Tapis Documentation

Joe Stubbs

May 12, 2021

Introduction:

CHAPTER 1

Welcome to Tapis v3

1.1 What is Tapis v3

Tapis is an NSF-funded web-based API framework for securely managing computational workloads across infrastructure and institutions, so that experts can focus on their research instead of the technology needed to accomplish it. As part of work funded by the National Science Foundation starting in 2019, Tapis is delivering a version 3 ("v3") of its platform with several new capabilities, including a multi-site Security Kernel, Streaming Data APIs, and first-class support for containerized applications.

Tapis v3 is a new project with an initial "Beta" release that will be available late July, 2020. This documentation describes **Tapis v3**. If you are looking for documentation on the cuurent production Tapis v2 platform, please go here.

1.2 About This Documentation

This documentation includes the *Getting Started* guide – a basic introduction to Tapis v3 – as well as an in-depth Technical guide; see the *Technical Guide - Overview* or browse directly to the section of interest. All Tapis APIs are defined using the OpenAPI v3 specification. If you are looking for the complete Tapis API reference or to download the Tapis OpenAPI spec files, see our LiveDocs.

CHAPTER 2

Getting Started

This Getting Started guide will walk you through the initial steps of setting up the necessary accounts and installing the required software before moving to the Tapis Quickstart. If you are already using Docker Hub and the TACC Cloud APIs, feel free to jump right to the *Tapis Quickstart* or check out the Tapis Live Docs site.

2.1 Account Creation and Software Installation

2.1.1 Create a TACC account

The main instance of the Tapis platform is hosted at the Texas Advanced Computing Center (TACC). TACC designs and deploys some of the world's most powerful advanced computing technologies and innovative software solutions to enable researchers to answer complex questions. To use the TACC-hosted Tapis platform, please create a TACC account .

2.2 Tapis Quickstart

In this Quickstart, we will use the Tapis APIs to manage files on a TACC storage system. To begin we use our credentials to get a Tapis token from the authenticator. For this quickstart, we will be using the `tacc` tenant with base URL `https://tacc.tapis.io`.

With CURL:

```
$ curl -H "Content-type: application/json" -d '{"username": "apitest", "password":
↪"abcde123", "grant_type": "password" }' \
https://tacc.tapis.io/v3/oauth2/tokens
```

Be sure to export the access token returned from the above CURL command as an environment variable:

```
$ export JWT=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJqdGkiOiJmN2....
```

With PySDK

```
from tapipy.tapis import Tapis
# Create python Tapis client for user
t = Tapis(base_url= "https://tacc.tapis.io", username="your_tacc_username", password=
↪"your_tacc_password")
# Call to Tokens API to get access token
t.get_tokens()
```

Now that we have an access token, we are ready to create a Tapis system. For this quick-start, we will register an S3 bucket we have pre-create with Amazon's AWS S3 service.

We assume an S3 bucket has been registered with AWS with URL `https://<your_bucket_id>.s3.us-east-1.amazonaws.com/` and that you have access to the bucket `accessKey` and `accessSecret`.

To register the S3 bucket with Tapis we do the following:

With PySDK

```
# the description of an S3 bucket
s3_bucket = {
  "name":"my.test.bucket",
  "description":"Test Tapis Bucket",
  "host":"https://<your_bucket_id>.s3.us-east-1.amazonaws.com/",
  "systemType":"S3",
  "defaultAccessMethod":"ACCESS_KEY",
  "effectiveUserId":"<your_tacc_username>",
  "bucketName":"tapis-files-bucket",
  "rootDir":"/",
  "canExec": False,
  "accessCredential":
  {
    "accessKey":"***",
    "accessSecret":"***"
  }
}

# create the system in Tapis
t.systems.createSystem(**s3_bucket)
```

The output should look similar to the following; it describes the System that was just created:

```
accessCredential: None
bucketName: my.test.bucket
created: 2020-06-25T16:11:52.543Z
defaultAccessMethod: ACCESS_KEY
deleted: False
description: Test Tapis Bucket
effectiveUserId: <your_tacc_username>
enabled: False
```

(continues on next page)

(continued from previous page)

```
host: https://tapis-demo.s3.us-east-1.amazonaws.com/
id: 2
canExec: False
jobCapabilities: []
jobLocalArchiveDir: None
jobLocalWorkingDir: None
jobRemoteArchiveDir: None
jobRemoteArchiveSystem: None
name: tapis-demo
notes:

owner: <yout_tacc_username>
port: 0
proxyHost:
proxyPort: 0
rootDir: /
systemType: S3
tags: []
tenant: dev
updated: 2020-06-25T16:11:52.543Z
useProxy: False
```

We are now able to list files in our bucket using the Files API.

With PySDK

```
t.files.listFiles(systemId="my.test.bucket", path="/")
```

The output should include a list of all files in the bucket; for example

```
[
 lastModified: 2020-06-12T16:29:10Z
 name: Bora2.jpg
 path: Bora2.jpg
 size: 390672,

 lastModified: 2020-07-21T16:27:53Z
 name: plot_2020-07-21T01:29:26.640144Z.png
 path: plot_2020-07-21T01:29:26.640144Z.png
 size: 31211
```

]

CHAPTER 3

Technical Guide - Overview

The Technical Guide for Tapis provides a detailed reference to the primary services in the platform.

- *Sites, Tenancy and Authentication*: Complete reference for Tapis authentication.
- *Systems*: Registering and working with remote storage and execution resources.
- *Files*: Managing data on Tapis systems.
- *Applications*: Registering and working with applications.
- *Jobs*: Running an application on a system.
- *Meta*: Storing and retrieving metadata pertaining to your projects.
- *PgREST*: Hosted, HTTP RESTful API to a managed Postrges database instance.
- *Streams*: Managing streaming data from sensors and other instruments.
- *Actors*: Execute containerized functions ("actors") in response to events or other messages.
- *Security*: Securing the digital assets comprising your project.

CHAPTER 4

Sites, Tenancy and Authentication

4.1 Sites

Tapis supports geographically distributed deployments where different components are running in different data centers and managed by different institutions. These physically isolated installations of Tapis software are referred to as *sites*. There is a single *primary site* and zero or more *associate sites* within a Tapis installtion.

4.1.1 Primary Site

The primary site in a Tapis installation runs a complete set of Tapis API services and all associated 3rd-party services, such as databases and message queues. The creation of new sites is coordinated through the primary site, and the primary site runs the unique instance of the Sites and Tenants API (see *Tenants* below) which maintain the complete registry of all sites and tenants in the installation.

The primary site of the main Tapis installation is hosted at the Texas Advanced Computing Center, at URL https://tapis.io.

4.1.2 Associate Sites

Associate sites are required to run the Tapis Security Kernel, a compliant Token Generator API, and one or more additional Tapis services. Each associate site is managed and operated by a separate, partner institution. For Tapis services not run at the associate site, the corresponding service at the primary site is used for requests. In this way, partner institutions can choose which Tapis services to run within their institution and leverage the primary site deployment for the rest.

4.1.3 Deployment

The official Tapis deployment tooling targets the Kubernetes container orchestration platform. The project maintains a set of deployment templates which can be used in conjunction with a configuration file to deploy any number of Tapis services. If your institution is interested in becomming a Tapis associate site please contact us.

Details about the current list of sites is available from the tenants API. For example, one can retrieve the full list of sites as follows:

With PySDK:

```
>>> t.tenants.list_sites()
```

With CURL:

```
$ curl https://admin.tapis.io/v3/sites
```

The response will look similar to the following (the response below is truncated for brevity):

```
[
 base_url: https://tapis.io
 primary: True
 services: ['systems', 'files', 'security', 'tokens', 'streams', 'authenticator',
↪'meta', 'actors']
 site_admin_tenant_id: admin
 site_id: tacc
 tenant_base_url_template: https://${tenant_id}.tapis.io]
 .  .  .
]
```

Each site has a `site_id` as well as a list of Tapis services it provides and the tenant ID of the administrative tenant (`admin_tenant`) associated with it.

4.2 Tenants

Tapis is a *multi-tenant* platform, meaning that different projects (or *tenants*) can have logically isolated views of the Tapis objects (i.e., the systems, files, actors, etc.) they create for their project.

Each tenant is made up of the following:

1. A base URL with which to access the tenant; by default, the base URL takes the form `https://<tenant_id>.tapis.io` where `tenant_id` is a short, unique identifier for the tenant in the Tapis system. For example, `https://tacc.tapis.io` is the base URL for the `tacc` tenant.
2. An *authenticator* providing the rules for who can authenticate in the tenant.

Additionally, each tenant is "managed" by a site.

To see the current list of tenants registered with Tapis, we can use the tenants API.

With PySDK:

```
>>> t.tenants.list_tenants()
```

With CURL:

```
$ curl https://tacc.tapis.io/v3/tenants
```

The response will look similar to the following (the response below is truncated for brevity):

```
allowable_x_tenant_ids: ['tacc']
authenticator: https://tacc.tapis.io/v3/oauth2
base_url: https://tacc.tapis.io
create_time: Thu, 02 Jul 2020 23:45:16 GMT
```

(continues on next page)

(continued from previous page)

```
description: Production tenant for all TACC users.
is_owned_by_associate_site: False
last_update_time: Thu, 02 Jul 2020 23:45:16 GMT
owner: CICSupport@tacc.utexas.edu
public_key: -----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAz7rr5CsFM7rHMFs7uKIdcczn0uL4ebRMvH8pihrg1tW/
↪fp5Q+5ktltoBTfIaVDrXGF4DiCuzLsuvTG5fGElKEPPcpNqaCzD8Y1v9r3tfkoPT3Bd5KbF9f6eIwrGERMTs1kv7665pliwehz
↪EnSg1N3Oj1x8ktJPwbReKprHGiEDlqdyT6j58l/I+9ihR6ettkMVCq7Ho/
↪bsIrwm5gP0PjJRvaD5Flsze7P4gQT37D1c5nbLR+K6/T0QTiyQIDAQAB
-----END PUBLIC KEY-----
security_kernel: https://tacc.tapis.io/v3/security
service_ldap_connection_id: None
tenant_id: tacc
token_service: https://tacc.tapis.io/v3/tokens
user_ldap_connection_id: tacc-all,

allowable_x_tenant_ids: ['dev']
authenticator: https://dev.tapis.io/v3/oauth2
base_url: https://dev.tapis.io
create_time: Fri, 19 Jun 2020 20:36:38 GMT
description: The dev tenant.
is_owned_by_associate_site: False
last_update_time: Fri, 19 Jun 2020 20:36:38 GMT
owner: CICSupport@tacc.utexas.edu
public_key: -----BEGIN PUBLIC KEY-----

. . .
```

Here we see the first two tenants registered in the Tapis framework, the `tacc` and `dev` tenants.

In general, the rules for authentication vary from tenant to tenant within Tapis. For example, in the `tacc` tenant, any user with a valid TACC account can authenticate and use the APIs. The `dev` tenant is a development sandbox with test accounts used by the core Tapis team.

This documentation focuses on the `tacc` tenant; however, much of what follows in the subsequent sections will be the same regardless of the tenant you are using.

4.3 Authentication

The default authenticator provided by the Tapis project is based on OAuth2, and this is the authentication mechanism in place for the `tacc` tenant. The OAuth2-based authentication services are available via the `/v3/oauth2` endpoints.

OAuth uses different *grant type flows* for generating tokens in different situations. We do not provide a comprehensive guide to OAuth2; for that, we refer the reader to the OAuth2 docs. Instead, we provide a guide to the two most common use cases for users: generating tokens for themselves using the *password* grant flow, and generating tokens on behalf of others in a web application using the *authorization code* grant flow.

In the PySDK examples that follow, we will tacitly assume the `tapipy.tapis.Tapis` object has been instantiated as the Python object `t`. There are several options in the `Tapis` constructor, but the basic options include `base_url` and `username`, for example:

```
>>> t = Tapis(base_url='https://tacc.tapis.io', username='jdoe')
```

The simplest case is that you want to generate a Tapis OAuth token for yourself; to do this you can use the *password* grant flow, providing your username and password.

Tapis v3 tries to make this process as easy as possible by providing a simplified version of the password grant flow that does not require an OAuth client (see the oauth-clients-label section).

With PySDK:

```
>>> t = Tapis(base_url='https://tacc.tapis.io', username='apitest', password='abcd123
↪')
>>> t.get_tokens()
```

With CURL:

```
> curl -H "Content-type: application/json" -d '{"username": "apitest", "password":
↪"abcde123", "grant_type": "password" }' \
https://tacc.tapis.io/v3/oauth2/tokens
```

In the PySDK, the access token is a first-class Python object stored on the Tapis object (the `t` in the examples above). We can inspect it

```
>>> t.access_token

access_token: eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.
↪eyJqdGkiOiJmN2I5YjE5ZS02MDk5LTRmODItYTcyMi1iNjEwYzVkMGJhMGMiLCJpc3MiOiJodHRwczovL3RhY2MudGFwaXMuaW8
↪alC8rRM-zNsHKcUiz3-tOJPaYtFksKb4Bit_aFE1HH_X_znnP2QkJaqc-
↪xaRoMlQu26MN72TlJE0siIN3T38xXWBGDumHUYbvnNzT-
↪7lk7AQU5MHSyCWx8IRDmTSbqmWOG8WBMCIV9Dh84mDd-X6eLJQ_cz1QqMAiI_
↪cPgA9VVE22qDK3Lbz2pp9t0sm-l9XjE5y5Im8Y0B2p0ssPD0TjW20C5yngZ4-4jowDafboKlscog9ko-
↪adrsVIjG_r-ccCUX3r8SVwQLypZFZAPKqbVzl8jt_
↪mCi30W8AYwiaYGmH7INBbHI9hO7kwJNFMuSylejFhMslxgdzGlIAyXauwg
claims: {'jti': 'f7b9b19e-6099-4f82-a722-b610c5d0ba0c', 'iss': 'https://tacc.tapis.io/
↪v3/tokens', 'sub': 'apitest@tacc', 'tapis/tenant_id': 'tacc', 'tapis/token_type':
↪'access', 'tapis/delegation': False, 'tapis/delegation_sub': None, 'tapis/username
↪': 'apitest', 'tapis/account_type': 'user', 'exp': 1595099456, 'tapis/client_id':␣
↪None, 'tapis/grant_type': 'password'}
expires_at: 2020-07-18 19:10:56+00:00
expires_in: <function Tapis.set_access_token.<locals>._expires_in at 0x7f29e213c510>
jti: f7b9b19e-6099-4f82-a722-b610c5d0ba0c
original_ttl: 14400
```

What we see is that the `access_token.access_token` is a Python string representing a JSON Web Token (JWT). JWTs are cryptographically signed with the private key associated with the tenant, and anyone can validate the signature by using the corresponding public key associated with the tenant (see Tenants section above). The public key for each tenant is available from the Tenants API. The core Tapis services will validate the access token sent on a given API call using the public key associated with the tenant to verify the JWT signature.

In order to use an access token in an API request to Tapis, pass the token in as the value of the `X-Tapis-Token` header. The PySDK will automatically send the token via this header for you. In CURL examples used throughout this documentation, we assume the raw JWT string representing an access token (like the above) has been exported as a shell variable; i.e.,

```
$ export JWT=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.
↪eyJqdGkiOiJmN2I5YjE5ZS02MDk5LTRmODItYTcyMi1iNjEwYzVkMGJhMGMiLCJpc3MiOiJodHRwczovL3RhY2MudGFwaXMuaW8
↪alC8rRM-zNsHKcUiz3-tOJPaYtFksKb4Bit_aFE1HH_X_znnP2QkJaqc-
↪xaRoMlQu26MN72TlJE0siIN3T38xXWBGDumHUYbvnNzT-
↪7lk7AQU5MHSyCWx8IRDmTSbqmWOG8WBMCIV9Dh84mDd-X6eLJQ_cz1QqMAiI_
↪cPgA9VVE22qDK3Lbz2pp9t0sm-l9XjE5y5Im8Y0B2p0ssPD0TjW20C5yngZ4-4jowDafboKlscog9ko-
↪adrsVIjG_r-ccCUX3r8SVwQLypZFZAPKqbVzl8jt_
↪mCi30W8AYwiaYGmH7INBbHI9hO7kwJNFMuSylejFhMslxgdzGlIAyXauwg
```

With that variable set, we can use the `-H` flag with curl to set the `X-Tapis-Token` header as follows:

```
$ curl -H "X-Tapis-Token: $JWT" ....
```

Note also the *claims* associated with the access token. These claims provide information about the token, including the user it represents (`apitest` in the above example), the tenant it belongs to (`tacc` above) when it expires, etc. Tapis tokens always include the following standard claims:

Claim	Description	Example Value
sub	The subject of the token; the subject uniquely identifies the user in a Tapis installation. The format is `user @ tenant`	apitest@tacc
exp	The expiry associated with the token.	1595099456
jti	Unique identifier for the token.	f7b9b19e-6099-4f82-a722-b610c5d0ba0c
iss	The identifier (URL) of the issuer of the JWT. For Tapis, the issuer will be a Tokens API.	https://tacc.tapis.io/v3/tokens

Additional custom claims specific to Tapis are namespaced with `tapis/` at the beginning of the claim name. The authenticator for each tenant may optionally choose to support one or more of these additional claims. The following claims are encouraged and supported by the default OAuth2 Tapis authenticator.

Claim	Description	Example Value
tapis/tenant_id	The tenant of the subject.	tacc
tapis/username	The username of the subject.	apitest
tapis/token_type	Type of token: `access` or `refresh`	access
tapis/account_type	Type of account: `user` or `service`	user
tapis/delegation	Whether a delegation flow was used to generate this token. (`true` or `false`).	false
tapis/delegation_sub	For a delegation token, the subject who actually generated the token. In form `user @ tenant`	superuser@tacc
tapis/client_id	The id of the OAuth client used to generate the token.	tacc.CIC.tokenapp
tapis/grant_type	The grant type used to generate the token.	authorization_code

The authenticator for your tenant may include additional claims not listed here. In order to use the more advanced OAuth2 flows, including any use of the authorization code grant type and to generate refresh tokens with the password grant type, you must generate an OAuth2 *client*. Clients in OAuth2 represent applications (for example, a web or mobile application) that will interact with the OAuth2 server to generate tokens on behalf of one or more users. Clients are created and managed using the `/v3/oauth2/clients` endpoints.

4.3.1 Creating Clients

To create a client, make a POST request the the Clients API. All fields are optional; if you do not pass a `client_id` or `client_key` in the request, the clients API will generate random ones for you. In order to use the `authorize_code` grant type you will need to set the `callback_url` when registering your client (see :ref: *_auth_code*). For a complete list of available parameters, see the API live-docs for Clients.

With PySDK:

```
>>> t.authenticator.create_client(client_id='test', callback_url='https://foo.example.
↪com/oauth2/callback')
```

With CURL:

```
$ curl -H "X-Tapis-Token: $JWT" -H "Content-type: application/json" -d '{"client_id":
↪"test", "callback_url": "https://foo.example.com/oauth2/callback" https://tacc.
↪tapis.io/v3/oauth2/clients
```

The response will be similar to

```
callback_url: https://foo.example.com/oauth2/callback
client_id: test
client_key: WQZlQlMoxOynW
create_time: Sat, 18 Jul 2020 19:09:47 GMT
description:
display_name: https://foo.example.com/oauth2/callback
last_update_time: Sat, 18 Jul 2020 19:09:47 GMT
owner: apitest
tenant_id: tacc
```

4.3.2 Listing Clients

With PySDK:

```
>>> t.authenticator.list_clients()
```

With CURL:

```
$ curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/oauth2/clients
```

The response will be similar to

```
[
callback_url: https://foo.example.com/oauth2/callback
client_id: test
client_key: WQZlQlMoxOynW
create_time: Sat, 18 Jul 2020 19:09:47 GMT
description:
display_name: https://foo.example.com/oauth2/callback
last_update_time: Sat, 18 Jul 2020 19:09:47 GMT
owner: apitest
tenant_id: tacc]
```

4.3.3 Deleting Clients

You can also delete clients you are no longer using; just pass the `client_id` of the client to be deleted:

With PySDK:

```
>>> t.authenticator.delete_client(client_id='test')
```

With CURL:

```
$ curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/oauth2/clients/test
```

A null response is returned from a successful delete request.

Authorization Code Grant - Generating Tokens For Users

An important aspect of OAuth2 is that it enables applications to generate tokens on behalf of users without the applications needing to possess user credentials (i.e., passwords). In this section, we discuss using the OAuth2 *authorization code* grant type to do just that.

Assuming a Model-View-Controller (MVC) architecture, there are two controllers that must be written to support the authorization code grant type flow.

1. A controller to determine if the user already has a valid access token and direct them to the OAuth2 authorization server when they do not. This controller starts the authorization code process. To do so, it should:

- First inform the user that they will be asked to authenticate with their tenant username and password and then be asked to grant authorization to your client application.
- Redirect the user to the OAuth2 server's authorization URL. In the default Tapis authenticator, the authorization URL path is `/v3/oauth2/authorize`; for example, `https://tacc.tapis.io/v3/oauth2/authorize` in the `tacc` tenant.
- The redirect request should include the following query parameters:
  - `client_id`: the id of your client.
  - `client_redirect_uri`: the URI to redirect back to with the authorization code. This must match the `callback_url` parameter associated with your client.
  - `response_type`: should always have the value `code`.

2. A controller to process the authorization code returned and retrieve an access token on the user's behalf. This controller receives requests containing authorization codes from the OAuth2 server after the user has successfully authenticated with said OAuth2 server, and it immediately exchanges the code for a token.

- Responds to `GET` requests to the URL defined in the `callback_url` parameter of your client.
- Retrieves the `code` query parameter from the request.
- Makes a `POST` request to the OAuth2 server's tokens endpoint to generate a token. In the default Tapis authenticator, the tokens URL path is `/v3/oauth2/tokens`; for example, `https://tacc.tapis.io/v3/oauth2/tokens` in the `tacc` tenant. The POST body must include the following parameters:
  - `code`: the code the controller just received in the request from the OAuth2 server.
  - `redirect_uri`: should be the same as the `callback_url` parameter of your client.
  - `grant_type`: should always have the value `authorization_code`.

Note that many popular web frameworks support OAuth2 flows with minimal custom coding required.

The final step to using the authorization code grant type is to register a client (see above) with a `callback_url` parameter equal to the URL within your web application where it will handle converting authorization codes into access tokens (i.e., controller 2 above).

The Tapis Token Web Application

Tapis provides a graphical interface via a web application that enables users to generate tokens. The Tapis Web Application is available by default for any tenant using the default Tapis authenticator, including the `tacc` tenant. The Tapis Token Web Application serves as an example of an application using the authorization code grant type.

The Tapis Token Web Application and its source code are available at the following URLs:

- Token App (`tacc` tenant): https://tacc.tapis.io/v3/oauth2/webapp
- Token App source code: https://github.com/tapis-project/authenticator

CHAPTER 5

Systems

Once you are authorized to make calls to the various services, one of first things you may want to do is view storage and execution resources available to you or create your own. In Tapis a storage or execution resource is referred to as a **system**.

5.1 Overview

A Tapis system represents a server or collection of servers exposed through a single host name or IP address. Each system is associated with a specific tenant. A system can be used for the following purposes:

- Running a job, including:
  - Staging files to an execution system in preparation for running a job.
  - Executing a job on an execution system.
  - Archiving files and data on a remote storage system after job execution.
- Storing and retrieving files and data.

Each system is of a specific type and owned by a specific user who has special privileges for the system. The system definition also includes the user that is used to access the system, referred to as *effectiveUserId*. This access user can be a specific user (such as a service account) or dynamically specified as `${apiUserId}` in which case the user name is extracted from the identity associated with the request to the service.

At a high level a system represents the following information:

Id A short descriptive name for the system that is unique within the tenant.

Description An optional more verbose description for the system.

Type of system LINUX or S3

Owner A specific user set at system creation. By default this is `${apiUserId}`, the user making the request to create the system.

Host name or IP address. FQDN or IP address

Enabled flag Indicates if system is currently considered active and available for use. Default is true.

