Modeling Concepts

Migrated to ReadTheDocs

Further updates must be done in the corresponding RST file(s) by following the build process for documentation.

https://docs.onap.org/projects/onap-ccsdk-cds/en/latest/modelingconcepts/index.html

2020-08-11, Jakob Krieg

CDS is a framework to automate the **resolution of resources** for **instantiation** and any **config** provisioning operation, such as day0, day1 or day2 configuration.

CDS has a both **design time** and **run time** activities; during design time, **Designer** can **define** what **actions** are required for a given service, along with anything comprising the action. The design produce a CBA Package . Its **content** is driven from a **catalog** of **reusable data dictionary** and **comp onent**, delivering a reusable and simplified **self service** experience.

CDS modelling is mainly based on TOSCA standard, using JSON as reprensentation.

Most of the TOSCA modeled entity presented in the bellow documentation can be found here.

CBA

Controller Blueprint Archive (.cba)

The **C**ontroller **B**lueprint **A**rchive is the overall service design, fully model-driven, intent based **package** needed for provisioning and configuration management automation.

The CBA is **.zip** file, comprised of the following folder structure, the files may vary:

Table of Contents

- Controller Blueprint Archive
- Tosca Meta
- Dynamic payload
- Modeling Concepts#Enrichment
- External Systems support
- Modeling Concepts#Expression
  - get_input
  - get_property
  - get_attribute
  - get_operation_output
  - get_artifact
- Data Dictionary
- Data Type
  - Property
  - Resource Assignment
- Artifact Type
  - Resource Mapping
  - Jinja Template
  - Velocity Template
  - Directed Graph
- Node Type
  - Component
    - Resource Resolution
    - Script Executor
    - Remote Python Executor
    - Remote Ansible Executor
  - Source
    - Modeling Concepts #Input
    - Default
    - Rest
    - SQL
    - Modeling Concepts #Capability
  - Other
    - DG Generic
    - VNF
- Workflow
  - Single Action
  - Multi Action
  - Properties
- Template
  - Resource Accumulator
- Scripts
  - Netconf Client
  - Resolution Helper
- Southbound Interfaces
- Tests


```
 Definitions
    blueprint.json
Overall TOSCA service template (workflow + node_template)
    artifact_types.json                                          (generated
by enrichment)
    data_types.json
(generated by enrichment)
    policy_types.json
(generated by enrichment)
    node_types.json
(generated by enrichment)
    relationship_types.json                                  (generated by
enrichment)
    resources_definition_types.json                (generated by
enrichment)
|    *-mapping.json
One per Template

Environments
Contains *.properties files as required by the service

Plans
Contains Directed Graph

Tests
Contains uat.yaml file for testing cba actions within a cba package

Scripts
Contains scripts

python
Python scripts

kotlin
Kotlin scripts
 TOSCA-Metadata
    TOSCA.
meta                                                                Meta-
data of overall package

Templates
Contains combination of mapping and template
```

To process a CBA for any service we need to enrich it first. This will gather all the node-type, data-type, artifact-type, data-dictionary definitions provided in the blueprint.json.

Tosca.Meta

Tosca Meta

Tosca meta file is captures the model entities that compose the cba package name, version, type and searchable tags.

Attribute	R /C /O	Data Type	Description
TOSCA-Meta-File-Version	Required	String	The attribute that holds TOSCA-Meta-File-Version. Set to 1.0.0
CSAR-Version	Required	String	The attribute that holds CSAR-version. Set to 1.0
Created-By	Required	String	The attribute that holds the entry points
Entry-Definitions	Required	String	The attribute that holds the entry points file PATH to the main cba tosca definition file or non tosca script file.
Template-Name	Required	String	The attribute that holds the blueprint name

Template-Version	Required	String	The attribute that holds the blueprint version X.Y.Z X=Major version Y=Minor Version Z=Revision Version Ex. 1.0.0
Template-Type	Required	String	The attribute that holds the blueprint package types. Valid Options: • "DEFAULT" – .JSON file consistent of tosca based cba package that describes the package intent. • "KOTLIN_DSL" – .KT file consistent of tosca based cba package that describes the package intent composed using Domain Specific Language (DSL). • "GENERIC_SCRIPT" – Script file consistent of NONE tosca based cba package that describes the package intent using DSL Language. If not specified in the tosca.meta file the default is "DEFAULT"
Template-Tags	Required	String	The attribute that holds the blueprint package comma delimited list of Searchable attributes.

Template Type Reference

Default Template Type

https://git.onap.org/ccsdk/cds/tree/components/model-catalog/blueprint-model/test-blueprint/capability_cli/TOSCA-Metadata/TOSCA.meta

KOTLIN_DSL Template Type

https://git.onap.org/ccsdk/cds/tree/components/model-catalog/blueprint-model/test-blueprint/resource-audit/TOSCA-Metadata/TOSCA.meta

GENERIC_SCRIPT Template Type

https://git.onap.org/ccsdk/cds/tree/components/model-catalog/blueprint-model/test-blueprint/capability_python/TOSCA-Metadata/TOSCA.meta

Dynamic Payload

Dynamic payload

One of the most important API provided by the run time is to execute a CBA Package.

The nature of this API **request** and **response** is **model driven** and **dynamic**.

Here is how the a **generic request** and **response** look like.

request

```
{
  "commonHeader": {
    "originatorId": "",
    "requestId": "",
    "subRequestId": ""
  },
  "actionIdentifiers": {
    "blueprintName": "",
    "blueprintVersion": "",
    "actionName": "",
    "mode": ""
  },
  "payload": {
    "$actionName-request": {
      "$actionName-properties": {
      }
    }
  }
}
```

response

```
{
  "commonHeader": {
    "originatorId": "",
    "requestId": "",
    "subRequestId": ""
  },
  "actionIdentifiers": {
    "blueprintName": "",
    "blueprintVersion": "",
    "actionName": "",
    "mode": ""
  },
  "payload": {
    "$actionName-response": {
    }
  }
}
```

The **`actionName`**, under the `actionIdentifiers` refers to the name of a Workflow (see Modeling Concepts#workflow)

The content of the **`payload`** is what is fully dynamic / model driven.

The first **top level element** will always be either `$actionName-request` for a request or `$actionName-response` for a response.

Then the **content within this element** is fully based on the **workflow inputs** and **outputs**.

During the **Enrichment**, CDS will aggregate all the resources defined to be resolved as input, within mapping definition files, as data-type, that will then be use as type of an input called `$actionName-properties`.

Enrichment

Enrichment

The idea is that the CBA is a self-sufficient package, hence requires all the various types definition its using.

Reason for this is the types its using might evolve. In order for the CBA to be bounded to the version it has been using when it has been designed, these types are embedded in the CBA, so if they change, the CBA is not affected.

