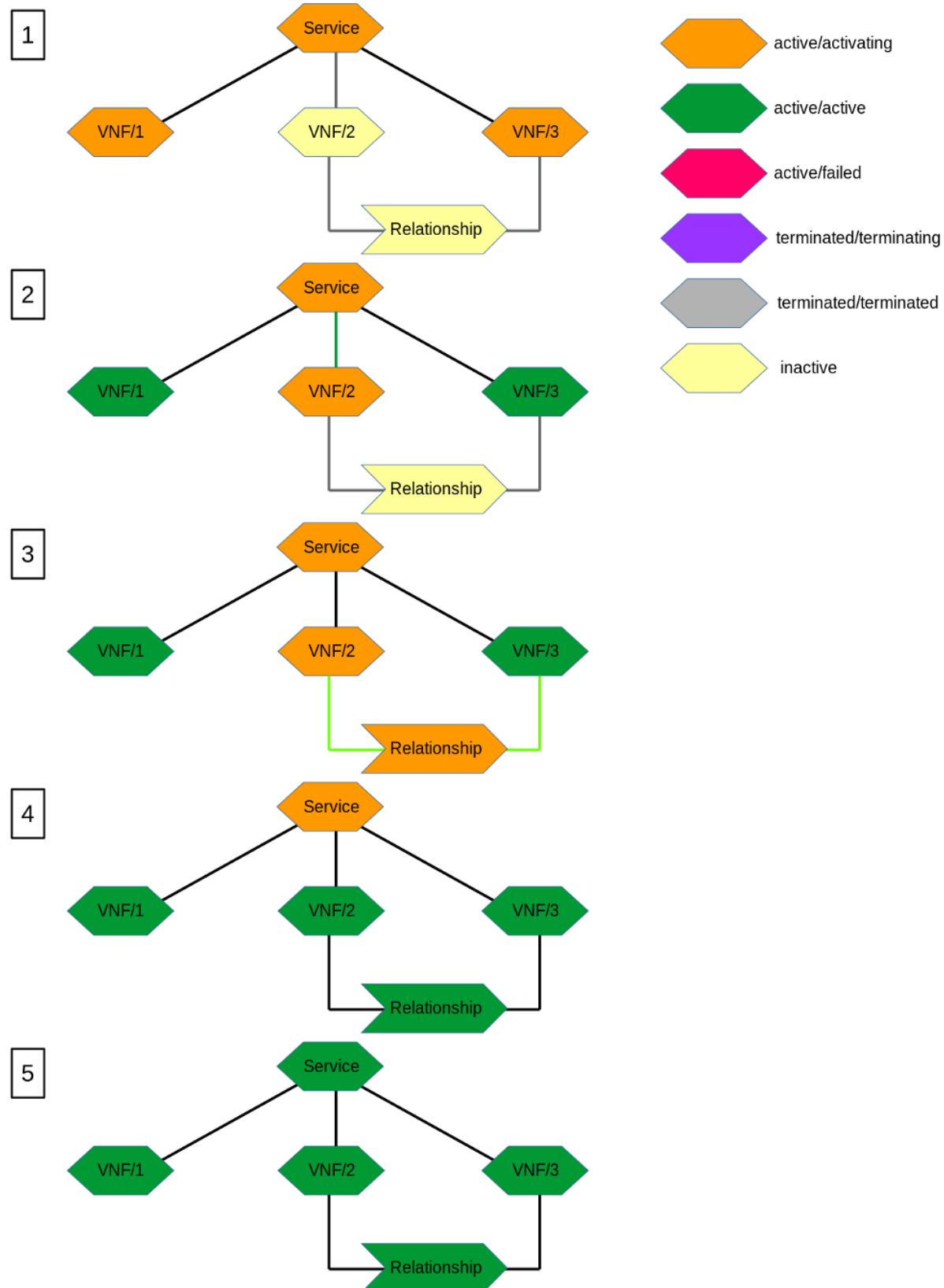


Figure 4. Template Engine Requirements and Activation Ordering

1. The orchestrator resolves dependencies between resources and begins parallel activation of VNF/1 and VNF/3 while deferring VNF/2 due to its relationship.
2. VNF/1 and VNF/3 are both successfully activated. VNF/2 is held in a requested state.
3. The orchestrator resolves the requirement from VNF/2 to VNF/3.
4. The Relationship to VNF/3 is created and only then is VNF/2 moved to an active/activating state which may include running activation plans.
5. The Service is now fully activated.

Activation Failure and Auto Clean Operation

Should a Requirement fail to be satisfied and [Auto Clean](#) is enabled, the service template will be rolled back in a controlled fashion. The following diagram shows this behavior, again on a three resource template where **VNF/2** tries to create a relationship with **VNF/3** but fails due to some error:

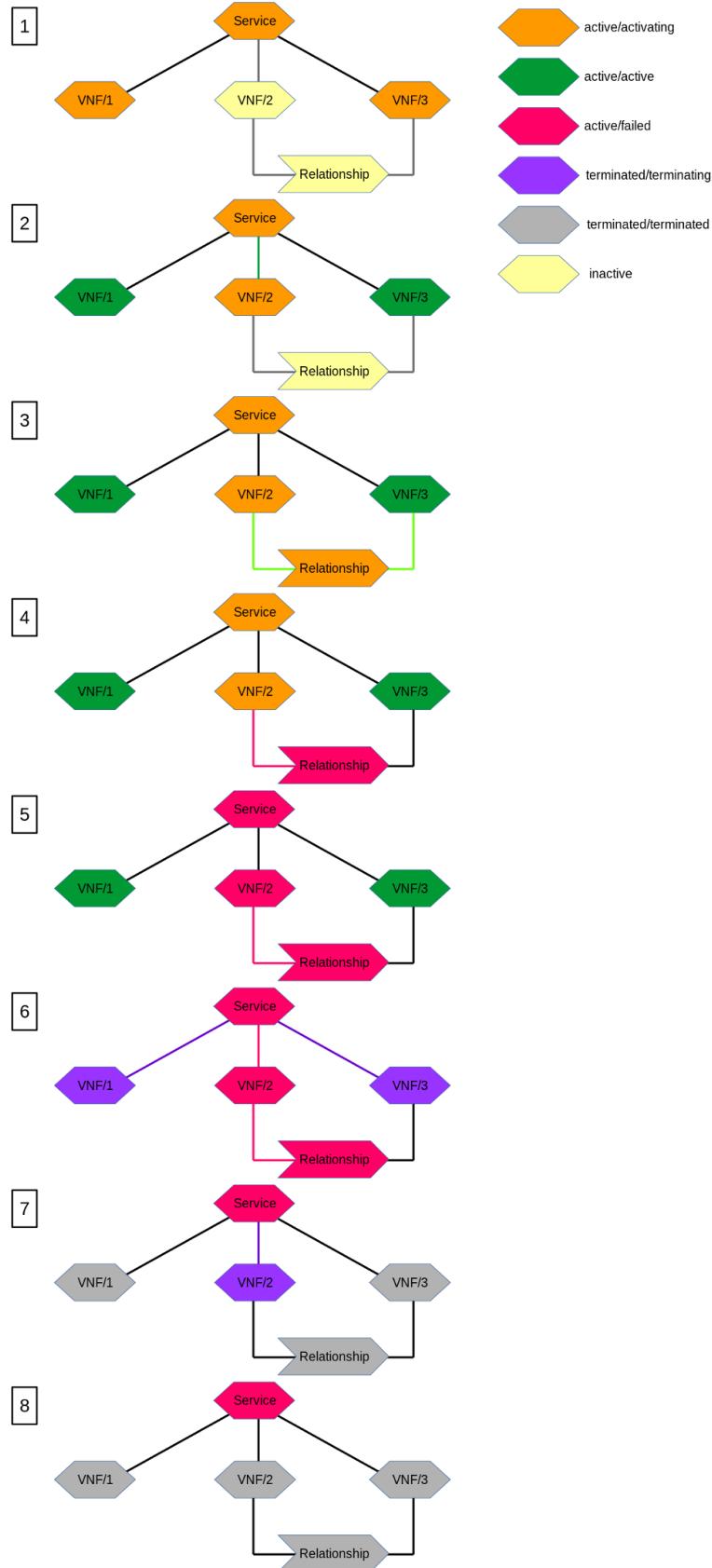


Figure 5. Template Engine Requirements: Activation Failure and AutoClean Operation

1. The orchestrator resolves dependencies between resources and begins parallel activation of VNF/1 and VNF/3 while deferring VNF/3 due to its relationship.
2. VNF/1 and VNF/3 are both successfully activated. VNF/2 is held in a requested state.
3. The orchestrator resolves the requirement from VNF/2 to VNF/3.
4. The Relationship fails to activate for some reason.
5. This failure propagates back to VNF/2 and the Host Resource.
6. The orchestrator begins terminating VNF/1 and VNF/3.
7. The orchestrator begins terminating VNF/2.
8. The Host Resource is then left in a failed state.

Directives

The template engine supports a number of JSON directives available inside the template body to manipulate data and to support more complex logic. These are conceptually function calls that can be placed in places where otherwise a value is expected. Refer to the [Service Template Directives](#) reference for more details.

Service Template Directives

Service templates support expressing limited functional behaviors using language **directives**. Like the rest of the template file, the directives are written in JSON (or HOCON) and are available to be used inside the template body to manipulate data. These are conceptually function calls that can be placed in places where otherwise a value is expected. For example, lets assume you want to assign a value to a resource property:

```
propX = "NODE001"
```

If the value is not a constant but something that needs to be fetched from someplace else, or needs to be derived from other values, a directive (function call) can be used. For example, to fetch the value from one of the input properties of the host resource (e.g., "inputParam1"), you can write:

```
propX = { getParam = inputParam1 }
```

Note: If this was not JSON, we would have something like `propX = getParam("inputParam1")`, but this is JSON (or HOCON), so our syntax must work in JSON's constraints.

Anywhere where a value is expected, a directive can be used instead. Such directives can be nested arbitrarily. Here is a somewhat contrived nested example:

```
propX = { getParam = { join = [ "", "inputParam", { getAttr = [ resource1, output ] } ] } }
```

The directives in this example are explained below.

Patterns that hold for all directives:

- Each directive is a JSON object with one key/value pair, where the key is the name of the directive.
- If the directive has one argument, it is provided as the value of the key/value pair.
- If the directive has two or more arguments, these are listed in an array. This is the conceptual equivalent of positional argument passing in programming languages.

Important note: if a directive used in a resource (say, resource "A") refers to one of the other peer resources (say, resource "B") in the template, this implicitly adds a sequencing constraints between the activation of the two resources involved: the activation of resource "A" will wait till activation of resource "B" is concluded and the orchestration state of "B" becomes *active*.

JSON Path

Properties are generally addressed using a [JSONPath](#) dot-notation syntax to describe a property path. For example, the property `inputParam` may be addressed as either `$.inputParam` or simply `inputParam`.

A more complex query expression can be used to navigate objects and arrays. For example, to obtain the IP subnet of the 4th address on the first interface, the JSONPath may be given as:

```
$.interfaces[0].addresses[3].subnet
```

Note:

- indices always start with 0
- Use quotes (" ") around a JSON path when used within a directive (the \$ is used to indicate a substitution in HOCON and [] is used to define a list)

By definition, a JSONPath query always returns an array of results. This is true even if there is exactly one match. Some directives automatically pop the first entry in the result while others return the entire array. Consult individual directives for specific usage.

Summary Table of Available Template Directives

DIRECTIVE	SYNOPSIS
activateAfter	Specify that the creation of a specific resource follows another.
and	Return boolean result of logical and of two operands.
createIf	Specify that a given resource is created conditionally.
extract	Use a JSONPath query string to extract data from an arbitrary JSON structure.
fetch	Fetch some data from an external data store, such as a Web server or a SQL server.
filterArray	Filter an array based on a boolean condition.
forEach	Directive used in sub-resource definition to create variable number of sub-resources of that type.
getAttr	Fetch a property of one of the other peer resources.
getContext	Fetch information from the execution context.
getCurrent	Fetch the instance value in a looping directive.
getCurrentIndex	Fetch the iterator's index value in looping directives.
getDomain	Fetch the ID of the domain containing a resource.
getDomainWith	Query to find a matching domain.
getIpv4Subnet	Return the subnet property of an IPv4 interface address in 'x.x.x.x/y' format.
getObservedOptionalAttr	Fetch a property of one of the other peer resources checking observed first then fallback or return nothing
getOptionalAttr	Fetch a property of one of the other peer resources with fallback or return nothing.

DIRECTIVE	SYNOPSIS
getOptionalParam	Fetch a property of the host resource if it exists or return nothing.
getParam	Fetch a property of the host resource. If it doesn't exist, raise an exception.
getProviderResourceId	Fetch the provider resource id of a peer resource.
getResourceAttr	Fetch a property of an arbitrary resource stored in the resource catalog.
getOptionalResourceAttr	Fetch a property of an arbitrary resource, returning nothing if the property doesn't exist.
getResourceId	Obtain the resource id of a peer resource or the host resource.
getResourceTenantId	Obtain the tenant ID of a resource.
getResourceWith	Query to find a matching resource id.
getResourcesWith	Query to find an array of matching resource id's.
getSubResourceNames	Query to find names of all sub-resources in the Output section.
ifElse	Select between two values based on a boolean condition.
isEqual	Return boolean result of equality test on two operands.
isGreater	Return boolean result of > test on two operands.
isGreaterEqual	Return boolean result of >= test on two operands.
isLess	Return boolean result of < test on two operands.
isLessEqual	Return boolean result of \neg test on two operands.
isNotEqual	Return boolean result of inequality test on two operands.
join	Concatenate arbitrary number of values into one string, using a "glue" string between the components.
len	Extract the length of an array, an object, or a string.
lookupInStringMap	Look up a string value in a hash-map by its key.
map	Map an array of things into another array of things.
or	Return boolean result of logical <code>or</code> of two operands.
oscall (deprecated)	Fetch some data using OS Shell call.
py2eval	Execute standalone Python scripts.
range	Generate an array of integers in a given range.
renderTemplate	Construct a string using a "moustache-style" text template with variable substitution.
setOnlyIf	Set a value only if the condition is true.
split	Return a positional part from a string split by a field separator.
terminateBefore	Specify that the termination of a specific resource must be done before another.

Glossary of Service Template Directives

activateAfter

Specify that the creation of a specific resource follows another. When you add an activateAfter attribute to a resource, that resource is created after the creation of the resource or resources specified in the activateAfter attribute. You can use the activateAfter attribute with any resource.

Syntax:

```
{ activateAfter = <resource-name> }
```

or if specifying multiple resources

```
{ activateAfter = [ <resource1-name>, <resource2-name>, ..., <resourcek-name> ] }
```

where:

- <resource-name> is a resource named in the same template.

Example:

```
resources {
    subnet {
        ...
    }
    testVm {
        ...
        activateAfter = subnet
    }
}
```

and

Return boolean result of logical and test on two operands.

Syntax:

```
{ and = [ <left-hand-side>, <right-hand-side> ] }
```

where:

- <left-hand-side> is one of the boolean values or expressions to evaluate.
- <right-hand-side> is the other boolean value or expression to evaluate.

Example:

```
goldStatus = { and = [ { isEqual = [ { getParam = "service" }, "gold" ] }, { getParam = "upStatus" } ] }
```

createIf

Specify that the creation of a specific resource is conditional on a given boolean expression. When you add the `createIf` attribute to a resource, that resource is only created if the condition evaluates to `true`. You can use the `createIf` with any resource.

Syntax:

```
{ createIf = <condition> }
```

where:

- the `<condition>` is a boolean expression as described in [ifElse](#)

Example

To create a sub-resource named **subnet** if and only if the **existingSubnet** parameter does not exist.

```
resources {
    subnet {
        ...
        createIf = {ifElse = [{getOptionalParam = existingSubnet}, false, true]}
    }
}
```

extract

Use a [JSONPath](#) query string to extract data from an arbitrary JSON structure. Syntax:

```
{ extract = [ <json-blob>, <json-path>, <pop-array> ] }
```

where:

- `<json-blob>` is a JSON object or array.
- `<json-path>` is the query string compliant with the [JSONPath](#) syntax. By definition, a JSONPath query always returns an array of results. This is true even if there is exactly one match.
- `<pop-array>` is a boolean value (`true` or `false`). If it is set to `false`, the JSONPath result array will be used as the result of the `extract` directive. If it is set to `true`, the first element in the array will be "popped" and used as the result of the `extract` directive.

Example:

```
publicAddress = { extract = [ { getAttr = [ stack, templateOutputArgs ] } ,
    "$.PublicIP.value", true ] }
```

fetch

Fetch some data from an external data store, such as a Web server or a SQL server.

Syntax:

```
{ fetch = [ <url>, <json-path>, <pop-array> ] }
```

where:

- <url> is the URL describing the information that needs to be fetched. Currently the following protocols are supported:
 - MySQL: fetch data from a MySQL database using a select query embedded in the URL (see syntax below). In case of MySQL the result of the select query will be turned into an array of JSON objects, where each record is an item in the array, and the column name / value pairs are turned into key/value pairs in the JSON objects.
 - HTTP: fetch data from a Web server using a simple GET call. It is assumed that the call succeeds and it returns with a JSON object.
- <json-path> defines what data to extract from the result JSON array or object. The usage is the same as for the `extract` directive above.
- <pop-array> controls if the first result should be "popped" from the resulting JSON array. The usage is the same as for the `extract` directive above.

Example:

```
myname = { fetch = [ "mysql://root:root@sqlserver/mydb?SELECT * FROM mytable WHERE id='myid' LIMIT 1", "${[0].name}", true ] }
```

This would perform a query on a MySQL database running on server "sqlserver", using `root` as `username` and `password`, using `mydb` as `database`, and lookup a record in table `mytable` that has its `id` value set to "myid". The `name` column of the database record will be assigned to `myname`.

filterArray

Filter an array based on a boolean condition.

Syntax:

```
{ filterArray: [ <iterator-name>, <boolean condition>, <input-array> ] }
```

where:

- <iterator-name> is the symbol name to refer to the current item in the generator expression.
- <boolean-condition> is evaluated to specify if the instance in the array should be included or not.
- <input-array> is a array which needs to be filtered. It can be a list of integers, strings, objects, etc.

Return: A filtered array based on the condition specified.

Please refer to [this](#), for more details on `{}` and `getCurrent` referencing for the iterator value.

Example:

```
{ filterArray: [ "name", { "isEqual" : [ "${name}", "abc" ] }, [ "abc", "def" ] ] }
```

forEach

Directive used in sub-resource definition to create variable number of sub-resources of that type.

Syntax:

```
forEach = [ <instance-name>, <resource-name>, <input-array>, <create-sequentially>
]
```

where:

- <instance-name> is the symbol name to refer the current item in the generator input-array expression.
- <resource-name> is evaluated as the name of the resource to be created. All evaluated names should be unique.
- <input-array> is an array from which variable resources will be created. It can be a list of integers, strings, objects, etc.
- <create-sequentially> is an optional boolean argument to specify whether forEach resources should be created sequentially. It defaults to true. If the resources are created sequentially, a `terminateBefore` relationship will be added between them, so that they are terminated in reverse order. If `create-sequentially` is set to false, the resources will be created in parallel.

Accessing input array instances in resource definition

Depending on the type of elements in the input array, instances can be referred in resource definition via `getCurrent` directive or moustache-styled `{ }` string substitution. `{ }` referencing is used to get the string value of the instance. For more details on `getCurrent` directive, refer [this](#).

Example 1:

```
resources {
    subRes {
        type = tosca.resourceTypes.Noop
        forEach = ["index", "foo_{index}", [0,1], false]
        properties {
            data = {p1 = {getCurrent = index}}
        }
    }
}
```

The example above will expand into 2 sub-resources when the Service Template is executed namely `foo_0` and `foo_1`. Its is equivalent to defining two sub-resources like:

```
resources {
    foo_0 {
        type = tosca.resourceTypes.Noop
        properties {
            data = {p1 = 0}
        }
    }
    foo_1 {
        type = tosca.resourceTypes.Noop
        properties {
            data = {p1 = 1}
        }
    }
}
```

`forEach` directive is evaluated and expanded before the dependency tree is calculated, so these `forEach` resources can have dependency on other sub-resources and vice-versa. For example:

Example 2:

```

resources {
    subRes {
        type = tosca.resourceTypes.Noop
        forEach = [ "index", "foo_{index}" ], [ 0,1 ], false
        properties {
            data = { p1 = "Example" }
        }
    }
    t1 {
        title = "non forEach resource"
        type = tosca.resourceTypes.Noop
        activateAfter = foo_0
        properties {
            data = { p1 = "This is an Example" }
        }
    }
    t2 {
        title = "non forEach resource"
        type = tosca.resourceTypes.Noop
        properties {
            data = { s2_p1 = { getAttr = [foo_1, data.p1] } }
        }
    }
}

```

In the above example, `t1` sub-resource will be dependent(`activateAfter`) on `foo_0` resource and `t2`'s `data.s2_p1` property will be populated from `foo_1`'s `data.p1` attribute. `ForEach` block name (subRes) does not have any meaning or tie to the actual sub-resources after the foreach is expanded and would cause error if referred to by any sub-resources.

forEach Input Array

The input array of a `forEach` directive is static and is evaluated only once during activation process. If the parameter on which input array depends is updated after activation, the `forEach` block will **not** be re-evaluated.

Example:

In the example below, `forEach` sub-resource `s1`'s input array is evaluated on `intArray` attribute. Initially, if `intArray = [1,2]` it will result in 2 sub-resources `too_1` and `too_2`. Once the resource is activated and we update the `intArray` attribute to [4, 6, 2], the `forEach` block will still expand to 2 sub-resource `too_1` and `too_2` instead of `too_4`, `too_6` and `too_2`.

```

resources {
  s1 {
    type = test.resourceTypes.SomeDeclarativeServiceTemplate
    forEach = [ "foo", "too_{{foo}}", {getParam = intArray}, false ]
    activateAfter = too_str
    properties {
      value1 { getAttr = [ too_str, data.p1] }
      value2 { extract = [ {getParam = myIntArray}, "${{{foo}}}", true ] }
      value3 { getParam = myIntArray }
    }
  }
  s2 {
    type = test.resourceTypes.TestNoop
    forEach = [ "foo", "too_{{foo}}", ["str", "btr"], false ]
    properties {
      data {
        p1 = "{{foo}}"
      }
    }
  }
}

```

In value2 { extract = [{getParam = myIntArray}, "\${{{foo}}}", true]}, foo is initially evaluated to 1 and 2. As foo is not re-evaluated again after update, it will extract 1st and 2nd element from the new array, instead of new values 4, 6 and 2. getCurrent will also evaluate to old array values.

getAttr

Fetch a property of one of the other peer resources.

Syntax:

```
{ getAttr = [ <resource-name>, <json-path>, <observed> ] }
```

where:

- <resource-name> is the name of the peer resource as it is named in the template.
- <json-path> is either the name of the property you want to fetch as is, or it can be JSON path dot-notation describing the attribute of an object property. If the property or attribute is optional it may not have been supplied by the user. In this case, the directive will fallback to the observed value if possible.
- <observed> is an optional boolean value (default false) which determines the order in which desired and observed are tried. If false, the desired value is tried first, with fallback to the observed value. If true, the observed value is tried first, with fallback to the desired value. Note: the distinction between an observed value and desired value only exists if it is a config = true property.

Examples:

```
vlan = { getAttr = [ l2vpn, gwPortVlanId ] }
```

```
aEnd = { getAttr = [ l2vpn, data.nodeIdA ] }
```

```
aEnd = { getAttr = [ l2vpn, "data.endpoints[0].nodeId" ] }
```

In each example the data is fetched from the `l2vpn` peer resource. The second example shows a simple nested addressing of the `nodeIdA` attribute of the `data` property of `l2vpn`. The third example shows the nesting of an `endpoints` array under the `data` object. The first entry in the JSONPath query response is provided to the `getAttr` directive.

getContext

Fetch information from the execution context.

Syntax:

```
{ getContext = "userId" | "tenantId" | "roleIds" | "uri" }
```

where:

- `userId` is the UUID of the user that is operating on the resource.
- `tenantId` is the UUID of the tenant that is operating on the resource.
- `roleIds` are the comma-separated UUIDs of the roles for the user operating on the resource.
- `uri` is the scheme and host/port portion of the URI to reach the Orchestrator.

Example:

```
uri = { join = [ "/", { getContext = uri }, "market/api/v1/resources" ] }
```

getCurrent

Fetch the instance value in a looping directive.

Syntax:

```
{getCurrent = <iterator-name>}
```

where:

- `iterator-name` is the symbol name to refer the current item in the generator expression.

Example 1:

```
{ map = ["curr", [1, 3, 5], {join = ["_", "vnf", {getCurrent = "curr"}]} ]}
```

We prepend `vnf_` to each element of the array, returning `["vnf_1", "vnf_3", "vnf_5"]`.

Example 2:

```
subRes {
    type = tosca.resourceTypes.Noop
    forEach = ["current", { join = ["", "vnf_", {extract = [{getCurrent = current}, ".$.y", true]}]}, {getParam = prop2}, false]
    properties {
        data = {p1 = {extract = [{getCurrent = current}, ".$.x", true]}}
    }
}
```

In the above example, if `prop2 = [{"x": 1, "y": 100}, {"x": 2, "y": 200}]`, the resource section will expand to:

```
resources {
    vnf_100 {
        type = tosca.resourceTypes.Noop
        properties {
            data = {p1 = {extract = [{"x": 1, "y": 100}, ".$.x", true]}}
        }
    }
    vnf_200 {
        type = tosca.resourceTypes.Noop
        properties {
            data = {p1 = {extract = [{"x": 2, "y": 200}, ".$.x", true]}}
        }
    }
}
```

getCurrentIndex

Fetch iterator's index value in any looping directives (e.g `map`, `FilterArray`, `getSubResourceNames` and `forEach`).

Syntax:

```
{getCurrentIndex = <iterator-name>}
```

where:

- `iterator-name` is the symbol name to refer the current item in the generator expression.

Example:

```
{ map = ["curr", ["hello", "hi", "bye"], { "join": ["_", {getCurrent = curr}, {getCurrentIndex = curr}]] } }
```

In the above example, we will concatenate array's value with their index value i.e it will return ["hello_0", "hi_1", "bye_2"].

getDomain

Fetch the ID of the domain containing a peer resource. This directive is used to affect how the [Product Locator](#) selects a product (and corresponding domain) for a resource.

Syntax:

```
{ getDomain = <resource-id> }
```

where:

- `<resource-id>` is the orchestrator unique id of an arbitrary resource stored in the resource catalog or `this` to refer to the host resource.

Examples:

```
providerId = { getDomain = "067e6162-3b6f-4ae2-a171-2470b63dff00" }
```

```
providerId = { getDomain = this }
```

```
providerId = { getDomain = { getResourceId = nodeA } }
```

getDomainWith

Query to find the ID of a domain that matches the query parameters specified in the directive. This directive is used to affect how the [Product Locator](#) selects a product (and corresponding domain) for a resource.

Syntax:

```
{ getDomainWith = <query-parameters> }
```

where:

- <query-parameters> is a JSON object of key-value parameters to query on the domain

Examples:

1. Query the ID of the first domain that is of type WorkflowManager

```
domainId = { getDomainWith = { domainType = "WorkflowManager" } }
```

2. Query the ID of the first domain that is owned by the same tenant as the host resource and is of type OpenStack

```
domainId = {
    getDomainWith = {
        tenantId = { getResourceTenantId = this }
        domainType = "urn:cyaninc:bp:domain:openstack"
    }
}
```

3. Query the ID of the first domain that is owned by the same tenant as the sibling resource subnet and is of type Firefly

```
domainId = {
    getDomainWith = {
        tenantId = { getResourceTenantId = { getResourceId = "subnet" } }
        domainType = "urn:cyaninc:bp:domain:firefly"
    }
}
```

getIpv4Subnet

Return the subnet property of an IPv4 interface address in CIDR x.x.x.x/y format. The subnet length is also included in the resulting subnet value.

Syntax:

```
{ getIpv4Subnet = [ <ipv4-interface-address> ] }
```

where:

- <ipv4-interface-address> is the valid IPv4 interface address string in x.x.x.x/y format.

The following inputs will result in an error:

- <ipv4-interface-address> as malformed IPv4 address
- <ipv4-interface-address> with broadcast address in network part
- <ipv4-interface-address> with all zeros in host part
- <ipv4-interface-address> missing subnet length
- <ipv4-interface-address> with subnet length > 30

Example:

```
ipv4Subnet = { getIpv4Subnet = [ "192.168.121.31/24" ] }
```

which would yield `ipv4Subnet = "192.168.121.0/24"`

getObservedOptionalAttr

Fetch the observed property of one of the other peer resources. Fallback to the desired value. Failing that fallback to the default value. Failing that return nothing. Note: the distinction between an observed value and desired value only exists if it is a config = true property.

Syntax:

```
{ getObservedOptionalAttr = [ <resource-name>, <json-path>, <default-value> ] }
```

where:

- <resource-name> is the name of the peer resource as it is named in the template.
- <json-path> is either the name of the property you want to fetch as is, or it can be JSON path dot-notation describing the attribute of an object property. If the property or attribute is optional it may not have been supplied by the user. In this case, the directive will fallback to the observed value if possible.
- <default-value> is the fallback to use if the attribute does not exist on the peer resource. This value is optional. If this value IS NOT specified and the attribute DOES NOT exist on the peer resource, the directive returns nothing.

The first entry in the JSONPath query response is provided to the directive.

Example:

```
az = { getObservedOptionalAttr = [ vml, availabilityZone, "west-1" ] }
```

getOptionalAttr

Fetch the desired property of one of the other peer resources. If the desired property does not exist, fetch the observed value. If the observed value also does not exist then use the default fallback or return

nothing. Note: the distinction between an observed value and desired value only exists if it is a config = true property.

Syntax:

```
{ getOptionalAttr = [ <resource-name>, <json-path>, <default-value> ] }
```

where:

- <resource-name> is the name of the peer resource as it is named in the template.
- <json-path> is either the name of the property you want to fetch as is, or it can be JSON path dot-notation describing the attribute of an object property. If the property or attribute is optional it may not have been supplied by the user. In this case, the directive will fallback to the observed value if possible.
- <default-value> is the fallback to use if the attribute does not exist on the peer resource. This value is optional. If this value IS NOT specified and the attribute DOES NOT exist on the peer resource, the directive returns nothing.

The first entry in the JSONPath query response is provided to the directive.

Example:

```
az = { getOptionalAttr = [ vml, availabilityZone, "west-1" ] }
```

getOptionalParam

Fetch a property of the host resource. If the property does not exist, return nothing.

Syntax:

```
{ getOptionalParam = <json-path> }
```

where:

- <json-path> is either the name of the input property provided to the host-resource, or it can be JSON path dot-notation describing the attribute of an object property. The first entry in the JSONPath query response is provided to the directive. If the parameter does not exist, the directive evaluates to nothing.

Examples:

```
input = { getOptionalParam = property1 }
```

```
input = { getOptionalParam = obj1.arr1[4].c }
```

getParam

Fetch a property of the host resource. If the property does not exist, raise an exception.

Syntax:

```
{ getParam = <json-path> }
```

where:

- <json-path> is either the name of the input property provided to the host-resource, or it can be JSON path dot-notation describing the attribute of an object property. The first entry in the JSONPath query response is provided to the directive. A template exception is thrown if the parameter does not exist on the host resource.

Example:

```
input = { getParam = property1 }
```

```
input = { getParam = obj1.arr1[4].c }
```

getProviderResourceId

Fetch the provider resource id of a peer resource.

Syntax:

```
{ getProviderResourceId = <resource-id> }
```

where:

- <resource-id> is the orchestrator unique id of an arbitrary resource stored in the resource catalog.

Example:

```
provId = { getProviderResourceId = "067e6162-3b6f-4ae2-a171-2470b63dff00" }
```

getResourceAttr

Fetch a property of an arbitrary resource stored in the resource catalog.

Syntax:

```
{ getResourceAttr = [ <resource-id>, <property-path>, <observed> ] }
```

where:

- <resource-id> is the orchestrator unique id of an arbitrary resource stored in the resource catalog.
- <property-path> is either the name of the property you want to fetch as is, or it can be a dot-notation path into a attribute of an object property. The dot notation can also refer to a more deeply nested attribute as longs as the nesting is object in an object in an object in an object, etc. The path is based from the top of the resource so the prefix `properties` needs to be included to access the value of an onboarded property.
- <observed> is an optional boolean value (default false) which will determine the order in which values are tried. If false, the desired value is tried, if it does not exist, then the observed value is tried. If true, then the observed value is tried, if it does not exist, then the desired value is tried. Note: the distinction between an observed value and desired value only exists if it is a config = true property.

Examples:

- Get the `ip_address` property from the specified resource

```
value = { getResourceAttr = [ "067e6162-3b6f-4ae2-a171-2470b63dff00",  
    properties.ip_address ] }
```

- Get the `orchState` top-level property from a given resource

```
value = { getResourceAttr = [ "067e6162-3b6f-4ae2-a171-2470b63dff00", orchState  
    ] }
```

- Get the observed value (if different from desired and config = true, otherwise the observed value) for `ip_address` from the specified resource

```
value = { getResourceAttr = [ "067e6162-3b6f-4ae2-a171-2470b63dff00",  
    properties.ip_address, true ] }
```

getOptionalResourceAttr

Fetch a property of an arbitrary resource stored in the resource catalog. If the Resource or property doesn't exist, return nothing. This contrasts with `getResourceAttr` which will throw a

TemplateExecutionException if the specified Resource ID is not found, or the property is not found.

Syntax:

```
{ getOptionalResourceAttr = [ <resource-id>, <property-path>, <observed> ] }
```

where:

- <resource-id> is the orchestrator unique id of an arbitrary resource stored in the resource catalog.
- <property-path> is either the name of the property you want to fetch as is, or it can be a dot-notation path into a attribute of an object property. The dot notation can also refer to a more deeply nested attribute as longs as the nesting is object in an object in an object in an object, etc. The path is based from the top of the resource so the prefix properties needs to be included to access the value of an onboarded property.
- <observed> is an optional boolean value (default false) which will determine the order in which values are tried. If false, the desired value is tried, if it does not exist, then the observed value is tried. If true, then the observed value is tried, if it does not exist, then the desired value is tried. Note: the distinction between an observed value and desired value only exists if it is a config = true property.

Examples:

- Get the ip_address property from the specified resource. If properties.ip_address is not defined on the resource, the result will be nothing.

```
value = { getOptionalResourceAttr = [ "067e6162-3b6f-4ae2-a171-2470b63dff00",  
properties.ip_address ] }
```

- Get the observed value (if different from desired and config = true, otherwise the observed value) for ip_address from the specified resource. Likewise, if there is no observed value for ip_address, it will return nothing.

```
value = { getOptionalResourceAttr = [ "067e6162-3b6f-4ae2-a171-2470b63dff00",  
properties.ip_address, true ] }
```

getResourceld

Obtain the resource id of a peer resource.

Syntax:

```
{ getResourceId = <resource-name> }
```

where:

- <resource-name> is the name of the peer resource as it is named in the template or `this` to refer to the host resource.

Example:

```
value = { getResourceId = nodeA }
```

getResourceTenantId

Obtain the tenant ID for a resource based on the resource ID. This directive can be used with [getDomainWith](#) to control the domain in which a resource will be created.

Syntax:

```
{ getResourceTenantId = <resource-id> }
```

where:

- <resource-id> is the ID of a resource or `this` to refer to the host resource.

Examples:

1. Get the tenant ID of the host resource

```
tenantId = { getResourceTenantId = this }
```

2. Get the tenant ID for a sibling resource named nodeA

```
tenantId = { getResourceTenantId = { getResourceId = nodeA } }
```

3. Get the tenant ID for a resource with the specified ID

```
tenantId = { getResourceTenantId = "57741aa2-df13-4bc4-ab89-922e04c560f4" }
```

getResourceWith

Query to find a matching resource ID.

Syntax:

```
{ getResourceWith = { <query-parameters> } }
```

where:

- <query-parameters> is a JSON object of key-value parameters to query on the Resource.

See [Resource Locator](#) for more details.

Example:

```
value = { getResourceWith = { resourceTypeId = "example.ResourceTypes.ExampleType" } }
```

getResourcesWith

Query to find an array of matching resource IDs.

Syntax:

```
{ getResourcesWith = { <query-parameters> } }
```

where:

- <query-parameters> is a JSON object of key-value parameters to query on the Resource.

See [Resource Locator](#) for more details.

Example:

```
value = { getResourcesWith = { resourceTypeId = "example.ResourceTypes.ExampleType" } }
```

getSubResourceNames

Query to find names of all sub-resources in the Output section.

Syntax:

```
{ getSubResourceNames = [ <iterator-name>, <boolean filter condition> ] }
```

where:

- iterator-name is the name of the instance from input-array which can be used for filtering array elements.
- boolean-condition is evaluated to specify if the instance in the array should be included or not.

Return Type: The directive returns a filtered array based on the condition specified.

`iterator-name` and `boolean-condition` are optional fields, if these fields are not provided, the directive will return all resource identifiers.

Example:

```
output {  
    value1 {getSubResourceNames = []}  
    value2 {getSubResourceNames = ["curr", { "isEqual" : [{"getCurrent": "curr"},  
    "resourceName"] } ]}  
}
```

ifElse

Select between two values based on a boolean condition.

Syntax:

```
{ ifElse = [ <condition>, <true-value>, <false-value> ] }
```

where:

- `<condition>` is a boolean value or expression.
- `<true-value>` is the value or expression to be used if the condition is satisfied.
- `<false-value>` is the value or expression to be used if the condition is not satisfied.

The following JSON values evaluate to true when used in the `<condition>`:

- boolean true
- array with length greater than 0
- object with 1 or more fields
- non-empty string
- integer or floating point numbers not equal to 0

The following JSON values evaluate to false when used in the `<condition>`:

- boolean false
- array with zero length
- object with zero fields
- empty string
- integer or floating point numbers equal to 0
- nothing or null values

Example:

```
subnet = {
    ifElse = [
        { getParam = firewallEnabled },
        "067e6162-3b6f-4ae2-a171-2470b63dff00",
        "54947df8-0e9e-4471-a2f9-9af509fb5889"
    ]
}
```

isEqual

Return boolean result of equality test on two operands.

Syntax:

```
{ isEqual = [ <left-hand-side>, <right-hand-side> ] }
```

where:

- <left-hand-side> is one of the values or expressions to compare.
- <right-hand-side> is the other value or expression to compare.

Example:

```
bitrate = { ifElse = [ { isEqual = [ { getParam = "service" }, "gold" ] }, "1Gbps",
    "100Mbps" ] }
```

isGreater

Return boolean result of > test on two numerical operands.

Syntax:

```
{ isGreater = [ <left-hand-side>, <right-hand-side> ] }
```

where:

- <left-hand-side> is one of the values or expressions to compare.
- <right-hand-side> is the other value or expression to compare.

Example:

```
service = { ifElse = [ { isGreater = [ { getParam = "bitrate" }, 1000000000 ] },
"gold", "silver" ] }
```

isGreaterEqual

Return boolean result of \geq test on two numerical operands.

Syntax:

```
{ isGreaterEqual = [ <left-hand-side>, <right-hand-side> ] }
```

where:

- *<left-hand-side>* is one of the values or expressions to compare.
- *<right-hand-side>* is the other value or expression to compare.

Example:

```
service = { ifElse = [ { isGreaterEqual = [ { getParam = "bitrate" }, 1000000000 ] },
}, "gold", "silver" ] }
```

isLess

Return boolean result of $<$ test on two numerical operands.

Syntax:

```
{ isLess = [ <left-hand-side>, <right-hand-side> ] }
```

where:

- *<left-hand-side>* is one of the values or expressions to compare.
- *<right-hand-side>* is the other value or expression to compare.

Example:

```
service = { ifElse = [ { isLess = [ { getParam = "bitrate" }, 1000000000 ] },
"silver", "gold" ] }
```

isLessEqual

Return boolean result of \neg test on two numerical operands.

Syntax:

```
{ isLessEqual = [ <left-hand-side>, <right-hand-side> ] }
```

where:

- <left-hand-side> is one of the values or expressions to compare.
- <right-hand-side> is the other value or expression to compare.

Example:

```
service = { ifElse = [ { isLessEqual = [ { getParam = "bitrate" }, 1000000000 ] },
"silver", "gold" ] }
```

isNotEqual

Return boolean result of inequality test on two operands.

Syntax:

```
{ isNotEqual = [ <left-hand-side>, <right-hand-side> ] }
```

where:

- <left-hand-side> is one of the values or expressions to compare.
- <right-hand-side> is the other value or expression to compare.

Example:

```
bitrate = { ifElse = [ { isNotEqual = [ { getParam = "service" }, "gold" ] },
"1Gbps", "100Mbps" ] }
```

join

Concatenate arbitrary number of values into one string, using a "glue" string between the components.

Syntax:

```
{ join = [ <glue-string>, <value1>, <value2>, ... ] }
```

where:

- <glue-string> is the string that will be placed between the values provided. It can be an empty

string " ", or it can be any other string value.

- <valueX> are the values that need to be concatenated. If a non-string value is used, it will be converted to string before the join operation is performed.

Example:

```
output = { join = [ " ", Hello, "World!" ] }
```

len

Extract the length of an array, an object, or a string.

Syntax:

```
{ len = <value> }
```

where:

- <value> is an array, an object, a string, or an expression that resolves to these three types. Return the number of items (in case of array), the number of attributes (in case of an object), or the number of characters (in case of a string).

Examples:

```
a = { len = [1, 2, 3, "foo"] }
b = { len = {a = 42, b = "bar"} }
c = { len = "baz" }
```

which would yield a = 4, b = 2, and c = 3.

lookupInStringMap

Look up a string value in a hash-map by its key.

Syntax:

```
{ lookupInStringMap = [ <string-map>, <entry-separator>, <key-value-separator>,
<key> ] }
```

where:

- <string-map> is a string-encoded map, using consistent entry-separator and key-value separator strings. For example, "a=12,b=true;c=foo" or "a:12|b:true|c:foo" are both proper encoding of the same 3-entry map.

- <entry-separator> is the string or character delineating the records in the map. Examples: , , :, .
- <key-value-separator> is the string or character delineating each key from its value in the map. Examples: =, :.
- <key> is the key that is used to look up a value in the map.

Example:

```
value = { lookupInStringMap = [ "a=12;b=true;c=foo", ";" , "=" , "c" ] }
```

The above will assign the value "foo" to value.

map

Map an array of things into another array of things.

Syntax:

```
{ map = [ <iterator-param>, <input-array>, <expression> ] }
```

where

- <iterator-param> is the symbol name to refer to the current item in the generator expression.
- <input-array> is the array that will be iterated over. In each iteration the value referenced by the parameter <iterator-param> is the value of the corresponding item in the input array.
- <expression> is a JSON value (typically an array or object) which can contain arbitrary template directives to construct values. The {getParam = <iterator-param>} or {getCurrent = <iterator-param>} expression can be used to fetch the current input item.

Example:

```
things = { map = [ i, [ 1, 2, 42 ], { value = { getParam = i } } ] }
```

will assign [{value = 1}, {value = 2}, {value = 42}] to things.

or

Return boolean result of logical or test on two operands.

Syntax:

```
{ or = [ <left-hand-side>, <right-hand-side> ] }
```

where:

- <left-hand-side> is one of the boolean values or expressions to evaluate.
- <right-hand-side> is the other boolean value or expression to evaluate.

Example:

```
premiumStatus = { or = [ { isEqual = [ { getParam = "service" }, "gold" ] }, { isEqual = [ { getParam = "service" }, "silver" ] } ] }
```

oscall

Fetch some data using OS Shell call

Syntax:

```
{ oscall = [ <shell-command>, <args\...> ] }
```

where:

- <shell-command> is the shell command to be executed
- <args...> is the command arguments

Examples:

```
a = { oscall = [ "/bin/echo", "abc" ] }
b = { oscall = [ "/usr/bin/python /tmp/template_test.py", { "getParam": "in6" } ] }
```

py2eval

Execute short Python scripts in declarative sub-resources definition.

Syntax:

```
{ py2eval = [ <expr>, <arg_0>, <arg_1>, ... <arg_k> ] }
```

where:

- `expr` is the Python fragment to be evaluated given as a string. It follow Python 2.7 syntax.
- `arg_0`, `args_1` are variable number of input arguments to the expression.

The python expression should return a value using a `return` statement.

Input arguments are accessed as positional arguments in the expression i.e `arg_0` can be access as `args[0]`, `arg_1` as `args[1]` and so on.

- The only importable modules are `json` and `re`.

Example 1:

```
{ py2eval = [ """ip = args[0]; return ip.rsplit(.) """, { getParam = myIP } ] }
```

Example 2:

```
{ py2eval = [ "return args[0]+ args[1]", { getParam = myString }, {getAttr = [s2, data.p1]} ] }
```

Example 3:

```
{ py2eval = [ "inter = args[0][0]; temp1 = args[1]; temp2 = args[2][0]; return inter+temp1+temp2", {getParam = myIntArray}, 1, [555, 224, 666] ] }
```

Example 4:

Multi-line examples (Do not use triple quotes in the fragment definition):

```
p1 { py2eval = [ "for x in args[0]:\n\tfor y in args[1]:\n\t\treturn x", { getParam = myIntArray }, [4,5,6] ] }
```

```
p2 { py2eval = [ "a=[ ]\nfor x in args[0]:\n\tfor y in\nargs[1]:\n\t\tta.append((x,y))\nreturn a", { getParam = myIntArray }, [4,5,6] ] }
```

```
p3 { py2eval = [ "a=[ ]\nb=[6,7]\nfor x in args[0]:\n\tfor y in\nargs[1]:\n\t\tta.append((x,y))\nfor x in b:\n\tta.append(x)\nreturn a", { getParam = myIntArray }, [4,5] ]` }
```

range

Generate an array of integers in a given range.

Syntax:

```
{ range = <max> }
{ range = [ <min>, <max> ]
{ range = [ <min>, <max>, <step> ] }
```

where:

- <min> is the lower end of the generated range (inclusive). Defaults to 0 if not provided.
- <max> is the upper end of the generated range (not inclusive).
- <step> is the increment between the values. Defaults to 1 if not provided.

Examples:

```
a = { range = 10 }
b = { range = [5, 10] }
c = { range = [5, 10, 2] }
```

which is the equivalent of:

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
b = [5, 6, 7, 8, 9]
c = [5, 7, 9]
```

renderTemplate

Construct a string using a "moustache-style" text template with variable substitution.

Syntax:

```
{ renderTemplate = [ <substitution-map>, <text-template> ] }
```

where:

- <substitution-map> is a JSON object assigning arbitrary values to arbitrary keys.
- text-template is a string containing double-moustache expressions to denote variable substitutions. For instance, if the substitution-map contains a date = "2008-11-11" key/value pair, then anywhere in the template the {{date}} string can be used to inject the above date value.

Example:

```
text = { renderTemplate = [
    to = Joe
    subject = "Firedrill"
    date = "2008-11-11"
  },
  "Dear {{to}}, Regarding your query about the {{subject}}, we are pleased to
  confirm that it will happen on {{date}}."
]
```

setOnlyIf

Set a value only if the boolean condition evaluates to true. Otherwise, the directive evaluates to `Nothing`.

Note that this is similar to `ifElse` except `Nothing` is the value returned when false which cannot be expressed in the `ifElse` statement (i.e., `Nothing` is different than an empty object `{ }`).

Syntax:

```
{ setOnlyIf = [ <condition>, <true-value> ] }
```

where:

- `<condition>` is a boolean value or expression as described in `[ifElse](#ifElse)`
- `<true-value>` is the value or expression to be used if the condition is satisfied.

Example:

```
subnet = { setOnlyIf = [ { getParam = firewallEnabled }, "067e6162-3b6f-4ae2-a171-2470b63dff00" ] }
```

split

Return a positional part from a string split by a field separator.

Syntax:

```
{ split = [ <string-to-split>, <field-separator>, <positional-index> ] }
```

where:

- `<string-to-split>` is the string that will be split by the delimiter provided. It must be string value.
- `<field-separator>` is the delimiter used by the split operation. It must be string value.
- `<positional-index>` is the index to the array of strings when the split operation is applied on the `<string-to-split>`. It must be integer value. The string array resulting from `<split>` is zero-based.

The following input parameter combinations will result in an error:
* `<field-separator>` not found in `<string-to-split>`: `<string-to-split>` is returned in index 0
* `<positional-index>` out-of-bounds in the expected string array
* `<field-separator>` or `<string-to-split>` not string value

Example:

```
subnetLength = { split = [ "192.168.121.0/24", "/", 1 ] }
```

which would yield subnetLength = 24.

terminateBefore

Specify that the termination of a specific resource must precede the termination of another. When you add a terminateBefore attribute to a resource, that resource is terminated before the termination of the target resource or resources specified in the terminateBefore attribute. You can use the terminateBefore attribute with any resource. If termination of the resource with the terminateBefore attribute is not successful, the target resource will not be terminated.

Syntax:

```
{ terminateBefore = <resource-name> }
```

```
or if specifying multiple resources
```

```
{ terminateBefore = [ <resource1-name>, <resource2-name>, ... <resourcek-name> ] }
```

where:

- <resource-name> is a resource named in the same template.

Example:

```
resources {
    subnet {
        ...
    }
    testVm {
        ...
        terminateBefore = subnet
    }
}
```

Additional Directive Examples and Usage Guidelines

Optional Property

In some cases, it is necessary to have the existence of a property to be conditional. If a property object evaluates to **Nothing**, the property will not exist on the resource. If the directive is used as an item in an array, the array will be constructed without this item.

The example below will define the property `staticRouting` on a sub-resource if the parent resource has a defined value for the optional property `parentStaticRouting`. Otherwise, it evaluates to an empty object.

Example:

```
properties {
    staticRouting = {getOptionalParam = parentStaticRouting}
}
```

or with a more complex condition where the `staticRouting` property is only defined if the `parentStaticRouting` array has a length greater than 1.

```
properties {
    staticRouting = {setOnlyIf = [{isGreater = [{len = {getOptionalParam =
parentStaticRouting}}, 1]}, {getOptionalParam = parentStaticRouting}]}}
}
```

Implicit Dependencies

Properties of all resources will be parsed during activation and implicit `activateAfter` dependencies will be added between resources based on some directive used. List of these directives are

```
getAttr, getOptionalAttr, getObservedOptionalAttr and getResourceId
```

In the example below, `subnet` resource has a property that uses `getAttr` directive to fetch a property of `testVm` resource. This will create an implicit `activateAfter` dependency between `subnet` and `testVm` i.e `subnet` will be activated after `testVm`.

```
resources {
    subnet {
        properties {
            someProperty = {getAttr = [testVm, "VmProp"]}
        }
    }
    testVm {
        properties {
            VmProp = ...
        }
    }
}
```

Note: There will be no implicit dependencies added if resource names in the above directives are expressed as `getCurrent` directive which evaluate to a string.

Example:

```
resources {
    subnet {
        properties {
            someProperty = {map = ["curr", ["testVm1", "testVm2"], { getAttr =
[getCurrent = "curr", "VmProp"]}]}
        }
    }
    testVm1 {
        properties {
            VmProp = ...
        }
    }
    testVm2 {
        properties {
            VmProp = ...
        }
    }
}
```

This will not create any implicit dependencies between `subnet`, `testVm1` and `testVm2`.

Resource names as fixed string

Some directives which take resource names as arguments require that names be provided as a fixed string— directives which evaluate to a string are not permitted. List of these directives are:

```
activateAfter and terminateBefore
```

Imperative Plans

Service templates may associate **imperative plans** with a resource type to implement [lifecycle](#), [validation](#), and [custom](#) resource operations.

There are two types of imperative plans:

- [Remote Scripts](#) (`type=remote`) are written as a Python module. They are executed by the `scriptplan` app in a Python virtual environment and can specify arbitrary Python dependencies. They can also use the Plan SDK library, which eases interaction with orchestrator APIs.
- [Legacy Scripts](#) (`type=script`) are written as Python 2.7 scripts and executed directly by the `bpo-core` app. They can make use of a limited number of Python modules.

Lifecycle and custom operation plans are specified as blocks in the **plans** section of a [service template](#). For example, to specify a custom terminate plan a service template could include this section:

```
plans {
    terminate {
        type = remote
        language = python
        path = scripts.customterminate.Terminate
    }
}
```

Plans for validation operations are specified in the **validators** section of the service template.

This section describes the various operations plans can implement, then delves into the details of implementing remote scripts and legacy scripts.

Imperative Lifecycle Operations

See [Lifecycle Operations](#) for general information on lifecycle operations.

NOTE

It is possible mix declarative templating (**resources** section) and **plans**, subject to some restrictions. See [Mixing Imperative and Declarative Lifecycle Operations](#) for details.

The values in the resource instance's **properties** correspond to the **desired state** of the `config` properties on the resource. It is the responsibility of the plan to execute the required logic to *make it so*.

The **operative resource** is a template managed resource provided by the built-in resource provider. This means that it doesn't have a Resource Adapter mediator managing the corresponding entity within a managed domain. The plan is responsible for managing the truth of the observed state for the operative resource.

Imperative Activate

An imperative activate plan must implement the [Activate Operation](#).

The plan is responsible for determining the truth of the observed state for the operative resource. By default, the differences are empty. This implies that the observed state and the desired state are the same or alternatively the observed state is not yet known. If the observed state doesn't match the desired state, the plan should use the PUT/PATCH `resources/{resourceId}/observed` API to set the differences before exiting.

Common behaviors in an **activate** plan include:

- Creating sub-resources
- Creating relationships between resources

Imperative Update

An imperative update plan must implement the [Update Operation](#).

The differences on the operative resource indicate any differences between the currently observed values for the resource's config properties and the desired values. The GET `resources/{resourceId}/observed` API returns the observed values of the resource (i.e., the result of applying the JSON patch differences to the desired state).

The update plan is responsible for applying the required logic to bring the resource state in line with the desired state.

If `autoClearDifferences` is set to true on the plan, the differences on the resource are cleared when the plan successfully exits. This indicates that the **observed state** has reached the **desired state**.

If `autoClearDifferences` is set to false on the plan, the differences are not cleared automatically. The script should explicitly PUT or PATCH the **observed** state of the resource.

```
PATCH /resources/{resourceId}/observed
```

See [Differences](#) for more information.

Imperative Terminate

The terminate plan is responsible for applying the required logic to terminate the resource. Common behaviors include:

- Removing relationships between resources
- Terminating sub-resources

A successful exit code results in the resource `orchState` being set to terminated.

Validation Operations

A validation operation may be defined for each of the lifecycle operations: [activate](#), [update](#), and [terminate](#).

Validate Activate

For a create validation plan, there will be no `RESOURCE_ID` passed as an environment variable, as there is not yet a resource instance in market. The `INPUTS` variable will contain a `resource` and a `full`, as in the following example

```
{  
    "full": false,  
    "resource": {  
        "product": "017fdc46-973c-11e6-ae22-56b6b6499611",  
        "properties": {  
            "prop1": "value1"  
        }  
    }  
}
```

The `full` value indicates whether the `resource` value should be validated as a full or partially-filled resource. How the `full` flag is used is up to the plan author.

If the validation passes, the plan should return 0 and optionally output a passed Validation Report to the standard output.

If the validation fails, the plan should return 1 and output a failed Validation Report to the standard error with a list of failure details.

Validate Update

For an update validation, the `RESOURCE_ID` will be passed as an environment variable. In addition, the `INPUTS` variable will contain a `resource` that represents the **intended** state for the resource after the update. The plan code can callback to BPO using the `RESOURCE_ID` to get the current state of the resource.

The update validation plan should return and output the same way as the create validation plan.

Validate Terminate

For a delete validation, the `RESOURCE_ID` will be passed as an environment variable. The plan code can callback to BPO using the `RESOURCE_ID` to get the current state of the resource.

The delete validation plan should return and output the same way as the create and update validation plans.

Custom Operations

In addition to `RESOURCE_ID`, a custom plan will be called with `OPERATION_ID` in the environment variables. Using both IDs, the plan can callback to BPO to get the Resource Operation that ran the plan, and the `inputs` of the Resource Operation.

If the operation is successful, the plan should produce outputs conforming to the schema defined in the corresponding interface of in the Resource Type. The plan can observe-patch the host resource if the operation changes the state of the resource. The plan can potentially call market API to trigger lifecycle or custom operations of another resource.

NOTE For a specific resource, lifecycle and custom operations are queued and a new one will not start until the previous one finishes or fails. As a result, if a custom operation calls (non-observed) update or create another custom operation on the host resource and waits for it to complete, the wait will timeout since the new operation will not start until the current one completes.

Remote Scripts

Legacy Scripts

Legacy scripts accept parameters via environment variables and provide output by writing JSON to stdout. This type of plan is indicated in the `plans` section by `type = script`. You must also specify `language = python` and the path to your `.py` file relative to the root of the model-definitions repository:

```
plans {  
  terminate {  
    type = script  
    language = python  
    path = "path/to/script.py"  
  }  
}
```

NOTE Legacy scripts are deprecated and will be removed in a future orchestrator release. See [Migrating Legacy Scripts to Remote Scripts](#) for migration tips.

The structure and requirements of an imperative plan script are described below.

Environment Variables

Arguments are provided to the plan as environment variables. For example, obtain the resource ID for the operative resource from the `RESOURCE_ID` environment variable.