Effective User The user name to use when accessing the system. Referred to as *effectiveUserId*. A specific user (such as a service account) or the dynamic user `${apiUserId}`

Default authorization method How access authorization is handled by default. Authorization method can also be specified as part of a request. Supported methods: PASSWORD, PKI_KEYS, ACCESS_KEY.

Bucket name For an S3 system this is the name of the bucket.

Effective root directory Directory to be used when listing files or moving files to and from the system.

DTN system Id An alternate system to use as a Data Transfer Node (DTN).

DTN mount point Mount point (aka target) used when running the mount command on this system.

DTN mount source path The path exported by *dtnSystemId* that matches the *dtnMountPoint* on this system. This will be relative to *rootDir* on *dtnSystemId*.

isDtn flag Indicates if system will be used as a data transfer node (DTN). By default this is *false*.

canExec flag Indicates if system can be used to execute jobs.

Job related attributes Various attributes related to job execution such as *jobRuntimes*, *jobWorkingDir*, *jobIsBatch*, *batchScheduler*, *batchLogicalQueues* and *jobCapabilities*

When creating a system the required attributes are: *id*, *systemType*, *host*, *defaultAuthnMethod* and *canExec*. Depending on the type of system and specific values for certain attributes there are other requirements.

5.2 Getting Started

Before going into further details about Systems, here we give some examples of how to create and view systems. In the examples below we assume you are using the TACC tenant with a base URL of `tacc.tapis.io` and that you have authenticated using PySDK or obtained an authorization token and stored it in the environment variable JWT, or perhaps both.

5.2.1 Creating a System

Create a local file named `system_s3.json` with json similar to the following:

```
{
  "id":"tacc-bucket-sample-<userid>",
  "description":"My Bucket",
  "host":"https://tapis-sample-test-<userid>.s3.us-east-1.amazonaws.com/",
  "systemType":"S3",
  "defaultAuthnMethod":"ACCESS_KEY",
  "effectiveUserId":"${owner}",
  "bucketName":"tapis-tacc-bucket-<userid>",
  "rootDir":"/",
  "canExec": false,
  "authnCredential":
  {
    "accessKey":"***",
    "accessSecret":"***"
  }
}
```

where <userid> is replaced with your user name, your S3 host name is updated appropriately and if desired you have filled in your access key and secret. Note that credentials are stored in the Security Kernel and may also be set or updated using a separate API call. However, only specific Tapis services are authorized to retrieve credentials.

Using PySDK:

```
import json
from tapipy.tapis import Tapis
t = Tapis(base_url='https://tacc.tapis.io', username='<userid>', password=
↪'************')
with open('system_s3.json', 'r') as openfile:
    my_s3_system = json.load(openfile)
t.systems.createSystem(**my_s3_system)
```

Using CURL:

```
$ curl -X POST -H "content-type: application/json" -H "X-Tapis-Token: $JWT" https://
↪tacc.tapis.io/v3/systems -d @system_s3.json
```

5.2.2 Viewing Systems

Retrieving details for a system

To retrieve details for a specific system, such as the one above:

Using PySDK:

```
t.systems.getSystemById(systemId='tacc-bucket-sample-<userid>')
```

Using CURL:

```
$ curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/systems/tacc-bucket-sample-
↪<userid>
```

The response should look similar to the following:

```
{
   "result": {
       "tenant": "dev",
       "id": "tacc-bucket-sample-<userid>",
       "description": "My Bucket",
       "systemType": "S3",
       "owner": "<userid>",
       "host": "tapis-sample-test-<userid>.s3.us-east-1.amazonaws.com",
       "enabled": true,
       "effectiveUserId": "<userid>",
       "defaultAuthnMethod": "ACCESS_KEY",
       "authnCredential": null,
       "bucketName": "tapis-tacc-bucket-<userid>",
       "rootDir": "/",
       "port": 9000,
       "useProxy": false,
       "proxyHost": "",
       "proxyPort": -1,
       "dtnSystemId": null,
       "dtnMountPoint": null,
```

(continues on next page)

(continued from previous page)


```
        "dtnMountSourcePath": null,
        "isDtn": false,
        "canExec": false,
        "jobRuntimes": [],
        "jobWorkingDir": null,
        "jobEnvVariables": [],
        "jobMaxJobs": 2147483647,
        "jobMaxJobsPerUser": 2147483647,
        "jobIsBatch": false,
        "batchScheduler": null,
        "batchLogicalQueues": [],
        "batchDefaultLogicalQueue": null,
        "jobCapabilities": [],
        "tags": [],
        "notes": {},
        "uuid": "f83606bf-7a1a-4ff0-9953-dd732cc07ac0",
        "deleted": false,
        "created": "2021-04-26T18:45:40.771Z",
        "updated": "2021-04-26T18:45:40.771Z"
    },
    "status": "success",
    "message": "TAPIS_FOUND System found: tacc-bucket-sample-<userid>",
    "version": "0.0.1",
    "metadata": null
}
```


Note that authnCredential is null. Only specific Tapis services are authorized to retrieve credentials.

Retrieving details for all systems

To see the current list of systems that you are authorized to view:

(NOTE: See the section below on searching and filtering to find out how to control the amount of information returned)

Using PySDK:

```
t.systems.getSystems()
```

Using CURL:

```
$ curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/systems?select=allAttributes
```

The response should contain a list of items similar to the single listing shown above.

5.3 Minimal Definition and Restrictions

When creating a system the required attributes are: *id*, *systemType*, *host*, *defaultAuthnMethod* and *canExec*. Depending on the type of system and specific values for certain attributes there are other requirements. The restrictions are:

- If *systemType* is S3 then *bucketName* is required, *canExec* and *isDtn* must be false.
- If *systemType* is LINUX then *rootDir* is required.
- If *effectiveUserId* is `${apiUserId}` (i.e. it is not static) then *authnCredential* may not be specified.

- If *isDtn* is true then *canExec* must be false and following may not be specified: *dtnSystemId*, *dtnMountSourcePath*, *dtnMountPoint*, all job execution related attributes.
- If *canExec* is true then *jobWorkingDir* is required and *jobRuntimes* must have at least one entry.
- If *jobIsBatch* is true then *batchScheduler* must be specified.
- If *jobIsBatch* is true then *batchLogicalQueues* must have at least one item.
  - If *batchLogicalQueues* has more than one item then *batchLogicalDefaultQueue* must be specified.
  - If *batchLogicalQueues* has exactly one item then *batchLogicalDefaultQueue* is set to that item.

5.4 Permissions

At system creation time the owner is given full system authorization. If the effective access user *effectiveUserId* is a specific user (such as a service account) then this user is given the same authorizations. If the effective access user is the dynamic user `${apiUserId}` then the authorizations for each user must be granted and credentials created in separate API calls. Permissions for a system may be granted and revoked through the systems API. Please note that grants and revokes through this service only impact the default role for the user. A user may still have access through permissions in another role. So even after revoking permissions through this service when permissions are retrieved the access may still be listed. This indicates access has been granted via another role.

Permissions are specified as either `*` for all permissions or some combination of the following specific permissions: `("READ","MODIFY","EXECUTE")`. Specifying permissions in all lower case is also allowed. Having `MODIFY` implies `READ`.

5.5 Authorization Credentials

At system creation time the authorization credentials may be specified if the effective access user *effectiveUserId* is a specific user (such as a service account) and not a dynamic user, i.e. `${apiUserId}`. If the effective access user is dynamic then authorization credentials for any user allowed to access the system must be registered in separate API calls. Note that the Systems service does not store credentials. Credentials are persisted by the Security Kernel service and only specific Tapis services are authorized to retrieve credentials.

5.6 Capabilities

In addition to the system capabilities reflected in the basic attributes each system definition may contain a list of additional capabilities supported by that system. An Application or Job definition may then specify required capabilities. These are used for determining eligible systems for running an application or job.

5.7 Deletion

A system may be deleted. Deletion means the system is marked as deleted and is no longer available for use. It will no longer show up in searches and operations on the system will no longer be allowed. The system definition is retained for auditing purposes. Note this means that system IDs may not be re-used after deletion.

5.8 Table of Attributes

Attribute	Type	Example	Notes
tenant	String	designsafe	• Name of the tenant for which the system is defined. • *tenant* + *id* must be unique.
id	String	ds1.storage.default	• Identifier for the system. URI safe, see RFC 3986. • *tenant* + *id* must be unique. • Allowed characters: Alphanumeric [0-9a-zA-Z] and special characters [-._~].
description	String	Default storage	• Description
systemType	enum	LINUX	• Type of system. • Types: LINUX, S3
owner	String	jdoe	• User name of *owner*. • Variable references: *${apiUserId}*
host	String	data.tacc.utexas.edu	• Host name or ip address of the system
enabled	boolean	FALSE	• Indicates if system currently enabled for use. • May be updated using the enable/disable endpoints.

Continued on next page

Table 1 – continued from previous page

Attribute	Type	Example	Notes
effectiveUserId	String	tg869834	• User to use when accessing the system. • May be a static string or a variable reference. • Variable references: *${apiUserId}*, *${owner}* • On output variable reference will be resolved.
defaultAuthnMethod	enum	PKI_KEYS	• How access authorization is handled by default. • Can be overridden as part of a request to get a system or credentials. • Methods: PASSWORD, PKI_KEYS, ACCESS_KEY

Continued on next page

Table 1 – continued from previous page

Attribute	Type	Example	Notes
authnCredential	Credential		• On input credentials to be stored in Security Kernel. • *effectiveUserId* must be static, either a string constant or ${owner}. • May not be specified if *effectiveUserId* is dynamic, i.e. *${apiUserId}*. • On output contains credentials for *effectiveUserId*. • Returned credentials contain relevant information based on *systemType*. • Credentials may be updated using the systems credentials endpoint.
bucketName	String	tapis-ds1-jdoe	• Name of bucket for an S3 system. • Required if *systemType* is S3. • Variable references: *${apiUserId}*, *${owner}*, *${tenant}*

Continued on next page

Table 1 – continued from previous page

Attribute	Type	Example	Notes
rootDir	String	$HOME	• Required if *systemType* is LINUX or *isDtn* = true. Must be an absolute path. • Serves as effective root directory when listing or moving files. • For DTN must be source location used in mount command. • Optional for an S3 system but may be used for a similar purpose. • Variable references: *${apiUserId}*, *${owner}*, *${tenant}*
port	int	22	• Port number used to access the system
useProxy	boolean	TRUE	• Indicates if system should be accessed through a proxy.
proxyHost	String		• Name of proxy host.
proxyPort	int		• Port number for *proxyHost*
dtnSystemId	String	default.corral.dtn	• An alternate system to use as a Data Transfer Node (DTN). • This system and *dtnSystemId* must have shared storage.

Continued on next page

Table 1 – continued from previous page

Attribute	Type	Example	Notes
dtnMountPoint	String	/gpfs/corral3/repl	• Mount point (aka target) used when running the mount command on this system. • Base location on this system for files transferred to *rootDir* on *dtnSystemId.*
dtnMountSourcePath	String	/gpfs/corral3/repl	• Relative path defining DTN source directory relative to rootDir on *dtnSystemId.*
isDtn	boolean	FALSE	• Indicates if system will be used as a data transfer node (DTN).
canExec	boolean		• Indicates if system will be used to execute jobs.
jobRuntimes	[Runtime]		• List of runtime environments supported by the system.
jobWorkingDir	String	HOST_EVAL($SCRATCH)	• Parent directory from which a job is run. • Relative to the effective root directory *rootDir*. • Variable references: *${apiUserId}*, *${owner}*, *${tenant}*

Continued on next page

Table 1 – continued from previous page

Attribute	Type	Example	Notes
jobEnvVariables	[String]		• Environment variables added to the shell environment in which the job is running. • Added to environment variables specified in job and application definitions. • Will overwrite job and application variables with same names. • Each string in the list must have the format *<env_name>=<env_value>*
jobMaxJobs	int		• Max total number of jobs . • Set to -1 for unlimited.
jobMaxJobsPerUser	int		• Max total number of jobs associated with a specific user. • Set to -1 for unlimited.
jobIsBatch	boolean		• Indicates if system uses a batch scheduler to run jobs.
batchScheduler	String	SLURM	• Type of scheduler used when running batch jobs. • Schedulers: SLURM

Continued on next page

Table 1 – continued from previous page

Attribute	Type	Example	Notes
batchLogicalQueues	[LogicalQueue]		• List of logical queues available on the system. • Each logical queue maps to a single HPC queue. • Multiple logical queues may be defined for each HPC queue.
batchDefaultLogical Queue	LogicalQueue		• Default logical batch queue for the system.
jobCapabilities	[Capability]		• List of additional job related capabilities supported by the system.
tags	[String]		• List of tags as simple strings.
notes	String	"{}"	• Simple metadata in the form of a Json object.
uuid	UUID		• Auto-generated by service.
created	Timestamp	2020-06-19T15:10:43Z	• When the system was created. Maintained by service.
updated	Timestamp	2020-07-04T23:21:22Z	• When the system was last updated. Maintained by service.

5.9 Searching

The service provides a way for users to search for systems based on a list of search conditions provided either as query parameters for a GET call or a list of conditions in a request body for a POST call to a dedicated search endpoint.

5.9.1 Search using GET

To search when using a GET request to the `systems` endpoint a list of search conditions may be specified using a query parameter named `search`. Each search condition must be surrounded with parentheses, have three parts separated by the character `.` and be joined using the character `~`. All conditions are combined using logical AND. The general form for specifying the query parameter is as follows:

```
?search=(<attribute_1>.<op_1>.<value_1>)~(<attribute_2>.<op_2>.<value_2>)~ ... ~(
↪<attribute_N>.<op_N>.<value_N>)
```

Attribute names are given in the table above and may be specified using Camel Case or Snake Case.

Supported operators: `eq` `neq` `gt` `gte` `lt` `lte` `in` `nin` `like` `nlike` `between` `nbetween`

For more information on search operators, handling of timestamps, lists, quoting, escaping and other general information on search please see <TBD>.

Example CURL command to search for systems that have `Test` in the id, are of type LINUX, are using a port less than `1024` and have a default authorization method of either `PKI_KEYS` or `PASSWORD`:

```
$ curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/systems?search="(id.like.
↪*Test*)~(system_type.eq.LINUX)~(port.lt.1024)~(DefaultAuthnMethod.in.PKI_KEYS,
↪PASSWORD)"
```

Notes:

- For the `like` and `nlike` operators the wildcard character `*` matches zero or more characters and `!` matches exactly one character.
- For the `between` and `nbetween` operators the value must be a two item comma separated list of unquoted values.
- If there is only one condition the surrounding parentheses are optional.
- In a shell environment the character `&` separating query parameters must be escaped with a backslash.
- In a shell environment the query value must be surrounded by double quotes and the following characters must be escaped with a backslash in order to be properly interpreted by the shell:
  - `" \ ``
- Attribute names may be specified using Camel Case or Snake Case.
- Following complex attributes not supported when searching:
  - `authnCredential` `jobRuntimes` `jobEnvVariables` `jobCapabilities` `batchLogicalQueues` `tags` `notes`

5.9.2 Dedicated Search Endpoint

The service provides the dedicated search endpoint `systems/search/systems` for specifying complex queries. Using a GET request to this endpoint provides functionality similar to above but with a different syntax. For more complex queries a POST request may be used with a request body specifying the search conditions using an SQL-like syntax.

Search using GET on Dedicated Endpoint

Sending a GET request to the search endpoint provides functionality very similar to that provided for the endpoint `systems` described above. A list of search conditions may be specified using a series of query parameters, one for each attribute. All conditions are combined using logical AND. The general form for specifying the query parameters is as follows:

```
?<attribute_1>.<op_1>=<value_1>&<attribute_2>.<op_2>=<value_2>)& ... &<attribute_N>.
↪<op_N>=<value_N>
```

Attribute names are given in the table above and may be specified using Camel Case or Snake Case.

Supported operators: `eq` `neq` `gt` `gte` `lt` `lte` `in` `nin` `like` `nlike` `between` `nbetween`

For more information on search operators, handling of timestamps, lists, quoting, escaping and other general information on search please see <TBD>.

Example CURL command to search for systems that have `Test` in the name, are of type `LINUX`, are using a port less than `1024` and have a default authorization method of either `PKI_KEYS` or `PASSWORD`:

```
$ curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/systems/search/systems?name.
↪like=*Test*\&enabled.eq=true\&system_type.eq=LINUX\&DefaultAuthnMethod.in=PKI_KEYS,
↪PASSWORD
```

Notes:

- For the `like` and `nlike` operators the wildcard character `*` matches zero or more characters and `!` matches exactly one character.
- For the `between` and `nbetween` operators the value must be a two item comma separated list of unquoted values.
- In a shell environment the character `&` separating query parameters must be escaped with a backslash.
- Attribute names may be specified using Camel Case or Snake Case.
- Following complex attributes not supported when searching:
  - `authnCredential` `jobRuntimes` `jobEnvVariables` `jobCapabilities` `batchLogicalQueues` `tags` `notes`

Search using POST on Dedicated Endpoint

More complex search queries are supported when sending a POST request to the endpoint `systems/search/systems`. For these requests the request body must contain json with a top level property name of `search`. The `search` property must contain an array of strings specifying the search criteria in an SQL-like syntax. The array of strings are concatenated to form the full search query. The full query must be in the form of an SQL-like `WHERE` clause. Note that not all SQL features are supported.

For example, to search for systems that are owned by `jdoe` and of type `LINUX` or owned by `jsmith` and using a port less than `1024` create a local file named `system_search.json` with following json:

```
{
  "search":
    [
      "(owner = 'jdoe' AND system_type = 'LINUX') OR",
      "(owner = 'jsmith' AND port < 1024)"
    ]
}
```

To execute the search use a CURL command similar to the following:

```
$ curl -X POST -H "content-type: application/json" -H "X-Tapis-Token: $JWT" https://
↪tacc.tapis.io/v3/systems/search/systems -d @system_search.json
```

Notes:

- String values must be surrounded by single quotes.
- Values for BETWEEN must be surrounded by single quotes.
- Search query parameters as described above may not be used in conjunction with a POST request.
- SQL features not supported include:
  - `IS NULL` and `IS NOT NULL`
  - Arithmetic operations
  - Unary operators
  - Specifying escape character for `LIKE` operator

Map of SQL operators to Tapis operators

Sql Operator	Tapis Operator
=	eq
<>	neq
<	lt
<=	lte
>	gt
>=	gte
LIKE	like
NOT LIKE	nlike
BETWEEN	between
NOT BETWEEN	nbetween
IN	in
NOT IN	nin

5.10 Sort, Limit and Select

When a list of Systems is being retrieved the service provides for sorting and limiting the results. When retrieving either a list of resources or a single resource the service also provides a way to *select* which fields (i.e. attributes) are included in the results. Sorting, limiting and attribute selection are supported using query parameters.

5.10.1 Selecting

When retrieving systems the fields (i.e. attributes) to be returned may be specified as a comma separated list using a query parameter named `select`. Attribute names may be given using Camel Case or Snake Case.

Notes:

- Special select keywords are supported: `allAttributes` and `summaryAttributes`
- Summary attributes include:

  - id, systemType, owner, host, effectiveUserId, defaultAuthnMethod, canExec

- By default all attributes are returned when retrieving a single resource via the endpoint systems/<system_id>.
- By default summary attributes are returned when retrieving a list of systems.
- Specifying nested attributes is not supported.
- The attribute id is always returned.

For example, to return only the attributes host and effectiveUserId the CURL command would look like this:

```
$ curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/systems?select=host,
↪effectiveUserId
```

The response should look similar to the following:


```
{
 "result": [
        {
            "id": "CSys_CltSrchGet_011",
            "host": "hostCltSrchGet_011",
            "effectiveUserId": "effUserCltSrchGet_011"
        },
        {
            "id": "CSys_CltSrchGet_012",
            "host": "hostCltSrchGet_012",
            "effectiveUserId": "effUserCltSrchGet_012"
        },
        {
            "id": "CSys_CltSrchGet_013",
            "host": "hostCltSrchGet_013",
            "effectiveUserId": "effUserCltSrchGet_013"
        }
    ],
    "status": "success",
    "message": "TAPIS_FOUND Systems found: 12 systems",
    "version": "0.0.1-SNAPSHOT",
    "metadata": {
        "recordCount": 3,
        "recordLimit": 100,
        "recordsSkipped": 0,
        "orderBy": null,
        "startAfter": null,
        "totalCount": -1
    }
}
```


5.10.2 Sorting

The query parameter for sorting is named orderBy and the value is the attribute name to sort on with an optional sort direction. The general format is <attribute_name>(<dir>). The direction may be asc for ascending or desc for descending. The default direction is ascending.

Examples:

- orderBy=id
- orderBy=id(asc)

- orderBy=name(desc),created
- orderBy=id(asc),created(desc)

5.10.3 Limiting

Additional query parameters may be used in order to limit the number and starting point for results. This is useful for implementing paging. The query parameters are:

- `limit` - Limit number of items returned. For example limit=10.
  - Use 0 or less for unlimited.
  - Default is service dependent.
- `skip` - Number of items to skip. For example skip=10.
  - May not be used with startAfter.
  - Default is 0.
- `startAfter` - Where to start when sorting. For example limit=10&orderBy=id(asc),created(desc)&startAfter=101
  - May not be used with `skip`.
  - Must also specify `orderBy`.
  - The value of `startAfter` applies to the major `orderBy` field.
  - Condition is context dependent. For ascending the condition is value > `startAfter` and for descending the condition is value < `startAfter`.

When implementing paging it is recommend to always use `orderBy` and when possible use `limit+startAfter` rather than `limit+skip`. Sorting should always be included since returned results are not guaranteed to be in the same order for each call. The combination of `limit+startAfter` is preferred because `limit+skip` is more likely to result in inconsistent results as records are added and removed. Using `limit+startAfter` works best when the attribute has a natural sequential ordering such as when an attribute represents a timestamp or a sequential ID.

5.11 Tapis Responses

For requests that return a list of resources the response result object will contain the list of resource records that match the user's query and the response metadata object will contain information related to sorting and limiting.

The metadata object will contain the following information:

- `recordCount` - Actual number of records returned.
- `recordLimit` - The limit query parameter specified in the request. -1 if query parameter was not specified.
- `recordsSkipped` - The skip query parameter specified in the request. -1 if query parameter was not specified.
- `orderBy` - The orderBy query parameter specified in the request. Empty string if query parameter was not specified.
- `startAfter` - The startAfter query parameter specified in the request. Empty string if query parameter was not specified.
- `totalCount` - Total number of records that would have been returned without a limit query parameter being imposed. -1 if total count was not computed.

For performance reasons computation of `totalCount` is only determined on demand. This is controlled by the boolean query parameter `computeTotal`. By default `computeTotal` is false.

Example query and response:

Query:

```
$ curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/systems?limit=2&
↪orderBy=id(desc)
```

Response:


```
{
 "result": [
   {
     "id": "testMin0",
     "systemType": "S3",
     "owner": "testuser",
     "host": "my.example.host",
     "defaultAccessMethod": "ACCESS_KEY",
     "canExec": false
   },
   {
     "id": "MinSystem1c",
     "systemType": "LINUX",
     "owner": "testuser",
     "defaultAccessMethod": "PASSWORD",
     "host": "data.tacc.utexas.edu",
     "canExec": true
   }
 ],
 "status": "success",
 "message": "TAPIS_FOUND Systems found: 2 systems",
 "version": "0.0.1-SNAPSHOT",
 "metadata": {
   "recordCount": 2,
   "recordLimit": 2,
   "recordsSkipped": 0,
   "orderBy": "id(desc)",
   "startAfter": null,
   "totalCount": -1
 }
```


5.11.1 Heading 2

Heading 3

Heading 4

CHAPTER 6

Files

The files service is the central point of interaction for doing all file operations in the Tapis ecosystem. Users can preform file listing, uploading, operations such as move/copy/delete and also transfer files between systems. All Tapis files APIs accept JSON as inputs.

Currently the files service includes support for S3 and SSH type file systems. Other storage systems like IRODS will be included in future releases.