The enrichment process will complete the package by providing all the definition of types used:

- gather all the node-type used and put them into a `node_types.json` file
- gather all the data-type used and put them into a `data_types.json` file
- gather all the artifact-type used and put them into a `artifact_types.json` file
- gather all the data dictionary definitions used from within the mapping files and put them into a `resources_definition_types.json` file

Before uploading a CBA, it must be enriched. If your package is already enrich, you do not need to perform enrichment again.

The enrichment can be run using REST API, and required the **.zip** file as input. It will return an **enriched-cba.zip** file.

```
curl -X POST \
  'http://{{ip}}:{{cds-designtime}}/api/v1/blueprint-model/enrich' \
  -H 'content-type: multipart/form-data' \
  -F file=@cba.zip
```

The enrichment process will also, for all resources to be resolved as input and default:

- dynamically gather them under a data-type, named `dt-${actionName}-properties`
- will add it as a input of the workflow, as follow using this name: `${actionName}-properties`

Example for workflow named *resource-assignment*:

dynamic input

```
  "resource-assignment-properties": {
    "required": true,
    "type": "dt-resource-assignment-properties"
  }
```

Flexible Plug-In

External Systems support

Interaction with **external systems** is made **dynamic**, removing development cycle to support new endpoint.

In order to define the external system information, TOSCA provides **`dsl_definitions`**. Link to TOSCA spec info 1, info 2.

Use cases:

- Resource resolution using **Modeling Concepts#REST** or **Modeling Concepts#SQL** external systems
- **gRPC** is supported for remote execution
- Any REST endpoint can be dynamically injected as part of the scripting framework.

Here are some examples on how to populate the system information within the package:

token-auth

```
{
  . . .
  "dsl_definitions": {
    "ipam-1": {
      "type": "token-auth",
      "url": "http://netbox-nginx.netprog:8080",
      "token": "Token 0123456789abcdef0123456789abcdef01234567"
    }
  }
  . . .
}
```

basic-auth

```
{
  . . .
  "dsl_definitions": {
    "ipam-1": {
      "type": "basic-auth",
      "url": "http://localhost:8080",
      "username": "bob",
      "password": "marley"
    }
  }
  . . .
}
```

ssl-basic-auth

```
{
  . . .
  "dsl_definitions": {
    "ipam-1": {
      "type" : "ssl-basic-auth",
      "url" : "http://localhost:32778",
      "keyStoreInstance": "JKS or PKCS12",
      "sslTrust": "trusture",
      "sslTrustPassword": "trustore password",
      "sslKey": "keystore",
      "sslKeyPassword: "keystore password"
    }
  }
  . . .
}
```

grpc-executor

```
{
  . . .
  "dsl_definitions": {
    "remote-executor": {
      "type": "token-auth",
      "host": "cds-command-executor.netprog",
      "port": "50051",
      "token": "Basic Y2NzZGthcHBzOmNjc2RrYXBwcw=="
    }
  }
  . . .
}
```

maria-db

```
{
  . . .
  "dsl_definitions": {
    "netprog-db": {
      "type": "maria-db",
      "url": "jdbc:mysql://10.195.196.123:32050/netprog",
      "username": "netprog",
      "password": "netprog"
    }
  }
  . . .
}
```

Expression

Expression

TOSCA provides for a set of functions to reference elements within the template or to retrieve runtime values.

Below is a list of supported expressions

get_input

get_input

The **get_input** function is used to retrieve the values of properties declared within the inputs section of a TOSCA Service Template.

Within CDS, this is mainly Workflow inputs.

TOSCA specification

Example:

https://github.com/onap/ccsdk-cds/blob/master/components/model-catalog/blueprint-model/test-blueprint/golden/Definitions/golden-blueprint.json#L210

```
"resolution-key": {
  "get_input": "resolution-key"
}
```

get_property

get_property

The **get_property** function is used to retrieve property values between modelable entities defined in the same service template.

TOSCA specification

Example

TBD

get_attribute

get_attribute

The **get_attribute** function is used to retrieve the values of named attributes declared by the referenced node or relationship template name.

TOSCA specification

Example:

https://github.com/onap/ccsdk-cds/blob/master/components/model-catalog/blueprint-model/test-blueprint/golden/Definitions/golden-blueprint.json#L64-L67

```
"get_attribute": [
  "resource-assignment",
  "assignment-params"
]
```

get_operation_output

get_operation_output

The **get_operation_output** function is used to retrieve the values of variables exposed / exported from an interface operation.

TOSCA specification

Example

TBD

get_artifact

get_artifact

The get_artifact function is used to retrieve artifact location between modelable entities defined in the same service template.

TOSCA specification

Example

TBD

Data dictionary

Data Dictionary

A data dictionary **models the how** a specific **resource** can be **resolved**.

A resource is a **variable/parameter** in the context of the service. It can be anything, but it should not be confused with SDC or Openstack resources.

A data dictionary can have **multiple sources** to handle resolution in different ways.

The main goal of data dictionary is to define **re-usable** entity that could be shared.

Creation of data dictionaries is a **standalone** activity, separated from the blueprint design.

As part of modelling a data dictionary entry, the following generic information should be provided:

Property	Description	Scope
name	Data dictionary name	Mandatory
tags	Information related	Mandatory
updated-by	The creator	Mandatory
property	Defines type and description, as nested JSON	Mandatory
sources	List of resource source instance (see resource source)	Mandatory

Bellow are properties that all the resource source can have

The modeling does allow for **data translation** between external capability and CDS for both input and output key mapping.

Property	Description	Scope
input-key-mapping	map of resources required to perform the request/query. The left hand-side is what is used within the query/request, the right hand side refer to a data dictionary instance.	Optional
output-key-mapping	name of the resource to be resolved mapped to the value resolved by the request/query.	Optional
key-dependencies	list of data dictionary instances to be resolved prior the resolution of this specific resource. during run time execution the key dependencies are recursively sorted and resolved in batch processing using the acyclic graph algorithm.	Optional

Example:
`vf-module-model-customization-uuid` and `vf-module-label` are two data dictionaries. A SQL table, VF_MODULE_MODEL, exist to correlate them.

Here is how input-key-mapping, output-key-mapping and key-dependencies can be used:

vf-module-label data dictionary

```
{
  "name" : "vf-module-label",
  "tags" : "vf-module-label",
  "updated-by" : "adetalhouet",
  "property" : {
    "description" : "vf-module-label",
    "type" : "string"
  },
  "sources" : {
    "primary-db" : {
      "type" : "source-primary-db",
      "properties" : {
        "type" : "SQL",
        "query" : "select sdnctl.VF_MODULE_MODEL.vf_module_label as
vf_module_label from sdnctl.VF_MODULE_MODEL where sdnctl.VF_MODULE_MODEL.
customization_uuid=:customizationid",
        "input-key-mapping" : {
          "customizationid" : "vf-module-model-customization-uuid"
        },
        "output-key-mapping" : {
          "vf-module-label" : "vf_module_label"
        },
        "key-dependencies" : [ "vf-module-model-customization-uuid" ]
      }
    }
  }
}
```

Data Type

Data type

Represents the **schema** of a specific type of **data**.