```
import os
resource_id = os.environ[ "RESOURCE_ID" ]
```

The environment variables provided as arguments to the script are in the table below:

VARIABLE	TYPE	DEFINITION	EXAMPLE VALUES
API_URI	URL	Base URL for API calls back into the orchestrator	"http://localhost:8181"
LOG_FILE	Path	String to use as the fully-qualified file name for opening a file for logging. Path to a file to open for writing progress to logs (script manages opening and closing log).	"/bp2/log/plan-log"
BP_USER_ID	UUID	User ID on whose behalf the operation is being run	"b54b86b0-0ebd-4652-a343-e9dd25732d60"
BP_TENANT_ID	UUID	Tenant ID on whose behalf the operation is being run	"7c7da24e-d2af-35c7-8f6d-d8d16c7f0738"
BP_ROLE_ID	UUID	Set of Role IDs associated with the user on whose behalf the operation is being run	[{"670afffa-11c0-40cb-8187-3da2242fbe0d"}]
C_TRACE	UUID	Trace ID associated with the top-level operation. Pass through unaltered in downstream calls.	"2d8bcc4a-d18d-4ef6-894e-a2340c84584f"
C_UPSTREAM	UUID	Hop ID assigned by the caller	"2d8bcc4a-d18d-4ef6-894e-a2340c84584f"
RESOURCE_ID	UUID	BPO resource ID of the resource for whom operation is applied, the operative resource	"917f85bd-6314-48dd-9b54-4410d35a6d18"
OPERATION	String	Name of the operation	"activate", "update", "terminate"
OPERATION_ID	String	ID of the resource operation	"dd2ff52a-972b-11e6-ae22-56b6b6499611"
INPUTS	String	JSON-encoded string for inputs resource validation inputs	'{"full":true, "resource":{"properties":{"prop1":"value1"}}}'

NOTE

Some environment variables can be None, depending on the operation or validation the plan is associated with. For example, for resource lifecycle operations, the OPERATION_ID and INPUTS will be None. For a resource create validation, RESOURCE_ID will be None.

The resource instance or resource operation data may be obtained by calling back into BPO with the specified IDs. The values in the resource instance's **properties** corresponds to the **desired state** of the config properties on the resource.

Script Output

The convention for script output is based on the type of operation.

Activation

The script signals a successful activation by exiting with code 0. The script may also (optionally) provide output formatted as JSON data to standard output. Do not include `properties` in the field names of the output object as it is assumed to be nested within the `properties` object. The output isn't directly applied to the operative resource. Instead, the output is applied to the `activate` task object accessible within the execution of the service template, but not persisted as an actual resource in the system.

The output values on the `activate` object can be referenced in the `output` section of the service template and used to set output (i.e., `output=true`) properties or non-config (i.e., `config=false`) properties. For example, if an output `assigned_name` is determined during the execution of the plan, the value may be set on the `activate` task object by writing the value to `stdout` in the script:

```
print json.dumps(dict(assigned_name="Alfred"))
```

and transferring the value in the `output` section of the service template:

```
output = {
    assigned_name = { getAttr = [ activate, assigned_name ] }
}
```

Update

Outputs are ignored on `update` operations.

Termination

Outputs are ignored on `terminate` operations.

Validation Plans

Outputs from a Validation Plan work the same way as outputs from an activation plan, i.e. by outputting a formatted JSON object to the standard output. The output JSON object should represent a [Validation Report](#).

Custom Operations

Outputs from a custom operation plan work the same way as outputs from an activation plan, i.e. by outputting a formatted JSON object to the standard output. The output JSON object should conform to the `outputs` schema of the corresponding interface in the Resource Type Definition.

Exit Status

0 indicates success. 1 or non-zero indicates failure.

A failure status results in the marking of the resource `orchState` as "failed".

The last line observed on stderr will be used as the `reason` property value on the resource.

An unhandled exception in the plan also results in a script failure and the marking of the resource `orchState` as "failed" with the exception message used as the `reason` property value on the resource.

Activation

A successful `activate` operation will result in marking the resource `orchState` as "active".

The failure of the `activate` operation should be signaled with a non-zero exit code or exception (as described above).

Update

A successful `update` operation does not change the `orchState` of the resource.

A failure of the `update` operation should be signaled with a non-zero exit code or exception (as described above).

Termination

A successful `terminate` operation results in the `orchState` of the resource being set to `terminated`.

A failure of the `terminate` operation should be signaled with a non-zero exit code or exception (as described above).

Validation Plans

If the validation passes, the plan should return 0 and optionally output a passed Validation Report to the standard output.

If the validation fails, the plan should return 1 and output a failed Validation Report to the standard error with a list of failure details.

No change to the market database should be done as a result of validation plan execution.

Custom Operations

A successful custom operation plan results in the state of the Resource Operation being set to successful.

A failure of a custom plan should be signaled with a non-zero exit code or exception (as described above). This will cause the state of the corresponding Resource Operation to be set to failed.

Logging

A plan can output details, progress, error conditions, etc. during execution by managing a file whose location is defined by the LOG_FILE environment variable.

```
import os
with open(os.environ["LOG_FILE"], "w") as f:

    def log(msg):
        print >> f, msg

    log("First line of log")
```

The log files are located in the directory /bp2/log/plan-log and the names encode the operation, resource ID, and time of execution.

```
plan-script-56730b85-b40b-42a0-9c33-d224a0ce06f0-activate-2015-12-17T19:22:45.089Z
```

API Calls Back into BPO

When making calls back into BPO, the context information provided in the inputs should be provided as headers. The Cyan-Internal-Request header should be added to indicate to BPO to accept the identity information without performing authentication on the request. The imperative plan is being executed as a trusted process and onboarding should be restricted based on trust. For example, to call back into BPO to get the operative resource, set the headers:

```
import os
self.headers = {
    'content-type': 'application/json',
    'accept': 'application/json',
    'Cyan-Internal-Request': '1',
    'bp-user-id': os.environ["BP_USER_ID"],
    'bp-tenant-id': os.environ["BP_TENANT_ID"],
    'bp-role-ids': os.environ["BP_ROLE_ID"],
    'C-Trace': os.environ["C_TRACE"],
    'C-Upstream': os.environ["C_UPSTREAM"],
}
```

and include the headers in the request for the resource:

```
rid = os.environ["RESOURCE_ID"]
res_uri = os.environ['API_URI'] + '/market/api/v1' + '/resources/%s' % rid
req_response = requests.get(res_uri, verify=True, headers=headers)
resource = req_response.json()
```

Similarly, a callback to get the resource operation for custom plans:

```
rid = os.environ[ "RESOURCE_ID" ]
opid = os.environ[ "OPERATION_ID" ]
op_uri = os.environ[ 'API_URI' ] + '/market/api/v1' + '/resources/%s/operations/%s' %
(rid, opid)
req_response = requests.get(res_uri, verify=True, headers=headers)
operation = req_response.json()
```

Execution Environment

The script is executed in a separate Python 2.7 process in the same container as **bpoCore**. The current working directory is set to the script directory.

A few Python packages are provided:

- [Requests 2.6](#)
- [Afkak](#)
- [dateutil](#)

It is not possible to add additional Python libraries to the legacy script execution environment, nor are there any guarantees as to the exact version of those provided. For more control, migrate to [Remote Scripts](#) – see the next section.

Migrating Legacy Scripts to Remote Scripts

While the legacy scripts and remote scripts are similar, the APIs are not compatible. You must make the following adjustments when migrating:

- Add a `requirements.txt` file which specifies dependencies for the remote script. Remote scripts do not have access to libraries beyond the Python standard library by default. For example, add `python-dateutil==2.5.3` to specify the same version of [dateutil](#) that legacy scripts have access to.
- Add `plansdk ~ 2.1` to the `requirements.txt` file. The `plan-sdk` library is a more featureful replacement for the `porchplan.py` that has shipped with BPO. You will want to consult the [plansdk API documentation](#) in detail, but there are a few major API differences:
 - Plans must subclass `plansdk.apis.plan.Plan` and override the `run()` method.
 - Parameters are available via `self.params`, which is a Python dict, rather than via environment variables. Parameter names have changed, and are now follow a "camelCase" convention rather than "UPPER_UNDERSCORES."

- Remote scripts must be importable Python modules.
 - Any containing directory structure must be a Python package: add `__init__.py` files to indicate this and ensure that directory names are valid Python identifiers (no hyphens).
 - The `path` specified in the service template's `plan` section is the fully-qualified name of the plan class rather than the filesystem path to the Python file.

Capability Types

A Capability Type defines categories of features which are provided or used by Resources. These categories can serve as reusable descriptors of common functionality. Capability Types are defined inside [Definition Modules](#).

Both the [Capabilities and Requirements](#) defined on Resource Types are instances of Capability Types. Capability Types are reused for both capability and requirement definitions to avoid the overhead of defining a separate set of artifacts (e.g, Requirement Types) when in practice they would be identical.

By default BPO ships a few basic Capability Types which can be used to form simple [Relationships](#) and/or serve as the basis for more complex Capability Types through inheritance.

The table below lists basic Capability Types shipped with BPO.

FQN	CAPABILITY PURPOSE	REQUIREMENT PURPOSE
<code>tosca.capabilityTypes.Feature</code>	Recommended base type for all Capability Types	Same as Capability Purpose
<code>tosca.capabilityTypes.Endpoint</code>	Indicates a Resource's ability to be the target of a <code>tosca.relationshipTypes.ConnectionsTo</code> Relationship	Indicates a Resource's need to be connected to another Resource as the source of a <code>tosca.relationshipTypes.ConnectionsTo</code> Relationship
<code>tosca.capabilityTypes.Container</code>	Indicates a Resource Instance's ability to contain other Resource Instances	Indicates a Resource Instance's need for a containing Resource Instance

Structure

Capability Types are another type of artifact which may be included in [Definition Modules](#). As such, they can be treated as JSON objects. The schema of Capability Type definitions can be found in the `json-schemas/tosca-lite-v1/capability-type.hocon` file in the Model Definitions area.

The following table describes the attributes of Capability Type definitions.

ATTRIBUTE	TYPE	REQUIRED	DEFAULT	PURPOSE
<code>title</code>	string	no	None	Used for display purposes in UIs.
<code>description</code>	string	yes	N/A	Used for help text and other places where detailed information is displayed on UIs.
<code>derivedFrom</code>	string	no	None	Specifies one or more Capability Types from which this Capability Type inherits from.
<code>properties</code>	object	no	{}	Defines properties which may be specified in Capabilities and Requirements on Resource Types.

Inheritance of properties on Capability Types is dealt similarly to properties on [Resource Types](#). If

the derived type defines any properties which also exist on the parent, the parent's version is dropped in favor of the derived type's version. This is done on a per property basis, so properties which exist on the derived type will be merged with the parent's properties, preferring the derived type's version.

Relationship Types

A Relationship Type defines characteristics of Relationship Instances (just "Relationships") that are specific to the type. Relationship Types are defined inside [Definition Modules](#).

Important characteristics of Relationships found in the Relationship Type definition include:

1. The schema of the Relationship's `properties` attribute.
2. The valid type of Capability and Requirements on the source and target Resource Instances.

BPO ships a few basic Relationship Types which can be used directly or through inheritance to represent more domain specific types of Relationships.

The table below lists basic Relationship Types shipped with BPO.

FQN	COMPATIBLE CAPABILITY TYPE	PURPOSE
<code>tosca.relationshipTypes.DependsOn</code>	<code>tosca.capabilityTypes.Feature</code>	Recommended base type for all Relationship Types.
<code>tosca.relationshipTypes.ConnectsTo</code>	<code>tosca.capabilityTypes.Endpoint</code>	Express a connection between Resources.
<code>tosca.relationshipTypes.HostedOn</code>	<code>tosca.capabilityTypes.Container</code>	Express hosting feature between Resources.
<code>tosca.relationshipTypes.MadeOf</code>	<code>tosca.capabilityTypes.Container</code>	Express composition of sub-resources. This is the default Relationship Type of Relationships created by the Template Engine.

Structure

The Relationship Type definition schema is defined in the `json-schemas/tosca-lite-v1/relationship-type.hocon` file in the Model Definitions area.

The following table describes the attributes of Relationship Type definitions:

ATTRIBUTE	TYPE	REQUIRED	DEFAULT	PURPOSE
<code>title</code>	<code>string</code>	<code>yes</code>	<code>N/A</code>	Used for display purposes in UIs.
<code>description</code>	<code>string</code>	<code>yes</code>	<code>N/A</code>	Used for help text and other places where detailed information is displayed on UIs.
<code>derivedFrom</code>	<code>string or [string]</code>	<code>no</code>	<code>N/A</code>	Specifies one or more Relationship Types to derive from.
<code>properties</code>	<code>object</code>	<code>no</code>	<code>{}</code>	Specifies the schema of the <code>properties</code> attribute of Relationships.

ATTRIBUTE	TYPE	REQUIRED	DEFAULT	PURPOSE
validSource	string	no	None	Specifies valid type of the Requirement on the source Resource.
validTarget	string	no	None	Specifies the valid type of the Capability on the target Resource.

Property Definitions

Property definitions specify the schema and access modifiers of Relationships "properties". The property definition support for Relationships is the same as for Resources. For more details see [Property Definitions](#).

Inheritance

Inheritance of properties on Relationship Types is the same as [inheritance of properties on Resource Types](#). If the derived type defines any properties with the same as the parent, the parent's version is dropped in favor of the derived type's version.

If both parent and derived Relationship Types specify validSource then the derived version must be derived from the parent's version. The same rule applies to the validTarget attribute.

`validSource` and `validTarget`

The `validSource` and `validTarget` attributes restrict the valid type Capability Type of the Requirement or Capability on the source or target Resource (respectively).

These attributes are used to check the type compatibility of the (relationship, source, target) triplet in requests which create or update Relationships.

By specifying the valid Capability Types, BPO is able to validate these requests without forcing Relationship Types to be coupled to specific Resource Types. This allows definition authors to create a meaningful set of base Relationship Types to be reused across many different types of (source, target) pairs. For more information on Relationship Validation, see [Relationship Instances](#).

Onboarding

At a very high level, the entire onboarding process involves:

1. Modifying a local copy of the production branch of an Assets area.
2. Committing the changes to a private branch and pushing the branch back to BPO's repository.
3. Submitting a Pull Request (PR) to incorporate the changes into production.
4. Waiting for the PR to be accepted or rejected.

In between steps 3 and 4, BPO processes the PR to determine if it should be accepted or rejected. This process is managed by the [Asset Manager](#) component (just "Assets" for short). This process is essentially a 2-phase commit protocol (2PC).

When a PR is submitted, Assets may immediately begin processing the PR. When PR processing begins, Assets creates a temporary copy of the area based on (1) the current production version of the area and (2) the changes proposed in the PR. At this point, the 2PC protocol is started.

The first phase of the protocol is referred to as "validation". All BPO components which registered interest in the area are notified. The notification includes a reference to the temporary copy of the area. The components are expected to perform validation on this copy and reply to indicate whether they accept or reject the changes. An error in the protocol (e.g, a timeout between Assets and another component) is treated as a rejection.

All components must accept the changes in order for the PR to be accepted.

If all components accept the changes, the PR is "accepted" and the components are notified. This is referred to as the "commit" phase. The notification indicates that new changes have been committed and the production copy of the area has been updated. Components use this notification to trigger updates to their state based on the area. They also release any temporary resources allocated during the validation phase.

If any components reject the changes, the PR is "rejected" and the components are notified. This is referred to as the "abort" phase. The notification indicates that the changes HAVE NOT been committed and the production copy remains unchanged. Components use this notification to release any temporary resources allocated during the validation phase.

When a rejection occurs, the PR entity is updated to reflect the reason for the rejection. The reason depends on the types of changes and the current state of the system. E.g., the changes may introduce invalid Definition Modules or alter them in a way which invalidates invariants defined by BPO.

There are two important points about PR processing:

1. Components validate the **entire** temporary copy of the area. If any part is invalid the changes are rejected.
2. Assets only validates a single PR at a time. This prevents conflicting PRs from being submitted in parallel and accepted (i.e, because they are both valid against the current production version of the

area).

3. The "commit" phase (when a PR is accepted) is asynchronous. Assets does not wait for a response from the components before setting the PR's status to "accepted". As a result, if you need to sequence operations **after** onboarding, you must use APIs exposed by those components to determine when they have finished processing the "commit" notification.

Model Definitions

The Model Definitions area stores [Definition Modules](#) used throughout BPO. These files includes artifacts which define schema and behavior inside BPO.

Market is an important consumer of the Model Definitions area. To determine when Market has completed it's "commit" phase, use the `GET /type-artifacts/realm` API to obtain the commit ID of the Model Definitions area which Market has incorporated:

```
market> type-artifacts realm get
+-----+
| Attribute | Value
+-----+
| version   | "10abdc78e0d649125a327081bec10c7547f2f67c" |
+-----+
```

When Market has incorporated the latest version of the Model Definition area, the `version` attribute of the `realm` object should be the same as the `commitHash` attribute of the area:

```
assets> areas model-definitions get
+-----+
| Attribute | Value
+-----+
| area      | "model-definitions"
| commitHash | "10abdc78e0d649125a327081bec10c7547f2f67c"
| releaseSignature | "10d10eb62603246083d6ecb32008ce628dbd04ad;Release Unknown"
| upgradeAvailable | false
| uri       | "ssh://git@192.168.99.100:2222/~/repos/model-definitions"
| versionSignature | "10d10eb62603246083d6ecb32008ce628dbd04ad;Release Unknown"
+-----+
```

Definition Module Changes

When definitions are changed (i.e, modified or removed), BPO's PR processing validates the changes against the current state of the system. This process is similar to when definitions added; however, there are extra constraints which are enforced to maintain consistency.

The following sub-sections describe limitations when changing Definition Modules.

Resource Types

The supported modifications are described below with example values and any additional limitations:

ATTRIBUTE	FROM	TO	LIMITATIONS
title	"foo"	"bar"	None
description	"foo"	"bar"	None
final	true	false	None
final	false	true	There are no derived Resource Types in the Type Catalog.
abstract	true	false	None
abstract	false	true	There are no Service Templates implementing the Resource Type in the Type Catalog or Products (of the Resource Type) in the Product Catalog.

There are two general cases in which type modifications can be made:

1. When there are zero Products of the given Resource Type in the Product Catalog:
 - Any changes can be made, including the addition, removal, and modification of properties, as well as changing of `derivedFrom` inheritance
2. When there are Products of the given Resource Type (or Products for `derivedFrom` Resource Types) in the Product Catalog:
 - Only the addition and modification of properties is allowed. Furthermore, when adding properties, they must be purely additive by being either `optional = true`, or having a default value set, or both. This includes adding an inheritance rule via `derivedFrom`; The newly added Parent Resource Type's properties must also be purely additive
 - Properties can be moved from a parent to child (or any combination of) as long as the property continues to be defined somewhere for the types which use it. See [Changing Property Inheritance Levels](#) for details
 - Modified properties must continue to meet any [Product Constraints](#) that have been defined for Products using this type
 - Property type changes must adhere to the table rules as specified in [Changing Property Types](#)

Changing Properties

When there are existing Products (of the Resource Type) in the Instance Catalog, properties cannot be removed, however new properties can be onboarded, while some modifications to existing properties are also allowed.

If there are existing Products with constraints defined on them, the changes must also not conflict with the constraints.

Furthermore, new properties must be set to `optional = true` or include a default value. This ensures the changes are backwards compatible with existing Products and Resources in the system.

If a property's `type`, `fulltext`, or `history` flag is modified, or a new property is added, the Resources using this type will be lazily migrated when accessed. This means a change to `type`, `fulltext`, `history`, new default value, or non-top level property type change will be applied when the Resource is retrieved. Additionally, any differences which reflected the old property types will also be migrated. The transaction which makes these changes will include a `migration` flag which is visible when querying the Resource's history. If you want to force the migration process, you can list all Resources of the specific Resource Type after the onboarding has completed. When forcing migration in this way, you must iterate through every page in the `GET /bpocore/market/api/v1/resources?resourceTypeId=<type>` API result list in order to migrate every resource of a given type. Using a `limit` of 0 will have no effect. See [Pagination](#) for more information about pagination in bpocore.

If there are no existing Products (of the Resource Type) in the system, then properties may be modified or removed without additional constraints.

Changing Property Types

Depending on the existing type, a property may be able to change types while Products and Resources exist in the system. The following table illustrates the valid changable types and the expected behavior if such a type change is onboarded:

Original Type	New Type	Operation
<code>string</code>	<code>boolean</code>	If "false" or "0" then false else true
<code>string</code>	<code>array</code>	The original value will be inserted into the new array
<code>integer</code>	<code>string</code>	Converts integer to string representation
<code>integer</code>	<code>number</code>	Converts integer to floating point representation
<code>integer</code>	<code>boolean</code>	If 0 then false else true
<code>integer</code>	<code>array</code>	The original value will be inserted into the new array
<code>number</code>	<code>string</code>	Converts floating point to string representation
<code>number</code>	<code>integer</code>	Truncates floating point to an integer
<code>number</code>	<code>boolean</code>	If 0.0 false else true
<code>number</code>	<code>array</code>	The original value will be inserted into the new array
<code>boolean</code>	<code>string</code>	false becomes "false" and true becomes "true"
<code>boolean</code>	<code>integer</code>	If false then 0 else 1
<code>boolean</code>	<code>number</code>	If false then 0.0 else 1.0

boolean	array	The original value will be inserted into the new array
object	string	The original object value will be converted to a compact JSON string
object	array	The original object value will be inserted into the new array
array	string	The original array value will be converted to a compact JSON string

Conversion to an array can also involve a conversion of the type as well. Any conversion which is valid from some type $T \rightarrow U$ is also valid for $T \rightarrow \text{Array}[U]$.

For conversions to an array of objects, the schema for the array objects should conform to the original object, E.x.:

```

properties {
  myObject {
    title = "myObject"
    description = "An object property"
    type = object
    config = true
    updatable = true
    properties {
      mySubProperty {
        title = "mySubProperty"
        description = "A property inside an object"
        type = string
        config = true
        updatable = true
      }
    }
  }
}
  
```

Would become:

```

properties {
  myObject {
    title = "myObject"
    description = "Is now an array!"
    type = array
    config = true
    updatable = true
    items {
      type = object
      properties {
        mySubProperty {
          title = "mySubProperty"
          description = "A property inside an object"
          type = string
          config = true
          updatable = true
        }
      }
    }
  }
}

```

Changing Property Inheritance Levels

BPO allows [Resource Types](#) to inherit properties from other resource types via the `derivedFrom` keyword. If a Resource Type inherits a property from some other type, the inheritance level can be changed in several ways. Given the following example:

Resource Type A inherits a property P from Resource Type B. A's dependency on B can be entirely removed by implementing property P directly on Resource Type A, while removing the `derivedFrom` statement and onboarding the change. The updated implementation of P must be compatible with the original inherited implementation of P and any normally allowed changes (including changing types) can be performed.

More concretely, a change removing the dependency would look as follows:

```

resourceTypes {
  ParentTypeB {
    properties {
      propP {
        type = string
      }
    }
  }
  ChildTypeA {
    derivedFrom = "ParentTypeB"
    properties {
    }
  }
}

```

Would become:

```
resourceTypes {
    ChildTypeA {
        properties {
            propP {
                type = string
            }
        }
    }
}
```

Inversely, the dependency tree could be expanded, as long as some compatible property P exists in the final result. In the next example, A depended on B, and is changed to now depend on C which depends on B.

```
resourceTypes {
    ParentTypeB {
        properties {
            propP {
                type = string
            }
        }
    }
    ChildTypeA {
        derivedFrom = "ParentTypeB"
        properties {
        }
    }
}
```

Would become:

```
resourceTypes {
    ParentTypeB {
        properties {
            propP {
                type = string
            }
        }
    }
    ParentTypeC {
        derivedFrom = "ParentTypeB"
        properties {
        }
    }
    ChildTypeA {
        derivedFrom = "ParentTypeC"
        properties {
            propP {
                type = string
            }
        }
    }
}
```

More radical changes can also be performed, including keeping `derivedFrom` while moving the property from the parent to child types (provided the parent has no Products), or moving it to a new type in between both types, or any other combination. The key is that a property P must be defined *somewhere* (at any level of inheritance, or simply on A itself) for A and that it must be compatible with the previous version of P (regardless of where it was inherited from).

Resource Type Removal

Resource Types may only be removed from the system if they are not in-use. Resource Types are said to be in-use if any of the following are true:

- There are existing derived Resource Types
- There are existing Service Templates in the Type Catalog which implement the Resource Type
- There are existing Products in the Product Catalog created from the Resource Type

When a Resource Type is removed, the history of the Resource and Instance Catalog will be unaffected.

Property Definitions

The supported modifications are described below with example values and any additional constraints:

ATTRIBUTE	FROM	TO	CONSTRAINTS
<code>title</code>	<code>"foo"</code>	<code>"bar"</code>	<code>None</code>
<code>description</code>	<code>"foo"</code>	<code>"bar"</code>	<code>None</code>
<code>output</code>	<code>true</code>	<code>false</code>	<code>None</code>
<code>output</code>	<code>false or unset</code>	<code>true</code>	<code>None</code>
<code>config</code>	<code>true or unset</code>	<code>false</code>	<code>None</code>
<code>config</code>	<code>false</code>	<code>true</code>	<code>None</code>
<code>updatable</code>	<code>true</code>	<code>false</code>	<code>None</code>
<code>updatable</code>	<code>false or unset</code>	<code>true</code>	<code>None</code>
<code>obfuscate</code>	<code>true</code>	<code>false</code>	<code>None</code>
<code>obfuscate</code>	<code>false or unset</code>	<code>true</code>	<code>None</code>
<code>optional</code>	<code>false or unset</code>	<code>true</code>	<code>Only supported as of bpocore version 1.3.0.</code>

These modifications can be applied to definitions inside Resource Types or reusable Property Type definitions.

Example: Attempt to remove a Resource Type while there are existing Products

```
assets> areas model-definitions pullrequests 9dc79f5f-6421-4380-a69b-1b60ba484877
get
+-----+
+-----+
-----+ | Attribute | Value
| |
+-----+
-----+ | comment | null
| |
| commitDate | "2015-11-21T00:22:32.000Z"
| |
| commitHash | "1b7c0a47e5fa720e4acfaf64be5c5593f6d84aff"
| |
| details | [
| |
| | {
| |
| | | "component": "market-type-manager",
| |
| | | "description": "Model conflicts with catalogs.",
| |
| | | "issues": [
| |
| | | {
| |
| | | | "message": "'example.resourceTypes.ExampleService' cannot be deleted because it is referred to by existing products.",
| |
| | | | "extra": [
| |
| | | | | "564fb91f-8ac8-43a5-a60e-da472ce06ef9"
| |
| | | | ]
| |
| | | }
| |
| | ]
| |
| | }
| |
| ]
email | "tsandall@ciena.com"
productionCommitHash | null
reason | "Validation failed against one or more components"
requestDate | "2015-11-21T00:22:33.061Z"
requestId | "9dc79f5f-6421-4380-a69b-1b60ba484877"
```

```
| status           | "rejected"
| title          | "Auto-generated pull request from \"onboard\""
|
+-----+
+-----+
-----+
```

Example: Attempt to add a required property to a Resource Type with existing Resources

```
assets> areas model-definitions pullrequests 542c4f2a-44d0-433a-8448-0bdba9306fdf
get
+-----+
+-----+
-----+
| Attribute | Value
|
+-----+
+-----+
-----+
| comment | null
| commitDate | "2015-11-21T00:26:19.000Z"
| commitHash | "43c94839435e7680eed475e6f27aa26cf33a3090"
| details | [
|   {
|     "component": "market-type-manager",
|     "description": "Model conflicts with catalogs.",
|     "issues": [
|       {
|         "message": "'example.resourceTypes.ExampleService' changes conflict with existing resources | because they are not backwards compatible.",
|         "extra": [
|           {
|             "path": "/properties/exampleProperty",
|             "value": {
|               "description": "A new required property which is non-backwards compatible",
|               "title": "Example Property",
|               "type": "string",
|             }
|           }
|         ]
|       }
|     ]
|   }
| 
```

Onboarding Failures

If the changes are invalid, the PR will be rejected. Callers can determine if the PR was accepted by polling the PR and checking the `status` attribute. If the `status` goes to `rejected` then the PR processing has stopped (because the changes are invalid) and the `reason` is set to indicate the cause.

Product Catalog

The Product Catalog is introduced in [Main Concepts](#).

Product Instances

Product Constraints

Products can apply constraints to the property definitions of the associated Resource Type. The constraints for a product are represented by the `constraints` attribute on the Product.

Constraints allow providers to customize the offerings of a particular Resource Type.

For example, a provider may decide to offer a network service. Initially, the network service will have three levels of quality (e.g, "bronze", "silver", and "gold") which map to a larger bandwidth allocation (respectively). To achieve this, the provider would create three Products against the network service type. Each product would have a constraint which restricts the bandwidth allocation to a specific value.

Currently, constraints are specified when the Product is created and are not mutable.

Constant Constraints

Constant constraints restrict property values to a specific value and remove the need to specify the property when instantiating Resources from the corresponding Product. Constant constraints effectively give properties a default value and an `enum` attribute containing a single item.

Format:

```
{ "constant": <value> }
```

Default Constraints

Default constraints provide default values for properties of Resources instantiated from the corresponding Product. Default constraints differ from [Constant Constraints](#) by allowing users to specify a different value for the property when instantiating or modifying the Resource.

Format:

```
{ "default": <value> }
```

Enumeration Constraints

Enumeration constraints restrict the value of a property to a collection of items. If the property definition

already contains an `enum` value, then the enumeration constraint may only **remove** items from the existing `enum` collection.

Format:

```
{ "enum": [ <value_1>, <value_2>, ..., <value_n> ] }
```

Example

1. Define an onboard a new Resource Type, e.g:

```
"$schema"      = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-module#"
title          = "Network Service"
package        = example
version        = "1.0"
description    = "This document defines a dummy Network Service resource type."
authors        = [ "Torin Sandall <tsandall@ciena.com>" ]
resourceTypes {
    NetworkService {
        title = Network Service
        description = ""
        derivedFrom = tosca.resourceTypes.Root
        properties {
            allocatedBandwidth {
                type = integer
                title = "Allocated Bandwidth (Mbps)"
                description = "The allocated bandwidth for the service in Mbps."
            }
        }
    }
}
```

2. Create two Products with different constraints:

```
market> products post {title: Silver, resourceTypeId:  
example.resourceTypes.NetworkService, domainId: built-in, constraints:  
{allocatedBandwidth: {constant: 1000}}}  
+-----  
+-----+  
| Attribute | Value  
|  
+-----+  
+-----+  
| active | true  
|  
| constraints.allocatedBandwidth.constant | 1000  
|  
| description | null  
|  
| domainId | "built-in"  
|  
| id | "56410209-1eca-4c2c-b100-  
01a256f808ee" |  
| providerProductId | null  
|  
| resourceTypeId |  
"example.resourceTypes.NetworkService" |  
| title | "Silver"  
|  
+-----  
+-----+  
market> products post {title: Gold, resourceTypeId:  
example.resourceTypes.NetworkService, domainId: built-in, constraints:  
{allocatedBandwidth: {constant: 10000}}}  
+-----  
+-----+  
| Attribute | Value  
|  
+-----+  
+-----+  
| active | true  
|  
| constraints.allocatedBandwidth.constant | 10000  
|  
| description | null  
|  
| domainId | "built-in"  
|  
| id | "56410211-3792-46f3-b6a1-  
4935dc0a59f6" |  
| providerProductId | null  
|  
| resourceTypeId |  
"example.resourceTypes.NetworkService" |  
| title | "Gold"  
|  
+-----  
+-----+
```

3. Instantiate a new Resource from the "Gold" Product

```
market> resources post {productId: 56410211-3792-46f3-b6a1-4935dc0a59f6}
+-----+-----+
| Attribute          | Value   |
+-----+-----+
| autoClean          | false   |
| createdAt          | "2015-11-09T20:30:00.867Z" |
| description         | null    |
| desiredOrchState   | "active"|
| differences         | []      |
| discovered          | false   |
| id                  | "56410248-490a-4952-a00d-f4801ce61b87" |
| label               | null    |
| nativeState         | null    |
| orchState           | "requested" |
| orderId              | null    |
| productId            | "56410211-3792-46f3-b6a1-4935dc0a59f6" |
| properties.allocatedBandwidth | 10000  |
| providerResourceId | null    |
| reason              | ""     |
| resourceTypeId       | "example.resourceTypes.NetworkService" |
| shared               | false   |
| tenantId             | "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738" |
| updatedAt            | "2015-11-09T20:30:00.867Z" |
+-----+-----+
```

4. Try to instantiate a new Resource from the "Silver" Product with a different value for allocatedBandwidth. This will be rejected.

```
market> resources post {productId: 56410209-1eca-4c2c-b100-01a256f808ee,
properties: {allocatedBandwidth: 10000}}
validation failed: error: instance value (10000) not found in enum (possible
values: [1000])
  level: "error"
  schema: {"loadingURI": "http://cyaninc.com/products/56410209-1eca-4c2c-b100-
01a256f808ee#", "pointer": "/properties/allocatedBandwidth"}
  instance: {"pointer": "/allocatedBandwidth"}
  domain: "validation"
  keyword: "enum"
  value: 10000
  enum: [1000]
```

Instance Catalog

The Instance Catalog is introduced in [Main Concepts](#). The Instance Catalog consists of two kinds of instances:

- [Resource Instances](#) - represent (physical or logical) entities in managed Domains.
- [Relationship Instances](#) - represent relationships between Resource Instances.

The Instance Catalog exposes APIs to query and manipulate Resource Instances and Relationship Instances.

Resource Instances

Resource Instances (referred to as "Resources" for short) are a central component in BPO's model. In addition to services offered through BPO, entities in external systems managed by BPO are typically represented as Resources. This allows BPO to expose a common interface for orchestration of services and external systems. For example, BPO represents all of the following as Resources:

- Network Interfaces, Firewalls, Virtual Routers, NAT
- Devices, Cross Connects, Termination Points
- Network Services
- Containers, Virtual Machines, Hypervisors, Data Centers
- Machine Images, Load Balancers, Databases

Since Resources are used to model entities from different domains, all type specific data is stored in the Resource's `properties` attribute. The schema of the data stored in the `properties` attribute is defined on the Resource Type associated with the Resource.

Given the OO analogies in the [Resource Types](#) and [Service Templates](#) references, Resources are conceptually similar to objects created at runtime in C++ or Java programs.

Below, we often refer to **users** and **providers** of Resources. In this context, users are not necessarily the operators or administrators of BPO, users are just entities which create and manage Resources through BPO's APIs. On the other hand, providers are (almost always) Resource Adapters or BPO's Template Engine component. The duality between users and providers is important. Users specify the **desired** state of Resources. Providers attempt to realize the desired state and also report the **observed** state of Resources.

Top-level Attributes

All Resources have several common, top-level attributes. Updates to these attributes are differentiated by the API calls employed:

- `Updatable` refers to the PUT and PATCH /market/api/v1/resources/{resourceId} API calls.

- Observed Updatable refers to the PUT and PATCH /market/api/v1/resources/{resourceId}/observed API calls.

The following table describes these attributes:

NAME	TYPE	FORMAT	OPTIONAL	UPDATABLE	OBSERVED UPDATABLE	DESCRIPTION
autoClean	boolean	n/a	false	false	false	Specified when the Resource is created. Indicate BPO should attempt to rollback the activation process if it fails.
createdAt	string	date-time	false	n/a	n/a	Specifies when the Resource was created.
description	string	n/a	true	true	false	Used for help text and other places where detailed information is displayed on UIs.
desiredOrchState	string	enum	false	true	false	Identifies the desired orchestration state of the Resource. See Lifecycle Operations for more details.
differences	array	n/a	false	n/a	n/a	Describes the differences between desired and observed state of configurable properties. See Differences for more details.
discovered	boolean	n/a	false	true	false	Indicates whether this Resource is discovered or managed. See Discovered Resources for more details.
id	string	uuid	false	false	false	Uniquely identifies the Resource within the catalog. This value is generated by Market when the resource is created.
label	string	n/a	true	true	false	Used for display purposes in UIs.
orchState	string	enum	false	true	true	Identifies the observed orchestration state of the Resource. See Lifecycle Operations for more details.
productId	string	uuid	false	false	false	Identifies the Product which the Resource is associated with. See Product Catalog reference for more details on Products.
properties	object	n/a	false	true	true	Contains Resource Type specific data. See Properties for more details.
providerData	object	n/a	true	true	true	Contains provider specific data for the Resource. This value is optional and is controlled by the provider of the Resource. Users should not rely on the values in this attribute.

NAME	TYPE	FORMAT	OPTIONAL	UPDATABLE	OBSERVED UPDATABLE	DESCRIPTION
providerResource Id	string	n/a	true	false	true	Uniquely identifies the Resource within the Domain. This value is optional and is controlled by the provider of the resource.
reason	string	n/a	false	true	true	Used for display purposes in UIs. Contains additional information in case orchState moves to failed.
resourceType Id	string	uuid	false	false	false	Identifies the Resource type which the Resource is associated with. See the Resource Types reference for more details.
shared	boolean	n/a	false	true	false	Indicates whether this Resource is visible to sub-tenants.
tags	object	n/a	false	true	true	User controlled metadata associated with Resource.
tenantId	string	uuid	false	false	false	Identifies the tenant which the Resource is associated with.
updatedAt	string	date-time	false	n/a	n/a	Specifies when the Resource was last updated.

Note: Unobserved updating orchState and providerData exist solely for backwards compatibility and should be considered deprecated pending future removal.

Discovered Resources

Resources are either **managed** or **discovered**. Discovered Resources represent existing entities found in external systems managed by BPO. Discovered Resources differ from Managed Resources in a few important ways:

- Their discovered flag will be set to true.
- Their desiredOrchState is always set to unspecified. Because of this, if their orchState is set to terminated, they are removed from the Instance Catalog assuming there are no incoming relationships from non-terminated Resources.
- Their differences will always be empty.

Users can add Discovered Resources in the Instance Catalog manually, however, they are most often added by Resource Adapters as part of discovery.

Properties

Resources store all type specific data in the properties attribute. The schema of the data stored in the

`properties` attribute is defined on the Resource Type associated with the Resource.

The schema language for properties extends JSON Schema. All of the standard JSON Schema metadata can be specified in addition to BPO specific **Access Modifiers**. Access Modifiers control how properties may be read and written. For example:

- some properties may be updated while others cannot.
- some properties may only be set by the provider.

Users and other consumers of Resource data should assume that values in the `properties` conform to the schema defined on the Resource Type.

For more details on property schema, see the [Resource Types](#) reference.

Configurable Properties

Properties which the user can supply as input (when creating the Resource) are referred to as **configurable properties**. By default, properties are configurable but this can be overridden by using the `config` Access Modifier in the property definition.

Conceptually, configurable properties have both a desired and observed value. This duality exists because the value specified (desired) by the user is not necessarily what is realized (observed) in the managed Domain. For Managed Resources, the `properties` attribute always includes the desired value. The observed value is encoded in the `differences` attribute (see [Differences](#) for more details). For discovered resources, there is no distinction between desired and observed values so the value is simply stored in the `properties` attribute.

Example

The following Resource Type definition models a simple Virtual Machine type:

```
"$schema" = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-module#"
title = "Example Virtual Machine"
description = """
An example definition containing a Virtual Machine resource type.
"""

version = "1.0"
package = example
authors = [
    "John Smith <john.smith@example.com>"
]

resourceTypes {
    AcmeVirtualMachine {
        title = Virtual Machine
        description = "An simplistic Virtual Machine model."
        derivedFrom = tosca.resourceTypes.Root
        properties {
            cpuCount {
                type = integer
                minimum = 1
                title = "CPU Count"
                description = "Number of CPUs to allocate to this VM."
            }
            memorySize {
                type = integer
                minimum = 128
                units = "MiB"
                title = "Memory Size"
                description = "Amount of memory to allocate to this VM."
            }
            ipv4Address {
                type = string
                format = "ipv4-address"
                output = true
                title = "IPv4 Address"
                description = "IPv4 address assigned to this VM."
            }
        }
    }
}
```

The following resource represents an instance of the Virtual Machine type defined above:

```
market> resources 56b0f198-0186-4fe2-9d49-ef275a8475c0 get
+-----+-----+
| Attribute | Value |
+-----+-----+
| autoClean | false |
| createdAt | "2016-02-02T18:12:40.798Z" |
| description | null |
| desiredOrchState | "active" |
| differences | [] |
| discovered | false |
| id | "56b0f198-0186-4fe2-9d49-ef275a8475c0" |
| label | null |
| nativeState | null |
| orchState | "active" |
| orderId | null |
| productId | "56b0ee9c-727a-44d6-9a75-417325addc2e" |
| properties.cpuCount | 2 |
| properties.ipv4Address | "10.98.1.62" |
| properties.memorySize | 128 |
| providerResourceId | null |
| reason | "" |
| resourceTypeId | "example.resourceTypes.AcmeVirtualMachine" |
| shared | false |
| tenantId | "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738" |
| updatedAt | "2016-02-02T18:12:44.110Z" |
+-----+-----+
```

Differences

The `differences` attribute on a Resource encodes the difference between the desired and observed state of the Resource. The differences are encoded as a sequence of [RFC 6902 JSON Patch](#) operations on the Resource. Callers can obtain the current observed version of the Resource by applying the JSON Patch operations from the `differences` attribute.

Differences are only set for configurable properties. For example, properties which enable the `output` Access Modifier or are explicitly not configurable (`config` Access Modifier set to `false`) will never have differences.

Differences are computed by the backend when processing update requests (e.g, PUT or PATCH). Updates may either be performed against the desired or observed state of the Resource. The former is used when the user of a Resource performs a configuration change and the latter is used when the provider of a Resource is reflecting the current state of the managed entity.

This table shows how updates to desired values affect the properties and differences:

NEW DESIRED	OLD DESIRED	CURRENT OBSERVED	PROPERTY VALUE	DIFFERENCE
X	Y	Y	X	Replace(Y)
X	Y	Z	X	Replace(Z)

NEW DESIRED	OLD DESIRED	CURRENT OBSERVED	PROPERTY VALUE	DIFFERENCE
X	W	Y	X	Replace(Y)
X	Y	None	X	Remove()
None	X	Y	None	Add(Y)
X	Y	X	X	None

This table shows how updates to observed values affect the properties and differences:

NEW OBSERVED	OLD OBSERVED	PROPERTY VALUE	DIFFERENCE
X	Y	Y	Replace(X)
None	Y	X	Remove()
X	Y	None	Add(X)
X	Y	X	None

In both of these tables, None is used to represent the lack of a value or unsetting of a value in the case of New Desired and New Observed.

Example

1. Create a new Resource and wait for it to activate

```
market> resources post {productId: 56146326-3f3c-405e-9297-47ec1728cbd3,
label:
difference_example, properties: {numCpus: 8, memSize: 4096, diskSize: 128}}
+-----+-----+
| Attribute | Value |
+-----+-----+
| autoClean | false |
| createdAt | "2015-10-07T00:13:23.293Z" |
| description | null |
| desiredOrchState | "active" |
| differences | [] |
| discovered | false |
| id | "561463a3-2525-4d7b-8142-0877799b5482" |
| label | "difference_example" |
| nativeState | null |
| orchState | "requested" |
| orderId | null |
| productId | "56146326-3f3c-405e-9297-47ec1728cbd3" |
| properties.diskSize | 128 |
| properties.memSize | 4096 |
| properties.numCpus | 8 |
| providerResourceId | null |
| reason | "" |
| resourceTypeId | "example.resourceTypes.ExampleService" |
| shared | false |
| tenantId | "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738" |
| updatedAt | "2015-10-07T00:13:23.293Z" |
+-----+-----+
```

2. Perform a configuration change on the "difference_example" Resource

```
market> resources difference_example patch {properties: {numCpus: 16,
memSize: 8192}}
+-----+
+-----+
-----+ | Attribute | Value
| +-----+
+-----+
-----+ | autoClean | false
| | createdAt | "2015-10-07T00:13:23.293Z"
| | description | null
| | desiredOrchState | "active"
| | differences | [{"path": "/properties/memSize", "value": 4096,
"op": "replace"}, {"path": "/properties/numCpus", |
| | "value": 8, "op": "replace"}]
| | discovered | false
| | id | "561463a3-2525-4d7b-8142-0877799b5482"
| | label | "difference_example"
| | nativeState | null
| | orchState | "active"
| | orderId | null
| | productId | "56146326-3f3c-405e-9297-47ec1728cbd3"
| | properties.diskSize | 128
| | properties.memSize | 8192
| | properties.numCpus | 16
| | providerResourceId | null
| | reason | ""
| | resourceTypeId | "example.resourceTypes.ExampleService"
| | shared | false
| | tenantId | "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738"
| | updatedAt | "2015-10-07T00:14:24.600Z"
| +-----+
+-----+
-----+
```

3. Wait for the configuration change to propagate

```
market> resources difference_example get
+-----+-----+
| Attribute | Value |
+-----+-----+
| autoClean | false |
| createdAt | "2015-10-07T00:13:23.293Z" |
| description | null |
| desiredOrchState | "active" |
| differences | [] |
| discovered | false |
| id | "561463a3-2525-4d7b-8142-0877799b5482" |
| label | "difference_example" |
| nativeState | null |
| orchState | "active" |
| orderId | null |
| productId | "56146326-3f3c-405e-9297-47ec1728cbd3" |
| properties.diskSize | 128 |
| properties.memSize | 8192 |
| properties.numCpus | 16 |
| providerResourceId | null |
| reason | "" |
| resourceTypeId | "example.resourceTypes.ExampleService" |
| shared | false |
| tenantId | "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738" |
| updatedAt | "2015-10-07T00:14:24.600Z" |
+-----+-----+
```

Resource Update Rules

This section describes the rules for how updates are applied to Resource data stored in the Instance Catalog.

Resource updates are supported via `PUT` and `PATCH` methods. For example, via the CLI, you can modify a Resource's properties:

```
market> resources 56b39b0a-f77e-4a39-bde7-0a68320b695d patch {properties: {prop1: bar}}
```

Most top-level attributes can be updated via similar requests. See [Top-level Attributes](#) for their restrictions.

The access modifiers, as described in [Property Definitions](#), determine whether a property is updatable or observed updatable as well as whether the property can be removed (i.e. is retractable). The rules determining whether such a property is updatable are shown in the following table where the left side represents properties attributes' values and the right side represents the expected behavior:

CONFIG	OUTPUT	UPDATABLE	OPTIONAL		UPDATABLE	RETRACTABLE	OBSERVE D UPDATABLE	OBSERVE D RETRACTABLE
false	false	n/a	n/a		no	no	yes	yes
false	true	n/a	n/a		no	no	no*	no
true	n/a	false	false		no	no	yes**	no
true	n/a	false	true		no	no	yes**	yes**
true	n/a	true	false		yes	no	yes	no
true	n/a	true	true		yes	yes	yes	yes

*Output only properties can only be set once and are then read-only.

** The user will not be able to clear the differences directly. This must be done by the observed API in order to match the user's expectation of the value.

For related information on updating resources, see:

NOTE

- [Resource Updates](#) reference on methods to update resources
- [Lifecycle Operations](#) reference on update as a lifecycle operation

Resource History

The Instance Catalog exposes the history of Resources via two types of queries:

- Queries for the history of the Instance Catalog itself. This is the record of all Resources being added and removed.
- Queries for the history of individual Resources. This is the record of Resource attributes being set, updated, and unset.

The history of the Instance Catalog and individual Resources is useful for auditing as well as root cause analysis of failures in the orchestration process.

For example, via the CLI, you can query for the history of the Instance Catalog. The output below shows (among other information) two resources being instantiated at approximately 2016-02-04T18:40:10 and then terminated approximately 11 minutes later:

```
market> resources history get
+-----+-----+
+-----+-----+-----+
| time | id | label |
| userId | added | |
+-----+-----+-----+
| "2016-02-04T18:37:16.419Z" | "56b39a5c-c4e8-4059-af68-bc78554a3eee" | "IT-ESO-KEY"
| "admin" | true |
| "2016-02-04T18:37:17.051Z" | "56b39a5c-c024-408f-8b7c-19ad391497a3" | "default"
| "admin" | true |
| "2016-02-04T18:37:17.075Z" | "56b39a5c-dafe-491a-b81d-110a6ecd94ac" | "vpc-e6b38283"
| "admin" | true |
| "2016-02-04T18:37:17.081Z" | "56b39a5c-c329-48e0-bb39-9593427c1938" | "launch-wizard-1"
| "admin" | true |
| "2016-02-04T18:37:17.115Z" | "56b39a5c-b828-4486-bd4d-410e54d9708b" | "vpc-39a1f45c"
| "admin" | true |
| "2016-02-04T18:37:17.417Z" | "56b39a5d-5540-4383-9c76-605a75822638" | "us-west-2a"
| "admin" | true |
| "2016-02-04T18:37:17.462Z" | "56b39a5d-ca55-4a14-b729-f6d82a9947bb" | "us-west-2c"
| "admin" | true |
| "2016-02-04T18:37:17.464Z" | "56b39a5d-2df3-4abc-9378-5e5a51c8ef1b" | "us-west-2b"
| "admin" | true |
| "2016-02-04T18:37:58.093Z" | "56b39a85-11a6-4b91-bc40-d96fe97b6e5e" | "subnet-64abf313"
| "admin" | true |
| "2016-02-04T18:37:58.106Z" | "56b39a85-9883-4a47-be8f-8ab3210eabec" | "subnet-31b7ef46"
| "admin" | true |
| "2016-02-04T18:38:39.916Z" | "56b39aaf-305a-4117-a4af-110fb7f68894" | "eni-70f1b03b"
| "admin" | true |
| "2016-02-04T18:38:39.919Z" | "56b39aaf-4000-4ea3-8db6-db96e2a2fc47" | "eni-8b125dc0"
| "admin" | true |
| "2016-02-04T18:38:39.958Z" | "56b39aaf-46ef-4d14-936b-caff93046ed5" | "eni-fba6eab0"
| "admin" | true |
| "2016-02-04T18:38:39.960Z" | "56b39aaf-dfe6-46e9-a184-6ale8e48b2ac" | "eni-760b4b3d"
| "admin" | true |
| "2016-02-04T18:38:43.647Z" | "56b39ab3-76fb-41d2-a58c-961d22ec7a0c" | "i-d1f21809"
| "admin" | true |
| "2016-02-04T18:38:43.724Z" | "56b39ab3-7dfd-48d5-b42e-931a0b32ce5a" | "i-d9cd2701"
| "admin" | true |
| "2016-02-04T18:38:43.728Z" | "56b39ab3-46fd-40aa-aeeb-06da674e91c0" | "i-f752b82f"
| "admin" | true |
| "2016-02-04T18:38:43.731Z" | "56b39ab3-f6b6-4d80-976f-6e299fbc8195" | "i-3458b2ec"
| "admin" | true |
| "2016-02-04T18:40:10.029Z" | "56b39b0a-93e4-4e16-8cad-b7bf7e98e9ab" | null
| "admin" | true |
| "2016-02-04T18:40:10.186Z" | "56b39b0a-3783-40ff-97ac-23a5d10d9508" | null
| "admin" | true |
| "2016-02-04T19:47:45.782Z" | "56b3aae1-485a-4992-9335-321d12bdc34d" | "eni-63460e28"
| "admin" | true |
| "2016-02-04T19:48:24.251Z" | "56b3ab08-9086-4ed0-bdb9-bb383a5d4867" | "i-7c6190a4"
| "admin" | true |
| "2016-02-04T19:51:15.982Z" | "56b39b0a-3783-40ff-97ac-23a5d10d9508" | null
| "admin" | false |
| "2016-02-04T19:51:18.950Z" | "56b39b0a-93e4-4e16-8cad-b7bf7e98e9ab" | null
| "admin" | false |
+-----+-----+-----+
```

Alternatively, you can query for the history of a specific Resource. The output below shows a simple resource (with a single property "data") being instantiated and then successfully activating.

```
market> resources 56abc6ba-da69-48b6-b274-5f76e9b7fd30 history get
+-----+-----+
+-----+-----+
| time | userId | attrName | newValue
| changeType |
+-----+-----+
+-----+-----+
| "2016-01-29T20:08:26.180Z" | "admin" | "autoClean" | false
| "Set" |
| "2016-01-29T20:08:26.180Z" | "admin" | "desiredOrchState" | "active"
| "Set" |
| "2016-01-29T20:08:26.180Z" | "admin" | "discovered" | false
| "Set" |
| "2016-01-29T20:08:26.180Z" | "admin" | "id" | "56abc6ba-da69-48b6-
b274-5f76e9b7fd30"
| "Set" |
| "2016-01-29T20:08:26.180Z" | "admin" | "orchState" | "requested"
| "Set" |
| "2016-01-29T20:08:26.180Z" | "admin" | "productId" | "7344b1d4-900f-4da0-
a4a5-319a77250c70"
| "Set" |
| "2016-01-29T20:08:26.180Z" | "admin" | "properties.data" | {}
| "Set" |
| "2016-01-29T20:08:26.180Z" | "admin" | "providerData" | ""
| "Set" |
| "2016-01-29T20:08:26.180Z" | "admin" | "reason" | ""
| "Set" |
| "2016-01-29T20:08:26.180Z" | "admin" | "shared" | false
| "Set" |
| "2016-01-29T20:08:26.180Z" | "admin" | "tenantId" | "7c7da24e-d2af-35c7-
8f6d-d8d16c7f0738"
| "Set" |
| "2016-01-29T20:08:26.375Z" | "admin" | "orchState" | "activating"
| "Updated" |
| "2016-01-29T20:08:26.461Z" | "admin" | "orchState" | "active"
| "Updated" |
+-----+-----+
+-----+-----+
```

In some cases, it is not desirable to maintain the history of specific Resource properties. In this case, the property definition can include the `history` Access Modifier. For more details see the [Resource Types](#) reference.

Deletion

Because of the [\[History\]](#) feature, Resource data is not automatically removed from BPO. However, from a practical point of view, after a Resource has terminated successfully, it will no longer be accessible via the normal BPO APIs. As a result, a successfully terminated Resource is often considered "deleted" or "removed".

If the Resource is the Target of any [Relationships](#), any deletion attempts will fail until the Relationship is manually removed. If the Resource is the Source of a Relationship, the Relationship will be deleted when

the Resource is deleted.

For more information on termination, see the [Lifecycle Operations](#) reference.

Tags

In addition to having user-defined properties, resources may also be **tagged**. Tags are {key, value} string pairs that are applied on a per-resource instance basis. Tags are useful for categorizing resources and supporting contextual queries. Like user-defined properties, tags associate information with a resource, but tags differ from properties in important ways:

- Tags are intended to be applied to resources at the orchestrator or user level and not by the resource provider
- Tags are not based on a resource-type schema definition
- Any tag may be applied to any resource
- Unlike configurable properties, tags do not have **desired** and **observed** state and do not contribute to **differences** on a resource
- Tags are always {key, value} or {key, [values]} string pairs rather than arbitrary collections, objects, and data types supported with user-defined properties
- Tag definitions are defined using REST APIs and not onboarded as part of **model-definitions**

Tags are defined before they are applied to resources. The system has a defined set of **tag-keys** and for each there is a set of **tag-values** that are defined. A resource may have zero, one, or multiple values for a given key. If there are multiple tag-values, the order of the set is not defined.

In order to tag resources:

1. Define a **tag-key** using the Market **tag-keys** API.

```
POST /tag-keys
```

2. Define one or more **tag-values** for the **tag-key** using the Market **tag-keys** API.

```
POST /tag-keys/{tagKeyId}/tag-values
```

3. Assign the desired tag {key, [values]} pairs to a resource on creation or during a resource update.

The assigned tags are stored on a resource's **tags** top-level property.

For example, consider an application that classifies each resource as domestic, international, or multi-national.

Define the **Citizen** key.

```
POST /tag-keys
{
  "key": "Citizen",
  "description": "Resource's place in the world",
  "autoIndexed": false
}
```

Create each of the allowable values for the key.

```
POST /tag-keys/Citizen/tag-values
{
  "value": "Domestic"
}
```

```
POST /tag-keys/Citizen/tag-values
{
  "value": "International"
}
```

```
POST /tag-keys/Citizen/tag-values
{
  "value": "Multinational"
}
```

Assign tag key-values to resources.