6.1 Overview

All file operations act upon *Storage* systems. If you are unfamiliar with the Systems service, please refer to the *systems* section

6.1.1 Basic File Operations

File Listings

To list the files in the root directory of that system:

Using the official Tapis Python SDK:

```
t.files.listing("/")
```

Or using curl:

```
curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/files/ops/my-system/
```

And to list a sub-directory in the system, just add the path to the request:

Using CURL

```
curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/files/ops/aturing-storage/
↪subDir1/subDir2/subDir3/
```

Query Parameters

limit integer - Max number of results to return, default of 1000

offset integer - Skip the first N listings

The JSON response of the API will look something like this:


```
{
  "status": "success",
  "message": "ok",
  "result": [
      {
          "mimeType": "text/plain",
          "type": "file",
          "owner": "1003",
          "group": "1003",
          "nativePermissions": "drwxrwxr-x",
          "uri": "tapis://dev/aturing-storage/file1.txt",
          "lastModified": "2021-04-29T16:55:57Z",
          "name": "file1.txt",
          "path": "file1.txt",
          "size": 313
      },
      {
          "mimeType": "text/plain",
          "type": "file",
          "owner": "1003",
          "group": "1003",
          "nativePermissions": "-rw-rw-r--",
          "uri": "tapis://dev/aturing-storage/file2.txt",
          "lastModified": "2020-12-17T22:46:29Z",
          "name": "file2.txt",
          "path": "file2.txt",
          "size": 21
      }
  ],
  "version": "1.1-84a31617",
  "metadata": {}
```


}

Move/Copy

To move or copy a file or directory using the files service, make a PUT request with the path to the current location of the file or folder.

For example, to copy a file located at */file1.txt* to */subdir/file1.txt*

```
curl -H "X-Tapis-Token: $JWT" -X PUT -d @body.json "https://tacc.tapis.io/v3/files/
↪content/aturing-storage/file1.txt"
```

with a JSON body of

```
{
    "operation": "COPY",
    "newPath": "/subdir/file1.txt"
}
```

Delete

To delete a file or folder, just iss a DELETE request on the path to the resource

```
curl -H "X-Tapis-Token: $JWT" -X DELETE "https://tacc.tapis.io/v3/files/ops/aturing-
↪storage/file1.txt"
```

The request above would delete `file1.txt`

File Uploads

To upload a new file to the files service, just POST a file to the service. The file will be placed at the location specified in the *{path}* parameter in the request. For example, given the system *my-system*, and you want to insert the file in a folder located at */folderA/folderB/folderC*:

Using the official Tapis Python SDK:

```
with open("experiment-results.hd5", "r") as f:
    t.files.upload("my-system", "/folderA/folderB/folderC/someFile.txt", f)
```

```
curl -H "X-Tapis-Token: $JWT" -F "file=@someFile.txt" https://tacc.tapis.io/v3/files/
↪content/my-system/folderA/folderB/folderC/someFile.txt
```

Any folders that do not exist in the specified path will automatically be created.

Create a new directory

For S3 storage systems, an empty key is created ending in /

```
$ curl -H "X-Tapis-Token: $JWT" -d @body.json -X POST https://tacc.tapis.io/v3/files/
↪content/my-system/
```

with a JSON body of

```
{
    "path": "/path/to/new/directory/"
}
```

File Contents - Serving files

To return the actual contents (raw bytes) of a file (Only files can be served, not folders):

```
$ curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/files/content/my-system/
↪image.jpg > image.jpg
```

Query Parameters

startByte integer - Start at byte N of the file

count integer - Return this number of bytes after startByte

zip boolean - Zip the contents of the folder

Header Parameters

more integer - Return 1 KB chunks of UTF-8 encoded text from a file starting after page *more*. This call can be used to

page through a text based file. Note that if the contents of the file are not textual (such as an image file or other binary format) the output will be bizarre.

6.1.2 File Permissions

Permissions model - Only the system *owner* may grant or revoke permissions on a storage system. The Tapis permissions are also *not* duplicated or otherwise implemented in the underlying storage system.

Grant permissions

Lets say our user `aturing` has a storage system with ID `aturing-storage`. Alan wishes to allow his collaborator `aeinstein` to view the results of an experiment located at `/experiment1`

```
curl -H "X-Tapis-Token: $JWT" -d @body.json -X POST https://tacc.tapis.io/v3/files/
↪perms/aturing-storage/experiment1/
```

with a JSON body with the following shape:

```
{
    "username": "aeinstein",
    "permission": "READ"
}
```

Other users can also be granted permission to write to the system by granting the `MODIFY` permission. The JSON body would then be:

```
{
    "username": "aeinstein",
    "permission": "MODIFY"
}
```

Revoke permissions

Our user `aturing` now wished to revoke his former collaborators access to the folder he shared above. He can just issue a DELETE request on the path that was shared and specify the username to revoke access:

```
curl -H "X-Tapis-Token: $JWT" -X DELETE https://tacc.tapis.io/v3/files/perms/aturing-
↪storage/experiment1?username=aeinstein
```

6.1.3 Transfers

File transfers are used to move data between different storage systems, and also for bulk data operations that are too large for the REST api to perform. Transfers occur *asynchronously*, and are parallelized where possible to increase performance. As such, the order in which the files are transferred to the target system is somewhat arbitrary.

Notice in the above examples that the Files services works identically regardless of whether the source is a file or directory. If the source is a file, it will copy the file. If the source is a directory, it will recursively process the contents until everything has been copied.

When a transfer is initiated, a "Bill of materials" is created that creates a record of all the files on the target system that are to be transferred. Unless otherwise specified, all files in the bill of materials must successfully transfer for the overall transfer to be completed successfully. A transfer task has a STATUS which is updated as the transfer progresses. The states possible for a transfer are:

ACCEPTED - The initial request has been processed and saved. IN_PROGRESS - The bill of materials has been created and transfers are either in flight or awaiting resources to begin FAILED - The transfer failed. There are many reasons COMPLETED - The transfer completed successfully, all files have been transferred to the target system

Unauthenticated HTTP endpoints are also possible to use as a source for transfers. This method can be utilized to include outputs from other APIs into Tapis jobs.

Creating Transfers

Lets say our user `aturing` needs to transfer data between two systems that are registered in tapis. The source system has an id of `aturing-storage` with the results of an experiment located in directory `/experiments/experiment-1/` that should be transferred to a system with id `aturing-compute`

```
curl -H "X-Tapis-Token: $JWT" -X POST -d @body.json https://tacc.tapis.io/v3/files/
↪tranfers
```

```
{
    "tag": "An optional identifier",
    "elements": [
        {
            "sourceUri": "tapis://aturing-storage/experiments/experiment-1/",
            "destinationUri": "tapis://aturing-compute/"
        }
    ]
}
```

The request above will initiate a transfer that copies all files and folders in the `experiment-1` folder on the source system to the root directory of the destination system `aturing-compute`

HTTP Inputs

Unauthenticated HTTP endpoints can also be used as a source to a file transfer. This can be useful when, for instance, the inputs for a job to run are from a separate web service, or perhaps stored in an S3 bucket on AWS.

```
curl -H "X-Tapis-Token: $JWT" -X POST -d @body.json https://tacc.tapis.io/v3/files/
↪tranfers
```

```
{
    "tag": "An optional identifier",
    "elements": [
        {
            "sourceUri": "https://some-web-application.io/calculations/12345/",
            "destinationUri": "tapis://aturing-compute/inputs.csv"
        }
    ]
}
```

The request above will place the output of the source URI into a file called `inputs.csv` in the `aturing-compute` storage system.

Get transfer information

To retrieve information about a transfer such as its status, bytes transferred, etc just make a GET request to the transfers API with the UUID of the transfer.

```
curl -H "X-Tapis-Token: $JWT"  https://tacc.tapis.io/v3/files/tranfers/{UUID}
```

The JSON response should look something like :


```
{
    "status": "success",
    "message": "ok",
    "result": {
        "id": 1,
        "username": "aturing",
        "tenantId": "tacc",
        "tag": "some tag",
        "uuid": "b2dcf71a-bb7b-409a-8c01-1bbs97e749fb",
        "status": "COMPLETED",
        "parentTasks": [
            {
                "id": 1,
                "tenantId": "tacc",
                "username": "aturing",
                "sourceURI": "tapis://sourceSystem/file1.txt",
                "destinationURI": "tapis://destSystem/folderA/",
                "totalBytes": 100000,
                "bytesTransferred": 100000,
                "taskId": 1,
                "children": null,
                "errorMessage": null,
                "uuid": "8fdccda6-a504-4ddf-9464-7b22sa66bcc4",
                "status": "COMPLETED",
                "created": "2021-04-22T14:21:58.933851Z",
                "startTime": "2021-04-22T14:21:59.862356Z",
                "endTime": "2021-04-22T14:22:09.389847Z"
            }
        ],
        "estimatedTotalBytes": 100000,
        "totalBytesTransferred": 100000,
        "totalTransfers": 1,
        "completeTransfers": 1,
        "errorMessage": null,
        "created": "2021-04-22T14:21:58.933851Z",
        "startTime": "2021-04-22T14:21:59.838928Z",
        "endTime": "2021-04-22T14:22:09.376740Z"
    },
    "version": "1.1-094fd38d",
    "metadata": {}
}
```

CHAPTER 7

Applications

In order to run a job on a system you will need to create or have access to a Tapis **application**.

7.1 Overview

A Tapis application represents all the information required to run a Tapis job on a Tapis system and produce useful results. Each application is versioned and is associated with a specific tenant and owned by a specific user who has special privileges for the application. In order to support this purpose an application definition includes information which allows the *Jobs* service to:

- Stage input prior to launching the application
- Launch the application
- Monitor the application during execution
- Archive output after application execution

7.1.1 Dynamic Execution System Selection

Tapis supports dynamic selection of an execution system at runtime. Each Tapis system has certain capabilities inherent in the definition of the system, such as the batch scheduler type, supported container runtimes, certain information about the HPC queues, etc. Additional job related capabilities may also be included in a system definition. A job request or an application may specify a list of constraints based on these capabilities. These are used for determining eligible systems at job execution time.

7.1.2 Application Definition Creation and Validation in a Portal or Gateway

As discussed above most of the information in an application definition is in place in order to allow Tapis to stage input, execute and archive output for an application. In addition, an application definition may include information to facilitate portal and gateway developers. This information is in the form of metadata that can be associated with the file inputs and the various collections of arguments. The collections are contained in the `ParameterSet`, please see the

table below. The collections include the `appArgs`, `containerArgs` and `schedulerOptions`. The metadata for each argument includes attributes for `metaName`, `metaDescription`, `metaRequired`, and `metaKvPairs` (a list of free form key-value pairs). The intent of the `metaDescription` is to allow the application designer to document the argument in detail. The free form key-value pairs allow for specifying various validation requirements, such as argument type, maximum value or length, minimum value or length, etc.

7.2 Model

At a high level an application contain some information that is independent of the version and some information that varies by version.

7.2.1 Non-Versioned Attributes

Id A short descriptive name for the application that is unique within the tenant.

Latest Version Applications are expected to evolve over time. This is the latest version of the application. The value is updated by the service as new versions are created.

Type of application BATCH or FORK

Owner A specific user set at application creation. Default is `${apiUserId}`, the user making the request to create the application.

Enabled flag Indicates if application is currently considered active and available for use. Default is true.

Containerized flag Indicates if application has been fully containerized.

Note: NOTE: Currently only containerized applications are supported

---

7.2.2 Versioned Attributes

Version Applications are expected to evolve over time. `Id` + `version` must be unique within a tenant.

Description An optional more verbose description for the application.

Runtime Runtime to be used when executing the application. DOCKER, SINGULARITY. Default is DOCKER.

Runtime version Runtime version to be used when executing the application.

Container image Reference to be used when running the container image. Required if `containerized` is true.

Interactive flag Indicates if the application is interactive. Default is false.

Max jobs Maximum total number of jobs that can be queued or running for this application on a given execution system at a given time. Note that the execution system may also limit the number of jobs on the system which may further restrict the total number of jobs. Set to -1 for unlimited. Default is unlimited.

Max jobs per user Maximum total number of jobs associated with a specific job owner that can be queued or running for this application on a given execution system at a given time. Note that the execution system may also limit the number of jobs on the system which may further restrict the total number of jobs. Set to -1 for unlimited. Default is unlimited.

Strict file inputs flag Indicates if a job request is allowed to have unnamed file inputs. If value is true then a job request may only use the named file inputs defined in the application. See attribute *fileInputs* in the JobAttributes table. Default is false.

Job related attributes Various attributes related to job execution such as *jobDescription*, *execSystemId*, *execSystemExecDir*, *execSystemInputDir*, *execSystemLogicalQueue appArgs*, *fileInputs*, etc.

7.2.3 Required Attributes

When creating a application the required attributes are: `id`, `version`, `appType` and `containerImage`. Depending on the type of application and specific values for certain attributes there are other requirements.

The restrictions are:

- If `dynamicExecSystem` is true then `execSystemConstraints` must be specified.
- If `dynamicExecSystem` is false then `execSystemId` must be specified.
- If `archiveSystemId` is specified then `archiveSystemDir` must be specified.

7.3 Getting Started

Before going into further details about applications, here we give some examples of how to create and view applications. In the examples below we assume you are using the TACC tenant with a base URL of `tacc.tapis.io` and that you have authenticated using PySDK or obtained an authorization token and stored it in the environment variable JWT, or perhaps both.

7.3.1 Creating an application

Create a local file named `app_sample.json` with json similar to the following:

```
{
  "id":"tacc-sample-app-<userid>",
  "version":"0.1",
  "appType":"FORK",
  "description":"My sample application",
  "runtime":"DOCKER",
  "containerImage":"docker.io/hello-world:latest",
  "jobAttributes": {
    "description": "default job description",
    "execSystemId": "execsystem1"
  }
}
```

where <userid> is replaced with your user name.

Note: `execSystemId` must reference a system that exists and has `canExec` set to true.

Using PySDK:

```
import json
from tapipy.tapis import Tapis
t = Tapis(base_url='https://tacc.tapis.io', username='<userid>', password=
↪'************')
with open('app_sample.json', 'r') as openfile:
    my_app = json.load(openfile)
t.apps.createAppVersion(**my_app)
```

Using CURL:

```
$ curl -X POST -H "content-type: application/json" -H "X-Tapis-Token: $JWT" https://
↪tacc.tapis.io/v3/apps -d @app_sample.json
```

7.3.2 Viewing Applications

Retrieving details for an application

To retrieve details for a specific application, such as the one above:

Using PySDK:

```
t.apps.getAppLatestVersion(appId='tacc-sample-app-<userid>')
```

Using CURL:

```
$ curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/apps/tacc-sample-app-<userid>
```

The response should look similar to the following:


```
{
   "result": {
       "tenant": "dev",
       "id": "tacc-sample-app-<userid>",
       "version": "0.1",
       "description": "My sample application",
       "appType": "FORK",
       "owner": "<userid>",
       "enabled": true,
       "containerized": true,
       "runtime": "DOCKER",
       "runtimeVersion": null,
       "runtimeOptions": [],
       "containerImage": "docker.io/hello-world:latest",
       "maxJobs": 0,
       "maxJobsPerUser": 0,
       "strictFileInputs": false,
       "jobAttributes": {
           "description": "default job description",
           "dynamicExecSystem": false,
           "execSystemConstraints": [],
           "execSystemId": "execsystem1",
           "execSystemExecDir": null,
           "execSystemInputDir": null,
           "execSystemOutputDir": null,
           "execSystemLogicalQueue": null,
           "archiveSystemId": null,
           "archiveSystemDir": null,
           "archiveOnAppError": false,
           "parameterSet": {
               "appArgs": [],
               "containerArgs": [],
               "schedulerOptions": [],
               "envVariables": [],
               "archiveFilter": {
```


(continues on next page)

(continued from previous page)


```
                    "includes": [],
                    "excludes": [],
                    "includeLaunchFiles": true
                }
            },
            "fileInputDefinitions": [],
            "nodeCount": 1,
            "coresPerNode": 1,
            "memoryMB": 100,
            "maxMinutes": 10,
            "subscriptions": [],
            "tags": []
        },
        "tags": [],
        "notes": {},
        "uuid": "40a60a11-41fe-45ea-8674-d2cfe04992f6",
        "deleted": false,
        "created": "2021-04-22T21:30:10.590999Z",
        "updated": "2021-04-22T21:30:10.590999Z"
    },
    "status": "success",
    "message": "TAPIS_FOUND App found: tacc-sample-app-<userid>",
    "version": "0.0.1-SNAPSHOT",
    "metadata": null
}
```


Retrieving details for all applications

To see the current list of applications that you are authorized to view:

Using PySDK:

```
t.apps.getApps()
```

Using CURL:

```
$ curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/apps?select=allAttributes
```

The response should contain a list of items similar to the single listing shown above.

7.4 Minimal Definition and Restrictions

When creating an application the required attributes are: *id*, *version*, *appType* and *containerImage*. Depending on the type of application and specific values for certain attributes there are other requirements. The restrictions are:

- If *dynamicExecSystem* is true then *execSystemConstraints* is required.
- If *dynamicExecSystem* is false then *execSystemId* is required.
- If *archiveSystemId* is specified then *archiveSystemDir* is required.
- If *appType* is FORK then the following attributes may not be specified: *maxJobs*, *maxJobsPerUser*, *nodeCount*, *coresPerNode*, *memoryMB*, *maxMinutes*.

7.5 Version

Versioning scheme is at the discretion of the application author. The combination of `tenant+id+version` uniquely identifies an application in the Tapis environment. It is recommended that a two or three level form of semantic versioning be used. The fully qualified application reference within a tenant is constructed by appending a hyphen to the name followed by the version string. For example, the first two versions of an application might be myapp-0.0.1 and myapp-0.0.2. If a version is not specified when retrieving an application then by default the most recently created version of the application will be returned.

7.6 Containerized Application

An application that has been containerized is one that can be executed using a single container image. When the flag *containerized* is set to true then the attribute *containerImage* must be specified. Tapis will use the appropriate container runtime command and provide support for making the input and output directories available to the container when running the container image.

Note: NOTE: Currently only containerized applications are supported

7.7 Directory Semantics and Macros

At job submission time the Jobs service supports the use of macros based on template variables. These variables may be referenced when specifying directories in an application definition. For a full list of supported variables please see the Jobs Service. Here are some examples of variables that may be used when specifying directories for an application:

- *jobId* - The Id of the job determined at job submission.
- *jobOwner* - The owner of the job determined at job submission.
- *jobWorkingDir* - Default parent directory from which a job is run. This will be relative to the effective root directory *rootDir* on the execution system. *rootDir* and *jobWorkingDir* are attributes of the execution system.
- *HOST_EVAL($<ENV_VARIABLE>)* - The value of the environment variable *ENV_VARIABLE* when evaluated on the execution system host when logging in under the job's effective user ID. This is a dynamic value determined at job submission time. The function *HOST_EVAL()* extracts specific environment variable values for use during job setup. In particular, the TACC specific values of *$HOME*, *$WORK*, *$SCRATCH* and *$FLASH* can be referenced. The specified environment variable name is used **as-is**. It is **not** subject to macro substitution. However, the function call can have a path string appended to it, such as in *HOST_EVAL($SCRATCH)/tmp/${jobId}*, and macro substitution will be applied to the path string.

7.8 Permissions

At application creation time the owner is given full authorization. Authorizations for other users must be granted in separate API calls. Permissions may be granted and revoked through the applications API. Please note that grants and revokes through this service only impact the default role for the user. A user may still have access through permissions in another role. So even after revoking permissions through this service when permissions are retrieved the access may still be listed. This indicates access has been granted via another role.

Permissions are specified as either * for all permissions or some combination of the following specific permissions: ("READ","MODIFY","EXECUTE"). Specifying permissions in all lower case is also allowed. Having MODIFY implies READ.

7.9 Deletion

An application may be deleted. Deletion means the application is marked as deleted and is no longer available for use. It will no longer show up in searches and operations on the application will no longer be allowed. The application definition is retained for auditing purposes. Note this means that application IDs may not be re-used after deletion.

7.10 Table of Attributes

Attribute	Type	Example	Notes
tenant	String	designsafe	• Name of the tenant for which the application is defined. • *tenant* + $version* + *name* must be unique.
id	String	my-ds-app	• Name of the application. URI safe, see RFC 3986. • *tenant* + $version* + *id* must be unique. • Allowed characters: Alphanumeric [0-9a-zA-Z] and special characters [-._~].
version	String	0.0.1	• Version of the application. URI safe, see RFC 3986. • *tenant* + $version* + *id* must be unique. • Allowed characters: Alphanumeric [0-9a-zA-Z] and special characters [-._~].
description	String	A sample application	• Description
appType	enum	BATCH	• Type of application. • Types: BATCH, FORK
owner	String	jdoe	• User name of *owner*. Default is *${apiUserId}*. • Variable references: *${apiUserId}*
enabled	boolean	FALSE	• Indicates if application currently enabled for use. Default is TRUE.
containerized	boolean	TRUE	• Indicates if application has been fully containerized. Default is TRUE.

7.11 JobAttributes Table

Attribute	Type	Example	Notes
description	String		• Description to be filled in when this application is used to run a job. • Macros allow this to act as a template to be filled in at job runtime.
dynamicExecSystem	boolean		• Indicates if constraints are to be used to select an execution system. • The default is FALSE.
execSystem Constraints	[String]	**["A=aval AND",** "B=bval"]	• Capability constraints to use when dynamically searching for an execution system.
execSystemId	String		• Specific system on which the application is to be run. • Ignored if dynamicExecSystem is true.
execSystemExecDir	String		• Directory where application assets are staged. • Current working directory at application launch time. • Macro template variables such as ${jobWorkingDir} may be used. • Default is ${jobWorkingDir}/jobs/${jobId}
execSystemInputDir	String		• Directory where Tapis is to stage the inputs required by the application. • Macro template variables such as ${jobWorkingDir} may be used. • Default is

7.12 ParameterSet Table

Attribute	Type	Example	Notes
appArgs	[Arg]		• Command line arguments passed to the application. • See table below.
containerArgs	[Arg]		• Command line arguments passed to the container runtime. • See table below.
schedulerOptions	[Arg]		• Scheduler options passed to the HPC batch scheduler. • See table below.
envVariables	[String]		• Environment variables placed into the runtime environment. • Specified in the form <key>=<value> where <value> is optional.
archiveFilter	ArchiveFilter		• Sets of files to include or exclude when archiving. • Default is to include all files in *execSystemOutputDir*. • See table below.

7.13 ArchiveFilter Table

Attribute	Type	Example	Notes
includes	[String]		• Files to include when archiving after execution of the application. • excludes list has precedence.
excludes	[String]		• Files to skip when archiving after execution of the application. • excludes list has precedence.
includeLaunchFiles	boolean		• Indicates if Tapis generated launch scripts are to be included when archiving.

7.14 Arg Table

Attribute	Type	Example	Notes
value	String		• Value for the argument
metaName	String		• Identifying label associated with the argument.
metaDescription	String		•
metaRequired	boolean		• Indicates if input must be present prior to execution of the application. • Default is FALSE.
metaKvPairs	[String]		• Additional information as key-value pairs. • Each pair must be in the form <key>=<value> where <value> is optional.

7.15 FileInput Table

Attribute	Type	Example	Notes
sourceUrl	String		• Source used by the Jobs service when transferring files.
targetPath	String		• Target path used by the Jobs service when transferring files.
inPlace	boolean		• Default is FALSE.
metaName	String		• Identifying label associated with the input. Typically used during a job request.
metaDescription	String		•
metaRequired	boolean		• Indicates if input must be present prior to execution of the application. • Default is FALSE.
metaKvPairs	[String]		• Additional information as key-value pairs. • Each pair must be in the form <key>=<value> where <value> is optional.

7.16 Searching

The service provides a way for users to search for applications based on a list of search conditions provided either as query parameters for a GET call or a list of conditions in a request body for a POST call to a dedicated search endpoint.

