Croupier Documentation

Release 2.1.2

Javier Carnero

Feb 16, 2022

Contents:

CHAPTER 1

Modelling

Application run by Cloudify/Croupier must be defined in a TOSCA file(s) - *The Blueprint*. A blueprint is typically composed by a *header*, an *inputs* section, a *node_templates* section, and an outputs section. Optinally can have a *node_types* section.

Tip

Example blueprints can be found at the Croupier resources repository.

1.1 Header

The header include the TOSCA version used and other imports. In Croupier the Cloudify 1.1.3 tosca version, built-in types and the croupier are mandatory:

```
tosca_definitions_version: cloudify_dsl_1_3

imports:
   # to speed things up, it is possible to download this file,
   - http://raw.githubusercontent.com/ari-apc-lab/croupier/master/resources/types/cfy_
↪types.yaml
   # Croupier pluging
   - http://raw.githubusercontent.com/ari-apc-lab/croupier/master/plugin.yaml
   # Openstack plugin (Optional)
   - http://www.getcloudify.org/spec/openstack-plugin/2.14.7/plugin.yaml
   # The blueprint can be composed by multiple files, in this case we split the␣
↪inputs section (Optional)
   - inputs-def.yaml
```

Other TOSCA files can also be imported in the `inports` list to compose a blueprint made of more than one file. See *Advanced: Node Types* for more info.

1.2 Inputs

In this section is where it is defined all the inputs that the blueprint need. These then can be passed as an argument list in the CLI, or prefereably by an inputs file. An input can define a default value. (See the CLI docs and the files *inputs-def* and *local-blueprint-inputs-example.yaml* in the examples).

```
inputs:
   hpc_base_dir:
      description: HPC working directory
      default: $HOME

   partition_name:
      default: thinnodes
```

In the example above, two inputs are defined:

- `hpc_base_dir` as the base working directory, $HOME by default.
- `partition_name` as the partition to be used in an HPC, _thinnodes_ by default.

1.3 Node Templates

In the `node_templates` section is where your application is actually defined, by stablishing *nodes* and *relations* between them.

To begin with, every *node* is identified by its name (`hpc_interface` in the example below), and a type is assigned to it.

Infrastructure Interface example.

```
node_templates:
    hpc_interface:
    type: croupier.nodes.InfrastructureInterface
    properties:
        config: { get_input: hpc_interface_config }
        credentials: { get_input: hpc_interface_credentials }
        external_monitor_entrypoint: { get_input: monitor_entrypoint }
        job_prefix: { get_input: job_prefix }
        base_dir: { get_input: "hpc_base_dir" }
        monitor_period: 15
        workdir_prefix: "single"
```

The example above represents a infrastructure interface, with type *croupier.nodes.InfrastructureInterface*. All computing infrastructures must have a infrastructure interface defined (_Slurm_ or _Torque_ for HPC supported, plain _SHELL_ for Cloud VMs). Then the WM is configured using the inputs (using fuction *get_input*). Detailed information about how to configure the HPCs is in the Plugin specification section.

The following code uses `hpc_interface` to describe four jobs that should run in the hpc that represents the node. Two of them are of type `croupier.nodes.SingularityJob` which means that the job will run using a Singularity container, while the other two of type *croupier.nodes.Job* describe jobs that are going to run directly in the HPC. Navigate to Croupier plugin types to know more about each parameter.

Four jobs example.

```
first_job:
    type: croupier.nodes.Job
```

(continues on next page)

(continued from previous page)

```
    properties:
        job_options:
            partition: { get_input: partition_name }
            commands: ["touch fourth_example_1.test"]
            nodes: 1
            tasks: 1
            tasks_per_node: 1
            max_time: "00:01:00"
        skip_cleanup: True
    relationships:
        - type: task_managed_by_interface
          target: hpc_interface

second_parallel_job:
    type: croupier.nodes.Job
    properties:
        job_options:
            partition: { get_input: partition_name }
            commands: ["touch fourth_example_2.test"]
            nodes: 1
            tasks: 1
            tasks_per_node: 1
            max_time: "00:01:00"
        skip_cleanup: True
    relationships:
        - type: task_managed_by_interface
          target: hpc_interface
        - type: job_depends_on
          target: first_job

third_parallel_job:
    type: croupier.nodes.Job
    properties:
        job_options:
            script: "touch.script"
            arguments:
                - "fourth_example_3.test"
            nodes: 1
            tasks: 1
            tasks_per_node: 1
            max_time: "00:01:00"
            partition: { get_input: partition_name }
        deployment:
            bootstrap: "scripts/create_script.sh"
            revert: "scripts/delete_script.sh"
            inputs:
                - "script_"
        skip_cleanup: True
    relationships:
        - type: task_managed_by_interface
          target: hpc_interface
        - type: job_depends_on
          target: first_job

fourth_job:
    type: croupier.nodes.Job
    properties:
```