```
PATCH /resources/{resourceId}
{
  "tags": {"Citizen": ["Domestic"]}
}
```

To query resources based on tag values, refer to the [Tag Query](#) API reference.

Relationship Instances

Relationship Instances, or simply Relationships, are the active relationships between Resources in BPO's model. Relationships are defined by their [Relationship Type](#) information and expressed in terms of a Source Resource and a Target Resource, indicating the direction of the relationship in a directed graph. These relationships express one Resource's requirement against another Resource's capability as is shown in the following diagram:

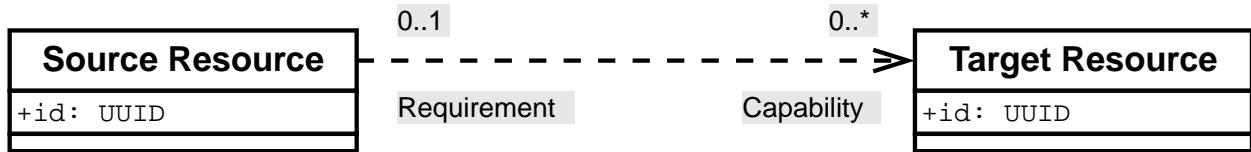


Figure 6. Relationship Modeling with Requirements and Capabilities

The source resource has a requirement which needs to be satisfied by some compatibility on a corresponding target resource. The requirements are generally optional and resources can be created without them. This is not true when used in conjunction with [Service Template Requirements](#), which need successfully created relationships as a precondition for the activation of resources relying on them. Capabilities are many-to-one at present with constraints being implemented in the future.

As Relationships are first class objects in BPO, the implementation of Relationships differ from the logical view of relationships in that the Relationship object itself forms the link between a source and target resource. This can be seen in the following diagram:

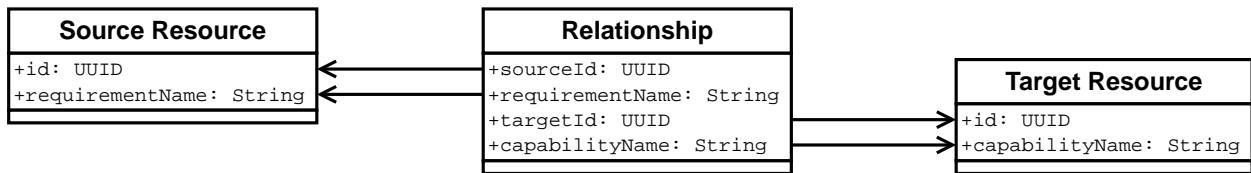


Figure 7. Relationship Instances

The **Relationship** instance links a source resource's id and requirement name to a target resource's id and capability name. It also has several other attributes on it besides those which associate both sides of the relationship between resources.

Top-Level Attributes

Relationships have the following attributes associated with them:

Name	Type	Format	Description
id	string	uuid	Uniquely identifies the Relationship within the catalog. This value is generated by Market when the relationship is created.
relationshipType	string	n/a	Identifies the Relationship type which the relationship is associated with. See the Relationship Types reference for more details.
properties	object	n/a	Contains Relationship Type specific data. See Properties for more details.
sourceId	string	uuid	Contains the UUID of the Source Resource in the relationship.

requirementName	string	n/a	Contains the Source Resource's Requirement name. This name must match an entry defined in the source resource type's requirements section.
targetId	string	uuid	Contains the UUID of the Target Resource in the relationship.
capabilityName	string	n/a	Contains the Target Resource's Capability name. This name must match an entry defined in the target resource type's capabilities section.
orchState	string	enum	Identifies the orchestration state of the Relationship.
reason	string	n/a	Used for display purposes in UIs. Contains additional information in case orchState moves to failed.
providerData	object	n/a	Contains provider specific data for the Relationship. This value is optional and is controlled by the provider of the Relationship. Users should not rely on the values in this attribute.

Relationship Validation

Bpocore validates all relationship creation attempts by checking the compatibility of Requirements and Capabilities against the desired Relationship Type, as well as compatibility of the source and target Resource Types. Relationship Types always have a validSource and validTarget field which describe whether a given Capability Type can be used. For more details on these fields, see [Relationship Types](#). resourceTypes on Capabilities and Requirements are optional fields which, when left empty, will accept all Resource Types. For more details on these fields, see the [Capabilities and Requirements section on Resource Types](#). The validation of a Relationship instantiation is shown in the following diagram:

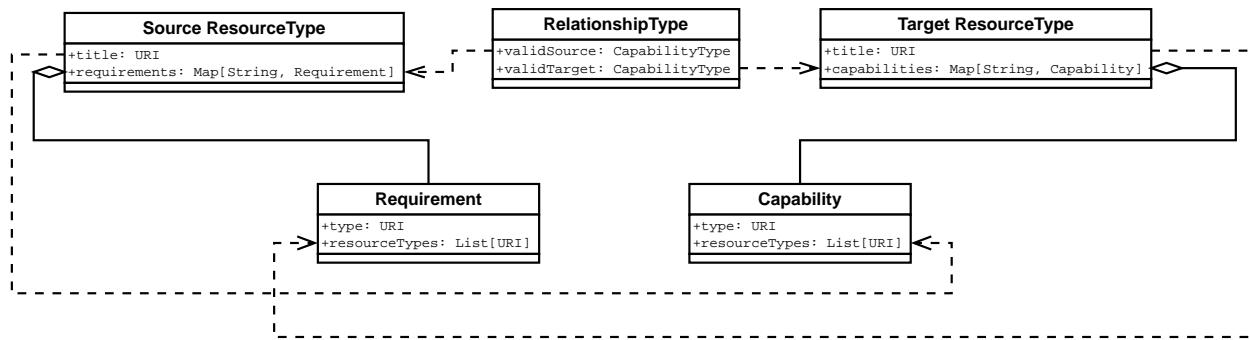


Figure 8. Relationship Validation

The Source ResourceType must be a valid ResourceType (or inherit from one) for the specified Capability.

The Target ResourceType must be a valid ResourceType (or inherit from one) for

the specified Requirement.

The Requirement's Capability Type must be a validSource (or inherit from one) for the Relationship.

The Capability's Capability Type must be a validTarget (or inherit from one) for the Relationship.

Only when all of these conditions are satisfied will the relationship instance be created successfully.

Creation

The POST /relationships API is used to create these relationships by specifying a JSON object conforming to the [Top-Level Attributes](#), minus the id field. The request will be validated and may fail if the requirementName, capabilityName or relationship type is [incompatible](#). On success, a 201 Created response and the newly created Relationship will be returned. Relationships are also dynamically created by the [Template Engine](#) and play a large role in [Service Template](#) usage.

Deletion

The DELETE /relationships/{relationshipId} API will remove the relationship and reply with 201 Deleted. If the Id was invalid, Bpocore will reply with a 404 Not Found instead. There are no normal scenarios under which a relationship cannot be deleted.

Lifecycle Operations

BPO supports three types of lifecycle operations on Resources:

- **Activate** – the process of moving the Resource to the active state.
- **Update** – the process of propagating configuration changes applied to the Resource's properties attribute.
- **Terminate** – the process of moving the Resource to the terminated state.

BPO models both the **desired** and **observed** lifecycle state of Resources as top-level attributes:

- Desired Orchestration State (`desiredOrchState`) – specifies the desired lifecycle state of the Resource as specified by the user (northbound).
- Orchestration State (`orchState`) – specifies the observed lifecycle state of the Resource as reported by the provider (southbound).

The `orchState` can be used to determine whether the user's `desiredOrchState` has been reached or if a failure has occurred in the orchestration process. Often, the `orchState` is similar to the "operational status" of entities in other systems, e.g., "up" and "down" states on network interfaces.

When Resources are created in BPO, the `desiredOrchState` defaults to active and the `orchState` defaults to requested. When discussing Resource lifecycle, it is useful to concisely refer to these states together, this is often done by specifying them in the form `<desiredOrchState>/<orchState>`, e.g., active/requested.

The following table describes the supported `desiredOrchState` values in BPO.

Table 2. Possible `desiredOrchState` values

NAME	DESCRIPTION
active	Specifies the Resource should be activated.
terminated	Specifies the Resource should be terminated.
unspecified	Used only for Discovered Resources .

The following table describes the supported `orchState` values in BPO:

Table 3. Possible `orchState` values

NAME	DESCRIPTION
requested	Only used for Managed Resources on initial creation.

NAME	DESCRIPTION
activating	Indicates the Resource is in the process of activating.
active	Indicates the Resource has activated and is operational from an orchestration perspective.
terminating	Indicates the Resource is in the process of terminating.
terminated	Indicates the Resource has terminated.
failed	Indicates the most recent lifecycle operation on the Resource did not complete successfully.

The following table describes the orchestration state combinations found on Resources in BPO:

Table 4. Orchestration state matrix

DESIREDORCHSTATE	ORCHSTATE	DESCRIPTION
active	requested	Resource has been created and the provider may be performing the activation operation.
active	activating	Provider is in the process of activating the Resource.
active	active	Resource has been activated and is considered operational.
active	failed	Provider failed to perform the activation or most recent update operation.
active	terminating	Provider has observed that the Resource is terminating unexpectedly.
active	terminated	Provider has observed that the Resource has terminated unexpectedly.
terminated	requested	User has requested termination before the provider has completed the activation operation. The provider may still process the in-flight activation operation before processing the termination operation.
terminated	activating	User has requested termination before the provider has completed the activation operation. The provider may continue waiting for the activation operation to complete before processing the termination operation.
terminated	active	User has requested termination but the provider has not started processing the termination operation.

DESIREDORCHSTATE	ORCHSTATE	DESCRIPTION
terminated	failed	User has requested termination but The provider failed to complete the most recent lifecycle operation.
terminated	terminating	Provider is in the process of terminating the resource.
terminated	terminated	Resource was successfully terminated by the provider. Terminated Resources are only returned by history queries.
unspecified	activating	Provider has observed the Discovered Resource is in the process of activating.
unspecified	active	Provider has observed the Discovered Resource is operational.
unspecified	terminating	Provider has observed the Discovered Resource is in the process of terminating.
unspecified	terminated	Provider has observed the Discovered Resource has terminated. Terminated Resources are only returned by history queries.

The following diagram illustrates the valid lifecycle transitions:

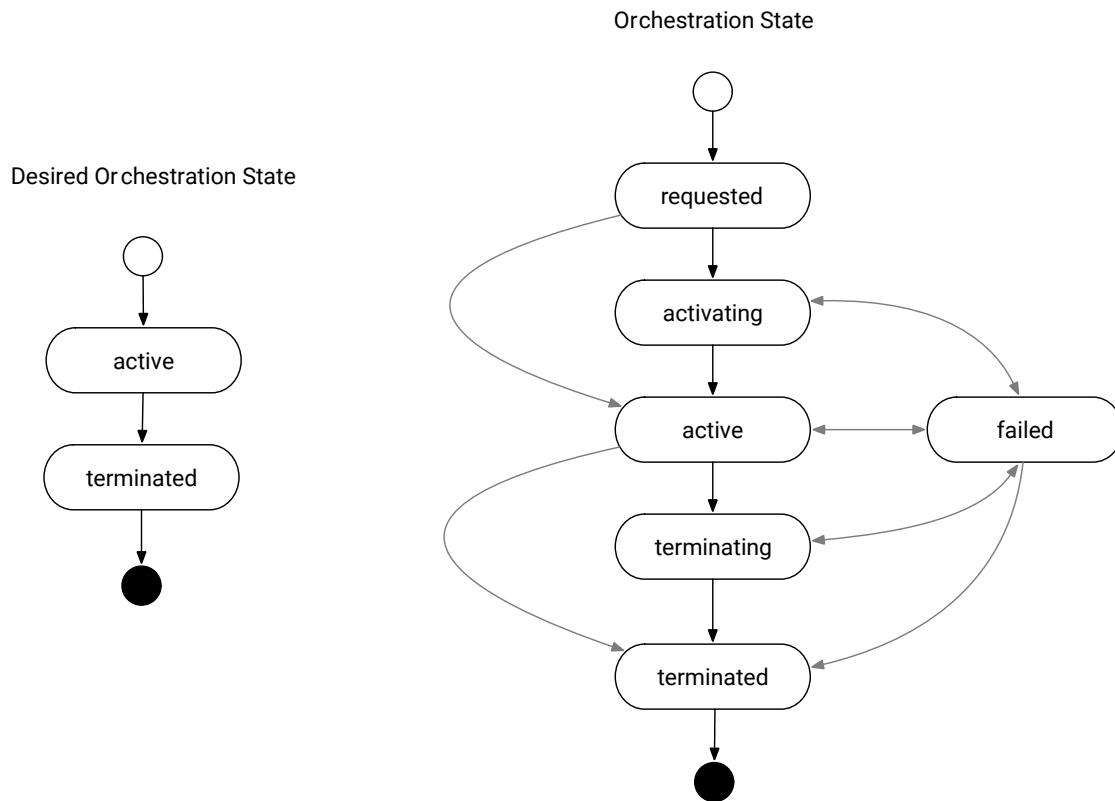


Figure 9. Valid Lifecycle Transitions

Activate

Activation is the process of realizing Resources in managed Domains. Activation is started by making a request to create the top-level resource:

```
POST /market/api/v1/resources
```

In some simple cases, realizing a Resource in a managed Domain may only involve calling a single API to create a new entity. However, activation may also require creating or updating multiple entities which are abstracted by the Resource. In addition, activation may also require extra steps to ensure the Resource arrives in an acceptable baseline state which is operational from an orchestration perspective. Once activation completes, the Resource is considered **active**. What active means depends on the resource. Here are some examples of what active could mean:

- The VM is running and has a publicly accessible IP address.

- The IP VPN service is provisioned and there is connectivity between sites.
- The Load Balancer is running and requests can be forwarded to backend processes.

The steps involved in activation depend on the Resource and the provider's implementation. For composite Resources managed by the Template Engine, the process involves creation (and activation) of sub-resources. This process may repeat recursively. For resources provided by Resource Adapters, the process involves interaction with external systems.

When the `desiredOrchState` is set to `active`, the Resource's provider is notified asynchronously. Since the `desiredOrchState` defaults to `active` this typically occurs immediately after the Resource is created. However, in the future, BPO may support other Desired Orchestration States, so the distinction between Resource creation and activation is important.

When a provider receives an activation notification, the provider may choose to update the `orchState` of the Resource to reflect that activation is in progress. This is done by setting the `orchState` to `activating`. The provider will update the `orchState` to `active` to reflect that the activation process is complete. Alternatively, if the provider's processing of the activation notification will block (until activation completes) the provider may update the `orchState` from `requested` to `active` directly.

When updating the `orchState` to `active` the provider is responsible for setting any properties with the `output` Access Modifier enabled.

While a resource is `activating`, it is possible to terminate the resource by deleting the resource.

Auto Clean

Auto Clean is a feature of BPO which executes rollback behavior when Resource activation fails. It is enabled on individual Resources by setting the `autoClean` attribute to `true` when instantiating the Resource. The rollback process is primarily driven by BPO's Template Engine component; however, it can be generalized to all providers:

When Auto Clean is enabled on the Resource and activation fails, the provider must undo any effects introduced up to the point where the activation process failed.

BPO's Template Engine component implements Auto Clean by automatically terminating existing sub-Resources when Resource activation fails.

As an example, the following diagram illustrates the Auto Clean process in 5 stages:

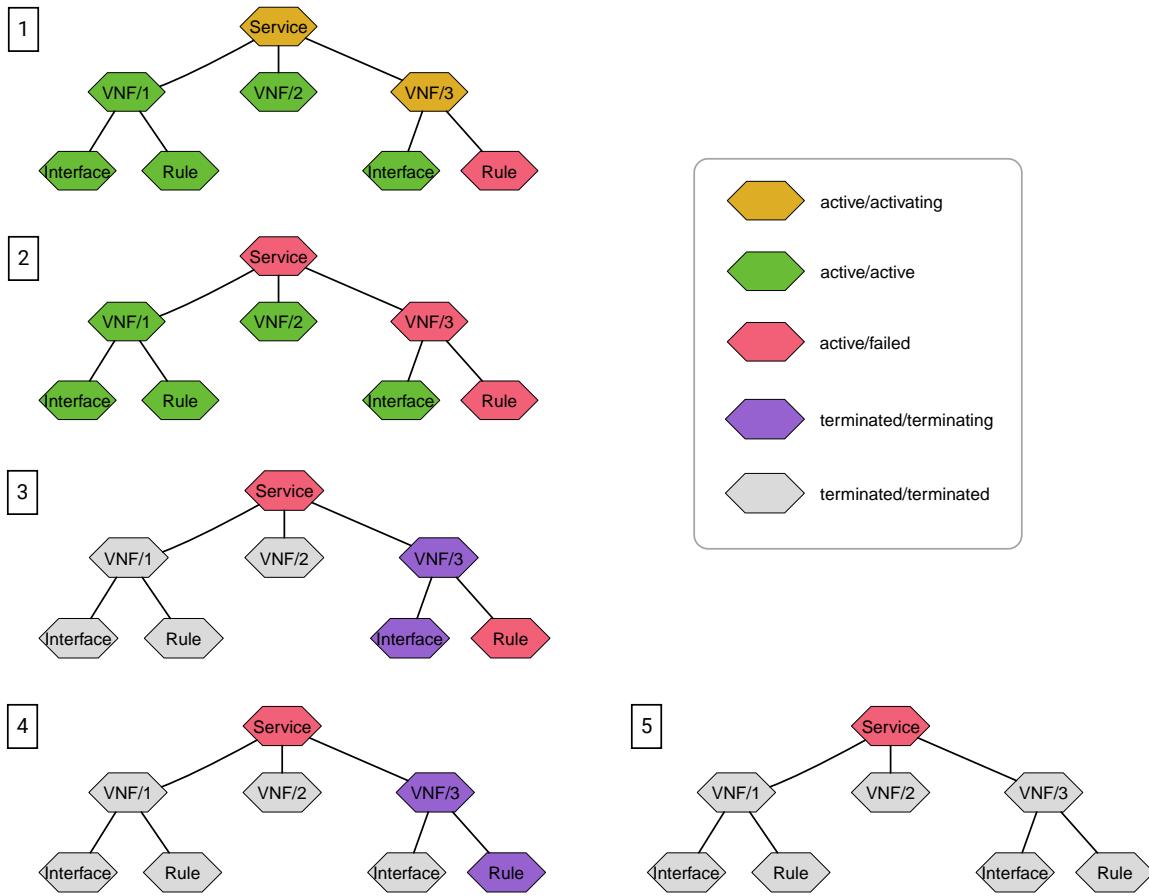


Figure 10. Auto Clean

- The first stage shows in-progress activation of the top-level "Service" Resource. However, activation of the "VNF/3:Rule" Resource has failed.
- The second stage shows "VNF/3:Rule" activation failure having propagated upwards. Once the "Service" activation fails, Auto Clean is triggered.
- The third and fourth stages show Auto Clean in-progress.
- The fifth (and final) stage shows "Service" and all sub-Resources after Auto Clean has completed successfully.

Update

Update is the process of applying changes to configurable properties on Resources. Updates are initiated by making a request to:

```
PUT/PATCH /market/api/v1/resources/{resourceId}
```

The `PUT` request requires the full state of the updated resource in the body while the `PATCH` requires only

the changed properties in the body.

Configurable properties may be modified if:

- The property definition contains `updatable = true`.
- The Resource is in one of the following states: `active/active`, `active/failed`, `active/requested`.

When a configurable property is modified, the `differences` attribute is updated to reflect the difference between the existing observed state and the new desired state. When the update is successfully processed (i.e. the new value has been recorded in BPO), the associated provider is notified. The provider is responsible for realizing the change in the Domain and updating the observed state of the Resource resulting in the differences being removed. Users performing configuration changes can determine when a configuration change has propagated by checking the `differences` attribute on the Resource.

If the provider fails to realize the changes in the Domain, the provider should set the Resource's `orchState` to `failed` and update the `reason` attribute to indicate the cause of the failure.

Terminate

Termination is the process of undoing steps taken to realize Resources in managed domains. Like activation, the steps depend on the type of Resource and the implementation of the Resource's provider.

Termination is initiated by a delete operation.

```
DELETE /market/api/v1/resources/{resourceId}
```

The delete operation sets the `desiredOrchState` property to `terminated` and the resource's provider is notified asynchronously. The provider may choose to update the `orchState` of the Resource to `terminating` to reflect that termination is in progress. The provider will update the `orchState` to `terminated` to reflect that the termination process is complete. Alternatively, if the provider's processing of the termination notification will block (until termination completes) the provider may update the `orchState` to `terminated`.

When the termination process completes (i.e. the resource is `terminated/terminated`) the Resource is automatically removed from the [Instance Catalog](#) along with any incoming and outgoing Relationships. Termination may only be performed on Resources which have no incoming Relationships or Resources where all **dependents** have been terminated.

Declarative Service Templates

When a configuration change is made to a Resource provided by a Declarative Service Template, the Template Engine is notified. The Template Engine reacts to configuration changes by updating sub-resources which depend on the affected properties. The Template Engine determines which sub-resource properties to update by identifying:

1. Sub-resource properties which are directly dependent via use of `getParam` or `getOptionalParam` directives.
2. Sub-resource properties which are indirectly dependent via use of `getAttr` on another dependent sub-resource property.

The Template Engine will update the observed state of the Resource when all of the related differences on the sub-resources have been cleared.

If the Template Engine fails to update one of the sub-resources, the process will be aborted and the Resource will transition into the `failed` orchestration state with the `reason` message set accordingly.

For resources having declarative update plans, a new (update/terminate) request while a prior request is being processed will preempt the presently running plan and the new request will be processed.

See [Service Templates](#) for more information.

Imperative Plans

When a configuration change is made to a Resource whose lifecycle operation is provided by an Imperative Service Plan, the Template Engine is notified. The Template Engine reacts to the configuration change by executing the "update" plan associated with the template. For resources having imperative update plans, all new (update/terminate) requests while a prior request is being processed will be queued until the prior request has been completed. Only the last (update/delete) request received is stored in the queue, all other requests are ignored.

See [Imperative Plans](#) for more details.

Failure

When providers encounter errors while processing lifecycle operations, they should update the `orchState` to `failed` and set the `reason` attribute to indicate the source of the failure.

Mixing Imperative and Declarative Lifecycle Operations

A [Service Template](#) may specify the activate, update, and terminate lifecycle operations in two ways:

1. *Declaratively*, by specifying what to create via the [resources](#) section via the template language DSL
2. *Imperatively*, by referencing scripts from the [plans](#) section

A service template may use just one of these methods, or mix them subject to the following rules:

1. If a service template has a [resources](#) section, declarative activate, update and terminate plans are created for that resource by the [Template Engine](#). The [update](#) and [terminate](#) plans may be overridden by imperative plans in the [plans](#) section (see [Declarative Activate](#) below).

NOTE

An empty resource section is the same as not specifying a resource section.

2. Imperative plans override any declarative plans. However, a service template must not specify both a [resources](#) section *and* an activate imperative plan (see [Imperative Activate](#) below).

For example, the following resource is activated by a declarative plan based on the [resources](#) section, but the update and terminate operations are handled by imperative plans:

```
serviceTemplates {
    someresource {
        title = "someresource"
        description = ""
        implements = somepackage.resourceTypes.someresource

        plans {
            update {
                type = script
                language = python
                path = "types/tosca/someresource_update.py"
            }
            terminate {
                type = script
                language = python
                path = "types/tosca/imperative/imperative_queue_requests_terminate.py"
            }
        }

        resources {
            sub-resource {
                type = openstack.resourceTypes.VlanPortSet
                properties {
                    domain_id = {getDomain {getParam = image}}
                    port_info = "foo"
                }
            }
        }
    }
}
```

Finally, when no plan is specified for an operation a fallback behavior applies (**Default** in the tables below):

- The default activate plan simply sets the resource `orchState` to `active`. It does not create any sub-resources.
- The default update plan simply clears the resource differences. It does not modify any sub-resources, which is useful for imperatively activated resources which do not support update—effectively, the resource ignores the operation.
- The default terminate plan sets the resource `orchState` to `terminated`.

A resource which uses the default terminate plan should not have any sub-resources. If any sub-resources are present termination will fail, resulting in a *failed* `orchState`.

The following sections explain these rules from the perspective of resources with [imperative activate plans](#) and [declarative activate plans](#).

Imperative Activate

If a resource is [activated through an imperative plan](#), it *must not* have declarative operations (it must not specify a `resources` section).

Table 5. Rules for service templates with an [imperative activate plan](#)

ACTIVATE	UPDATE	TERMINATE	SUPPORTED?	CAVEATS
Imperative	Imperative	Imperative	Yes	
Imperative	Declarative	Imperative	No	
Imperative	Default	Imperative	Yes	
Imperative	Imperative	Declarative	No	
Imperative	Declarative	Declarative	No	
Imperative	Default	Declarative	No	
Imperative	Imperative	Default	Yes	The activate and update plans should not create any sub-resources.
Imperative	Declarative	Default	No	
Imperative	Default	Default	Yes	The activate plan should not create any sub-resources.

Default indicates that the plan type is missing from the template definition (i.e. there is no `resources` section and no imperative plan specified in the `plans` section).

Declarative Activate

For a service template with a declarative `activate` plan ([resources](#) section), the `update` and `terminate` lifecycle operations *may* be handled by imperative plans, subject to caveats:

Table 6. Rules for service templates with a declarative activate plan

ACTIVATE	UPDATE	TERMINATE	SUPPORTED?	CAVEATS
Declarative	Declarative	Declarative	Yes	
Declarative	Imperative	Imperative	Yes	
Declarative	Declarative	Imperative	Yes	
Declarative	Imperative	Declarative	Yes	The update plan must not create any sub-resources, or the declarative terminate plan will fail.

Template Engine

Overview

The Template Engine is introduced in [Main Concepts](#). The Template Engine is responsible for providing composite resources in BPO via the built-in **WorkflowManager** resource provider. Template Engine managed resources are in a built-in **Planet Orchestrate** domain. Composite Resources are used to aggregate or chain resources (which may themselves be composite resources) into higher level service abstractions. Often, composite resources are used to represent logical entities spanning multiple managed domains.

The **schema** of composite resources are defined using [Resource Types](#) definitions in the same way as other resources. Composite resources are **implemented** using [Service Templates](#) definitions. The Service Template definition describes the **sub-resources** which should be **composed** by the composite resource. The description includes how desired property values are specified for sub-resources. These desired values can either be constants (i.e. hardcoded strings, numbers, integers, booleans, objects, or arrays) or dynamic values computed with **directives**. For details on the suite of supported directives, see [Service Template Directives](#).

The Template Engine's main responsibility is to provide support for the [Lifecycle Operations](#) on composite resources. For example, when a composite resource is activated, the Template Engine is notified in the same way as a Resource Adapter is notified for resources provided by an RA. Update and termination are handled similarly.

Sub-resources may be provided by any supported resource provider. A sub-resource may be:

- another composite resource provided by the template engine via the built-in **WorkflowManager** resource provider
- a managed resource provided by a Resource Adapter

Declarative and Imperative Lifecycle Operations

Service Templates may define lifecycle operations using either **imperative programming** or **declarative programming**.

Imperative programming involves specifying the sequence of steps which the machine must execute to achieve some goal. Programs written in most programming languages (e.g., Python, C, Java, shell scripts) are written in an imperative style.

Declarative programming avoids specifying the sequence of steps and instead focuses on providing a description of the end goal. Typically this is done in a way which removes side effects and control flow. If you have written SQL queries or Makefiles before, you have worked with declarative programming.

Lifecycle Plans

The Template Engine uses information in a service template to realize the lifecycle operations (e.g., activate, terminate, and update) for a composite resource. The Template Engine is responsible for generating a sequence of tasks, known as a plan, to realize lifecycle operations. The plans are specified by:

- `resources` - attribute which contains a **declarative** specification of the lifecycle plans with respect to how they affect sub-resources
- `plans` - attribute which specifies imperative plans (i.e., Python scripts) that implement the lifecycle plans for the resource

A service template may specify **lifecycle plans** by either specifying a `resources` section, or specifying lifecycle (activate, terminate, update) plans in the `plans` section. `resources` and `plans` sections may be mixed in certain configurations (e.g., a declarative `resources` section for activate, and imperative `plans` for update and terminate—see [Mixing Imperative and Declarative Lifecycle Operations](#)). It is always valid to specify a lifecycle declarative plan while defining custom operations in a `plans` section (see [Custom Operations](#)) and validation operations in a `validators` section (see [Custom Validation Operations](#)).

Once the Template Engine has obtained the Plan for a composite resource, it can proceed to execute the plan.

Executing Declarative Plans

When using a declarative service template to implement a resource type, the template engine:

- acts as the provider of the resource
- requests the creation of sub-resources
- creates a `MadeOf` relationship between the composite resource and sub-resources. This relationship uses the composed requirement and composable capability inherent to all resource types derived from `tosca.resourceTypes.Root`.
- creates any other relationships as defined in a `requirements` section
- propagates updates from the composite resource to sub-resources based on the specified directives

These details are described in the following sections.

Activation

This section focuses on how the Template Engine handles activation of composite resources associated with Service Templates containing a declarative specification of the sub-resources (i.e, the `resources` attribute is given).

When the Template Engine receives the notification to activate a composite resource it will generate a Plan which is a series of requests to BPO's [Instance Catalog](#). The requests will ask for the instantiation

of:

- Resource Instances which map to the sub-resources specified in the Service Template.
- madeOf Relationship Instances which will relate the composite Resource to the sub-resources.
- Any other Relationship Instances as defined in sub-resources' requirements sections.

The ordering the of requests is determined by dependencies of the sub-resources. Specifically:

- If sub-resource "B" contains a `getAttr` or `getOptionalAttr` directive on sub-resource "A", then "A" will be instantiated AND activated first.
- If sub-resource "C" contains an `activateAfter` directive on sub-resource "B", then "B" will be instantiated AND activated first.

The activation of a Service containing three sub-resources ("A", "B", and "C") with the dependencies described above is shown in the **Sequential Activation** figure below.

If sub-resources do not depend on each other (directly or indirectly) then they can be instantiated in parallel. For example, given the same example, if "C" does not depend on "B" then the activation of sub-resources may be done in parallel. The plan activation is shown in the **Parallel Activation** figure below.

Sequential Activation



Parallel Activation

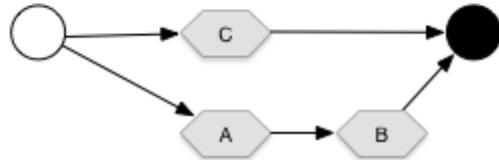


Figure 11. Service Activation Sub-resource Sequence

The ordering of tasks in a plan is important because the values required for the request to instantiate a sub-resource may depend on values gathered from the activation of another sub-resource. A task in the plan is executed when the task is **Ready** (i.e., the task does not wait on any prerequisite tasks to complete).

After a sub-resource is created, the Template Engine creates a `MadeOf` relationship from the composite resource to the sub-resource automatically using the `composed` requirement on the composite resource and the `composable` capability on the sub-resource. As a result, we can query the dependencies of the composite resource to get to the sub-resources and query the `dependents` of the sub-resource to get

the composite resource.

Once all of the sub-resources have been activated, the Template Engine updates the state of the composite resource to active/active.

NOTE A resource configuration update is not allowed while the resource is in the *activating* orchestration state. An attempt to update a resource while it is activating will result in a 409 Conflict response.

Activation Failure

The successful activation of the composite resource requires:

- Successful activation of all sub-resources
- Successful setting of all `output` properties of the composite resource

If the activation is not successful, the failure `reason` property is set on any failing sub-resource and the composite resource. The `orchState` property is set to `failed` on any failing sub-resource and the composite resource. If `autoClean` is enabled on the composite resource, the sub-resources will be cleaned. See [Auto Clean](#). The top-level composite resource will remain in the `failed` state.

Termination

The Termination process is similar to the Activation process described above. When the Template Engine receives the request to terminate a composite resource it will generate a Plan which is a series of tasks to terminate the service. The requests will:

- Terminate sub-resources defined in the template
- Delete the `MadeOf` relationships between the composite resource and the sub-resources

By default, the termination of sub-resources is done in parallel. It does not assume that the termination order needs to be the reverse of the activation order. If a certain termination order is required for the sub-resources, use the `terminateBefore` directive to explicitly enforce the sequence.

Termination Failure

The successful termination of the composite resource requires:

- Successful termination of all sub-resources

If the termination is not successful, the failure `reason` property is set on any failing sub-resource and the composite resource. The `orchState` property is set to `failed` on any failing sub-resource and the composite resource.

Update

Template engine processing is only performed on a resource configuration update to a templated resource where either:

- the update includes a configuration change
- the resource still has uncleared differences (regardless of whether the update includes a new configuration change)

The REST API to perform a resource configuration update is:

```
PUT/PATCH /resources/{resourceId}
```

A change to a sub-resource does not cause the template engine to evaluate a plan for the host resource. See the [Trigger](#) section for details on how to cause an update on the host resource based on observed changes to a sub-resource.

An update of the composite resource instructs the template engine to reevaluate the `resources` section of the template to generate a declarative update plan based on dependencies. For example, if there are `getParam` or `getAttr` directives that propagate changes to sub-resources, these changes are included as tasks in the update plan. A change to a sub-resource is issued as a `PATCH` request on the resource. As with activation plans, the tasks are sequenced based on the dependency order. A `PATCH` request to a sub-resource results in [Differences](#) on the sub-resource between the desired state and the observed state. A task is not considered complete until the [differences](#) on the sub-resource have been cleared. It is the responsibility of the resource provider for the sub-resource to clear the differences. The update plan of the composite resource is not considered complete until all tasks are complete. When the tasks are complete, the differences on the composite resource are cleared by the template engine.

For more information on resource updating, see the [Lifecycle Update](#) and [Resource Updates](#) sections.

NOTE

- `output` properties are intended to be **set-once** properties and are not reevaluated during an update of the composite resource
- service template directives are reevaluated only if there is a change in a dependency (either direct or indirect)
- a `PUT/PATCH` to the observed state of the composite resource for either `config` or non-`config` properties does not cause the update plan to be executed

```
PUT/PATCH /resources/{resourceId}/observed
```

Executing Imperative Plans

When using an imperative service template to implement a resource, the template engine executes the specified imperative plan (i.e., script) when a lifecycle operation is requested. The imperative plan is

responsible for:

- providing the composite resource (shared responsibility with the template engine)
- creating any sub-resources
- creating any relationships between the composite resource and sub-resources and between sibling sub-resources
- propagating changes when the composite resource is updated
- clearing differences when the composite resource is updated (if `autoClearDifferences` is set to false)

In general, imperative plans allow for greater flexibility in controlling how lifecycle operations are implemented, but require the author to implement more complex behaviors. See [Imperative Plans](#) for details on how to implement an imperative plan.

Update Concurrency

Unless there are dependencies between resources, the lifecycle operations for different resources are performed concurrently by the Template Engine. Update requests for a given resource are applied serially subject to queueing behavior (see below). Resource updates are performed according to the following process:

1. Update is applied to set the desired attributes and properties and the differences are set on the BPO resource entity
2. Request is made to the resource provider to process the update (i.e., achieve the desired state on the managed resource)
3. When the update is complete, the resource provider clears the differences on the resource by patching the observed state

Regardless of the resource provider, step (1) is performed serially for a given resource. For a template managed resource, the request in step (2) is performed by an update plan for the resource. Update plans for a given resource are handled in a per-resource queue.

If a declarative resource update plan is being executed when another update request is received, all of the tasks in the current plan are terminated and the new plan is executed.

If an imperative resource update plan is being executed when another update request is received, the new update request will be queued until the current execution is completed. Only the last (update/delete)request received is stored in the queue. If a previous update request has been enqueued, it is evicted from the queue and the new request is enqueued. The resource differences represent the accumulated set of differences on the resource that are yet to be cleared (i.e., the overall differences between the desired state and observed state rather than just the differences associated with the latest update request).

An imperative update plan **should not update** the configuration on the resource on which it is executing; however, it may update the *observed* state of the resource.

Resource Updates

Modifications to resources are made through resource *updates*. There are two types of updates:

1. Resource *desired* configuration update
2. Resource *observed* state update

Resource update behavior is governed by rules based on resource type property definitions and access modifiers. See the [Property Definitions](#) reference section for details.

Resource Configuration Update

A *resource configuration update* modifies properties that define a resource's desired configuration (i.e., `config=true` properties). A property must be designated as updatable in order to be changed (i.e., `updatable=true`).

A configuration change may be made only when the resource is in one of the following *desiredOrchState* /*orchState*:

```
active/active, active/failed, active/requested
```

When a configurable property is modified, the `differences` attribute is updated to reflect the difference between the existing observed state and the new desired state.

When the update is successfully processed (i.e, the new value has been recorded in BPO), the associated resource provider is notified. The provider is responsible for realizing the change in the Domain and updating the observed state of the Resource.

The resource provider notification takes the form of:

- executing an update plan for a templated resource
- issuing a PUT to the RA for an RA-provided resource

The resource provider notification is made if the either:

NOTE

- the update includes a `config=true` property change
- the resource still has uncleared differences (regardless of whether the update introduces any new changes)

A configuration change is made using the REST API:

```
PUT/PATCH /resources/{resourceId}
```

The PUT/PATCH REST API responds with a 200 OK response immediately after the new desired configuration has been validated and committed to the database. The resource provider handles the configuration change notification asynchronously.

Users performing configuration changes can determine when a configuration change has been realized by checking the `differences` attribute on the Resource. The `differences` attribute is cleared after the provider of the resource reports back an observed state that matches the configuration change. While the configuration change is propagating, there will be differences set for the affected properties. Once the configuration change has propagated, the related differences will be removed. If the provider fails to realize the changes in the Domain, the provider should set the Resource's `orchState` to failed and update the `reason` attribute to indicate the cause of the failure.

For templated resources, there are rules for how the update plans for concurrent updates are processed. See [Update Concurrency](#) in the [Template Engine](#) section for details.

NOTE

Because a resource configuration change update on a templated resource executes the update plan for the resource, an *imperative* update plan **should not update** the configuration on the resource on which it is executing; however, it may update the *observed* state of the resource.

Resource Observed State Update

A *resource observed state update* modifies the observed state of a resource's properties. Observed updates apply to both `config=true` and `config=false` properties and do not require that the property be designated as `updatable=true`.

If a property is designated as `output=true`, it cannot be updated to a different value.

An observed update to a resource does not execute an update plan for a templated resource or a sync to the RA for an RA-provided resource.

An observed state update may be made while the resource is in any `desiredOrchState/orchState`.

Observed state updates are normally made implicitly by BPO based on data provided by a resource provider to reflect the status of the resource in a managed domain. An observed update may be made directly using the REST API:

```
PUT/PATCH /resources/{resourceId}/observed
```

The PUT/PATCH REST API responds with a 200 OK response immediately after the observed state has been committed to the database.

NOTE

If a property is updatable and has a default value, a retraction will behave as an update reinserting the property's default value.

REST Methods for Resource Updates - PUT versus PATCH

Resource updates may be issued using either the REST `PUT` or `PATCH` request methods. A `PUT` takes the full resource in the body while a `PATCH` requires only the changed attributes and properties in the body.

Updates using `PUT`

A `PUT` API call takes the full resource¹ in the body and changes the current resource to match the resource passed in the body. Typically the `PUT` is used in conjunction with a `GET` where the client code first does `GET` to retrieve the current state of the resource, makes changes to specific properties, and then uses the changed resource in the body of the `PUT`.

If the `PUT` changes any properties that are not permitted to be changed or is missing any properties that are required , the `PUT` will return a failure. For example, failures will result from:

- Omitting a non-optional property (even omitting a `config=false` property in a resource configuration update)
- Modifying a `config=false` property in a resource configuration update
- Modifying a `updatable=false`, `config=true` property in a resource configuration update

A `PUT` can be used to retract an optional property by omitting the property from the body in the request.

¹The `differences`, `createdAt`, and `updatedAt` attributes do not need to be included in the body and are ignored if they are in the body.

Updates using `PATCH`

A `PATCH` API need only contain the changed properties in the request body. `PATCH` allows for selective modification of array and object properties. A `PATCH` is subject to the same restrictions as `PUT` with respect modifying properties. It may include an `updatable=false` property if the value doesn't cause a change. A `PATCH` cannot contain the `differences` attribute.

There are two variants of `PATCH`. Both variants use the `PATCH` HTTP method and the `application/json` content-type. The standard `PATCH` specifies a sub-tree of the resource's properties containing the changes. For example, to change the number of CPUs in a Virtual Machine, the `PATCH` may contain only the property changed:

```
{  
  "properties": {  
    "numCpus": 24  
  }  
}
```

The rules for handling a standard `PATCH` request on a Resource's properties are as follows:

1. If the property is a boolean, integer, number, or string the value from the PATCH overwrites the old value.
2. If the property is an array, the value from the PATCH overwrites the old value completely. The entire array content must be specified in the PATCH.
3. If the property is an object, the updated object is the union of the old object and the object specified in the PATCH. For nested sub-objects, the result is the union of the specified sub-object and the existing sub-object with preference given to values from the object specified in the request.

Alternatively, the PATCH API may be specified using the [RFC 6902 JSON Patch](#) syntax which specifies changes as a sequence of add, replace, remove operations. For example:

```
[  
  {  
    "op": "replace",  
    "path": "/properties/numCpus",  
    "value": 24  
  }  
]
```

Unlike the standard patch, the JSON Patch form is able to retract optional properties from the resource.

Updating Orchestration States

The `desiredOrchState` is considered part of the resource's configuration and should be updated using a resource configuration update. `label` and `description` are also considered part of the resource configuration.

The `orchState` and `providerData` are considered part of the resource's observed state and should be updated using an observed state update.

Updating Discovered Resources

The resource updates described in the previous sections are primarily written with respect to managed resources (i.e., `discovered=false` resources). Because discovered resources have only *observed* state, they should be updated using observed updates. In most cases, discovered resources are updated by an RA and not directly changed through REST APIs.

See [Discovered Resources](#) for more details.

Validation Operations

This section describes:

- The [built-in validation](#) of the `POST /resources` and `PUT/PATCH/DELETE /resources/{resourceId}` Market APIs;
- Extending validation via [Custom Validation Operations](#);
- How to [execute custom validation operations](#) when modifying a resource; and
- [Pre-validating requests](#) via the `POST /resources/validate` and `POST /resources/{resourceId}/validate` Market APIs.

Built-in Validation

When a POST request is made to the `/resources` Market API, the following validations will be done before a Resource Instance is created in the Market Database.

1. The body should conform to the schema of a Resource. For example, it should include `productId`, and if `label` is specified, it should be a string.
2. The properties attribute of the body should conform to the corresponding Resource Type definition. For example, if `prop1` is of type `string` in the Type Definition, the body should specify a string for this property.
3. The properties should also conform to the access modifiers defined the type definition. For example, an `output=true` property should not be specified by the user when the resource is created, and a property without `optional=true` and without a `default` value should exist in the body. See [Property Definitions](#) for more information on schema and access modifiers
4. If a `format` is defined for a property, the property should conform to the format. For example, if `vlan-id` format is used, the property can not be a value outside `[1, 4094]`
5. If the specified product has [constraints](#), the body will also be validated against the constraints.

A POST call will be rejected if any of the validations above has failed. A similar check is also done for PATCH and PUT of an existing Resource Instance.

Custom Validation Operations

The built-in validations above provide basic checks against the to-be-created Resource. However, sometimes there are cases not covered by the built-in validations, such as:

1. A more complex validation across different properties. Examples:
 - a. `prop1` should be specified only when `prop2` is `true`
 - b. `prop3` should be in the range of `[0, prop4]`
2. A validation process that could potentially involve custom computation or external system checking. For example, calling out to an external reservation system to see if there is availability

BPO provides a way for the Service and RA author to define **Custom Validation Operations** which can be optionally executed in addition to the built-in ones.

Templated Resources

Custom validators can be defined in Service Templates by configuring a [validators section](#).

Defined similarly to lifecycle and custom operation plans, validation plans can be run to validate corresponding lifecycle operations.

See [Service Template validators](#) and [Imperative Plans: Validation Operations](#) for details on configuration and execution of a validation plan.

RA Resources

Custom validators can be defined in a resource adapter using bp-prov commands or custom resource drivers. For more information, please refer to the RPSDK documentation.

Execution of Custom Validation Operations

To execute a custom validation for `POST` to **/resources** or `PATCH`, `PUT`, and `DELETE` to **/resources/{resourceId}** Market APIs, the caller must pass `validate=true` as a parameter.

If defined, the custom validation will be run after the built-in validation. The first failure item's status code and reason in the Validation Report returned by the custom validator will be returned as an error to the API caller.

The Custom Validator will not be executed if built-in validation fails

If custom validation is not defined but `validate=true` is passed in, only the built-in validation will be performed.

When the validation is run as a part of the lifecycle operation, the API call will block until the validation passes and the operation is committed to the database.

Resource Validation APIs

As an alternative option to perform validation as a part of the Resource lifecycle API call, BPO provides **validate** APIs that validates the body of the call without actually creating, updating, or deleting the resource when the validation passes. In addition, the validate API returns a **Validation Report** with more details and potentially with multiple errors for a single validation call.

These APIs are intended for use by a user interface. For example, as a user fills out a form the inputs can be submitted to the **/resources/validate** API to rapidly provide feedback to the user. A Validation Report provides sufficient detail to note issues on individual form elements.

NOTE

Unlike many BPO APIs, the validate APIs are blocking. HTTP requests made to them do not return a response until the validation operation is complete.

It is important to note that an operation which passes validation may still fail when actually executed.

API Paths

- POST **/resources/validate**: Validate a Resource [activate operation](#).
- POST **/resources/{resourceId}/validate**: Validate a proposed [update](#) or [terminate](#) of an existing Resource.

Both APIs only support the HTTP POST method.

API Parameters

full (true/false)

Only applicable to **/resources/validate** API. When `true` (the default), the validator should validate the inputs as a full Resource. When `false`, the validator should validate the resource only based on the information provided.

For the built-in validator, full validation will fail if the resource to-be-validated is missing required properties.

For custom validations, it will be up to the validator author to determine what a partial validation means. It could still fail the validation for reasons such as "B should not be empty when A is specified".

NOTE

Even if `full=false`, `productId` is still be required for a resource validation.

method (PATCH/PUT/DELETE)

Only applicable to **/resources/{resourceId}/validate** API. Used to specify whether the validation is for the [update](#) (PUT or PATCH), or [terminate](#) (DELETE) of the existing resource.

custom (true/false)

Whether custom validation should be executed. When `true` (the default), the custom validation will be executed in addition to the built-in validations. When `false`, only the built-in validation will be executed.

NOTE

If the built-in validations fail, the custom validation will be skipped even if `custom=true`. An item with code 409 and a reason will be added to the Validation Report.

API response

The **/validate** API returns a 200 OK as long as validation is executed.

Validation Report

The **/validate** API returns a 200 OK as long as validation is executed. Any validation failure will be indicated in the Validation Report returned in the response body, which conforms to the following schema:

```
{  
    state {  
        title = State of the report  
        description = Indicator of whether the validation overall is successful or failed.  
        type = string  
        enum = [ passed, failed ]  
    }  
    results {  
        title = Detailed results of the validation  
        description = Detailed results, each indicating whether a specific validation test passes or not. It can be empty, but if specified, the logical conclusion of all the results should be coherent with the overall state of the report.  
        type = array  
        items {  
            type = object  
            properties = {  
                status {  
                    type = integer  
                    title = HTTP status code of the validation  
                }  
                pointer {  
                    type = string  
                    optional = true  
                    title = JsonPath pointer to place in the body that produces this result  
                }  
                detail {  
                    type = string  
                    optional = true  
                    title = Detail of the validation  
                }  
                validator {  
                    type = string  
                    optional = true  
                    title = Name of the validator that produces the result  
                }  
            }  
        }  
    }  
}
```

For custom validations, it is up to the validation author to decide whether to stop and return at the first error, or to continue checking and return all errors. The built-in validator will try to execute as many validations as possible.

Custom Operations

BPO supports definition and execution of **Custom Operations** in addition to the [Lifecycle Operations](#). Custom operations are always implemented as [Imperative Plans](#).

TIP

Prior versions of this document referred to custom operations as "resource operations". The terminology has changed to reduce ambiguity, as lifecycle operations also operate on resources. The capitalized term "Resource Operation" used in this document indicates the objects returned by the `/resources/{resourceId}/operations` Market API.

A Custom Operation is a procedure performed on a resource after its activation process finishes (the process may be successful or failed). The procedure takes user parameters and the state of the resource when it started as inputs. The operation can change the resource's state as it performs by observed-patching the host resource via market API, and can also produce outputs when it finishes. The outputs are saved on a Resource Operation object attached to the Resource.

NOTE

Custom Operations are only supported for templated Resources. RA-managed resources and other built-in types (such as Number Pools or Monitor) may not have custom operations.

Components of Custom Operations

Interfaces

An interface defines the `inputs` and `outputs` parameters of an operation. The same interface can be implemented by different plans to perform it differently. An interface is defined in the resource type definition and is inherited and can be overridden by a sub-type.

For more details on defining interfaces, see [Interface Definitions](#).

Implement Interfaces with plans in a Service Template

A Service Template implementing a Resource Type can optionally implement one or more interfaces defined on the Resource Type. This is done by creating plan objects in the `plans` section in the Service Template, with each having the same name as the interface to be implemented by the plan. Only the interfaces that are actually be implemented by a Service Template are available on a resource activated by that Service Template.

For more details on implementing interfaces, see [Plans in Service Templates](#).

Imperative Plans for Execution

When a Resource Operation is created against an interface for a Resource, the corresponding imperative plan defined on the Service Template will be called to perform the operation.

For details on how to write a script to implement a custom operation, see [the Imperative Plan reference](#).

Typical Workflow

After a resource finishes its activation, an instance of a Resource Creation can be created against an interface of the Resource Type implemented by the Service Template. The following illustrates a typical workflow.

Query for available interfaces

The `/resources/{resourceId}/interfaces` Market API can be called to query the available interfaces for a resource.

The following is an example of the API's response with one interface called "provision".