7.16.1 Search using GET

To search when using a GET request to the `apps` endpoint a list of search conditions may be specified using a query parameter named `search`. Each search condition must be surrounded with parentheses, have three parts separated by the character `.` and be joined using the character `~`. All conditions are combined using logical AND. The general form for specifying the query parameter is as follows:

```
?search=(<attribute_1>.<op_1>.<value_1>)~(<attribute_2>.<op_2>.<value_2>)~ ... ~(
↪<attribute_N>.<op_N>.<value_N>)
```

Attribute names are given in the table above and may be specified using Camel Case or Snake Case.

Supported operators: `eq` `neq` `gt` `gte` `lt` `lte` `in` `nin` `like` `nlike` `between` `nbetween`

For more information on search operators, handling of timestamps, lists, quoting, escaping and other general information on search please see <TBD>.

Example CURL command to search for applications that have `Test` in the id, are of type FORK and allow for *maxJobs* greater than `5`:

```
$ curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/apps?search="(id.like.
↪*Test*)~(app_type.eq.FORK)~(max_jobs.gt.5)"
```

Notes:

- For the `like` and `nlike` operators the wildcard character `*` matches zero or more characters and `!` matches exactly one character.
- For the `between` and `nbetween` operators the value must be a two item comma separated list of unquoted values.
- If there is only one condition the surrounding parentheses are optional.
- In a shell environment the character `&` separating query parameters must be escaped with a backslash.
- In a shell environment the query value must be surrounded by double quotes and the following characters must be escaped with a backslash in order to be properly interpreted by the shell:
  - `" \ `
- Attribute names may be specified using Camel Case or Snake Case.
- Following complex attributes not supported when searching:
  - `jobAttributes` `tags` `notes`

7.16.2 Dedicated Search Endpoint

The service provides the dedicated search endpoint `apps/search/apps` for specifying complex queries. Using a GET request to this endpoint provides functionality similar to above but with a different syntax. For more complex queries a POST request may be used with a request body specifying the search conditions using an SQL-like syntax.

Search using GET on Dedicated Endpoint

Sending a GET request to the search endpoint provides functionality very similar to that provided for the endpoint `apps` described above. A list of search conditions may be specified using a series of query parameters, one for each attribute. All conditions are combined using logical AND. The general form for specifying the query parameters is as follows:

```
?<attribute_1>.<op_1>=<value_1>&<attribute_2>.<op_2>=<value_2>)& ... &<attribute_N>.
↪<op_N>=<value_N>
```

Attribute names are given in the table above and may be specified using Camel Case or Snake Case.

Supported operators: `eq` `neq` `gt` `gte` `lt` `lte` `in` `nin` `like` `nlike` `between` `nbetween`

For more information on search operators, handling of timestamps, lists, quoting, escaping and other general information on search please see <TBD>.

Example CURL command to search for applications that have `Test` in the id, are of type FORK and allow for *maxJobs* greater than `5`:

```
$ curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/apps/search/apps?id.
↪like=*Test*\&app_type.eq=FORK\&max_jobs.gt=5
```

Notes:

- For the `like` and `nlike` operators the wildcard character `*` matches zero or more characters and `!` matches exactly one character.
- For the `between` and `nbetween` operators the value must be a two item comma separated list of unquoted values.
- In a shell environment the character `&` separating query parameters must be escaped with a backslash.
- Attribute names may be specified using Camel Case or Snake Case.
- Following complex attributes not supported when searching:
  - `jobAttributes` `tags` `notes`

Search using POST on Dedicated Endpoint

More complex search queries are supported when sending a POST request to the endpoint `apps/search/apps`. For these requests the request body must contain json with a top level property name of `search`. The `search` property must contain an array of strings specifying the search criteria in an SQL-like syntax. The array of strings are concatenated to form the full search query. The full query must be in the form of an SQL-like `WHERE` clause. Note that not all SQL features are supported.

For example, to search for apps that are owned by `jdoe` and of type `FORK` or owned by `jsmith` and allow for *maxJobs* less than `5` create a local file named `app_search.json` with following json:

```
{
  "search":
    [
      "(owner = 'jdoe' AND app_type = 'FORK') OR",
      "(owner = 'jsmith' AND max_jobs < 5)"
    ]
}
```

To execute the search use a CURL command similar to the following:

```
$ curl -X POST -H "content-type: application/json" -H "X-Tapis-Token: $JWT" https://
↪tacc.tapis.io/v3/apps/search/apps -d @app_search.json
```

Notes:

- String values must be surrounded by single quotes.

- Values for BETWEEN must be surrounded by single quotes.
- Search query parameters as described above may not be used in conjunction with a POST request.
- SQL features not supported include:
  - `IS NULL` and `IS NOT NULL`
  - Arithmetic operations
  - Unary operators
  - Specifying escape character for `LIKE` operator

Map of SQL operators to Tapis operators

Sql Operator	Tapis Operator
=	eq
<>	neq
<	lt
<=	lte
>	gt
>=	gte
LIKE	like
NOT LIKE	nlike
BETWEEN	between
NOT BETWEEN	nbetween
IN	in
NOT IN	nin

7.17 Sort, Limit and Select

When a list of applications is being retrieved the service provides for sorting and limiting the results. When retrieving either a list of resources or a single resource the service also provides a way to *select* which fields (i.e. attributes) are included in the results. Sorting, limiting and attribute selection are supported using query parameters.

7.17.1 Selecting

When retrieving applications the fields (i.e. attributes) to be returned may be specified as a comma separated list using a query parameter named `select`. Attribute names may be given using Camel Case or Snake Case.

Notes:

- Special select keywords are supported: `allAttributes` and `summaryAttributes`
- Summary attributes include:
  - `id`, `version`, `appType`, `owner`
- By default all attributes are returned when retrieving a single resource via the endpoint apps/<app_id>.
- By default summary attributes are returned when retrieving a list of applications.
- Specifying nested attributes is not supported.
- The attribute `id` is always returned.

For example, to return only the attributes `version` and `containerImage` the CURL command would look like this:

```
$ curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/apps?select=version,
↪containerImage
```

The response should look similar to the following:


```
{
   "result": [
        {
            "id": "TestApp1",
            "version": "0.0.1",
            "containerImage": "containterimage1"
        },
        {
            "id": "JobApp1",
            "version": "0.0.1",
            "containerImage": "containterimage1"
        },
        {
            "id": "JobAppWithInput",
            "version": "0.0.1",
            "containerImage": "containterimage1"
        },
        {
            "id": "SleepSeconds",
            "version": "0.0.1",
            "containerImage": "tapis/testapps:main"
        }
   ],
   "status": "success",
   "message": "TAPIS_FOUND Apps found: 11 applications",
   "version": "0.0.1-SNAPSHOT",
   "metadata": {
        "recordCount": 4,
        "recordLimit": 100,
        "recordsSkipped": 0,
        "orderBy": null,
        "startAfter": null,
        "totalCount": -1
   }
}
```


7.17.2 Sorting

The query parameter for sorting is named `orderBy` and the value is the attribute name to sort on with an optional sort direction. The general format is `<attribute_name>(<dir>)`. The direction may be `asc` for ascending or `desc` for descending. The default direction is ascending.

Examples:

- orderBy=id
- orderBy=id(asc)
- orderBy=name(desc),created
- orderBy=id(asc),created(desc)

7.17.3 Limiting

Additional query parameters may be used in order to limit the number and starting point for results. This is useful for implementing paging. The query parameters are:

- `limit` - Limit number of items returned. For example limit=10.
  - Use 0 or less for unlimited.
  - Default is service dependent.
- `skip` - Number of items to skip. For example skip=10.
  - May not be used with startAfter.
  - Default is 0.
- `startAfter` - Where to start when sorting. For example limit=10&orderBy=id(asc),created(desc)&startAfter=101
  - May not be used with `skip`.
  - Must also specify `orderBy`.
  - The value of `startAfter` applies to the major `orderBy` field.
  - Condition is context dependent. For ascending the condition is value > `startAfter` and for descending the condition is value < `startAfter`.

When implementing paging it is recommend to always use `orderBy` and when possible use `limit+startAfter` rather than `limit+skip`. Sorting should always be included since returned results are not guaranteed to be in the same order for each call. The combination of `limit+startAfter` is preferred because `limit+skip` is more likely to result in inconsistent results as records are added and removed. Using `limit+startAfter` works best when the attribute has a natural sequential ordering such as when an attribute represents a timestamp or a sequential ID.

7.18 Tapis Responses

For requests that return a list of resources the response result object will contain the list of resource records that match the user's query and the response metadata object will contain information related to sorting and limiting.

The metadata object will contain the following information:

- `recordCount` - Actual number of records returned.
- `recordLimit` - The limit query parameter specified in the request. -1 if query parameter was not specified.
- `recordsSkipped` - The skip query parameter specified in the request. -1 if query parameter was not specified.
- `orderBy` - The orderBy query parameter specified in the request. Empty string if query parameter was not specified.
- `startAfter` - The startAfter query parameter specified in the request. Empty string if query parameter was not specified.
- `totalCount` - Total number of records that would have been returned without a limit query parameter being imposed. -1 if total count was not computed.

For performance reasons computation of `totalCount` is only determined on demand. This is controlled by the boolean query parameter `computeTotal`. By default `computeTotal` is false.

Example query and response:

Query:

```
$ curl -H "X-Tapis-Token: $JWT" https://tacc.tapis.io/v3/apps?limit=2&orderBy=id(desc)
```

Response:


```
{
   "result": [
       {
           "id": "TestApp1",
           "version": "0.0.1",
           "appType": "BATCH",
           "owner": "testuser2"
       },
       {
           "id": "tacc-sample-app",
           "version": "0.1",
           "appType": "FORK",
           "owner": "testuser2"
       }
   ],
   "status": "success",
   "message": "TAPIS_FOUND Apps found: 2 applications",
   "version": "0.0.1-SNAPSHOT",
   "metadata": {
       "recordCount": 2,
       "recordLimit": 2,
       "recordsSkipped": 0,
       "orderBy": "id(desc)",
       "startAfter": null,
       "totalCount": -1
   }
}
```


7.18.1 Heading 2

Heading 3

Heading 4

CHAPTER 8

Jobs

8.1 Introduction to Jobs

The Tapis v3 Jobs service is specialized to run containerized applications on any host that supports container runtimes. Currently, Docker and Singularity containers are supported. The Jobs service uses the Systems, Apps, Files and Security Kernel services to process jobs.

8.1.1 Implementation Status

The following table describes the current state of the Beta release of Jobs. All UrlPaths shown start with /v3/jobs. The unauthenticated health check, ready and hello APIs do not require a Tapis JWT in the request header.

Name	Method	UrlPath	Status
Submit	POST	/submit	Implemented
Resubmit	POST	/{jobUuid}/resubmit	Implemented
Get	GET	/{jobUuid}	Implemented
Get Status	GET	/{jobUuid}/status	Implemented
Health Check	GET	/healthcheck	Implemented
Ready	GET	/ready	Implemented
Hello	GET	/hello	Implemented

8.1.2 Job Processing Overview

Before discussing the details of how to construct a job request, we take this opportunity to describe overall lifecycle of a job. When a job request is recieved as the payload of an POST call, the following steps are taken:

1. **Request authorization** - The tenant, owner, and user values from the request and Tapis JWT are used to authorize access to the application, execution system and, if specified, archive system.

2. **Request validation** - Request values are checked for missing, conflicting or improper values; all paths are assigned; required paths are created on the execution system; and macro substitution is performed to finalize all job parameters.
3. **Job creation** - A Tapis job object is written to the database.
4. **Job queuing** - The Tapis job is queue on an internal queue serviced by one or more Job Worker processes.
5. **Response** - The initial Job object is sent back to the caller in the response. This ends the synchronous portion of job submission.

Once a response to the submission request is sent to the caller, job processing proceeds asynchronously. Job worker processes read jobs from their work queues. The number of workers and queues is limited only by hardware resource constraints. Each job is assigned a worker thread. This thread shepards a job through its lifecycle until the job completes, fails or becomes blocked due to a transient resource constraint. The job lifecycle is reflected in the *Job Status* and generally progresses as follows:

```
a) Stage inputs to execution system
b) Stage application artifacts to execution system
c) Queue or invoke job on execution system
d) Monitor job until it terminates
e) Collect job exit code
f) Archive job output
```

8.1.3 Simple Job Submission Example

The POST payload for the simplest job submission request looks like this:

```
{
 "name": "myJob"
 "appId": "myApp"
 "appVersion": "1.0"
}
```

In this example, all input and output directories are either specified in the *myApp* definition or are assigned their default values. Currently, the execution system on which an application runs must be specified in either the application definition or job request. Our example assumes that *myApp* assigns the execution system. Future versions of the Jobs service will support dynamic execution system selection.

An archive system can also be specified in the application or job request; the default is to be the same as the execution system.

8.2 The Job Submission Request

A job submission request must contain the name, appId and appVersion values as shown in the *Simple Job Submission Example*. Those values are marked *Required* in the list below, a list of all possible values allowed in a submission request. If a parameter has a default value, that value is also shown.

In addition, some parameters can inherit their values from the application or system definitions as discussed in *Parameter Precedence*. These parameters are marked *Inherit*. Parameters that merge inherited values (rather than override them) are marked *InheritMerge*.

Parameters that do not need to be set are marked *Not Required*. Finally, parameters that allow macro substitution are marked *MacroEnabled* (see *Macro Substitution* for details).

name The user chosen name of the job. *MacroEnabled*, *Required.*

appId The Tapis application to execute. *Required.*

appVersion The version of the application to execute. *Required.*

owner User ID under which the job runs. Administrators can designate a user other than themselves.

tenant Tenant of job owner. Default is job owner's tenant.

description Human readable job description. *MacroEnabled*, *Not Required*

archiveOnAppError Whether archiving should proceed even when the application reports an error. Default is *true*.

dynamicExecSystem Whether the best fit execution system should be chosen using *execSystemConstraints*. Default is *false*.

execSystemId Tapis execution system ID. *Inherit*.

execSystemExecDir Directory into which application assets are staged. *Inherit*, see *Directories* for default.

execSystemInputDir Directory into which input files are staged. *Inherit*, see *Directories* for default.

execSystemOutputDir Directory into which the application writes its output. *Inherit*, see *Directories* for default.

execSystemLogicalQueue Tapis-defined queue that corresponds to a batch queue on the execution system. *Inherit* when applicable.

archiveSystemId Tapis archive system ID. *Inherit*, defaults to *execSystemId*.

archiveSystemDir Directory into which output files are archived after application execution. *Inherit*, see *Directories* for default.

nodeCount Number of nodes required for application execution. *Inherit*, default is 1.

coresPerNode Number of cores to use on each node. *Inherit*, default is 1.

memoryMB Megabytes of memory to use on each node. *Inherit*, default is 100.

maxMinutes Maximum number of minutes allowed for job execution. *Inherit*, default is 10.

fileInputs Input files that need to be staged for the application. *InheritMerge*.

parameterSet Runtime parameters organized by category. *Inherit*.

execSystemConstraints Constraints applied against execution system capabilities to validate application/system compatibility. *InheritMerge*.

subscriptions Subscribe to the job's events. *InheritMerge*.

tags An array of user-chosen strings that are associated with a job. *InheritMerge*.

The following subsections discuss the meaning and usage of each of the parameters available in a job request. The schema and its referenced library comprise the actual JSON schema definition for job requests.

8.2.1 Parameter Precedence

The runtime environment of a Tapis job is determined by values in system definitions, the app definition and the job request, in low to high precedence order as listed. Generally speaking, for values that can be assigned in multiple definitions, the values in job requests override those in app definitions, which override those in system definitions. There are special cases, however, where the values from different definitions are merged.

See the jobs/apps/systems parameter matrix for a detailed description of how each parameter is handled.

8.2.2 Directories

The execution and archive system directories are calculated before the submission response is sent. This calculation can include the use of macro definitions that get replaced by values at submission request time. The *Macro Substitution* section discusses what macro defintions are available and how substitution works. In this section, we document the default directory assignments which may include macro definitions.

Directory Definitions

The directories assigned when a system is defined:

```
rootDir - the root of the file system that is accessible through this Tapis system.
jobWorkingDir - the default directory for temporary files used or created during job␣
↪execution.
dtnMountPoint - the path relative to the execution system's rootDir where the DTN␣
↪file system is mounted.
```

An execution system may define a *Data Transfer Node* (DTN). A DTN is a high throughput node used to stage job inputs and to archive job outputs. The goal is to improve transfer performance. The execution system mounts the DTN's file system at the *dtnMountPoint* so that executing jobs have access to its data, but Tapis will connect to the DTN rather than the execution system during transfers. See *Data Transfer Nodes* for details.

The directories assigned in application definitions and/or in a job submission requests:

```
execSystemExecDir
execSystemInputDir
execSystemOutputDir
archiveSystemDir
```

Directory Assignments

The rootDir and jobWorkingDir are always assigned upon system creation, so they are available for use as macros when assigning directories in applications or job submission requests.

When a job request is submitted, each of the job's four execution and archive system directories are assigned as follows:

1. If the job submission request assigns the directory, that value is used. Otherwise,
2. If the application definition assigns the directory, that value is used. Otherwise,
3. The default values shown below are assigned:

```
No DTN defined:
  execSystemExecDir:    ${jobWorkingDir}/jobs/${jobUUID}
  execSystemInputDir:   ${jobWorkingDir}/jobs/${jobUUID}
  execSystemOutputDir:  ${jobWorkingDir}/jobs/${jobUUID}/output
  archiveSystemDir:     /jobs/${JobUUID}/archive                (if archiveSystemId␣
↪is set)
DTN defined:
  execSystemExecDir:    ${dtnMountPoint}/jobs/${jobUUID}
  execSystemInputDir:   ${dtnMountPoint}/jobs/${jobUUID}
  execSystemOutputDir:  ${dtnMountPoint}/jobs/${jobUUID}/output
  archiveSystemDir:     ${dtnMountPoint}/jobs/${JobUUID}/archive (if archiveSystemId␣
↪is set)
```

8.2.3 FileInputs

The *fileInputs* in application definitions are merged with those in job submission requests to produce a complete list of input files that need to be staged for a job. The fileInputs array contains elements that conform to the following JSON schema.

```
"InputSpec": {
    "$comment": "Used to specify file inputs on Jobs submission requests",
    "type": "object",
        "properties": {
            "sourceUrl":  {"type": "string", "minLength": 1, "format": "uri"},
            "targetPath": {"type": "string", "minLength": 0},
            "inPlace":    {"type": "boolean"},
            "meta":       {"type": "object", "$ref": "#/$defs/ArgMetaSpec"}
        },
    "required": ["sourceUrl"],
    "additionalProperties": false
}
```

Since all input directories or files are staged to the *execSystemInputDir*, the only required field is the *sourceUrl*. Any URL protocol accepted by the Tapis Files service can be used here. The most common protocols used are tapis, http, and https. The standard tapis URL format is *tapis://<tapis-system>/<path>*; please see the Files service for the complete list of supported protocols.

If provided, the *targetPath* indicates a path relative to the *execSystemInputDir* into which the input is copied. When not provided, the directory or file named in *sourceUrl* is copied directly into *execSystemInputDir*.

The *inPlace* value defaults to false when not provided. When true, it instructs the Jobs service to **not** copy the input. This setting is used to indicate that the input has already been put in place in the *execSystemInputDir* subtree by some means outside of Tapis, so no copying is needed. The use of *inPlace* documents all inputs, even those that do not need to be transferred.

See the *ArgMetaSpec* for a discussion of the *meta* field, which allows one to name the input, designate the input as optional, and attach arbitrary key/value pairs.

8.2.4 ParameterSet

The job *parameterSet* argument is comprised of these objects:

Name	JSON Schema Type	Description
appArgs	*ArgSpec* array	Arguments passed to user's application
containerArgs	*ArgSpec* array	Arguments passed to container runtime
schedulerOptions	*ArgSpec* array	Arguments passed to HPC batch scheduler
envVariables	*KeyValuePair* array	Environment variables injected into application container
archiveFilter	object	File archiving selector

Each of these objects can be specifed in Tapis application definitions and/or in job submission requests. In addition, the execution system can also specify environment variable settings.

appArgs

Specify one or more command line arguments for the user application using the *appArgs* parameter. Arguments specified in the application definition are appended to those in the submission request. Metadata can be attached to any argument.

containerArgs

Specify one or more command line arguments for the container runtime using the *containerArgs* parameter. Arguments specified in the application definition are appended to those in the submission request. Metadata can be attached to any argument.

schedulerOptions

Specify HPC batch scheduler arguments for the container runtime using the *schedulerOptions* parameter. Arguments specified in the application definition are appended to those in the submission request. The arguments for each scheduler are passed using that scheduler's conventions. Metadata can be attached to any argument.

envVariables

Specify key/value pairs that will be injected as environment variables into the application's container when it's launched. Key/value pairs specified in the execution system definition, application definition, and job submission request are aggregated using precedence ordering (system < app < request) to resolve conflicts.

archiveFilter

The *archiveFilter* conforms to this JSON schema:

```
"archiveFilter": {
   "type": "object",
   "properties": {
      "includes": {"type": "array", "items": {"type": "string", "minLength": 1},
↪"uniqueItems": true},
      "excludes": {"type": "array", "items": {"type": "string", "minLength": 1},
↪"uniqueItems": true},
      "includeLaunchFiles": {"type": "boolean"}
   },
   "additionalProperties": false
}
```

An *archiveFilter* can be specified in the application definition and/or the job submission request. The *includes* and *excludes* arrays are merged by appending entries from the application definition to those in the submission request.

The *excludes* filter is applied first, so it takes precedence over *includes*. If *excludes* is empty, then no output file or directory will be explicitly excluded from archiving. If *includes* is empty, then all files in *execSystemOutputDir* will be archived unless explicitly excluded. If *includes* is not empty, then only files and directories that match an entry and not explicitly excluded will be archived.

Each *includes* and *excludes* entry is a string, a string with wildcards or a regular expression. Entries represent directories or files. The wildcard semantics are that of glob (*), which is commonly used on the command line. Tapis implements Java glob semantics. To filter using a regular expression, construct the pattern using Java regex semantics and then preface it with **REGEX:** (case sensitive). Here are examples of globs and regular expressions that could appear in a filter:

```
"myfile.*"
"*2021-*-events.log"
"REGEX:^[\\p{IsAlphabetic}\\p{IsDigit}_\\.\\-]+$"
"REGEX:\\s+"
```

When *includeLaunchFiles* is true (the default), then the script (*tapisjob.sh*) and environment (*tapisjob.env*) files that Tapis generates in the *execSystemExecDir* are also archived. These launch files provide valuable information about how a job was configured and launched, so archiving them can help with debugging and improve reproducibility. Since these files may contain application secrets, such database passwords or other credentials, care must be taken to not expose private data through archiving.

If no filtering is specified at all, then all files in *execSystemOutputDir* and the launch files are archived.

8.2.5 ExecSystemConstraints

Not implementated yet.

8.2.6 Subscriptions

Not implementated yet.

8.2.7 Shared Components

ArgSpec

The JSON schema for defining elements in various *ParameterSet* components is below.

```
"ArgSpec": {
    "$comment": "Used to specify parameters on Jobs submission requests",
    "type": "object",
        "properties": {
            "arg":  {"type": "string", "minLength": 1},
            "meta": {"type": "object", "$ref": "#/$defs/ArgMetaSpec"}
        },
    "required": ["arg"],
    "additionalProperties": false
}
```

The required *arg* value is an arbitrary string and is used as-is. See the *ArgMetaSpec* for a discussion of the *meta* field, which allows one to name arguments, designate them as optional, and attach arbitrary key/value pairs to them.

ArgMetaSpec

The JSON schema for metadata objects used in *FileInputs* and other job parameters is below.

```
"ArgMetaSpec": {
    "$comment": "An open-ended way to name and annotate arguments",
    "type": "object",
        "properties": {
            "description": {"type": "string", "minLength": 1, "maxLength": 8096},
            "name":        {"type": "string", "minLength": 1},
            "required":    {"type": "boolean"},
            "kv":          {"type": "array",
                            "items": {"$ref": "#/$defs/KeyValuePair"},
                            "uniqueItems": true}
        },
     "required": ["name", "required"],
```

(continues on next page)

(continued from previous page)

```
    "additionalProperties": false
}
```

The *ArgMetaSpec* is always a child its enclosing job parameter. The *ArgMetaSpec* requires that a name be assigned it parent and that whether the parent parameter is required or not. Optionally, a description and a map of key/value strings can be included. The complete *ArgMetaSpec* object is saved in the job, so the key/value pairs can be used to pass arbitrary information to any program that queries the job. For example, a web application might submit a job request and embed display information in the metadata for use whenever the job is queried.