(continues on next page)

(continued from previous page)

```
        job_options:
            script: "touch.script"
            arguments:
                - "fourth_example_4.test"
            nodes: 1
            tasks: 1
            tasks_per_node: 1
            max_time: "00:01:00"
            partition: { get_input: partition_name }
        deployment:
            bootstrap: "scripts/create_script.sh"
            revert: "scripts/delete_script.sh"
            inputs:
                - "script_"
        skip_cleanup: True
    relationships:
        - type: task_managed_by_interface
          target: hpc_interface
        - type: job_depends_on
          target: second_parallel_job
        - type: job_depends_on
          target: third_parallel_job
```

Finally, jobs have two main types of relationships: **task_managed_by_interface**, to stablish which infrastructure interface will run the job, and **job_depends_on**, to describe the dependency between jobs. In the example above, *fourth_job* depends on *three_parallel_job* and *second_parallel_job*, so it will not execute until the other two have finished. In the same way, *three_parallel_job* and *second_parallel_job* depends on *first_job*, so they will run in parallel once the first job is finished. All jobs are contained in *hpc_interface*, so they will run on the HPC using the credentials provided. A third one, **interface_contained_in** is used to link the Infrastructure Interface to other Cloudify plugins, sush as Openstack. See relationships for more information.

1.4 Outputs

The last section, `outputs`, helps to publish different attributes of each *node* that can be retrieved after the install workflow of the blueprint has finished (See *Execution*).

Each output has a name, a description, and value.

outputs:

first_job_name: description: first job name value: { get_attribute: [first_job, job_name] }

second_job_name: description: second job name value: { get_attribute: [second_parallel_job, job_name] }

third_job_name: description: third job name value: { get_attribute: [third_parallel_job, job_name] }

fourth_job_name: description: fourth job name value: { get_attribute: [fourth_job, job_name] }

1.5 Advanced: Node Types

Similarly to how *node_templates* are defined, new node types can be defined to be used as types. Usually these types are going to be defined in a separate file and imported in the blueprint through the *import* keyword in the *header* section, although they can be in the same file.

Framework example.

```
node_types:
    croupier.nodes.fenics_iter:
        derived_from: croupier.nodes.Job
        properties:
            iter_number:
                description: Iteration index (two digits string)
            job_options:
                default:
                    pre:
                        - 'module load gcc/5.3.0'
                        - 'module load impi'
                        - 'module load petsc'
                        - 'module load parmetis'
                        - 'module load zlib'
                    script: "$HOME/wing_minimal/fenics-hpc_hpfem/unicorn-minimal/
↪nautilus/fenics_iter.script"
                    arguments:
                        - { get_property: [SELF, iter_number] }

    croupier.nodes.fenics_post:
        derived_from: croupier.nodes.Job
        properties:
            iter_number:
                description: Iteration index (two digits string)
            file:
                description: Input file for dolfin-post postprocessing
            job_options:
                default:
                    pre:
                        - 'module load gcc/5.3.0'
                        - 'module load impi'
                        - 'module load petsc'
                        - 'module load parmetis'
                        - 'module load zlib'
                    script: "$HOME/wing_minimal/fenics-hpc_hpfem/unicorn-minimal/
↪nautilus/post.script"
                    arguments:
                        - { get_property: [SELF, iter_number] }
```

Above there is dummy example of two new types of the FEniCS framework, derived from `croupier.nodes.Job`.

The first type, `croupier.nodes.fenics_iter`, defines an iteration of the FEniCS framework. A new property has been defined, `iter_number`, with a description and no default value (so it is mandatory). Besides the `job_options` property default value has been overriden with a concrete list of modules, script and arguments.