```
{
  "items": [
    {
      "id": "provision",
      "title": "Provision the Virtual Circuit",
      "description": "Provision the Virtual Circuit",
      "inputs": {
        "bandwidth": {
          "title": "Desired Bandwidth",
          "description": "Desired bandwidth to provision",
          "type": "integer",
          "output": false,
          "config": true,
          "updatable": false,
          "obfuscate": false,
          "store": true,
          "optional": false,
          "fulltext": false,
          "history": true
        }
      },
      "outputs": {
        "path": {
          "title": "New Path",
          "description": "Path after sizing",
          "type": "array",
          "output": true,
          "config": false,
          "updatable": false,
          "obfuscate": false,
          "store": true,
          "optional": false,
          "fulltext": false,
          "history": true,
          "items": {
            "type": "string"
          }
        }
      }
    }
  ],
  "total": 1,
  "offset": 0,
  "limit": 1000
}
```

Only the interfaces that are both defined in the Resource Type Definition and implemented by the Service Template will be shown in the list of available interfaces.

NOTE

The **/resource-types/{resourceTypeId}** Market API can be called to query for all interfaces defined in the Resource Type, and the **/type-artifacts/{serviceTemplateId}** API can be called to query for all plans in a Service Template

Create and Get a Resource Operation

A POST to the `/resources/{resourceId}/operations` Market API will create a Resource Operation.

The request body must conform to the following JSON schema:

```
{
  interface {
    type = string
    title = Interface name
    description = Name of the interface this operation belongs to
  }
  title {
    type = string
    title = Title
    description = Title of the operation
    optional = true
  }
  description {
    type = string
    title = Description
    description = Description of the operation
    optional = true
  }
  inputs {
    type = object
    title = inputs
    description = Inputs to the operation
    optional = true
  }
  resourceStateConstraints {
    type = object
    title = inputs
    description = Constraints of the resource state for this operation to execute
    optional = true
  }
}
```

NOTE The `interface` must be defined in the Type Definition and implemented by the Service Template. The `inputs` object must also conform to the schema of the interface definition.

The body of the API response represents a Resource Operation object, which conforms to the following schema:

```
{
  id {
    type = string
    title = ID of the operation
    description = Unique identifier for the operation
  }
  resourceId {
    type = string
  }
```

```
    title = ID of the resource
    description = UUID of the resource this operation is executed on
}
interface {
    type = string
    title = Interface name
    description = Name of the interface this operation belongs to
}
title {
    type = string
    title = Title
    description = Title of the operation
    optional = true
}
description {
    type = string
    title = Description
    description = Description of the operation
    optional = true
}
inputs {
    type = object
    title = inputs
    description = Inputs to the operation
    optional = true
}
outputs {
    type = object
    title = outputs
    description = Output of the operation
    optional = true
}
state {
    type = string
    title = State of the operation
    enum = [requested, pending, executing, successful, failed, cancelled]
}
reason {
    type = string
    title = reason
    description = Reason for state of the operation
    optional = true
}
resourceStateConstraints {
    type = object
    title = Resource state constraints
    description = Constraints on the host resource for this operation to execute
    optional = true
}
createdAt {
    type = string
    title = Time created
    description = The time this operation was created
}
updatedAt {
    type = string
    title = Time updated
    description = The time this operation was last updated
}
```

```
}
```

The following is an example of an Resource Operation when created (shown as bpocore-cli output):

Attribute	Value
createdAt	"2016-10-22T05:28:29.582Z"
description	"Provision 10Gbps"
id	"580af8fd-e3a4-4fc8-a74d-b452dd99fffc6"
inputs.bandwidth	10000
interface	"provision"
reason	" "
resourceId	"580af8e5-218a-4796-b604-08ca2477065e"
state	"requested"
title	"Provision 10Gbps"
updatedAt	"2016-10-22T05:28:29.582Z"

The state of the operation can be retrieved with a GET request to the **/resources/{resourceId}/operations/{operationId}** Market API. The following shows an example of the Operation in successful state with outputs populated when the plan completes successfully:

market> resources 580af8e5-218a-4796-b604-08ca2477065e operations 580af8fd-e3a4-4fc8-a74d-b452dd99fffc6 get	
Attribute	Value
createdAt	"2016-10-22T05:28:29.582Z"
description	"Provision 10Gbps"
id	"580af8fd-e3a4-4fc8-a74d-b452dd99fffc6"
inputs.bandwidth	10000
interface	"provision"
outputs.path	["e3dd5dd7-278a-4fc2-a399-bd941b0ec0c1", "e3dd5dd7-278a-4fc2-a399-bd941b0ec0c3", "e3dd5dd7-278a-4fc2-a399-bd941b0ec0c2"]
reason	" "
resourceId	"580af8e5-218a-4796-b604-08ca2477065e"
state	"successful"
title	"Provision 10Gbps"
updatedAt	"2016-10-22T05:28:33.050Z"

Resource Operation Lifecycle

The following diagram illustrates the possible state transitions of a Resource Operation:

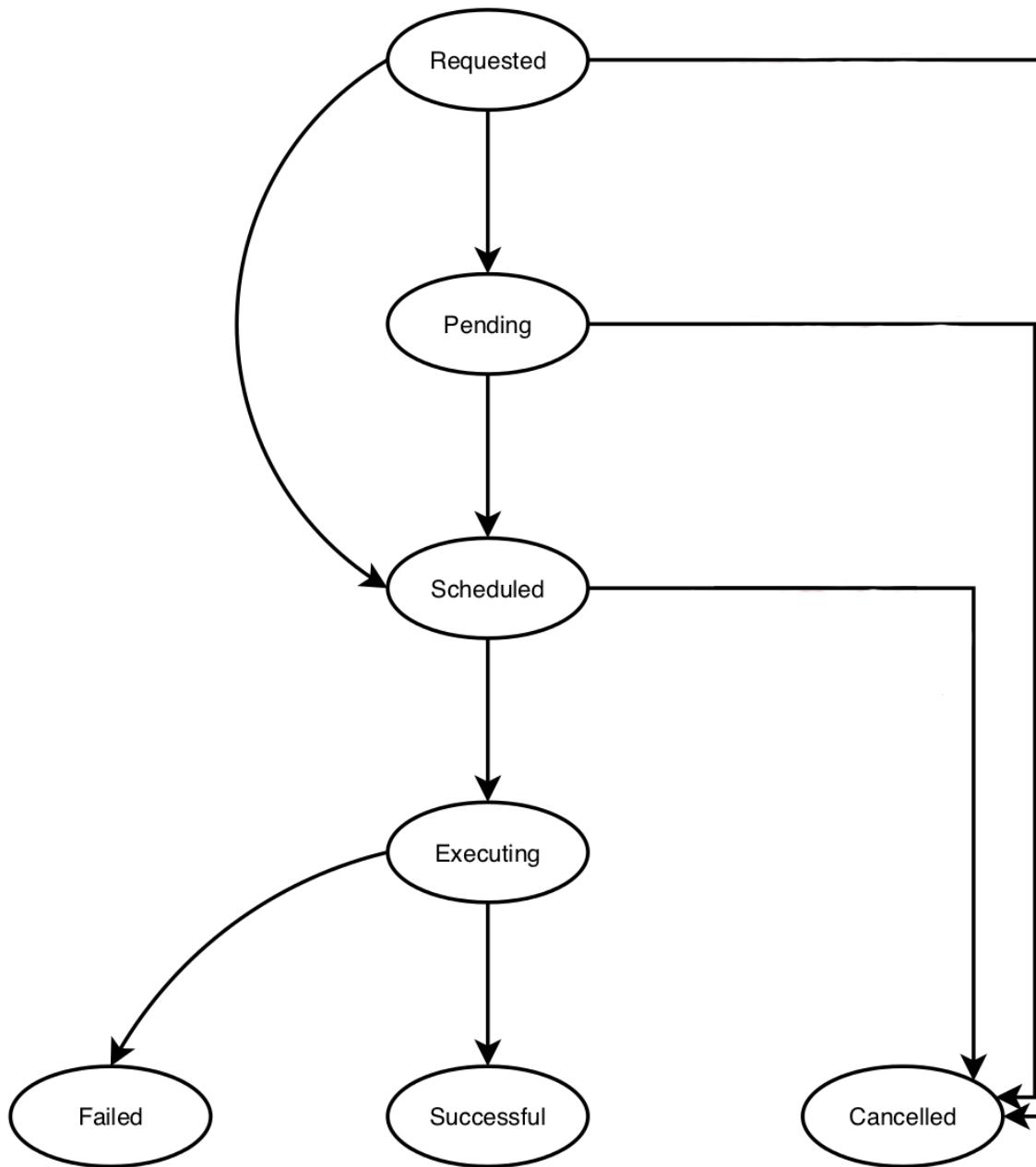


Figure 12. Resource Operation States

When a Resource Operation is created, it will be in **Requested** state. If there is no other (lifecycle or custom) operation being executed, the state will move to **Scheduled** immediately. Otherwise, the Operation will be put into a queue and wait for the currently executing and other operations in the queue to finish execution.

When the Resource Operation enters the **Scheduled** state, a check against the resource will be done to make sure the resource is still valid for the Operation to be executed. If the check passes, the Operation will enter **Executing** state and the corresponding script will be called. See the next section for details on constraints checking.

Depending on the script's result, the Resource Operation will be put in either `Successful` or `Failed` state, with `outputs` or `reason` optionally set.

Resource State Constraints

Since a Resource Operation may not be executed immediately after it is created, it is possible that the resource state has changed between the creation and execution time of the Operation. If a Resource Operation should only be executed when the resource is at a specific state, constraints can be specified using `resourceStateConstraints` in the body of the POST API call to create the Operation.

ETag constraint

If the caller wants to execute the Operation only if the Resource has not changed since the creation of the Operation, the ETag constraint can be specified as the following

```
{
  // ...
  "resourceStateConstraints": {
    "ETag": "current"
  }
}
```

The return value of the POST call will have the `ETag` set to the actual ETag of the Resource at the moment, as the following:

Attribute	Value
<code>createdAt</code>	"2016-10-22T07:49:55.867Z"
<code>description</code>	"Provision 10Gbps"
<code>id</code>	"580b1a23-a969-48a9-9381-717acde8ebd9"
<code>inputs.bandwidth</code>	10000
<code>interface</code>	"provision"
<code>reason</code>	" "
<code>resourceId</code>	"580af8e5-218a-4796-b604-08ca2477065e"
<code>resourceStateConstraints.ETag</code>	"13194139535417"
<code>state</code>	"requested"
<code>title</code>	"Provision 10Gbps"
<code>updatedAt</code>	"2016-10-22T07:49:55.867Z"

The ETag value will be compared against the ETag of the Resource when the Operation enters the `Scheduled` state. If the Resource has been changed in any way after the Operation is created, its ETag will change and as a result would not match the value on the Operation. The Operation will therefore be `Cancelled` in this case.

NOTE BPO currently does not expose the ETag explicitly through the API, so the only valid value for the ETag constraint is "current". Exposing the ETag explicitly will be an enhancement in future releases.

Explicit constraints

An Operation with an ETag constraint will be cancelled due to any change in the resource, which may be too strict in some cases. Alternatively, the caller can specify equality constraints explicitly using JSON path. For example, if the Operation should only be executed when the Resource is in `active` orchState, with the "currentBW" property set to 20000 and the "path" property set to a specific array of nodes, the following constraints can be specified:

```
{  
    // ...  
    "resourceStateConstraints": {  
        "orchState": "active",  
        "properties.currentBW": 20000  
        "properties.path": [ "node1", "node2", "node3" ]  
    }  
}
```

If the Resource is not in the state specified by the constraints when the Operation is scheduled, the Operation will be cancelled, as the following example shows

Attribute	Value
createdAt	"2016-10-22T08:01:19.893Z"
description	"Provision 10Gbps"
id	"580b1ccf-2a79-4ae7-bc17-d70ce6889e22"
inputs.bandwidth	10000
interface	"provision"
reason	"Resource has properties.currentBW' set to 10000, different from constraint value 20000"
resourceId	"580af8e5-218a-4796-b604-08ca2477065e"
resourceStateConstraints.orchState	"active"
resourceStateConstraints.properties.currentBW	20000
resourceStateConstraints.properties.path	["node1", "node2", "node3"]
state	"cancelled"
title	"Provision 10Gbps"
updatedAt	"2016-10-22T08:01:19.940Z"

Policy Enforcement

BPO supports the creation and enforcement of policies. The existing policy types are:

- [Authorization Policies](#) - policies to allow or deny the access to BPO REST APIs.
- [Event Policies](#) - policies to monitor the BPO event bus and to react to specific events.

Table of Contents

- [Common Policy Structure](#)
- [Policy System](#)
- [Policy Condition Definition Language](#)
 - [Authorization Policy Fields](#)
- [Authorization Policies](#)
 - [Default Policies](#)
 - [Authorization Process](#)
 - [Installation of an Authorization Policy](#)
- [Event Policies](#)
 - [Monitor Resource](#)
 - [Monitor Event Extraction Syntax](#)
 - [Monitor TOSCA definition](#)
 - [Event Formats](#)
- [Policy Database](#)

Common Policy Structure

Each policy is made up of three parts:

- Trigger - indicates when a policy is to be evaluated.
- Condition - defines the logic to be evaluated to determine if the policy conditions are met.
- Action - indicates the resulting action to take if the policy conditions are met.

A policy is defined within a **realm** that determines the scope or context to which the policy applies. For example, each component within the system (e.g. Market) has a defined realm allowing policies to be defined on a per-component basis. At present the Asset Manager realm ("**Assets**") and the Market realm ("**Market**") have defined policies. Both of these realm's API access are controlled by authorization policies. Event policies are associated with the Market realm.

Policies are owned and apply to the tenant which created the policy.

Policy System

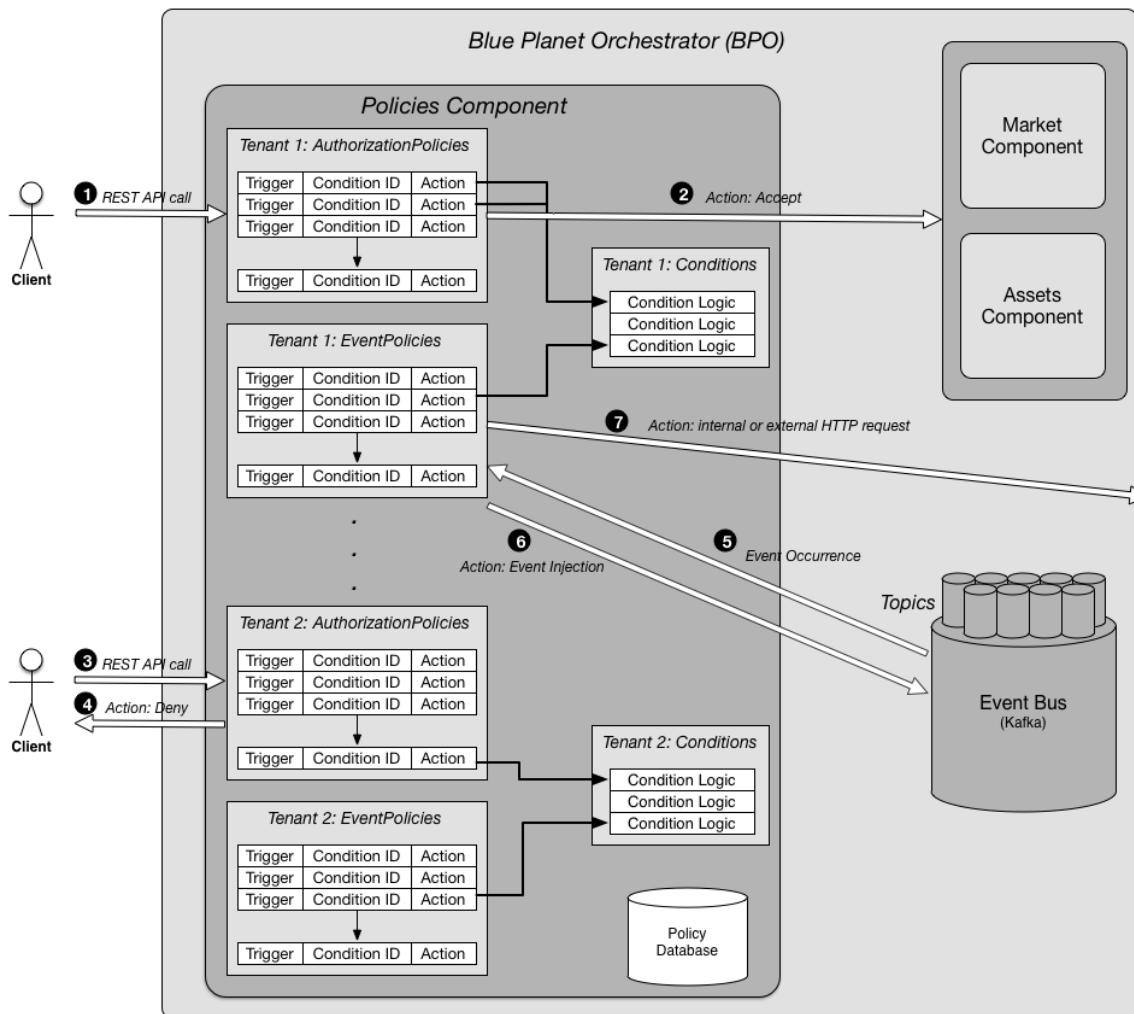


Figure 13. Policy System

The diagram above shows at a high level the policy system:

- Policies and their conditions are stored in a policy database. Evaluation and storage is on a per tenant basis.
- When triggered, the policy engine evaluates the applicable policies and their conditions to determine if an action should be taken.
- Multiple policies may refer to the same condition logic. A typical example is where many authorization policies require the user to have an admin role.
- Condition logic may refer to other conditions to allow further sharing of common condition logic.

For authorization policies (sequences 1,2 and 3,4 in diagram):

- REST API calls into BPO act as triggers to evaluate the authorization policies.
- The policy name is associated with a specific REST API operation (e.g. GetDomain). Multiple policies may apply against the same REST operation and their logic is combined to determine if the overall policy conditions are met. See [Authorization Policy Fields](#) for a list of the policy names.
- The possible actions are to accept or deny the REST API call to BPO.

For event policies (sequence 5,6,7 in diagram):

- Events from the event bus act as triggers. A specific event policy states the event topic and optionally the event type to trigger on.
- The possible actions are to send out an HTTP message and/or emit an event on the event bus.
- Policies and conditions can be created automatically through instantiating a BPO [Monitor Resource](#) in a service template. A typical use of a monitor is to look for changes on a resource in the market (e.g. via monitoring the alarm event topic) and signaling (via HTTP) the market to take action (e.g. autoheal, autoscale, etc).
- Policies and conditions can also be installed via BPO REST API calls.

Policy Condition Definition Language

Policy conditions define the logic that determines if a triggered policy applies. Conditions are defined with their own language. It supports:

- boolean operators: (&&, ||, !, true, false)
- comparisons: ==, !=, <, >, \>=
- arithmetic: +, -, *, /
- integer and real numbers: e.g. 1, -2, 10., 10.0, 10e2, 10e-2, -10e+2
- string literals: e.g. "admin"
- references to other conditions: e.g. cond:11111111-2222-3333-4444-555555555555

for authorization policies:

- references to the REST API caller's common "user" information: e.g. user:role
- references to the REST API call's "request" body common information: e.g. request:resourceTypeIid
- references to the REST API call's "target" common information for the intended target(s): e.g. target:domainType
- references to the REST API call's "request" or "target" resource specific information via JSONPath syntax (e.g. target:properties\$.segmentationId).

NOTE

See event policies note below on the possible need for a trailing '#' in the JSONPath expression.

for event policies:

- references to the event bus's event information via JSONPath syntax: e.g. event:\$severity

NOTE

Complex policy condition JSONPath parsing (other than dotted subfields i.e. `$field1.field2`) at present requires a trailing '#' delimiter. The trailing '#' is optional for simple dotted subfield uses. A policy condition should not mix the two forms with the use of the trailing '#' being preferable.

```
event:$diff[?(@.op==replace)].path# == "/properties/hypervisorId"
```

further supports:

- standard operator precedence rules
- grouping with parentheses:

```
target:resourceTypeId == "openstack.resourceTypes.VirtualMachine" &&
(user:role == "admin" || user:role == "operator")
```

- checking on the existence of a field before accessing it via JSONPath syntax to check for optional information:

```
exists(event:$field7) && event:$field7 == 42 ||
exists(event:$field3) && event:$field3 > 100
```

Notes:

- references to "user", "target", "request" and "event" information take one of two forms:
 - if the field is common to all messages it is accessed directly, e.g. `user:role`.
 - if the field is specific to the request, target or event then it is accessed via JSONPath syntax, e.g. `event:$severity` or `request:properties$.vlan` (see [JSONPath](#) for details). In the case of event JSON Paths their root (\$) is the event portion of a event bus message. In the case of request or target information the JSON Path root (\$) starts at the "properties" attribute which is unique per resource type.
- for the set of user, target and request fields available for use in authorization policies see [Authorization Policy Fields](#).

Example condition logic:

- check that a REST API caller has admin role and is trying to set an OpenStack vlan-based virtual network in the range 100 to 200 (to be used by an authorization Accept policy):

```
(user:role == "admin") &&
(request:resourceTypeId == "openstack.resourceTypes.Network") &&
(request:properties$.segmentationId >= 100 &&
 request:properties$.segmentationId <= 200)
```

- check that a REST API call is not allowed on an OpenStack VM in a specific zone (to be used by an authorization Deny policy):

```
(target:resourceTypeId == "openstack.resourceTypes.VirtualMachine") &&
(target:properties$.zone == "restrictedZone")
```

- check that an event is for a service affecting alarm on a vport:

```
(event:$._type == "bp.v1.AlarmEvent") &&
(event:$._op == "set") && (event:$._serviceAffecting == "SERVICE_AFFECTING") &&
(event:$._nativeConditionTypeQualifier == "vport")
```

Authorization Policies

Authorization policies restrict access to BPO's REST API calls.

From a policy perspective:

- the **trigger** is the reception of a BPO REST API call.
- the **conditions** referred to by the applicable policies can make use of the requesting "user:" information, the "target:" resource information and the information in the REST API "request:" call.
- the **action** is to either Accept or Deny the REST request.

The name of an authorization policy is tied to the REST API call it authorizes. See [Authorization Policy Fields](#) for a list of valid policy names and the accessible user, target and request fields.

Default Policies

Policies and their conditions are created and apply to a specific tenant. If no authorization policy exists for a specific REST API operation on that tenant then a set of default policies are consulted. Use the UUID **956345eb-0949-3aa2-8575-72ad9d5f9a61** to designate that the tenant for the policy is unspecified. The default policies may be changed via REST calls. Samples of portions of the default policies and conditions files are show below.

Policies:

```

} , {
  "id" : "73912345-bdb8-4249-bd47-717455278580",
  "policyType": "authorization",
  "name": "CreateDomain",
  "authRealm": "Market",
  "tenantId": "956345eb-0949-3aa2-8575-72ad9d5f9a61",
  "description": "Create a Domain",
  "conditionId": "63912345-bdb8-4249-bd47-818455278570",
  "action": "Accept"
} , {

```

Conditions:

```

} , {
  "id" : "63912345-bdb8-4249-bd47-818455278570",
  "name": "role_is_admin",
  "tenantId": "956345eb-0949-3aa2-8575-72ad9d5f9a61",
  "description": "User must have admin role",
  "definition" : "user:role == \"admin\""
} , {

```

There are a set of default conditions and policies that are loaded at startup.

Authorization Process

Multiple authorization policies may exist for the same REST API operation and may contain a mixture of Accept and Deny actions. Conditions are stand alone entities and may be reused by different policies.

When triggered by the receipt of a REST API call to BPO the following processing takes place:

1. Gather all the applicable authorization policies for the requesting tenant. These will match the "name" (i.e. REST API operation) and realm.
2. If there are no such policies then gather the default policies that apply.
3. If there are still no policies then deny the request.
4. If there is a gathered policy with a Deny action whose condition is satisfied then deny the request.
5. If there are gathered policies with Accept actions then at least one must be satisfied or else deny the request.
6. Or else allow the request.

The HTTP response code for a denied request is dependent on the type of request:

- operations on a single resource (e.g. Get, Create, ...) result in a 403 Forbidden response.
- operations on a group of resources (e.g. List) result in a 200 OK response but the response only contains the resources that passed the policy checks.

Policy denials are recorded in BPO's audit logs.

Installation of an Authorization Policy

The following shows the use of the Swagger tool to make REST calls to BPO to install a new condition and a policy referring to the condition. The policy could have referred to an existing condition.

First install the condition. Note the definition can be arbitrarily complex and follows the condition language mentioned earlier. The tenantId should match that of the creating user.

POST /conditions

Response Class (Status 200)

Model Model Schema

Response Content Type application/json

Parameters

Parameter	Value
conditionRequest	{ "name": "admin role", "description": "must have admin role", "id": "1234", "definition": "user:role == \"admin\"", "tenantId": "956345eb-0949-3aa2-8575-72ad9d5f9a61" }

Parameter content type: application/json

Figure 14. Create Condition

The response:

Request URL
<code>http://localhost:8181/bpocore/policies/api/v1/conditions</code>
Response Body
<pre>{ "id": "1234", "name": "admin role", "tenantId": "956345eb-0949-3aa2-8575-72ad9d5f9a61", "description": "must have admin role", "definition": "user:role == \"admin\" " }</pre>
Response Code
<code>201</code>

Figure 15. Create Condition Response

Then create the policy. Note:

- The "name" field must match the REST Operation. See [Authorization Policy Fields](#) for a list of valid policy names.
- The "policyActions" field should have a value of "Accept" or "Deny".
- The "policyType" field should have the value of "authorization".
- The "conditionId" field matches the earlier created condition.
- The "tenantId" must match the earlier created condition and that of the creating user.

POST /policies

Implementation Notes
Create a policy in the policies

Response Class (Status 200)

Model Model Schema

Response Content Type application/json

Parameters

Parameter	Value
policyRequest	{ "name": "CreateDomain", "description": "Create a Domain only if admin role", "policyActions": "Accept", "authRealm": "Market", "filter": [], "policyType": "authorization", "id": "4444", "conditionId": "1234", "tenantId": "956345eb-0949-3aa2-8575-72ad9d5f9a61" }

Parameter content type: application/json

Figure 16. Create Policy

The response:

Request URL
<code>http://localhost:8181/bpocore/policies/api/v1/policies</code>
Response Body
<pre>{ "id": "4444", "policyType": "authorization", "name": "CreateDomain", "authRealm": "Market", "tenantId": "956345eb-0949-3aa2-8575-72ad9d5f9a61", "description": "Create a Domain only if admin role", "conditionId": "1234", "policyActions": "Accept", "filter": [] }</pre>
Response Code
<code>201</code>

Figure 17. Create Policy Response

Event Policies

Event policies monitor BPO's event bus for events that satisfy applicable policies.

From a policy perspective:

- The **trigger** is the arrival of an event on an event bus topic/channel which has policies defined against it.
- The **conditions** referred to by the applicable policies can make use of the "event:" information.
- The **action** is to send an HTTP request and/or an event bus message.

An event policy provides a reaction mechanism to external or internal events for use in autohealing, autoscaling or similar functions. The event policy may be installed via a BPO REST call to the policy component or more typically is installed via the creation of a built-in Monitor resource.

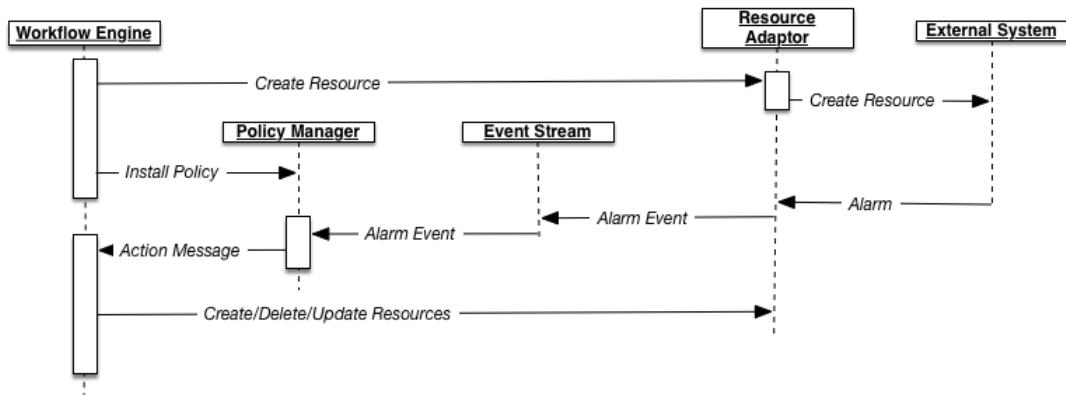


Figure 18. Typical Workflow

The typical workflow is show above:

1. the Template Engine executes a TOSCA template that implements a service. It:
 - a. creates a resource on an external system
 - b. installs an event policy (via a Monitor resource) which monitors the external resource for specific events (e.g. an alarm)
2. the external system at some point raises an alarm on the external resource
3. the Policy Manager sees the alarm via the alarm event stream
4. the event policy is evaluated and its condition logic is satisfied. The triggered action sends a BPO HTTP API request back to the market to update the service.
5. the update of the service triggers autohealing, autoscaling or some such function.

Monitor Resource

BPO has a built-in Monitor resource that is used in imperative or declarative TOSCA templates to install an event policy. It captures all the information required by the event policy including the `actionInfo` and `monitoredTopic` fields.

The code snippet below shows the creation of a Monitor resource from an imperative script. Some of the syntax shown is from the python environment in which it runs:

```

name = "{}_Monitor".format(label)
mon_info = self.create_active_resource(
    "firefly_monitor", resource_id, {
        "label": name,
        "productId": monitor_product_id,
        "properties": {
            "topic": "bp.ra.v1.alarms",
            "condition": """event:$severity == "MAJOR" """,
            "actionInfo": {
                "transport": "HTTP",
                "httpTransportInfo": {
                    "method": "PATCH",
                    "url": "%s/bpocore/market/api/v1/" +
                        "resources/%s" % (self.uri, resource_id),
                    "headers": [
                        { "name": "bp-user-id", "value": self.user },
                        { "name": "bp-tenant-id", "value": self.tenant },
                        { "name": "bp-role-ids", "value": self.role },
                    ],
                    "body": {
                        "contentType": "application/json",
                        "contents": """{
                            "properties": {
                                "triggerVnfMove": "[[event:$time]]"
                            }
                        }"""
                    }
                }
            }
        }
    }
)

```

The example above shows:

- monitoring of the Resource Adaptor alarm event bus topic (bp.ra.v1.alarms)
- looking for alarms flagged as MAJOR (i.e. the condition) using JSONPath access to the event fields
- reacting (i.e. the action) with a BPO HTTP PATCH request to the relevant resource/service (via the calculated url)
- forming a request body that indicates a move operation should take place on the VNF with a timestamp that is extracted from the event fields via JSONPath

Note:

- The double square brackets allowed in the "url" and body "contents" fields flags text that should be replaced with extracted data from the event (similar to how event data can be referenced in the condition logic). (e.g. "url": "http://blueplanet/bpocore/market/api/v1/resources/[[\$event:\$resource.id]]"). See [Monitor Event Extraction Syntax](#) for more details.
- The "transport" field has been deprecated (i.e. is optional and ignored) as both an HTTP request and an event bus event may now be emitted by one policy action.

The code snippet below shows the creation of a Monitor resource from a declarative script. It also shows the emitting of an event bus event along with the HTTP request.

```
serviceTemplates {  
  
    MonitorExample {  
        title = Declarative Monitor Example  
        description = Declarative Monitor Example  
        implements = test.resourceTypes.MonitorExample  
  
        resources {  
  
            resource1 {  
                type = test.resourceTypes.ExampleResource  
                properties {  
                    field1 = {getParam = field1}  
                }  
                activateAfter = [ ]  
            }  
  
            monitor {  
                type = tosca.resourceTypes.Monitor  
                properties {  
                    topic = "com.cyaninc.bp.events"  
                    condition = """event:$._type == "com.cyaninc.bp.events.Heartbeat"""  
                    actionInfo = {  
                        eventTransportInfo = {  
                            topic = "bp.example"  
                            eventType = "example_event"  
                            contents = """{"field1": "[[event:$.sequenceNr]]"}"""  
                        }  
                        httpTransportInfo = {  
                            method = PATCH  
                            url = {join = [ "/", {getContext = uri},  
"market/api/v1/resources",  
                                {get resourceId = resource1}]}  
                            headers = [  
                                { name = bp-user-id, value = {getContext = userId} }  
                                { name = bp-tenant-id, value = {getContext = tenantId} }  
                                { name = bp-role-ids, value = {getContext = roleIds} }  
                            ]  
                            body = {  
                                contentType = "application/json"  
                                contents = """{ "properties":  
                                    {"field1": "[[event:$.sequenceNr]]"} }"""  
                            }  
                        }  
                    }  
                }  
                activateAfter = [ resource1 ]  
                terminateBefore = [ resource1 ]  
            }  
        }  
    }  
}
```

Note:

- The "contents" field of the eventTransportInfo also allows the double square bracket syntax to indicate that event information should be substituted. The event information is identified using JSON Path syntax similar to how it is done in the condition logic. See [Monitor Event Extraction Syntax](#) for more details.
- In the example above the eventTransportInfo field emits the event on a non-existent topic. The policy component will never create a new topic but rather will wait until the owner of the topic creates it before any events will be emitted.
- The topic being monitored ("com.cyaninc.bp.events") emits a Heartbeat message every 10 seconds and is a good event to test the monitor logic when in development.
- The "headers" fields in the httpTransportInfo provide the context required when calling the market component from within the blue planet system. They populate the user, tenant and role fields which normally are filled in when an external request is authenticated.
- Either one or both of eventTransportInfo and httpTransportInfo information must be present for a valid monitor definition.
- The example shows how a resource ("resource1") and a monitor which calls back to update it are declaratively defined.

A REST message example of installing a Monitor is shown below. This was taken from the bpocore.restserver.log log file when a Monitor was created on the BPO UI.

```
2016-07-21 22:49:54,251 INFO r.R.c.c.b.m.MarketRestService$Resources$  
Http Method: POST  
URI: http://localhost:8181/market/api/v1/resources  
Protocol: HTTP/1.0  
x-bpo-auth: {"token": "3a79ad4b-f20a-349b-b06e-87309637fd3f", "userId": "admin",  
x-bpo-api-version: v1  
x-bpo-resource: es  
x-bpo-component: market  
c-trace: 154298f3-872b-444f-b1b5-ebd1eb1ba170  
Remote-Address: 127.0.0.1  
Host: localhost:8181  
Connection: close  
Content-Length: 570  
Accept: application/json, text/javascript, */*;q=0.01  
Origin: http://localhost:8888  
X-Requested-With: XMLHttpRequest  
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_3) AppleWebKit/537.36  
Content-Type: application/json  
Referer: http://localhost:8888/index_dev.html  
Accept-Encoding: gzip, deflate  
Accept-Language: en-US, en;q=0.8  
Entity: {  
    "label": "ex_monitor",  
    "productId": "ca28602-8263-11e5-8bcf-feff819cdc9f",  
    "tenantId": "53beb5ea-ca9a-48fc-8431-398425b1c674",  
    "properties": {  
        "actionInfo": {  
            "eventTransportInfo": {  
                "topic": "bp.example",
```

```

    "eventType" : "example_event",
    "contents" : "{\"field1\": \"[[event:$._sequenceNr]]\"}"
},
"httpTransportInfo" : {
    "method" : "POST",
    "url" : "http://127.0.0.1:80"
    "body" : {
        "contentType" : "application/json",
        "contents": "{\"field1\": \"[[event:$._sequenceNr]]\"}"
    }
}
},
"topic" : "com.cyaninc.bp.events",
"condition" : "event:$._type == \"com.cyaninc.bp.events.Heartbeat\""
},
"providerResourceId" : "",
"discovered" : false,
"orchState" : "unkown",
"reason" : "",
"autoClean" : false,
"description" : "example monitor"
}
Response Status: 201 Created
C-Trace: 8d112c70-9364-4e59-b459-7b8f2bbc4fda
Entity: {
    "id" : "5791b402-61cd-45b5-bf91-51f68498f814",
    "label" : "ex_monitor",
    "description" : "example monitor",
    "resourceTypeId" : "tosca.resourceTypes.Monitor",
    "productId" : "ca286602-8263-11e5-8bcf-feff819cdc9f",
    "tenantId" : "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738",
    "shared" : false,
    "properties" : {
        "topic" : "com.cyaninc.bp.events",
        "condition" : "event:$._type == \"com.cyaninc.bp.events.Heartbeat\"",
        "actionInfo" : {
            "eventTransportInfo" : {
                "topic" : "bp.example",
                "eventType" : "example_event",
                "contents" : "{\"field1\": \"[[event:$._sequenceNr]]\"}"
            },
            "httpTransportInfo" : {
                "method" : "POST",
                "url" : "http://127.0.0.1:80",
                "body" : {
                    "contentType" : "application/json",
                    "contents": "{\"field1\": \"[[event:$._sequenceNr]]\"}"
                }
            }
        }
    },
    "discovered" : false,
    "differences" : [ ],
    "desiredOrchState" : "active",
    "orchState" : "requested",
    "reason" : "",
    "tags" : { },
    "providerData" : { },
    "updatedAt" : "2016-07-22T05:49:54.224Z",
}

```

```

    "createdAt" : "2016-07-22T05:49:54.224Z",
    "autoClean" : false
}
```

Monitor Event Extraction Syntax

Information may be extracted from the triggering event using the JSONPath syntax (see [JSONPath](#) for use in forming the `actionInfo.eventTransportInfo.contents`, `httpTransportInfo.url` or `httpTransportInfo.body.contents` fields. The extracted data may be further processed using `split`, `join`, `lstrip` and `rstrip` operations.

NOTE Unlike the policy condition definition JSONPath handling no trailing '#' delimiter is required for complex JSONPath expressions.

NOTE At present the `split`, `join`, `lstrip` and `rstrip` operations may not be nested within one another.

The examples that follow assume the following kafka event snippet as the trigger:

```
{
  "version":1,
  "header":{ ... },
  "event":{
    "_type":"testEvent",
    "resources": [
      {"resource":"1234", "title":"resource1"},
      {"resource":"5678", "title":"resource2"}
    ],
    "subnet": "192.168.121.0/24",
    "split1": "abc, def",
    "strip1": " ghi "
  }
}
```

Extractions that return multiple elements

By definition, a JSONPath query always returns an array of results. This is true even if there is exactly one match. An event extraction returns the first element of the array if it starts with "event:" and the whole array if it starts with "event::" (i.e. double colons).

Example of returning first element, 1234, from a query that returns multiple elements:

```
[[event:$..resource]]
```

Example of returning all elements ["1234" , "5678"] from a query:

```
[[event::$..resource]]
```

Split Operation

A **split** operation may be performed on the extracted data. It returns a positional part from a string split by a field separator.

Syntax:

```
[[split( <string-to-split>, <field-separator>, <positional--index> )]]
```

where:

- **<string-to-split>** is the string or extraction that will be split by the delimiter provided. It must be a string value.
- **<field-separator>** is the delimiter used by the split operation. It must be a string value. It may be a regular expression if the string is followed by a lowercase *r* after the quotes.
- **<positional-index>** is the index into the array of strings that result when the split operation is applied on the **<string-to-split>**. It must be a positive or negative integer and is zero-based. Negative integers are indexed from the end of the resulting array of split values with -1 referencing the last element.

Examples:

```
[[split(event:$._subnet, "/", 1)]]
[[split(event:$._split1, "\s*,\s*"r, -1)]]
```

which would yield 24 for the first example and def for the second example.

Join Operation

A **join** operation may be performed to concatenate an arbitrary number of strings or extracted data items into one string using a "glue" string between the components.

Syntax:

```
[[join( <glue-string>, <string1>, <string2>, ... )]]
```

where: * **<glue-string>** is the string that will be placed between the values provided. It can be an empty string " " or it can be any other string value. * **<stringX>** are the strings that are to be concatenated. There must be a minimum of two strings to concatenate.

Examples:

```
[[join(":", event:$resources[0].title, event:$resources[0].resource)]]
```

which would yield `resource1:1234`.

Lstrip Operation

An `lstrip` operation may be performed to remove preceding whitespace from extracted data.

Syntax:

```
[[lstrip( <string-to-strip> )]]
```

where: * `<string-to-strip>` is the string whose preceding whitespace is to be removed.

Examples:

```
[[lstrip(event:$strip1)]]
```

which would yield `'ghi '`.

Rstrip Operation

An `rstrip` operation may be performed to remove trailing whitespace from extracted data.

Syntax:

```
[[rstrip( <string-to-strip> )]]
```

where: * `<string-to-strip>` is the string whose trailing whitespace is to be removed.

Examples:

```
[[rstrip(event:$strip1)]]
```

which would yield `' ghi'`.

Monitor TOSCA definition

The TOSCA definition of a Monitor including the expected format of the `actionInfo` field is shown below:

```
Monitor {
    derivedFrom = tosca.resourceTypes.Root
    title = "Monitor"
    description = """
        A virtual resource capable of monitoring a message bus event stream and
        sending a message to a destination when a certain event is seen.
    """
    properties {

        topic {
            title = "Event bus topic"
            description = "The event bus topic to be monitored. (e.g.
bp.ra.v1.alarms)"
            type = string
        }

        eventType {
            title = "Event type"
            description = """
                Optional event bus event type to filter on. (e.g. bp.v1.AlarmEvent)
            """
            type = string
            optional = true
        }

        condition {
            title = "Event condition"
            description = """
                The condition logic to monitor on the received events.
                (e.g. event:$severity == \"MAJOR\" )
            """
            type = string
        }

        actionInfo {
            title = "Action"
            description = """
                Messaging action information used when an incoming event satisfies the
                policy conditions.
            """
            type = object
            default = {}
            properties {
                transport {
                    title = "Message transport"
                    description = "DEPRECATED: The message transport protocol to use"
                    type = string
                    enum = [ "HTTP" ]
                    optional = true
                }

                httpTransportInfo {
                    title = "HTTP message transport information"
                    description = "Required information to send an HTTP message."
                    type = object
                    optional = true
                    default = {}
                    properties {
                        method {
```

```
title = "HTTP method"
description = "The HTTP method for HTTP message"
type = string
enum = ["POST", "PATCH", "PUT", "DELETE"]
optional = true // default = "PATCH"
}

url {
    title = "HTTP URL"
    description = "The destination URL for HTTP message"
    type = string
    format = url
    optional = true // keep UI-schema optional parent object happy
}

headers {
    title = "HTTP Headers"
    description = "The destination URL for HTTP message"
    type = array
    optional = true
    items = {
        type = object
        properties {
            name {
                title = "HTTP header name"
                description = "The HTTP header name"
                type = string
            }
            value {
                title = "HTTP header content"
                description = "The HTTP header content"
                type = string
            }
        }
    }
}

body {
    title = "HTTP Body"
    description = "The HTTP message body content"
    type = object
    optional = true
    properties {
        contentType {
            title = "Content-Type"
            description = "Value of HTTP Content-Type"
            type = string
            optional = true // default = "application/json"
        }
        contents {
            title = "Contents"
            description = """
                Body Contents. For an application/json contentType is a
                well
                formed JSON object.
                """
            type = string
        }
    }
}
```

```

        optional = true // default = "{}"
    }
}
}
}

eventTransportInfo {
    title = "Event message transport information"
    description = "Required information to send an 'event' on event bus."
    type = object
    optional = true
    default = {}
    properties {
        topic {
            title = "Event bus topic"
            description = """
                The event bus topic to send the event on (e.g. bp.ra.v1.alarms)
"""
            type = string
            optional = true // keep UI-schema optional parent object happy
        }

        eventType {
            title = "Event Type"
            description = """
                Value of event._type entry for the event (e.g.
bp.v1.AlarmEvent).
"""
            type = string
            optional = true // keep UI-schema optional parent object happy
        }

        contents {
            title = "Contents"
            description = """
                A well formed JSON object containing the 'event' portion of a
message
                queue event message (i.e. does not contain version or header
fields of
                an event). The _type field will be added automatically from the
eventType setting.
                e.g. {\"key1\": \"value1\", \"key2\": \"value2\"}
"""
            type = string
            optional = true // keep UI-schema optional parent object happy
        }
    }
}
}

policyInfo {
    title = "Policy Info"
    description = """
        Policy ID and Condition ID used in policy manager to monitor event bus
"""
    type = object
    output = true
}

```

```

        properties {
          policyId {
            title = "Policy ID of monitor policy"
            description = "The policy ID of the policy used in monitoring"
            type = string
          }

          conditionId {
            title = "Condition ID used by monitor policy"
            description = "The condition ID used by policy for monitoring"
            type = string
          }
        }
      }
    }
  }
}

```

Event Formats

There are various ways to determine the event bus topics that exist and their message formats:

- use the BPO Event REST API (also available in the Swagger tool) to learn of the event topics and event formats.
- use the kafka command line tools to learn of the topics and events that have been emitted on them:

```

# enter kafka docker container from the blueplanet host
$ docker ps | grep kafka
$ docker exec -it <docker container id> bash

# source some zookeeper environment variables as kafka makes use of zookeeper
$ source /dev/shm/zookeeper.sh

# list the current set of kafka topics
$ /opt/kafka/bin/kafka-topics.sh --list --zookeeper
$KAFKA_ZOOKEEPER_CONNECT/kafka

# dump all the events on a specific topic from the beginning of the collection
$ /opt/kafka/bin/kafka-console-consumer.sh --from-beginning \
--topic com.cyaninc.bp.events --zookeeper $KAFKA_ZOOKEEPER_CONNECT/kafka

```

Policy Database

Policies and conditions are persisted in the policy database.

A condition contains the following fields:

FIELD	TYPE	DESCRIPTION
id	UUID	BPO's identifier for the condition
name	String	Name given to the condition

FIELD	TYPE	DESCRIPTION
tenantId	UUID	Tenant ID that created the condition
description	String	Optional description for the condition
definition	String	Actual condition logic in condition definition language

A policy contains the following fields:

FIELD	TYPE	DESCRIPTION
id	UUID	BPO's identifier for the policy
name	String	Name given to the policy. For authorization policies it must match a known REST API operation
authRealm	String	Realm to which the policy applies. A choice from: "Market", "Assets"
policyType	Enumeration	Type of policy. Either "authorization" or "event"
tenantId	UUID	Tenant ID that created the policy
description	String	Optional description for the policy
conditionId	UUID	Reference to applicable condition to evaluate
action	Enumeration	Action to perform. "Accept" or "Deny" for authorization policies or "Message" for event policies
filter	Set of Strings	Specific to policy type. Filters fields returned in REST responses for "authorization" policies. Optionally filters the events to be evaluated on "event" policies
monitoredTopic	String	Specific to "event" policies it chooses the event bus channel to monitor for events
actionInfo	String	Specific to "event" policies it defines the message to send if the policy action is invoked

Authorization Policy Fields

This section lists the information available for use in authorization policy condition definitions.

To be evaluated the policy name must match the REST API operation. In general the mapping of Policy Names to REST verbs is:

POLICY NAME PREFIX	REST VERB
Create	POST
List	GET
Get	GET
Update	PUT
Patch	PATCH
Delete	DELETE

For example:

- an HTTP GET of all Domains in the market (path of /bpocore/market/api/v1/domains) has a policy name of "ListDomains".
- an HTTP GET of an individual Domain in the market (path of /bpocore/market/api/v1/domains/<UUID of Domain>) has the policy name of "GetDomain".

The Request/Target field tables that follow list the policy name in the first column.

User Fields

These fields are available in all REST API requests and represent information about the user making the request.

The fields are accessed as "user:`field`".

FIELD NAME	TYPE	COMMENTS
role	String	e.g. user:role == "admin"
tenantId	UUID	e.g. user:tenantId == "40985920-c137-11e5-9912-ba0be0483c18"
userId	UUID	e.g. user:userId == "40985920-c137-11e5-9912-ba0be0483c20"

Request and Target Fields

The request and target fields vary based on the actual REST API call.

Request fields are found in the body of the REST API call request. These fields are accessed as "request:'field'".

Target fields are contained within the body of the REST API call responses. These fields are accessed as "target:'field'".

Realm: Assets, bpocore/asset-manager/api/v1/...

POLICY NAME	REST PATH	TGT/REQ	FIELD NAME	TYPE	NOTES
ListAreas	/areas	target	externalGitUrl	URL	
		target	internalGitUrl	URL	
		target	name	String	
GetArea	/areas/{areaName}				a1
GetVersionSignature	/areas/{areaName}?fields=versionSignature				a2
GetReleaseSignature	/areas/{areaName}?fields=releaseSignature				a2
GetUpgradeAvailable	/areas/{areaName}?fields=upgradeAvailable				a2
GetHeadCommitHash	/areas/{areaName}?fields=commitHash				a2
GetGitUri	/areas/{areaName}?fields=uri				a2
ListRecentChanges	/areas/{areaName}/changes	target	changeId	String	
ListFiles	/areas/{areaName}/files	target	path	String	
GetFile	/areas/{areaName}/files/{path}	target	path	String	
CreatePullRequest	/areas/{areaName}/pullrequests				
ListPullRequests	/areas/{areaName}/pullrequests	target	requestId	String	
GetPullRequest	/areas/{areaName}/pullrequests/{requestId}	target	requestId	String	
GetFileBytes	/areas/{areaName}/raw-files/{path}				
CreateAreaUpgrade	/areas/{areaName}/upgrades				
ListAreaUpgrades	/areas/{areaName}/upgrades	target	area	String	
GetAreaUpgrade	/areas/{areaName}/upgrades/{requestId}	target	area	String	

POLICY NAME	REST PATH	TGT/REQ	FIELD NAME	TYPE	NOTES
AddAuthorizedKey	/keys	request	id	String	
ListAuthorizedKeys	/keys	target	id	String	
RemoveAuthorizedKey	/keys/{keyId}	target	id	String	

Notes:

- a1) No policy of its own. Controlled by next 5 policies that represent data that can be returned. All 5 fields returned if no URL 'fields' parameters present.
- a2) Sub-authorization of GetArea

Realm: Market, bpocore/market/api/v1/...

POLICY NAME	REST PATH	TGT/REQ	FIELD NAME	TYPE	NOTES
ListDomainTypes	/domain-types	target	name	String	
CreateDomain	/domains	request	properties	JSONObject	m1
		request	rpld	String	
		request	title	String	
ListDomains	/domains	target	domainType	String	
		target	id	String	
		target	properties	JSONObject	m1
		target	rpld	String	
		target	tenantId	String	
		target	title	String	
GetDomain	/domains/{domainId}	target	domainType	String	
		target	id	String	
		target	properties	JSONObject	m1
		target	rpld	String	
		target	tenantId	String	
		target	title	String	
UpdateDomain	/domains/{domainId}	target	domainType	String	
		target	id	String	

POLICY NAME	REST PATH	TGT/REQ	FIELD NAME	TYPE	NOTES
		target	properties	JObject	m1
		target	rpld	String	
		target	tenantId	String	
		target	title	String	
		request	domainType	String	
		request	id	String	
		request	properties	JObject	m1
		request	rpld	String	
		request	tenantId	String	
		request	title	String	
PatchDomain	/domains/{domainId}	target	domainType	String	
		target	id	String	
		target	properties	JObject	m1
		target	rpld	String	
		target	tenantId	String	
		target	title	String	
DeleteDomain	/domains/{domainId}	target	domainType	String	
		target	id	String	
		target	properties	JObject	m1
		target	rpld	String	
		target	tenantId	String	
		target	title	String	
ListProductsBy Domain	/domains/{domainId}/products	target	domainId	UUID	
		target	id	UUID	
		target	resourceTypeld	URI	
CreateProduct	/products	request	domainId	UUID	
		request	id	UUID	
		request	resourceTypeld	URI	
ListProducts	/products	target	domainId	UUID	

POLICY NAME	REST PATH	TGT/REQ	FIELD NAME	TYPE	NOTES
		target	id	UUID	
		target	resourceTypeld	URI	
GetProduct	/products/{productld}	target	domainId	UUID	
		target	id	UUID	
		target	resourceTypeld	URI	
UpdateProduct	/products/{productld}	target	domainId	UUID	
		target	id	UUID	
		target	resourceTypeld	URI	
		request	domainId	UUID	
		request	id	UUID	
		request	resourceTypeld	URI	
PatchProduct	/products/{productld}	target	domainId	UUID	
		target	id	UUID	
		target	resourceTypeld	URI	
DeleteProduct	/products/{productld}	target	domainId	UUID	
		target	id	UUID	
		target	resourceTypeld	URI	
CreateRelationship	/relationships				
ListRelationships	/relationships				
DeleteRelationship	/relationships/{relationshipId}				
CreateResourceProvider	/resource-providers				
ListResourceProviders	/resource-providers				
GetResourceProvider	/resource-providers/{resourceProviderId}				
UpdateResourceProvider	/resource-providers/{resourceProviderId}				

POLICY NAME	REST PATH	TGT/REQ	FIELD NAME	TYPE	NOTES
PatchResourceProvider	/resource-providers/{resourceProviderId}				
DeleteResourceProvider	/resource-providers/{resourceProviderId}				
ListDomainsByRp	/resource-providers/... ... {resourceProviderId}/domains	target	domainType	String	
		target	id	String	
		target	properties	JObject	m1
		target	rpld	String	
		target	tenantId	String	
		target	title	String	
ListResourceTypes	/resource-types	target	experimental	Boolean	
		target	id	String	
		target	title	String	
		target	version	String	
GetResourceType	/resource-types/{resourceTypeIdd}	target	experimental	Boolean	
		target	id	String	
		target	title	String	
		target	version	String	
ListProductsByResource... ... Type	/resource-types/{typeld}/products	target	domainId	UUID	
		target	id	UUID	
		target	resourceTypeld	URI	
CreateResource	/resources	request	productId	UUID	
		request	properties	JObject	m1
ListResources	/resources	target	id	UUID	
		target	orchState	String	m2
		target	orderId	UUID	

POLICY NAME	REST PATH	TGT/REQ	FIELD NAME	TYPE	NOTES
		target	productId	UUID	
		target	properties	JObject	m1
		target	resourceTypeld	URI	
		target	tenantId	UUID	
GetResourceBy Provider... ... Resourceid	/resources	target	id	UUID	
		target	orchState	String	m2
		target	orderId	UUID	
		target	productId	UUID	
		target	properties	JObject	m1
		target	resourceTypeld	URI	
		target	tenantId	UUID	
GetResource	/resources/{re sourceld}, /resources/{re sourceld}/obse rved	target	id	UUID	
		target	orchState	String	m2
		target	orderId	UUID	
		target	productId	UUID	
		target	properties	JObject	m1
		target	resourceTypeld	URI	
		target	tenantId	UUID	
UpdateResourc e	/resources/{re sourceld}, /resources/{re sourceld}/obse rved	target	id	UUID	
		target	orchState	String	m2
		target	orderId	UUID	
		target	productId	UUID	
		target	properties	JObject	m1
		target	resourceTypeld	URI	
		target	tenantId	UUID	
		request	id	UUID	

POLICY NAME	REST PATH	TGT/REQ	FIELD NAME	TYPE	NOTES
		request	orchState	String	m2
		request	orderId	UUID	
		request	productId	UUID	
		request	properties	JObject	m1
		request	resourceTypeId	URI	
		request	tenantId	UUID	
PatchResource	/resources/{resourceId}, /resources/{resourceId}/observed	target	id	UUID	
		target	orchState	String	m2
		target	orderId	UUID	
		target	productId	UUID	
		target	properties	JObject	m1
		target	resourceTypeId	URI	
		target	tenantId	UUID	
DeleteResource	/resources/{resourceId}	target	id	UUID	
		target	orchState	String	m2
		target	orderId	UUID	
		target	productId	UUID	
		target	properties	JObject	m1
		target	resourceTypeId	URI	
		target	tenantId	UUID	
ListDependenciesOf... ... Resource	/resources/{resourceId}/... ... dependencies	target	id	UUID	
		target	orchState	String	m2
		target	orderId	UUID	
		target	productId	UUID	
		target	properties	JObject	m1
		target	resourceTypeId	URI	
		target	tenantId	UUID	

POLICY NAME	REST PATH	TGT/REQ	FIELD NAME	TYPE	NOTES
ListDependentsOf... ... Resource	/resources/{resourceId}/dependents	target	id	UUID	
		target	orchState	String	m2
		target	orderId	UUID	
		target	productId	UUID	
		target	properties	JObject	m1
		target	resourceTypeId	URI	
		target	tenantId	UUID	
ListResourceHistory	/resources/{resourceId}/history				
ListResources History	/resources/history				
CreateTagKey	/tag-keys				
ListTagKeys	/tag-keys	target	key	String	
		target	autoIndexed	Boolean	
GetTagKey	/tag-keys/{tagKey}	target	key	String	
		target	autoIndexed	Boolean	
UpdateTagKey	/tag-keys/{tagKeyId}	target	key	String	
		target	autoIndexed	Boolean	
		request	key	String	
		request	autoIndexed	Boolean	
PatchTagKey	/tag-keys/{tagKeyId}	target	key	String	
		target	autoIndexed	Boolean	
DeleteTagKey	/tag-keys/{tagKey}	target	key	String	
		target	autoIndexed	Boolean	
CreateTagValue	/tag-keys/{tagKey}/tag-values	request	key	String	
		request	value	String	

POLICY NAME	REST PATH	TGT/REQ	FIELD NAME	TYPE	NOTES
ListTagValues	/tag-keys/{tagKey}/tag-values	target	key	String	
		target	value	String	
GetTagValue	/tag-keys/{tagKey}/tag-values/{tagValue}	target	key	String	
		target	value	String	
UpdateTagValue	/tag-keys/{tagKey}/tag-values/{tagValueId}	target	key	String	
		target	value	String	
		request	key	String	
		request	value	String	
PatchTagValue	/tag-keys/{tagKey}/tag-values/{tagValueId}	target	key	String	
		target	value	String	
DeleteTagValue	/tag-keys/{tagKey}/tag-values/{tagValue}	target	key	String	
		target	value	String	
ListTypeArtifacts	/type-artifacts	target	experimental	Boolean	
GetTypeLayerVersion	/type-artifacts/realm	target	version	String	
GetTypeArtifact	/type-artifacts/{typeArtifactId}	target	experimental	Boolean	

Notes:

- m1) JSONPath access, e.g. request:properties\$.field
- m2) Value: "unknown", "unspecified", "requested", "scheduled", "activating", "failed", "active", "inactive", "terminating", "terminated"

Tutorials

This is a collection of tutorials for bpocore. The intention is help you familiarize yourself with bpocore APIs and workflows.

Requirements

Please follow the [Setup Instructions for the Tutorials](#) to prepare your environment.

Overview of the Tutorials

Getting Started

If this your first time using bpocore see the [Hello World Tutorial](#).

Beginner

These tutorials are intended for beginners who have already done the [Hello World Tutorial](#) and who want to see more examples of interacting with bpocore.

These examples continue using curl, git, and python to interact with bpocore:

- [Bootstrap Steps for bpocore-dev and Environment](#) – a few common steps referenced in the beginner tutorials
- [Web Callout Tutorial](#) – an example of making web callouts from inside a service template
- [Allocating Numbers in a Number Pool Tutorial](#) – demonstrates creating and using a number pool to manage the allocation of integer resource values

Intermediate

These tutorials build on the beginner material by introducing higher level abstractions for working with bpocore APIs.

- [bpocore CLI Bootstrap Tutorial](#) – a few common steps referenced in the bpocore CLI based tutorials
- [Using bpocore CLI for Fast Onboarding](#) – an introduction to the bpocore CLI tool
- [Developing Remote Plans with script-dev](#) – how to use bpocore-dev, script-dev, and bpocore-cli to develop remote imperative plans.
- [First OpenStack Tutorial](#) – a first excursion into how to manage an OpenStack domain
- [How to Reuse Property Types in Resource Type Definitions](#) – shows one way of reusing property type definitions inside a given TOSCA file (HOCON-based)

Advanced

These are more advanced examples, including Resource Provider (RP) development:

- [Service Template Tutorial](#) – creating declarative- and imperative-style Service Templates
- [Multi-Tenant Tutorial: Managing Customers as Separate Tenants](#) – creating a service with imperative-style plans executed in `scriptplan` to create a tenant and domain for a new customer
- [Hello RP Tutorial](#) – an example of creating a Python-based RP to integrate with `bpoCore`

Setup Instructions for the Tutorials

The tutorials assume familiarity with running command line tools in a Linux environment. They are written against the current Ubuntu LTS release.

On other platforms we recommend using the Blue Planet DevOps Toolkit. For more information, see blueplanet.com.

You should have the following commands available before continuing:

- ssh
- curl
- python
- docker
- git

Docker Installation

On Ubuntu 16.04 you can install an appropriate version of Docker (1.12) from the package repositories:

```
sudo apt install docker.io
```

For other platforms (including earlier Ubuntu releases), we recommend using the DevOps Toolkit, but you may also be able to [install Docker from upstream](#).

Git Setup

- Make sure you have Git installed and configured:

```
sudo apt install git
```

- Make sure you have Git configured with your username and e-mail address:

```
git config --global user.name "<YOUR NAME HERE>"  
git config --global user.email "<YOUR EMAIL HERE>"
```

Python Setup

- Make sure you have the necessary Python packages installed:

```
sudo apt install python-virtualenv python-dev build-essential
```

Bootstrap Steps for bpocore-dev and Environment

This tutorial shows the prerequisite steps required for most of the other tutorials.

Unless specified otherwise, the commands below and those in the tutorial should be run in the same terminal.

Steps

1. If this is the very first time you have used bpdr.io

```
docker login \
--password="secret" \
--username="bp2" \
--email="bp2@ciena.com" \
bpdr.io
```

2. Run bpocore in a terminal separate from the other steps

```
docker pull bpdr.io/blueplanet/bpocore-dev:1.7.0-8289-bcff141
docker run --name bpocore-dev -it --rm bpdr.io/blueplanet/bpocore-dev:1.7.0-
8289-bcff141
```

3. Set up constants, aliases, environment, etc.

```
function jq { python -c "import sys,json; print json.load(sys.stdin)$1"; }
BPOCORE_CID=$(docker ps | grep "bpocore-dev" | cut -f 1 -d ' ')
DOCKER_BRIDGE_IP=$( docker inspect bpocore-dev | jq
'[0]["NetworkSettings"]["Gateway"]' )
BPOCORE_IP=$( docker inspect bpocore-dev | jq
'[0]["NetworkSettings"]["IPAddress"]' )
MARKET_URL="http://$BPOCORE_IP:8181/bpocore/market/api/v1"
ASSETS_URL="http://$BPOCORE_IP:8181/bpocore/asset-manager/api/v1"
alias jcurl='curl -s -H "Content-Type: application/json"'
alias jpp='python -m json.tool'
ssh-keygen -R $BPOCORE_IP
ssh-keyscan -H $BPOCORE_IP >> ~/.ssh/known_hosts
```

- Note: since each `docker run ...` invocation typically assigns a new IP address to the container's network interface(s) you SHOULD re-run this step each time you kill and re-run the container.

4. Upload your public key

```
PUBLIC_KEY=`cat ~/.ssh/id_rsa.pub`
jcurl -d "{\"key\": \"$PUBLIC_KEY\"}" $ASSETS_URL/keys | jpp
```

- Note: since each `docker run . . .` invocation starts with a clean filesystem (i.e, unaffected by modifications made in previous runs) you SHOULD re-run this step each time you kill and re-run the container.

5. Clone the definitions repository and create a private branch from the production branch

```
BRANCH_NAME="mybranch"
git clone ssh://git@$BPOCORE_IP:22/home/git/repos/model-definitions
DEFINITIONS_DIR=$PWD/model-definitions
(cd $DEFINITIONS_DIR; git checkout -b $BRANCH_NAME origin/production)
```

- See note(s) in previous steps about re-running this step.

Running the `frost-orchestrate-ui-dev` Docker container

The `frost-orchestrate-ui-dev` Docker container pairs with `bpocore-dev` to allow development and testing of UI schema and workflows without a full Blue Planet deployment.

NOTE

Before running `frost-orchestrate-ui-dev` make sure that `bpocore-dev` is running. See [Bootstrap Steps for `bpocore-dev` and Environment](#)

Pull the `frost-orchestrate-ui-dev` Docker image

First pull the `frost-orchestrate-ui-dev` Docker image from the Docker registry:

```
docker pull bpdr.io/blueplanet/frost-orchestrate-ui-dev:1706.4.18
```

`frost-orchestrate-ui-dev` needs to run in its own terminal space. After pulling the image, open a new terminal window and run it with the following command:

```
docker run --rm -it --name frost-orchestrate-ui-dev --link bpocore-dev:bpocore-dev  
bpdr.io/blueplanet/frost-orchestrate-ui-dev:1706.4.18
```

After the `frost-orchestrate-ui-dev` container is running, you'll need to inspect it and discover its IP address to access it from your browser, run the following command to get this address:

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}' frost-orchestrate-ui-dev
```

Enter this IP address into your browser in order to access the Orchestrate-UI application. For example, if the container IP address is 172.16.0.3, enter the following URL into your browser:

```
http://172.16.0.3/orchestrate
```

Accessing the Orchestrate-UI from a Vagrant Box

The IP address of the `frost-orchestrate-ui-dev` container is in the private IP address space used for the Docker bridge between containers. If you are running the browser on a different machine, it needs to have the ability to route to the private IP address. This happens, for example, when running the Docker containers inside of a Vagrant box while accessing the UI from a browser running on the host machine. You can make the private IP address routable by adding a route in the routing table that sets the Vagrant IP address as the next-hop.