KeyValuePair

The JSON schema for defining key/value pairs of strings in various *ParameterSet* components is below.

```
"KeyValuePair": {
    "$comment": "A simple key/value pair",
    "type": "object",
        "properties": {
            "key":   {"type": "string", "minLength": 1},
            "value": {"type": "string", "minLength": 0}
        },
     "required": ["key", "value"],
     "additionalProperties": false
}
```

Both the *key* and *value* are required, though the *value* can be an empty string.

8.3 Job Execution

8.3.1 Environment Variables

The following standard environment variables are passed into each application container run by Tapis as long as they have been assigned a value.

```
_tapisAppId - Tapis app ID
_tapisAppVersion - Tapis app version
_tapisArchiveOnAppError - true means archive even if the app returns a non-zero exit␣
↪code
_tapisArchiveSystemDir - the archive system directory on which app output is archived
_tapisArchiveSystemId - Tapis system used for archiving app output
_tapisCoresPerNode - number of cores used per node by app
_tapisDtnMountPoint - the mountpoint on the execution system for the source DTN␣
↪directory
_tapisDtnMountSourcePath - the directory exported by the DTN and mounted on the␣
↪execution system
_tapisDtnSystemId - the Data Transfer Node system ID
_tapisDynamicExecSystem - true if dynamic system selection was used
_tapisEffeciveUserId - the user ID under which the app runs
_tapisExecSystemExecDir - the exec system directory where app artifacts are staged
_tapisExecSystemHPCQueue - the actual batch queue name on an HPC host
_tapisExecSystemId - the Tapis system where the app runs
```

(continues on next page)

(continued from previous page)

```
_tapisExecSystemInputDir - the exec system directory where input files are staged
_tapisExecSystemLogicalQueue - the Tapis queue definition that specifies an HPC queue
_tapisExecSystemOutputDir - the exec system directory where the app writes its output
_tapisJobCreateDate - ISO 8601 date, example: 2021-04-26Z
_tapisJobCreateTime - ISO 8601 time, example: 18:44:55.544145884Z
_tapisJobCreateTimestamp - ISO 8601 timestamp, example: 2021-04-26T18:44:55.544145884Z
_tapisJobName - the user-chosen name of the Tapis job
_tapisJobOwner - the Tapis job's owner
_tapisJobUUID - the UUID of the Tapis job
_tapisJobWorkingDir - exec system directory that the app should use for temporary␣
↪files
_tapisMaxMinutes - the maximum number of minutes allowed for the job to run
_tapisMemoryMB - the memory required per node by the app
_tapisNodes - the number of nodes on which the app runs
_tapisSysBatchScheduler - the HPC scheduler on the execution system
_tapisSysBucketName - an object store bucket name
_tapisSysHost - the IP address or DNS name of the exec system
_tapisSysRootDir - the root directory on the exec system
_tapisTenant - the tenant in which the job runs
```

8.3.2 Macro Substitution

Tapis defines macros or template variables that get replaced with actual values at well-defined points during job creation. The act of replacing a macro with a value is often called macro substitution or macro expansion. The complete list of Tapis macros can be found at JobTemplateVariables.

There is a close relationship between these macro definitions and the Tapis environment variables just discussed: Macros that have values assigned are passed as environment variables into application containers. This makes macros used during job creation available to applications at runtime.

Most macro definitions are *ground* definitions because their values do not depend on any other macros. On the other hand, *derived* macro definitions can include other macro definitions. For example, in *Directory Assignments* we that that the default input file directory is constructed with two macro definitions:

```
execSystemInputDir = ${jobWorkingDir}/jobs/${jobUUID}
```

Macro values are referenced using the ${macro-name} notation. Since derived macro definitions reference other macros, there is the possibility of circular references. Tapis detects these errors and aborts job creation.

Below is the complete, ordered list of derived macros. Each macro in the list can be defined using any ground macro and any macro that preceeds it in the list. Result are undefined if a derived macro references a macro that follows it in the derived list.

1. JobName
2. JobWorkingDir
3. ExecSystemInputDir
4. ExecSystemExecDir
5. ExecSystemOutputDir
6. ArchiveSystemDir

Finally, macro substitution is applied to the job *description* field, whether the description is specified in an application or a submission request.

Macro Functions

Directory assignments in systems, applications and job requests can also use the **HOST_EVAL($var)** function at the beginning of their path assignments. This function dynamically extracts the named environment variable's value from an execution or archive host *at the time the job request is made*. Specifically, the environment variable's value is retrieved by logging into the host as the Job owner and issuing "echo $var". The example in *Data Transfer Nodes* uses this function.

To increase application portability, an optional default value can be passed into the **HOST_EVAL** function. The function's complete signature with the optional path parameter is:

HOST_EVAL($VAR, path)

If the environment variable VAR does not exist on the host, then the literal path parameter is returned by the function. This added flexibility allows applications to run in different environments, such as on TACC HPC systems that automatically expose certain environment variables and VMs that might not. If the environment variable does not exist and no optional path parameter is provided, the job fails due to invalid input.

8.3.3 Job Status

The list below contains all possible states of a Tapis job, which are indicated in the *status* field of a job record. The initial state is PENDING. Terminal states are FINISHED, CANCELLED and FAILED. The BLOCKED state indicates that the job is recovering from a resource constraint, network problem or other transient problem. When the problem clears, the job will restart from the state in which blocking occurred.

```
PENDING - Job processing beginning
PROCESSING_INPUTS - Identifying input files for staging
STAGING_INPUTS - Transferring job input data to execution system
STAGING_JOB - Staging runtime assets to execution system
SUBMITTING_JOB - Submitting job to execution system
QUEUED - Job queued to execution system queue
RUNNING - Job running on execution system
ARCHIVING - Transferring job output to archive system
BLOCKED - Job blocked
PAUSED - Job processing suspended
FINISHED - Job completed successfully
CANCELLED - Job execution intentionally stopped
FAILED - Job failed
```

Normal processing of a successfully executing job proceeds as follows:

```
PENDING->PROCESSING_INPUTS->STAGING_INPUTS->STAGING_JOB->SUBMITTING_JOB->
  QUEUED->RUNNING->ARCHIVING->FINISHED
```

8.3.4 Notifications

Not implemented yet.

8.3.5 Dynamic Execution System Selection

Not implementated yet.

8.3.6 Data Transfer Nodes

A Tapis system can be designated as a Data Transfer Node (DTN) as part of its definition. When an execution system specifies DTN usage in its definition, then the Jobs service will use the DTN to stage input files and archive output files.

The DTN usage pattern is effective when (1) the DTN has high performance network and storage capabilities, and (2) an execution system can mount the DTN's file system. In this situation, bulk data transfers performed by Jobs benefit from the DTN's high performance capabilities, while applications continue to access their execution system's files as usual. From an application's point of view, its data are simply where they are expected to be, though they may have gotten there in a more expeditious manner.

DTN usage requires the coordinated configuration of a DTN, an execution system and a job. In addition, outside of Tapis, a system administrator must mount the exported DTN file system at the expected mountpoint on an execution system. We use the example below to illustrate DTN configuration and usage.

```
System: ds-exec
  rootDir: /execRoot
  dtnMountSourcePath: tapis://corral-dtn/
  dtnMountPoint: /corral-repl
  jobWorkingDir: HOST_EVAL($SCRATCH)

System: corral-dtn
  host: cic-dtn01
  isDtn: true
  rootDir: /gpfs/corral3/repl

Job Request effective values:
  execSystemId:         ds-exec
  execSystemExecDir:    ${jobWorkingDir}/jobs/${jobUUID}
  execSystemInputDir:   ${dtnMountPoint}/projects/NHERI/shared/{$jobOwner}/jobs/$
↪{jobUUID}
  execSystemOutputDir:  ${dtnMountPoint}/projects/NHERI/shared/{$jobOwner}/jobs/$
↪{jobUUID}/output

NFS Mount on ds-exec (done outside of Tapis):
  mount -t nfs cic-dtn01:/gpfs/corral3/repl /execRoot/corral-repl
```

The example execution system, **ds-exec**, defines two DTN related values (both required to configure DTN usage):

dtnMountSourcePath The tapis URL specifying the exported DTN path; the path is relative to the DTN system's rootDir (which is just "/" in this example).

dtnMountPoint The path relative to the execution system's rootDir where the DtnMountSourcePath is mounted.

The execution system's jobWorkingDir is defined to be the runtime value of the $SCRATCH environment variable; its rootDir is defined at /execRoot.

The Tapis DTN system, **corral-dtn**, host machine is cic-dtn01. The DTN's rootDir (/gpfs/corral3/repl) is the directory prefix used on all mounts. Mounting takes place outside of Tapis by system administrators. The actual NFS mount command has this general format:

```
mount -t nfs <dtn_host>:/<dtn_root_dir>/<path> <exec_system_mount_point>
```

The Job Request effective values depend on the DTN configuration are also shown. These values could have been set in the application definition, the job request or in both. Values set in the job request are given priority. The execSystemId refers to the **ds-exec** system, which in this case specifies a DTN.

Continuing with the above example, let's say user *Bud* issues an Opensees job request that creates a job with id 123. The Jobs service will stage the application's input files using the DTN. The transfer request to the Files service will

write to this target URL:

tapis://corral-dtn/gpfs/corral3/repl/projects/NHERI/shared/Bud/jobs/123

This is the standard tapis URL format: tapis://<tapis-system>/<path>. After inputs are staged, the Job service will inject this environment variable value (among others) into the launched job's container:

execSystemInputDir=/corral-repl/projects/NHERI/shared/Bud/jobs/123

Since **ds-exec** mounts the corral root directory, the files staged to corral /gpfs/corral3/repl are accessible at execSystemInputDir on **ds-exec**, relative to rootDir /execRoot. A similar approach would be used to transfer files to an archive system using the DTN, except this time **corral-dtn** is the source of the file transfers rather than the target.

8.4 Container Runtimes

The Tapis v3 Jobs service currently supports Docker and Singularity containers run natively (i.e., not run using a batch scheduler like Slurm). In general, Jobs launches an application's container on a remote system, monitors the container's execution, and captures the application's exit code after it terminates. Jobs uses SSH to connect to the execution system to issue Docker, Singularity or native operating system commands.

To launch a job, the Jobs service creates a bash script, **tapisjob.sh**, with the runtime-specific commands needed to execute the container. This script references **tapisjob.env**, a file Jobs creates to pass environment variables to application containers. Both files are staged in the job's execSystemExecDir and, by default, are archived with job output on the archive system. See *archiveFilter* to override this default behavior, especially if archives will be shared and the scripts pass sensitive information into containers.

8.4.1 Docker

To launch a Docker container, the Jobs service will SSH to the target host and issue a command using this template:

```
docker run [docker options] image[:tag|@digest] [application args]
```

1. docker options: (optional) user-specified arguments passed to docker
2. image: (required) user-specified docker application image
3. application arguments: (optional) user-specified command line arguments passed to the application

The docker run-command options *–cidfile*, *-d*, *-e*, *–env*, *–name*, *–rm*, and *–user* are reserved for use by Tapis. Most other Docker options are available to the user. The Jobs service implements these calling conventions:

1. The container name is set to the job UUID.
2. The container's user is set to the user ID used to establish the SSH session.
3. The container ID file is specified as <*jobUUID*>*.cid* in the execSystemExecDir, i.e., the directory from which the container is launched.
4. The *-rm* option is always set to remove the container after execution.

Volume Mounts

In addition to the above conventions, bind mounts are used to mount the execution system's standard Tapis directories at the same locations in every application container.

```
execSystemExecDir   on host is mounted at /TapisExec in the container.
execSystemInputDir  on host is mounted at /TapisInput in the container.
execSystemOutputDir on host is mounted at /TapisOutput in the container.
```

8.4.2 Singularity

Tapis provides two distinct ways to launch a Singularity containers, using *singluarity instance start* or *singularity run*.

Singularity Start

Singularity's support for detached processes and services is implemented natively by its instance start, stop and list commands. To launch a container, the Jobs service will SSH to the target host and issue a command using this template:

```
singularity instance start [singularity options] <image id> [application arguments]
↪<job uuid>
```

where:

1. singularity options: (optional) user-specified argument passed to singularity start
2. image id: (required) user-specified singularity application image
3. application arguments: (optional) user-specified command line arguments passed to the application
4. job uuid: the job uuid used to name the instance (always set by Jobs)

The singularity options *–pidfile*, *–env* and *–name* are reserved for use by Tapis. Users specify the environment variables to be injected into their application containers via the *envVariables* parameter. Most other singularity options are available to users.

Jobs will then issue *singularity instance list* to obtain the container's process id (PID). Jobs determines that the application has terminated when the PID is no longer in use by the operating system.

By convention, Jobs will look for a **tapisjob.exitcode** file in the Job's output directory after containers terminate. If found, the file should contain only the integer code the application reported when it exited. If not found, Jobs assumes the application exited normally with a zero exit code.

Finally, Jobs issues a *singularity instance stop <job uuid>* to clean up the singularity runtime environment and terminate all processes associated with the container.

Singularity Run

Jobs also supports a more do-it-yourself approach to running containers on remote system using singularity run. To launch a container, the Jobs service will SSH to the target host and issue a command using this template:

```
nohup singularity run [singularity options.] <image id> [application arguments] >␣
↪tapisjob.out 2>&1 &
```

where:

1. nohup: allows the background process to continue running even if the SSH session ends.
2. singularity options: (optional) user-specified arguments passed to singularity run.
3. image id: (required) user-specified singularity application image.
4. application arguments: (optional) user-specified command line arguments passed to the application.

5. redirection: stdout and stderr are redirected to **tapisjob.out** in the job's output directory.

The singularity *–env* option is reserved for use by Tapis. Users specify the environment variables to be injected into their application containers via the *envVariables* parameter. Most other singularity options are available to users.

Jobs will use the PID returned when issuing the background command to monitor the container's execution. Jobs determines that the application has terminated when the PID is no longer in use by the operating system.

Jobs uses the same **TapisJob.exitcode** file convention introduced above to attain the application's exit code (if the file exists).

Required Scripts

The Singularity Start and Singularity Run approaches boath allow SSH sessions between Jobs and execution hosts to end without interrupting container execution. Each approach, however, requires that the application image be appropriately constructed. Specifically,

```
Singularity start requires the startscript to be defined in the image.
Singularity run requires the runscript to be defined in the image.
```

Required Termination Order

Since Jobs monitors container execution by querying the operating system using the PID obtained at launch time, the initially launched program should be the last part of the application to terminate. The program specified in the image script can spawn any number of processes (and threads), but it should not exit before those processes complete.

Optional Exit Code Convention

Applications are not required to support the **TapisJob.exitcode** file convention as described above, but it is the only way in which Jobs can report the application specified exit status to the user.

8.5 Querying Jobs

tbd

8.6 Job Actions

CHAPTER 9

Meta

Meta V3 is a REST API Microservice for MongoDB which provides server-side Data, Identity and Access Management for Web and Mobile applications.

9.1 Why Meta V3

Meta V2 functionality was built on MongoDB version 1.0 technology. Although limited, it brought the basic functions of a document store to Agave/Tapis platform. Some projects decided to use basic key/value storage to add metadata to Jobs, Apps, Systems and Files. Other projects stored complex document data to associate Jobs and other entities to add a richer information layer to their portals. The limited search functionality and imposed document structure created impediments to using MongoDB as projects had envisioned. Meta V3 removes these impediments.

Meta V3 is built on top of MongoDB version 4.2 technology. The REST API opens up the full functionality of MongoDB as a document store. and delivers MongoDB as a service so that projects are free to create metadata and documents in a fashion that fits their needs. Some of the Meta V3 advantages over Meta V2 include:

- Any valid MongoDB document structure can be used.
- If a search runs in MongoDB CLI, it should run from the API.
- Aggregations are available.
- Database, collection and document creation can be managed by tenant administrator.
- Performance is many times faster.

9.2 Migration from Meta V2 to V3

Migration can simply be accomplished by creating a new database with one or more collections for your project. The tenant administrator can request the initial permissions setup. Once your collection(s) are in place move the result and associatedIds into your new document model.

9.3 Overview

Meta V3 is:

A Stateless Microservice. With Meta V3 projects can focus on building Angular or other frontend applications, because most of the server-side logic necessary to manage database operations, authentication / authorization and related APIs is automatically handled, without the need to write any server-side code except for the UX/UI.

For example, to insert data into MongoDB a developer has to just create client-side JSON documents and then execute POST operations via HTTP to Meta V3. Other functions of a modern MongoDB installation like flexible schema, geoJson and aggregation pipelines ease the development process.

Every tenant will have access to at least one database where they can store and manage json documents. Documents are the trailing end of a nested hierarchy of data that begins with a database that houses one or more collections. The collections house json documents the structure of which is left up to the administrators of the tenant database.

Permissions for access to databases, collections and documents must be predefined before accessing those resources. The definitions for access are defined within the Security Kernel API of Tapis V3 and must be added by a tenant or service administrator. See the Permissions section below for some examples of permissions definitions and access to resources in the Meta V3 API.

9.4 Getting Started

9.4.1 Create a document

We have a database named MyTstDB and a collection name MyCollection. To add a json document to MyCollection, we can do the following:

With CURL:

```
$ curl -v -X POST -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" --data '
↪{"name": "test document slt 7.21.2020-14:27","jimmyList": ["1","3"],"description":
↪"new whatever",}'  $BASE_URL/v3/meta/MyTstDB/MyCollection?basic=true
```

The response will have an empty response body with a status code of 201 “Created” unless the “basic” url query parameter is set to true. Setting the “basic” parameter to true will give a Tapis Basic response along with the “_id” of the newly created document. A more detailed discussion of autogenerated ids and specified ids can be found in the “Create Document” section of “Document Resources”.


```
{
  "result": {
    "_id": "5f189316e37f7b5a692285f3"
  },
  "status": "201",
  "message": "Created",
  "version": "0.0.1"
}
```


9.4.2 List documents

Using our MyTstDb/MyCollection resources we can ask for a default list of documents in MongoDB default sorted order. The document we created earlier should be listed with a new “_id” field that was autogenerated by MongoDB.

With CURL:

```
$ curl -v -X GET -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" $BASE_
↪URL/v3/meta/MyTstDB/MyCollection
```

The response will be an array of json documents from MyCollection :


```
[
  {
    "_id": {
      "$oid": "5f189316e37f7b5a692285f3"
    },
    "name": "test document slt 7.21.2020-14:27",
    "jimmyList": [
      "1",
      "3"
    ],
    "description": "new whatever",
    "_etag": {
      "$oid": "5f189316296c81742a6a3e4c"
    }
  },
  {
    "_id": {
      "$oid": "5f1892ece37f7b5a692285e9"
    },
    "name": "test document slt 7.21.2020-14:25",
    "jimmyList": [
      "1",
      "3"
    ],
    "description": "new whatever",
    "_etag": {
      "$oid": "5f1892ec296c81742a6a3e4b"
    }
  }
]
```


9.4.3 Get a document

If we know the "_id" of a created document, we can ask for it directly.

With CURL:

```
$ curl -v -X GET -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" $BASE_
↪URL/v3/meta/MyTstDB/MyCollection/5f1892ece37f7b5a692285e9
```

The response will be a json document from MyCollection with the "_id" of 5f1892ece37f7b5a692285e9 :


```
{
  "_id": {
    "$oid": "5f1892ece37f7b5a692285e9"
  },
  "name": "test document slt 7.21.2020-14:25",
  "jimmyList": [
    "1",
    "3"
  ],
```


(continues on next page)

(continued from previous page)

```
  "description": "new whatever",
  "_etag": {
    "$oid": "5f1892ec296c81742a6a3e4b"
  }
}
```

9.4.4 Find a document

We can pass a query parameter named "filter" and set the value to a json MongoDB query document. Let's find a document by a specific "name".

With CURL:

```
$ curl -v -X GET -H "Content-Type:application/json" -H "X-Tapis-Token:$jwt" --data-
↪urlencode filter='{"name": "test document slt 7.21.2020-14:25"}' $BASE_URL/v3/meta/
↪MyTstDB/MyCollection
```

The response will be an array of json documents from MyCollection :

```
[
  {
    "_id": {
      "$oid": "5f1892ece37f7b5a692285e9"
    },
    "name": "test document slt 7.21.2020-14:25",
    "jimmyList": [
      "1",
      "3"
    ],
    "description": "new whatever",
    "_etag": {
      "$oid": "5f1892ec296c81742a6a3e4b"
    }
  }
]
```

9.5 Resources

9.5.1 General resources

An unauthenticated Health check is in included in the Meta V3 API to let any user know the current condition of the service.

Health Check

An unauthenticated request for the health status of Meta V3 API.

With pySDK operation:

```
$ t.meta.healthCheck()
```

With CURL:

```
$ curl -v -X GET -H "Content-Type:application/json" $BASE_URL/v3/meta/
```

The response will be a Basic Tapis response on health:

```
{
  "result": "",
  "status": "200",
  "message": "OK",
  "version": "0.0.1"
}
```

9.5.2 Root resources

The Root resource space represents the root namespace for databases on the MongoDB host. All databases are located here. Requests to this space are limited to READ only for tenant administrators.

List DB Names

A request to the Root resource will list Database names found on the server. This request has been limited to those users with tenant administrative roles.

With pySDK operation:

```
$ t.meta.listDBNames()
```

With CURL:

```
$ curl -v -X GET -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" $BASE_
↪URL/v3/meta/
```

The response will a json list of database names:

```
[
    "StreamsDevDB",
    "v1airr"
]
```

9.5.3 Database resources

The Database resource is the top level for many tenant projects. The resource maps directly to a MongoDB named database in the database server. Case matters for matching the name of the database and must be specified when making requests for collections or documents. Currently

List Collection Names

This request will return a list of collection names from the specified database {db}. The permissions for access to the database are set prior to access.

With pySDK operation:

```
$ t.meta.listCollectionNames(db='')
```

With CURL:

```
$ curl -v -X GET -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d ''
↪$BASE_URL/v3/meta/{db}
```

Here is an example response:

```
[
  "streams_alerts_metadata",
  "streams_channel_metadata",
  "streams_instrument_index",
  "streams_project_metadata",
  "streams_templates_metadata",
  "tapisKapa-local"
]
```

Get DB Metadata

This request will return the metadata properties associated with the database. The core server generates an etag in the _properties collection for a database that is necessary for future deletion.

With pySDK operation:

```
$ t.meta.getDBMetadata(db='')
```

With CURL:

```
$ curl -v -X GET -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d ''
↪$BASE_URL/v3/meta/{db}/_meta
```

Here is an example response:

```
{
   "_id": "_meta",
   "_etag": { "$oid": "5ef6232b296c81742a6a3e02" }
}
```

Create DB

TODO: this implementation is not exposed. Creation of a database by tenant administrators is scheduled for inclusion in an administrative interface API in a future release.

This request will create a new named database in the MongoDB root space by a tenant or service administrator.

With pySDK operation:

```
$ t.meta.createDB(db='')
```

With CURL:

```
$ curl -v -X PUT -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d ''
↪$BASE_URL/v3/meta/{db}
```

Here is an example response:

```
{ }
```

Delete a DB TODO: this implementation is not exposed. Deletion of a database by tenant administrators is scheduled for inclusion in an administrative interface API in a future release.

This request will delete a named database in the MongoDB root space by a tenant or service administrator.

With pySDK operation:

```
$ t.meta.deleteDB(db='')
```

With CURL:

```
$ curl -v -X DELETE -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d ''
↪$BASE_URL/v3/meta/{db}
```

Here is an example response:

```

```

{ }

9.5.4 Collection Resources

The Collection resource allows requests for managing and querying json documents within a MongoDB collection.