The second type, `croupier.nodes.fenics_post`, described a simulated postprocessing operation of FEniCS, defining again the `iter_number` property and another one `file`. Finally the job options default value has been overriden with a list of modules, script and arguments.

> **Note**
>
> The arguments reference the built-in function `get_property`. This allows the orchestrator to compose the arguments based on other properties. To see all the functions available, check the Cloudify intrinsic functions.

1.6 Execution

Execution of an application is performed through the CLI docs in your local machine or a host of your own.

1.7 Steps

1. **Upload the blueprint**

   Before doing anything, the blueprint we want to execute needs to be uploaded in the orchestrator with an assigned name.

   ```
   cfy blueprints upload -b [BLUEPRINT-NAME] [BLUEPRINT-FILE].yaml
   ```

2. **Create a deployment**

   Once we have a blueprint installed, we create a *deployment*, which is a blueprint with an input file attached. This is usefull to have the same blueprint that represents the application, with different configurations (*deployments*). A name has to be assigned to it as well.

   ```
   cfy deployments create -b [BLUEPRINT-NAME] -i [INPUTS-FILE].yaml
   --skip-plugins-validation [DEPLOYMENT-NAME]
   ```

   Note

   `--skip-plugins-validation` is mandatory as we want that the orchestrator download the plugin from a source location (GitHub in our case). This is for testing purposes, and will be removed in future releases.

3. **Install a deployment**

   Install workflow puts everything in place to run the application. Usual tasks in this workflow are data movements, binary downloads, HPC configuration, etc.

   ```
   cfy executions start -d [DEPLOYMENT-NAME] install
   ```

4. **Run the application**

   Finally to start the execution we run the `run_jobs` workflow to start sending jobs to the different infrastructures. The execution can be followed in the output.

   ```
   cfy executions start -d [DEPLOYMENT-NAME] run_jobs
   ```

   Note

   The CLI has a timeout of 900 seconds, which normally is not enough time for an application to finish. However, if the CLI timeout, the execution will still be running on the MSOOrchestrator. To follow the execution just follow the instructions in the output.

1.7.1 Revert previous Steps

The following revert the steps above, in order to uninstall the application, recreate the deployment with new inputs, or remove the blueprint (and possibly upload an updated one), follow the following steps.

1. **Uninstall a deployment**

   On the contraty of the *install* workflow, in this case the orchestrator is tipically goint to perform the revert operation of *install*, by deleting execution files or moving data to an external location.

   ```
   cfy executions start -d [DEPLOYMENT-NAME] uninstall -p ignore_failure=true
   ```

Note

The `ignore_failure` parameter is optional, to perform the *uninstall* even if an error occurs.

2. **Remove a deployment**

   ```
   cfy deployments delete [DEPLOYMENT-NAME]
   ```

3. **Remove a blueprint**

   ```
   cfy blueprints delete [BLUEPRINT-NAME]
   ```

1.7.2 Troubleshooting

If an error occurs the revert steps can be followed revert the last steps made. However there are sometimes when the execution is stucked, or you want simply to cancel a runnin execution, or clear a blueprint or deployment that can be uninstall for whatever the reason. The following commands help you resolve these kind of situations.

1. **See executions list and status**

   ```
   cfy executions list
   ```

2. **Check one execution status**

   ```
   cfy executions get [EXECUTION-ID]
   ```

3. **Cancel a running (started) execution**

   ```
   cfy executions cancel [EXECUTION-ID]
   ```

4. **Hard remove a deployment with all its executions and living nodes**

   ```
   cfy deployments delete [DEPLOYMENT-NAME] -f
   ```

CHAPTER 2

Installation

Croupier, as a Cloudify plugin, must to be run inside the Cloudify Server, a.k.a Cloudify Manager. This section describes how to install Cloudify Manager, with the Croupier plugin.