For example, assume 192.168.56.5 is the address of the `vboxnet0` interface on the host and 172.16.0.0/16 is the private IP subnet used for the Docker bridge.

```
sudo route add 172.16.0.0/16 192.168.56.5
```

NOTE

If your Vagrant box doesn't have a host-only interface, you might need to add one and then add the route to that interface.

The route will only be active until your host is rebooted. You can permanently add the route. Commands and procedures vary based on the host OS.

bpocore CLI Bootstrap Tutorial

This tutorial shows the prerequisite steps for the other bpocore CLI based tutorials.

Unless specified otherwise, the commands below and those in the tutorial should be run in the same terminal.

For more details on bpocore CLI see [Orchestrate/bpocore-cli](#).

Prerequisites

1. The user must have a private/public key pair in the `~/.ssh` directory. If the key pair isn't configured, create one:

```
$ ssh-keygen -b 2048 -t rsa -f id_rsa
```

2. The user must have configured user name and email information for `git`. Check the configuration:

```
$ git config -l | grep user
user.name=Joe
user.email=jorchester@cienna.com
```

If `user.name` and `user.email` are not configured, configure them now:

```
git config --global user.email "jorchester@cienna.com"
git config --global user.name "Joe Orchestrator"
```

Steps

1. Install the bpocore CLI tool on your system

```
virtualenv env
source env/bin/activate
pip install -i https://artifactory.cienna.com/api/pypi/blueplanet-pypi/simple
bpocore-cli
bpocore --help
```

2. Run bpocore in a terminal separate from the other steps (**choose appropriate bpocore-dev version number based on the applicable release**)

```
docker pull bpdr.io/blueplanet/bpocore-dev:1.7.0-8289-bcff141
docker run --name bpocore-dev -it --rm bpdr.io/blueplanet/bpocore-dev:1.7.0-8289-bcff141
```

3. Create a default site configuration file

```
bpocore init-config --local-docker bpocore-dev/8181
```

There are other init-config modes that are useful. See `bpocore init-config --help` for more details.

NOTE Since each `docker run . . .` invocation typically assigns a new IP address to the container's network interface you SHOULD re-run this step each time you kill and re-run the container.

4. Fetch the identity of the host serving the Assets component

```
bpocore assets pull-identity
```

This command will run `ssh-keyscan` against the host where the Assets component is running. This will add a hashed entry to the specified `known_hosts` file.

NOTE Since each `docker run . . .` invocation typically assigns a new IP address to the container's network interface you SHOULD re-run this step each time you kill and re-run the container.

5. Push your identity to the Assets component

```
bpocore assets push-identity
```

NOTE Since each `docker run . . .` invocation starts with a clean filesystem (i.e, unaffected by modifications made in previous runs) you must re-run this command each time you kill and re-run the container.

Hello World Tutorial

This tutorial serves as the "Hello World" example for bpocore. If you complete this tutorial you will know how to:

- run bpocore in "dev-ops" mode
- onboard and instantiate service templates
- instantiate, query, and teardown resource instances

Prerequisites

Some minimal setup is required to have an environment in which you can follow the steps below. See [Setup Instructions for the Tutorials](#).

Steps

1. If this is the very first time you have used bpdr.io

```
docker login --password="secret" --username="bp2user" \
--email="bp2@ciena.com" bpdr.io
```

2. Run bpocore in a terminal separate from the other steps

```
docker pull bpdr.io/blueplanet/bpocore-dev:1.7.0-8289-bcff141
docker run --name bpocore-dev -it --rm bpdr.io/blueplanet/bpocore-dev:1.7.0-
8289-bcff141
```

3. Set up constants, aliases, environment, etc.

```
BPOCORE_CID=$(docker ps | grep "bpocore-dev" | cut -f 1 -d ' ')
BPOCORE_IP=$(docker inspect --format '{{ .NetworkSettings.IPAddress }}' \
bpocore-dev)
MARKET_URL="http://$BPOCORE_IP:8181/bpocore/market/api/v1"
ASSETS_URL="http://$BPOCORE_IP:8181/bpocore/asset-manager/api/v1"
BRANCH_NAME="helloworld"
ssh-keygen -R $BPOCORE_IP
ssh-keyscan -H $BPOCORE_IP >> ~/.ssh/known_hosts
```

4. Upload your public key

```
PUBLIC_KEY=`cat ~/.ssh/id_rsa.pub`
curl -s -H "Content-Type: application/json" \
-d "{\"key\": \"$PUBLIC_KEY\"}" $ASSETS_URL/keys | python -m json.tool
```

5. Clone the definitions repository and checkout a private branch from the **production** branch

```
git clone ssh://git@$BPOCORE_IP:22/home/git/repos/model-definitions  
DEFINITIONS_DIR=$PWD/model-definitions  
cd $DEFINITIONS_DIR  
git checkout -b $BRANCH_NAME origin/production
```

6. Add your service template

```
cd $DEFINITIONS_DIR/types/tosca
mkdir -p example
cat <<EOF > example/hello_world.tosca
"\$schema"      = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-
module#"
title          = "A very simple service built from a template"
package        = example
version        = "1.0"
description    = """Defines a Hello World service template."""
authors        = ["Royal Joe (royal.joe@ciena.com)"]

resourceTypes {

    HelloWorld {
        title = Hello World
        description = "Hello World"
        derivedFrom = tosca.resourceTypes.Root
        properties {
            property1 {
                title = Property 1
                description = "Some property of the service"
                type = string
            }
            property2 {
                title = Property 2
                description = "Some other property of the service"
                type = string
            }
        }
    }
}

serviceTemplates {

    HelloWorld {

        title = Hello World
        description = "Hello World"
        implements = example.resourceTypes.HelloWorld

        resources {
            // various random strings modeled as subresources
            dbName.type = tosca.resourceTypes.RandomString
            dbUser.type = tosca.resourceTypes.RandomString
            dbUserPw.type = tosca.resourceTypes.RandomString
            dbRootPw.type = tosca.resourceTypes.RandomString
        }
    }
}
EOF
```

7. Commit and push your changes

```
git add .
git commit -m "Add Hello World definitions"
git push origin $BRANCH_NAME
```

8. Submit a pull request

```
cat <<EOF > pullrequest.json
{
  "branch": "$BRANCH_NAME",
  "title": "Add Hello World definitions",
  "comment": ""
}
EOF
curl -s -H "Content-Type: application/json" \
  -d @pullrequest.json $ASSETS_URL/areas/model-definitions/pullrequests \
  | python -m json.tool
```

Let's record the pull request ID for later use:

```
PULL_REQUEST_ID='<USE "requestId" FROM OUTPUT ABOVE>'
```

9. Wait for the pull request to be "accepted"

```
curl -s $ASSETS_URL/areas/model-definitions/pullrequests/$PULL_REQUEST_ID \
  | python -m json.tool
```

10. Create a corresponding product

```
cat <<EOF > product.json
{
  "resourceTypeId": "example.resourceTypes.HelloWorld",
  "title": "Hello World",
  "active": true,
  "domainId": "built-in",
  "providerData": {
    "template": "example.serviceTemplates.HelloWorld"
  }
}
EOF
curl -s -H "Content-Type: application/json" \
  -d @product.json $MARKET_URL/products | python -m json.tool
```

Let's record the product ID for later use:

```
PRODUCT_ID='<USE "id" FROM OUTPUT ABOVE>'
```

11. Instantiate a resource for the product

```
cat <<EOF > resource.json
{
    "productId": "$PRODUCT_ID",
    "label": "Hello World 0001",
    "properties": {
        "property1": "foo",
        "property2": "bar"
    }
}
EOF
curl -s -H "Content-Type: application/json" -d @resource.json
$MARKET_URL/resources | python -m json.tool
```

Let's record the resource ID for later use:

```
RESOURCE_ID='<USE "id" FROM OUTPUT ABOVE>'
```

12. Query for the resource and its dependencies

```
curl -s $MARKET_URL/resources/$RESOURCE_ID | python -m json.tool
curl -s $MARKET_URL/resources/$RESOURCE_ID/dependencies | python -m json.tool
```

13. Teardown the resource

```
curl -X DELETE -s $MARKET_URL/resources/$RESOURCE_ID
curl -s $MARKET_URL/resources/$RESOURCE_ID | python -m json.tool # THE RESOURCE
WILL BE GONE
```

Example

The output below show the tutorial steps being run on the command line along with their output. This is only an example; the identifiers you see will be different in some cases.

```
#terminal 1

$ docker pull bpdr.io/blueplanet/bpocore-dev:1.7.0-8289-bcff141
Pulling repository bpdr.io/blueplanet/bpocore-dev
f83c82570fa8: Download complete
511136ea3c5a: Download complete
27d47432a69b: Download complete
5f92234dcf1e: Download complete
51a9c7c1f8bb: Download complete
5ba9dab47459: Download complete
c02320a6f792: Download complete
83a4c3577cb8: Download complete
bcd874cca324: Download complete
```

```

d3595ab7f891: Download complete
5f9d73ebfdcb: Download complete
0b53bd122b09: Download complete
c961e9577e85: Download complete
b7caebcbce04: Download complete
7df33a28b8d8: Download complete
2d0e812fc8cb: Download complete
cd06b7d4bc9b: Download complete
31386af49137: Download complete
79783377caf9: Download complete
e95e24a8914a: Download complete
c6caale22c60: Download complete
b0466c5c44db: Download complete
caf82d1bal6f: Download complete
f08d5b6bc355: Download complete
cfb8c63dfa83: Download complete
9b2a6886d8fc: Download complete
994c055178b0: Download complete
4a649e137d9a: Download complete
5324bfe7ae9a: Download complete
c32826eabe3d: Download complete
786e482e13c0: Download complete
Status: Downloaded newer image for bpdr.io/blueplanet/bpocore-dev:1.7.0-8289-
bcff141

$ docker run --name bpocore-dev -it --rm bpdr.io/blueplanet/bpocore-dev:1.7.0-8289-
bcff141

# terminal 2

$ BPOCORE_CID=$(docker ps | grep "bpocore-dev" | cut -f 1 -d ' ')
$ BPOCORE_IP=$(docker inspect --format '{{ .NetworkSettings.IPAddress }}' bpocore-
dev)
$ MARKET_URL="http://$BPOCORE_IP:8181/bpocore/market/api/v1"
$ ASSETS_URL="http://$BPOCORE_IP:8181/bpocore/asset-manager/api/v1"
$ BRANCH_NAME="helloworld"

$ ssh-keygen -R $BPOCORE_IP
/home/gbolt/.ssh/known_hosts updated.
Original contents retained as /home/gbolt/.ssh/known_hosts.old
$ ssh-keyscan -H $BPOCORE_IP >> ~/.ssh/known_hosts
# 172.16.0.119 SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2
# 172.16.0.119 SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2

$ PUBLIC_KEY=`cat ~/.ssh/id_rsa.pub`
$ curl -s -H "Content-Type: application/json" -d "{\"key\": \"$PUBLIC_KEY\"}"
$ ASSETS_URL/keys | python -m json.tool
{
    "id": "c573acf773a6affed69df2ff4c2ef2df",
    "key": "ssh-rsa"
AAAAB3NzaC1yc2EAAAQABAAQDAREvQTFujLmXDZu2H8p1ZKKaLS3JyOMiljV5A3rqWpi21I0PHxjt
4BoieKXnyQMeTRLnMk6J+pb3o3YM3ZzVt0EQbGK06Zk3zRgcdxNvZmi/+/_DbcxquimMGPv4zVoy+rk0Pyu3
abxSWVrrlyX2jxKXyNDrj0zLRhlgsM4GO7PAhrTfAOwjm8yscme6ziC1p3Iadt2fcXcC7qKt174PpnNN3Ez
x/2KYVJro3NNQW0EBpcKu9ns01Qmq7Lng36W1dco/QgwyVdNdsBFMLjDJK9cPMZe5DJocsVsQpFfIdAfvt
UPuA4804K2VV1PxhhtZhUxjCkk6/yiKus1B+etWP gbolt@GBOLT-6530-Ubuntu"
}

$ git clone ssh://git@$BPOCORE_IP:22/home/git/repos/model-definitions
Cloning into 'model-definitions'...

```

```
remote: Counting objects: 303, done.
remote: Compressing objects: 100% (171/171), done.
remote: Total 303 (delta 128), reused 301 (delta 127)
Receiving objects: 100% (303/303), 598.39 KiB | 0 bytes/s, done.
Resolving deltas: 100% (128/128), done.
Checking connectivity... done.

$ DEFINITIONS_DIR=$PWD/model-definitions
$ cd $DEFINITIONS_DIR
$ git checkout -b $BRANCH_NAME origin/production
Branch helloworld set up to track remote branch production from origin.
Switched to a new branch 'helloworld'

$ cd $DEFINITIONS_DIR/types/tosca
gbolt@GBOLT-6530-Ubuntu:/tmp/hello/model-definitions/types/tosca$ mkdir -p example
$ cat <<EOF > example/hello_world.tosca
> "\$schema"      = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-module#"
> title          = "A very simple service built from a template"
> package         = example
> version         = "1.0"
> description     = """Defines a Hello World service template."""
> authors         = ["Royal Joe (royal.joe@ciena.com)"]
>
> resourceTypes {
>
>   HelloWorld {
>     title = Hello World
>     description = "Hello World"
>     derivedFrom = tosca.resourceTypes.Root
>     properties {
>       property1 {
>         title = Property 1
>         description = "Some property of the service"
>         type = string
>       }
>       property2 {
>         title = Property 2
>         description = "Some other property of the service"
>         type = string
>       }
>     }
>   }
> }
>
> serviceTemplates {
>
>   HelloWorld {
>
>     title = Hello World
>     description = "Hello World"
>     implements = example.resourceTypes.HelloWorld
>
>     resources {
>       // various random strings modeled as subresources
>       dbName.type = tosca.resourceTypes.RandomString
>       dbUser.type = tosca.resourceTypes.RandomString
>       dbUserPw.type = tosca.resourceTypes.RandomString
>       dbRootPw.type = tosca.resourceTypes.RandomString
>
```

```

>      }
>    }
>  }
> EOF

$ git add .

$ git commit -m "Add Hello World definitions"
[helloworld 354abec] Add Hello World definitions
 1 file changed, 46 insertions(+)
 create mode 100644 types/tosca/example/hello_world.tosca

$ git push origin $BRANCH_NAME
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 888 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To ssh://git@172.16.0.119:22/home/git/repos/model-definitions
 * [new branch]      helloworld -> helloworld

$ cat <<EOF > pullrequest.json
> {
>   "branch": "$BRANCH_NAME",
>   "title": "Add Hello World definitions",
>   "comment": ""
> }
> EOF

$ curl -s -H "Content-Type: application/json" -d @pullrequest.json
$ASSETS_URL/areas/model-definitions/pullrequests | python -m json.tool
{
  "comment": "",
  "commitDate": "2016-01-28T01:52:58.000Z",
  "commitHash": "354abec5c6285bb1a76cdfac9828f508be929b21",
  "details": [],
  "email": "royal.joe@ciena.com",
  "reason": "",
  "requestDate": "2016-01-28T01:53:35.776Z",
  "requestId": "f3cb359c-0691-4309-822c-06e3d0567be0",
  "status": "pending",
  "title": "Add Hello World definitions"
}

$ PULL_REQUEST_ID=f3cb359c-0691-4309-822c-06e3d0567be0

$ curl -s $ASSETS_URL/areas/model-definitions/pullrequests/$PULL_REQUEST_ID |
python -m json.tool
{
  "comment": "",
  "commitDate": "2016-01-28T01:52:58.000Z",
  "commitHash": "354abec5c6285bb1a76cdfac9828f508be929b21",
  "details": [],
  "email": "royal.joe@ciena.com",
  "productionCommitHash": "354abec5c6285bb1a76cdfac9828f508be929b21",
  "reason": "",
  "requestDate": "2016-01-28T01:53:35.776Z",
}

```

```

    "requestId": "f3cb359c-0691-4309-822c-06e3d0567be0",
    "status": "accepted",
    "title": "Add Hello World definitions"
}

$ cat <<EOF > product.json
> {
>     "resourceTypeId": "example.resourceTypes.HelloWorld",
>     "title": "Hello World",
>     "active": true,
>     "domainId": "built-in",
>     "providerData": {
>         "template": "example.serviceTemplates.HelloWorld"
>     }
> }
> EOF

$ curl -s -H "Content-Type: application/json" -d @product.json $MARKET_URL/products
| python -m json.tool
{
    "active": true,
    "constraints": {},
    "domainId": "built-in",
    "id": "56a974f3-f657-449f-899a-87eb2e9241ed",
    "providerData": {
        "template": "example.serviceTemplates.HelloWorld"
    },
    "resourceTypeId": "example.resourceTypes.HelloWorld",
    "title": "Hello World"
}

$ PRODUCT_ID=56a974f3-f657-449f-899a-87eb2e9241ed

$ cat <<EOF > resource.json
> {
>     "productId": "$PRODUCT_ID",
>     "label": "Hello World 0001",
>     "properties": {
>         "property1": "foo",
>         "property2": "bar"
>     }
> }
> EOF

$ curl -s -H "Content-Type: application/json" -d @resource.json
$MARKET_URL/resources | python -m json.tool
{
    "autoClean": false,
    "createdAt": "2016-01-28T01:55:52.144Z",
    "desiredOrchState": "active",
    "differences": [],
    "discovered": false,
    "id": "56a97528-49e4-4665-ac31-96ff0600081c",
    "label": "Hello World 0001",
    "orchState": "requested",
    "productId": "56a974f3-f657-449f-899a-87eb2e9241ed",
    "properties": {
        "property1": "foo",
        "property2": "bar"
}

```

```
        },
        "providerData": {},
        "reason": "",
        "resourceTypeId": "example.resourceTypes.HelloWorld",
        "shared": false,
        "tags": {},
        "tenantId": "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738",
        "updatedAt": "2016-01-28T01:55:56.144Z"
    }
}

$ RESOURCE_ID=56a97528-49e4-4665-ac31-96ff0600081c

$ curl -s $MARKET_URL/resources/$RESOURCE_ID | python -m json.tool
{
    "autoClean": false,
    "createdAt": "2016-01-28T01:55:52.144Z",
    "desiredOrchState": "active",
    "differences": [],
    "discovered": false,
    "id": "56a97528-49e4-4665-ac31-96ff0600081c",
    "label": "Hello World 0001",
    "orchState": "active",
    "productId": "56a974f3-f657-449f-899a-87eb2e9241ed",
    "properties": {
        "property1": "foo",
        "property2": "bar"
    },
    "providerData": {},
    "reason": "",
    "resourceTypeId": "example.resourceTypes.HelloWorld",
    "shared": false,
    "tags": {},
    "tenantId": "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738",
    "updatedAt": "2016-01-28T01:55:56.200Z"
}

$ curl -s $MARKET_URL/resources/$RESOURCE_ID/dependencies | python -m json.tool
{
    "items": [
        {
            "autoClean": false,
            "createdAt": "2016-01-28T01:55:52.407Z",
            "desiredOrchState": "active",
            "differences": [],
            "discovered": false,
            "id": "56a97528-a449-412a-897f-a221d868d586",
            "label": "Hello World 0001.dbName",
            "orchState": "active",
            "productId": "c9bfce3c-149e-4495-b861-94fe801bdf6e",
            "properties": {
                "format": "hexadecimal",
                "length": 32,
                "value": "a67af51988c1a75a4a5f93298a388741"
            },
            "providerData": {},
            "reason": "",
            "resourceTypeId": "tosca.resourceTypes.RandomString",
            "shared": false,
            "tags": {}
        }
    ]
}
```

```
        "tenantId": "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738",
        "updatedAt": "2016-01-28T01:55:53.070Z"
    },
    {
        "autoClean": false,
        "createdAt": "2016-01-28T01:55:52.433Z",
        "desiredOrchState": "active",
        "differences": [],
        "discovered": false,
        "id": "56a97528-a897-4ebf-98df-487033998629",
        "label": "Hello World 0001.dbUserPw",
        "orchState": "active",
        "productId": "c9bfce3c-149e-4495-b861-94fe801bdf6e",
        "properties": {
            "format": "hexadecimal",
            "length": 32,
            "value": "504b1d528d77d7e94025cccaab7b5cdc"
        },
        "providerData": {},
        "reason": "",
        "resourceTypeId": "tosca.resourceTypes.RandomString",
        "shared": false,
        "tags": {},
        "tenantId": "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738",
        "updatedAt": "2016-01-28T01:55:53.271Z"
    },
    {
        "autoClean": false,
        "createdAt": "2016-01-28T01:55:52.476Z",
        "desiredOrchState": "active",
        "differences": [],
        "discovered": false,
        "id": "56a97528-dd2f-4b2c-9d46-75d1186e59a5",
        "label": "Hello World 0001.dbRootPw",
        "orchState": "active",
        "productId": "c9bfce3c-149e-4495-b861-94fe801bdf6e",
        "properties": {
            "format": "hexadecimal",
            "length": 32,
            "value": "caac32c11565e2a5ae24c8eebacd0e2b"
        },
        "providerData": {},
        "reason": "",
        "resourceTypeId": "tosca.resourceTypes.RandomString",
        "shared": false,
        "tags": {},
        "tenantId": "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738",
        "updatedAt": "2016-01-28T01:55:53.275Z"
    },
    {
        "autoClean": false,
        "createdAt": "2016-01-28T01:55:52.440Z",
        "desiredOrchState": "active",
        "differences": [],
        "discovered": false,
        "id": "56a97528-2743-488b-906f-68fa32cf9f8e",
        "label": "Hello World 0001.dbUser",
        "orchState": "active",
        "productId": "c9bfce3c-149e-4495-b861-94fe801bdf6e",
```

```
        "properties": {
            "format": "hexadecimal",
            "length": 32,
            "value": "1210d637b50df56cd5e96731ce828973"
        },
        "providerData": {},
        "reason": "",
        "resourceTypeId": "tosca.resourceTypes.RandomString",
        "shared": false,
        "tags": {},
        "tenantId": "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738",
        "updatedAt": "2016-01-28T01:55:53.239Z"
    }
],
"offset": 0,
"total": 4
}

$ curl -X DELETE -s $MARKET_URL/resources/$RESOURCE_ID

$ curl -s $MARKET_URL/resources/$RESOURCE_ID | python -m json.tool
{
    "autoClean": false,
    "createdAt": "2016-01-28T01:55:52.144Z",
    "desiredOrchState": "terminated",
    "differences": [],
    "discovered": false,
    "id": "56a97528-49e4-4665-ac31-96ff0600081c",
    "label": "Hello World 0001",
    "orchState": "terminating",
    "productId": "56a974f3-f657-449f-899a-87eb2e9241ed",
    "properties": {
        "property1": "foo",
        "property2": "bar"
    },
    "providerData": {},
    "reason": "",
    "resourceTypeId": "example.resourceTypes.HelloWorld",
    "shared": false,
    "tags": {},
    "tenantId": "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738",
    "updatedAt": "2016-01-28T01:57:01.250Z"
}

$ curl -s $MARKET_URL/resources/$RESOURCE_ID | python -m json.tool
{
    "failureInfo": {
        "detail": "",
        "reason": "Resource '56a97528-49e4-4665-ac31-96ff0600081c' not found"
    },
    "statusCode": 404
}
```

Web Callout Tutorial

This tutorial shows you how to make web callouts from within service templates. A web callout allows you to send an HTTP request to an endpoint from within your service template. The HTTP response can be used to customize the behaviour of the service template.

There are two ways to do web callouts in bpocore:

- The 'Fetch' directive can be used to directly set a property, however it only supports the GET method.
- `tosca.resourceType.WebCallout` allows you to perform more complex HTTP methods such as POST or DELETE.

This tutorial will cover both types.

Steps

1. Bootstrap your environment by following the steps in [Bootstrap Steps for bpocore-dev and Environment](#).
2. Open a new terminal and run an HTTP server. This terminal will be tied up with the HTTP server, so you will need another terminal for the rest of the tutorial.

```
function jq { python -c "import sys,json; print json.load(sys.stdin)$1"; }
GITHOME=https://github.cyanoptics.com/Orchestrate/bpocore-docs
SERVER_PY="raw/master/tutorials/web-callout-simple_web_server.py"
curl -s -L $GITHOME/$SERVER_PY > simple_web_server.py
DOCKER_BRIDGE_IP=$( docker inspect --format '{{ .NetworkSettings.Gateway }}' bpocore-dev )
python simple_web_server.py $DOCKER_BRIDGE_IP 8000
```

Note: The HTTP server will act as the endpoint for your web callouts.

3. Go back to the terminal where you bootstrapped your environment variables, and add your service template.

```
DOCKER_BRIDGE_IP=$( docker inspect --format '{{ raw }}{{ .NetworkSettings.Gateway }}{{ raw }}' bpocore-dev )
cd $DEFINITIONS_DIR/types/tosca
mkdir -p example
cat <<EOF > example/web_callout.tosca
"\$schema"    = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-module#"
title        = "Web Callout Demo"
package      = example
version      = "1.0"
description  = """A sample type and template to demo web callouts."""
authors      = ["Royal Joe (royal.joe@ciena.com)"]

resourceTypes {
```

```
WebCalloutDemo {
    title = Web Callout Demo
    description = """
        Type definition for demonstrating web callouts using a combination of
        techniques.
    """
    derivedFrom = tosca.resourceTypes.Root
    properties {
        nodeNameA {
            title = Node Name A
            description = """
                Node name which we will use as a lookup key against the endpoint
            """
            type = string
        }

        nodeNameZ {
            title = Node Name Z
            description = """
                Node name which we will use as a lookup key against the endpoint
            """
            type = string
        }

        interfaceNameA {
            title = Interface Name A
            description = "Name of the interface on a-end node"
            type = string
        }

        interfaceNameZ {
            title = Interface Name Z
            description = "Name of the interface on z-end node"
            type = string
        }
    }
}

WebCalloutResult {
    title = Web Callout Result
    description = """
        Type definition for demonstrating storage of web callout results.
    """
    derivedFrom = tosca.resourceTypes.Root
    properties {
        aEndHwAddr {
            title = "A-end HWADDR"
            description = "Hardware address of a-end interface"
            type = string
        }

        zEndHwAddr {
            title = "Z-end HWADDR"
            description = "Hardware address of z-end interface"
            type = string
        }
    }
}
```

```

    }

}

serviceTemplates {

    WebCalloutDemo {
        title = Web Callout Demo
        description = """
            Template definition for demonstrating web callouts using a combination of
            techniques.
        """
        implements = example.resourceTypes.WebCalloutDemo
    }

    resources {

        resolveNodes {
            title = "Resolve Node Names"
            type = tosca.resourceTypes.Noop
            properties {
                data {
                    nodeIdA { fetch = [
                        {
                            renderTemplate = [
                                { name = {getParam = nodeNameA} },
                                "http://$DOCKER_BRIDGE_IP:8000/resolve/{name}"
                            ]
                        },
                        "$.items[0].id",
                        true
                    ]}
                    nodeIdZ { fetch = [
                        {
                            renderTemplate = [
                                { name = {getParam = nodeNameZ} },
                                "http://$DOCKER_BRIDGE_IP:8000/resolve/{name}"
                            ]
                        },
                        "$.items[0].id",
                        true
                    ]}
                }
            }
        }

        aEnd {
            type = tosca.resourceTypes.WebCallout
            properties {
                onCreate {
                    method = POST
                    url = { join = [
                        "",
                        "http://$DOCKER_BRIDGE_IP:8000/nodes/",
                        { getAttr = [ resolveNodes, data.nodeIdA ] },
                        "/interfaces"
                    ]}
                    data = {
                        interface {
                            name = {getParam = interfaceNameA }
                        }
                    }
                }
            }
        }
    }
}

```

```

        mbps = 1000
    }
}
onTerminate {
    method = DELETE
    url = { join = [
        "",
        "http://$DOCKER_BRIDGE_IP:8000/nodes/",
        { getAttr = [ resolveNodes, data.nodeIdA ] },
        "/interfaces/",
        { getParam = interfaceNameA }
    ]}
    data = {}
}
}
zEnd {
    type = tosca.resourceTypes.WebCallout
    properties {
        onCreate {
            method = POST
            url = { join = [
                "",
                "http://$DOCKER_BRIDGE_IP:8000/nodes/",
                { getAttr = [ resolveNodes, data.nodeIdZ ] },
                "/interfaces"
            ]},
            data = {
                interface {
                    name = { getParam = interfaceNameZ }
                    mbps = 1000
                }
            }
        }
    }
    onTerminate {
        method = DELETE
        url = { join = [
            "",
            "http://$DOCKER_BRIDGE_IP:8000/nodes/",
            { getAttr = [ resolveNodes, data.nodeIdZ ] },
            "/interfaces/",
            { getParam = interfaceNameZ }
        ]}
        data = {}
    }
}
}
result {
    type = example.resourceTypes.WebCalloutResult
    properties {
        aEndHwAddr = { getAttr = [aEnd, onCreateOutput.hwaddr] }
        zEndHwAddr = { getAttr = [zEnd, onCreateOutput.hwaddr] }
    }
}

```

```
WebCalloutResult {
    title = Web Callout Result
    description = "Placeholder template for storing result of web callouts."
    implements = example.resourceTypes.WebCalloutResult
}
EOF
```

Note how the Fetch directive directly modifies the `tosca.resourceType.Noop` 'data' property whereas `WebCallout` stores the output on the `data` object of the `onCreate` and `onTerminate` properties.

4. Commit and push your changes from your private branch.

```
git add .
git commit -m "Add Web Callout Demo definitions"
git push origin $BRANCH_NAME
```

5. Submit a pull request.

```
cat <<EOF > pullrequest.json
{
  "branch": "$BRANCH_NAME",
  "title": "Add Web Callout Demo definitions",
  "comment": ""
}
EOF
jcurl -d @pullrequest.json $ASSETS_URL/areas/model-definitions/pullrequests |
jpp | tee pr.json
PULL_REQUEST_ID=$(cat pr.json | jq '[{"requestId"}]')
```

6. Wait for the pull request to be "accepted."

```
jcurl $ASSETS_URL/areas/model-definitions/pullrequests/$PULL_REQUEST_ID | jpp
```

7. Create the product for the `WebCalloutDemo`.

```

cat <<EOF > product1.json
{
    "resourceTypeId": "example.resourceTypes.WebCalloutDemo",
    "title": "Web Callout Demo",
    "active": true,
    "domainId": "built-in",
    "providerData": {
        "template": "example.serviceTemplates.WebCalloutDemo"
    }
}
EOF
jcurl -d @product1.json $MARKET_URL/products | jpp | tee product1.out.json
PRODUCT_ID=$(cat product1.out.json | jq '["id"]')

```

8. Create the product for the WebCalloutResult.

```

cat <<EOF > product2.json
{
    "resourceTypeId": "example.resourceTypes.WebCalloutResult",
    "title": "Web Callout Result",
    "active": true,
    "domainId": "built-in",
    "providerData": {
        "template": "example.serviceTemplates.WebCalloutResult"
    }
}
EOF
jcurl -d @product2.json $MARKET_URL/products | jpp

```

9. Instantiate a resource for the product.

```

cat <<EOF > resource.json
{
    "productId": "$PRODUCT_ID",
    "label": "Web Callout Demo 0001",
    "discovered": false,
    "properties": {
        "nodeNameA": "cerulean",
        "nodeNameZ": "aqua",
        "interfaceNameA": "xe0",
        "interfaceNameZ": "xe0"
    }
}
EOF
jcurl -d @resource.json $MARKET_URL/resources | jpp | tee resource.out.json
RESOURCE_ID=$(cat resource.out.json | jq '["id"]')

```

10. Query for the resource and the dependencies. You will see the values from the various web callouts reflected in the sub-resources.

```
jcurl $MARKET_URL/resources/$RESOURCE_ID | jpp  
jcurl $MARKET_URL/resources/$RESOURCE_ID/dependencies | jpp
```

11. Delete the templated resource instance:

```
jcurl -X DELETE $MARKET_URL/resources/$RESOURCE_ID
```

12. After a short while, the resource should be gone and you should see a 404 error:

```
jcurl $MARKET_URL/resources/$RESOURCE_ID | jpp
```

Further Notes

There is a much simpler way to do these kinds of experimental onboardings, using the [bpocore CLI](#). See the [bpocore CLI Bootstrap Tutorial](#).

Allocating Numbers in a Number Pool Tutorial

A Number Pool is a useful product that is offered by the built-in resource provider. For example, an appliance may support a fixed number of sub-units or a domain may have a specific range from which VLANs may be allocated with the requirement that a value may be only allocated once. The Planet Orchestrate built-in resource provider supports managing integer pools and address pools. In this tutorial, you will use REST APIs to:

- define an integer number pool
- allocate a number from the number pool
- query the state of the number pool
- delete a pooled number returning the value to the pool

The **tosca/resource_type_pools.tosca** standard template file defines the resource types from which the products are created that are used for managing pools.

- **NumberPool** - General product for a pool of integer values. Creating a resource instance from the **NumberPool** results in a concrete number pool with a defined range and a corresponding **PooledNumber** product for allocating numbers in the **NumberPool**.
- **PooledNumber** - The product for allocating numbers from a **NumberPool**. Each **PooledNumber** resource is a value allocated from the pool.

Steps

1. Bootstrap your environment following the steps in [Bootstrap Steps for bpocore-dev and Environment](#).
2. Find the standard number pool product in the product catalog.

```
jcurl "$MARKET_URL/products?q=(resourceTypeId:tosca.resourceTypes.NumberPool)" \
| jpp | tee s1.json
NUMBER_POOL_PRODUCT_ID=$(cat s1.json | jq '["items"][0]["id"]')
```

- The REST API filtering syntax is used to find only the product corresponding to the **NumberPool** resource type.
3. Instantiate a number pool resource for the product

```
cat <<EOF > number_pool_resource.json
{
  "label": "NumberPoolOf5",
  "productId": "$NUMBER_POOL_PRODUCT_ID",
  "properties": {
    "lowest": 11,
    "highest": 15
  }
}
EOF
jcurl -d @number_pool_resource.json $MARKET_URL/resources | jpp | tee s2.json
NUMBER_POOL_RESOURCE_ID=$(cat s2.json | jq '[{"id"}]')
```

- Creating a number pool resource requires specifying the range by lowest and highest values.
- The response indicates the resource is in the **requested** orchestration state. A subsequent GET request on the resource will indicate resource has transitioned to the **active** orchestration state.
- Creating this number pool resource results in a new pooled number product being created based on the specification for the resource.

4. Find the new pooled number product

```
jcurl "$MARKET_URL/products?q=providerProductId:$NUMBER_POOL_RESOURCE_ID" \
| jpp | tee s3.json
POOLED_NUMBER_PRODUCT_ID=$(cat s3.json | jq '[{"items"}][0][{"id"}]')
```

- The **providerProductId** will match the number pool resource id created in the previous step and therefore is used as a query parameter in the GET request

5. Allocate a number resource from the pool

```
cat <<EOF > pooled_number_resource.json
{
  "productId": "$POOLED_NUMBER_PRODUCT_ID"
}
EOF
jcurl -d @pooled_number_resource.json $MARKET_URL/resources | jpp | tee s4.json
ALLOCATED_NUMBER_RESOURCE_ID=$(cat s4.json | jq '[{"id"}]')
```

- The **productId** is the minimum data required in the request and will result in a default allocation from the number pool
- To request a specific value from the pool, also specify the **properties.requestedValue** attribute in the request.
- Note, for bpocore<1.2.0 the requested value is specified through the **properties.value** attribute in the request.
- The allocation response will have an **orchState** of **requested** and the allocation is not confirmed until the **orchState** has transitioned to **active**

6. Get the allocated value

```
jcurl $MARKET_URL/resources/$ALLOCATED_NUMBER_RESOURCE_ID | jpp
```

- Obtain the allocated value from the **properties.allocatedValue** attribute in the JSON response. In this case, the value **11** has been allocated from the pool.
- Note, for bpocore<1.2.0 the allocated value is specified through the **properties.value** attribute in the request.
- Also check the **orchState** is **active** to confirm the allocation was successful

7. Get detailed information on the number pool resource

```
jcurl "$MARKET_URL/resources/$NUMBER_POOL_RESOURCE_ID?full=true" | jpp
```

- Get the Number Pool resource with the full option set to true. This will return the resource information showing the number of free numbers in the pool.

8. Return the number to the pool

```
jcurl -X DELETE $MARKET_URL/resources/$ALLOCATED_NUMBER_RESOURCE_ID
```

- After the pooled number resource has terminated, the value is returned to the pool. A GET of the full Number Pool resource will again show that all of the values are free.

```
jcurl "$MARKET_URL/resources/$NUMBER_POOL_RESOURCE_ID?full=true" | jpp
```

How to Reuse Property Types in Resource Type Definitions

There is a natural requirement to be able to reuse a property type definition across multiple properties of the same resource type or even properties across various resource types. While the latter can be partially addressed by resource type inheritance, inheritance has its constraints.

One way to address property type reuse is to rely on HOCON's built-in capabilities, such as the [resolve](#), [substitute](#), and merge features. Once enabled, these allow an already existing property type definition be used repeatedly, as is or with arbitrary overrides, inside the same HOCON file.

This tutorial demonstrates the solution via a simple working example. For simplicity we will use the `bpocore` CLI to efficiently onboard and work with the new resource type.

After completing this tutorial you will know how to speed up type development by reusing existing property definitions.

Steps

1. Run the bootstrap steps inside [bpocore CLI Bootstrap Tutorial](#).
2. Let's create a service template using the technique

```

cat <<"EOF" > property-def-reuse.tosca
"$schema"      = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-module#"
title          = "A very simple example showing property type reuses"
package         = example
version         = "1.0"
description     = """Defines a Hello World service template."""
authors         = [ "Royal Joe (royal.joe@ciena.com)" ]

resourceTypes {

    MyService {
        title = Hello World
        description = "Hello World"
        derivedFrom = tosca.resourceTypes.Root
        properties {

            aEndPort {
                title = A-End
                description = "A-End termination identifier"
                type = string
                enum = [ pt1, pt2, pt3, pt4, pt5, pt6, pt7, pt8, null, loopback ]
            }

            zEndPort = ${resourceTypes.MyService.properties.aEndPort}
            zEndPort.title = Z-End
            zEndPort.description = "Z-End termination identifier"

        }
    }
}

serviceTemplates {

    MyService {
        title = Hello World
        description = "Hello World"
        implements = example.resourceTypes.MyService
        resources {
            // sub-resources are not needed to demo the reuse solution
        }
    }
}
EOF

```

The applicable HOCON features are all in the zEndPort definition lines:

- The 1st line here tells the HOCON parser to assign zEndPort the value (JSON object) found under the provided path "resourceTypes.MyResource.properties.aEndPort", which is the aEndPort property object. At this point zEndPort is defined identically to aEndPort.
- The 2nd and 3rd lines override two of the inherited values: title and description.

Note: the zEndPort block could have been written in another way with identical outcome, but better illustrating the "merge" behavior:

```
zEndPort = ${resourceTypes.MyService.properties.aEndPort}
zEndPort {
    title = Z-End
    description = "Z-End termination identifier"
}
```

But for now we will stay with the style we have.

3. Establish a baseline state which can be restored to later

```
bpocore baseline save
```

4. Onboard your service template and also attempt to create a resource instance

```
bpocore onboard property-def-reuse.tosca --auto-resources \
"example.resourceTypes.MyService:{aEndPort=pt2,zEndPort=foo}"
```

- See `bpocore onboard --help` for more information about the `--auto-products` and `--auto-resources` flags.

5. The previous step has failed, because the provided value `foo` does not comply with the enum definition of `zEndPort`, which was reused from `aEndPort`. Let's try again, this time with a proper value:

```
bpocore onboard property-def-reuse.tosca --auto-resources \
"example.resourceTypes.MyService:{aEndPort=pt2,zEndPort=pt7}"
```

This now succeeded.

Note: This step also illustrates that you can use the `onboard` command repeatedly. It is smart enough to know what it can skip and what it needs to do. In this second run, it correctly detected that the new types have already been onboarded, and the product has been defined, so its task is reduced to a resource instantiation.

6. Roll back to the pre-onboarding state

```
bpocore baseline restore
```

Using bpocore CLI for fast onboarding

The bpocore CLI tool exposes a high level interface for interacting with bpocore APIs. This tutorial introduces two important features of the bpocore CLI tool:

- saving and restoring a baseline state of bpocore
- fast onboarding new types and templates

This interface is useful for managing deployments as well as developing definitions.

After completing this tutorial you will know how to speed up the type definition development cycle using the bpocore CLI tool.

Steps

1. Run the bootstrap steps in [bpocore CLI Bootstrap Tutorial](#).
2. Create a service template

```

mkdir -p ./example/types/tosca
cat <<EOF > ./example/types/tosca/hello_world.tosca
"\$schema"      = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-
module#"
title          = "A very simple service built from a template"
package        = example
version        = "1.0"
description    = """Defines a Hello World service template."""
authors       = ["Royal Joe (royal.joe@ciena.com)"]

resourceTypes {

    HelloWorld {
        title = Hello World
        description = "Hello World"
        derivedFrom = tosca.resourceTypes.Root
        properties {
            property1 {
                title = Property 1
                description = "Some property of the service"
                type = string
            }
            property2 {
                title = Property 2
                description = "Some other property of the service"
                type = string
            }
        }
    }
}

serviceTemplates {

    HelloWorld {

        title = Hello World
        description = "Hello World"
        implements = example.resourceTypes.HelloWorld

        resources {
            // various random strings modeled as subresources
            dbName.type = tosca.resourceTypes.RandomString
            dbUser.type = tosca.resourceTypes.RandomString
            dbUserPw.type = tosca.resourceTypes.RandomString
            dbRootPw.type = tosca.resourceTypes.RandomString
        }
    }
}
EOF

```

3. Establish a baseline state which can be restored to later

```
bpoctl baseline save
```

4. Onboard your service template and create a default product

```
bpocore onboard ./example --auto-products
```

5. Do it again, this time also create a resource. You should expect this to fail.

```
bpocore onboard ./example \
--auto-resources "example.resourceTypes.HelloWorld:{property1=foo}"
```

The tool is intelligent enough to notice that the type layer is up-to-date and the product is already created, so it will do only the minimum necessary to achieve the requested command: attempt to create a resource from the recently onboarded type and template.

See `bpocore onboard --help` for more information about the `--auto-products` and `--auto-resources` flags.

6. The previous step has failed, and the output indicates that we missed a property from the create request. Let's try again:

```
bpocore onboard ./example \
--auto-resource
"example.resourceTypes.HelloWorld:{property1=foo,property2=bar}"
```

This time it succeeds.

7. Make some changes to the template (e.g., add a new sub-resource to the template):

```
sed -i=''' 44'i \
    oneMoreSubresource.type = tosca.resourceTypes.RandomString' \
./example/types/tosca/hello_world.tosca
```

8. Try onboarding the modified template and create a new resource from it

```
bpocore onboard ./example \
--auto-resource
"example.resourceTypes.HelloWorld:{property1=foo,property2=bar}"
```

This succeeded: you can change a template even if it has been already used by an existing resource.

9. Make some changes to the resource type definition:

```
sed -i=''' 25'i \
property3 { \
    title = Property 3 \
    description = "Yes a nother property of the service" \
    type = string \
}' ./example/types/tosca/hello_world.tosca
```

10. Try onboarding the modified type file and create a new resource from it

```
bpocore onboard ./example --auto-resource \
"example.resourceTypes.HelloWorld:{property1=foo,property2=bar,property3=zulu}"
```

This will fail because our change conflicts with the existing state of bpocore: there is already a resource created from the resource type we want to change.

11. Restore your baseline state before trying again

```
bpocore baseline restore
```

Since the properties of the HelloWorld resource were invalid, the resource was not created at all. The bpocore baseline restore command did not have to revert any resources. If resources had been created, the command would have deleted them.

12. Re-onboard your service template and try again instantiating a resource with the right properties:

```
bpocore onboard ./example --auto-resource \
"example.resourceTypes.HelloWorld:{property1=foo,property2=bar,property3=zulu}"
```

First OpenStack Tutorial

This tutorial shows how to connect to an OpenStack "cloud" as an example of modeling, discovering, and managing real-world resources as **Resource-Providers**, **Domains**, **Products**, and **Resources** in BPO. This tutorial leverages the OpenStack Resource Adapter (RA) which talks to the OpenStack controller using REST interface and "adapt" the entities in the OpenStack domain as BPO models.

Specifically, we will:

- Create a domain to connect to an OpenStack controller
- Browse among the discovered products of the new domain
- Browse among the discovered resources of the new domain
- Create a new VM instance in the domain
- Terminate the VM instance
- Delete the domain

We will be using the `bpocore` CLI tool during this tutorial to simplify REST API interactions with BPO. The CLI tool calls BPO's HTTP API with appropriate headers based on the configuration. Same functionality can be achieved by using curl or other REST clients with appropriate authentication, or through the Swagger API interface.

Note: in order to follow along, you need to have access to a functioning OpenStack cloud. You will need the access URL, a user account, password, and a project name you can use. Please ensure that you have sufficient privileges and quotas to create at least one new VM. If in doubt, verify it first using the nova CLI or OpenStack Horizon.

Steps

Solution setup

1. Follow installation steps to have a Blue Planet **Orchestrate** solution running (either a full solution or a **lite** version of the solution). Note that this tutorial cannot be run using **bpocore-dev** as it requires interaction with the **OpenStack RA** solution.
2. Install the **polling** flavor of **OpenStack RA** solution `solution-raopenstack:yy.mm.poll.xxxx`.

*Note: The OpenStack RA comes in **polling** and **async** (i.e. non-polling) flavors. The async flavor requires extra setup on the OpenStack server to work. For simplicity, we will be using the polling version in this tutorial.*

Environment setup

Here we will set up some variables and functions that we will use later

1. Set up the following shell variable per the example below. Replace the IP address value with the **Site IP** of your deployed Orchestrate solution.

```
SITE_IP="192.168.33.10"
```

2. Set up the OpenStack controller access and credentials. Modify to match the setting of your OpenStack controller

```
OPENSTACK_URL="http://10.70.198.39:5000"
OPENSTACK_USER=admin
OPENSTACK_PW=admin
OPENSTACK_PROJECT=demo
```

3. Set up a function in the environment to simplify processing JSON data at the command line.

```
function jq { python -c "import sys,json; print json.load(sys.stdin)$1"; }
```

bpocore CLI setup

The deployed Orchestrate solution comes with a pip server from where the bpocore CLI tool can be installed. This section shows the steps to install and configure the CLI tool to talk to the deployed solution.

1. Install the bpocore CLI tool on your system.

```
virtualenv env
source env/bin/activate
pip install --trusted-host $SITE_IP -i https://$SITE_IP/bpocore-
docs/pypi/simple/ orchestrate-cli
bpocore --help
```

The first 3 commands install the CLI tool, and the last should show the bpocore command help if the installation is successful.

2. Configure the CLI tool to work with the site

```
bpocore init-config --hostname $SITE_IP --no-verify
```

3. Confirm the CLI tool is configured properly and working against the site by running

```
bpocore ping
```

The tool should respond with **up**.

Get the Resource Provider registered by the OpenStack RA

When a Resource Adapter starts up, it will **onboard** its Resource Types to BPO, and **register** itself as a **Resource Provider (RP)**. The user can then create **Domains** against the RP to connect to the southbound endpoint through the RA.

1. Execute

```
bpocore market api --format=json resource-providers get
```

In the output, look for the entry that has "domainType":

"urn:cyaninc:bp:domain:openstack". You should find something like this:

```
{
  "items": [
    ...
    {
      "asyncProtocolVersion": "v1",
      "description": "Exposes OpenStack endpoint in Orchestrate domain",
      "domainSettings": {
        ...
      },
      "domainType": "urn:cyaninc:bp:domain:openstack",
      "id": "59277e60-e0ed-4342-b124-88a7a461bcc",
      "properties": {
        "password": {
          "description": "Password for the user of the OpenStack domain",
          "obfuscate": true,
          "title": "User password",
          "type": "string"
        },
        "projectname": {
          "description": "Name of the OpenStack project/tenant",
          "title": "OpenStack project",
          "type": "string"
        },
        "username": {
          "description": "Name for the user of the OpenStack domain",
          "title": "User Name",
          "type": "string"
        },
        ...
      },
      "protocolVersion": "v2",
      "providerId": "urn:cyaninc:bp:ra:openstack",
      "relationships": [
        ...
      ],
      "resourceTypes": [
        ...
      ],
      "title": "External OpenStack provider",
      "uri": "http://blueplanet:80/ractrl/api/v1"
    }
    ...
  ],
  "offset": 0,
  "total": 2
}
```

Note that the Resource Provider entity includes information of how to reach the Resource Adapter (the `uri`), the **Resource Types** and **Relationships** it provides, as well as the **properties** it expects the user to enter when creating a Domain (`username`, `password`, `projectname`, etc.) and other information. Notice that the `password` property has `obfuscate` set to `true`, which will affect how the property shows up when a domain is returned by a GET API call, as we shall see later.

Let's record the `id` value of the RP. A simple trick is to make use the `q` parameter which filters based on

exact matching. We want to filter based on the `domainType`
`domainType:urn:cyaninc:bp:domain:openstack`

```
RP_ID=$(bpocore market api --format=json resource-providers get
q=\`domainType:urn:cyaninc:bp:domain:openstack\` | jq '[ "items" ][0][ "id" ]')
```

Create a Domain and inspect discovered Products and Resources

We will create a Domain against the OpenStack RP next. Notice that the OpenStack RP's `id` will be passed in as the `rpId` of the Domain, thus associate the Domain with the RP.

1. Create a file to be used as the body of the POST call to create the Domain

Copy and paste the following to the OS command prompt

```
cat <<EOF > domain_prop.json
{
    "title": "Demo OpenStack",
    "description": "stacked like pancakes",
    "accessUrl": "$OPENSTACK_URL",
    "properties": {
        "username": "$OPENSTACK_USER",
        "password": "$OPENSTACK_PW",
        "projectname": "$OPENSTACK_PROJECT"
    },
    "address": {
        "city": "Petaluma",
        "zip": "94954",
        "street": "1383 N McDowell",
        "state": "CA",
        "country": "USA",
        "latitude": 38,
        "longitude": -110
    },
    "rpId": "$RP_ID"
}
EOF
```

2. Issue the following command.

```
DOMAIN_ID=$(bpocore market api --format=json domains post ./domain_prop.json |
jq '[ "id" ]')
```

The bpocore CLI tool will issue a POST call to the **Market domains API** to create a Domain using the file created from the last step, and then assign the `id` of the created Domain as `DOMAIN_ID`

3. Let's look at the domain created.

Execute

```
bpocore market api --format=json domains $DOMAIN_ID get
```

The tool should return a Domain entity like the following

```
{
    "accessUrl": "http://10.70.198.39:5000",
    "address": {
        "city": "Petaluma",
        "country": "USA",
        "latitude": 38.0,
        "longitude": -110.0,
        "state": "CA",
        "street": "1383 N McDowell",
        "zip": "94954"
    },
    "connectionStatus": "CONNECTED",
    "description": "stacked like pancakes",
    "domainType": "urn:cyaninc:bp:domain:openstack",
    "id": "5927ab64-7566-4267-89e6-2fde492638b8",
    "initialDiscoveryStatus": "N/A",
    "lastConnected": "2017-05-26T04:13:39.048Z",
    "operationMode": "normal",
    "properties": {
        "password": "*****",
        "projectname": "demo",
        "username": "admin"
    },
    "reason": "",
    "rpId": "59277e60-e0ed-4342-b124-88a7a461bccca",
    "tenantId": "b600946e-c9c4-4409-bb4b-4ea4c08e4dbd",
    "title": "Demo OpenStack"
}
```

Note the connectionStatus is CONNECTED. This means the connection to the OpenStack server has been successfully established. If there are connectivity or credential issues, other status may be shown. If the connectionStatus shows CONNECTING, issue the command a couple of times more until it shows as CONNECTED.

Also note that the password of the Domain is shown as a bunch of asterisks instead of the actual value. Remember when we inspected the OpenStack Resource Provider, the password property has obfuscate set to true, which affects how the property is shown when the API returns a domain.

4. List Products advertised by the new domain. We can use the Domain ID above to narrow down the Product list by using the same exact matching parameter.

```
bpocore market api --format=json products get q='domainId:$DOMAIN_ID\'
```

The RA provides 2 active products, `Virtual Machine` and `VirtualMachine VNC Console`, as the following

```
{
  "items": [
    {
      "active": true,
      "constraints": {},
      "domainId": "5927b981-7d31-467a-a1d9-954a09608c9a",
      "id": "5927b982-219e-4eea-b2b8-7350e88283e8",
      "providerData": {},
      "providerProductId": "urn:cyaninc:bp:product:openstackvncconsole",
      "resourceTypeId": "openStack.resourceTypes.VncConsole",
      "title": "VirtualMachine VNC Console"
    },
    {
      "active": true,
      "constraints": {},
      "domainId": "5927b981-7d31-467a-a1d9-954a09608c9a",
      "id": "5927b982-8cbb-40a0-a4fc-a28225fbcc81",
      "providerData": {},
      "providerProductId": "urn:cyaninc:bp:product:openstackvm",
      "resourceTypeId": "openStack.resourceTypes.VirtualMachine",
      "title": "Virtual Machine"
    }
  ],
  "limit": 1000,
  "offset": 0,
  "total": 2
}
```

Note1: It may take a short while to get all Products populated. Run the above command again if no product is seen.

Note2: You could add **includeninactive=true** to the end of the command line to show **all** products advertised by the OpenStack RA. An active product can be created through the UI and API, while an inactive product can only be created through the API, if the RA supports creation of such a Resource Type.

- Let's set the ID of the Virtual Machine product as a variable, so we could use it later.

```
VM_PRODUCT_ID=$(bpocore market api --format=json products get
q='domainId:$DOMAIN_ID,resourceTypeId:openStack.resourceTypes.VirtualMachine'
| jq '[ "items" ][0][ "id" ]')
```

Note: Here a new query is done to filter based on the `resourceTypeId` of the product. You could alternatively just copy the id of the Virtual Machine product from the previous result, and assign it to the `VM_PRODUCT_ID` variable

- List all Resources discovered under the new domain.

Execute

```
bpocore market api --format=json resources get domainId=$DOMAIN_ID
```

You should see at least a few items in the result. Notice that all items have "discovered": true, indicating that these are all discovered resources from the OpenStack server the domain connects to. Here is a snippet from our output:

```
{
  "items": [
    ...
    {
      "autoClean": false,
      "createdAt": "2017-05-26T04:13:33.061Z",
      "desiredOrchState": "unspecified",
      "differences": [],
      "discovered": true,
      "id": "5927ab6d-67b4-44cf-bc24-1dae6be8ad29",
      "label": "Firefly-junos-vsrx-12.1X47-D20.7-domestic-CTL-V1-Policies",
      "orchState": "active",
      "productId": "5927ab65-86ed-4f74-b809-d1c51af36d89",
      "properties": {
        "ID": "756bccf2-2fd8-4df0-8913-be5952ad7158",
        "name": "Firefly-junos-vsrx-12.1X47-D20.7-domestic-CTL-V1-Policies",
        "pm": {
          "enabled": true,
          "interval": 300
        },
        "shared": false
      },
      "providerData": {},
      "providerResourceId": "756bccf2-2fd8-4df0-8913-be5952ad7158",
      "reason": "",
      "resourceTypeId": "openStack.resourceTypes.Image",
      "shared": false,
      "tags": {},
      "tenantId": "b600946e-c9c4-4409-bb4b-4ea4c08e4dbd",
      "updatedAt": "2017-05-26T04:13:33.061Z"
    }
  ],
  "limit": 1000,
  "offset": 0,
  "total": 19
}
```

Note: The domain ID is not an attribute of a Resource entity, so the q parameter can not be used directly here to filter by the domain ID. The **Market resources API** provides a domainId parameter for this purpose, which we used here. More information of the Market API and available parameters can be viewed by executing

```
bpocore market api
```

Inspect the OpenStack VirtualMachine Resource Type

We will create a VM from BPO in the later sections. Before a VM can be created, though, we need to know the information needed to for the creation. This can be done by inspecting the corresponding Resource Type to find the properties that are configurable and required.