Create a Collection

You can create a new collection of documents by specifying a collection name under a specific database. /v3/meta/{db}/{collection}

With pySDK operation:

```
$ t.meta.createCollection(db='',collection='')
```

With CURL:

```
$ curl -v -X PUT -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" $BASE_
↪URL/v3/meta/{db}/{collection}
```

Here is an example response:

```
Empty response with HTTP status of 201
```

List Documents

A default number of documents found in the collection are returned in an array of documents.

With pySDK operation:

```
$ t.meta.listDocuments(db='',collection='',filter='')
```

With CURL:

```
$ curl -v -X GET -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d ''
↪$BASE_URL/v3/meta/{db}/{collection}
```

The response will look like the following:

```
[
  {
    "_id": {
      "$oid": "5f1892ece37f7b5a692285e9"
    },
    "name": "test document slt 7.21.2020-14:25",
    "description": "new whatever",
```

(continues on next page)

(continued from previous page)


```
    "_etag": {
      "$oid": "5f1892ec296c81742a6a3e4b"
    }
  },
  {
    "_id": {
      "$oid": "5f1892ece37f7b5a69228533"
    },
    "name": "test document slt 7.21.2020-14:25",
    "description": "new whatever",
    "_etag": {
      "$oid": "5f1892ec296c81742a6a3e444"
    }
  }
]
```


List Documents Large Query

A default number of documents found in the collection are returned in an array of documents.

With pySDK operation:

```
$ t.meta.submitLargeQuery(db='',collection='',page='',pagesize='',sort='',keys='',
↪fileinput)
```

With CURL:

```
$ curl -v -X GET -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d␣
↪@FILENAME '' $BASE_URL/v3/meta/{db}/{collection}/_filter
```

The response will look like the following:


```
[
  {
    "_id": {
      "$oid": "5f1892ece37f7b5a692285e9"
    },
    "name": "test document slt 7.21.2020-14:25",
    "description": "new whatever",
    "_etag": {
      "$oid": "5f1892ec296c81742a6a3e4b"
    }
  },
  {
    "_id": {
      "$oid": "5f1892ece37f7b5a69228533"
    },
    "name": "test document slt 7.21.2020-14:25",
    "description": "new whatever",
    "_etag": {
      "$oid": "5f1892ec296c81742a6a3e444"
    }
  }
]
```


Delete a Collection

This administrative method is only available to tenant or meta administrators and requires an If-Match header param-

eter of the Etag for the collection. The Etag value, if not already known, can be retrieved from the "_meta" call for a collection.

With pySDK operation:

```
$ t.meta.deleteCollection(db='',collection='')
```

With CURL:

```
$ curl -v -X DELETE -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" $BASE_
↪URL/v3/meta/{db}/{collection}
```

Here is an example response:

```
Empty response body with status code 204
```

Get Collection Size

You can find the given size or number of documents in a given collection by calling "_size" on a collection.

With pySDK operation:

```
$ t.meta.getCollectionSize(db='',collection=' ')
```

With CURL:

```
$ curl -v -X GET -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" $BASE_
↪URL/v3/meta/{db}/{collection}/_size
```

Here is an example response:

```
TODO
```

Get Collection Metadata

You can find the metadata properties of a given collection by calling "_meta" on a collection. This would include the Etag value for a collection that is needed for deletion.

With pySDK operation:

```
$ t.meta.getCollectionMetadata(db='',collection=' ')
```

With CURL:

```
$ curl -v -X GET -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" $BASE_
↪URL/v3/meta/{db}/{collection}/_meta
```

Here is an example response:


```
{
  "_id": "_meta",
  "_etag": {
    "$oid": "5f2b2b7a204ce7637579c85f"
  }
}
```

9.5.5 Document Resources

Document resources are json documents found in a collection. Reading, creating, deleting and updating documents along with batch processing make up the operations that can be applied to documents in a collection. There various ways to retrieve one or more documents from a collection, including using a filter query parameter and value in the form of a MongoDB query document. Batch addition of documents, as well as, batch updates based on queries is also allowed.

Create a Document

Creating a new document within a collection. Submitting a json document within the request body of a POST request will create a new document within the specified collection with a MongoDB autogenerated "_id". Batch document addition is possible by POSTing an array of new documents with a request body for the specified collection. The rules for "_id" creation operates the same way on multiple documents as they do with a single document.

The default representation returned is an empty response body along with a 201 Http status code "Created". However if an additional query parameter named "basic" is added with the value of "true" a basic Tapis response is returned along with the newly created "_id" of the document.

With pySDK operation:

```
$ t.meta.createDocument(db='',collection=' ',basic=' ',body=' ')
```

With CURL:

```
$ curl -v -X POST -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d '{
↪"docName":"test doc"}' $BASE_URL/v3/meta/{db}/{collection}
```

Here is an example response:

```
Empty response
```

Multiple documents can be added to a collection by POSTing a json array of documents. The batch addition of documents only supports the default response.

With pySDK operation:

```
$ t.meta.createDocument(db='',collection=' ',basic=' ',body=' ')
```

With CURL:

```
$ curl -v -X POST -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d '[{
↪"docName":"test doc1"},{"docName":"test doc2"}]' $BASE_URL/v3/meta/{db}/{collection}
```

The response body will be empty:

Get a Document

Get a specific document by its "_id".

With pySDK operation:

```
$ t.meta.getDocument(db='',collection=' ',documentId=' ')
```

With CURL:

```
$ curl -v -X GET -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt"  $BASE_
↪URL/v3/meta/{db}/{collection}/{document_id}
```

The response will be the standard json response:

```
{
    "_id"}
```

Replace a Document

This call replaces an existing document identified by document id ("_id"), with the json supplied in the request body.

With pySDK operation:

```
$ t.meta.replaceDocument(db='',collection=' ',documentId=' ')
```

With CURL:

```
$ curl -v -X PUT -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d '{
↪"docName":"test doc another one"}' $BASE_URL/v3/meta/{db}/{collection}/{document_id}
```

Here is an example response:

```
TODO
```

Modify a Document

This call will replace a portion of a document identified by document id ("_id") with the supplied json.

With pySDK operation:

```
$ t.meta.modifyDocument(db='',collection=' ',documentId=' ')
```

With CURL:

```
$ curl -v -X PATCH -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d '{
↪"docName":"test changed"}' $BASE_URL/v3/meta/{db}/{collection}/{document_id}
```

Here is an example response:

```
TODO
```

Delete Document

Deleting a document with a specific document id ("_id"), removes it from the collection.

With pySDK operation:

```
$ t.meta.deleteDocument(db='',collection=' ',documentId=' ')
```

With CURL:

```
$ curl -v -X DELETE -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d ''
↪$BASE_URL/v3/meta/{db}/{collection}/{document_id}
```

Here is an example response:

```
TODO
```

9.5.6 Index Resources

Indexes can help speed up queries of your collection and the API gives you the ability to define and manage your indexes. You can create an index for a collection, list indexes for a collection and delete an index. Indexes can't be updated they must be deleted and recreated.

List Indexes

List the indexes defined for a collection.

With pySDK operation:

```
$ t.meta  TODO
```

With CURL:

```
$ curl -v -X POST -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt"  $BASE_
↪URL/v3/meta/{db}/{collection}/_indexes
```

Here is an example response:

```
TODO
```

Create Index

Create a new Index with a new name. To create an index you have to specify the keys and the index options. Let's create an unique, sparse index on property qty and name our index "qtyIndex".

PUT /v3/meta/{db}/{collection}/_indexes/qtyIndex

```
{"keys": {"qty": 1},"ops": {"unique": true, "sparse": true }}
```

With pySDK operation:

```
$ t.meta
```

With CURL:

```
$ curl -v -X PUT -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d '{
↪"keys":  <keys>, "ops": <options> }' $BASE_URL/v3/meta/{db}/{collection}/_indexes/
↪{indexName}
```

Here is an example response:

```
TODO
```

Delete Index

Remove a named Index from the index list.

With pySDK operation:

```
$ t.meta TODO
```

With CURL:

```
$ curl -v -X DELETE -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" $BASE_
↪URL/v3/meta/{db}/{collection}/_indexes/{indexName}
```

Here is an example response:

```
TODO
```

9.5.7 Aggregation Resources

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. Aggregations in the API are predefined and added to a collections properties. They may also be parameterized for use with multiple sets of inputs.

Create an Aggregation

Create an aggregation pipeline by adding the aggregation to the collection for future execution. The aggregation may have variables that are defined so that a future request may pass variable values for aggregation execution. See "Execute an Aggregation".

```
{ "aggrs" : [
      { "stages" : [ { "$match" : { "name" : { "$var" : "n" } } },
            { "$group" : { "_id" : "$name",
                  "avg_age" : { "$avg" : "$age" }
                } }
          ],
        "type" : "pipeline",
        "uri" : "example-pipeline"
      }
    ]
}
```

Property	Mandatory	Description
type	yes	• for aggregation pipeline operations is "pipeline"
uri	yes	• specifies the URI when the operation is bound under the path /<db>/<collection>/_aggrs.
stages	yes	• the MongoDB aggregation pipeline stages.

For more information refer to https://docs.mongodb.org/manual/core/aggregation-pipeline/

With pySDK operation:

```
$ t.meta  TODO
```

With CURL:

```
$ curl -v -X PUT -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt"
     -d '{ "aggrs" : [{ "stages" : [ { "$match" : { "name" : { "$var" : "n" } } },{ "
↪$group" : { "_id" : "$name","avg_age" : { "$avg" : "$age" }} } ],
        "type" : "pipeline","uri" : "example-pipeline"}]}' $BASE_URL/v3/meta/{db}/
↪{collection}
```

Here is an example response:

```
TODO
```

Execute an Aggregation

TODO

With pySDK operation:

```
$ t.meta  TODO
```

With CURL:

```
$ curl -v -X POST -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d ''
↪$BASE_URL/v3/meta/
```

Here is an example response:

```
TODO
```

Delete an Aggregation

TODO

With pySDK operation:

```
$ t.meta TODO
```

With CURL:

```
$ curl -v -X POST -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d ''
↪$BASE_URL/v3/meta/
```

Here is an example response:

```
TODO
```

CHAPTER 10

PgREST

The PgREST service provides an HTTP-based API to a managed Postgres database. As with the other Tapis v3 service, PgREST utilizes a REST architecture.

10.1 Overview

There are two primary collections in the PgREST API. The Management API, provided at the URL `/v3/pgrest/manage`, includes endpoints for managing the collection of tables, views, stored procedures, and other objects defined in the hosted Postgres database server. Each Tapis tenant has their own schema within PgREST's managed Postrgres database in which the tables and other objects are housed. When a table is created, endpoints are generated that allow users to interact with the data within a table. These endpoints comprise the Data API, available at the URL `/v3/pgrest/data`. Each collection, `/v3/pgrest/data/{collection}`, within the Data API corresponds to a table defined in the Management API. The Data API is used to create, update, and read the rows within the corresponding tables.

10.2 Authentication and Tooling

PgREST currently recognizes Tapis v2 and v3 authentication tokens and uses these for determining access levels.

A valid Tapis v2 OAuth token should be passes to all requests to PgREST using the header `Tapis-v2-token`.

For example, using curl:

```
$ curl -H "Tapis-v2-token: 419465dg63h8e4782057degk20e3371" https://tacc.tapis.io/v3/
↪pgrest/manage/tables
```

Tapis v3 OAuth authentication tokens should be passed to all requests to PgREST using the header `X-Tapis-Token`.

For example, using curl:

```
$ curl -H "X-Tapis-Token: TOKEN_HERE" https://tacc.tapis.io/v3/pgrest/manage/tables
```

Additionally, PgREST should be accessible from the Tapis v3 Python SDK (tapipy) now with the addition of v3 authentication.

10.3 Permissions and Roles

PgREST currently implements a basic, role-based permissions system that leverages the Tapis v3 Security Kernel (SK). We plan to expand on this system in the future to provide more fine-grained authorization controls. For now, three roles are recognized:

- `PGREST_ADMIN` – Grants user read and write access to all objects (e.g., tables) in the `/manage` API as well as read and write access to all associated data in the `/data` API.
- `PGREST_WRITE` – Grants user read and write access to all associated data in the `/data` API.
- `PGREST_READ` – Grants user read access to all associated data in the `/data` API.

Without any of the above roles, a user will not have access to any PgREST endpoints.

Note that these roles are granted at the *tenant* level, so a user may be authorized at one level in one tenant and at a different level (or not at all) in another tenant. In PgREST, the base URLs for a given tenant follow the pattern `<tenant_id>.tapis.io`, just as they do for all other Tapis v3 services. Hence, this request:

```
$ curl -H "Tapis-v2-token: $TOKEN" https://tacc.tapis.io/v3/pgrest/manage/tables
```

would list tables in the TACC tenant, while

```
$ curl -H "Tapis-v2-token: $TOKEN" https://cii.tapis.io/v3/pgrest/manage/tables
```

would list tables in the CII tenant.

10.4 Management API

The Management API includes subcollections for each of the primary Postgres objects supported by PgREST.

10.4.1 Tables

Table management is accomplished with the `/v3/pgrest/manage/tables` endpoint. Creating a table amounts to specifying the table name, the columns on the table, including the type of each column, and any additional validation to be performed when storing data in the column, the root URL where the associated collection will be available within the Data API, and, optionally, which HTTP verbs should not be available on the collection.

For example, suppose we wanted to manage a table of "widgets" with four columns. We could create a table by POSTing the following JSON document to the `/v3/pgrest/manage/tables` endpoint:

```
{
  "table_name": "widgets",
  "root_url": "widgets",
  "columns": {
    "name": {
      "data_type": "varchar",
      "char_len": 255,
      "unique": true,
      "null": false
```

(continues on next page)

(continued from previous page)

```
    },
    "widget_type": {
      "data_type": "varchar",
      "char_len": 100,
      "null": true
    },
    "count": {
      "data_type": "integer",
      "null": true
    },
    "is_private": {
      "data_type": "boolean",
      "null": "true",
      "default": "true"
    }
  }
}
```

The JSON describes a table with 4 columns, `name`, `widget_type`, `count`, and `is_private`. The fields within the JSON object describing each column include its type, defined in the `data_type` attribute (and supporting fields such as `char_len` for `varchar` columns), as well as optional constraints, such as the NOT NULL and UNIQUE constraint, an optional `default` value, and an optional `primary_key` value. Only the `data_type` attribute is required.

To create this table and the corresponding `/data` API, we can use curl like so:

```
$ curl -H "tapis-v2-token: $TOKEN" -H "Content-type: application/json"
  -d "@widgets.json" https://dev.develop.tapis.io/v3/pgrest/manage/tables
```

If all works, the response should look something like this:

```
{
  "status": "success",
  "message": "The request was successful.",
  "version": "dev",
  "result": {
    "table_name": "widgets",
    "table_id": 6,
    "root_url": "widgets",
    "endpoints": [
      "GET_ONE",
      "GET_ALL",
      "CREATE",
      "UPDATE",
      "DELETE"
    ]
  }
}
```

Since the `root_url` attribute has value `widgets`, an associated collection at URL `/v3/pgrest/data/widgets` is automatically made available for managing and retrieving the data (rows) on the table. See the *Data API* section below for more details.

10.5 Supported Data Types

Currently, PgREST supports the following column types:

- `varchar` – Variable length character field; Attribute `char_len` specifying max length is required.
- `text` – Variable length character field with no max length.
- `boolean` – Standard SQL boolean type.
- `integer` – 4 bytes integer field.

Todo... Complete list of supported column types coming soon

The project will be adding support for additional data types in subsequent releases.

10.6 Supported Constraints

Currently, PgREST supports the following SQL constraints:

- `unique` – PgREST supports specifying a single column as unique.
- `null` – If a column description includes `"null": false`, then the SQL `NOT NULL` constraint will be applied to the table.
- `primary_key` – A unique and not null column that acts as the primary key in order to access a specific row. If it is not set, a field named `{table_name}_id` will be created and used by default, it is an integer and increases by one. This field allows the user to instead use their own key in matters such as how to call a row, `/v3/pgrest/data/my_table/my_row_name`, rather than than being assigned a primary_key id which is just a random integer.

10.7 Retrieving Table Descriptions

You can list all tables you have access to by making a GET request to `/v3/pgrest/manage/tables`. For example

```
$ curl -H "tapis-v2-token: $tok" https://dev.tapis.io/v3/pgrest/manage/tables
```

returns a result like

```
[
   {
      "table_name": "initial_table",
      "table_id": 3,
      "root_url": "init",
      "tenant": "dev",
      "endpoints": [
        "GET_ONE",
        "GET_ALL",
        "CREATE",
        "UPDATE",
        "DELETE"
      ],
      "tenant_id": "dev"
    },
```

(continues on next page)

(continued from previous page)

```
    {
      "table_name": "widgets",
      "table_id": 6,
      "root_url": "widgets",
      "tenant": "dev",
      "endpoints": [
        "GET_ONE",
        "GET_ALL",
        "CREATE",
        "UPDATE",
        "DELETE"
      ],
      "tenant_id": "dev"
    }
]
```

We can also retrieve a single table by `id`. For example

```
$ curl -H "tapis-v2-token: $tok" https://dev.tapis.io/v3/pgrest/manage/tables/6

  {
      "table_name": "widgets",
      "table_id": 6,
      "root_url": "widgets",
      "endpoints": [
        "GET_ONE",
        "GET_ALL",
        "CREATE",
        "UPDATE",
        "DELETE"
      ],
      "tenant_id": "dev"
  }
```

We can also pass `details=true` query parameter to see the column definitions, validation schema, etc. For example:

```
$ curl -H "tapis-v2-token: $tok" https://dev.tapis.io/v3/pgrest/manage/tables/6?
↪details=true

{
    "table_name": "widgets",
    "table_id": 6,
    "root_url": "widgets",
    "endpoints": [
      "GET_ONE",
      "GET_ALL",
      "CREATE",
      "UPDATE",
      "DELETE"
    ],
    "columns": {
      "name": {
        "null": false,
        "unique": true,
        "char_len": 255,
```

(continues on next page)

(continued from previous page)


```
        "data_type": "varchar"
      },
      "count": {
        "null": true,
        "data_type": "integer"
      },
      "is_private": {
        "null": "true",
        "default": "true",
        "data_type": "boolean"
      },
      "widget_type": {
        "null": true,
        "char_len": 100,
        "data_type": "varchar"
      }
    },
    "tenant_id": "dev",
    "update schema": {
      "name": {
        "type": "string",
        "maxlength": 255
      },
      "count": {
        "type": "integer"
      },
      "is_private": {
        "type": "boolean"
      },
      "widget_type": {
        "type": "string",
        "maxlength": 100
      }
    },
    "create schema": {
      "name": {
        "type": "string",
        "required": true,
        "maxlength": 255
      },
      "count": {
        "type": "integer",
        "required": false
      },
      "is_private": {
        "type": "boolean",
        "required": false
      },
      "widget_type": {
        "type": "string",
        "required": false,
        "maxlength": 100
      }
    }
  }
}
```

10.7.1 Views

Coming soon

10.7.2 Stored Procedures

Coming soon

10.8 Data API

The Data API provides endpoints for managing and retrieving data (rows) stored on tables defined through the Management API. For each table defined through the Management API, there is a corresponding endpoint within the Data API with URL `/v3/pgrest/data/{root_url}`, where `{root_url}` is the associated attribute on the table.

Continuing with our widgets table from above, the associated endpoint within the Data API would have URL `/v3/pgrest/data/widgets` because the `root_url` property of the widgets table was defined to be `widgets`. Moreover, all 5 default endpoints on the `widgets` collection are available (none were explicitly restricted when registering the table). The endpoints within the `widgets` can be described as follows:

GET	POST	PUT	DELETE	Endpoint	Description
X	X	X		/v3/pgrest/data/widgets	List/create widgets; bulk update multiple widgets.
X		X	X	/v3/pgrest/data/widgets/{id}	Get/update/delete a widget by id.

Note that the `id` column is used for referencing a specific row. Currently, PgREST generates this column automatically for each table and calls it *{table_name}_id*. It is a sql serial data type. To override this generic `id` column, you may assign a key of your choice the `primary_key` constraint. We'll then use the values in this field to get a specified rows. `primary_key` columns, must be integers or varchars which are not null and unique.

Additionally, to find the `id` to use for your row, the data endpoints return a `_pkid` field in the results for each row for ease of use. `_pkid` is not currently kept in the database, but is added to the result object between retrieving the database result and returning the result to the user. As such, `where` queries will NOT work on the `_pkid` field.

10.8.1 Creating a Row

Sending a POST request to the `/v3/pgrest/data/{root_url}` URL will create a new row on the corresponding table. The POST message body should be a JSON document providing values for each of the columns inside a single `data` object. The values will first be validated with the json schema generated from the columns data sent in on table creation. This will enforce data types, max lengths, and required fields. The row is then added to the table using pure SQL format and is fully ATOMIC.

For example, the following JSON body could be used to create a new row on the widgets example table:

new_row.json:

```
{
    "data": {
      "name": "example-widget",
      "widget_type": "gear",
      "count": 0,
      "is_private": false
    }
}
```

The following curl command would create a row defined by the JSON document above

```
$ curl -H "tapis-v2-token: $TOKEN" -H "Content-type: application/json" -d "@new_row.
↪json" https://<tenant>.tapis.io/v3/pgrest/data/widgets
```

if all goes well, the response should look like

```
{
  "status": "success",
  "message": "The request was successful.",
  "version": "dev",
  "result": [
    {
      "widgets_id": 1,
      "name": "example-widget",
      "widget_type": "gear",
      "count": 0,
      "is_private": false
    }
  ]
}
```

Note that an `id` of `1` was generated for the new record.

10.8.2 Updating a Row

Sending a PUT request to the `/v3/pgrest/data/{root_url}/{id}` URL will update an existing row on the corresponding table. The request message body should be a JSON document providing the columns to be updates and the new values. For example, the following would update the `example-widget` created above:

update_row.json

```
{
  "data": {
    "count": 1
  }
}
```

The following curl command would update the `example-widget` row (with `id` of `1`) using the JSON document above

```
$ curl -H "tapis-v2-token: $TOKEN" -H "Content-type: application/json" -d "@update_
↪row.json" https://<tenant>.tapis.io/v3/pgrest/data/widgets/1
```

Note that since only the `count` field is provided in the PUT request body, that is the only column that will be modified.

10.8.3 Updating Multiple Rows

Update multiple rows with a single HTTP request is possible using a `where` filter (for more details, see the section *Where Stanzas* below), provided in the PUT request body. For example, we could update the `count` column on all rows with a negative count to 0 using the following

update_rows.json

```
{
    "count": 0,
    "where": {
        "count": {
            "operator": "<",
            "value": 0
        }
    }
}
```

This update_rows.json would be used in a PUT request to the root `widgets` collection, as follows:

```
$ curl -H "tapis-v2-token: $TOKEN" -H "Content-type: application/json" -d "@update_
↪rows.json" https://<tenant>.tapis.io/v3/pgrest/data/widgets
```

10.8.4 Where Stanzas

In PgREST, `where` stanzas are used in various endpoints throughout the API to filter the collection of results (i.e., rows) that an action (such as retrieving or updating) is applied to. The `where` stanza should be a JSON object with each key being the name of a column on the table and the value under each key being a JSON object with two properties:

- `operator` – a valid operator for the comparison. See the *Valid Operators* table below.
- `value` – the value to compare the row's column to (using the operator).

Naturally, the type (string, integer, boolean, etc.) of the `value` property should correspond to the type of the column specified by the key. Note that multiple keys corresponding to the same column or different columns can be included in a single `where` stanza. For example, the following where stanza would pick out rows whose `count` was between `0` and `100` and whose `is_private` property was `true`:

```
{
    "where": {
        "count": {
            "operator": ">",
            "value": 0
        },
        "count": {
            "operator": "<",
            "value": 100
        },
        "is_private": {
            "operator": "=",
            "value": true
        }
}
```

10.8.5 Valid Operators

PgREST recognizes the following operators for use in `where` stanzas.

Operator	Postgres Equivalent	Description
<	<	Less than
>	>	Greater than
=	=	Equal
...	...	...

Todo... Full table coming soon

10.8.6 Retrieving Rows

To retrieve data from the `/data` API, make an HTTP GET request to the associated URL; an HTTP GET to `/v3/pgrest/data/{root_url}` will retrieve all rows on the associated table, while an HTTP GET to `/v3/pgrest/data/{root_url}/{id}` will retrieve the individual row.

