How to write Blueprints and Workflows

What is a Blueprint?

In the object model that the Regional Controller provides a ***Blueprint*** is a formal description (in the form of a YAML file) of all the requirements for how an Akraino Blueprint can be installed and lifecycle managed on an Edgesite (a cluster of nodes). As such, a Blueprint provides:

- a Blueprint schema version (currently always 1.0.0) which identifies the YAML schema being used
- a name and description (for use by User Interfaces)
- a version number for the blueprint
- a description of the hardware that is required in order to install the Blueprint
- one or more *workflows*, which are scripted procedures used by the Regional Controller to create/update/delete the Edgesite. Each workflow in turn may contain:
    - the URL of a workflow script
    - zero or more artifacts which may be pre-fetched by the Regional Controller before executing the workflow. Artifacts may be any URL-identified item that the user wishes to use.
    - An input schema of required data items that the user must provide to the Regional Controller API when starting one of these workflows.

An Example Blueprint

The bare bones of a Blueprint looks like:

```
# The Blueprint schema is required and is always 1.0.0
blueprint: 1.0.0

# The name and version are required, and should describe the Blueprint
name: Hello World Blueprint
version: 1.0.0

# The description is optional
description: This Blueprint demonstrates what is needed in a minimal ARC blueprint.

# The YAML is required and contains the workflow and hardware details
yaml:
  # The hardware_profile section describes what types of hardware this Blueprint may run on
  hardware_profile:
    or:
      - { uuid: c1dfa1ac-53e0-11e9-86c2-c313482f1fdb }
      - { name: 'HPE DL380.*' }

  # The workflow section lists all of the workflows
  workflow:
    # The create workflow is invoked when the POD is created
    create:
      # The Python script that is run to create the POD
      url: http://www.example.org/hello-world/create.py
      # input_schema describes what data is required to be POST-ed when the POD is created
      input_schema:
        name: { type: string }
        region: { type: string }
    # The delete workflow is invoked when the POD is DELETE-ed
    delete:
    # All other workflows are invoked when the POD is updated via PUT
    update_1:
```

How to Write Workflows

The Workflow Names

Workflows are pulled from the `yaml.workflow` stanza of the Blueprint associated with a POD. Every Blueprint should have a workflow named `create` for creating the POD, and one named `delete`, for use when the POD is torn down (deleted).

'update' workflows (which are invoked via a PUT on the POD's URL) may have any name other than `create` or `delete`. They should probably start with `u pdate_`, and there may be several update workflows (for different types of updates). Workflow names must be alphanumeric (_ is also allowed) and may not be longer than 36 characters long.

The Workflow Execution Procedure

When the RC prepares to start a workflow, it performs the following steps:

1. Checks for the stanza in the Blueprint associated with the POD for a workflow of the type requested. For example, when first creating a POD, the `create` workflow is used, and so, the Blueprint is checked for a `yaml.workflow.create` stanza.
2. It then builds a workspace directory for the workflow to operate in. This directory will contain all scripts, components, and other files needed by the workflow engine to perform its job. The directory is shared between the instances of Airflow and the API server.
3. It then attempts to pull in the workflow script specified in the Blueprint, as well as any required component files (also specified in the Blueprint). These are placed into the workspace directory.
4. It then creates a parent workflow from an internal template. This workflow consists of three tasks that are executed in succession.

    a. A preamble task that sets things up by appending the workspace directory to `sys.path` (Python) or `$PATH` (bash), and importing the user's workflow file.
    b. The main task that just invokes the Blueprints' workflow.
    c. A postamble task that cleans the workspace and performs some final bookkeeping (marks the POD as being ACTIVE. Note that if the the main task is a long running one, or never finishes, then the postamble may not run. However, the postamble must run before any other workflows may be scheduled for a particular POD.
5. The parent workflow file is then moved into the Airship `dags` directory in order to cause Airship to register and start the workflow. While the workflow is running, no other operations may be performed upon the POD. `Note:` It can take Airflow up to 5 minutes before it notices that a new DAG has been placed in the dags directory; as a result, there is usually a lag between when a workflow is started and the workflow actually executes inside Airship.

Workflows may be either Python scripts or shell (bash) scripts. Python is recommended since it provides a bit more flexibility when it comes to working with DAGs and YAML files. The filename at the end of the URL specifying the workflow determines both the filename of the workflow file, and how it will be invoked. If it ends in `.py`, it is assumed to be a Python script; if it ends with `.sh` it is assumed to be a bash script. Any other ending is invalid. If the workflow is to have multiple tasks in a sub-DAG, then the script **must** be a Python script written according to the rules of Airflow.

Blueprint workflows may invoke other shell scripts or Python scripts. If specified in the Blueprint in the `components` section, then the RC will automatically download them into the workspace. Otherwise, it is up to the workflow itself to fetch any required files. This includes, for example, any Docker containers which the workflow may start.

Python Execution Environment

Workflows written in Python may assume the following is available:

- `sys.path` set to include the user's workspace directory
- the *PyYAML* package (https://pypi.org/project/PyYAML/) is available for import (to help with parsing YAML files)
- the *docker* package (https://docker-py.readthedocs.io/en/stable/) is available for import (to help with interacting with the Docker system running on the RC)
- all of the Airflow packages are available
- the RC will create a `POD.py` file in the workspace which will contain all the data (written as Python variables) describing the Blueprint, Edgesite, and user data to be used for the workflow. (BLUEPRINT, EDGESITE, USER_DATA, POD_UUID, NODES[])
- the RC will also create a `INPUT.yaml` file in the workspace which will contain all the data (in YAML form) that was provided to the RC in the POST or PUT request. This data should match the input_schema in the Blueprint, with optional extra values not specified in the input_schema.

The version of Python used inside Airflow is 3.6.

The Python script itself will be started via the main task (see above). It will call the `start` function in the Python workflow file which is required to have the following signature:

```
def start(ds, **kwargs):
```

Bash Execution Environment

Workflows written in bash may assume the following is available:

- the *ssh* command
- the *docker* command
- the RC will create a `POD.sh` file in the workspace which will contain all the data (written as bash environment variables) describing the Blueprint, Edgesite, and user data to be used for the workflow.

Note: any Docker containers run by the workflow should be run with the `--rm` option, in order to be automatically cleaned up on exit. In addition, these containers will run parallel to the Airflow containers in the RC. A such, they should use the `--net=arc-net` option, if they need to share the same Docker network with the rest of the RC.

The version of bash used inside Airflow is 4.4.12.