Supports both **primitive** and **complex** data types:

Primitive	Complex
• string • integer • float • double • boolean • timestamp • null	• json • list • array

For complex data type, an **entry schema** is required, defining the type of value contained within the complex type, if list or array.

Users can **create** as many **data type** as needed.

Creating Custom Data Types

To create a custom data-type you can use a POST call to CDS endpoint: "<cds-ip>:<cds-port>/api/v1/model-type"
Payload:

```
{

  "model-name": "<model-name>",
  "derivedFrom": "tosca.datatypes.Root",

  "definitionType": "data_type",

  "definition": {

    "description": "<description>",

    "version": "<version-number: eg 1.0.0>",

    "properties": {<add properties of your custom data type in JSON format>},

    "derived_from": "tosca.datatypes.Root"

  },

  "description": "<description",

  "version": "<version>",

  "tags": "<model-name>,datatypes.Root.data_type",

  "creationDate": "<creation timestamp>",

  "updatedBy": "<name>"

}
```

Data type are useful to manipulate data during resource resolution. They can be used to format the JSON output as needed.

List of existing data type: https://github.com/onap/ccsdk-cds/tree/master/components/model-catalog/definition-type/starter-type/data_type

TOSCA specification

Below is a list of existing data types

resources-assignment

datatype-resource-assignment

Used to define entries within Modeling Concepts#artifact-mapping-resource

That datatype represent a **resource** to be resolved. We also refer this as an **instance of a data dictionary** as it's directly linked to its definition.

Property	Description
property	Defines how the resource looks like (see Modeling Concepts#datatype-property)
input-param	Whether the resource can be provided as input.
dictionary-name	Reference to the name of the data dictionary (see Data Dictionary).
dictionary-source	Reference the source to use to resolve the resource (see Resource source).
dependencies	List of dependencies required to resolve this resource.
updated-date	Date when mapping was upload.
updated-by	Name of the person that updated the mapping.

https://github.com/onap/ccsdk-cds/blob/master/components/model-catalog/definition-type/starter-type/data_type/datatype-resource-assignment.json

datatype-resource-assignment

```
{
        "version": "1.0.0",
        "description": "This is Resource Assignment Data Type",
        "properties": {
                "property": {
                        "required": true,
                        "type": "datatype-property"
                },
                "input-param": {
                        "required": true,
                        "type": "boolean"
                },
                "dictionary-name": {
                        "required": false,
                        "type": "string"
                },
                "dictionary-source": {
                        "required": false,
                        "type": "string"
                },
                "dependencies": {
                        "required": true,
                        "type": "list",
                        "entry_schema": {
                                "type": "string"
                        }
                },
                "updated-date": {
                        "required": false,
                        "type": "string"
                },
                "updated-by": {
                        "required": false,
                        "type": "string"
                }
        },
        "derived_from": "tosca.datatypes.Root"
}
```

property

datatype-property

Used to defined the **property** entry of a **resource assignment**.

Property	Description
type	Whether it's a primitive type, or a defined data-type
description	Description of for the property
required	Whether it's required or not
default	If there is a default value to provide
entry_schema	If the type is a complex one, such as list, define what is the type of element within the list.

https://github.com/onap/ccsdk-cds/blob/master/components/model-catalog/definition-type/starter-type/data_type/datatype-property.json

datatype-property

```
{
  "version": "1.0.0",
  "description": "This is Entry point Input Data Type, which is dynamic
datatype, The parameter names will be populated during the Design time for
each inputs",
  "properties": {
    "type": {
      "required": true,
      "type": "string"
    },
    "description": {
      "required": false,
      "type": "string"
    },
    "required": {
      "required": false,
      "type": "boolean"
    },
    "default": {
      "required": false,
      "type": "string"
    },
    "entry_schema": {
      "required": false,
      "type": "string"
    }
  },
  "derived_from": "tosca.datatypes.Root"
}
```

Artifact Type

Artifact Type

Represents the **type of a artifact**, used to **identify** the **implementation** of the functionality supporting this type of artifact.

TOSCA definition

This node was created, derived from `tosca.artifacts.Root` to be the root TOSCA node for all artifact.

tosca.artifacts.Implementation

```
{
  "description": "TOSCA base type for implementation artifacts",
  "version": "1.0.0",
  "derived_from": "tosca.artifacts.Root"
}
```

Bellow is a list of supported artifact types

Velocity

artifact-template-velocity

Represents an Apache Velocity template.

Apache Velocity allow to insert **logic** (if / else / loops / etc) when processing the output of a template/text.

File must have **.vtl** extension.

The **template** can represent **anything**, such as device config, payload to interact with 3rd party systems, resource-accumulator template, etc...

Often a template will be **parameterized**, and each **parameter** must be defined within an **mapping** file.

Velocity reference document

Here is the TOSCA artifact type:

artifact-template-velocity

```
{
  "description": " Velocity Template used for Configuration",
  "version": "1.0.0",
  "file_ext": [
    "vtl"
  ],
  "derived_from": "tosca.artifacts.Implementation"
}
```

Jinja

artifact-template-jinja

Represents an Jinja template.

Jinja template allow to insert **logic** (if / else / loops / etc) when processing the output of a template/text.

File must have **.jinja** extension.

The **template** can represent **anything**, such as device config, payload to interact with 3rd party systems, resource-accumulator template, etc...

Often a template will be **parameterized**, and each **parameter** must be defined within an **mapping** file.

Jinja reference document

Here is the TOSCA artifact type:

artifact-template-jinja

```
{
  "description": " Jinja Template used for Configuration",
  "version": "1.0.0",
  "file_ext": [
    "jinja"
  ],
  "derived_from": "tosca.artifacts.Implementation"
}
```

Mapping

artifact-mapping-resource

This type is meant to represent **mapping** files defining the **contract of each resource** to be resolved.

Each **parameter** in a template **must** have a corresponding mapping definition, modeled using Modeling Concepts#datatype-resource-assignment.

Hence the mapping file is meant to be a **list of** entries defined using **Modeling Concepts#datatype-resource-assignment**.

File must have **.json** extension.

Here is the TOSCA artifact type:

artifact-mapping-resource

```
{
  "description": "Resource Mapping File used along with Configuration
template",
  "version": "1.0.0",
  "file_ext": [
    "json"
  ],
  "derived_from": "tosca.artifacts.Implementation"
}
```

The mapping file basically contains a reference to the data dictionary to use to resolve a particular resource.

The data dictionary defines the HOW and the mapping defines the WHAT.

Relation between data dictionary, mapping and template.

Below are two examples using color coding to help understand the relationships.