2.1 Cloudify Manager

2.1.1 Docker

Cloudify provides a docker image of the manager. It cannot be configured so, among other things, it is not secure (user admin/admin).

```
sudo docker run --name cfy_manager -d --restart unless-stopped \
  -v /sys/fs/cgroup:/sys/fs/cgroup:ro \
  --tmpfs /run \
  --tmpfs /run/lock \
  --security-opt seccomp:unconfined \
  --cap-add SYS_ADMIN \
  --network host \
  cloudifyplatform/community:19.01.24

docker run --name cfy_manager -d --restart unless-stopped \
  -v /sys/fs/cgroup:/sys/fs/cgroup:ro \
  --tmpfs /run --tmpfs /run/lock \
  --security-opt seccomp:unconfined \
  --cap-add SYS_ADMIN \
  -p 80:80 \
  -p 8000:8000 \
  cloudifyplatform/community-cloudify-manager-aio
```

OpenStack plugin (Optional)

```
cfy plugins upload \
  -y http://www.getcloudify.org/spec/openstack-plugin/2.14.7/plugin.yaml \
  http://repository.cloudifysource.org/cloudify/wagons/cloudify-openstack-plugin/2.14.
↪7/cloudify_openstack_plugin-2.14.7-py27-none-linux_x86_64-centos-Core.wgn
```

2.1.2 Requirements

Cloudify Manager is supported for installation on a 64-bit host with RHEL/CentOS 7.4.

	Minimum	Recommended
vCPUs	2	8
RAM	4GB	16GB
Storage	5GB	64GB

The minimum requirements are enough for small deployments that only manage a few compute instances. Managers that manage more deployments or large deployments need at least the recommended resources.

Check Cloudify docs for full prerequisites details.

2.2 Croupier Plugin

TODO

CHAPTER 3

Croupier Cloudify plugin

3.1 Requirements

- **Python version**
  - 2.7.x

3.2 Compatibility

- Slurm based HPC by ssh user & key/password.
- Moab/Torque based HPC by ssh user & key/password.
- Tested with Openstack plugin.

  Tip

  Example blueprints can be found at the Croupier resources repository.

3.3 Configuration

The Croupier plugin requires credentials, endpoint and other setup information in order to authenticate and interact with the computing infrastructures.

This configuration properties are defined in *credentials* and *config* properties.

```
credentials:
  host: "[HPC-HOST]"
  user: "[HPC-SSH-USER]"
  private_key: |
    ----BEGIN RSA PRIVATE KEY----
    ......
```

(continues on next page)

(continued from previous page)

```
    -----END RSA PRIVATE KEY-----
  private_key_password: "[PRIVATE-KEY-PASSWORD]"
  password: "[HPC-SSH-PASS]"
  login_shell: {true|false}
  tunnel:
      host: ...
      ...
```

1. HPC and ssh credentials. At least `private_key` or `password` must be provided.

   a. *tunnel*: Follows the same structure as its parent (credentials), to connect to the infrastructure through a tunneled SSH connection.

   b. *login_shell*: Some systems may require to connect to them using a login shell. Default `false`.

```
config:
  country_tz: "Europe/Madrid"
  infrastructure_interface: {SLURM|TORQUE|SHELL}
```

1. *country_tz*: Country Time Zone configured in the the HPC.
2. *infrastructure_interface*: Infrastructure Interface used by the HPC.

   Warning

   Only Slurm and Torque are currently accepted as infrastructure interfaces for HPC. For cloud providers, SHELL is used as interface.

3.4 Types

This section describes the node type definitions. Nodes describe resources in your HPC infrastructures. For more information, see node type.

3.5 croupier.nodes.InfrastructureInterface

Derived From: cloudify.nodes.Compute

Use this type to describe the interface of a computing infrastructure (HPC or VM)

Properties:

- `config`: type of interface and system time zone, as described in *config*.
- `credentials`: Access credentials, as described in *credentials*.
- `base_dir`: Root directory of the working directory. Default `$HOME`.
- `workdir_prefix`: Prefix name of the working directory that will be created by this interface.
- `job_prefix`: Job name prefix for the jobs created by this interface. Default `cfyhpc`.
- `monitor_period`: Seconds to check job status. This is necessary because infrastructure interfaces can be overloaded if asked too much times in a short period of time. Default `60`.
- `skip_cleanup`: True to not clean all files when destroying the deployment. Default `False`.
- `simulate`: If true, it performs a dry run where jobs are not really executed and simulate that they finish inmediately. Useful for testing. Default `False`.