Note: The OpenStack RA, as well as many other Resource Adapters, comes with **UI Schema** for the **active Products** it provides, such as the Virtual Machine in this case. With a proper made UI schema, the user can use drop-down or other UI selections to construct the body needed to fill in properties when creating or updating a resource. When the operation is done through direct API call, though, the format and the data will need to be obtained by the caller, as we will do in this and next section.

1. Execute the command to inspect the **openStack.resourceTypes.VirtualMachine** Resource Type

```
bpoctl market api --format=json resource-types  
openStack.resourceTypes.VirtualMachine get
```

This will return a JSON Object describing every details of this Resource Type. The `properties` section describes the properties of this Resource Type and the Access Modifiers for each of the property. The following shows a subset of the properties that we will be using for creating a Virtual Machine later. Note that some properties have the `config` modifier set to `true`, while others `false`. Only the properties with `config=true` can be specified by the user when creating a resource. If the `optional` is `true` or if `default` exists, the user can choose to not specify the corresponding property.

```
"properties": {
    "flavorID": {
        "config": true,
        "description": "Flavor ID",
        "fulltext": false,
        "history": true,
        "obfuscate": false,
        "optional": true,
        "output": false,
        "prefix": "openStack.resourceTypes.VirtualMachine",
        "store": true,
        "title": "Flavor ID",
        "type": "string",
        "updatable": false
    },
    "imageID": {
        "config": true,
        ...
    },
    "keyPair": {
        "config": true,
        ...
    },
    "networks": {
        "config": true,
        ...
        "items": {
            "properties": {
                ...
                "id": {
                    "config": true,
                    ...
                    "optional": false,
                    ...
                }
            },
            "type": "object"
        },
        "type": "array",
        ...
    },
    ...
},
...
"securityGroups": {
    "config": true,
    ...
}
},
```

We will be specifying flavorID, imageID, keyPair, securityGroups, and networks in this tutorial. We will find the actual ID to be set in these properties next.

Find the Resource IDs needed for creating a VM

In this section, we will find the IDs of an **Image**, a **Flavor**, and a ***Network** for the VM. We will also find the ID of a **SecurityGroup** for the VM to use.

Note: Through a UI schema, the UI call corresponding Market APIs when the user clicks on the UI menus to get relevant IDs, like we do manually in this section.

1. Let's find an **Image** ID first. This can be done again by doing a list using the query filters on the **domain ID** and the **Resource Type** `openStack.resourceTypes.Image`, such as

```
bpocore market api --format=json resources get domainId=$DOMAIN_ID  
resourceTypeId=openStack.resourceTypes.Image
```

This returns a couple of **Image** resources in our case.

```
{
  "items": [
    ...
    {
      "autoClean": false,
      "createdAt": "2017-05-26T05:13:51.723Z",
      "desiredOrchState": "unspecified",
      "differences": [],
      "discovered": true,
      "id": "5927b98f-f407-4701-a2f4-5e593e4425cb",
      "label": "cirros",
      "orchState": "active",
      "productId": "5927b982-9a6f-435e-acdd-c4d666c9357e",
      "properties": {
        "ID": "ce0909be-cba2-4086-b779-03de8881369d",
        "name": "cirros",
        "pm": {
          "enabled": true,
          "interval": 300
        },
        "shared": false
      },
      "providerData": {},
      "providerResourceId": "ce0909be-cba2-4086-b779-03de8881369d",
      "reason": "",
      "resourceTypeId": "openStack.resourceTypes.Image",
      "shared": false,
      "tags": {},
      "tenantId": "b600946e-c9c4-4409-bb4b-4ea4c08e4dbd",
      "updatedAt": "2017-05-26T05:13:51.723Z"
    },
    ...
  ],
  "limit": 1000,
  "offset": 0,
  "total": 3
}
```

Pick one image in your OpenStack domain to create the VM from. We will pick the **cirrus** image from above. Assign its id to `IMAGE_ID` variable (remember to use the actual ID from your result)

```
IMAGE_ID="5927b98f-f407-4701-a2f4-5e593e4425cb"
```

If the name of the image to use is already known, it can be specified in the query filter (by `label`), such as

```
IMAGE_ID=$(bpocore market api --format=json resources get domainId=$DOMAIN_ID
resourceTypeId=openStack.resourceTypes.Image q='label:cirros' | jq
'["items"][0]["id"]')
```

2. Repeat the procedure for the **Flavor ID**.

Here we will use the **m1.medium** flavor and filter by the label directly. You could alternatively do a list and pick the flavor you want to use, and assign its id to FLAVOR_ID

```
FLAVOR_ID=$(bpocore market api --format=json resources get domainId=$DOMAIN_ID
resourceTypeId=openStack.resourceTypes.Flavor q='label:m1.medium' | jq
'["items"][0]["id"]')
```

3. Repeat similar procedure for the **Network ID**

Pick the network for the VM to connect to. We will use same approach to find the **private** network. Note that the OpenStack RA calls for the providerResourceId of the network resource to be specified the networks property, so instead of the id, the providerResourceId is set to the NETWORK_ID

```
NETWORK_ID=$(bpocore market api --format=json resources get domainId=$DOMAIN_ID
resourceTypeId=openStack.resourceTypes.Network q='label:private' | jq
'["items"][0]["providerResourceId"]')
```

4. Repeat for the **Security Group ID**

Pick the Security Group for the VM to use. We will use same approach to find the **default** group. Set the id of the resource to SECURITY_GROUP_ID

```
SECURITY_GROUP_ID=$(bpocore market api --format=json resources get
domainId=$DOMAIN_ID resourceTypeId=openStack.resourceTypes.SecurityGroup
q='label:default' | jq '["items"][0]["id"]')
```

Create the VM

With the IDs needed to create the VM obtained from the previous section, we can now create a VM.

1. Create a file to use as the POST body

This uses the IDs obtained from the previous section and the VM product we saved earlier to fill in the body needed for VM creation **POST** call.

```
cat <<EOF > vm_prop.json
{
    "productId": "$VM_PRODUCT_ID",
    "label": "Our shiny VM",
    "properties": {
        "networks": [ {"id": "$NETWORK_ID"} ],
        "imageID": "$IMAGE_ID",
        "flavorID": "$FLAVOR_ID",
        "securityGroups": [ "$SECURITY_GROUP_ID" ]
    }
}
EOF
```

2. Create the VM Execute

```
VM_ID=$(bpocore market api --format=json resources post ./vm_prop.json | jq
'["id"]')
```

A **POST** call to the **Market resources API** with the body above will create a VM resource in market, which will in turn instruct the RA to issue commands to perform the creation in the OpenStack server.

Note here we save the `id` of the created VM as the `VM_ID` variable.

3. Check the VM state: We can call **GET** to the **resources API** using the ID of the newly created VM

```
bpocore market api --format=json resources $VM_ID get
```

Here is a sample output of our VM:

```
{
  "autoClean": false,
  "createdAt": "2017-05-26T06:54:02.804Z",
  "desiredOrchState": "active",
  "differences": [],
  "discovered": false,
  "id": "5927d10a-be76-4299-b865-7fd27b32cf7b",
  "label": "Our shiny VM",
  "orchState": "active",
  "productId": "5927b982-8cbb-40a0-a4fc-a28225fbcc81",
  "properties": {
    "addresses": [
      {
        "fixedIpAddresses": [
          "10.0.0.100"
        ],
        "floatingIpAddresses": [],
        "networkId": "5927b98e-3813-447f-8fe9-9ce517a9952f"
      }
    ],
    "availabilityZone": "nova",
    "cpuUtilizationThreshold": 90,
    "diskSize": 40,
    "diskUtilizationThreshold": 90,
    "flavorID": "5927b984-1b4d-4828-865f-0606e88d0b63",
    "hypervisorId": "5927b9be-fd73-43b8-9d6e-ed610a3b695d",
    "id": "418e9252-2206-451f-9e2a-0995080d8f48",
    "imageID": "5927b98f-f407-4701-a2f4-5e593e4425cb",
    "memSize": 4096,
    "memoryUtilizationThreshold": 90,
    "name": "Our shiny VM",
    "networks": [
      {
        "id": "677f3b6e-0be9-4698-9c8e-83da08c2402f"
      }
    ],
    "numCpus": 2,
    "pm": {
      "enabled": true,
      "interval": 300
    },
    "securityGroups": [
      "5927b985-5d56-4287-81cc-9cdbacee73c8"
    ],
    "status": "ACTIVE"
  },
  "providerData": {},
  "providerResourceId": "418e9252-2206-451f-9e2a-0995080d8f48",
  "reason": "",
  "resourceTypeId": "openStack.resourceTypes.VirtualMachine",
  "shared": false,
  "tags": {},
  "tenantId": "b600946e-c9c4-4409-bb4b-4ea4c08e4dbd",
  "updatedAt": "2017-05-26T06:55:49.731Z"
}
```

As the VM being created and activated, the `orchState` of the resource may become requested,

unknown, or activating before finally goes to active if everything goes well. Note that in our case, a fixed IP 10.0.0.100 is assigned to our VM. Extra information is also populated to the VM resource's **properties** from the information obtained from the OpenStack server by the RA

4. Look at the dependencies of the VM

Relationships between resources of different Resource Types can be modeled in BPO. Here we will look at an example of how this is done in the OpenStack domain. Let's call a **dependencies API** which will list all the resources our new VM depends on.

Execute

```
bpocore market api --format=json resources $VM_ID dependencies get
```

The following is a sample example from our VM. We can see the VM depends on the `Image`, `SecurityGroup`, `Network`, and the `Flavor` that we picked earlier. In addition, the VM is hosted on the Hypervisor named `c1-pod1` which is also modeled as a Resource. Finally, a `Port` is created on the OpenStack server as a result of the VM creation. The RA discovered the new Port as a part of a **scheduled polling**, associate it with the new VM, and report back to BPO to install a resource and relationship to the Market Database.

```
{
  "items": [
    {
      "autoClean": false,
      "createdAt": "2017-05-26T05:13:51.723Z",
      "desiredOrchState": "unspecified",
      "differences": [],
      "discovered": true,
      "id": "5927b98f-f407-4701-a2f4-5e593e4425cb",
      "label": "cirros",
      "orchState": "active",
      "productId": "5927b982-9a6f-435e-acdd-c4d666c9357e",
      "properties": {
        "ID": "ce0909be-cba2-4086-b779-03de8881369d",
        "name": "cirros",
        "pm": {
          "enabled": true,
          "interval": 300
        },
        "shared": false
      },
      "providerData": {},
      "providerResourceId": "ce0909be-cba2-4086-b779-03de8881369d",
      "reason": "",
      "resourceTypeId": "openStack.resourceTypes.Image",
      "shared": false,
      "tags": {},
      "tenantId": "b600946e-c9c4-4409-bb4b-4ea4c08e4dbd",
      "updatedAt": "2017-05-26T05:13:51.723Z"
    },
    {
      "dependsOn": [
        {
          "id": "5927b98f-f407-4701-a2f4-5e593e4425cb",
          "label": "cirros"
        }
      ],
      "label": "cirros"
    }
  ]
}
```

```
"autoClean": false,
"createdAt": "2017-05-26T06:56:31.858Z",
"desiredOrchState": "unspecified",
"differences": [],
"discovered": true,
"id": "5927d19f-3af6-482f-81df-9alea5e028fd",
"label": "",
"orchState": "active",
"productId": "5927b982-c9b1-420d-9f6c-30a6cac2283f",
"properties": {
    "admin_state_up": "True",
    "device_compute_id": "5927d10a-be76-4299-b865-7fd27b32cf7b",
    "fixedIPs": [
        {
            "ip_address": "10.0.0.100",
            "subnet_id": "5927b98e-49c1-4f37-84e5-3c2887586370"
        }
    ],
    "host_id": "cl-pod1",
    "id": "bff411ce-0c3d-4a7c-8f8b-7e6f9d1fecc1",
    "ipV4Addresses": [
        "10.0.0.100"
    ],
    "macAddresses": [
        "fa:16:3e:bf:7a:3a"
    ],
    "name": "",
    "network": "5927b98e-3813-447f-8fe9-9ce517a9952f",
    "pm": {
        "enabled": true,
        "interval": 300
    },
    "securityGroups": [
        "5927b985-5d56-4287-81cc-9cdbacee73c8"
    ],
    "status": "ACTIVE",
    "tenant_id": "e02f78d36b9644d893d43aaaf8b9129d2",
    "vif_type": "ovs",
    "vnicType": "normal"
},
"providerData": {},
"providerResourceId": "bff411ce-0c3d-4a7c-8f8b-7e6f9d1fecc1",
"reason": "",
"resourceTypeId": "openStack.resourceTypes.Port",
"shared": false,
"tags": {},
"tenantId": "b600946e-c9c4-4409-bb4b-4ea4c08e4dbd",
"updatedAt": "2017-05-26T06:56:31.858Z"
},
{
    "autoClean": false,
    "createdAt": "2017-05-26T05:14:38.656Z",
    "desiredOrchState": "unspecified",
    "differences": [],
    "discovered": true,
    "id": "5927b9be-fd73-43b8-9d6e-ed610a3b695d",
    "label": "cl-pod1",
    "orchState": "active",
    "productId": "5927b982-8cbe-4c80-8b09-62654dff61d8",
}
```

```
        "properties": {
            "diskSize": 821,
            "hypervisor_hostname": "cl-pod1",
            "id": "Hypervisor::1",
            "memSize": 128714,
            "numCpus": 24,
            "remoteAccess": "10.70.129.20",
            "state": "up",
            "status": "enabled",
            "type": "QEMU",
            "zone": "internal"
        },
        "providerData": {},
        "providerResourceId": "Hypervisor::1",
        "reason": "",
        "resourceTypeId": "openStack.resourceTypes.Hypervisor",
        "shared": false,
        "tags": {},
        "tenantId": "b600946e-c9c4-4409-bb4b-4ea4c08e4dbd",
        "updatedAt": "2017-05-26T05:38:59.510Z"
    },
    {
        "autoClean": false,
        "createdAt": "2017-05-26T05:13:41.720Z",
        "desiredOrchState": "unspecified",
        "differences": [],
        "discovered": true,
        "id": "5927b985-5d56-4287-81cc-9cdbacee73c8",
        "label": "default",
        "orchState": "active",
        "productId": "5927b982-ad53-462e-a024-b08dc8b0318a",
        "properties": {
            "description": "default",
            "id": "SecurityGroup::7a00af9c-12e9-4420-8be9-4fde0772338c",
            "name": "default",
            "rules": [
                {
                    "ip_protocol": "None"
                },
                {
                    "cidr": "0.0.0.0/0",
                    "ip_protocol": "tcp"
                },
                {
                    "ip_protocol": "None"
                },
                {
                    "cidr": "0.0.0.0/0",
                    "ip_protocol": "udp"
                },
                {
                    "cidr": "0.0.0.0/0",
                    "ip_protocol": "tcp"
                },
                {
                    "cidr": "0.0.0.0/0",
                    "ip_protocol": "icmp"
                }
            ]
        }
    }
}
```

```
        "cidr": "0.0.0.0/0",
        "ip_protocol": "tcp"
    },
    {
        "cidr": "0.0.0.0/0",
        "ip_protocol": "tcp"
    }
]
},
"providerData": {},
"providerResourceId": "SecurityGroup::7a00af9c-12e9-4420-8be9-4fde0772338c",
"reason": "",
"resourceTypeId": "openStack.resourceTypes.SecurityGroup",
"shared": false,
"tags": {},
"tenantId": "b600946e-c9c4-4409-bb4b-4ea4c08e4dbd",
"updatedAt": "2017-05-26T05:13:41.720Z"
},
{
    "autoClean": false,
    "createdAt": "2017-05-26T05:13:50.509Z",
    "desiredOrchState": "unspecified",
    "differences": [],
    "discovered": true,
    "id": "5927b98e-3813-447f-8fe9-9ce517a9952f",
    "label": "private",
    "orchState": "active",
    "productId": "5927b982-cbc4-43d5-a788-19532f7c4f1f",
    "properties": {
        "admin_state_up": true,
        "external": false,
        "id": "677f3b6e-0be9-4698-9c8e-83da08c2402f",
        "name": "private",
        "networkType": "vlan",
        "physicalNetwork": "physnet1",
        "pm": {
            "enabled": true,
            "interval": 300
        },
        "segmentationId": 101,
        "shared": false,
        "status": "ACTIVE",
        "subnets": [
            "0d80de20-5ele-4edc-9ef9-8ce3719ad31d"
        ],
        "tenant_id": "e02f78d36b9644d893d43aaaf8b9129d2"
    },
    "providerData": {},
    "providerResourceId": "677f3b6e-0be9-4698-9c8e-83da08c2402f",
    "reason": "",
    "resourceTypeId": "openStack.resourceTypes.Network",
    "shared": false,
    "tags": {},
    "tenantId": "b600946e-c9c4-4409-bb4b-4ea4c08e4dbd",
    "updatedAt": "2017-05-26T05:13:50.509Z"
},
{
    "autoClean": false,
```

```

    "createdAt": "2017-05-26T05:13:40.634Z",
    "desiredOrchState": "unspecified",
    "differences": [],
    "discovered": true,
    "id": "5927b984-1b4d-4828-865f-0606e88d0b63",
    "label": "m1.medium",
    "orchState": "active",
    "productId": "5927b982-62fe-4f24-9623-25df2ae92f8d",
    "properties": {
        "ID": "Flavor::3",
        "disk": "40",
        "name": "m1.medium",
        "ram": "4096",
        "vcpus": "2"
    },
    "providerData": {},
    "providerResourceId": "Flavor::3",
    "reason": "",
    "resourceTypeId": "openStack.resourceTypes.Flavor",
    "shared": false,
    "tags": {},
    "tenantId": "b600946e-c9c4-4409-bb4b-4ea4c08e4dbd",
    "updatedAt": "2017-05-26T05:13:40.634Z"
}
],
"offset": 0,
"total": 6
}

```

Feel free to play around using the CLI commands. We will delete the VM next.

Delete the VM and Domain

1. Delete the VM:

Issue the following command to delete the VM

```
bpocore market api resources $VM_ID delete
```

The bpocore CLI performs a DELETE call to the **Market resources API**, which will in turn issue a DELETE call to the RA, and eventually a DELETE call to the OpenStack server for the specific VM.

2. Monitor the VM until it is gone

```
bpocore market api --format=json resources $VM_ID get
```

While the VM is terminating, you will see the `orchState` value as `terminating`. Once it is gone, the API will return a 404 and the CLI should mention the resource no longer being found in Market, like the following:

```
Resource '5927d10a-be76-4299-b865-7fd27b32cf7b' not found
```

3. If there are no more Resources left in the domain (other than discovered resources), you can remove the domain:

```
bpocore market api domains $DOMAIN_ID delete
```

This will remove the domain together with any products and discovered resources. If there are any **managed** resource in the domain (such as the VM we created and deleted from BPO earlier), the API will reject the domain deletion request until all such resources are terminated.

Further Notes

- In addition to the properties we set when creating a VM, configurations such as an **SSH key pair** can be specified. The VM can also be created to connect to more than one networks. See the OpenStack RA documentation for more information.
- You can also create Resources of other Types, such as Networks, Subnets, or Floating IPs using a similar approach.
- By creating additional domains with different **accessUri**, you can connect to other OpenStack clouds through the OpenStack RA. The entities like VM, Network etc. corresponding to a specific OpenStack server will be discovered as Resources associated with the corresponding Domain.
- The OpenStack RA enables the modeling and management of OpenStack resources in BPO. By adding extra RAs, resources of different nature can all be modeled in a unified representation, which enables Management, Provisioning, and Orchestration of different Types of resources by means of Service Templates and other functionalities provided by the BPO framework.

Hello RP Tutorial

This tutorial shows you how to create a basic Resource Provider (RP) which integrates with bpocore. RPs act as proxies between bpocore and southbound domains. While acting as a proxy, the RP translates messages from one model to the other.

This tutorial will be focused on creating a Python-based RP using the [rpsdk-python](#) framework. For the purpose of the tutorial the RP will expose a single domain type which manages files on your local filesystem.

Steps

1. Create a new directory for this tutorial

```
mkdir hello-rp && cd hello-rp
```

2. Bootstrap your environment following the steps in [Bootstrap Steps for bpocore-dev and Environment](#)

3. Install the [rpsdk-paste](#) package into a virtualenv

```
virtualenv main27
source main27/bin/activate
pip install -i https://artifactory.ciena.com/api/pypi/blueplanet-pypi/simple
rpsdk-paste==0.0.5
```

4. Create the RP skeleton

```
# Answer the questions for "title", "author", and "author_email". The
# other settings can be left as defaults for now.
#
# title = Example
# author = <YOUR NAME>
# author_email = <YOUR EMAIL>
#
paster create -t bprp example
```

5. Install the necessary dependencies for your RP

```
cd example
make prepare-venv
source env/bin/activate
```

6. Create a dummy type definition

```
mkdir example/definitions
```

```
cat <<EOF > example/definitions/dummy.tosca
"\$schema"      = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-
module#"
title          = "Dummy"
package        = example
version        = "1.0"
description    = "This document defines the Dummy resource type."
authors        = [ "Royal Joe (royal.joe@ciena.com)" ]

resourceTypes {

    Dummy {
        derivedFrom = tosca.resourceTypes.Root
        title = Dummy
        description = """
                    A dummy resource type for example purposes in the Hello RP
                    tutorial. This resource type models a file in a flat filesystem.
                    """
    }

    properties {

        path {
            type = string
            title = "Path"
            description = "Uniquely identifies the file on the filesystem"
            optional = false
        }

        fileType {
            type = string
            enum = [directory, file]
            title = "File Type"
            description = "Specifies the type of file"
            optional = false
        }

        parent {
            type = string
            title = "Parent"
            description = "Identifies the directory which contains this file"
            optional = true
        }

    }

    capabilities {

        contains {
            type = tosca.capabilityTypes.Container
            resourceTypes = [example.resourceTypes.Dummy]
        }

    }

    requirements {

        contained {
            type = tosca.capabilityTypes.Container
            resourceTypes = [example.resourceTypes.Dummy]
        }

    }

}
```

```

        minimum = 0
    }

}

}
EOF

```

Note: This defines the properties of a file/directory in the file system, as well as the relationships among the resources of this type. A directory can contain zero or more file/directory, while a file/directory can be contained in at most one parent. If not specified explicitly, a configured capability has a default of minimum = 0 and no maximum, while a requirement has a default of minimum = 1 and maximum = 1. Since the requirement is optional in this case, it is set to 0 explicitly.

7. Update the RP configuration

```

cat <<EOF >> example/config/example.yaml

# Location where RP specific definitions are stored. E.g, type definitions, UI
# schema, etc. This value can be absolute or relative to this file however it
# must be resolvable when deployed. If this value is null then no definitions
# will be onboarded.
definitions: ../definitions

# Domain definitions specify the types of domains supported by this RP.
# Developers must define their own domains. See
# https://github.cyanoptics.com/Orchestrate/rpsdk-python for more details.
domains:
  - id: "urn:cyaninc:bp:domain:dummy"
    title: "Dummy"
    description: "Exposes a directory on the local filesystem"
    properties: []
    resource_types:
      - id: "example.resourceTypes.Dummy"

# Relationship definitions specify how relationships between resources managed
# by this RP are formed. See
# https://github.cyanoptics.com/Orchestrate/rpsdk-python for more details.
relationships:
  - sourceTypeId: "example.resourceTypes.Dummy"
    targetType: "example.resourceTypes.Dummy"
    capabilityName: "contains"
    requirementName: "contained"
    relationshipTypeId: "tosca.relationshipTypes.HostedOn"
    fieldKind: "provider"
    fieldLocator: "parent"
EOF

```

Note: After you perform this step you should clean up example.yaml so it does not contain the skeleton sections anymore.

8. Implement your RP

```
GITREPO=https://github.cyanoptics.com/raw/Orchestrate/bpocore-docs  
curl -s $GITREPO/master/tutorials/hello-rp__init__.py > example/__init__.py
```

9. Create a deployment configuration

```
cat <<EOF > dev.yaml  
orchestrate:  
    market_url: $MARKET_URL  
    assets_url: $ASSETS_URL  
identity:  
    public_key: $HOME/.ssh/id_rsa.pub  
server:  
    bind_address: $DOCKER_BRIDGE_IP  
    bind_port: 9191  
EOF
```

10. Run your RP in a terminal separate from the other steps

```
cd hello-rp/example  
source env/bin/activate  
example example/config/example.yaml dev.yaml
```

11. Query for your resource provider

```
jcurl $MARKET_URL/resource-providers?q=domainType:urn:cyaninc:bp:domain:dummy |  
jpp  
RP_ID='<USE "id" FROM OUTPUT ABOVE>'
```

12. Create a domain against your RP

```

cat <<EOF > domain.json
{
    "rpId": "$RP_ID",
    "title": "${USER}'s Hello RP Example",
    "description": "Test domain which is part of the Hello RP tutorial",
    "accessUrl": "file:///tmp/hello-rp/",
    "properties": {},
    "address": {
        "country": "",
        "state": "",
        "zip": "",
        "street": "",
        "latitude": -77.529722,
        "longitude": 167.1553333,
        "city": ""
    }
}
EOF
jcurl -d @domain.json $MARKET_URL/domains | jpp

DOMAIN_ID='<USE "id" FROM OUTPUT ABOVE>'
```

13. Create some files for discovery purposes

```

mkdir -p /tmp/hello-rp/foo/bar
touch /tmp/hello-rp/foo/bar/baz
touch /tmp/hello-rp/foo/bar/qux
```

14. Query for the product which has been discovered

```

jcurl $MARKET_URL/products?q=providerProductId:urn:example:product:default-dummy
| jpp
PRODUCT_ID='<USE "id" FROM OUTPUT ABOVE>'
```

15. Query for resources which have been discovered

```
jcurl $MARKET_URL/resources?domainId=$DOMAIN_ID | jpp
```

Note: the discovery process involves periodically polling the RP. The default polling intervals are fairly relaxed. It may take around one minute for all of the resources to show up.

16. Create a new resource for the discovered product

```
cat <<EOF > resource.json
{
    "productId": "$PRODUCT_ID",
    "discovered": false,
    "properties": {
        "fileType": "file",
        "path": "foo/bar/`date +%s`"
    }
}
EOF
jcurl -d @resource.json $MARKET_URL/resources | jpp
RESOURCE_ID='<USE "id" FROM OUTPUT ABOVE>'
```

17. See relationships that have been created automatically

```
jcurl
"$MARKET_URL/relationships?q=capabilityName:contains,requirementName:contained"
\|
| jpp
```

18. Delete the domain and unregister the RP

```
jcurl -X DELETE $MARKET_URL/resources/$RESOURCE_ID
jcurl -X DELETE $MARKET_URL/domains/$DOMAIN_ID
jcurl -X DELETE $MARKET_URL/resource-providers/$RP_ID
```

Service Template Tutorial

This tutorial shows how to implement a Resource Type using Service Templates. A Service Template "implements" a Resource Type by providing business logic for lifecycle operations (e.g., activation, termination, update) of the Resource Type.

A Resource Type can be implemented by one or more Service Templates. A **Product** links a specific Service Template implementation to the Resource Type. When the user creates a resource from a product associated with a Service Template, the "activation plan" corresponding to the Service Template will be executed. Sub-resources and relationships may be created based on the configuration of the Service Template and the configuration properties the user specifies for the resource. When the execution of the "activation plan" completes, the service-level resource will have **orchState** set to **active** if the activation was successful or set to **failed** if the activation was unsuccessful.

Similarly, "update plan" and "termination plan" will be executed when the user attempts to update the resource (via PUT or PATCH API calls), or attempts to delete it. An user-update will generally cause the resource to have **differences**, and the update plan should try to "clear" the differences by moving the state of the managed resources (a.k.a observed state) to the expected state from the user. A termination plan should clean up any sub-resources and relationships created by the activation and/or update plans by either deleting the sub-resources, or moving any shared resource to a reverted state.

A Service Template can be defined declaratively using BPO's Domain Specific Language, or imperatively by providing executables to handle the lifecycle operations against associated Resources.

Please refer to the following links for more reference information:

- [Resource Types](#)
- [Template Engine](#)
- [Service Templates](#)
- [Directives for declarative templates](#)
- [Imperative Plans](#)

In this tutorial, you will provide alternative implementations of the same resource type by implementing a declarative service template and an imperative service template. Each will achieve the same behavior for the lifecycle operations of the type.

After completing this tutorial, you will know:

- How to create declarative Service Templates
- How to create imperative Service Templates and scripted plans
- How to set output properties of a resource activated by a Service Template
- Simple Resource Types that are useful for Service Template developing
- Differences between declarative and imperative styles

- Advantages and limitations of each style

Steps

Prerequisites

1. We will be using the bpocore-cli tool during this tutorial to simplify REST API and git interface interactions with BPO.

To set up bpocore-cli, run the bootstrap steps in [bpocore CLI Bootstrap Tutorial](#). You will also need to follow the steps to have a bpocore-dev container running.

2. Run frost-orchestrate-ui-dev by following the steps in [Running the frost-orchestrate-ui-dev Docker container](#). You will not use the UI to create resources in this tutorial, but the UI is useful for visualizing the end result.

Define ExampleService Resource Type

1. First, create the **ExampleService** Resource Type the Service Templates will implement. Start from an empty directory in your machine and copy/paste the following content directly to the OS prompt.

NOTE: Paste the copied context to the OS prompt directly. Do not paste to an editor.

```

mkdir -p ./example/types/tosca/example
cat <<EOF > ./example/types/tosca/example/example_service.tosca
"\$schema"      = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-
module#"
title          = "A resource type for example service"
package        = example
version        = "1.0"
description    = """
A resource type for an example service that creates sub-instances and use
random-generated strings as their name
"""
authors        = ["Royal Joe (rjoe@ciena.com)"]

resourceTypes {

    ExampleService {
        title = "Example Service"
        description = "An example service"
        derivedFrom = "tosca.resourceTypes.Root"
        properties {

            numInstance {
                title = "Number of Instances"
                description = "Number of instances to create"
                type = integer
                default = 2
                enum = [1, 2, 3]
            }

            namePrefix {
                title = "Name Prefix"
                description = "String to be prepended to the random string name for each
instance created"
                type = string
                default = "Name"
                updatable = true
            }

            initialNames {
                title = "Initial Names"
                description = "Initial names for the instances created"
                type = array
                items.type = string
                output = true
            }
        }
    }
}
EOF

```

2. Take a look at the Resource Type just created. The Resource Type of **example.resourceTypes.ExampleService** has 3 properties:

- **numInstance**, an integer with enum of `[1, 2, 3]` and defaulting to 2
- **namePrefix**, a string default to "Name". This property is updatable

- **initialNames**, an output property which is a array of strings
3. The expected behavior of an **ExampleService** is as follows:
- During Activation:
- a. When an **ExampleService** resource is created, it should in turn create **numInstance** resources of type **Instance**
 - b. The **data.myName** property of each **Instance** resource should start with the **namePrefix** value specified by the user, and follow by a colon
 - c. After the colon, each **Instance** resource should obtain a randomly-generated ID from a different **RandomString** resource
 - d. Finally, when all the **Instance** resources are activated, the **data.myName** property of each **Instance** resource should be collected and set to the **initialNames** property of the **ExampleService** resource as an output

During Update:

- a. If **namePrefix** is not changed, no action is needed
- b. The **data.myName** property of each **Instance** resource should be re-evaluated and updated to reflect the new **namePrefix** value specified by the user
- c. Each **Instance** resource should again consult the same **RandomString** resource it used during activation for the ID part of the property

During Termination:

- a. All resources created during activation should be deleted

We will next create the Service Templates, starting with the declarative style

Part 1: Declarative Service Template

1. Copy/paste the following content into OS prompt to create a Service Template that implements **ExampleService**

```
cat <<EOF > ./example/types/tosca/example/template_declarative.tosca
"\$schema"      = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-
module#"
title          = "An declarative template to implement example service"
package        = example
version        = "1.0"
description    = "An declarative template to implement example service."
authors        = ["Royal Joe (rjoe@ciena.com)"]

imports {
  Instance = tosca.resourceTypes.Noop
  RandomString = tosca.resourceTypes.RandomString
}

serviceTemplates {
```

```
ExampleServiceDeclarative {
    title = Declarative Implementation
    description = To demo an declarative implementation of ExampleService
    implements = example.resourceTypes.ExampleService

    resources {
        Instance1 {
            type = Instance
            properties {
                data {
                    myName {
                        join = [":", {getParam = namePrefix}, {getAttr = [RandStr1,
value]}]
                    }
                }
            }
            terminateBefore = RandStr1
        }

        Instance2 {
            createIf = { isGreaterEqual = [ { getParam = numInstance}, 2] }
            type = Instance
            properties {
                data {
                    myName {
                        join = [":", {getParam = namePrefix}, {getAttr = [RandStr2,
value]}]
                    }
                }
            }
            terminateBefore = RandStr2
        }

        Instance3 {
            createIf = { isEqual = [ { getParam = numInstance}, 3] }
            type = Instance
            properties {
                data {
                    myName {
                        join = [":", {getParam = namePrefix}, {getAttr = [RandStr3,
value]}]
                    }
                }
            }
            terminateBefore = RandStr3
        }

        RandStr1 {
            type = RandomString
        }

        RandStr2 {
            createIf = { isGreaterEqual = [ { getParam = numInstance}, 2] }
            type = RandomString
        }

        RandStr3 {
            createIf = { isEqual = [ { getParam = numInstance}, 3] }
        }
    }
}
```

```

        type = RandomString
    }
}

output {
    initialNames = [
        {extract = [{getAttr = [Instance1, data]}, ".$.myName", true]},
        {setOnlyIf = [{isGreaterEqual = [ {getParam = numInstance}, 2] }]},
        {extract = [{getAttr = [Instance2, data]}, ".$.myName", true]}]
        {setOnlyIf = [{isEqual = [ {getParam = numInstance}, 3] }], {extract =
        [{getAttr = [Instance3, data]}, ".$.myName", true]}]
    ]
}
}
EOF

```

- This file defines the Service Template **example.serviceTemplates.ExampleServiceDeclarative**, which implements the **ExampleService** Resource Type created in the previous step. This Service Template defines a **resources** section, which makes it declarative.
- In the **resources** section, 6 sub-resources are defined. Among them, 3 are of type **Instance**, and the other 3 are of type **RandomString**.

Note: **Instance** type is an imported name for Resource Type **tosca.resourceTypes.Noop**. This is a Resource Type that has a single object-type property and a built-in product. A **Noop** resource will become active when created, and will clear the differences automatically after an update. Using the **Noop** Resource Type is a convenient way to test the Service Template before hooking up with actual Resource Types.

- The resources with indexes 2 and 3 have different **createIf** conditions defined, so the template will create the second set of Instance and RandomString resources only when **numInstance** is greater or equal than 2. Similarly, the third set of resources will only be created when **numInstance** is equal to 3 in the user-created **ExampleService** resource.
- The **Instance** resources make use of a **join** directive together with **getParam** and **getAttr** directives to set the **data.myName** property. **getParam** gets data from the properties the user passed in, and **getAttr** gets data from the properties of a peer sub-resource.
- Since an **Instance** resource uses **getAttr** to obtain a property from the corresponding **RandomString** resource, an implicit **createAfter** dependency is created among the 2 resources. This means the template engine will create the **RandomString** first, wait for it to become active, before attempting to create the corresponding **Instance** resource.
- Notes on sub-resource dependencies:
 - Although there is dependency between the **Instance** and **RandomString** sub-resources of same index, there is no **createAfter** dependency between sets of resources that have different indexes. As a result, all the **RandomString** sub-resources may be created in parallel.
 - An explicit dependency can be specified with a **createAfter** directive. For more complex templates, it is best-practice to make use of **createAfter** directive to mark the dependencies explicitly (even though the dependencies can be determined implicitly), so the dependencies can be more easily understandable.

- A **createAfter** dependency does not automatically imply a **terminateBefore** dependency. In other words, there isn't an implicit termination order inferred from any of the other directives. If a specific termination order is required, the **terminateBefore** directive needs to be used to specify the termination order. If no direct or indirect **terminateBefore** dependency can be found between 2 sub-resources, the template engine will attempt to terminate them in parallel.
- An **output** block is specified to set the **initialNames** array property of the user-created **ExampleService** resource. It uses a combination of **extract** and **getAttr** directives to fetch the **data.myName** property from each **Instance** sub-resource.

Note that the **setOnlyIf** directive is used for the second and third rows with the same condition as the corresponding **createIf** used when creating the **Instance** resource. When the condition of **setOnlyIf** evaluates to false, the value is not set or added to the array.

2. Onboard the Resource Type and the Service Template.

```
bpocore onboard ./example
```

This command:

- clones the BPO git repo
- adds and commits the 2 new files
- pushes the changes back to BPO
- submits a pull request
- waits for BPO to validate the pull request, accept it, and integrate the new Resource Type and Service Template into the system

Wait for the command to complete

3. Verify the onboarded Resource Type

```
bpocore market api resource-types example.resourceTypes.ExampleService get
```

This command issues a *GET* call to */market/api/v1/resource-types/example.resourceTypes.ExampleService*.

Take a look at the compiled Resource Type with the 3 properties

4. Show the onboarded Service Template

```
bpocore market api type-artifacts example.serviceTemplates.ExampleServiceDeclarative get
```

This command issues a *GET* call to */market/api/v1/type-artifacts/example.serviceTemplates.ExampleServiceDeclarative*.

Take a look at the compiled Service Template

5. Create a product for the Resource Type. First, create product's content.

```
cat <<EOF > product_declarative.json
{
  "resourceTypeId" : "example.resourceTypes.ExampleService",
  "title" : "Example Service Declarative",
  "active" : true,
  "domainId" : "built-in",
  "providerData" : {
    "template" : "example.serviceTemplates.ExampleServiceDeclarative"
  }
}
EOF
```

Note that the name of the Service Template is specified in the **providerData** using the key **template**. This product links the Service Template and the Resource Type.

6. Create the product

```
bpoctl market api products post @product_declarative.json
```

This command calls the *POST* method to */market/api/v1/products* to create a product. BPO will respond with the product created with its ID as one of the attributes

7. Store the product ID for further use by copying the id field (without the quotes) and paste to a variable

```
D_PRODUCT_ID=<id>_from_the_response_without_the_quotes
```

8. With a product created, we can now create a ExampleService resource. Create the body of the resource

```
cat <<EOF > resource_svc_1.json
{
  "label": "Service1",
  "productId": "$D_PRODUCT_ID",
  "properties": {
    "numInstance": 2,
    "namePrefix": "Name"
  }
}
EOF
```

Note how the product ID is put into the content. Through the product ID, the system knows which type this resource is and which Service Template to execute when the resource is created. Also note that we specify the properties so that 2 **Instance** resources (along with 2 **RandomString** resources) will be created as a result, and **Name** will be used as the name prefix for each.

9. Create a resource

```
bpocore market api resources post @resource_svc_1.json
```

This command calls the *POST* method to */market/api/v1/resources* to create a resource. Similar to the product creation, BPO will return a resource with its ID. Note that the **orchState** of the resource is now **requested**. This is the state when the resource is created, before the Template Engine starts to process it.

10. Store the resource ID for further use

```
SVC1_ID=<id>_from_the_response_without_the_quotes
```

11. The template engine will start to process the resource. Keep issuing the command below until the **orchState** shows **active**

```
bpocore market api resources $SVC1_ID get
```

An example output when the **orchState** is **active**

Attribute	Value
autoClean	false
createdAt	"2016-05-04T05:07:16.244Z"
description	null
desiredOrchState	"active"
differences	[]
discovered	false
id	"57298384-f52d-4e20-be71-6443ffe4b05e"
label	"Service1"
nativeState	null
orchState	"active"
orderId	null
productId	"572982a3-79e7-4cbe-9803-ddb0ee4e8847"
properties.initialNames	["Name:9eb0cf7a7cfa3d58917b5cbbe991e65d", "Name:23c6b47655ff3239bb64461ce4b7fc4a"]
properties.namePrefix	"Name"
properties.numInstance	2
providerResourceId	null
reason	" "
resourceTypeId	"example.resourceTypes.ExampleService"
shared	false
tenantId	"7c7da24e-d2af-35c7-8f6d-d8d16c7f0738"
updatedAt	"2016-05-04T05:07:23.414Z"

Note that once the orchState becomes active, **properties.initialNames** is also set to a list of 2 strings that appear to match the format we expected. We will verify this when we look at the sub-resources created.

12. Since this is a simple template, it is possible that the resource goes to active when you first issue the get command.

Let's call a command to see the history of this service resource to look at the changes along the time

```
bpocore market api resources $SVC1_ID history get
```

As you can now guess, this command executes a *GET* call to */market/api/v1/resources/\$SVC1_ID/history* to get the history of a resource

Here's an example output.

time	userId	attrName	newValue
changeType			
"2016-05-04T17:26:43.278Z"	"admin"	"autoClean"	false
"Set"			
"2016-05-04T17:26:43.278Z"	"admin"	"desiredOrchState"	"active"
"Set"			
"2016-05-04T17:26:43.278Z"	"admin"	"discovered"	false
"Set"			
"2016-05-04T17:26:43.278Z"	"admin"	"id"	"572a30d3-4eca-4504-8962-0ff2fd0ea3b3"
			"Set"
"2016-05-04T17:26:43.278Z"	"admin"	"label"	"Service1"
"Set"			
"2016-05-04T17:26:43.278Z"	"admin"	"orchState"	"requested"
"Set"			
"2016-05-04T17:26:43.278Z"	"admin"	"productId"	"57298c94-b580-4a3d-aa24-5f521b328e9a"
			"Set"
"2016-05-04T17:26:43.278Z"	"admin"	"properties.namePrefix"	"Name"
"Set"			
"2016-05-04T17:26:43.278Z"	"admin"	"properties.numInstance"	2
"Set"			
"2016-05-04T17:26:43.278Z"	"admin"	"providerData"	" "
"Set"			
"2016-05-04T17:26:43.278Z"	"admin"	"reason"	" "
"Set"			
"2016-05-04T17:26:43.278Z"	"admin"	"shared"	false
"Set"			
"2016-05-04T17:26:43.278Z"	"admin"	"tenantId"	"7c7da24e-d2af-35c7-8f6d-d8d16c7f0738"
			"Set"
"2016-05-04T17:26:43.338Z"	"admin"	"orchState"	
"activating"			
"Updated"			
"2016-05-04T17:26:49.538Z"	"admin"	"orchState"	"active"
"Updated"			
"2016-05-04T17:26:49.538Z"	"admin"	"properties.initialNames"	["Name:86da172bdec3cb11cd72e74d9adb9a5b", "Name:68ee44ed8a5b2e99fb020267740abc21"]
			"Set"

We can see that the **orchState** of the resource was set to **activating** 50 milliseconds after the resource was created (by the template engine, which also used the same userId, "admin" in this case, as the initial request.) This roughly marks the time that the template engine started to process the resource. After 6 seconds, the template engine finished processing and set the **orchState** to **active**, along with the **properties.initialNames** output property.

13. Look at the sub-resources created by the Template Engine as a result of processing the Service Template. When processing a declarative Service Template, the Template Engine creates MadeOf relationships from the top resource to the sub-resources automatically. As a result, we can query the dependencies of the top resource to get to the sub-resources

```
bpoctl market api resources $SVC1_ID dependencies get
```

Here is an example output

label	desiredOrchState	providerResourceId	id	discovered	resourceTypeId	orchState
"Service1.Instance1"	"active"		"572a30d6-c076-4aac-af3e-b6a3da359e42"	false	"tosca.resourceTypes.Noop"	null
"Service1.Instance2"	"active"		"572a30d6-6dd8-4d85-baf2-d72f40e53ef1"	false	"tosca.resourceTypes.Noop"	null
"Service1.RandStr1"	"active"		"572a30d3-cd40-4910-ae3d-a0f5bdb98224"	false	"tosca.resourceTypes.RandomString"	null
"Service1.RandStr2"	"active"		"572a30d3-1bab-475a-81ca-d78d7e08f0e2"	false	"tosca.resourceTypes.RandomString"	null

Note that 4 sub-resources are created as a result of processing the Service Template, 2 of each types, as we expected.

14. Store the resource IDs for the first Instance/RandStr pair for further use

```
SVC1_INSTANCE1_ID=<id>_from_the_Service1.Instance1_without_the_quotes
```

```
SVC1_RANDSTR1_ID=<id>_from_the_Service1.RandStr1_without_the_quotes
```

15. Look at the random string first

```
bpoctl market api resources $SVC1_RANDSTR1_ID get
```

Note that it has a **properties.value** property which is a generated random string when this resource is created

16. Next, look at the instance resource

```
bpocore market api resources $SVC1_INSTANCE1_ID get
```

Note its **properties.data.myName** property and see that

- It is in the form of Name:ID
- The ID part matches the **properties.value** from the random string resource
- The **properties.data.myName** shows up as one of the values in the output array of the service resource above

We have verified that the declarative Service Template does achieve the expected behavior for activation.

Let's try to update the resource next. We will be using PATCH API call to a resource to change its properties. Alternatively, a PUT call can be used by passing the entire resource with the expected state.

17. Create the patch content. This sets the **namePrefix** property to **InstanceId**

```
cat <<EOF > patch.json
{
  "properties" : {
    "namePrefix" : "InstanceId"
  }
}
EOF
```

We expect that after the PATCH is called, the Instance sub-resources will eventually have its **data.myName** property changed to the format of **InstanceId:<id>**

18. Call the patch using bpocore-cli

```
bpocore market api resources $SVC1_ID patch @patch.json
```

Look at the return from bpocore, as the following example:

Attribute	Value
autoClean	false
createdAt	"2016-05-04T17:26:43.278Z"
description	null
desiredOrchState	"active"
differences	[{ "path": "/properties/namePrefix", "value": "Name", "op": "replace" }]
discovered	false
id	"572a30d3-4eca-4504-8962-0ff2fd0ea3b3"
label	"Service1"
nativeState	null
orchState	"active"
orderId	null
productId	"57298c94-b580-4a3d-aa24-5f521b328e9a"
properties.initialNames	["Name:86da172bdec3cb11cd72e74d9adb9a5b", "Name:68ee44ed8a5b2e99fb020267740abc21"]
properties.namePrefix	"InstanceId"
properties.numInstance	2
providerResourceId	null
reason	" "
resourceTypeId	"example.resourceTypes.ExampleService"
shared	false
tenantId	"7c7da24e-d2af-35c7-8f6d-d8d16c7f0738"
updatedAt	"2016-05-04T19:10:54.605Z"

Note that **properties.namePrefix** is now set to the new value we want after the patch. In addition, some **differences** now show up in the resource. The differences show what has been changed from the old state – it is a JSON patch that would bring the properties back to the **observed** state after applied to the properties.

When the Template Engine sees the patched resource, it looks at the new value and the differences to determine what needs to be done to move from the existing state to the desired state. In this case, it must reconstruct the **data.myName** property on the **Instance** resources based on the **namePrefix** change.

The template engine will formulate the new name and call a PATCH to the corresponding **Instance** resource. This will in turn create differences in each **Instance** resource and trigger the processing for the provider of that resource (which could be another template, or an Resource Adapter), and so on so forth. Once the provider of a resource verifies that the southbound state is indeed the same as the desired, it should "clear the differences".

The template engine monitors all sub-resources and clears the service resource differences when it confirms all the sub-resources have cleared their differences. This completes the update plan.

19. Verify this by first get the state of this resource again.

```
bpocore market api resources $SVC1_ID get
```

If everything goes well, you should eventually see the differences cleared (i.e., the **differences** property becomes the empty list []).

Note that the **initialNames** properties stays at the initial value. Since the output property is supposed to be set only once when the resource becomes active, it does not change after further updates.

20. View the history of the top level resource

```
bpocore market api resources $SVC1_ID history get
```

We can see that a couple of new entries has been added to the history. When the patch is called, **properties.namePrefix** and **differences** were set at the same time. Then after a while, the **differences** were cleared as the sub-resources all have their differences cleared.

21. Look at one of the **Instance** resources to verify

```
bpocore market api resources $SVC1_INSTANCE1_ID get
```

Note that **properties.data.myName** now starts with "Instanceld" as we would expect.

22. Look at the history of this Instance resource

```
bpocore market api resources $SVC1_INSTANCE1_ID history get
```

Note that near the end of the history, it has a similar pattern as the service level resource. The template engine patched this resource with the expected **myName** property which resulted in **differences**. The provider of the **Instance** type cleared that difference. Remember in this tutorial the **Instance** resource is a **Noop** type which accepts any change and clears the differences. For an actual RA-managed resource, a call would likely be made to the managed domain which would adjust configuration on the managed resource (e.g., device or VM).

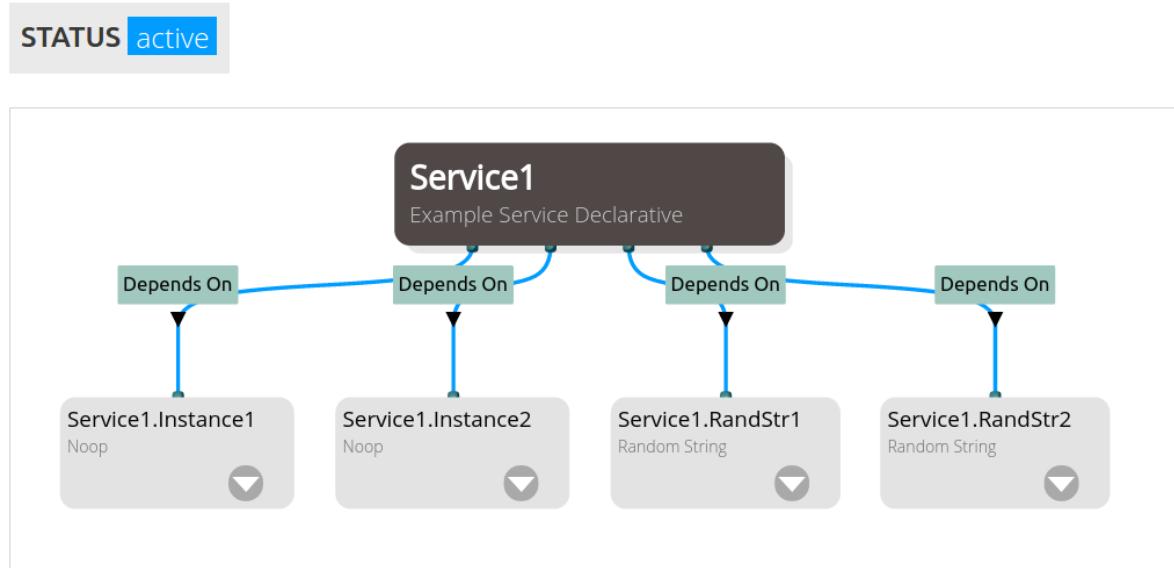
If we compare this history with the one we did earlier for the service resource, you will notice that the clearing of the differences on the **Instance** resource happens earlier than that on the service resource, since the template engine does not clear its own differences until it sees all the differences cleared on affected sub-resources.

23. Finally, look at the relationships between the service resources and sub-resources it creates. Go to the UI from browser (Please refer to the **Prerequisites** section at the beginning of this tutorial for setting up a dev UI), click on Domains→Planet Orchestrate→Resources (top right corner), and click on "Service1" and see the relationship graph between the Service1 resource and its sub-resources, as the

following picture. Feel free to click around in the UI.

ExampleService: Service1

Created On **05/04/16, 1:47:23 am**, Last Updated **05/04/16, 1:47:29 am**



24. Delete the service resource and verify everything gets deleted.

```
bpoctl market api resources $SVC1_ID delete
```

This will kick off the termination plan.

25. Execute the following command repeatedly and see resources get terminated until there are not any resources left.

```
bpoctl market api resources get
```

Remember we have explicit **terminateBefore** specified on the **Instance** resources on the Service Template, so the Template Engine will delete the **Instance** resources before it deletes the corresponding **RandomString** resource. If you executed the command fast enough, you would have seen the **Instance** resources gone first, followed by the **RandomString** resources, and eventually the service resource itself.

26. One could still view the history of a resource after it is deleted, as long as the ID is known.

```
bpoctl market api resources $SVC1_ID history get
```

Note that the delete call we made set the `**desiredOrchState**` to `**terminated**`. The Template Engine then started the termination plan by setting the `**orchState**` to `**terminating**` and called `delete` on the sub-resources in the `**terminateBefore**` dependency order. After all the sub-resources are deleted, the Template Engine set the `**orchState**` to `**terminated**` and Market removed the resource from the database and null'd out all its attributes.

You have successfully created a declarative Service Template and used it to manage the whole lifecycle of a service resource. We will next try the same using a imperative Service Template.

Part 2: Imperative Service Template

1. Start by creating an imperative Service Template

```

cat <<EOF > ./example/types/tosca/example/template_imperative.tosca
"\$schema"      = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-
module#"    # format version and validator
title          = "A sample type and template to demo imperative style operations"
package         = example
version         = "1.0"
description     = """
A type and a template to show how a callout to active and terminate can be used.
"""
authors        = ["Royal Joe (rjoe@ciena.com)"]

serviceTemplates {
  ExampleServiceImperative {
    title = Imperative Implementation
    description = To demo an imperative implementation of ExampleService
    implements = example.resourceTypes.ExampleService

    plans {
      activate {
        type = script
        language = python
        path = "types/tosca/example/example_service_activate.py"
      }
      terminate {
        type = script
        language = python
        path = "types/tosca/example/example_service_terminate.py"
      }
      update {
        type = script
        language = python
        path = "types/tosca/example/example_service_update.py"
        autoClearDifferences = false
      }
    }

    output {
      initialNames = { getAttr = [ activate, outputNames ] }
    }
  }
}
EOF

```

Looking at this Service Template, we can see that:

- It implements the same Resource Type as the Declarative one in Part 1.
- Instead of **resources**, it defines a **plans** section, which makes this an imperative Service Template.
- There are 3 items in the **plans** section: **activate**, **terminate**, and **update**. Each of the items specifies what to execute for the corresponding lifecycle operation.

Python is the only language supported as of now, so the **language** property needs to be **python**.

The **type** should be **script** if the specific script is to be executed locally within the bpcore container, or **remote** if the execution of the script is to be done in another micro-service. We use **script** in this tutorial, as we don't have an external script-execution service set up.

path specifies the relative path of each python script from the root directory of the BPO model-definitions git repo

Note that **autoClearDifferences = false** is specified in the **update** section. We will discuss this later

- d. A similar **output** section to that of the declarative Service Template is specified to set the **initialNames** properties. Instead of fetching the data from individual sub-resources, a **getAttr** is used to fetch the **outputNames** property from the **activate** plan. We will discuss how the output of the plan can be set by the activation script.
2. Before we make each of the scripts, let's create a utility script that can help us interface with BPO's API.

This script defines the helper functions that simplify the API operations like get, post, patch, and delete, and also some helper functions like create a resource and wait until it activates, or fail the plan, etc.

Since the Service Template only defines which scripts to execute in different lifecycle operations, it has no idea of sub-resources at the Service Template level. As a result, the scripts need to leave hints of how to find resources that it creates, or any dependencies between them. Adding relationships is one of the methods for this. By adding relationships, the resources become dependent on each other and they can also be found by following the dependencies.

Setting extra properties is another way when setting a relationship is not appropriate.

As an example, look at the **add_resource_wait_active** function below. This function creates a resource, and immediately after the resource is created, it creates a **MadeOf** relationship from the service level resource to the sub-resource just created. Same functionality is done automatically by the Template Engine when executing the declarative activation, but the script needs to take care of it in the imperative style.