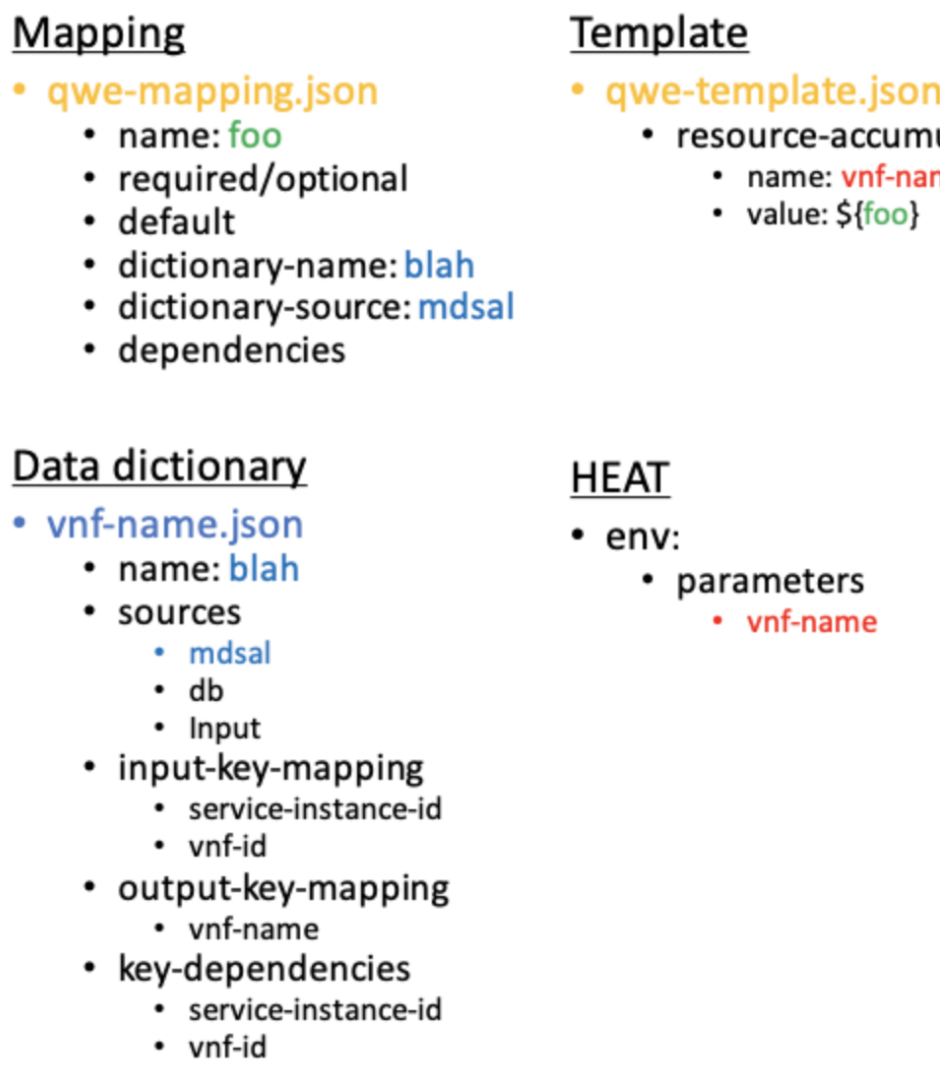
In orange is the information regarding the template. As mentioned before, template is part of the blueprint itself, and for the blueprint to know what template to use, the name has to match.

In green is the relationship between the value resolved within the template, and how it's mapped coming from the blueprint.

In blue is the relationship between a resource mapping to a data dictionary.

In red is the relationship between the resource name to be resolved and the HEAT environment variables.

The key takeaway here is that whatever the value is for each color, it has to match all across. This means both right and left hand side are equivalent; it's all on the designer to express the modeling for the service. That said, best practice is example 1.



Directed Graph

artifact-directed-graph

Represents a directed graph.

This is to represent a **workflow**.

File must have **.xml** extension.

Here is the list of executors currently supported (see here for explanation and full potential list: Service Logic Interpreter Nodes)

- execute
- block
- return
- break
- exit

Here is the TOSCA artifact type:

artifact-directed-graph

```
{
  "description": "Directed Graph File",
  "version": "1.0.0",
  "file_ext": [
    "json",
    "xml"
  ],
  "derived_from": "tosca.artifacts.Implementation"
}
```

Node Type

Node type

TOSCA definition

In CDS, we have mainly two distinct types: components and source. We have some other type as well, listed in the other section.

Component

Component

Used to represent a **functionality** along with its **contract**, such as **inputs**, **ouputs**, and **attributes**

Here is the root component TOSCA node type from which other node type will derive:

tosca.nodes.Component

```
{
  "description": "This is default Component Node",
  "version": "1.0.0",
  "derived_from": "tosca.nodes.Root"
}
```

Bellow is a list of supported components

resource-resolution

component-resource-resolution

Used to perform **resolution of resources.**

Requires as many as Modeling Concepts#artifact-mapping-resource AND Modeling Concepts#artifact-template-velocity or artifact-template-jinja as needed.

Ouput result

Will put the resolution result as an **attribute** in the workflow context called **assignment-params**.

Using the Modeling Concepts#get_attribute expression, this attribute can be retrieve to be provided as workflow output (see Workflow).

Specify which template to resolve

Currently, resolution is bounded to a template. To specify which template to use, you need to fill in the `artifact-prefix-names` field.

See Template to understand what the artifact prefix name is.

Storing the result

To store each resource being resolved, along with their status, and the resolved template, `store-result` should be set to `true`.

Also, when storing the data, it must be in the context of either a `resource-id` and resource-type`, or based on a given `resolution-key`

The concept of resource-id / resource-type, or resolution-key, is to uniquely identify a specific resolution that has been performed for a given action. Hence the resolution-key has to be unique for a given blueprint name, blueprint version, action name.

Through the combination of the fields mentioned previously, one could retrieved what has been resolved. This is useful to manage the life-cycle of the resolved resource, the life-cycle of the template, along with sharing with external systems the outcome of a given resolution.

The resource-id / resource-type combo is more geared to uniquely identify a resource in AAI, or external system. For example, for a given AAI resource, say a PNF, you can trigger a given CDS action, and then you will be able to manage all the resolved resources bound to this PNF. Even we could have a history of what has been assigned, unassigned for this given AAI resource.

Important not to confuse and AAI resource (e.g. a topology element, or service related element) with the resources resolved by CDS, which can be seen as parameters required to derived a network configuration.

Run the resolution multiple time

If you need to run the same resolution component multiple times, use the field `occurence`. This will add the notion of occurrence to the resolution, and if storing the results, resources and templates, they will be accessible for each occurrence.

Occurrence is a number between 1 and N; when retrieving information for a given occurrence, the first iteration starts at 1.

This feature is usefull when you need to apply the same configuration accross network elements.