- external_monitor_entrypoint: Entrypoint of the external monitor that Cloudify will use instead of the internal one.
- external_monitor_type: Type of the monitoring system when using an external one. Default {uri-prometheus}[PROMETHEUS].
- external_monitor_port: Port of the monitor when using an external monitoring system. Default :9090.
- external_monitor_orchestrator_port: Port of the external monitor to connect with Croupier. Default :8079.

Example

This example demonstrates how to describe a SLURM interface on an HPC.

```
hpc_interface:
  type: croupier.nodes.InfrastructureInterface
  properties:
    credentials:
      host: "[HPC-HOST]"
      user: "[HPC-SSH-USER]"
      password: "[HPC-SSH-PASS]"
    config:
      country_tz: "Europe/Madrid"
      infrastructure_interface: "SLURM"
    job_prefix: crp
    workdir_prefix: test
  ...
```

Mapped Operations:

- cloudify.interfaces.lifecycle.configure Checks that there is a connection between Cloudify and the infrastructure interface, and creates a new working directory.
- cloudify.interfaces.lifecycle.delete Clean up all data generated by the execution.
- cloudify.interfaces.monitoring.start If the external monitor orchestrator is available, sends a notification to start monitoring the infrastructure.
- cloudify.interfaces.monitoring.stop If the external monitor orchestrator is available, sends a notification to end monitoring the infrastructure.

3.6 croupier.nodes.Job

Use this type to describe a job (a task that will execute on the infrastructure).

Properties:

- job_options: Job parameters and needed resources.
  - pre: List of commands to be executed before running the job. Optional.
  - post: List of commands to be executed after running the job. Optional.
  - partition: Partition in which the job will be executed. If not provided, the HPC default will be used.
  - commands: List of commands to be executed. Mandatory if *script* property is not present.
  - script: Script to be executed. Mandatory if *commands* property is not present.
  - arguments: List of arguments to be passed to execution command. Variables must be scaped like *"\$USER"*

  - `nodes`: Nodes to use in job. Default `1`.
  - `tasks`: Number of tasks of the job. Default `1`.
  - `tasks_per_node`: Number of tasks per node. Default `1`.
  - `max_time`: Set a limit on the total run time of the job allocation. Mandatory if no script is provided, or if the script does not define such property.
  - `scale`: Execute in parallel the job N times according to this property. Only for HPC. Default `1` (no scale).
  - `scale_max_in_parallel`: Maximum number of scaled job instances that can be run in parallel. Only works with scale > `1`. Default same as scale.
  - `memory`: Specify the real memory required per node. Different units can be specified using the suffix [`K|M|G|T`]. Default value `""` lets the infrastructure interface assign the default memory to the job.
  - `stdout_file`: Define the file where to gather the standard output of the job. Default value `""` sets `<job-name>.err` filename.
  - `stderr_file`: Define the file where to gather the standard error output. Default value `""` sets `<job-name>.out` filename.
  - `mail-user`: Email to receive notification of job state changes. Default value `""` does not send any mail.
  - `mail-type`: Type of event to be notified by mail, can define several events separated by comma. Valid values `NONE, BEGIN, END, FAIL, TIME_LIMIT, REQUEUE, ALL`. Default value `""` does not send any mail.
  - `reservation`: Allocate resources for the job from the named reservation. Default value `""` does not allocate from any named reservation.
  - `qos`: Request a quality of service for the job. Default value `""` lets de infrastructure interface assign the default user `qos`.

- `deployment`: Scripts to perform deployment operations. Optional.
  - `bootstrap`: Relative path to blueprint to the script that will be executed in the HPC at the install workflow to bootstrap the job (like data movements, binary download, etc.)
  - `revert`: Relative path to blueprint to the script that will be executed in the HPC at the uninstall workflow, reverting the bootstrap or other clean up operations.
  - `inputs`: List of inputs that will be passed to the scripts when executed in the HPC.
- `publish`: A list of outputs to be published after job execution. Each list item is a dictionary containing:
  - `type`: Type of the external repository to be published. Only `CKAN` is supported for now. The rest of the parameters depends on the type.
  - `type: CKAN`
    * `entrypoint`: ckan entrypoint
    * `api_key`: Individual user ckan api key.
    * `dataset`: Id of the dataset in which the file will be published.
    * `file_path`: Local path of the output file in the computation node.
    * `name`: Name used to publish the file in the repository.
    * `description`: Text describing the data file.
- `skip_cleanup`: Set to true to not clean up orchestrator auxiliar files. Default `False`.