Copy the following block and paste it directly to the OS prompt

```
cat <<EOF > ./example/types/tosca/example/porchplan.py
import json
import os
import requests
import sys
import time
import traceback

class Plan(object):

    market_path = '/bpocore/market/api/v1'

    uri_instance      = 'tosca.resourceTypes.Noop'
    url_randomstring = 'tosca.resourceTypes.RandomString'

    @classmethod
    def main(cls):
        logpath = os.environ['LOG_FILE']
```

```
with file(logpath, 'w') as log:
    try:
        print >> log, time.asctime()
        for (key, val) in sorted(os.environ.iteritems()):
            print >> log, '%40s=%s' % (key, val)
        log.flush()
        e = os.environ
        cls(log,
             e['API_URI'],
             e['RESOURCE_ID'],
             e['DOMAIN'],
             e["BP_USER_ID"],
             e["BP_TENANT_ID"],
             e["BP_ROLE_ID"],
             e["C_TRACE"],
             e["C_UPSTREAM"],
             e['OPERATION']).run()

    except Exception, e:
        print >> log, e.message
        print e.message
        traceback.print_exc(file=log)
        traceback.print_exc()
        sys.exit(42)

def __init__(self, logfile, uri, rid, domain, usr_id, tenant_id, role_ids,
c_trace, c_upstream, op):
    self.logfile = logfile
    self.rid = rid
    self.op = op
    self.domain = domain
    self.usr_id = usr_id
    self.tenant_id = tenant_id
    self.role_ids = role_ids
    self.c_trace = c_trace
    self.c_upstream = c_upstream
    self.uri = uri
    self.headers = {
        'content-type': 'application/json',
        'accept': 'application/json',
        'Cyan-Internal-Request': '1',
        'bp-user-id': self.usr_id,
        'bp-tenant-id': self.tenant_id,
        'bp-role-ids': self.role_ids,
        'C-Trace': self.c_trace,
        'C-Upstream': self.c_upstream
    }
    self.verify = True

def log(self, msg):
    print >> self.logfile, msg
    self.logfile.flush()

def get_url(self, comp, path):
    uri = self.uri + comp + path
    self.log("URL: %s" % uri)
    return uri

def mget(self, path):
```

```
url = self.get_url(self.market_path, path)
self.log("GET %s" % url)
res = requests.get(url, verify=self.verify, headers=self.headers)
self.log(" %d" % (res.status_code, ))
if res.status_code >= 300:
    self.log(" Error: %s" % res.text)
return res

def mpost(self, path, data=None):
    url = self.get_url(self.market_path, path)
    self.log("POST %s %s" % (url, data))
    res = requests.post(url, data=data, headers=self.headers,
verify=self.verify)
    self.log(" %d" % (res.status_code, ))
    if res.status_code >= 300:
        self.log(" Error: %s" % res.text)
    return res

def mpatch(self, path, data):
    url = self.get_url(self.market_path, path)
    self.log("PATCH %s %s" % (url, data))
    res = requests.patch(url, data=json.dumps(data), headers=self.headers,
verify=self.verify)
    self.log(" %d" % (res.status_code, ))
    if res.status_code >= 300:
        self.log(" Error: %s" % res.text)
    return res

def mput(self, path, data=None):
    url = self.get_url(self.market_path, path)
    self.log("PUT %s %s" % (url, data))
    res = requests.put(url, data=data, headers=self.headers,
verify=self.verify)
    self.log(" %d" % (res.status_code, ))
    if res.status_code >= 300:
        self.log(" Error: %s" % res.text)
    return res

def mdelete(self, path):
    url = self.get_url(self.market_path, path)
    self.log("DELETE %s" % (url,))
    res = requests.delete(url, headers=self.headers, verify=self.verify)
    self.log(" %d" % (res.status_code, ))
    if res.status_code >= 300:
        self.log(" Error: %s" % res.text)
    return res

def add(self, what, path, data):
    self.log("Adding %s" % what)
    r = self.mpost(path, json.dumps(data))
    self.log(" result: %s\n" % r.status_code)
    if r.status_code >= 210:
        self.fail_plan("Adding %s returns with code %d" % (what,
r.status_code))
    obj = r.json()
    self.log("%s:" % what)
    self.log(obj)
    return obj
```

```

def get_products_by_type_and_domain(self, resource_type_uri, domain_id):
    return [
        p for p in
self.mget('/products?includeInactive=true').json()['items']
        if p['resourceTypeId'] == resource_type_uri and p['domainId'] ==
domain_id
    ]

def await_till(self, what, condFunc, interval = 1.0, max = 90.0):
    t0 = time.time()
    while True:
        if condFunc():
            break
        remaining = (t0 + max) - time.time()
        if remaining < 0:
            self.fail_plan("Timed out while waiting for: %s" % what)
            self.log("Waiting %s for %s (remaining time %s)" % (interval, what,
remaining))
        time.sleep(interval)
    self.log("Completed: %s" % what)

def await_resource_states(self, what, resourceId, states = ['active'],
interval = 1.0, max = 90.0):
    t0 = time.time()
    status = ""
    while True:
        resource = self.mget('/resources/%s' % resourceId).json()
        status = resource['orchState']
        if status in states:
            break
        if status == 'failed':
            self.fail_plan("%s failed with reason '%s'" % (what,
resource['reason']))
        remaining = (t0 + max) - time.time()
        if remaining < 0:
            self.fail_plan("Timed out while waiting for: %s to be in %s" %
(what, states))
            self.log("Waiting %s for %s to be in state %s (remaining time %s)" %
(interval, what, states, remaining))
        time.sleep(interval)
    self.log("Completed: %s %s" % (what, status))

def await_differences_cleared(self, what, resourceId, interval=1.0,
                               max=90.0):
    t0 = time.time()
    status = ""
    while True:
        resource = self.mget('/resources/%s' % resourceId).json()
        differences = resource.get("differences", [])
        if not differences:
            break
        if status == 'failed':
            self.fail_plan("%s failed with reason '%s'" % (what,
resource['reason']))
        remaining = (t0 + max) - time.time()
        if remaining < 0:
            self.fail_plan("Timed out while waiting for: %s differences to
clear" % what)
            self.log("Waiting %s for %s differences to clear (remaining time %s)"
```

```
% (interval, what, remaining))
    time.sleep(interval)
    self.log("Completed: %s %s" % (what, status))

def add_resource_wait_active(self, i, title_prefix, product_id, parent_id,
props = {}):

    # CONSTRUCT THE LABEL FOR THE NEW RESOURCE BASED IN THE TITLE PREFIX
    label = title_prefix + str(i)

    # CALL MARKET TO CREATE THE NEW RESOURCE
    new_resource = self.add("Resource", "/resources", {
        "label": label,
        "productId": product_id,
        "properties": props
    })

    # GET THE ID OF THE CREATED RESOURCE
    new_resource_id = new_resource['id']
    self.log("%s ID: %s" % (label, new_resource_id))

    # ADD A MADEOF RELATIONSHIP FROM THIS RESOURCE TO THE NEWLY-CREATED
    RESOURCE
    self.add("Relationship", "/relationships", {
        "relationshipTypeId": "tosca.relationshipTypes.MadeOf",
        "sourceId": parent_id,
        "requirementName": "composed",
        "targetId": new_resource_id,
        "capabilityName": "composable",
        "orchState": "active"
    })

    # WAIT FOR THE NEW RESOURCE TO BECOME ACTIVE
    self.await_resource_states(new_resource, new_resource_id, interval=5,
max=180)

    # GET THE ACTIVATED RESOURCE FROM MARKET AND RETURN
    return self.mget("/resources/%s" % new_resource_id).json()

def not_found_or_terminated(self, id):

    ret = self.mget('/resources/%s' % id)

    if ret.status_code == 404:
        self.log('Resource is gone')
        return True

    if ret.status_code != 200:
        return False

    if ret.json()['orchState'] == "terminated":
        self.log('Resource is in terminated state')
        return True

    return False

def fail_plan(self, message=None):
    if message:
        self.log(message)
```

```

    sys.stderr.write(message)
    sys.exit(1)

EOF

```

3. Create the activation script

The **run** method will be executed when the script is called. When it is called, the ID of the service resource is passed to the script alone with some other environmental variables, but the content of the resource is not. As a result, the first step of the script execution is usually to call back to market to get the details of the service resource, as indicated in the code.

After getting the resource, it extracts the numInstance and namePrefix from the properties.

Then it tries to find the product IDs for the **Instance** and **RandomString** using the Resource Type ID of each, with the help of the utility function we created earlier.

When the products are both found, the script starts by creating **numInstance** numbers of **RandomString** resources using a for loop and waits for each to become active. This in turn creates a relationship from the service resource to each sub-resource, as described in the previous section.

Next, the script will formulate the name for the **Instance** resource and use it to create the actual resource. One thing to note here – remember the expected behavior indicates that the same **RandomString** resource needs to be consulted again during the update. In the declarative Service Template, the Template Engine left hints implicitly and is able to infer the actual resource from the configuration. However, hints need to be created for the update script to match a **Instance** resource to its **RandomString** resource. The script is therefore creating a **MadeOf** relationship from the **Instance** resource to its **RandomString** resource.

When all the resources are finally active, the script will collect data from the **Instance** resources for its output. Note that the script collects the name list, and constructs a dictionary with key **outputNames**, and prints it to the STDOUT using **print** command. The Template Engine will gather output from the print command and use it as the output of the plan. In this case, the output of the activation plan will be a single property called **outputNames** with values being a list of names from the **Instance** resources. This output is then harvested by the Service Template itself and eventually set to the **initialNames** property of the user-created resource.

Note: Since STDOUT is used for output, when developing a scripted plan NEVER print any debug messages. Instead, use the **self.log** method for them.

Copy the following code and paste to the OS prompt

```

cat <<EOF > ./example/types/tosca/example/example_service_activate.py
from porchplan import Plan
import json

class Activate(Plan):
    """ Activation of an example service.
    """

```

```

def run(self):

    # EXTRACTING DATA FROM REQUESTED RESOURCE
    self.log("Grabbing resource")
    resource = self.mget("/resources/%s" % self.rid).json()
    self.log(resource)

    top_label = resource['label']
    props = resource['properties']
    name_prefix = props['namePrefix']
    num_instance = props['numInstance']

    domain_id = 'built-in'

    # FIND PRODUCT IDS FOR INSTANCE AND RANDOMSTRING TYPES
    instance_products =
    self.get_products_by_type_and_domain(self.uri_instance, domain_id)
    if len(instance_products) == 0:
        self.fail_plan("Product for Instance type is not found in domain %s"
% domain_id)
    else:
        instance_product_id = instance_products[0]["id"]
        self.log("Instance product id: %s" % instance_product_id)

        random_string_products =
    self.get_products_by_type_and_domain(self.url_randomstring, domain_id)
    if len(random_string_products) == 0:
        self.fail_plan("Product for Random String is not found in domain %s"
% domain_id)
    else:
        random_string_product_id = random_string_products[0]["id"]
        self.log("Random String product id: " + random_string_product_id)

    # CREATE RANDOM STRING RESOURCES AND WAIT FOR THEM TO BE ACTIVE

    self.log("Creating Random String Resource(s)... ")
    random_strings = map(
        lambda i: self.add_resource_wait_active(i + 1,
"{} .RandStr".format(top_label),
                                         random_string_product_id,
                                         resource['id']),
        range(num_instance)
    )

    # CREATE INSTANCE RESOURCES AND WAIT FOR THEM TO BE ACTIVE

    self.log("Creating Instance Resource(s)... ")
    instances = []
    for (index, random_string) in enumerate(random_strings):
        # CREATE THE INSTANCE RESOURCE
        instance = self.add_resource_wait_active(index + 1,
"{} .Instance".format(top_label),
                                         instance_product_id,
                                         resource['id'],
                                         {

```

```

        "data": {
            "myName": ""
        }
    })
}

instances.append(instance)

# ADD A MADEOF RELATIONSHIP FROM INSTANCE RESOURCE TO RANDOM STRING
RESOURCE
    self.add("Relationship", "/relationships", {
        "relationshipTypeId": "tosca.relationshipTypes.MadeOf",
        "sourceId": instance['id'],
        "requirementName": "composed",
        "targetId": random_string['id'],
        "capabilityName": "composable",
        "orchState": "active"
    })

# CONSTRUCT THE OUTPUTS
names = [r['properties']['data']['myName'] for r in instances]
data = dict(outputNames = names)
self.log("Sending back as response: %s" % str(data))

# PRINT THE OUTPUT VALUE TO STDOUT
print json.dumps(data)

def formulate_name(self, name_prefix, resource):
    return name_prefix + ':' + resource['properties']['value']

if __name__ == '__main__':
    Activate.main()

EOF

```

4. Create the termination script.

The script starts by fetching the resource and relevant products, similar to the activation script. It then calls the **/resources/<id>/dependencies** API to get all sub-resources. This API call follows the relationship in background, so if the sub-resources were created without adding a relationship, they will never be found.

Remember that there is a **terminateAfter** dependency in the declarative case. The script needs to do it directly – it finds the **Instance** resources from the dependencies and delete them first, verifies they are gone, and then deletes the **RandomString** ones. The function returns normally if everything goes well, and the Template Engine terminates the top-level resource as a result.

```

cat <<EOF > ./example/types/tosca/example/example_service_terminate.py
from porchplan import Plan

class Terminate(Plan):
    """ Termination of the example service.
    """

```

```

def run(self):

    # EXTRACTING DATA FROM REQUESTED RESOURCE

    self.log("Grabbing resource")
    resource = self.mget("/resources/%s" % self.rid).json()
    self.log(resource)

    # FIND PRODUCT IDS FOR INSTANCE AND RANDOMSTRING TYPES
    domain_id = 'built-in'
    instance_products =
    self.get_products_by_type_and_domain(self.uri_instance, domain_id)
    if len(instance_products) == 0:
        self.fail_plan("Product for Instance type is not found in domain %s"
% domain_id)
    else:
        instance_product_id = instance_products[0]["id"]
        self.log("Instance product id: %s" % instance_product_id)

    random_string_products =
    self.get_products_by_type_and_domain(self.url_randomstring, domain_id)
    if len(random_string_products) == 0:
        self.fail_plan("Product for Random String is not found in domain %s"
% domain_id)
    else:
        random_string_product_id = random_string_products[0]["id"]
        self.log("Random String product id: " + random_string_product_id)

    # GET SUB-RESOURCES
    dependencies = self.mget('/resources/%s/dependencies' %
self.rid).json()['items']

    instances = [r for r in dependencies if r['productId'] ==
instance_product_id]
    self.log("Found %d instances" % len(instances))
    random_strings = [r for r in dependencies if r['productId'] ==
random_string_product_id]
    self.log("Found %d random strings" % len(random_strings))

    # DELETE INSTANCES
    for r in instances:
        # CALL MARKET TO DELETE THE SUB-RESOURCE
        id = r['id']
        res = self.mdelete('/resources/%s' % id)
        self.log('Deleting instance %s: %s' % (id, res.status_code))

        # WAIT FOR THE DELETED RESOURCE TO BE GONE FROM MARKET
        self.await_till("terminate instance %s" % r['label'],
                        lambda: self.not_found_or_terminated(id) == True,
                        interval=5, max=160)

    # DELETE RANDOM STRINGS
    for r in random_strings:
        id = r['id']
        # CALL MARKET TO DELETE THE SUB-RESOURCE
        res = self.mdelete('/resources/%s' % id)
        self.log('Deleting random string %s: %s' % (id, res.status_code))

        # WAIT FOR THE DELETED RESOURCE TO BE GONE FROM MARKET

```

```

        self.await_till("terminate instance %s" % r['label'],
                      lambda: self.not_found_or_terminated(id) == True,
                      interval=5, max=160)

if __name__ == '__main__':
    Terminate.main()

EOF

```

5. Finally, create the update script

After the script fetches the resource, it looks for any differences on the resource. If there are no differences, it will return right away.

If there is indeed differences, it should analyze the differences and decide what to do to clear them. In our simple case, since only the **namePrefix** property is updatable, the script can assume any differences come from changing it.

To set the **data.myName** property to the new value, the script finds all **Instance** resources by calling dependencies. For each **Instance** resource, it calls dependencies again to find its **RandomString** resource. The relationship added by the activation script makes it possible to find a corresponding **RandomString** resource from a **Instance** resource.

After constructing the new name from the updated **namePrefix** and the data from the **RandomString** resource, the script issues a patch to the **Instance** resource. As indicated in the Declarative part, a patch to the **Instance** (which is Noop type) will also create differences on that resource, which will then get cleared automatically. The script needs to wait for the difference to clear before going on.

After all the resources it patches have their differences cleared, the script can finally return. Before it returns, the top resource's differences need to be cleared to indicate the patch is a success.

Remember the **autoClearDifferences** settings when we were creating the Service Template in the first step of this part. When it is set to **false**, as what we have here, the script needs to call a patch to market to clear the differences. This is done by calling a PATCH or PUT to **/resources/<id>/observed** with the same properties as the desired one. If the **autoClearDifferences** is set to true, however, this step can be skipped and the Template Engine will clear the top-level differences as long as the script returns normally. There are usecases where the desired behavior is not to automatically clear the differences when the script completes successfully, in those cases, it is necessary to set this option to false.

```

cat <<EOF > ./example/types/tosca/example/example_service_update.py
from porchplan import Plan

class Update(Plan):
    """ Update of the example service.
    """

    def run(self):
        # EXTRACTING DATA FROM REQUESTED RESOURCE

```

```

        self.log("Grabbing resource")
        resource      = self.mget("/resources/%s" % self.rid).json()
        self.log(resource)

        # IF THERE IS NO DIFFERENCES, NO NEED TO UPDATE
        if resource['differences'] == []:
            self.log("No difference observed. No need to update")
            return

        props      = resource['properties']
        name_prefix = props['namePrefix']

        domain_id = 'built-in'

        # FIND THE INSTANCE RESOURCES
        dependencies = self.mget('/resources/%s/dependencies' %
self.rid).json()['items']
        self.log(dependencies)

        instance_products =
        self.get_products_by_type_and_domain(self.uri_instance, domain_id)
        if len(instance_products) == 0:
            self.fail_plan("Product for Instance is not found in domain %s" %
domain_id)
        else:
            instance_product_id = instance_products[0]["id"]
            self.log("Instance product id: %s" % instance_product_id)

        instance_resources = [d for d in dependencies if d['productId'] ==
instance_product_id]

        # UPDATE THE INSTANCE RESOURCES TO THE NEW NAME
        for instance_resource in instance_resources:
            instance_id = instance_resource['id']

            # FIND THE RANDOM STRING RESOURCE BY FOLLOWING THE RELATIONSHIPS
            rand_string_resource = self.mget('/resources/%s/dependencies' %
instance_id).json()['items'][0]

            # CONSTRUCT THE NEW NAME
            newName = self.formulate_name(name_prefix, rand_string_resource)

            propsPatch = {
                "properties": {
                    "data": {
                        "myName": newName
                    }
                }
            }

            # PATCH THE INSTANCE RESOURCE
            self.mpatch('/resources/%s' % instance_id, propsPatch)

            # WAIT UNTIL THE INSTANCE RESOURCE'S DIFFERENCE CLEARS
            self.await_differences_cleared('Instance', instance_id)

        # CLEAR DIFFERENCES OF THE TOP LEVEL RESOURCE
        clear_diff_patch = {
            "properties": resource['properties'],

```

```

        "orchState": "active",
        "reason": ""
    }
    self.mpatch('/resources/%s/observed' % resource['id'], clear_diff_patch)

def formulate_name(self, name_prefix, resource):
    return name_prefix + ':' + resource['properties']['value']

if __name__ == '__main__':
    Update.main()

EOF

```

6. Onboard these scripts and the Service Template to BPO

```
bpocore onboard ./example
```

This onboards the extra files that we put in to the system.

7. Verify the Service Template is in Market

```
bpocore market api type-artifacts
example.serviceTemplates.ExampleServiceImperative get
```

Note that plans, instead of resources, show up in the compiled template

8. We will follow a similar path as we did for the Declarative Template – create a product, create a resource, patch the resource, and delete it.

First, create the body for the product that links the new template to the service.

```

cat <<EOF > product_imperative.json
{
    "resourceTypeId" : "example.resourceTypes.ExampleService",
    "title" : "Example Service Declarative",
    "active" : true,
    "domainId" : "built-in",
    "providerData" : {
        "template" : "example.serviceTemplates.ExampleServiceImperative"
    }
}
EOF

```

9. Create the product

```
bpocore market api products post @product_imperative.json
```

This creates a different product from the one we created in the declarative case, but they are both products of **ExampleService** Resource Type

10. Store the product ID for further use

```
I_PRODUCT_ID=<id>_from_the_response_without_the_quotes
```

11. Create the body of the resource

```
cat <<EOF > resource_svc_2.json
{
    "label": "Service2",
    "productId": "$I_PRODUCT_ID",
    "properties": {
        "numInstance": 2,
        "namePrefix": "Name"
    }
}
EOF
```

Note that the resource has the same properties as the one in the declarative case, but with a different product ID. The resource creation will cause the imperative Service Template to be executed.

12. Create the resource

```
bpocore market api resources post @resource_svc_2.json
```

13. Store the resource ID for further use

```
SVC2_ID=<id>_from_the_response_without_the_quotes
```

14. The template engine will start to process the resource. Keep issuing the command below until the **orchState** shows **active**

```
bpocore market api resources $SVC2_ID get
```

An example output when the **orchState** is **active**

Attribute	Value
autoClean	false
createdAt	"2016-05-05T00:25:40.936Z"
description	null
desiredOrchState	"active"
differences	[]
discovered	false
id	"572a9304-b1a4-49d6-94ca-793c3a37ab8c"
label	"Service2"
nativeState	null
orchState	"active"
orderId	null
productId	"572a8dae-1f88-41b7-a05c-19446c4ae92b"
properties.initialNames	["Name:49e294fb632b2dcd4261bf340e7ea173", "Name:23556ccf6b5836a21724e31a7cc5cfc5"]
properties.namePrefix	"Name"
properties.numInstance	2
providerResourceId	null
reason	" "
resourceTypeId	"example.resourceTypes.ExampleService"
shared	false
tenantId	"7c7da24e-d2af-35c7-8f6d-d8d16c7f0738"
updatedAt	"2016-05-05T00:26:01.365Z"

This has a similar result to the declarative one.

15. Look at the instance resource's history next

```
bpcore market api resources $SVC2_ID history get
```

Here's an example output.

time	userId	attrName	newValue
changeType			
"2016-05-05T00:25:40.936Z"	"admin"	"autoClean"	false
"Set"			
"2016-05-05T00:25:40.936Z"	"admin"	"desiredOrchState"	"active"
"Set"			
"2016-05-05T00:25:40.936Z"	"admin"	"discovered"	false
"Set"			
"2016-05-05T00:25:40.936Z"	"admin"	"id"	"572a9304-b1a4-49d6-94ca-793c3a37ab8c"
			"Set"
"2016-05-05T00:25:40.936Z"	"admin"	"label"	"Service2"
"Set"			
"2016-05-05T00:25:40.936Z"	"admin"	"orchState"	"requested"
"Set"			
"2016-05-05T00:25:40.936Z"	"admin"	"productId"	"572a8dae-1f88-41b7-a05c-19446c4ae92b"
			"Set"
"2016-05-05T00:25:40.936Z"	"admin"	"properties.namePrefix"	"Name"
"Set"			
"2016-05-05T00:25:40.936Z"	"admin"	"properties.numInstance"	2
"Set"			
"2016-05-05T00:25:40.936Z"	"admin"	"providerData"	" "
"Set"			
"2016-05-05T00:25:40.936Z"	"admin"	"reason"	" "
"Set"			
"2016-05-05T00:25:40.936Z"	"admin"	"shared"	false
"Set"			
"2016-05-05T00:25:40.936Z"	"admin"	"tenantId"	"7c7da24e-d2af-35c7-8f6d-d8d16c7f0738"
			"Set"
"2016-05-05T00:25:40.974Z"	"admin"	"orchState"	
"activating"			
"Updated"			
"2016-05-05T00:26:01.365Z"	"admin"	"orchState"	"active"
"Updated"			
"2016-05-05T00:26:01.365Z"	"admin"	"properties.initialNames"	["Name:49e294fb632b2dcd4261bf340e7ea173", "Name:23556ccf6b5836a21724e31a7cc5cf5"]
			"Set"

This also looks similar to the Declarative one. The resource started in the **requested** state, went to **activating** state, and eventually settled in **active** state after around 20 seconds and has the **initialNames** property set.

16. Look at the dependencies of the service resource

```
bpocore market api resources $SVC2_ID dependencies get
```

label	desiredOrchState	providerResourceId	id	discovered	resourceTypeId	orchState
"Service2.Instance1"	"active"		"572a930f-9413-44b8-bf27-c7434fb26d44"	false	"tosca.resourceTypes.Noop"	null
"Service2.Instance2"	"active"		"572a9314-42bc-48dc-b44e-8b7a4fb15b97"	false	"tosca.resourceTypes.Noop"	null
"Service2.RandStr1"	"active"		"572a9305-46f7-4d80-8cf8-3c0823b886c0"	false	"tosca.resourceTypes.RandomString"	null
"Service2.RandStr2"	"active"		"572a930a-b9b6-4983-a1cc-1b4278b6f625"	false	"tosca.resourceTypes.RandomString"	null

As explained above, the dependencies are collected by following relationships from the service level resource. For the declarative case we get this automatically, but in the imperative case, relationships needs to be explicitly constructed (which we did) for this to show up.

17. Store the resource IDs for the first Instance/RandStr pair for further use

```
SVC2_INSTANCE1_ID=<id>_from_the_Service2.Instance1_without_the_quotes
```

```
SVC2_RANDSTR1_ID=<id>_from_the_Service2.RandStr1_without_the_quotes
```

18. Look at the random string first

```
bpocore market api resources $SVC2_RANDSTR1_ID get
```

Note that it has a **properties.value** property which is a generated random string when this resource is created

19. Look at the instance resource

```
bpoctl market api resources $SVC2_INSTANCE1_ID get
```

Verify **properties.data.myName** is set correctly and the ID part matches the value from the RandStr

We have verified that the activation plan does achieve the expected behavior.

- Let's try to update the resource next by using the same patch json we created for the first part.

```
bpoctl market api resources $SVC2_ID patch @patch.json
```

Attribute	Value
autoClean	false
createdAt	"2016-05-05T00:25:40.936Z"
description	null
desiredOrchState	"active"
differences	[{ "path": "/properties/namePrefix", "value": "Name", "op": "replace" }]
discovered	false
id	"572a9304-b1a4-49d6-94ca-793c3a37ab8c"
label	"Service2"
nativeState	null
orchState	"active"
orderId	null
productId	"572a8dae-1f88-41b7-a05c-19446c4ae92b"
properties.initialNames	["Name:49e294fb632b2dcd4261bf340e7ea173", "Name:23556ccf6b5836a21724e31a7cc5cf5"]
properties.namePrefix	"InstanceId"
properties.numInstance	2
providerResourceId	null
reason	""
resourceTypeId	"example.resourceTypes.ExampleService"
shared	false
tenantId	"7c7da24e-d2af-35c7-8f6d-d8d16c7f0738"
updatedAt	"2016-05-05T00:32:47.311Z"

Note that the resource has the same differences as in the declarative case.

- Execute the following command until the **differences** are cleared (i.e. become [])

```
bpoctl market api resources $SVC2_ID get
```

22. Look at one of the **Instance** resources

```
bpocore market api resources $SVC2_INSTANCE1_ID get
```

Verify that **properties.data.myName** is set to the expected value and differences are cleared.

23. If interested, look at the history of the service and Instance resources to see the change sequences.
The result should be very similar to that of the Declarative case.

```
bpocore market api resources $SVC2_ID history get
```

```
bpocore market api resources $SVC2_INSTANCE1_ID history get
```

24. Switch to the UI to look at the relationship graph.

Click on Domains→Planet Orchestrate→Resources (top right corner), and click on "Service2" and see the relationship graph between the Service2 resource and its sub-resources.

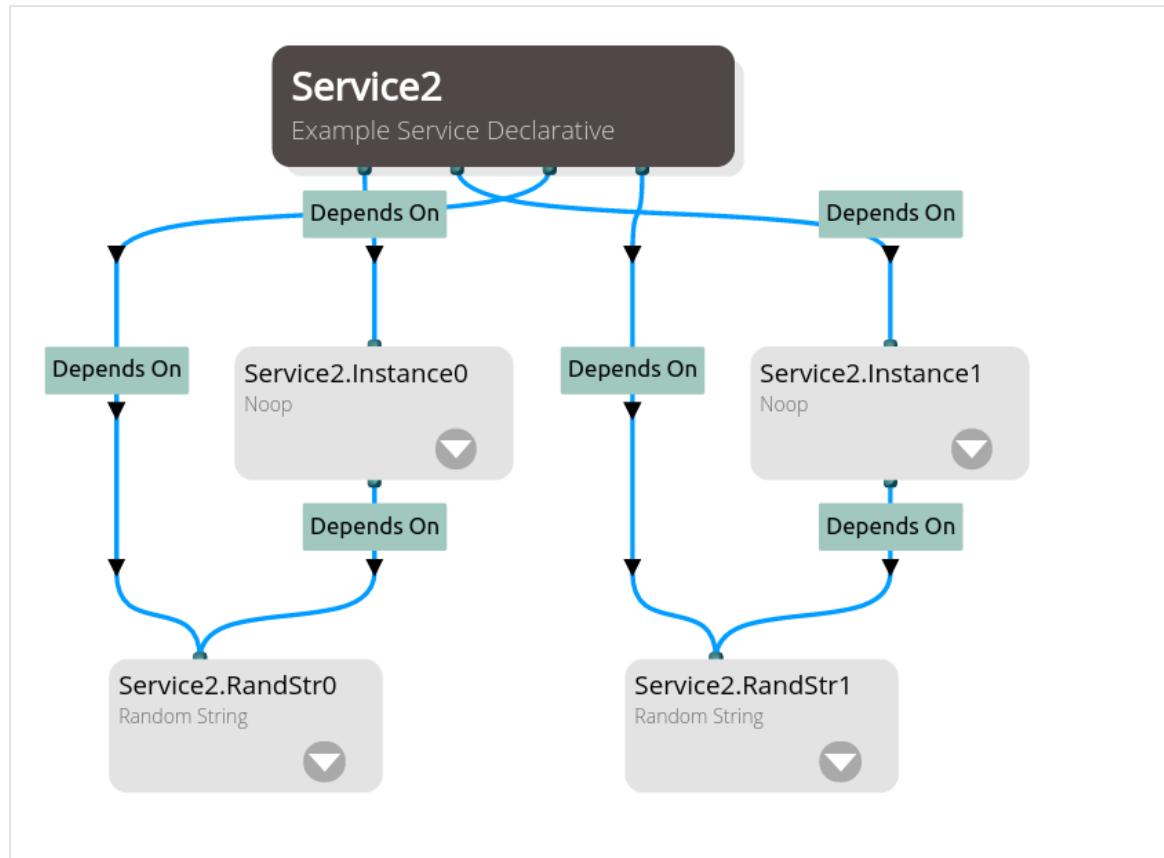
Click on the small triangle in the lower right corner of the 2 Instance resources to expand the relationship further.

The relationship graph should look similar to the following. Remember the extra relationships were added by the activation script so the script can later identify a specific **RandomString** resource from an **Instance** resource

ExampleService: Service2

Created On **05/04/16, 3:34:18 am**, Last Updated **05/04/16, 3:36:48 am**

STATUS active



25. Delete the resource

```
bpoctl market api resources $SVC2_ID delete
```

26. Execute the following command repeatedly and see the resources disappear one by one until nothing left

```
bpoctl market api resources get
```

Congratulations, you have finished the Imperative Service Template tutorial.

Summary

In this tutorial, you have learned to use both declarative and imperative Service Templates to implement the same Resource Type and active same end results.

A single declarative template is used to handle all the activation, update, and termination plans, while in the imperative case a template plus 3 scripts are used.

- In the declarative case, the relationships are created automatically and the order of execution is determined by a topological sorting based on the implicit or explicit specification of dependencies. In the implicit case, the order is controlled by the code flow explicitly.
- In the declarative case, independent sub-resources are kicked off simultaneously while in the imperative case, the resources are usually created one-by-one in sequence. If you noticed, the time to activate/terminate/update in the imperative case is much longer than that of the declarative case. The script could be coded in a way to allow execution in parallel, but more effort is needed to implement parallel activation and the complexity is more error-prone.
- Finally, the Template Engine knows about each sub-resources configured and how to find it in Market, while the scripts need to leave traces so a specific resource can be identified later (usually in the update script).

The imperative template does have its benefits. First, it is very flexible. If you noticed from the tutorial, although we limit the **numInstance** property to the enum of [1,2,3], the implementation of the imperative scripts are actually not limited to 3. The same cannot be done easily using the declarative syntax. The sub-resources need to be defined explicitly (for now) in the declarative case. The scripts can potentially have more branches based on different conditions during its execution, such as conditional rollback or selective update, etc.

In short, the declarative style provides a simple and clean way for laying out sub-resources, a simple rollback mechanism if anything goes wrong, and a code-free way to compose a service. The imperative is more powerful, but can potentially become complex and harder to comprehend at the first sight.

Developing Remote Plans with script-dev

This tutorial discusses development of service templates and remote scripts with the bpocore-dev and script-dev Docker containers, which parallel the bpocore and scriptplan apps. The -dev Docker images provide a development environment suitable for a developer laptop.

NOTE

This tutorial assumes that you have followed the [bpocore CLI Bootstrap Tutorial](#) and therefore already have a running bpocore-dev container.

It requires script-dev 2.1.4 or later (or Orchestrate 17.06.2).

Running script-dev

First, run the script-dev container:

```
docker run --name=script-dev --rm -it --link bpocore-dev:bpocore \
--link bpocore-dev:blueplanet --link bpocore-dev:kafka \
artifactory.ciena.com/blueplanet/script-dev:2.1.4
```

The --link flags tell Docker to create entries in /etc/hosts with the IP address of the bpocore-dev container.

Open another terminal to tell bpocore-dev how to access script-dev:

```
docker inspect --format='{{.NetworkSettings.IPAddress}}' scriptplan' script-dev \
| docker exec -i bpocore-dev tee -a /etc/hosts
```

Now bpocore-dev can execute remote scripts.

Create Definition Modules

Create a directory tree containing a [Service Template](#) and Python module implementing a custom [activate plan](#). It will look like this:

```
model-definitions
  --- requirements.txt  -
  --- plans
    --- __init__.py    -
    --- reverser.py    -
  --- types
    --- tosca
      --- reverser.tosca -
```

☒ Remote plan dependencies are listed in requirements.txt.

- ☒ The presence of `__init__.py` makes the `plans` directory an importable Python package. Putting your plan code in a package will help you keep it organized.
- ☒ The code to be run `reverser.py` contains the remote plan script implementation. This file is importable in Python as `plans.reverser`.
- ☒ `reverser.tosca` contains a **resource type** declaration and a **service template** which implements that type.

TIP

Unlike legacy plans, remote plans must be importable Python modules. This means that they must be named like [Python identifiers](#). Play it safe by keeping to lowercase A through Z, digits, and underscores – no hyphens or initial digits! `foo/bar.py` is okay, but `foo-bar/2baz.py` will fail to import.

First, create the directory structure:

```
mkdir -p model-definitions/plans model-definitions/types/tosca
```

And the Python package:

```
touch model-definitions/plans/__init__.py
```

The script itself is quite simple (click the link to download):

[model-definitions/plans/reverser.py](#)

```
from plansdk.apis.plan import Plan

class Activate(Plan):
    """
    An activate plan which reverses the string property "forward".
    """
    def run(self):
        resource_id = self.params['resourceId']
        resource = self.bpo.resources.get(resource_id)
        backward = ''.join(reversed(resource['properties']['forward']))
        return {"backward_value": backward}
```

TIP

Usually a real-world plan will create sub-resources, so you would also need to provide an **imperative terminate plan**. Otherwise the **terminate operation** would fail, leaving the resource in a *failed* state.

The `requirements.txt` file specifies where to acquire the Python libraries the script uses. In this case it only requires the `plansdk` library:

model-definitions/requirements.txt

```
--index-url https://artifactory.ciena.com/api/pypi/blueplanet-pypi/simple  
plansdk ~= 2.1
```

The first line of this file indicates the URL of the Python package index to use. The `plansdk` Python library is not on the public Python package index (`pypi.python.org`), so the `requirements.txt` file must specify where it can be acquired.

The final line specifies the latest `plansdk` in the `2.1.x` release series. `plansdk` is [semantically versioned](#), so we exclude version 3, as it may not be compatible. You may prefer to pin a specific version with `==`, like `plansdk == 2.1.3`.

See the [pip requirements.txt file format reference](#) for more information on what can be found in this file.

Finally the resource type and service template:

model-definitions/types/tosca/reverser.tosca

```

"$schema"      = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-module#"
title         = "Example of a remote activate script"
package        = example
version        = "1.0"
description    =
authors        = [ "Jane Dev <jd@example.com>" ]

resourceTypes.Reverser {
    title = "String Reverser"
    description = "A resource type which reverses a string via a remote plan."
    derivedFrom = tosca.resourceTypes.Root
    properties {
        forward {
            title = Forward String
            description = "This property is provided by the user when instantiating the
resource"
            type = string
            config = true
        }
        backward {
            title = Backward String
            description = "This property is filled in by the activate plan"
            type = string
            output = true
        }
    }
}

serviceTemplates.Reverser {
    title = String Reverser Implementation
    description = "A service template which reverses a string on activation."
    implements = example.resourceTypes.Reverser

    plans {
        activate {
            type = remote
            language = python
            path = plans.reverser.Activate
        }
    }

    output {
        // Set the backward property based on what the active plan returns.
        backward = { getAttr = [ activate, backward_value ] }
    }
}

```

In this example, `type = remote` designates a remote imperative plan and `path = plans.reverser.Activate` is the Python import path to the script's `Activate` class, relative to the `model-definitions` directory. See the [\[Remote Plans\]](#) reference documentation for details of how to declare a remote plan in a service template.

Onboard and Create a Resource

Using bpocore-cli, onboard the whole model-definitions directory:

```
$ bpocore onboard model-definitions/ --auto-products
```

The output will look like:

```
Synchronizing local definition changes

Cloning "ssh://git@172.17.0.2:22/~/repos/model-definitions"
Copying contents from "/home/jd/script-dev-tutorial/model-definitions"
Checking for changes in repository
Committing and pushing changes to "ssh://git@172.17.0.2:22/~/repos/model-
definitions" (branch = "jd/onboarding")

Submitting pull request against "model-definitions" on branch "jd/onboarding"

Waiting for pull request "a8a4a665-ce46-4452-91bf-8495df5910b1" to complete...
Pull request "a8a4a665-ce46-4452-91bf-8495df5910b1" completed successfully, HEAD
moved to "629b98e6c559f19ea7c8551c9a0d9442fa8567e9"!

Checking to see if type artifact changes have propagated

Waiting for type layer to move to "629b98e6c559f19ea7c8551c9a0d9442fa8567e9"...
Waiting for type layer to move to "629b98e6c559f19ea7c8551c9a0d9442fa8567e9"...
Type layer moved to "629b98e6c559f19ea7c8551c9a0d9442fa8567e9", all changes
propagated successfully!

Checking for undeclared products

Product for "example.resourceTypes.Reverser" created successfully! (id =
"597a8fef-1798-40a5-8cf5-ea8104e7d32a")
```

Wait 60 seconds for script-dev to notice the change. Now instantiate the product bpocore-cli created for the resource type:

```
$ bpocore market api -i
market> resources post {productId: "597a8fef-1798-40a5-8cf5-ea8104e7d32a",
properties: {forward: "hello, script-dev"}}
+-----+
| Attribute      | Value
+-----+
| autoClean      | false
| createdAt       | "2017-07-28T01:56:35.794Z"
| description     | null
| desiredOrchState | "active"
| differences     | []
| discovered      | false
| id              | "597a99d3-1fea-4870-b804-b81e6e86b79e"
| label            | null
| nativeState      | null
| orchState        | "requested"
| orderId          | null
| productId        | "597a8fef-1798-40a5-8cf5-ea8104e7d32a"
| properties.forward | "hello, script-dev"
| providerResourceId | null
| reason           | ""
| resourceTypeId   | "example.resourceTypes.Reverser"
| shared            | false
| tenantId         | "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738"
| updatedAt         | "2017-07-28T01:56:35.794Z"
+-----+
```

It should quickly achieve an [orchState](#) of "active". You can monitor this by typing `resources get` at the `market>` prompt.

```
market> resources get
+-----+-----+-----+-----+
| label | id               | orchState | desiredOrchState |
| discovered | resourceTypeId | providerResourceId |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| null   | "597a99d3-1fea-4870-b804-b81e6e86b79e" | "active" | "active"      |
| false  | "example.resourceTypes.Reverser" | null      |             |
+-----+-----+-----+-----+
```

Once the resource has gone active, get it to see its properties:

TIP

You can tab-complete the resource ID: given the above output that would mean typing `resources 59` and then the Tab key.

```
market> resources 597a99d3-1fea-4870-b804-b81e6e86b79e get
+-----+-----+
| Attribute | Value |
+-----+-----+
| autoClean | false |
| createdAt | "2017-07-28T01:56:35.794Z" |
| description | null |
| desiredOrchState | "active" |
| differences | [] |
| discovered | false |
| id | "597a99d3-1fea-4870-b804-b81e6e86b79e" |
| label | null |
| nativeState | null |
| orchState | "active" |
| orderId | null |
| productId | "597a8fef-1798-40a5-8cf5-ea8104e7d32a" |
| properties.backward | "ved-tpircs ,olleh" |
| properties.forward | "hello, script-dev" |
| providerResourceId | null |
| reason | "" |
| resourceTypeId | "example.resourceTypes.Reverser" |
| shared | false |
| tenantId | "7c7da24e-d2af-35c7-8f6d-d8d16c7f0738" |
| updatedAt | "2017-07-28T01:56:36.138Z" |
+-----+-----+
```

Success! The activate script was executed by script-dev and reversed the `forward` property's value. The service template used its return value to populate the `backward` property of the resource instance.

Debugging Scripts

Beginning with script-dev 2.1.4, you can interactively debug scripts running in the script-dev environment. Simply add the line `import pdb; pdb.set_trace()` to your script. When the script is run the `pdb` debug prompt will appear among script-dev's log output. For example, you would see something like this in the terminal you ran script-dev in if you added that line just above the `return` statement in the activate script:

```
22:39:51.204 INFO  script-dev Starting script-dev at 172.16.1.2
fe80::42:acff:fe10:102%eth0
[...more startup messages...]
22:40:18.709 INFO  syslog GET> /bpocore/market/api/v1/resources/598ce0d2-6aab-48ac-
a678-a4a6599ae0a3
[...script log messages ellipsized...]
> /bp2/data/contexts/9d84ebf9dad134a186f4908efa981a741b5d69f6/model-
definitions/plans/reverser.py(15)run()
-> return {"backward_value": backward}
(Pdb) pp backward
u'ved-tpircs ,olleh'
(Pdb) continue
```

Here the `pp` command is used to pretty-print the value of a variable, then the `continue` continues

execution normally. `pdb` allows you to step through the script line-by-line if you wish. Python Module of the Week has a [helpful tutorial on using `pdb`](#).

Be sure to remove any references to `pdb` when done debugging. A call to `pdb.set_trace()` will cause your script to fail when run under the production scriptplan app.

Multi-Tenant Tutorial: Managing Customers as Separate Tenants

Overview

A single deployment of Blue Planet Orchestrate can be used to manage the services of multiple customers while keeping their domains and resources isolated. Domains, products, and resources have **tenant** owners and are scoped such that they are visible to users in the tenant owner, parent tenant, and ancestor tenants.

In Blue Planet, there is always a single **master** tenant. Sub-tenants always have exactly one **parent** tenant. A tenant hierarchy is a tree with the **master** tenant as the root.

One multi-tenant example is the case where the top-level service provider (SP) is providing services to multiple enterprise or wholesaler customers. The SP controls the **master** tenant and creates a sub-tenant for each customer. Each customer is assigned domains owned by their sub-tenant which represent the isolated sandbox for their products, services, and resources. Role-based access control (RBAC) restrictions can be defined to limit the operations that a user in the sub-tenant can perform.

When a new customer is registered with the system, their tenant and domains need to be configured in Blue Planet Orchestrate. These configuration tests may be defined and automated as a **meta-service** executed by an administrator in the SP master tenant. The purpose of this tutorial is to walkthrough an example meta-service that would be executed by a user in the **master** tenant to perform the necessary setup and configuration to activate a new customer. The activation of a new customer involves:

1. Creating a tenant for the new customer in the Tron User Access Control (UAC) application
2. Creating a built-in Planet Orchestrate domain as an isolated domain for the new customer
3. Populating the customer domain with a specified set of products available to the new customer

The termination of a customer results in:

1. Terminating all resources in the customer's domain
2. Deleting the customer's domain
3. Deactivating the customer tenant in the UAC system

This tutorial shows how to implement an example `Customer` resource type implemented by a Service Template with imperative plans that perform the activation and termination lifecycle events for the `Customer`.

After completing this tutorial, you will know:

- How an SP user in the master tenant can use a service to perform the standard initialization steps to activate a new customer
- How to model a new customer as a resource type using TOSCA

- How to implement imperative plans to perform the necessary activation and terminating actions

Note: Imperative plans may be of type `script` or `remote`. Both of these types are Python scripts; however, the difference is that the `script` type is executed directly inside the `bpocontainer` while the `remote` script is executed inside the `scriptplan` container and provides greater flexibility. This tutorial uses the `remote` script type plans.

Prerequisites

If you are not familiar with defining Resource Types and Service Templates, review the [Service Template Tutorial](#) before starting this tutorial.

Please refer to the following links for more reference information:

- [Resource Types](#)
- [Template Engine](#)
- [Service Templates](#)
- [Imperative Plans](#)

Steps

Solution Setup

1. Follow installation steps to have a Blue Planet **Orchestrate** solution running (either a full solution or a **lite** version of the solution). Note that this tutorial cannot be run using `bpocontainer-dev` as it requires interaction with the Tron UAC application and the `scriptplan` application.
2. Install the UI solution `solution-orchestrate_ui:yy.mm.xx-y`
3. Install the NFVO usecase solution `solution-nfvo:yy.mm.xx-y`
4. Install the scriptpip solution `solution-scriptpip:yy.mm.xx-y`

Environment Setup

1. Set up the following shell variable per the example below. Replace the IP address value based on your environment.

```
ORCHESTRATE_URL="https://192.168.33.10"
```

2. Set up some aliases in the environment to simplify processing JSON data at the command line.

```
function jq { python -c "import sys,json; print json.load(sys.stdin)$1"; }
alias jpp='python -m json.tool'
```

bpocore CLI Setup

1. We will be using the `bpocore` CLI tool during this tutorial to simplify REST API and git interface interactions with BPO.

To set up the `bpocore` CLI, run the bootstrap steps in [bpocore CLI Bootstrap Tutorial](#). Ignore the step to start `bpocore-dev` since the `orchestrate` solution is being used. Instead of the instruction to run `bpocore config`, configure the site yaml file based on your orchestrate solution installation.

```
cat <<EOF > ~/.blueplanet/sites/default.yaml
base_url: ${ORCHESTRATE_URL}
auth_scheme: token
auth_params:
  username: admin
  password: adminpw
  tenant: master
verify_certificate: False
EOF
```

2. Confirm the `bpocore-clc` is configured properly and working against the site by running a `bpocore ping` command and confirming the site is up.

```
bpocore ping
```

Clone and Prepare the model-definitions Asset Area

1. Provide your ssh key information to `bpocore` in order to be able to perform git operations against the model-definitions asset area.

```
bpocore assets push-identity
bpocore assets pull-identity
```

2. Obtain the model-definitions asset area URL

```
ASSETS_URL=$(bpocore assets api --format=json areas model-definitions get | jq
'["uri"]')
```

3. Clone the model-definitions git repository into a clean directory and create a branch named `multi-tenant-tutorial` based on the `production` branch.

```
git clone ${ASSETS_URL} ./model-definitions
pushd ./model-definitions
git checkout -b multi-tenant-tutorial origin/production
popd
```

4. Create a working directory to contain the definitions to onboard.

```
mkdir -p ./model-definitions/types/tosca/tutorials
mkdir -p ./model-definitions/scripts/tutorials
```

Create Customer Resource Type

Define a Resource Type which contains the information necessary to create and activate a new customer in the Blue Planet Orchestration system. You can imagine an initialization process which is quite complex, but for this example the activation of a customer focuses only on the tasks described in the [Overview](#). The following input configuration properties are needed to create a new **Customer** resource:

1. tenantName - Name to assign the customer tenant in the UAC system
2. domainName - Name to assign the customer's Planet Orchestrate domain
3. adminRoleName - The role to assign the admin user in the new tenant
4. products - List of products from this domain to offer in the newly created customer's domain

The output properties are:

1. tenantId - The UUID assigned to the tenant when the customer is created
2. domainId - The UUID assigned to the customer's built-in domain when the customer is created

The resource type is implemented by a service template that associates the imperative plans to be invoked for activation and termination lifecycle operations. These plans are written in Python and executed as remote scripts in the `scriptplan` container.

When a new tenant is created, an `admin` user is automatically created; however, the user is not assigned any Blue Planet roles. During activation, the `admin` user will be assigned a role based on the `adminRoleName` which defaults to the `Application admin` role and has unrestricted access to BPO operations within the domain owned by the tenant.

Create the TOSCA definition file containing the resource type and service template definition by copying and pasting the text below.

NOTE

Copy the text below and paste directly to the OS command line prompt directly. Do not paste to an editor.

```
cat <<EOF > ./model-definitions/types/tosca/tutorials/resource_type_customer.tosca
"\$schema"      = "http://cyaninc.com/json-schemas/tosca-lite-v1/definition-module#"
title          = "Customer Configuration"
package         = tutorials
version         = "1.0"
description    = "This TOSCA file defines the configuration for creating a new
customer"
authors        = [ "bpocore-docs@ciena.com" ]
```

```
resourceTypes {  
  
    customer {  
        derivedFrom = tosca.resourceTypes.Root  
        title = Customer  
        description = """  
            Configuration for a new customer  
        """  
        properties {  
  
            tenantName {  
                title = "Tenant Name"  
                description = "Customer's tenant name in the UAC system"  
                type = string  
            }  
  
            domainName {  
                title = "Domain Name"  
                description = "Customer's Planet Orchestrate domain name"  
                type = string  
            }  
  
            adminRoleName {  
                title = "Admin Role Name"  
                description = "Role to assign to the admin user"  
                type = string  
                default = "Application admin"  
            }  
  
            products {  
                title = "Customer Products"  
                description = "List of Products from this domain to offer in the customer  
domain (by title)"  
                type = array  
                items.type = string  
            }  
  
            tenantId {  
                title = "Tenant UUID"  
                description = "Tenant UUID in Tron UAC system"  
                type = string  
                config = false  
                output = true  
                updatable = false  
            }  
  
            domainId {  
                title = "Domain UUID"  
                description = "Domain UUID in Market"  
                type = string  
                config = false  
                output = true  
                updatable = false  
            }  
        }  
    }  
}
```

```

serviceTemplates {
    customer {
        title = "Customer"
        description = """
            Implements the lifecycle for a customer
        """
        implements = tutorials.resourceTypes.customer

        plans {
            activate {
                type = remote
                language = python
                path = "scripts.tutorials.customer.Activate"
            }

            terminate {
                type = remote
                language = python
                path = "scripts.tutorials.customer.Terminate"
            }
        }

        output {
            domainId = { getAttr = [ activate, domainId ] }
            tenantId = { getAttr = [ activate, tenantId ] }
        }
    }
}

cat <<EOF > ./model-definitions/scripts/tutorials/planhelper.py
[PASTE BUFFER HERE]
EOF

```

Create Activation and Termination Plans

Two Python files will be used to execute the necessary plan logic. These files are onboarded into the `model-definitions` asset area. To organize these files separate from the TOSCA definitions, these Python scripts will be placed in the `scripts` sub-directory.

- [planhelper.py](#)
- [customer.py](#)

Copy the `planhelper.py` file to your buffer and paste:

```

cat <<EOF > ./model-definitions/scripts/tutorials/planhelper.py
[PASTE BUFFER HERE]
EOF

```

Copy the `customer.py` file to your buffer and paste:

```
cat <<EOF > ./model-definitions/scripts/tutorials/customer.py  
[PASTE BUFFER HERE]  
EOF
```

Create empty `init.py` files in order for Python to consider these packages:

```
touch ./model-definitions/scripts/tutorials/__init__.py  
touch ./model-definitions/scripts/__init__.py
```

All plans defined as `type = remote` are executed in the `scriptplan` container. The plans are expected to inherit from `Plan` and need to minimally define requirements for packages which are part of the `plansdk`. The `scriptplan` application will build a Python virtual environment based on the `requirements.txt` file defined at the base of the `model-definitions` area. Minimally, the plans depend on these packages which are provided by the `scriptpip` solution deployed earlier.

```
cat <<EOF > ./model-definitions/requirements.txt  
plansdk==1.5.1  
twigjack==4.1.2  
EOF
```

Onboard the Model Definitions

The files are ready to be onboarded into the `model-definitions` asset area.

```
bpoctl onboard model-definitions
```

Create the Product Used to Offer a New Customer

The SP will have a **Product** that is used to create a new customer. A product is always defined in the domain that will contain the resource. In this case, the `Customer` product will exist in the top-level built-in domain owned by the `master` tenant. When a `Customer` resource is created from this product, the resource will be in the SP's built-in domain (also known as the Planet Orchestrate domain).

```
cat <<EOF > ./create_customer_product.json
{
  "resourceTypeId": "tutorials.resourceTypes.customer",
  "title": "New Customer",
  "description": "Creation and configuration for a new customer",
  "active": true,
  "domainId": "built-in",
  "constraints": {},
  "providerData": {
    "template": "tutorials.serviceTemplates.customer"
  }
}
EOF
```

```
PRODUCT_ID=$(bpocore market api --format=json products post
@./create_customer_product.json | jq '[ "id" ]')
```

Review the product details by performing a GET on the product ID

```
bpocore market api --format=json products ${PRODUCT_ID} get
```

Create the New Customer Resource

The model definition and product for the New Customer is now defined and ready to be used. A new customer is created by creating a new resource based on the "New Customer" product created in the previous step.

For this example, define the new customer to be a customer named "Acme".

```
CUSTOMER_NAME="Acme"
```

When creating a customer domain, a product catalog is defined which controls the products that will be available to the new customer. For example purposes, only a single product named **Noop** is being offered. A copy of the product named "Noop" in the parent domain is copied to the customer's domain.

```
cat <<EOF > ./create_customer_resource.json
{
  "label": "${CUSTOMER_NAME}",
  "description": "Configuring sandbox for customer ${CUSTOMER_NAME}",
  "productId": "${PRODUCT_ID}",
  "properties": {
    "domainName": "${CUSTOMER_NAME}_Domain",
    "tenantName": "${CUSTOMER_NAME}_Tenant",
    "products": [ "Noop" ]
  }
}
EOF
```

```
RESOURCE_ID=$(bpocore market api --format=json resources post
@./create_customer_resource.json | jq '[ "id" ]')
bpocore market api resources ${RESOURCE_ID} get
```

Inspect the Results

Confirm the following in the Orchestration UI:

1. The **Planet Orchestrate** built-in domain owned by the master tenant has a new customer resource with label "Acme".
2. A new domain named **Acme_Domain** has been created and it is owned by the tenant **Acme_Tenant**.

Confirm the following in the UAC UI:

1. The new tenant **Acme_Tenant** has been created.

Create a Resource in the Customer Domain

The customer domain is offering the **Noop** product.

Get the **id** of the customer's domain from the **output** property populated when the service was successfully activated. An **output** property is a property that isn't specified as part of the creation request, but is set by the provider of the resource during initial activation. An output property does not change after activation.

```
CUSTOMER_DOMAIN_ID=$(bpocore market api --format=json resources ${RESOURCE_ID} get
| jq '[ "properties" ][ "domainId" ]')
```

Get the **id** of the **Noop** product created in the customer's domain.

```
NOOP_PRODUCT_ID=$(bpocore market api --format=json domains ${CUSTOMER_DOMAIN_ID}
products get q='title:Noop' | jq '[ "items" ][ 0 ][ "id" ]')
```

Create the body of a request to create a **Noop** resource instance.

```
cat <<EOF > ./create_noop_customer_resource.json
{
  "label": "${CUSTOMER_NAME} Test Resource",
  "description": "Noop resource created by customer ${CUSTOMER_NAME} in the
customer domain",
  "productId": "${NOOP_PRODUCT_ID}",
  "properties": {
    "data": {}
  }
}
EOF
```

Create a `Noop` resource instance in the customer domain.

```
NOOP_RESOURCE_ID=$(bpocore market api --format=json resources post  
@./create_noop_customer_resource.json | jq '[ "id" ]')  
bpocore market api resources ${NOOP_RESOURCE_ID} get
```

Explore the System

Explore as the Acme Tenant Admin User

Logout of the Orchestrator UI and re-login as the `admin` user (password `adminpw`) in the `Acme_Tenant` tenant. As that user, only the `Acme_Domain` is visible and only the `Noop` resource created in the customer's domain is visible.

Explore as the Master Tenant User

Logout and re-login as the `admin` user in the `master` tenant. In this context, you have visibility to the master's **Planet Orchestrate** built-in domain which includes the **Customer** resource. The master tenant also has visibility to sub-tenant domains including the `Acme_Domain` and the `Noop` resource in the domain.