Here is the definition:

component-resource-resolution

```
{
  "description": "This is Resource Assignment Component API",
  "version": "1.0.0",
  "attributes": {
    "assignment-params": {
      "required": true,
      "type": "string"
    }
  },
  "capabilities": {
    "component-node": {
      "type": "tosca.capabilities.Node"
    }
  },
  "interfaces": {
    "ResourceResolutionComponent": {
      "operations": {
        "process": {
          "inputs": {
            "resolution-key": {
              "description": "Key for service instance related
correlation.",
              "required": false,
              "type": "string"
            },
```

```
            "occurrence": {
              "description": "Number of time to perform the resolution.",
              "required": false,
              "default": 1,
              "type": "integer"
            },
            "store-result": {
              "description": "Whether or not to store the output.",
              "required": false,
              "type": "boolean"
            },
            "resource-type": {
              "description": "Request type.",
              "required": false,
              "type": "string"
            },
            "artifact-prefix-names": {
              "required": true,
              "description": "Template , Resource Assignment Artifact
Prefix names",
              "type": "list",
              "entry_schema": {
                "type": "string"
              }
            },
            "request-id": {
              "description": "Request Id, Unique Id for the request.",
              "required": true,
              "type": "string"
            },
            "resource-id": {
              "description": "Resource Id.",
              "required": false,
              "type": "string"
            },
            "action-name": {
              "description": "Action Name of the process",
              "required": false,
              "type": "string"
            },
            "dynamic-properties": {
              "description": "Dynamic Json Content or DSL Json reference.",
              "required": false,
              "type": "json"
            }
          },
          "outputs": {
            "resource-assignment-params": {
              "required": true,
              "type": "string"
            },
            "status": {
              "required": true,
              "type": "string"
            }
          }
        }
      }
    }
  },
  "derived_from": "tosca.nodes.Component"
}
```

script-executor

component-script-executor

Used to **execute** a script to perform **NETCONF, RESTCONF, SSH commands** from within the runtime container of CDS.

Two type of scripts are supported:

- Kotlin: offer a way more integrated scripting framework, along with a way faster processing capability. See more about Kotlin script: https://github.com/Kotlin/KEEP/blob/master/proposals/scripting-support.md
- Python: uses Jython which is bound to Python 2.7, end of life Januray 2020. See more about Jython: https://www.jython.org/

The `script-class-reference` field need to reference

- for kotlin: the package name up to the class. e.g. com.example.Bob
- for python: it has to be the path from the Scripts folder, e.g. Scripts/python/Bob.py

Here is the definition

component-script-executor

```
{
  "description": "This is Netconf Transaction Configuration Component API",
  "version": "1.0.0",
  "interfaces": {
    "ComponentScriptExecutor": {
      "operations": {
        "process": {
          "inputs": {
            "script-type": {
              "description": "Script type, kotlin type is supported",
              "required": true,
              "type": "string",
              "default": "internal",
              "constraints": [
                {
                  "valid_values": [
                    "kotlin",
                    "jython",
                    "internal"
                  ]
                }
              ]
            },
            "script-class-reference": {
              "description": "Kotlin Script class name with full package
or jython script name.",
              "required": true,
              "type": "string"
            },
            "dynamic-properties": {
              "description": "Dynamic Json Content or DSL Json reference.",
              "required": false,
              "type": "json"
            }
          },
          "outputs": {
            "response-data": {
              "description": "Execution Response Data in JSON format.",
              "required": false,
              "type": "string"
            },
            "status": {
              "description": "Status of the Component Execution ( success
or failure )",
              "required": true,
              "type": "string"
            }
          }
        }
      }
    }
  },
  "derived_from": "tosca.nodes.Component"
}
```

remote-script-executor

component-remote-script-executor

Used to **execute** a python script in a dedicated micro-service, providing a Python 3.6 environment.

Ouput result

prepare-environment-logs: will contain the logs for all the pip install of ansible_galaxy setup

execute-command-logs: will contain the execution logs of the script, that were printed into stdout

Using the Modeling Concepts#get_attribute expression, this attribute can be retrieve to be provided as workflow output (see Workflow).

Params

The `command` field need to reference the path from the Scripts folder of the scripts to execute, e.g. Scripts/python/Bob.py

The `packages` field allow to provide a list of **PIP package** to install in the target environment, or a requirements.txt file. Also, it supports **Ansible role**.
If **requirements.txt** is specified, then it should be **provided** as part of the **Environment** folder of the CBA.

Example

```
"packages": [
    {
      "type": "pip",
      "package": [
        "requirements.txt"
      ]
    },
    {
      "type": "ansible_galaxy",
      "package": [
        "juniper.junos"
      ]
    }
  ]
```

The `argument-properties` allows to specified input argument to the script to execute. They should be expressed in a DSL, and they will be ordered as specified.

Example

```
    "ansible-argument-properties": {
      "arg0": "-i",
      "arg1": "Scripts/ansible/inventory.yaml",
      "arg2": "--extra-vars",
      "arg3": {
        "get_attribute": [
          "resolve-ansible-vars",
          "",
          "assignment-params",
          "ansible-vars"
        ]
      }
    }
  }
```

The `dynamic-properties` can be anything that needs to be passed to the script that couldn't be passed as an argument, such as JSON object, etc... If used, they will be passed in as the last argument of the Python script.

Here is the definition

component-remote-script-executor

```
{
  "description": "This is Remote Python Execution Component.",
  "version": "1.0.0",
  "attributes": {
    "prepare-environment-logs": {
      "required": false,
      "type": "string"
    },
    "execute-command-logs": {
      "required": false,
      "type": "list",
      "entry_schema": {
        "type": "string"
      }
    },
    "response-data": {
      "required": false,
      "type": "json"
    }
  },
  "capabilities": {
    "component-node": {
      "type": "tosca.capabilities.Node"
    }
  },
  "interfaces": {
    "ComponentRemotePythonExecutor": {
      "operations": {
        "process": {
          "inputs": {
            "endpoint-selector": {
              "description": "Remote Container or Server selector name.",
              "required": false,
              "type": "string",
              "default": "remote-python"
            },
            "dynamic-properties": {
              "description": "Dynamic Json Content or DSL Json reference.",
              "required": false,
              "type": "json"
            },
            "argument-properties": {
              "description": "Argument Json Content or DSL Json
reference.",
              "required": false,
              "type": "json"
            },
            "command": {
              "description": "Command to execute.",
              "required": true,
              "type": "string"
            },
            "packages": {
              "description": "Packages to install based on type.",
              "required": false,
              "type" : "list",
              "entry_schema" : {
                "type" : "dt-system-packages"
              }
            }
          }
        }
      }
    }
  },
  "derived_from": "tosca.nodes.Component"
}
```

remote-ansible-executor

component-remote-ansible-executor

Used to **execute** an ansible playbook hosted in AWX/Anisble Tower.

Ouput result

ansible-command-status: status of the command

ansible-command-logs: will contain the execution logs of the playbook

Using the Modeling Concepts#get_attribute expression, this attribute can be retrieve to be provided as workflow output (see Workflow).