For example,

```
$ curl -H "tapis-v2-token: $TOKEN" https://dev.tapis.io/v3/pgrest/data/init
```

retrieves all rows of the table "init":


```
[
  {
    "_pkid": 1,
    "initial_table_id": 1,
    "col_one": "col 1 value",
    "col_two": 3,
    "col_three": 8,
    "col_four": false,
    "col_five": null
  },
  {
    "_pkid": 2,
    "initial_table_id": 2,
    "col_one": "val",
    "col_two": 5,
    "col_three": 9,
    "col_four": true,
    "col_five": "hi there"
  },
  {
    "_pkid": 3,
    "initial_table_id": 3,
    "col_one": "value",
    "col_two": 7,
    "col_three": 9,
    "col_four": true,
    "col_five": "hi there again"
  }
]
```


while the following curl:

```
$ curl -H "tapis-v2-token: $TOKEN" https://dev.tapis.io/v3/pgrest/data/init/3
```

retrieves just the row with id "3":

```
{
  "_pkid": 3,
  "initial_table_id": 3,
  "col_one": "value",
  "col_two": 7,
  "col_three": 9,
  "col_four": true,
  "col_five": "hi there again"
}
```

We can also search for specific rows using a `where` query parameter appended to the `/v3/pgrest/data/{root_url}` endpoint. The `where` query parameter takes the form `where_<column>=<value>`. For instance with the above example, we can search for all records where "col_four" equals `true` with the following:

```
$ curl -H "tapis-v2-token: $TOKEN" https://dev.tapis.io/v3/pgrest/data/init?where_col_
↪four=true

  [
    {
      "_pkid": 2,
      "initial_table_id": 2,
      "col_one": "val",
      "col_two": 5,
      "col_three": 9,
      "col_four": true,
      "col_five": "hi there"
    },
    {
      "_pkid": 3,
      "initial_table_id": 3,
      "col_one": "value",
      "col_two": 7,
      "col_three": 9,
      "col_four": true,
      "col_five": "hi there again"
    }
  ]
```

and similarly, we can search for records where "col_four" equals `false`

```
$ curl -H "tapis-v2-token: $TOKEN" https://dev.tapis.io/v3/pgrest/data/init?where_col_
↪four=false

  [
    {
      "_pkid": 1,
      "initial_table_id": 1,
      "col_one": "col 1 value",
      "col_two": 3,
      "col_three": 8,
      "col_four": false,
      "col_five": null
    }
  ]
```

Note that the result is always a JSON list, even when one or zero records are returned:

```
$ curl -H "tapis-v2-token: $TOKEN" https://dev.tapis.io/v3/pgrest/data/init?where_col_
↪two=2
  []
```

CHAPTER 11

Actors

11.1 Introduction to Abaco

11.1.1 What is Abaco

Abaco is an NSF-funded web service and distributed computing platform providing functions-as-a-service (FaaS) to the research computing community. Abaco implements functions using the Actor Model of concurrent computation. In Abaco, each actor is associated with a Docker image, and actor containers are executed in response to messages posted to their inbox which itself is given by a URI exposed over HTTP.

Abaco will ultimately offer three primary higher-level capabilities on top of the underlying Actor model:

- *Reactors* for event-driven programming
- *Asynchronous Executors* for scaling out function calls within running applications, and
- *Data Adapters* for creating rationalized microservices from disparate and heterogeneous sources of data.

Reactors and Asynchronous Executors are available today while Data Adapters are still under active development.

11.1.2 Using Abaco

Abaco is in production and has been adopted by several projects. Abaco is available to researchers and students. To learn more about the the system, including getting access, follow the instructions in getting-started/index.

11.2 Getting Started

This Getting Started guide will walk you through the initial steps of setting up the necessary accounts and installing the required software before moving to the Abaco Quickstart, where you will create and execute your first Abaco actor.

If you are already using Docker Hub and the TACC Cloud APIs, feel free to jump right to the *Abaco Quickstart* or check out the Abaco Live Docs site.

11.2.1 Account Creation and Software Installation

Create a TACC account

The main instance of the Abaco platform is hosted at the Texas Advanced Computing Center (TACC). TACC designs and deploys some of the world's most powerful advanced computing technologies and innovative software solutions to enable researchers to answer complex questions. To use the TACC-hosted Abaco service, please create a TACC account .

Create a Docker account

Docker is an open-source container runtime providing operating-system-level virtualization. Abaco pulls images for its actors from the public Docker Hub. To register actors you will need to publish images on Docker Hub, which requires a Docker account .

Install the Tapis Python SDK

To interact with the TACC-hosted Abaco platform in Python, we will leverage the Tapis Python SDK, tapipy. To install it, simply run:

```
$ pip3 install tapipy
```

Attention: `tapipy` works with Python 3.

11.2.2 Working with TACC OAuth

Authentication and authorization to the Tapis APIs uses OAuth2, a widely-adopted web standard. Our implementation of OAuth2 is designed to give you the flexibility you need to script and automate use of Tapis while keeping your access credentials and digital assets secure. This is covered in great detail in our Tenancy and Authentication section, but some key concepts will be highlighted here, interleaved with Python code.

Create an Tapis Client Object

The first step in using the Tapis Python SDK, tapipy, is to create a Tapis Client object. First, import the `Tapis` class and create python object called `t` that points to the Tapis server using your TACC username and password. Do so by typing the following in a Python shell:

```
# Import the Tapis object
from tapipy import Tapis

# Log into you the Tapis service by providing user/pass and url.
t = Tapis(base_url='https://tacc.tapis.io',
          username='your username',
          password='your password')
```

Generate a Token

With the `t` object instantiated, we can exchange our credentials for an access token. In Tapis, you never send your username and password directly to the services; instead, you pass an access token which is cryptographically signed by the OAuth server and includes information about your identity. The Tapis services use this token to determine who you are and what you can do.

```
# Get tokens that will be used for authenticated function calls
t.get_tokens()
print(t.access_token.access_token)

Out[1]: eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9...
```

Note that the tapipy `t` object will store and pass your access token for you, so you don't have to manually provide the token when using the tapipy operations. You are now ready to check your access to the Tapis APIs. It will expire though, after 4 hours, at which time you will need to generate a new token. If you are interested, you can create an OAuth client (a one-time setup step, like creating a TACC account) that can be used to generate access and refresh tokens. For simplicity, we are skipping that but if you are interested, check out the Tenancy and Authentication section.

Check Access to the Tapis APIs

The tapipy `t` object should now be configured to talk to all Tapis APIs on your behalf. We can check that the client is configured properly by making any API call. For example, we can use the authenticator service to retrieve the full TACC profile of our user. To do so, use the `get_profile()` function associated with the `authenticator` object on the `t` object, passing the username of the profile to retrieve, as follows.

```
t.authenticator.get_profile(username='apitest')

Out[1]:
create_time: None
dn: cn=apitest,ou=People,dc=tacc,dc=utexas,dc=edu
email: aci-cic@tacc.utexas.edu
username: apitest
```

11.3 Abaco Quickstart

In this Quickstart, we will create an Abaco actor from a basic Python function. Then we will execute our actor on the Abaco cloud and get the execution results.

11.3.1 A Basic Python Function

Suppose we want to write a Python function that counts words in a string. We might write something like this:

```
def string_count(message):
    words = message.split(' ')
    word_count = len(words)
    print('Number of words is: ' + str(word_count))
```

In order to process a message sent to an actor, we use the `raw_message` attribute of the `context` dictionary. We can access it by using the `get_context` method from the `actors` module in `tapipy`.

For this example, create a new local directory to hold your work. Then, create a new file in this directory called `example.py`. Add the following to this file:

```
# example.py

from tapipy.actors import get_context

def string_count(message):
    words = message.split(' ')
    word_count = len(words)
    print('Number of words is: ' + str(word_count))

context = get_context()
message = context['raw_message']
string_count(message)
```

11.3.2 Building Images From a Dockerfile

To register this function as an Abaco actor, we create a docker image that contains the Python function and execute it as part of the default command.

We can build a Docker image from a text file called a Dockerfile. You can think of a Dockerfile as a recipe for creating images. The instructions within a Dockerfile either add files/folders to the image, add metadata to the image, or both.

The FROM Instruction

Create a new file called `Dockerfile` in the same directory as your `example.py` file.

We can use the `FROM` instruction to start our new image from a known image. This should be the first line of our Dockerfile. We will start an official Python image:

```
FROM python:3.6
```

The RUN, ADD and CMD Instructions

We can run arbitrary Linux commands to add files to our image. We'll run the `pip` command to install the `tapipy` library in our image:

```
RUN pip install --no-cache-dir tapipy
```

(note: there is a `abacosample` image that contains Python and the tapipy library; see the Samples section for more details, coming soon.)

We can also add local files to our image using the `ADD` instruction. To add the `example.py` file from our local directory, we use the following instruction:

```
ADD example.py /example.py
```

The last step is to write the command from running the application, which is simply `python /example.py`. We use the `CMD` instruction to do that:

```
CMD ["python", "/example.py"]
```

With that, our `Dockerfile` is now ready. This is what is looks like:

```
FROM python:3.6

RUN pip install --no-cache-dir tapipy
ADD example.py /example.py

CMD ["python", "/example.py"]
```

Now that we have our `Dockerfile`, we can build our image and push it to Docker Hub. To do so, we use the `docker build` and `docker push` commands [note: user is your user on Docker, you must also $ docker login] :

```
$ docker build -t user/my_actor .
$ docker push user/my_actor
```

11.3.3 Registering an Actor

Now we are going to register the Docker image we just built as an Abaco actor. To do this, we will use the `Tapis` client object we created above (see *Working with TACC OAuth*).

To register an actor using the tapipy library, we use the `actors.add()` method and pass the arguments describing the actor we want to register through the `body` parameter. For example:

```
my_actor = {"image": "user/my_actor", "name": "word_counter", "description": "Actor␣
↪that counts words."}
t.actors.createActor(**my_actor)
```

You should see a response like this:

```
_links:
executions: https://tacc.tapis.io/actors/v3/JWpkNmBwKewYo/executions
owner: https://tacc.tapis.io/profiles/v3/jstubbs
createTime: 2020-10-21T17:20:20.718177
defaultEnvironment:
description: Actor that counts words.
hints: []
id: JWpkNmBwKewYo
image: abacosamples/wc
lastUpdateTime: 2020-10-21T17:20:20.718177
link:
mounts: [
container_path: /home/tapis/runtime_files/_abaco_data1
host_path: /home/apim/staging/runtime_files/data1
mode: ro,
container_path: /home/tapis/runtime_files/_abaco_data2
host_path: /home/apim/staging/runtime_files/data2/master/abaco
mode: rw]
owner: abaco
privileged: False
queue: default
state:
stateless: True
status: SUBMITTED
statusMessage:
token: false
type: none
useContainerUid: False
webhook:
```

Notes:

- Abaco assigned an id to the actor (in this case `JWpkNmBwKewYo`) and associated it with the image (in this case, `abacosamples/wc`) which it began pulling from the public Docker Hub.
- Abaco returned a status of `SUBMITTED` for the actor; behind the scenes, Abaco is starting a worker container to handle messages passed to this actor. The worker must initialize itself (download the image, etc) before the actor is ready.
- When the actor's worker is initialized, the status will change to `READY`.

At any point we can check the details of our actor, including its status, with the following:

```
t.actors.getActor(actor_id='JWpkNmBwKewYo')
```

The response format is identical to that returned from the `.add()` method.

11.3.4 Executing an Actor

We are now ready to execute our actor by sending it a message. We built our actor to process a raw message string, so that is what we will send, but there other options, including JSON and binary data. For more details, see the messages section.

We send our actor a message using the `sendMessage()` method:

```
t.actors.sendMessage(actor_id='JWpkNmBwKewYo',
                     request_body={'message': 'Actor, please count these words.'})
```

Abaco queues up an execution for our actor and then responds with JSON, including an id for the execution contained in the `executionId`:

```
_links:
messages: https://tacc.tapis.io/actors/v3/JWpkNmBwKewYo/messages
owner: https://tacc.tapis.io/profiles/v3/jstubbs
executionId: kA1P1m8NkkolK
msg: Actor, please count these words.
```

In general, an execution does not start immediately but is instead queued until a future time when a worker for the actor can take the message and start an actor container with the message. We can retrieve the details about an execution, including its status, using the `getExecution()` method:

```
>>> t.actors.getExecution(actor_id='JWpkNmBwKewYo', execution_id='kA1P1m8NkkolK')
```

The response will be similar to the following:

```
_links:
logs: https://tacc.tapis.io/actors/v3/JWpkNmBwKewYo/executions/kA1P1m8NkkolK/logs
owner: https://tacc.tapis.io/profiles/v3/jstubbs
actorId: JWpkNmBwKewYo
apiServer: https://tacc.tapis.io
cpu: 9678006850
executor: jstubbs
exitCode: 1
finalState:
Dead: False
Error:
ExitCode: 1
FinishedAt: 2020-10-21T17:26:49.77Z0
```

(continues on next page)

(continued from previous page)

```
OOMKilled: False
Paused: False
Pid: 0
Restarting: False
Running: False
StartedAt: 2020-10-21T17:26:45.24Z0
Status: exited
finishTime: 2020-10-21T17:26:49.77Z0
id: kA1P1m8NkkolK
io: 152287298
messageReceivedTime: 2020-10-21T17:26:44.367Z
runtime: 5
startTime: 2020-10-21T17:26:44.841Z
status: COMPLETE
workerId: QBmoQx4pOx1oA
```

Note that a status of `COMPLETE` indicates that the execution has finished and we are ready to retrieve our results.

11.3.5 Retrieving the Logs

The Abaco system collects all standard out from an actor execution and makes it available via the `logs` endpoint. Let's retrieve the logs from the execution we just made. We use the `getExecutionLogs()` method, passing out `actorId` and our `executionId`:

```
t.actors.getExecutionLogs(actor_id='JWpkNmBwKewYo', execution_id='kA1P1m8NkkolK')
```

The response should be similar to the following:

```
_links:
execution: https://tacc.tapis.io/actors/v3/JWpkNmBwKewYo/executions/kA1P1m8NkkolK
owner: https://tacc.tapis.io/profiles/v3/jstubbs
logs: Number of words is: 5\n
```

We see our actor output *Number of words is: 5*, which is the expected result!

11.3.6 Conclusion

Congratulations! At this point you have created, registered and executed your first actor, but there is a lot more you can do with the Abaco system. To learn more about the additional capabilities, please continue on to the Technical Guide.

11.4 Actor Registration

When registering an actor, the only required field is a reference to an image on the public Docker Hub. However, there are several other properties that can be set. The following table provides a list of the configurable properties available to all users and their descriptions.

Property Name	Description
image	The Docker image to associate with the actor. This should be a fully qualified image available on the public Docker Hub. We encourage users to use to image tags to version control their actors.
name	A user defined name for the actor.
description	A user defined description for the actor.
default_environment	The default environment is a set of key/value pairs to be injected into every execution of the actor. The environment can also be overridden when passing a message to the reactor in the query parameters (see messages).
hints	A list of strings representing user-defined "tags" or metadata about the actor. "Official" Abaco hints can be applied to control configurable aspects of the actor runtime, such as the autoscaling algorithm used. (see autoscaling).
link	Actor identifier (id or alias) of an actor to "link" this actor's events to. Requires execute permissions on the linked actor, and cycles are not permitted. (see complex).
privileged	(True/False) - Whether the actor runs in privileged mode and has access to the Docker daemon. *Note*: Setting this parameter to True requires elevated permissions.
stateless	(True/False) - Whether the actor stores private state as part of its execution. If True, the state API will not be available, but in a future release, the Abaco service will be able to automatically scale reactor processes to execute messages in parallel. The default value is False.
token	(True/False) - Whether to generate an OAuth access token for every execution of this actor. Generating an OAuth token add about 500 ms of time to the execution start up time. *Note: the default value for the `token` attribute varies from tenant to tenant. Always explicitly set the token attribute when registering new actors to ensure the proper behavior.
use_container_uid	Run the actor using the UID/GID set in the Docker image. *Note*: Setting this parameter to True requires elevated permissions.
webhook	URL to publish this actor's events to. (see complex).

- The `default_environment` can be used to provide sensitive information to the actor that cannot be put in the image.
- In order to execute privileged actors or to override the UID/GID used when executing an actor container, talk to the Abaco development team about your use case.
- Abaco supports running specific actors within a given tenant on dedicated and/or specialized hardware for performance reasons. It accomplishes this through the use of actor `queues`. If you need to run actors on dedicated resources, talk to the Abaco development team about your use case.

Here is an example using curl; note that to set the default environment, we *must* pass content type `application/json` and be sure to pass properly formatted JSON in the payload.

```
$ curl -H "X-Tapis-Token: $TOKEN" \
-H "Content-Type: application/json" \
-d '{"image": "abacosamples/test", "name": "test", "description": "My test actor␣
↪using the abacosamples image.", "default_environment":{"key1": "value1", "key2":
↪"value2"} }' \
https://tacc.tapis.io/v3/actors
```

To register the same actor using the tapipy library, we use the `actors.createActor()` method and pass the same arguments through the *request_body* parameter. In this case, the `default_environment` is just a standard Python dictionary where the keys and values are `str` type. For example,

```
from tapipy.tapis import Tapis
t = Tapis(api_server='https://tacc.tapis.io', username='<username>', password='
↪<password>')
t.get_tokens()
actor = {"image": "abacosamples/test",
         "name": "test",
         "description": "My test actor using the abacosamples image registered using␣
↪tapipy.",
         "default_environment":{"key1": "value1", "key2": "value2"} }
t.actors.createActor(**actor)
```

11.5 Abaco Context & Container Runtime

In this section we describe the environment that Abaco actor containers can utilize during their execution.

11.5.1 Context

When an actor container is launched, Abaco injects information about the execution into a number of environment variables. This information is collectively referred to as the `context`. The following table provides a complete list of variable names and their description:

Variable Name	Description
_abaco_actor_id	The id of the actor.
_abaco_actor_dbid	The Abaco internal id of the actor.
_abaco_container_repo	The Docker image used to launch this actor container.
_abaco_worker_id	The id of the worker for the actor overseeing this execution.
_abaco_execution_id	The id of the current execution.
_abaco_access_token	An OAuth2 access token representing the user who registered the actor.
_abaco_api_server	The OAuth2 API server associated with the actor.
_abaco_actor_state	The value of the actor's state at the start of the execution.
_abaco_Content-Type	The data type of the message (either 'str' or 'application/json').
_abaco_username	The username of the "executor", i.e., the user who sent the message.
_abaco_api_server	The base URL for the Abaco API service.
MSG	The message sent to the actor, as a raw string.

Notes

- The `_abaco_actor_dbid` is unique to each actor. Using this id, an actor can distinguish itself from other actors registered with the same function providing for SPMD techniques.
- The `_abaco_access_token` is a valid OAuth token that actors can use to make authenticated requests to other TACC Cloud APIs during their execution.
- The actor can update its state during the course of its execution; see the section state for more details.
- The "executor" of the actor may be different from the owner; see sharing for more details.

Access from Python

The `tapipy.actors` module provides access to the above data in native Python objects. Currently, the actors module provides the following utilities:

- **`get_context()` - returns a Python dictionary with the following fields:**
  - `raw_message` - the original message, either string or JSON depending on the Contetnt-Type.
  - `content_type` - derived from the original message request.
  - `message_dict` - A Python dictionary representing the message (for Content-Type: application/json)
  - `execution_id` - the ID of this execution.
  - `username` - the username of the user that requested the execution.
  - `state` - (for stateful actors) state value at the start of the execution.
  - `actor_id` - the actor's id.
- `get_client()` - returns a pre-authenticated `tapipy.Tapis` object.
- `update_state(val)` - Atomically, update the actor's state to the value `val`.

11.5.2 Runtime Environment

The environment in which an Abaco actor container runs has been built to accommodate a number of typical use cases encountered in research computing in a secure manner.

Container UID and GID

When Abaco launches an actor container, it instructs Docker to execute the process using the UID and GID associated with the TACC account of the owner of the actor. This practice guarantees that an Abaco actor will have exactly the same accesses as the original author of the actor (for instance, access to files or directories on shared storage) and that files created or updated by the actor process will be owned by the underlying API user. Abaco API users that have elevated privilleges within the platform can override the UID and GID used to run the actor when registering the actor (see registration).

POSIX Interface to the TACC WORK File System

When Abaco launches an actor container, it mounts the actor owner's TACC WORK file system into the running container. The owner's work file system is made available at `/work` with the container. This gives the actor a POSIX interface to the work file system.

11.6 Messages, Executions, and Logs

Once you have an Abaco actor created the next logical step is to send this actor some type of job or message detailing what the actor should do. The act of sending an actor information to execute a job is called sending a message. This sent message can be raw string data, JSON data, or a binary message.

Once a message is sent to an Abaco actor, the actor will create an execution with a unique `execution_id` tied to it that will show results, time running, and other stats which will be listed below. Executions also have logs, and when

the log are called for, you'll receive the command line logs of your running execution. Akin to what you'd see if you and outputted a script to the command line. Details on messages, executions, and logs are below.

Note: Due to each message being tied to a specific execution, each execution will have exactly one message that can be processed.

11.6.1 Messages

A message is simply the message given to an actor with data that can be used to run the actor. This data can be in the form of a raw message string, JSON, or binary. Once this message is sent, the messaged Abaco actor will queue an execution of the actor's specified image.

Once off the queue, if your specified image has inputs for the messaged data, then that messaged data will be visible to your program. Allowing you to set custom parameters or inputs for your executions.

Sending a message

cURL

To send a message to the `messages` endpoint with cURL, you would do the following:

```
$ curl -H "X-Tapis-Token: $TOKEN" \
-d "message=<your content here>" \
https://tacc.tapis.io/v3/actors/<actor_id>/messages
```

Python

To send a message to the `messages` endpoint with `tapipy` and Python, you would do the following:

```
t.actors.sendMessage(actor_id='<actor_id>',
                     request_body={'message':'<your content here>'})
```

Results

These calls result in a list similar to the following:

```
_links:
messages: https://tacc.tapis.io/actors/v3/NPpjZkmZ4elY8/messages
owner: https://tacc.tapis.io/profiles/v3/jstubbs
executionId: WrMk5EPmwYoL6
msg: <your content here>
```

Get message count

It is possible to retrieve the current number of messages an actor has with the `messages` end point.

cURL

The following retrieves the current number of messages an actor has:

```
$ curl -H "X-Tapis-Token: $TOKEN" \
https://tacc.tapis.io/v3/actors/<actor_id>/messages
```

Python

To retrieve the current number of messages with `tapipy` the following is done:

```
t.actors.getMessages(actor_id='<actor_id>')
```

Results

The result of getting the `messages` endpoint should be similar to:

```
_links:
owner: https://tacc.tapis.io/profiles/v3/jstubbs
messages: 12
```

Binary Messages

An additional feature of the Abaco message system is the ability to post binary data. This data, unlike raw string data, is sent through a Unix Named Pipe (FIFO), stored at /_abaco_binary_data, and can be retrieved from within the execution using a FIFO message reading function. The ability to read binary data like this allows our end users to do numerous tasks such as reading in photos, reading in code to be ran, and much more.

The following is an example of sending a JPEG as a binary message in order to be read in by a TensorFlow image classifier and being returned predicted image labels. For example, sending a photo of a golden retriever might yield, 80% golden retriever, 12% labrador, and 8% clock.

This example uses Python and `tapipy` in order to keep code in one script.