Note

The variable $CURRENT_WORKDIR is available in all operations and scripts. It points to the working directory of the execution in the HPC from the *HOME* directory: `/home/user/$CURRENT_WORKDIR/`.

Note

The variables `$SCALE_INDEX`, `$SCALE_COUNT` and `$SCALE_MAX` are available in all commands and inside the scripts where `# DYNAMIC VARIABLES` exist (they will be dynamicaly loaded after this line). They hold, for each job instance, the index, the total number of instances, and the maximun in parallel respectively.

Example

This example demonstrates how to describe a job.

```
hpc_job:
  type: croupier.nodes.Job
  properties:
    job_options:
      partition: { get_input: partition_name }
      commands: ["touch job-$SCALE_INDEX.test"]
      nodes: 1
      tasks: 1
      tasks_per_node: 1
      max_time: "00:01:00"
      scale: 4
    skip_cleanup: True
  relationships:
  - type: task_managed_by_interface
    target: hpc_interface
 ...
```

This example demonstrates how to describe an script job.

```
hpc_job:
  type: croupier.nodes.Job
  properties:
    job_options:
      script: "touch.script"
      arguments:
          - "job-\\$SCALE_INDEX.test"
      nodes: 1
      tasks: 1
      tasks_per_node: 1
      max_time: "00:01:00"
      partition: { get_input: partition_name }
      scale: 4
    deployment:
      bootstrap: "scripts/create_script.sh"
      revert: "scripts/delete_script.sh"
      inputs:
        - "script-"
    skip_cleanup: True
  relationships:
    - type: task_managed_by_interface
      target: hpc_interface
 ...
```

Mapped Operations:

- `cloudify.interfaces.lifecycle.start` Send and execute the bootstrap script.
- `cloudify.interfaces.lifecycle.stop` Send and execute the revert script.

- `croupier.interfaces.lifecycle.queue` Queues the job in the HPC.
- `croupier.interfaces.lifecycle.publish` Publish outputs outside the HPC.
- `croupier.interfaces.lifecycle.cleanup` Clean up operations after job is finished.
- `croupier.interfaces.lifecycle.cancel` Cancels a queued job.

3.7 croupier.nodes.SingularityJob

Derived From: *croupier_nodes_job*

Use this tipe to describe a job executed from a Singularity container.

Properties:

- `job_options`: Job parameters and needed resources.
  - `pre`: List of commands to be executed before running singularity container. Optional.
  - `post`: List of commands to be executed after running singularity container. Optional.
  - `image`: Singularity image file.
  - `home`: Home volume that will be bind with the image instance (Optional).
  - `volumes`: List of volumes that will be bind with the image instance.
  - `partition`: Partition in which the job will be executed. If not provided, the HPC default will be used.
  - `nodes`: Necessary nodes of the job. 1 by default.
  - `tasks`: Number of tasks of the job. 1 by default.
  - `tasks_per_node`: Number of tasks per node. 1 by default.
  - `max_time`: Set a limit on the total run time of the job allocation. Mandatory if no script is provided.
  - `scale`: Execute in parallel the job N times according to this property. Default `1` (no scale).
  - `scale_max_in_parallel`: Maximum number of scaled job instances that can be run in parallel. Only works with scale > `1`. Default same as scale.
  - `memory`: Specify the real memory required per node. Different units can be specified using the suffix [`K|M|G|T`]. Default value `""` lets the infrastructure interface assign the default memory to the job.
  - `stdout_file`: Define the file where to gather the standard output of the job. Default value `""` sets `<job-name>.err` filename.
  - `stderr_file`: Define the file where to gather the standard error output. Default value `""` sets `<job-name>.out` filename.
  - `mail-user`: Email to receive notification of job state changes. Default value `""` does not send any mail.
  - `mail-type`: Type of event to be notified by mail, can define several events separated by comma. Valid values `NONE, BEGIN, END, FAIL, TIME_LIMIT, REQUEUE, ALL`. Default value `""` does not send any mail.
  - `reservation`: Allocate resources for the job from the named reservation. Default value `""` does not allocate from any named reservation.
  - `qos`: Request a quality of service for the job. Default value `""` lets de infrastructure interface assign the default user `qos`.
- `deployment`: Optional scripts to perform deployment operations (bootstrap and revert).