Param

TBD

Here is the definition

component-remote-script-executor

```
{
  "description": "This is Remote Ansible Playbook (AWX) Execution
Component.",
  "version": "1.0.0",
  "attributes": {
    "ansible-command-status": {
      "required": true,
      "type": "string"
    },
    "ansible-command-logs": {
      "required": true,
      "type": "string"
    }
  },
  "capabilities": {
    "component-node": {
      "type": "tosca.capabilities.Node"
    }
  },
  "interfaces": {
    "ComponentRemoteAnsibleExecutor": {
      "operations": {
        "process": {
          "inputs": {
            "job-template-name": {
              "description": "Primary key or name of the job template to
launch new job.",
              "required": true,
              "type": "string"
            },
            "limit": {
              "description": "Specify host limit for job template to run.",
              "required": false,
              "type": "string"
            },
            "inventory": {
              "description": "Specify inventory for job template to run.",
              "required": false,
              "type": "string"
            },
            "extra-vars" : {
              "required" : false,
              "type" : "json",
              "description": "json formatted text that contains extra
variables to pass on."
            },
            "tags": {
              "description": "Specify tagged actions in the playbook to
run.",
              "required": false,
              "type": "string"
            },
            "skip-tags": {
              "description": "Specify tagged actions in the playbook to
omit.",
              "required": false,
              "type": "string"
            },
            "endpoint-selector": {
              "description": "Remote AWX Server selector name.",
              "required": true,
              "type": "string"
            }
          }
        }
      }
    }
  },
  "derived_from": "tosca.nodes.Component"
}
```

Source

Source

Used to represent a **type of source** to **resolve** a **resource**, along with the expected **properties**

Defines the **contract** to resolve a resource.

Here is the root component TOSCA node type from which other node type will derive:

tosca.nodes.Component

```
{
  "description": "TOSCA base type for Resource Sources",
  "version": "1.0.0",
  "derived_from": "tosca.nodes.Root"
}
```

Bellow is a list of supported sources

input

Input

Expects the **value to be provided as input** to the request.

Here is the definition:

source-input

```
{
  "description": "This is Input Resource Source Node Type",
  "version": "1.0.0",
  "properties": {},
  "derived_from": "tosca.nodes.ResourceSource"
}
```

default

Default

Expects the **value to be defaulted** in the model itself.

Here is the definition:

source-default

```
{
  "description": "This is Default Resource Source Node Type",
  "version": "1.0.0",
  "properties": {},
  "derived_from": "tosca.nodes.ResourceSource"
}
```

rest

REST

Expects the **URI along with the VERB and the payload**, if needed.

CDS is currently deployed along the side of SDNC, hence the **default** rest **connection** provided by the framework is to **SDNC MDSAL**.

Property	Description	Scope
type	Expected output value, only JSON supported for now	Optional
verb	HTTP verb for the request - default value is GET	Optional
payload	Payload to sent	Optional
endpoint-selector	**Specific REST system** to interact with to (see Dynamic Endpoint)	Optional
url-path	URI	Mandatory
path	JSON path to the value to fetch from the response	Mandatory
expression-type	Path expression type - default value is JSON_PATH	Optional

Here is the definition:

source-rest

```
{
  "description": "This is Rest Resource Source Node Type",
  "version": "1.0.0",
  "properties": {
    "type": {
      "required": false,
      "type": "string",
      "default": "JSON",
      "constraints": [
        {
          "valid_values": [
            "JSON"
          ]
        }
      ]
    },
    "verb": {
      "required": false,
      "type": "string",
      "default": "GET",
      "constraints": [
        {
          "valid_values": [
            "GET", "POST", "DELETE", "PUT"
          ]
        }
      ]
    },
    "payload": {
      "required": false,
      "type": "string",
      "default": ""
    },
    "endpoint-selector": {
      "required": false,
      "type": "string"
    },
    "url-path": {
      "required": true,
      "type": "string"
    },
    "path": {
      "required": true,
      "type": "string"
    },
    "expression-type": {
      "required": false,
      "type": "string",
      "default": "JSON_PATH",
```

```
      "constraints": [
        {
          "valid_values": [
            "JSON_PATH",
            "JSON_POINTER"
          ]
        }
      ]
    },
    "input-key-mapping": {
      "required": false,
      "type": "map",
      "entry_schema": {
        "type": "string"
      }
    },
    "output-key-mapping": {
      "required": false,
      "type": "map",
      "entry_schema": {
        "type": "string"
      }
    },
    "key-dependencies": {
      "required": true,
      "type": "list",
      "entry_schema": {
        "type": "string"
      }
    }
  },
  "derived_from": "tosca.nodes.ResourceSource"
}
```

sql

SQL

Expects the **SQL query** to be modeled; that SQL query can be parameterized, and the parameters be other resources resolved through other means. If that's the case, this data dictionary definition will have to define `key-dependencies` along with `input-key-mapping`.

CDS is currently deployed along the side of SDNC, hence the **primary** database **connection** provided by the framework is to **SDNC database**.

Property	Description	Scope
type	Database type, only SQL supported for now	Mandatory
endpoint-selector	**Specific Database system** to interact with to (see Dynamic Endpoint)	Optional
query	Statement to execute	Mandatory

Here is the definition:

source-db

```
{
  "description": "This is Database Resource Source Node Type",
  "version": "1.0.0",
  "properties": {
    "type": {
      "required": true,
      "type": "string",
      "constraints": [
        {
          "valid_values": [
            "SQL"
          ]
        }
      ]
    },
    "endpoint-selector": {
      "required": false,
      "type": "string"
    },
    "query": {
      "required": true,
      "type": "string"
    },
    "input-key-mapping": {
      "required": false,
      "type": "map",
      "entry_schema": {
        "type": "string"
      }
    },
    "output-key-mapping": {
      "required": false,
      "type": "map",
      "entry_schema": {
        "type": "string"
      }
    },
    "key-dependencies": {
      "required": true,
      "type": "list",
      "entry_schema": {
        "type": "string"
      }
    }
  },
  "derived_from": "tosca.nodes.ResourceSource"
}
```

capability

Capability

Expects a **script** to be provided.

Property	Description	Scope
script-type	The type of the script - default value is Koltin	Optional
script-class-reference	The name of the class to use to create an instance of the script	Mandatory

Here is the definition:

source-capability

```
{
  "description": "This is Component Resource Source Node Type",
  "version": "1.0.0",
  "properties": {
    "script-type": {
      "required": true,
      "type": "string",
      "default": "kotlin",
      "constraints": [
        {
          "valid_values": [
                       "internal",
            "kotlin",
            "jython"
          ]
        }
      ]
    },
    "script-class-reference": {
      "description": "Capability reference name for internal and kotlin,
for jython script file path",
      "required": true,
      "type": "string"
    },
    "key-dependencies": {
      "description": "Resource Resolution dependency dictionary names.",
      "required": true,
      "type": "list",
      "entry_schema": {
        "type": "string"
      }
    }
  },
  "derived_from": "tosca.nodes.ResourceSource"
}
```

Other

Other

DG

dg-generic

Identifies a Directed Graph used as **imperative workflow**.

