Workloads (WC)

Introduction

For information on how to access the API and API authorization see: .Obtaining API Keys

The workload API provides an API by which users can create, edit and manage workloads, and activate and deactivate them to create (allocate) or delete (de-allocate) sets of cloud resources. It also provides methods for monitoring and controlling the activation and deactivation process.

All workloads are described by means of a JSON document. This document contains workload elements, which can be added and deleted from the workload. Each workload has a name, which must be unique for the user.

Note: Names are made unique by lower casing and removing all spaces. So “VM 1” will be made unique as “vm1” which means “Vm 1” and “vM1” will be considered the same.

The workload CRUD API methods are used for editing the workload, including adding and deleting workload elements.

Once the workload is defined using the CRUD methods, it can be activated. Activation (or deactivation) is a two step process. First, the workload is planned. The planning process checks that when this workload is activated, the user will not exceed their limits for resource instances. It also checks the availability of resources at the provider, which will indicate the likelihood that this activation process will succeed or not. The output of the planning process is the workload plan, which is a set of workload steps that will need to be carried out to activate the workload.

- There are two types of plan. A plan for activation, and a plan for deactivation.
- The plan is transient, because it depends on the current state of the users instances.
- The plan is not stored in the database, but it is returned as part of the workload.
- The plan is valid for a limited time, it will expire.
- The plan will contain both serial and parallel steps, which indicate the sequence in which the steps will be executed.

The planning process considers the state of the “running workload”. The “running workload” is the set of instances that are running that were created for this workload. These instances are from the instance service and are tagged with the workloadId from the current workload in their metadata (a collection). To access these instances (and their status), you use the instance API, not the workload API.

During the planning process, the workload definition is checked against the running workload. For activation, if a workload element is defined in the workload definition that is not in the running workload, then the workload plan will create that instance. If a workload element was deleted from the workload definition, but is in the running workload, then the workload plan will delete that instance. So the activation plan will always try to get the running workload to match the workload definition. For deactivation, the running workload is checked and the workload plan deletes all elements in the running workload.

If the plan looks good, you can then go ahead and execute the workload.

The steps for workload activation are:

1. Create any key pairs.
2. Create any security groups.
3. Create requested VMs.
4. Create requested VS.
5. Attach any VS to the VM’s.

A transaction is the process of activating or deactivating the workload. Once that activation or deactivation has occurred, there is no longer a transaction. The transaction is only used for monitoring the activation or deactivation process and nothing else. We keep the last transaction available for each workload, but only the last transaction, and once the activation or deactivation process (the transaction) has completed (or failed) it is no longer useful - except for knowing what happened.

In general, any errors will be returned either directly as JSON from the REST call (code, message, ticket) or in some cases error information might be embedded into the workload JSON itself.

Once created, workloads have a unique workloadId (Guid), but workload elements are identified only by name within the scope of a workload, they do not have an id. Each workload element name must be unique within the scope of the workload.

To see an example of the api in action see: Getting Started with Workloads.

Workload Methods

Create workload

Create a workload.

POST /workload

Request Body

The workload JSON.

Notes

The workload JSON is validated against a basic schema defined here: Workload Schema.

The schema includes: name, description, metadata, parameters and elements (an array). Metadata here is metadata for the workload and can be any valid JSON.

Each workload element is validated against a basic schema defined here: Workload Element Schema.

For the schema validation, most properties are optional, but if they are provided, they must match the schema. Properties in the JSON which are not defined in the schema are ignored.

For create, the workload name is required. Must be unique for this user.

A user is limited in the maximum number of workloads they can have at one time. This is defined by the “user limits” for that user.

Example

POST /workload

Returns

The workload JSON. Will include the workloadId.

Clone workload

Clone the workload. This creates a copy of the original workload with a different name and a different workloadId.

POST /workload/clone/<workloadId>

Request Body

- name - required - the name for the new (cloned) workload. Must be unique for this user.

Example

POST /workload/ad5a66c3-8895-4d71-b786-1d129b33326e

Returns

The workload JSON. Will include the workloadId.

Retrieve workload

Retrieve one workload.

GET /workload/<workloadId>

Returns

The workload JSON. Will include the workloadId.

Retrieve multiple workloads

Retrieve multiple workloads.

GET /workload

Query Parameters

- name - optional. If you want to retrieve a workload by name, you can specify the workload name here

Note that even though only one workload will be returned (because workload names are unique) an array will always be returned from this method.

Returns

An array of workloads.

If there is only one workload returned then the full JSON for the workload will be returned.

If there is more than one workload returned then only limited information for the workloads will be returned (no elements, no parameters, no plan, etc.)

Update workload

Update the workload.

PUT /workload/<workloadId>

Request Body

The workload JSON.

Notes

The following properties can be updated by this method: name, description, parameters, metadata. They are replaced in their entirety. To update elements use the element CRUD methods below. Updates need to conform to the same schema requirements as the create above. If the name is changed, it must be unique for this user.

A workload cannot be updated while workload planning or execution is in progress.

When a workload is updated, any existing plan is invalid and hence is removed.