  - `bootstrap`: Relative path to blueprint to the script that will be executed in the HPC at the install workflow to bootstrap the job (like image download, data movements, etc.)
  - `revert`: Relative path to blueprint to the script that will be executed in the HPC at the uninstall workflow, reverting the bootstrap or other clean up operations (like removing the image).
  - `inputs`: List of inputs that will be passed to the scripts when executed in the HPC

- `skip_cleanup`: Set to true to not clean up orchestrator auxiliar files. Default `False`.

  Note

  The variable $CURRENT_WORKDIR is available in all operations and scripts. It points to the working directory of the execution in the HPC from the *HOME* directory: `/home/user/$CURRENT_WORKDIR/`.

  Note

  The variables $SCALE_INDEX, $SCALE_COUNT and $SCALE_MAX are available when scaling, holding for each job instance the index, the total number of instances, and the maximun in parallel respectively.

Example

This example demonstrates how to describe a new job executed in a Singularity container.

```
singularity_job:
  type: croupier.nodes.SingularityJob
    properties:
    job_options:
      pre:
      - { get_input: mpi_load_command }
      - { get_input: singularity_load_command }
      partition: { get_input: partition_name }
      image: {
          concat:
              [
                  { get_input: singularity_image_storage },
                  "/",
                  { get_input: singularity_image_filename },
              ],
      }
      volumes:
      - { get_input: scratch_voulume_mount_point }
      - { get_input: singularity_mount_point }
      commands: ["touch singularity.test"]
      nodes: 1
      tasks: 1
      tasks_per_node: 1
      max_time: "00:01:00"
    deployment:
        bootstrap: "scripts/singularity_bootstrap_example.sh"
        revert: "scripts/singularity_revert_example.sh"
        inputs:
        - { get_input: singularity_image_storage }
        - { get_input: singularity_image_filename }
        - { get_input: singularity_image_uri }
        - { get_input: singularity_load_command }
    skip_cleanup: True
  relationships:
      - type: task_managed_by_interface
        target: hpc_interface
 ...
```

Mapped Operations:

- `cloudify.interfaces.lifecycle.start` Send and execute the bootstrap script.
- `cloudify.interfaces.lifecycle.stop` Send and execute the revert script.
- `croupier.interfaces.lifecycle.queue` Queues the job in the HPC.
- `croupier.interfaces.lifecycle.publish` Publish outputs outside the HPC.
- `croupier.interfaces.lifecycle.cleanup` Clean up operations after job is finished.
- `croupier.interfaces.lifecycle.cancel` Cancels a queued job.

3.7.1 Relationships

See the relationships section.

The following plugin relationship operations are defined in the HPC plugin:

- `task_managed_by_interface` Sets a *croupier_nodes_job* to be executed by interface *croupier_nodes_interface*.
- `job_depends_on` Sets a *croupier_nodes_job* as a dependency of the target (another *croupier_nodes_job*), so the target job needs to finish before the source can start.
- `interface_contained_in` Sets a *croupier_nodes_interface* to be contained in the specific target (a computing node).

3.7.2 Tests

To run the tests Cloudify CLI has to be installed locally. Example blueprints can be found at *tests/blueprint* folder and have the `simulate` option active by default. Blueprint to be tested can be changed at *workflows_tests.py* in the *tests* folder.

To run the tests against a real HPC / Monitor system, copy the file *blueprint-inputs.yaml* to *local-blueprint-inputs.yaml* and edit with your credentials. Then edit the blueprint commenting the simulate option, and other parameters as you wish (e.g change the name ft2_node for your own hpc name). To use the openstack integration, your private key must be put in the folder *inputs/keys*.

Note

dev-requirements.txt needs to be installed (*windev-requirements.txt* for windows):

```
pip install -r dev-requirements.txt
```

To run the tests, run tox on the root folder

```
tox -e flake8,unit,integration
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search