Property	Description	Scope
dependency-node-templates	The node template the workflow depends on	Required

Here is the definition:

dg-generic

```
{
  "description": "This is Generic Directed Graph Type",
  "version": "1.0.0",
  "properties": {
    "content": {
      "required": true,
      "type": "string"
    },
    "dependency-node-templates": {
      "required": true,
      "description": "Dependent Step Components NodeTemplate name.",
      "type": "list",
      "entry_schema": {
        "type": "string"
      }
    }
  },
  "derived_from": "tosca.nodes.DG"
}
```

A node_template of this type always provide one artifact, of type `artifact-directed-graph`, which will be located under the Plans/ folder within the CBA.

node_template example

```
      "config-deploy-process" : {
        "type" : "dg-generic",
        "properties" : {
          "content" : {
            "get_artifact" : [ "SELF", "dg-config-deploy-process" ]
          },
          "dependency-node-templates" : [ "nf-account-collection",
"execute" ]
        },
        "artifacts" : {
          "dg-config-deploy-process" : {
            "type" : "artifact-directed-graph",
            "file" : "Plans/CONFIG_ConfigDeploy.xml"
          }
        }
      }
```

In the DG bellow, the *execute* node refers to the node_template.

Plans/CONFIG_ConfigDeploy.xml

```
<service-logic
  xmlns='http://www.onap.org/sdnc/svclogic'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:schemaLocation='http://www.onap.org/sdnc/svclogic ./svclogic.xsd'
module='CONFIG' version='1.0.0'>
    <method rpc='ConfigDeploy' mode='sync'>
        <block atomic="true">
            <execute plugin="nf-account-collection" method="process">
                <outcome value='failure'>
                    <return status="failure">
                    </return>
                </outcome>
                <outcome value='success'>
                    <execute plugin="execute" method="process">
                        <outcome value='failure'>
                            <return status="failure">
                            </return>
                        </outcome>
                        <outcome value='success'>
                            <return status='success'>
                            </return>
                        </outcome>
                    </execute>
                </outcome>
            </execute>
        </block>
    </method>
</service-logic>
```

VNF

tosca.nodes.VNF

Identifies a VNF, can be used to **correlate** any type of **VNF** related **information**.

https://github.com/onap/ccsdk-cds/blob/master/components/model-catalog/definition-type/starter-type/node_type/tosca.nodes.Vnf.json

tosca.nodes.vnf

```
{
  "description": "This is VNF Node Type",
  "version": "1.0.0",
  "derived_from": "tosca.nodes.Root"
}
```

vnf-netconf-device

Represents the VNF information to **establish** a **NETCONF communication**.

https://github.com/onap/ccsdk-cds/blob/master/components/model-catalog/definition-type/starter-type/node_type/vnf-netconf-device.json

vnf-netconf-device

```
{
  "description": "This is VNF Device with Netconf  Capability",
  "version": "1.0.0",
  "capabilities": {
    "netconf": {
      "type": "tosca.capabilities.Netconf",
      "properties": {
        "login-key": {
          "required": true,
          "type": "string",
          "default": "sdnc"
        },
        "login-account": {
          "required": true,
          "type": "string",
          "default": "sdnc-tacacs"
        },
        "source": {
          "required": false,
          "type": "string",
          "default": "npm"
        },
        "target-ip-address": {
          "required": true,
          "type": "string"
        },
        "port-number": {
          "required": true,
          "type": "integer",
          "default": 830
        },
        "connection-time-out": {
          "required": false,
          "type": "integer",
          "default": 30
        }
      }
    }
  },
  "derived_from": "tosca.nodes.Vnf"
}
```

Workflow

Workflow

Workflow Scope within CDS Framework

The workflow is within the scope of the micro provisioning and configuration management in **co ntroller domain** and does NOT account for the MACRO service orchestration workflow which is covered by the SO Project.

A workflow defines an overall **action** to be taken on the service, hence is an **entry-point** for the **run-time execution** of the CBA package.

A workflow also defines **inputs** and **outputs** that will defined the **payload contract** of the **request** and **re sponse** (see Dynamic API)

A workflow can be **composed** of one or multiple **sub-actions** to execute.

A CBA package can have as **many workflows** as needed.

Single action

The workflow is directly backed by a **`Component`**

In the example bellow, the target of the workflow's steps `resource-assignment` is `resource-assignment` which actually is the name of the `node_template` defined after, of type `component-resource-resolution`.
Link to the example.

Example

```
. . .
  "topology_template": {
    "workflows": {
      "resource-assignment": {
        "steps": {
          "resource-assignment": {
            "description": "Resource Assign Workflow",
            "target": "resource-assignment"
            ]
          }
        },
        "inputs": {
          "resource-assignment-properties": {
            "description": "Dynamic PropertyDefinition for workflow
(resource-assignment).",
            "required": true,
            "type": "dt-resource-assignment-properties"
          }
        },
        "outputs": {
          "meshed-template": {
            "type": "json",
            "value": {
              "get_attribute": [
                "resource-assignment",
                "assignment-params"
              ]
            }
          }
        }
      },
    "node_templates": {
      "resource-assignment": {
        "type": "component-resource-resolution",
        "interfaces": {
          "ResourceResolutionComponent": {
            "operations": {
              "process": {
                "inputs": {
                  "artifact-prefix-names": [
                    "vf-module-1"
                  ]
                }
              }
            }
          }
        },
        "artifacts": {
          "vf-module-1-template": {
            "type": "artifact-template-velocity",
            "file": "Templates/vf-module-1-template.vtl"
          },
          "vf-module-1-mapping": {
            "type": "artifact-mapping-resource",
            "file": "Templates/vf-module-1-mapping.json"
          }
        }
      }
    }
. . .
```

Multiple sub-actions

The workflow is backed by a Directed Graph engine, **`dg-generic`**, and is an **imperative** workflow.

A DG used as workflow for CDS is composed of multiple execute nodes; each individual execute node refers to an modelled **`Component`** instance.