Example

PUT /workload/ad5a66c3-8895-4d71-b786-1d129b33326e

Returns

The workload JSON. Will include the workloadId.

Delete workload

Delete a workload.

DELETE /workload/<workloadId>

Notes

A workload cannot be deleted while workload planning or execution is in progress.

Currently you can delete the workload even if it has running instances. This will cause those running instances to effectively be “orphaned” as they will no longer belong to a workload. So before you delete a workload you should check whether it has running instances.

Example

DELETE /workload/ad5a66c3-8895-4d71-b786-1d129b33326e

Returns

{ “deleted”: true } (if the workload was sucessfully deleted)

- or -

{ “deleted”: false } (if the workload was already deleted)

Create/Update workload element

Create or update a workload element.

PUT /workload/<workloadId>/element

Request Body

The workload element JSON.

Notes

A workload element cannot be created or updated while workload planning or execution is in progress.

When a workload is updated, any existing plan is invalid and hence is removed.

The workload element JSON is validated against the schema defined here: Workload Element Schema.

The schema includes: name, uri, parameters, and metadata. Metadata here is metadata for the workload element and can be any valid JSON.

That metadata will be attached to the instance when it is created. name and uri are required. All other parameters are optional.

Returns

The workload element JSON.

Delete workload element

Delete a workload element from the workload.

DELETE /workload/<workloadId>/element

Request Body

The workload element JSON. The only required property is name (to identify the workload element that is to be deleted).

Notes

A workload element cannot be deleted while workload planning or execution is in progress.

When a workload is updated (in this case by having an element deleted) any existing plan is invalid and hence is removed.

Returns

The workload element JSON.

Plan workload

Generate the workload plan. The plan can be for activation or deactivation.

PUT /workload/<workloadId>/plan

Query Parameters

- action = activate | deactivate

Example

PUT /workload/ad5a66c3-8895-4d71-b786-1d129b33326e/plan?action=activate

Returns

The workload element JSON. This will include the plan. The plan property starts with *serial*.

Execute workload plan

Execute the workload plan.

PUT /workload/<workloadId>/execute

Notes

The execute will fail if the workloadStatus is in-progress.

Returns:

```
{
        "workloadStatus": "in-progress",
        "action": "activate",
        "transactionId": "<the transaction id>"
}
```

Retrieve transaction steps

Retrieve transaction steps.

GET /transaction/<transactionId>/steps

Query Parameters

- begin = the beginning Log Sequence Number (LSN). Optional. If omitted then begin = 0.
- end = the ending LSN. Optional. If omitted then end = 10,000.

Notes

When a workload is executed, the results of all the steps in the workload plan are logged to a transaction log. Each entry in the log has a “Log Sequence Number” or LSN. Some of those log entries are relevant to the progress of the step and some are not. To monitor the progress of the step we are mostly interested in when the step started (step status = in-progress) and when it has finished (step status = completed or step status = failed). The *begin* and *end* query parameters can be used to limit the number of records returned. Note that once a step with a specific LSN has been returned it will never change, so the best way to poll with this method is to advance the begin LSN to the LSN of the last step that was returned in the previous call to this method.

Result

An array of log entries. These have a stepId so they can be correlated to the steps in the workload plan.

Each log entry is:

```
{
        "lsn": 456,
        "stepId": "<the unique step id - matches the plan>",
        "timestamp": "<UTC timestamp>",
        "status": "in-progress | completed | failed",
        "reason": "<failure reason if status = failed>",
        "elapsedTimeInSeconds": "<elapsed time in seconds for the workload step>"
}
```

Retrieve transaction errors

Retrieve transaction errors.

GET /transaction/<transactionId>/errors

Notes

In the workload API, errors from the provider side are not necessarily fatal. The workload API implements retries and retry delays. So it may be that an error was received from a provider, but that the operation was retried and succeeded the second time. This method can be used to see details for all errors that occur during the workload execution.

Returns

An array of log entries. These have a stepId so they can be correlated to the steps in the workload plan.

Each log entry is:

```
{
        "lsn": 456,
        "stepId": "<the unique step id - matches the plan>",
        "timestamp": "<UTC timestamp>",
        "code": "<error code>",
        "message": "<error message>",
        "ticket": "<error ticket>"
}
```

Retrieve transaction status

Retrieve transaction status.

GET /transaction/<transactionId>/status

Returns:

```
{
        "transactionId": "<the transaction id>",
        "workloadId": "<the workload id>",
        "status": <the transaction status: in-progress | completed | failed>",
        "reason": <failure reason if status = failed>",
        "stepId": <the stepId of the step that failed, if status = failed>",
        "started": "<UTC timestamp - start time>",
        "ended": "<UTC timestamp - end time>",
        "elapsedTimeInSeconds": "<elapsed time in seconds for transaction>"
}
```

Cancel transaction

Cancel a transaction that is currently in-progress.

PUT /transaction/<transactionId>/cancel

Returns:

```
{
  "transactionId": "<the transaction id>",
  "workloadId": "<the workload id>",
  "action": "cancel",
  "status": "in-progress"
}
```