Delete the Customer

The service definition included plans for activating a new customer and terminating the customer. Cleanup the resources, products, domain, and deactivate the tenant by deleting the customer resource in the `master` domain.

```
bpocore market api resources ${RESOURCE_ID} delete
```

Confirm the customer resource is deleted.

```
bpocore market api resources ${RESOURCE_ID} get
```

The customer domain and all of its resources are gone.

```
bpocore market api domains ${CUSTOMER_DOMAIN_ID} get
```

The tenant exists in the UAC system, but it is now marked as inactive and it is no longer possible to log in as a user in the customer tenant.

Conclusion

Blue Planet Orchestrate is a multi-domain and multi-tenant orchestrator. This tutorial has demonstrated how the tenancy and domain building blocks can be used to manage resources across customers and provide tenant isolation. In this tutorial, you

1. Created a product that defines a meta-service that can be run by a user in the SP administrative master tenant to activate a new customer.
2. Created an example resource for the customer that is contained in the customer's domain and owned by the customer's tenant.

Blue Planet Orchestrate REST APIs

Overview

The **REST APIs** section contains two subdirectories:

- [API General Content Sections](#) - General content that is not specific to a single component
- [REST Component Overview](#) - Per-component directories with content specific to a single component

API General Content Sections

- [Getting Started](#)
- [Authentication](#)
- [Filtering](#)
- [Sorting](#)
- [Pagination](#)
- [SSL Certificates](#)

Getting Started

Welcome to the documentation for Blue Planet Orchestrate's RESTful APIs.

Blue Planet Orchestrate is Ciena's multi-function orchestration platform. It coordinates and automates the creation, management, and synchronization of a wide range of digital resources, including networking (carrier networks, data center networks), storage, compute, applications, and services. The generic resource model allows effective and unified orchestration of physical, virtual, and logical resources. Blue Planet Orchestrate supports the following:

- Multi-domain service orchestration
- NFV (network function virtualization) orchestration
- Network orchestration
- Cloud orchestration

Blue Planet Orchestrate is a multi-component system. Each component exposes its own set of APIs. The common URL format is:

```
`<protocol>://<host>:<port>/bpocore/<component>/api/v1/<resource>`
```

For example,

```
`https://blueplanet:443/bpocore/market/api/v1/resource-types`
```

The API documentation for a given component may be accessed as a Swagger 2.0 Specification document by either replacing **api** with **api-doc** in the URL or requesting the **swagger.json** resource. For example to obtain the Swagger Spec for the **market** component:

```
`https://blueplanet:443/bpocore/market/api-doc/v1`  
`https://blueplanet:443/bpocore/market/api/v1/swagger.json`
```

The Swagger UI may be used to browse and try out the APIs. For example,

```
`https://blueplanet:443/bpocore/swagger`
```

Authentication

There are three methods to handle authentication of REST API requests from clients.

1. HMAC Signed Requests - This is the authentication method recommended for use in production
2. Token Authentication - User-based authentication using a short-lived token without message signing
3. No Authentication - (Development Only) Running the Planet Orchestrate application without performing API authentication

HMAC Signed Requests

For secure operation in production, REST clients of Planet Orchestrate should authenticate by including a Hashed Message Authentication Code (HMAC) signature in the **Authorization** header of each HTTP request. A key pair (key identifier and key secret) must be obtained from the Blue Planet User Access Control system (UAC). The key secret is used to compute the MAC signature. The header value is a string that starts with the "MAC" and then contains the key id, timestamp, nonce, and computed MAC.

VALUE	IDENTIFIER	DESCRIPTION	EXAMPLE
Key Identifier	id	String identifying the key pair	id="ae71d7d92d7d4c659a7d3336db6c4c99"
Timestamp	ts	Number of elapsed seconds from UNIX Epoch	ts="1411065195"
Nonce	nonce	A one-time use string (preferably a URL-safe Base64 Encoding per RFC 4648)	nonce="Jw1ctgzz2X2n+6DDOBI Eig=="
MAC	mac	Signature	mac = "oYhbGKDhOZZ9ReHQyZS0jML wOSQDGplmWbtY3d+dORM="

The following table shows an example of a full Authorization header:

HEADER	VALUE
Authorization	MAC id=ae71d7d92d7d4c659a7d3336db6c4c99,ts=1400863370,nonce=Jw1ctgzz2X2n+6DD OBI Eig==,mac=oYhbGKDhOZZ9ReHQyZS0jMLwOSQDGplmWbtY3d+dORM=

The signing algorithm constructs a canonical signature string from the timestamp, nonce, HTTP method (upper case), URI path, host, and port based on a [draft OAUTH v2 MAC](<https://tools.ietf.org/html/draft-ietf-oauth-v2-http-mac-02>). The MAC is based on the header information only without attention to the payload; therefore, the MAC is sufficient to authenticate that the sender of the HTTP request knows the keySecret, but it does not guarantee the integrity of the payload or ensure privacy. For this reason, secure HTTPS transport is recommended between the client and server.

An explanation of the algorithm with a reference implementation in Python is available at:
github.com/ciena-blueplanet/rsign

Token Authentication

Exchange user login credentials (user name, password, and tenant name) for a short-lived token (24 hour default). The token is included in the `Authorization` HTTP header and grants the caller access appropriate for the corresponding user roles. The header value is in the format `token <token_value>`.

Note for backward compatibility, the `X-Auth-Token` header may be used to carry the token value in the request header value field; however, this format is deprecated and may be removed in a future release.

Obtain the token either by providing user credentials to the UAC (Tron) application directly (**POST** to `/tron/api/v1/tokens` - the token is returned in the body) or through the Planet Orchestrate **authentication** component (**POST** `/bpocore/authentication/api/v1/tokens` - the token is returned in the `X-Subject-Token` HTTP response header).

The **Swagger Browser UI** may be used to execute the API call to the **authentication** component.

No Authentication

Run the **Planet Orchestrate** application with the `--no-auth` command line option to entirely disable authentication of API calls (**not suitable for production**)

Example

Python

As a first example, consider the requirement to:

- Determine the resource types known by the system
- Obtain the list of active resource instances of each type

The **Market** component is the primary component in the system for managing the orchestration domains and instances within the domains.

The following example Python code interacts with the Market to make API calls to get the required information as **application/json**. The example uses:

- `http://`
- `urllib`
- `json`

The example is using the **No Authentication** mode.

```
import json
import httplib
import urllib

connection = httplib.HTTPConnection ("localhost:8181")

headers = {"Content-type": "application/json",
           "Accept": "application/json"}

connection.request ('GET',
'http://localhost:8181/bpocore/market/api/v1/resource-types', None, headers)

result = connection.getresponse ()

data = result.read ()

resourceTypeList = json.loads(data).get("items")

print "Resource Type List: " + str(len(resourceTypeList)) + " types"
print "====="

for resourceType in resourceTypeList:
    resourceName = resourceType.get("title")
    resourceId = resourceType.get("id")
    print resourceName + " ==> " + resourceId
    params = urllib.urlencode({})
    connection.request('GET',
'http://localhost:8181/bpocore/market/api/v1/resources?resourceTypeId=' +
resourceId, None, headers)
    rtResult = connection.getresponse()

    if (rtResult.status == 200):
        rtData = rtResult.read()
        resourceList = json.loads(rtData).get("items")
        print resourceName + " count of " + str(len(resourceList))
    else:
        print "Error: result.status = " + str(rtResult.status)
```

Curl with Token Authentication

Consider the example of using **curl** to obtain the list of **products** active in the system using **token-based** authentication. The example below uses the **admin** user with a password **adminpw** in the default **master** tenant at a host named **orchtest1**. Replace these values with the appropriate values from your environment.

1. Exchange username, password, and tenant name credentials for a token by POSTing to the `/authentication/api/v1/tokens` endpoint. The token is returned in the `X-Subject-Token` header field.

```
curl -v -k -H "Content-Type:application/json" -d
'{"name":"admin","password":"adminpw", "domain": {"name": "master"} }' -X POST
https://orchtest1/bpocore/authentication/api/v1/tokens
HTTP/1.1 201 Created
Server: nginx/1.4.2
Date: Wed, 26 Nov 2014 22:00:29 GMT
Content-Type: application/json; charset=UTF-8
Content-Length: 139
Connection: keep-alive
X-Subject-Token: 2aadfd6f1d35849a459d
x-tracing-id: a053572b-b86a-47e6-a84d-58b2251bab22
```

2. Set the `Authorization` header field to the token value obtained in the previous step. To obtain the list of products, use the `/market/api/v1/products` endpoint.

```
curl -v -k -H "Content-Type:application/json" -H "Authorization:token
2aadfd6f1d35849a459d" -X GET https://orchtest1/bpocore/market/api/v1/products

HTTP/1.1 200 OK
Server: nginx/1.4.2
Date: Wed, 26 Nov 2014 22:06:51 GMT
Content-Type: application/json; charset=UTF-8
Content-Length: 7864
Connection: keep-alive
x-tracing-id: d129bbbc-70bb-4df4-bf7c-d75aa0ba0862

{
  "items": [
    {
      "active": true,
      "id": "7344b1d4-900f-4da0-a4a5-319a77250a67",
      "provider": "built-in",
      "providerData": {
        "template": "tosca.serviceTemplates.Vcpe"
      },
      "resourceType": "tosca.resourceTypes.Vcpe",
      "title": "vCPE"
    },
    {
      "active": true,
      "id": "c2737e5a-8d16-41c9-8824-d305f831c3db",
      "provider": "built-in",
      "providerData": {
        "template": "tosca.serviceTemplates.VcpeEms"
      },
      "resourceType": "tosca.resourceTypes.VcpeEms",
      "title": "vCPE EMS"
    }
  ]
}
```

Filtering

It is common behavior across all components and resources for a **GET** request returning a list response to support an optional query filter to restrict the entities returned in the result list based on the value of one or more entity attributes.

There are 3 types of query filters supported:

- Exact match query filter
- Partial string match filter
- Tag query filter

Exact Match Query Filter

For example, the following query will return all resources with a label "vCPE7001", that were not discovered, and have an **active** orchestration state:

```
https://orchtest1/bpocore/market/api/v1/resources?q=(label:vCPE7001,discovered:false,orchState:active)
```

Syntax

The filter query syntax is:

- Filters are query parameters specified by the parameter name **q**.
- Each filter is a key, value string pair separated by a ":" specifying an equality match.
- An AND condition may be specified using multiple comma separated filters in a single query (as in the example above)
- Parenthesis are optional around the entire value e.g.:

```
?q=(orchState:active,discovered:false) or ?q=orchState:active,discovered:false
```

- Filtering is supported for properties of type: String, Boolean, Number, and null. Complex data types (i.e., array and object) are not supported.
- Filtering is supported based on the top-level attributes and first level of onboarded properties:
 - First level properties must also be filtered based on exactly one of Resource Type ID, Product ID, or ExactType ID. for e.g:

```
?resourceTypeId=tosca.resourceTypes.VirtualMachine&q=properties.diskSize:4
```

- A query can have multiple first level properties. The returned result is an AND of resources having both properties. for e.g:

```
GET  
/resources?resourceTypeId=tosca.resourceTypes.VirtualMachine&q=properties.numCpus:8,properties.diskSize:4
```

- A query can have both top-level and first-level Resource property. The returned result is an AND of resources having both properties. for e.g:

```
GET  
/resources?resourceTypeId=tosca.resourceTypes.VirtualMachine&q=properties.numCpus:8,desiredOrchState:active
```

- Examples:

- gets all resources of type Instance (and derived) w/ numCpus = 8

```
GET  
/resources?resourceTypeId=aws.resourceTypes.Instance&q=properties.numCpus:8
```

- gets all resources of product someDummyProduct

```
GET /resources?productId=someDummyProduct&q=properties.numCpus:8
```

- gets all resources of type Server (which likely has children), similar to first example

```
GET  
/resources?resourceTypeId=tosca.resourceTypes.Server&q=properties.numCpus:8
```

- gets all resources of type VM (but no children)

```
GET  
/resources?exactTypeId=openstack.resourceTypes.VirtualMachine&q=properties.numCpus:8
```

- query with no resourceTypeld/ProductId/ExactTypeld will fail without a resource type context

```
GET /resources?q=properties.numCpus:8 // 400 failure
```

- Rule followed for multiple "q" filters: ((Top Level attribute AND Top Level attribute) AND First level Property)

Caveats

1. Multiple filter queries are NOT supported (i.e., only the first "q" parameter will be read)

2. The ":" is interpreted as an equality comparator and it is the only comparator supported (i.e., no support for <, >, !=, etc.)
3. The filter query is only supported for:
 - Top level attributes and the first level of onboarded properties for simple types
 - Attributes where the backend database property name matches the value presented at the user level

Partial String Match Query Filter

Syntax

```
https://orchtest/bpocore/market/api/v1/products?p=title:m1,description:VMWare
```

- Filters are query parameters specified by the parameter name **p**
- Each filter is a key, value string pair separated by a ":"
- Multiple comma separated filters may be separated in a single query with the interpretation as the OR of the filters (e.g., above query contains two filters and the response will be a union of products having title containing m1 or a description containing VMWare)
- Multiple filter queries are NOT supported (i.e., only the first "p" parameter will be read)
- Partial match query filter (p) can be used in conjunction with query parameter (q). The returned result will be an AND of both queries. For example, to query the resources having a label starting with "US" AND with the "active" attribute true

```
?p=label:US&q=active:true
```

- Parenthesis are optional around the entire value, for example:

```
?p=(label:US,description:English) or ?p=label:US,description:English
```

- A query attribute can take only one word value.
- A query attribute with empty value is not supported.
- The query is only supported for certain top-level attributes and first level of onboarded properties where the schema for the property marks it as **fulltext** searchable and the backend property name matches the name presented in the user level model.
- The query is only supported for properties of data type String
- First level properties must also be filtered based on exactly one of Resource Type ID, Product ID, or ExactType ID. for example:

```
?resourceTypeId=tosca.resourceTypes.PropertiesSample&p=properties.myFullTextSearchableString:hel*o
```

- A query can have multiple first level properties. The returned result is an AND of resources having both properties, for example:

```
GET  
/resources?resourceTypeId=tosca.resourceTypes.PropertiesSample&p=properties.myFullTextSearchableString:he*, properties.myFullTextSearchableString:tes?Str*
```

- A query can have both top-level resource attribute and first-level Resource property. The returned result is an AND of resources having both properties, for example:

```
GET  
/resources?resourceTypeId=tosca.resourceTypes.PropertiesSample&p=properties.myFullTextSearchableString:hello, label:US
```

- A query can contain both "p" and "q" filters. The result is an AND of resources having both properties, for example:

```
GET  
/resources?resourceTypeId=tosca.resourceTypes.PropertiesSample&q=properties.myBoolean:true, label:Dummy&p=properties.myFullTextSearchableString:test*
```

```
GET  
/resources?resourceTypeId=tosca.resourceTypes.PropertiesSample&q=properties.myBoolean:true&p=properties.myFullTextSearchableString:test*
```

```
GET  
/resources?resourceTypeId=tosca.resourceTypes.PropertiesSample&q=label:Dummy&p=properties.myFullTextSearchableString:test*
```

- Rule followed for multiple "p" filters: ((Top Level attribute OR Top Level attribute) AND First level Property)

Note: Partial String Match Search is supported only on products and resources APIs.

- searchable properties on resources include [label, description, reason, first-level onboarded string properties that specify **fulltext** searchability in their schema]
- searchable properties on products include [title, description]

String Matching Syntax

- It supports single and multiple character wildcard searches.
 - To perform a single character wildcard search use the ? symbol. The single character wildcard search looks for terms that match that with the single character replaced. For example, to search for "text" or "test" you can use the search: te?t.
 - To perform a multiple character wildcard search use the * symbol. Multiple character wildcard searches looks for 0 or more characters. For example, to search for test, tests or tester, you can use the search: test or t*t*.
- It supports fuzzy searches. To do a fuzzy search use the tilde, ~, symbol at the end of a Single word Term. For example to search for a term similar in spelling to "roam" use the fuzzy search: roam~
- The search is powered by [Apache Lucene](#) and exposes a subset of the Lucene query parser syntax. The fulltext searchable string is broken into small indexing elements called tokens, based on [standard tokenizer](#). Refer to the Lucene documentation to try more advanced searches. Special characters must be escaped using the backslash '\'. Special characters include:

```
+ - && || ! ( ) { } [ ] ^ " ~ * ? : \ /
```

Note: You cannot use a * or ? symbol as the first character of a search. The first character in a partial string match must be specified.

Tag Query

Resources may be assigned **tags** where tags are {key, value} pairs. Refer to the [Tags](#) section for details on using tags.

Query resources based on tag {key, value} pairs using the **tags** query parameter

Syntax

A Get of resources supports an optional **tags** query parameter. The syntax for this is as shown in the following example:

```
http://localhost:8181/bpocore/market/api/v1/resources?tags=layer:L3,layer:Eth
```

- Querying based on tag {key, value} pairs is done using the **tags** query parameter in the [GET /resources](#) API
- Values are specified as TagKey, TagValue pairs with the key and value separated by a ":"

```
?tags=layer:L3
```

- Multiple {key, value} pairs may be specified in a single query by separating them with a ","
- Multiple {key, value} pairs within a single "tags" parameter are interpreted as an OR condition (i.e., the example will return instances that are either L3 or Eth)
- The ":" is interpreted as an exact match equality comparator
- The key and value are case sensitive
- The query request will return a **404** error if either the tag key or value has not been defined
- Parenthesis are optional around the entire value, for example:

```
?tags=(layer:L3,layer:L2)
```

Error Handling

- Query on attributes not supported by query filters will return a 400 error code.
- Partial match query with un-supported attribute values will return a 400 error code.

Sorting

It is common behavior across all components and resources for a GET request returning a list response to sort the entities returned in the result list based on the value of one or more entity attributes.

Syntax

```
<base-url>/bpocore/market/api/v1/resources?sort=(attribute:title,descending:true)
```

- Sorting on an attribute is specified by the parameter name `sort`.
- Attribute name is specified by key, value string pair separated by a `:`.
- Order of returned result can be specified by `descending` keyword(optional). Default order is ascending.
- Multiple sorting attributes are NOT supported (i.e., only first "attribute" parameter will be read).
- Sorting can be used in conjunction with query parameter "`p`" (partial match query filter) and/or "`q`" (Exact match query filter).
- Sorting is supported only for certain top-level attribute properties where the backend property name matches the value presented at the user level.
- Parenthesis are optional around the entire value e.g., `?sort=(attribute:title,descending:true)` or `?sort=attribute:title,descending:true`

Error Handling

- Sorting on attributes that are not supported by will return a 400 error code.
- Keywords other than "attribute" and "descending" will be ignored.

Pagination

The standard convention, across all components and resources, is for a **GET** request returning a list response (i.e., those that do not specify a **resourceId**) to return a list of results wrapped in a **ListReply[T]**. The protocol supports receiving a specified subset of the results and iterating over **pages** of the result set.

Paged Request

The following optional query parameters are used to issue a paged request:

- **limit** (int) - The maximum number of entities in a single paged response.
- A **limit** of 0 will return just the ListReply envelope giving the total size of the query result, but without any entities.
- If the **limit >= total**, the query result is returned in a single page.
- Defaults to the configuration parameter **page-size-limit** specified for the rest-server component (currently 1000)
- **offset** (int) - The index of the first entity to include in the paged response [0, total)
- Defaults to an offset of 0
- An **offset >= total** returns an empty result
- **pageToken** (string) - When fetching multiple result pages for a given query, the **pageToken** may be used to allow the server to optimize the response by using a cached result. The pageToken also provides for a stable result by locking the database state to the basis specified by the token.
- The pageToken is obtained from the **nextPageToken** value in the previous response.
- The pageToken encodes the **limit** used in the initial query and encodes an **offset** value that will be used if not explicitly overridden when an **offset** query parameter.

The following example request obtains the 3rd page of results based on pages of size 100:

```
GET /bpocore/market/api/v1/type-artifacts?limit=100&offset=200
```

Paged Response

The **ListReply[T]** for a collection of type **T** has the schema:

```
{
  "items": array[T],
  "total": int, // Total size of the query result across all pages
  "limit": int, // The limit specified in the query
  "offset": int, // The index of the first item included in this response (0 by default)
  "nextPageToken": string // For multi-page results, the pageToken to use in a follow up call to get the next page
}
```

Page Token

The simplest use of the page token mechanism is to iterate over the pages sequentially. The initial request need only specify the **limit** parameter. The **nextPageToken** is only included in the response if the all of the results did not fit in a single page. For the next page of results, use the **nextPageToken** from the first response as the **pageToken** query parameter in the second request. The second request need not specify an **offset** or **limit** value as that information is encoded in the token. Repeat using the **k-1** **nextPageToken** value as the **kth pageToken** value until all of the results have been obtained.

The page token mechanism supports request explicitly overriding the **offset** and **limit** values as query parameters to either:

- Obtain arbitrary subsets of results or results out-of-order
- Execute multiple queries concurrently without waiting for the **kth** page before requesting the **k+1** page

The following example obtains pages of requests using the Page Token mechanism.

1st Request:

```
GET /bpocore/market/api/v1/resources?limit=10
```

Response:

```
{
  "items": [
    {
      truncated
    }
  ],
  "limit": 10,
  "nextPageToken": "tokenId1",
  "offset": 0,
  "total": 67
}
```

2nd Request:

```
GET /bpocore/market/api/v1/resources?pageToken=tokenId1
```

Response:

```
{  
  "items": [  
    {  
      truncated  
    }  
  ],  
  "limit": 10,  
  "nextPageToken": "tokenId2",  
  "offset": 10,  
  "total": 67  
}
```

3rd Request:

```
GET /bpocore/market/api/v1/resources?pageToken=tokenId2
```

And so forth while a nextPageToken is returned from the request.

SSL Certificates

REST APIs should be transported using the HTTPS protocol for message encryption and endpoint security. The **HTTP Load Balancer** application (**haproxy**) acts as the SSL termination point for external REST API requests over HTTPS. By default, the server is configured with a self-signed X.509 certificate for **localhost** (i.e., **Common Name** is set to **localhost** in the certificate signing request) in PEM format. For an API client to work with the default configuration, it is necessary to relax the verification of the certificate authority against the client trust store. For example when using **curl**, the **-k** or **--insecure** option must be used. In a deployment, best-practice is to replace the default certificate with a certificate signed by a trusted certificate authority.

Replacing the Default Certificate with a Trusted Certificate

1. Obtain a valid X.509 certificate file signed by a certificate authority (CA) and the private key used in the certificate signing request.
2. (Optional - depending on certificate details) Create a certificate bundle that concatenates the primary certificate with any necessary intermediate certificates and the root certificate provided by the CA. The exact steps depend on the format of the certificate from the granting CA. Refer to information provided by the CA. For example:
 - [digicert Creating a .pem File for SSL Certificate Installations](#)
 - [digicert Nginx SSL Certificate Installation](#)
 - [Comodo Certificate Installation for Nginx](#)
3. Create a server PEM file by concatenating the crt and key files into a **server.pem** file

```
cat <domain>.crt <domain>.key > server.pem
```

4. SSH into the Blue Planet host:
5. Remove the default SSL certificates (if they exist) in **/etc/bp2/haproxy/ssl**
6. Add the **server.pem** file to **/etc/bp2/haproxy/ssl** and **chmod 600**
7. Restart the **haproxy** service

```
solman "solution_app_restart bpdr.io.blueplanet.platform:17.06.08 haproxy"
quit
```

For more information, refer to **Blue Planet Administrators Guide**.

REST Component Overview

Each component of BPO specifies RESTful APIs using the Swagger 2.0 Specification language. The Swagger Spec is generated using automation from the annotated code. This supplemental material provides context and further explanation.

- [Application](#)
- [Asset Manager](#)
- [Authentication](#)
- [Component Registry](#)
- [Diagnostics](#)
- [Events](#)
- [Import/Export](#)
- [Market Component](#)
 - [Tutorial: Allocating Numbers in a Number Pool](#)
- [Policy Manager](#)
- [Resource Provider](#)
- [Rest-Server](#)

Application

The **Application** component exposes management and configuration APIs for the **Planet Orchestrate** application. For example, the **Loggers** resource exposes the ability to adjust application logging levels without restarting the system.

Asset Manager

The **Asset Manager** is responsible for managing file assets used by the system. These include:

- Resource types - TOSCA files defining resource types
- Service templates - TOSCA files defining services composed from resource types
- UI schemas - Definitions for how types are presented in the user interface

The Asset Manager uses **git** and is built on top of a **git** repository.

The Asset Manager exposes APIs to:

- List inventory of the managed files
- Initiate pull-requests to onboard changes
- List history of pull-requests

Authentication

The **Authentication** component is responsible for authenticating users with the **Blue Planet** User Access Control (UAC) system known as **Tron**. The UAC system manages tenants, users, and roles (per-user per application).

The Authentication APIs provide the ability to:

- Verify the application is able to properly connect to the UAC backend
- Obtain and release **tokens** that may be used in authenticated API calls

Component Registry

The **Component Registry** maintains a registry of active system components. When a system component registers, it may specify (tagKey, tagValue) pairs to further identify itself to other components. The **category** tagKey is a special key that is pre-defined and generally specified for all components.

The Component Registry exposes APIs to:

- Register, unregister a component
- List, query, or get registered components
- Manage tags

Diagnostics

The **Diagnostics** component exposes an interface to define, run, and view results of system self-diagnostic tests. Some of these interfaces are used by the Planet Orchestrate daemon to run tests and report back to the Nagios monitoring system.

The Diagnostics component exposes two resources:

- Testcases - Defines a test that is available to be run
- Test-instances - A run or instance of a specific test-case

The general workflow is to obtain a testcase identifier and then create a new test-instance of that testcase.

Events

The **Events** component provides an asynchronous message bus used to publish event information to topics. **Topics** can be thought of as event channels. Multiple types of event messages may be sent to each topic. A topic definition includes the types of event messages that may be sent to the topic. Each event type is defined by a **Model** that can include an arbitrary collection of **properties**.

The Events component API currently exposes APIs for topic management:

- Get or List topics
- Create an event topic

Import/Export

The **Import/Export** component is the component for importing and exporting (usually large amounts of) data in JSON format. Various types of Entities may be imported or exported. These types are:

- Tenants
- TagKeys
- TagValues
- Providers
- Domains
- Products
- Resources
- Relationships

Imports

The Import function is primarily intended for an initial import of data originally created in another system. This data must be transformed to meet the expected well-formed JSON format used by the Import function. The Import function acts as a conduit for inserting this 'bulk' of data in one shot. The Import function does not perform some actions found in using other means of creating similar data, such as service-template activation or the invocation of resource adapter operations. This **bulk** data import is expected to occur when the target BPO system is not in-use as a production system but is in a state that can tolerate the lack of essential steps just mentioned. This use-case is called **brown-field** import. Before the system is brought into a production-ready state, the activities that are needed, but not invoked by the Import function would have to be executed by other means.

Another possible use-case for utilizing the **bulk** Import function is when migrating data from a BPO system to the same BPO system in order to prune history from the database. The database tracks all changes to data over time. It may build up a history which, at some point, needs to be pruned. In this case, a full export would be used to create the latest snap-shot of data in JSON format. That same data could then be imported back into the BPO system after archiving the current database and starting with a new image. As with the previous use-case, the BPO system should not be one that is being used as production system, but one that is in a state to tolerate such changes. If this export-import cycle is intended to span version changes it is important to note that there must be no change to the schema.

The import data is defined as the body of a POST request. By default the maximum size of a POST body allowed by BPO is around 8 megabytes. In some cases the import body can reach 60 to 70 megabytes. At startup, a command line argument can be defined to relax this restriction. In the following example, this restriction is set to 70 megabytes by using a command line argument.

```
-Dspray.can.server.parsing.max-content-length=70m
```

In a production environment it might be better to add this to a configuration file. To do this, enter into the

bpocontainer and add the line:

```
spray.can.server.parsing.max-content-length=70m
```

to the /bp2/conf/deploy.conf file. This would have to be done in each bpocontainer instance in a multi-host system. Once the line has been added to the configuration file, then the solution would have to be restarted.

```
solution_app_restart orchestrate:17.06.1-40 bpocontainer
```

For example:

```
(Cmd) solution_app_restart bpdr.io.blueplanet.orchestrate:17.06.1-40 bpocontainer
```

Production Clusters

The message size between nodes of a cluster is limited by default to 128kb. In order to support Importing in a production cluster, the message size must be increased to allow the entire contents of the body of the import message to be passed from the receiving node to the node that will actually perform the import function. In order to accomplish this, three more settings are required. Keeping with the 70mb example already presented, these settings would be:

```
akka.remote.netty.tcp.maximum-frame-size = "35000000b"  
akka.remote.netty.tcp.send-buffer-size = "7000000b"  
akka.remote.netty.tcp.receive-buffer-size = "70000000b"
```

These entries would be entered into the deploy.conf file in the same manner as the spray.can.server.parsing.max-content-length setting.

When the POST is seen, BPO will attempt to kick off an asynchronous job to process the request. Only one import may be in-progress at any given time, and the POST will be rejected if one is already underway. If the job is accepted, the response to the POST will contain an id value that may be used to query the status of the job.

There are several query parameters that may accompany the POST request. These are introduced here and discussed in sections specific to each option.

- updateIfDifferent a boolean value that defaults to false
- preProcessKeys a boolean value that defaults to true
- publishResults a boolean value that defaults to false

updateIfDifferent

When an import is performed, there may be data already present in the database. In such cases if an import datum matches an entity in the database, but contains field datum that differs from that already in the database, it is ignored. This behavior may be changed by setting the parameter to true ,(i.e. updateIfDifferent=true). When defined and set to true, any import data that matches existing entities in the database, but differ in content, will update the data in the database to reflect the import value.

preProcessKeys

Data imported into BPO must be identified by a key. In BPO this key always a UUID with few exceptions. If the import is a **brown-field** import the source system's ids may not be in the proper format. Such source keys are known as 'foreign' keys. If the preProcessKeys parameter is defined and set to true (default), these **foreign** keys will be subjected to a set of processing steps prior to the actual import. There are five steps as follows:

- Add keys to entities that do not already have them
- Collect all key references into an internal **model**, change any key not currently in UUID format to one that is
- Process well-known references and convert them to UUIDs found in the **model** (e.g. productId, always refers to product)
- Add or process all Resources such that a providerResourceId exists for all
- Process property references and merge the entire collected model back into the JSON for final actual import

These steps are performed sequentially. If any step results in errors, the entire import job is halted and the errors are available in a status request.

publishResults

After an import has successfully completed, if this parameter is defined and set to true, then a select set of entities will be published in their entirety. These select types are:

- Domains
- Products
- Resources
- Relationships

Job id

The id assigned in the Import POST response may be used for several downstream API calls. It may be used to query about the status of a long-running Import job. It may also be used to obtain the contents of the internal **model** that is generated during the pre-process steps. The **model** that is being generated during an Import job is persisted in memory until it is explicitly deleted or BPO is restarted. This is done to

allow for an incremental import (discussed shortly). This **model** takes up memory space, and in the case of a very large **bulk** import, this could be large. The contents of this **model** may be useful to the importing agent for the following reasons.

- Activation assets refer to original (or **foreign**) keys
- On-boarded scripts may refer to **foreign** keys

In such cases, one can only modify the **foreign** key references by knowing what UUID was assigned by the pre-processing steps. By using the assigned job id, one may obtain the contents of the model by using another of the available Import APIs. After the **model** is extracted, and perhaps archived, the same id may be used to remove the internal **model** in order to release the memory it occupies.

Property **foreign** key references

In some cases **foreign** key references will occur in fields that clearly may only refer to a single type of entity. For example; The Resource entity contains a field entitled `productId`. This field may only refer to an entity of type **Product**. The pre-processing steps will attempt to reconcile such resource references by looking into the **model** entry for products. Some entities (e.g. Domains, Resources) have a properties field. This field may contain properties that refer to other entities. In this case, the pre-processing steps cannot resolve the reference a-priori, and additional information is required to give the pre-processing steps a hint. An example is a fictitious property field called "prodRef". This fictitious property field will refer to a product id. The pre-processing steps cannot infer from the name which part of the **model** to use as a source for the foreign key, so it needs help. In this case the field value must contain the string `%lookup-products%` in order to properly reconcile the key reference. The full example might look like the following

```
"prodRef": "%lookup-products% Product123Key"
```

The legal values for entity type in the `%lookup-XXX%` where XXX is replaced with the type are:

- `%lookup-domains%`
- `%lookup-providers%`
- `%lookup-products%`
- `%lookup-resources%`
- `%lookup-relationships%`

There is an optional terminator for this type of lookup. It is to allow for cases where the key itself is not terminated by a quote character. An example of this might be in a URL such as the following example.

```
"prodRef": "some url http://somepath/%lookup-
products%Product123Key%%/someotherPathElement"
```

Notice in this example the same 'Product123Key' is followed by `/someotherPathElement` which would prevent the key from being matched unless there is somehow to tell the lookup process that the end of the key is here. This is what the `%%` indicates. It is optional and, as in the first example, if the key is

immediately followed by a quote there is no need to add %% (although you could if you wish).

Incremental imports

An incremental import is one that is performed in a series of steps, or increments. The reason for approaching the import as a set of increments may be to reduce the size of any one import POST body. Care must be taken when attempting an incremental import to assure that references to **foreign** keys in any one increment may be resolved by the pre-process steps. As such any reference to a **foreign** key must refer to one that is defined in the same file as the reference, or refers to an entity that already exists in the database. To be clear, if a **foreign** key reference exists in an import that does not meet these criteria, the import will fail.

An example of a failure case would be an incremental import consisting of Domains. The Domain entity allows for properties. As covered in the section entities 'Property **foreign** key references' there may be a property field that refers to a product. Since the product entities have not been loaded, the import will fail.

There is an order that must be maintained when performing an incremental import in order to stand the best chance of avoiding a **forward** reference (a key reference that cannot be resolved because it does not exist yet). If an incremental import is to be attempted, it is best to break the entities along the following lines, and assure that they are loaded in this order:

- Domains
- TagKeys
- TagValues
- Providers
- Products
- Resources
- Relationships

Of course it is possible to group several smaller increments into one larger one, but they will still adhere to the order just mentioned. It is most likely that **Resources** will be the largest body. So: Domains, TagKeys, TagValues, Providers and Products could be combined into one import request (they can be in any order with the same POST body, the importer is smart enough to load them in the proper order). Resources and Relationships could incrementally imported separately and in sequence.

Exports

Export is the mirror image of import. The Export was motivated by the use-case whereby the Database history is to be pruned (as previously described). An Export POST request will attempt to kick-off an asynchronous job that will export all, or selected, entities in a well-defined JSON format. This JSON format exactly matches that expected by Import. It is possible to Export the database and use the output of the Export as source for an Import, as-is (as described previously for Database history pruning).

As with Import there may only be one Export in-progress at any given time. The POST response will contain an id which may be used for several downstream API calls. The id may be used to request the

status of the Export job. When the job has completed, the same id may be used to extract the content of the job.

Export jobs are maintained in memory. Once a job's content has been retrieved it is deleted to free up that memory. Any job whose results has not been previously extracted, and is older than five minutes, will also be deleted. This is to ensure that only one results remains in memory for any length of time.

By default all entity types will be exported. However it is possible to select (several) types from the following list

- Domains
- Tags (Both keys and values)
- Providers
- Products
- Resources
- Relationships

Export as a template to Import - Best Practice

Since the Export API will generate a JSON file that may be directly consumed by the Import API, it is possible to use export as a means of generating the well-formed JSON required by Import as an prototype model for the transformation step described below.

This practice will assure that the model that is exported is completely compatible with Import.

When performing a **brown-field** Import the steps would be to

- Extract the data from the source system
- Transform it into the format expected by Import
- Import or load the transformed data

By using a small pre-populated database and Exporting all types, the exact format for that transformation is made available. And the transform step may use this model as a prototype to follow in constructing data that will be Imported during the final step.

Correcting "bad" data from import - Best Practice

Since Import is intended to allow for a bulk of data to be imported in one request, the amount of validation that is done is minimal. It is possible to import incorrect data. One good example of this is the **providerData** field in the **Product** entity. It is string type field, and as such will accept any legitimate string data. An example of the correct format would be:

```
"providerData": "{\"template\": \"juniper.serviceTemplates.FireflyVcpeVnf\"}"
```

Notice that the actual data is quoted string with the json data using escaped quotes. It would be possible (although not advisable) to import the following data for this field:

```
"providerData": "Now is the time for all good men."
```

If this data were imported the product would import successfully. It is only when this product was retrieved using a GET command, that the 500 status would be returned indicating an internal error, with the details outlining a failure to convert this field.

Solution

The first step is identifying the offending data, and this might take some work. This would entail identifying the bad data perhaps by examining the original import file, and inspecting providerData fields for non-conforming data. Once found, however, we still need to correct the data.

We can do this by Exporting all **Product** data in the database, and identifying the offending product(s) among the exported data.

NOTE

It is important to note that we must use the exported data in the next step, re-import the product data. The reason for this is that the exported product data will contain the Id that is used in the database to identify this product. If your original import used foreign keys, the foreign key is discarded and a database Id is assigned in its place. The exported data will always contain the actual Id used for storage in the database.

Isolate the product(s) into a new import file by using the export data that contains the incorrect item(s). Correct the data (or simply replace providerData with "" if no providerData is to be defined), then re-import this data. This time, set **preProcessKeys** to false, and **updateIfDifferent** to true. This will overwrite the incorrect data and allow the products to once again be retrieved successfully by GET.

Best Practice

In order to minimize the time between the import of bad data, and its discovery, it is considered a best practice to GET all imported types shortly after the import is performed. This will identify any bad data early on, and reduce the possibility of such data being released into a production environment.

Market Component

The **Market** component is the primary component for defining, creating, and managing resources in the orchestration domains.

The Market component includes the following types exposed as REST resources:

- Domain-Type: A type of domain or system exposing orchestration entities (e.g., OpenStack, vCloud, Planet Operate)
- Domain: An instance of a domain-type specified with account credentials that is managed by the orchestrator (e.g., an OpenStack instance, Versa Manager), manages the resources in the domain, and provides the resources to the Market
- Resource-Type: A type of resource instance managed by the orchestrator (e.g., virtual machine, Ethernet Virtual Private Line)
- Product: An offering of a resource type in the market that is available to be instantiated as a resource
- Resource: An instance of a product that has been requested or created
- Relationship: An association between resources (e.g., a source-target dependency relationship)
- Type-Artifact: Defines the meta-information for all type information include Resource-Types (above) as well as other types such as service templates, relationship types, capability types, service types, etc.
- Tag-Key: Arbitrary (tag-key, tag-value) pairs that may be associated with a resource and used for filtering, etc. (e.g., layer)

Resource Types

Each of the objects managed by Blue Planet Orchestrate is an instance of some resource type. Example resource types include:

- Virtual machines (VM)
- Software image
- CPU
- Disk
- Virtual customer premise equipment (vCPE)
- VLAN
- Ethernet Virtual Private Line (EVPL)

Each resource type is identified by its ID. For example:

```
tosca.resourceTypes.P2pVcpe
```

Tutorial: Allocating Numbers in a Number Pool

Planet Orchestrate REST API Tutorial

Introduction

A Number Pool is a useful product that is offered by the built-in resource provider. For example, a virtual appliance may support a fixed number of sub-units or a domain may have a specific range from which VLANs may be allocated with the requirement that a value may be only allocated once. The Planet Orchestrate built-in resource provider supports managing integer pools and address pools. This example demonstrates the use of Planet Orchestrate REST APIs to:

- define an integer number pool
- allocate a number from the number pool
- query the state of the number pool
- delete a pooled number returning the value to the pool
- find the values allocated in a pool

The companion **Number Pool** tutorial is an interactive guide for executing the steps in this tutorial.

Definitions

The **tosca/resource_type_pools.tosca** standard template file defines the resource types from which the products are created that are used for managing pools.

- **NumberPool** - General product for a pool of integer values. Creating a resource instance from the **NumberPool** results in a concrete number pool with a defined range and a corresponding **PooledNumber** product for allocating numbers in the **NumberPool**.
- **PooledNumber** - The product for allocating numbers from a **NumberPool**. Each **PooledNumber** resource is a value allocated from the pool.

Tutorial Steps

1. Find the standard number pool product in the product catalog
2. Create a resource instance of the number pool specifying the range of values. A pooled number product is automatically created as a result of creating this resource.
3. Find the resulting pooled number product
4. Create resource instance(s) using the pooled number product
5. Obtain the value from the pooled number resource(s)
6. Obtain the number of free values in the number pool
7. Return a number to the number pool

8. Allocate a specific value from the pool
9. Allocation Failures
10. Finding All Values Allocated in a Pool

These steps are illustrated in detail below.

Notation

The API used in each step is listed using a URL shorthand starting with the resource. This format omits the scheme, host, port, and base path. Each API should include the following before the resource:

```
https://{{host}}:{{port}}/bpocore/market/api/v1
```

The example displays the full URL as executed against a server on **orchtest1**. The UUIDs and other values displayed are for example purposes only and are not expected to match values produced when running through the tutorial.

1. Find the Standard Number Pool Product

Find the standard product offering for the resource type **tosca.resourceTypes.NumberPool** in the Market's product catalog.

API: GET /products

The REST API filtering syntax may be used as shown in the example below to return only the product based on the resource type.

```
https://orchtest1/bpocore/market/api/v1/products?includeInactive=false&q=(resourceType:tosca.resourceTypes.NumberPool)
```

Response: 200

```
{
  items: [
    {
      id: "f97d1a40-3ff1-5a9d-9bcc-83f65d96ef56",
      resourceType: "tosca.resourceTypes.NumberPool",
      title: "Number Pool",
      active: true,
      provider: "built-in",
      providerData: {}
    }
  ],
  total: 1,
  offset: 0,
  limit: 1000
}
```

2. Create a Number Pool Resource

Creating a number pool resource based on the number pool product found in the previous step. The **productId** and range (specified by **lowest** and **highest** values) are required as input to create the resource.

API: POST /resources

```
POST https://orchtest1/bpocore/market/api/v1/resources
{
  "label": "NumberPoolOf5",
  "productId": "f97d1a40-3ff1-5a9d-9bcc-83f65d96ef56",
  "properties": {
    "lowest": 11,
    "highest": 15
  }
}
```

Response: 201

```
{
  "id": "54aef183-9884-4fd9-b556-14fabcfaba9c",
  "label": "NumberPoolOf5",
  "productId": "f97d1a40-3ff1-5a9d-9bcc-83f65d96ef56",
  "tenantId": "749d45ee-ae78-434c-9b49-5885bd28ace4",
  "properties": {
    "lowest": 11,
    "highest": 15
  },
  "discovered": false,
  "desiredOrchState": "active",
  "orchState": "requested",
  "reason": "",
  "tags": {},
  "providerData": {}
}
```

The response indicates the resource is in the requested orchestration state. The resource must transition to an **orchState** of **active** before numbers may be allocated from the pool. The resource may be polled until it enters the **active** state.

API: GET /resources/{resourceId}

```
GET https://orchtest1/bpocore/market/api/v1/resources/54aef183-9884-4fd9-b556-14fabcfaba9c
```

Response: 200

```
{  
    "id": "54aef183-9884-4fd9-b556-14fabcfaba9c",  
    "label": "NumberPoolOf5",  
    "productId": "f97d1a40-3ff1-5a9d-9bcc-83f65d96ef56",  
    "tenantId": "749d45ee-ae78-434c-9b49-5885bd28ace4",  
    "properties": {  
        "lowest": 11,  
        "highest": 15  
    },  
    "discovered": false,  
    "desiredOrchState": "active",  
    "orchState": "active",  
    "reason": "",  
    "tags": {},  
    "providerData": {  
        "state": [  
            11,  
            15,  
            0  
        ]  
    },  
    "updatedAt": "2015-01-08T21:07:16.566Z",  
    "createdAt": "2015-01-08T21:07:15.864Z"  
}
```

The response shows that the number pool resource is in the **active** state and may now be used in the next step.

3. Find the Pooled Number Product

When the number pool resource is created, the built-in resource provider automatically creates a pooled number product and makes it available in the Market. The next step is to find the product identifier for this new pooled number product. The title of the product is the same as the label specified for the resource. The **providerProductId** is the same as the resource id for the number pool resource created in the previous step (this will be used as a query parameter in the **GET** request). The description also contains the identifier of the resource.

API: GET /products

```
GET
```

```
https://orchtest1/bpocore/market/api/v1/products?includeInactive=false&q=providerProductID:54aef183-9884-4fd9-b556-14fabcfaba9c
```

Response: 200

```
{
    "id": "54aef184-a2a4-4171-9643-a44cab6523fc",
    "resourceType": "tosca.resourceTypes.PooledNumber",
    "title": "NumberPoolOf5",
    "description": "Product to allocate from NumberPool Some(54aef183-9884-4fd9-b556-14fabcfaba9c)",
    "active": true,
    "provider": "built-in",
    "providerProductId": "54aef183-9884-4fd9-b556-14fabcfaba9c",
    "providerData": {}
}
```

4. Allocate a Number Resource from the Pool

Create a resource based on the Pooled Number product. The value will be automatically assigned from the available values in the pool.

API: POST /resources

```
POST /resources
{
    "productId": "54aef184-a2a4-4171-9643-a44cab6523fc"
}
```

Response 201

```
{
    "id": "54aef4a4-5a0c-49c7-b4d7-da49e6d077c1",
    "productId": "54aef184-a2a4-4171-9643-a44cab6523fc",
    "tenantId": "749d45ee-ae78-434c-9b49-5885bd28ace4",
    "properties": {},
    "discovered": false,
    "desiredOrchState": "active",
    "orchState": "requested",
    "reason": "",
    "tags": {},
    "providerData": {}
}
```

5. Get the Allocated Value

As before, it may be necessary to wait briefly until the **orchState** transitions to **active**. Once the resource is active, get the resource data.

API: GET /resources/{resourceId}

```
GET https://orchtest1/bpocore/market/api/v1/resources/54aef4a4-5a0c-49c7-b4d7-da49e6d077c1
```

Response 200

```
{
  "id": "54aef4a4-5a0c-49c7-b4d7-da49e6d077c1",
  "productId": "54aef184-a2a4-4171-9643-a44cab6523fc",
  "tenantId": "749d45ee-ae78-434c-9b49-5885bd28ace4",
  "properties": {
    "value": 11
  },
  "discovered": false,
  "desiredOrchState": "active",
  "orchState": "active",
  "reason": "",
  "tags": {},
  "providerData": {},
  "updatedAt": "2015-01-08T21:20:37.579Z",
  "createdAt": "2015-01-08T21:20:36.781Z"
}
```

Obtain the allocated value from the **properties.value** field in the JSON response. In this case, the value **11** (the first free value in the number pool range) has been allocated from the pool.

6. Obtain Pool Information

In addition to allocating values from the pool, it is possible to obtain information about the pool including the range parameters and the number currently free. When getting a resource, normally only the resource properties that are stored in the database are included in the response. If the **full** argument is passed as **true**, computed or volatile attributes are also included in the response. In the case of a number pool, the number free is a volatile attribute and the **full** argument is needed in order to have it included in the response.

API: GET /resources

```
GET https://orchtest1/bpocore/market/api/v1/resources/54aef183-9884-4fd9-b556-14fabcfaba9c?full=true
```

Response 200

```
{  
    "id": "54aef183-9884-4fd9-b556-14fabcfaba9c",  
    "label": "NumberPoolOf5",  
    "productId": "f97d1a40-3ff1-5a9d-9bcc-83f65d96ef56",  
    "tenantId": "749d45ee-ae78-434c-9b49-5885bd28ace4",  
    "shared": false,  
    "properties": {  
        "size": 5,  
        "free": 4,  
        "lowest": 11,  
        "highest": 15  
    },  
    "discovered": false,  
    "desiredOrchState": "active",  
    "orchState": "active",  
    "reason": "",  
    "tags": {},  
    "providerData": {  
        "state": [  
            11,  
            15,  
            0  
        ]  
    },  
    "updatedAt": "2015-05-13T21:21:19.227Z",  
    "createdAt": "2015-05-13T21:21:19.068Z",  
    "autoClean": false  
}
```

The response shows that the pool size is **5** and after allocating one value from the pool, there are **4** free values.

7. Return a Number to the Pool

In order to return a number to the pool, delete the pooled number resource holding the allocated value.

API: DELETE /resources/{resourceId}

```
https://orchtest1/bpocore/market/api/v1/resources/54aef4a4-5a0c-49c7-b4d7-  
da49e6d077c1
```

Response 204

After the pooled number resource has terminated, the value is returned to the pool. A get of a terminated resource will return a **404** response. A get of the number pool resource will again show that all of the values are free.

API: GET /resources

```
GET https://orchtest1/bpocore/market/api/v1/resources/54aef183-9884-4fd9-b556-14fabcfaba9c?full=true
```

Response: 200 (*Showing just the properties portion of the response*)

```
"properties": {  
    "free": 5,  
    "highest": 15,  
    "lowest": 11,  
    "size": 0  
}
```

8. Allocate a Number Resource from the Pool with a Specific Value

Just as before, create a resource based on the Pooled Number product; however, this time specify the desired value in the **properties.value** attribute. The value will be granted if available.

API: POST /resources

```
POST /resources  
{  
    "productId": "54aef184-a2a4-4171-9643-a44cab6523fc",  
    "properties": {  
        "value": 13  
    }  
}
```

Response: 201

```
{  
    "id": "555be783-94dd-41dc-aff3-35339b06f6a3",  
    "productId": "54aef184-a2a4-4171-9643-a44cab6523fc",  
    "tenantId": "749d45ee-ae78-434c-9b49-5885bd28ace4",  
    "properties": {},  
    "discovered": false,  
    "desiredOrchState": "active",  
    "orchState": "requested",  
    "reason": "",  
    "tags": {},  
    "providerData": {}  
}
```

After the **orchState** transitions to **active**, get the resource data.

API: GET /resources/{resourceId}

```
GET https://orchtest1/bpocore/market/api/v1/resources/555be783-94dd-41dc-aff3-35339b06f6a3
```

Response: 200

```
{  
    "id": "555be783-94dd-41dc-aff3-35339b06f6a3",  
    "productId": "54aef184-a2a4-4171-9643-a44cab6523fc",  
    "tenantId": "749d45ee-ae78-434c-9b49-5885bd28ace4",  
    "properties": {  
        "value": 13  
    },  
    "discovered": false,  
    "desiredOrchState": "active",  
    "orchState": "active",  
    "reason": "",  
    "tags": {},  
    "providerData": {},  
    "updatedAt": "2015-01-08T21:20:37.579Z",  
    "createdAt": "2015-01-08T21:20:36.781Z"  
}
```

The allocated value from the **properties.value** field in the JSON response confirms the request value **13** was available and able to be granted.

9. Allocation Failure

If a pooled number cannot be allocated from the number pool, the pooled number resource will transition to the **failed** state instead of the **active** state. For example, if another allocation of the value **13** is requested, the allocation will fail because it is still in use from the previous step.

API: POST /resources

```
POST /resources  
{  
    "productId": "54aef184-a2a4-4171-9643-a44cab6523fc",  
    "properties": {  
        "value": 13  
    }  
}
```

Response: 201

```
{
  "id": "555bea46-7cdb-494e-ab8b-f6acdda35562",
  "productId": "54aef184-a2a4-4171-9643-a44cab6523fc",
  "tenantId": "749d45ee-ae78-434c-9b49-5885bd28ace4",
  "properties": {},
  "discovered": false,
  "desiredOrchState": "active",
  "orchState": "requested",
  "reason": "",
  "tags": {},
  "providerData": {}
}
```

Note that the creation of the resource is still successful with a **201** response; however, the **orchState** will transition to **failed** instead of **active**.

API: GET /resources/{resourceId}

```
GET https://orchtest1/bpocore/market/api/v1/resources/555bea46-7cdb-494e-ab8b-f6acdda35562
```

Response: 200

```
{
  "id": "555bea46-7cdb-494e-ab8b-f6acdda35562",
  "productId": "54aef184-a2a4-4171-9643-a44cab6523fc",
  "tenantId": "749d45ee-ae78-434c-9b49-5885bd28ace4",
  "properties": {
    "value": 13
  },
  "discovered": false,
  "desiredOrchState": "active",
  "orchState": "failed",
  "reason": "Value 13 is not available in the pool",
  "tags": {},
  "providerData": {},
  "updatedAt": "2015-01-08T21:20:37.579Z",
  "createdAt": "2015-01-08T21:20:36.781Z"
}
```

The **reason** data indicates why the activation failed.

10. Finding Pooled Numbers Allocated from a Number Pool

In order to find all of the current pooled number resources allocated from a number pool, use the **productId** query parameter when getting resources and further filter to only those resources currently with an **orchState** of **active**.

API: Get/resources

```
GET https://orchtest1/bpocore/market/api/v1/resources?productId=54aef184-a2a4-4171-9643-a44cab6523fc&q=orchState:active
```

Response: 200

```
{  
    "items": [  
        {  
            "id": "555be783-94dd-41dc-aff3-35339b06f6a3",  
            "productId": "54aef184-a2a4-4171-9643-a44cab6523fc",  
            "tenantId": "749d45ee-ae78-434c-9b49-5885bd28ace4",  
            "properties": {  
                "value": 13  
            },  
            "discovered": false,  
            "desiredOrchState": "active",  
            "orchState": "active",  
            "reason": "",  
            "tags": {},  
            "providerData": {},  
            "updatedAt": "2015-01-08T21:20:37.579Z",  
            "createdAt": "2015-01-08T21:20:36.781Z"  
        }  
    ],  
    "limit": 1000,  
    "offset": 0,  
    "total": 1  
}
```

Policy Manager

The **Policy Manager** is responsible for implementing authorization and other policies for the system. The **Authentication** component is responsible for determining identity while the **Policy Manager** is responsible for determining the operations and data that the user identity is authorized to have access.

The Policy Manager exposes APIs for the following resources:

- Realms: Define the scope or context to which a policy applies. For example, each **component** within the system has a defined realm allowing policies to be defined on a per-component basis.
- Conditions: A boolean statement used to build the applicability rule for a policy.
- Policies: A policy that defines matching conditions and policy actions.

Resource Provider

The **Resource Provider** exposes APIs to access information about the resource providers known by the system. For example, performing a **GET** to list all of the resource providers provides a response that shows the:

- resource provider unique identifier
- title to display for the resource provider
- type of resource provider
- resource types provided by the resource provider

The example response is shown below:

```
{  
  "items": [  
    {  
      "id": "54be8b54-b7aa-4ef7-bf77-4e3f208329a3",  
      "title": "demo",  
      "rpType": "OpenStack",  
      "resourceTypes": [  
        "tosca.resourceTypes.Hypervisor",  
        "tosca.resourceTypes.VlanPool",  
        "tosca.resourceTypes.EthernetPort",  
        "tosca.resourceTypes.HeatStack",  
        "tosca.resourceTypes.Subnet",  
        "tosca.resourceTypes.Image",  
        "tosca.resourceTypes.SecurityGroup",  
        "tosca.resourceTypes.DataCenter",  
        "tosca.resourceTypes.L2Gateway",  
        "tosca.resourceTypes.EndPoint",  
        "tosca.resourceTypes.KeyPair",  
        "tosca.resourceTypes.VirtualMachine",  
        "tosca.resourceTypes.EthernetNetwork",  
        "tosca.resourceTypes.L2Vpn",  
        "tosca.resourceTypes.GatewayPort"  
      ],  
      "internal": false,  
      "alive": true  
    }  
  ],  
  "total": 1,  
  "offset": 0  
}
```

The **Resource Provider** component exposes a single top-level resource, **resource-providers**. It defines the following sub-resources:

- Product - An offering of a resource type in the market from a specified resource provider that is available to be instantiated as a resource
- Resource - An instance of a product from a specified resource provider that has been requested or

created

- Relationship - An association between resources (e.g., a source-target dependency relationship)

Rest-Server

The **Rest-Server** component is the coordinating hub for all of the other components that expose RESTful APIs. The Rest-Server component exposes an API to list all of the other components. This API may be used to **seed** API discovery for the system. Each component is listed with a hypermedia **links** attribute that includes the URI used to reach the component's Swagger API specification.

For example:

```
{  
  "items": [  
    {  
      "name": "events",  
      "title": "Events",  
      "links": [  
        {  
          "rel": "spec",  
          "uri": "http://localhost:8181/bpocore/events/api-doc/v1"  
        }  
      ]  
    },  
    {  
      "name": "policies",  
      "title": "Policy Manager",  
      "links": [  
        {  
          "rel": "spec",  
          "uri": "http://localhost:8181/bpocore/policies/api-doc/v1"  
        }  
      ]  
    },  
    {  
      "name": "asset-manager",  
      "title": "Asset Manager",  
      "links": [  
        {  
          "rel": "spec",  
          "uri": "http://localhost:8181/bpocore/asset-manager/api-doc/v1"  
        }  
      ]  
    },  
    {  
      "name": "market",  
      "title": "Market",  
      "links": [  
        {  
          "rel": "spec",  
          "uri": "http://localhost:8181/bpocore/market/api-doc/v1"  
        }  
      ]  
    }  
,  
  "total": 4,  
  "offset": 0  
}
```