In the example above, you can see the target of the workflow's steps `execute-script` is `execute-remote-ansible-process`, which is a `node_template` of type `dg_generic`

Link of example

workflow plan example

```
. . .
  "topology_template": {
    "workflows": {
      "execute-remote-ansible": {
        "steps": {
          "execute-script": {
            "description": "Execute Remote Ansible Script",
            "target": "execute-remote-ansible-process"
            ]
          }
        },
        "inputs": {
          "ip": {
            "required": false,
            "type": "string"
          },
          "username": {
            "required": false,
            "type": "string"
          },
          "password": {
            "required": false,
            "type": "string"
          },
          "execute-remote-ansible-properties": {
            "description": "Dynamic PropertyDefinition for workflow
(execute-remote-ansible).",
            "required": true,
            "type": "dt-execute-remote-ansible-properties"
          }
        },
        "outputs": {
          "ansible-variable-resolution": {
            "type": "json",
            "value": {
              "get_attribute": [
                "resolve-ansible-vars",
                "assignment-params"
              ]
            }
          },
          "prepare-environment-logs": {
            "type": "string",
            "value": {
              "get_attribute": [
                "execute-remote-ansible",
                "prepare-environment-logs"
              ]
            }
          },
          "execute-command-logs": {
            "type": "string",
            "value": {
              "get_attribute": [
                "execute-remote-ansible",
                "execute-command-logs"
              ]
            }
```

```
        }
      }
    }
  },
  "node_templates": {
    "execute-remote-ansible-process": {
      "type": "dg-generic",
      "properties": {
        "content": {
          "get_artifact": [
            "SELF",
            "dg-execute-remote-ansible-process"
          ]
        },
        "dependency-node-templates": [
          "resolve-ansible-vars",
          "execute-remote-ansible"
        ]
      },
      "artifacts": {
        "dg-execute-remote-ansible-process": {
          "type": "artifact-directed-graph",
          "file": "Plans/CONFIG_ExecAnsiblePlaybook.xml"
        }
      }
    }
```

Properties of a workflow

Property	Description	
workflow-name	Defines the name of the action that can be triggered by external system	
inputs	They are two types of inputs, the dynamic ones, and the static one. **static** Specified at workflow level • can be inputs for the **`Component`**(s), see the *inputs* section of the component of interest. • represent inputs to derived custom logic within scripting These will end up under `${actionName}-request` section of the payload (see Dynamic API) **dynamic** Represent the resources defined as Modeling Concepts#input within mapping definition files. The **enrichment process** will (see Modeling Concepts#Enrichment) • dynamically gather all of them under a data-type, named `dt-${actionName}-properties` • will add it as a input of the workflow, as follow using this name: `${actionName}-properties` Example for workflow named *resource-assignment*: **dynamic input** `"resource-assignment-properties": { "required": true, "type": "dt-resource-assignment-properties" }`	
outputs	Defines the outputs of the execution; there can be as many output as needed. Depending on the **`Component`** of use, some attribute might be retrievable. type: data type (complex / primitive) value: value resolvable using expression	

<table>
<tr><td>steps</td><td>Defines the actual step to execute as part of the workflow<table>
<tr><th>step-name</th><th>description</th><th>target</th></tr>
<tr><td>name of the step</td><td>step description</td><td>a node_template implementing on of the supported Node Type, either a Component or a DG

(see single action or multiple action)</td></tr>
</table></td><td></td></tr>
</table>

Example:

workflow example

```
{
  "workflow": {
    "resource-assignment":
{                                                                    <-
workflow-name
      "inputs": {
        "vnf-id":
{
<- static inputs
          "required": true,
          "type": "string"
        },
        "resource-assignment-properties":
{                                          <- dynamic inputs
          "required": true,
          "type": "dt-resource-assignment-properties"
        }
      },
      "steps": {
        "call-resource-assignment":
{                                                 <- step-name
          "description": "Resource Assignment Workflow",
          "target": "resource-assignment-process"
<- node_template targeted by the step
        }
      },
      "outputs": {
        "template-properties":
{                                                        <- output
          "type":
"json",
<- complex type
          "value": {
            "get_attribute":
[                                                                    <- uses
expression to retrieve attribute from context
              "resource-assignment",
              "assignment-params"
            ]
          }
        }
      }
    }
  }
}
```

TOSCA definition

Template

Template

A template is an **artifact**, and uses Modeling Concepts#artifact-mapping-resource and artifact-template-velocity.

A template is **parameterized** and each parameter must be defined in a corresponding **mapping file**.

In order to know which mapping correlates to which template, the file name must start with an `artifact-prefix`, serving as identifier to the overall template + mapping.

The **requirement** is as follows:

```
${artifact-prefix}-template
${artifact-prefix}-mapping
```

Scripts

Scripts

Library

NetconfClient

In order to facilitate NETCONF interaction within scripts, a python NetconfClient binded to our Kotlin implementation is made available. This NetconfClient can be used when using the `component-netconf-executor`.

The client can be find here: https://github.com/onap/ccsdk-cds/blob/master/components/scripts/python/ccsdk_netconf/netconfclient.py

ResolutionHelper

When executing a component executor script, designer might want to perform resource resolution along with template meshing directly from the script itself.

The helper can be find here: https://github.com/onap/ccsdk-cds/blob/master/components/scripts/python/ccsdk_netconf/common.py

Southbound Interfaces

Southbound Interfaces

CDS comes with native python 3.6 support and Ansible AWX (Ansible Tower): idea is Network Ops are familiar with Python and/or Ansible, and our goal is not to dictate the SBI to use for their operations. Ansible and Python provide already many, and well adopted, SBI libraries, hence they could be utilized as needed.

CDS also provide native support for the following libraries:

- NetConf
- REST
- CLI
- SSH
- gRPC (hence gNMI / gNOI should be supported)

CDS also has extensible REST support, meaning any RESTful interface used for network interaction can be used, such as external VNFM or EMS.

Test

Tests

The **tests** folder contains the **uat.yaml** file for execution the cba actions for sunny day and rainy day scenario using mock data. The process to generate the uat file is documented TBD. The file can be dragged and drop to the Tests folder after the test for all actions are executed.

NOTE: You need to activate the "uat" Spring Boot profile in order to enable the spy/verify endpoints. They are disabled by default because the mocks created at runtime can potentially cause collateral problems in production. You can either pass an option to JVM (`-Dspring.profiles.active=uat`) or set and export an environment variable (`export spring_profiles_active=uat`).

A quick outline of the UAT generation process follows:

1. Create a minimum `uat.yaml` containing only the NB requests to be sent to the BlueprintsProcessor (BPP) service;
2. Submit the blueprint CBA and this draft `uat.yaml` to BPP in a single HTTP POST call:

   ```
   $ curl -u ccsdkapps:ccsdkapps -F cba=@<path to your CBA file> -F uat=@<path to the draft uat.yaml> http://localhost:8080/api/v1/uat/spy
   ```

3. If your environment is properly setup, at the end this service will generate the complete `uat.yaml`;
4. Revise the generate file, eventually removing superfluous message fields;
5. Include this file in your CBA under `Tests/uat.yaml`;
6. Submit the candidate CBA + UAT to be validated by BPP, that now will create runtime mocks to simulate all SB collaborators, by running:

   ```
   $ curl -u ccsdkapps:ccsdkapps -F cba=@<path to your CBA file> http://localhost:8080/api/v1/uat/verify
   ```

7. Once validated, your CBA enhanced with its corresponding UAT is eligible to be integrated into the CDS project, under the folder `components/model-catalog/blueprint-model/uat-blueprints`.

Reference link for sample generated **uat.yaml** file for pnf plug & play use case: uat.yaml file.

As UAT is part of unit testing, it runs in jenkins job ccsdk-cds-master-verify-java whenever a new commit /patch pushed on gerrit in ccsdk/cds repo.