Python with Tapipy

Setting up an `Tapis` object with token and API address information:

```
from tapipy.tapis import Tapis
t = Tapis(api_server='https://tacc.tapis.io', username='<username>', password='
↪<password>')
t.get_tokens()
```

Creating actor with the TensorFlow image classifier docker image:

```
my_actor = {'image': 'abacosamples/binary_message_classifier',
            'name': 'JPEG_classifier',
            'description': 'Labels a read in binary image'}
actor_data = t.actors.createActor(**my_actor)
```

The following creates a binary message from a JPEG image file:

```
with open('<path to jpeg image here>', 'rb') as file:
    binary_image = file.read()
```

Sending binary JPEG file to actor as message with the sendBinaryMessage function (You can also just set the headers with `Content-Type: application/octet-stream`):

```
result = t.actors.sendBinaryMessage(actor_id = actor_data.id,
                                    request_body = binary_image)
```

The following returns information pertaining to the execution:

```
execution = t.actors.getExecution(actor_id = actor_data.id,
                                  execution_id = result.executionId)
```

Once the execution has complete, the logs can be called with the following:

```
exec_logs = t.actors.getExecutionLogs(actor_id = actor_data.id,
                                      execution_id = result.executionId)
```

Sending binary from execution

Another useful feature of Abaco is the ability to write to a socket connected to an Abaco endpoint from within an execution. This Unix Domain (Datagram) socker is mounted in the actor container at /_abaco_results.sock.

In order to write binary data this socket you can use `tapipy` functions, in particular the `send_bytes_result()` function that sends bytes as single result to the socket. Another useful function is the `send_python_result()` function that allows you to send any Python object that can be pickled with `cloudpickle`.

In order to retrieve these results from Abaco you can get the `/actors/<actor_id>/executions/<execution_id>/results` endpoint. Each get of the endpoint will result in exactly one result being popped and retrieved. An empty result with be returned if the results queue is empty.

As a socket, the maximum size of a result is 131072 bytes. An execution can send multiple results to the socket and said results will be added to a queue. It is recommended to to return a reference to a file or object store.

As well, results are sent to the socket and available immediately, an execution does not have to complete to pop a result. Results are given an expiry time of 60 minutes from creation.

cURL

To retrieve a result with cURL you would do the following:

```
$ curl -H "X-Tapis-Token: $TOKEN" \
-d "message=<your content here>" \
https://tacc.tapis.io/v3/actors/<actor_id>/executions/<execution_id>/results
```

Synchronous Messaging

Important: Support for Synchronous Messaging was added in version 1.1.0.

Starting with *1.1.0*, Abaco provides support for sending a synchronous message to an actor; that is, the client sends the actor a message and the request blocks until the execution completes. The result of the execution is returned as an HTTP response to the original message request.

Synchronous messaging prevents the client from needing to poll the executions endpoint to determine when an execution completes. By eliminating this polling and returning the response as soon as it is ready, the overall latency is minimized.

While synchronous messaging can simplify client code and improve performance, it introduces some additional challenges. Primarily, if the execution cannot be completed within the HTTP request/response window, the request will time out. This window is usually about 30 seconds.

> **Warning:** Abaco strictly adheres to message ordering and, in particular, synchronous messages do not skip to the front of the actor's message queue. Therefore, a synchronous message *and all queued messages* must be processed within the HTTP timeout window. To avoid excessive synchronous message requests, Abaco will return a 400 level request if the actor already has more than 3 queued messages at the time of the synchronous message request.

To send a synchronous message, the client appends *_abaco_synchronous=true* query parameter to the request; the rest of the messaging semantics follows the rules and conventions of asynchronous messages.

cURL

The following example uses the curl command line client to send a synchronous message:

```
$ curl -H "X-Tapis-Token: $TOKEN" \
-d "message=<your content here>" \
https://tacc.tapis.io/v3/actors/<actor_id>/messages?_abaco_synchronous=true
```

As stated above, the request blocks until the execution (and all previous executions queued for the actor) completes. To make the response to a synchronous message request, Abaco uses the following rules:

1. If a (binary) result is registered by the actor for the execution, that result is returned with along with a content-type *application/octet-stream*.
2. If no result is available when the execution completes, the logs associated with the execution are returned with content-type *text/html* (charset utf8 is assumed).

11.6.2 Executions

Once you send a message to an actor, that actor will create an execution for the actor with the inputted data. This execution will be queued waiting for a worker to spool up or waiting for a worker to be freed. When the execution is initially created it is given an execution_id so that you can access information about it using the execution_id endpoint.

Access execution data

cURL

You can access the `execution_id` endpoint using cURL with the following:

```
$ curl -H "X-Tapis-Token: $TOKEN" \
https://tacc.tapis.io/v3/actors/<actor_id>/executions/<execution_id>
```

Python

You can access the `execution_id` endpoint using `tapipy` and Python with the following:

```
t.actors.getExecution(actor_id='<actor_id>',
                      execution_id='<execution_id>')
```

Results

Access the `execution_id` endpoint will result in something similar to the following:

```
_links:
logs: https://tacc.tapis.io/actors/v3/JWpkNmBwKewYo/executions/kA1P1m8NkkolK/logs
owner: https://tacc.tapis.io/profiles/v3/jstubbs
actorId: JWpkNmBwKewYo
apiServer: https://tacc.tapis.io
cpu: 9678006850
executor: jstubbs
exitCode: 1
finalState:
Dead: False
Error:
ExitCode: 1
FinishedAt: 2020-10-21T17:26:49.77Z0
OOMKilled: False
Paused: False
Pid: 0
Restarting: False
Running: False
StartedAt: 2020-10-21T17:26:45.24Z0
Status: exited
finishTime: 2020-10-21T17:26:49.77Z0
id: kA1P1m8NkkolK
io: 152287298
messageReceivedTime: 2020-10-21T17:26:44.367Z
runtime: 5
startTime: 2020-10-21T17:26:44.841Z
status: COMPLETE
workerId: QBmoQx4pOx1oA
```

List executions

Abaco allows users to retrieve all executions tied to an actor with the `executions` endpoint.

cURL

List executions with cURL by getting the `executions endpoint`

```
$ curl -H "X-Tapis-Token: $TOKEN" \
https://tacc.tapis.io/v3/actors/<actor_id>/executions
```

Python

To list executions with `tapipy` the following is done:

```
t.actors.listExecutions(actor_id='<actor_id>')
```

Results

Calling the list of executions should result in something similar to:

```
_links:
owner: https://master.staging.tapis.io/profiles/v3/abaco
actorId: WP7vMmRvrDXxN
apiServer: https://master.staging.tapis.io
executions: [
finishTime: Wed, 21 Oct 2020 17:48:33 GMT
id: QBmoQx4pOx1oA
messageReceivedTime: Wed, 21 Oct 2020 17:48:20 GMT
startTime: Wed, 21 Oct 2020 17:48:20 GMT
status: COMPLETE,
finishTime: None
id: QZY8W1Z30Zmbq
messageReceivedTime: Wed, 21 Oct 2020 17:49:56 GMT
startTime: None
status: SUBMITTED]
owner: abaco
totalCpu: 61248097463
totalExecutions: 2
totalIo: 752526010
totalRuntime: 13
```

Reading message in execution

One of the most important parts of using data in an execution is reading said data. Retrieving sent data depends on the data type sent.

Python - Reading in raw string data or JSON

To retrieve JSON or raw data from inside of an execution using Python and `tapipy`, you would get the message context from within the actor and then get it's `raw_message` field.

```
from tapipy.actors import get_context

context = get_context()
message = context['raw_message']
```

Python - Reading in binary

Binary data is transmitted to an execution through a FIFO pipe located at /_abaco_binary_data. Reading from a pipe is similar to reading from a regular file, however `tapipy` comes with an easy to use `get_binary_message()` function to retrieve the binary data.

Note: Each Abaco execution processes one message, binary or not. This means that reading from the FIFO pipe will result with exactly the entire sent message.

```
from tapipy.actors import get_binary_message

bin_message = get_binary_message()
```

11.6.3 Logs

At any point of an execution you are also able to access the execution logs using the `logs` endpoint. This returns information about the log along with the log itself. If the execution is still in the submitted phase, then the log will be an empty string, but once the execution is in the completed phase the log would contain all outputted command line data.

Retrieving an executions logs

cURL

To call the `log` endpoint using cURL, do the following:

```
$ curl -H "X-Tapis-Token: $TOKEN" \
https://tacc.tapis.io/v3/actors/<actor_id>/executions/<execution_id>/logs
```

Python

To call the `log` endpoint using `tapipy` and Python, do the following:

```
t.actors.getExecutionLogs(actor_id='<actor_id>',
                          execution_id='<executionId>')
```

Results

This would result in data similar to the following:

```
_links:
execution: https://tacc.tapis.io/actors/v3/JWpkNmBwKewYo/executions/kA1P1m8NkkolK
owner: https://tacc.tapis.io/profiles/v3/jstubbs
logs: <command line output here>
```

11.7 Database Search

With the introduction of Abaco 1.6.0 database searching has been introduced using the Mongo aggregation system, full-text searching, and indexing. Searching can be done on actor, worker, execution, and log information. This feature allows for users to search based on any information across all objects that they have permission to view. For example, search would allow checking of all viewable executions for ERRORS in one easy call. The search currently makes use of logical operators and datetime to allow for easy searching of any object based on any specific field.

Attention: Search in Abaco was implemented in version 1.6.0.

Search is available on the actors, workers, executions, and logs databases. Search has been implemented on a new `{base}/actors/search/{database}` endpoint alongside being implemented on the `{base}/actors`, `{base}/actors/{actor_id}/workers`, `{base}/actors/{actor_id}/executions`, and ‘‘{base}/actors/{actor_id}/executions/{execution_id}/logs‘‘endpoints.

To use search on the `{base}/actors/search/{database}` endpoint the database to be searched must be specified as either `actors`, `workers`, `executions`, or `logs` in the URL. With no query arguments Abaco will return all entries in the database that you have permission to view. To specify query arguments the user can add a `?` to the end of their url and specify the parameters they are looking to implement.

A table of search parameters, their function, and examples are below.

Parameter	Function	Examples
search	Completes a fuzzy full-text search based on inputs. Returns results by best accuracy/score.	?search=stringToSearchFor
exact-search	Completes a full-text search and looks for exact matches with inputs.	?exact-search=stringToMatchExactly
eq	Checks if given value is equal to db value matching given key.	?id.eq=AKY5o4Z847lB3
neq	Checks if given value is not equal to db value matching given key.	?id.neq=AKY5o4Z847lB3
gt	Checks if given value is greater than db value matching given key.	?start_time.gt=2020-04-29+06:00
gte	Checks if given value is greater than or equal to db value matching given key.	?runtime.gte=423
lt	Checks if given value is less than db value matching given key.	?message_received_time.lt=2020
lte	Checks if given value is less than or equal to db value matching given key.	?final_state.FinishedAt.lte=2020-04-29
in	Checks if db value matching given key match any values in the given list of values.	?status.in=[“BUSY”,”REQUESTED”,”READY”]
nin	Checks if db value matching given key does not match any values in the given list of values.	?status.nin=[“COMPLETED”, “READY”]
like	Checks if given value in (through regex) db value matching given key.	?image.like=abaco_docker_username
nlike	Checks if given value not in (through regex) db value matching given key.	?image.nlike=abaco_test
between	Checks if db value matching given key is greater than or equal to first given value, and less than or equal to second given value.	?start_time.between= 2020-04-29T20:15:52:246Z, 2021-06-24-05:00
limit	Sets a limit on total amount of results returned. Defaults to 10 results.	?limit=20
skip	Skips a specified amount of results when returning.	?skip=4

You may use as many parameters as you want in one query sans `limit` and `skip`, where each may only be used once.

11.7.1 Metadata

Abaco search slightly alters the expected result of a request in the fact that the returned result from a search now returns two objects, the expected result, `search`, and `_metadata`.

This new `_metadata` object returns pertinent information about the amount of records returned, the amount of records the return is limited to, the amount of records skipped (specified in query), and the total amount of records that match the query searched for. This is useful to implement paging or to only receive a set amount of records.

Important: A new `_metadata` object is now returned alongside the usual result in `result`.

```
{'message': 'Executions search completed successfully.',
 'result': {'_metadata': {'countReturned': 1,
                          'recordLimit': 10,
                          'recordsSkipped': 0,
                          'totalCount': 1},
            'search': [{'_links': {'logs': 'https://dev.tenants.aloedev.tacc.cloud/
↪v3/actors/joBjeDkWyBwLx/logs',
                                   'owner': 'https://dev.tenants.aloedev.tacc.cloud/
↪profiles/v3/testuser',
                                   'self': 'https://dev.tenants.aloedev.tacc.cloud/v3/
↪actors/joBjeDkWyBwLx/executions/1JKkQwX75vE56'},
                        'actorId': 'joBjeDkWyBwLx',
                        'cpu': 444097006,
                        'executor': 'testuser',
                        'exitCode': 0,
                        'finalState': {'Dead': False,
                                       'Error': '',
                                       'ExitCode': 0,
                                       'FinishedAt': '2020-04-29T21:47:21.385Z',
                                       'OOMKilled': False,
                                       'Paused': False,
                                       'Pid': 0,
                                       'Restarting': False,
                                       'Running': False,
                                       'StartedAt': '2020-04-29T21:47:19.382Z',
                                       'Status': 'exited'},
                        'id': '1JKkQwX75vE56',
                        'io': 716,
                        'messageReceivedTime': '2020-04-29T21:47:18.7Z00',
                        'runtime': 2,
                        'startTime': '2020-04-29T21:47:18.954Z',
                        'status': 'COMPLETE',
                        'workerId': '7kvAAKYKB6Qk6'}]},
 'status': 'success',
 'version': ':dev'}
```

11.7.2 Inputs

All inputs are given to the search function as query parameters and thus are converted to strings. It is then up to Abaco's side to convert these inputs back to the intended formats. Strings are left untouched. Booleans are expected to be "False" or "false" and "True" or "true" to be converted. Numbers are converted all to floats, these are still comparable to database instances of int, so there should be no issue. Lists are parsed with `json.loads` and will accept either `["test"]` or `['test']` with post-processing on Abaco's end to convert to lists.

The last consumed input type is datetime objects. Abaco accepts a broad range of ISO 8601 like strings. An example of the most detailed string accepted is `2020-04-29T20:15:52:246252-06:00`. `2020-04-29T20:15:52:246Z`, `2020-04-29T20:15:52-06:00`, `2020-04-29T20:15-06:00`, `2020-04-29T20-06:00`, `2020-04-29-06:00`, `2020-04Z`, and `2020` are also acceptable.

Attention: Abaco stores all times in UTC, so addition of your timezone or conversion to UTC is important. If no timezone information is given (`-06:00` or `Z` (to signal UTC)) the datetime is assumed to be in UTC.

Important: Comparison with datetime rounds to the minimum time possible. For instance if you want to see if 2020-12-30 is greater than 2020, you would receive True as 2020 is rounded to *2020-01-01T00:00:00Z*. This holds true until you reach millisecond accurate time.

Creating ISO 8601 formatted strings

Python - String with Timezone

The following gets the current time as an ISO 8601 formatted string with timezone:

```
import datetime
import pytz

austin_time_zone = pytz.timezone("America/Chicago")
isoString = datetime.datetime.now(tz=austin_time_zone).isoformat()
print(isoString)
```

This prints `2020-04-29T16:21:34.602078-05:00`.

Python - UTC String

The following gets the current UTC time as an ISO 8601 formatted string:

```
import datetime

isoString = datetime.datetime.utcnow().isoformat()
print(isoString)
```

This prints `2020-04-29T21:21:34.602078`. Feel free to add the Z or leave it absent.

11.7.3 Searching

Like mentioned above, a search may contain as many parameters as a user wants sans for `limit` and `skip`, where each may only be used once. Search on the new `{base}/actors/search/{database}` always takes place and when given no parameters returns any information the user has access to. To activate on search on the other endpoints, at least one query parameter must be declared.

Important: `x-nonce` queries will still work as expected and do not need any modification.

Performing searches on different endpoints

{base}/actors/search/actors

You can use `actors`, `workers`, `executions`, or `logs` as database inputs for the endpoints. Each queries the specified database.

cURL

```
$ curl -H "X-Tapis-Token: $TOKEN" \
https://tacc.tapis.io/v3/actors/search/actors?image=abacosamples/test&create_time.
↪gt=2020-04-29&status.in=["READY", "BUSY"]
```

Result

```
{'message': 'Search completed successfully.',
'result': {'_metadata': {'countReturned': 1,
                         'recordLimit': 10,
                         'recordsSkipped': 0,
                         'totalCount': 1},
           'search': [{'_links': {'executions': 'https://dev.tenants.aloedev.tacc.
↪cloud/v3/actors/joBjeDkWyBwLx/executions',
                                  'owner': 'https://dev.tenants.aloedev.tacc.cloud/
↪profiles/v3/testuser',
                                  'self': 'https://dev.tenants.aloedev.tacc.cloud/v3/
↪actors/joBjeDkWyBwLx'},
                       'createTime': '2020-04-29T21:46:53.393Z',
                       'defaultEnvironment': {'default_env_key1': 'default_env_value1
↪',
                                              'default_env_key2': 'default_env_value2'},
                       'description': '',
                       'gid': None,
                       'hints': [],
                       'id': 'joBjeDkWyBwLx',
                       'image': 'abacosamples/test',
                       'lastUpdateTime': '2020-04-29T21:46:53.393Z',
                       'link': '',
                       'maxCpus': None,
                       'maxWorkers': None,
                       'memLimit': None,
                       'mounts': [{'container_path': '/_abaco_data1',
                                   'host_path': '/data1',
                                   'mode': 'ro'}],
                       'name': 'abaco_test_suite_default_env',
                       'owner': 'testuser',
                       'privileged': False,
                       'queue': 'default',
                       'state': {},
                       'stateless': True,
                       'status': 'READY',
                       'statusMessage': ' ',
                       'tasdir': None,
                       'token': 'false',
```

(continues on next page)

(continued from previous page)

```
                        'type': 'none',
                        'uid': None,
                        'useContainerUid': False,
                        'webhook': ''}]},
'status': 'success',
'version': ':dev'}
```

{base}/actors/joBjeDkWyBwLx/executions

For a search from an endpoint like this the actor_id will already be in the query, so for this example you would only receive executions with the actor_id of `joBjeDkWyBwLx`. `{base}/actors/joBjeDkWyBwLx/workers` would result in the same behaviour, but for workers. This usage means that performing a search on `{base}/actors/joBjeDkWyBwLx/executions/1JKkQwX75vE56/logs` would always result in one result. Only search on the `{base}/actors` thus is the only full search available that does not use the `{base}/actors/search/{database}` endpoint.

Attention: Use the `{base}/actors/search/{database}` endpoint for a full search of the specified database.

cURL

```
$ curl -H "X-Tapis-Token: $TOKEN" \
https://tacc.tapis.io/v3/actors/search/actors/joBjeDkWyBwLx/executions?
↪status=COMPLETE&start_time.gt=2019
```

Result

```
{'message': 'Executions search completed successfully.',
 'result': {'_metadata': {'countReturned': 1,
                          'recordLimit': 10,
                          'recordsSkipped': 0,
                          'totalCount': 1},
            'search': [{'_links': {'logs': 'https://dev.tenants.aloedev.tacc.cloud/
↪v3/actors/joBjeDkWyBwLx/logs',
                                   'owner': 'https://dev.tenants.aloedev.tacc.cloud/
↪profiles/v3/testuser',
                                   'self': 'https://dev.tenants.aloedev.tacc.cloud/v3/
↪actors/joBjeDkWyBwLx/executions/1JKkQwX75vE56'},
                        'actorId': 'joBjeDkWyBwLx',
                        'cpu': 444097006,
                        'executor': 'testuser',
                        'exitCode': 0,
                        'finalState': {'Dead': False,
                                       'Error': '',
                                       'ExitCode': 0,
                                       'FinishedAt': '2020-04-29T21:47:21.385Z',
                                       'OOMKilled': False,
                                       'Paused': False,
                                       'Pid': 0,
```

(continues on next page)

(continued from previous page)

```
                                    'Restarting': False,
                                    'Running': False,
                                    'StartedAt': '2020-04-29T21:47:19.382Z',
                                    'Status': 'exited'},
                          'id': '1JKkQwX75vE56',
                          'io': 716,
                          'messageReceivedTime': '2020-04-29T21:47:18.7Z00',
                          'runtime': 2,
                          'startTime': '2020-04-29T21:47:18.954Z',
                          'status': 'COMPLETE',
                          'workerId': '7kvAAKYKB6Qk6'}]},
 'status': 'success',
 'version': ':dev'}
```

11.8 Actor State

In this section we describe the state that can persist through Abaco actor container executions.

11.8.1 State

When an actor is registered, its `stateless` property is automatically set to `true`. An actor must be registered with `stateless=false` to be stateful (maintain state across executions).

Once an actor is executed, the associated worker `GETs` data from the `/v3/actors/{actor_id}/state` endpoint and injects it into the actor's `_abaco_actor_state` environment variable. While an actor is executing, the actor can update its state by `POSTing` to the aforementioned endpoint.

- The worker only `GETs` data from the state endpoint one time as the actor is initiated. If the actor updates its state endpoint during execution, the worker does not inject the new state until a new execution.
- Stateful actors may only have one associated worker in order to avoid race conditions. Thus generally, stateless actors will execute quicker as they can operate in parallel.
- Issuing a state to a stateless actor will return a `actor is stateless.` error.
- The `state` variable must be JSON-serializable. An example of passing JSON-serializable data can be found under *Examples* below.

11.8.2 Utilizing State in Actors to Accomplish Something

WIP

11.8.3 Examples

curl

Here are some examples interacting with state using curl.

Registering an actor specifying statefulness: `stateless=false`.

```
$ curl -H "X-Tapis-Token: $TOKEN" \
-d "image=abacosamples/test&stateless=false" \
https://tacc.tapis.io/v3/actors
```

POSTing a state to a particular actor; keep in mind we must indicate in the header that we are passing content type `application/json`.

```
$ curl -H "X-Tapis-Token: $TOKEN" \
-H "Content-Type: application/json" \
-d '{"some variable": "value", "another variable": "value2"}' \
https://tacc.tapis.io/v3/actors/<actor_id>/state
```

GETting information about a particular actor's state.

```
$ curl -H "X-Tapis-Token: $TOKEN" \
https://tacc.tapis.io/v3/actors/<actor_id>/state
```

Python

Here are some examples interacting with state using Python. The `tapipy.actors` module provides access to an actor's environment data in native Python objects.

Registering an actor specifying statefulness: `stateless=false`.

```
from tapipy.tapis import Tapis
t = Tapis(api_server='https://tacc.tapis.io', username='<username>', password='
↪<password>')
t.get_tokens()
actor = {"image": "abacosamples/test",
         "stateless": "False"}
actor_res = t.actors.createActor(**actor)
```

POSTing a state to a particular actor; again keep in mind we must pass in JSON serializable data.

```
state = {"some variable": "value", "another variable": "value2"}
t.actors.updateState(actor_id = actor_res.id,
                     request_body = state)
```

GETting information about a particular actor's state. This function returns a Python dictionary with many fields one of which is state.

```
from tapipy.actors import get_context
get_context()
{'raw_message': '<text>', 'content_type': '<text>', 'execution_id': '<text>',
↪'username': '<text>', 'state': 'some_state', 'actor_dbid': '<text>', 'actor_id': '
↪<text>', 'raw_message_parse_log': '<text>', 'message_dict': {}}
```

11.9 Actor Sharing and Nonces

Abaco provides a basic permissions system for securing actors. An actor registered with Abaco starts out as private and only accessible to the API user who registered it. This API user is referred to as the "owner" of the actor. By making a POST request to the permissions endpoint for an actor, a user can manage the list of API users who have access to the actor.

11.9.1 Permission Levels

Abaco supports sharing an actor at three different permission levels; in increasing order, they are: *READ*, *EXECUTE* and *UPDATE*. Higher permission imply lower permissions, so a user with *EXECUTE* also has *READ* while a user with *UPDATE* has *EXECUTE* and *READ*. The permission levels provide the followig accesses:

- *READ* - ability to list the actor to see it's details, list executions and retrieve execution logs.
- *EXECUTE* - ability to send an actor a message.
- *UPDATE* - ability to change the actor's definition.

cURL

To share an actor with another API user, make a POST request to the */permissions* endpoint; the following example uses curl to grant READ permission to API user *jdoe*.

```
$ curl -H "X-Tapis-Token: $TOKEN" \
-d "user=jdoe&level=READ" \
https://tacc.tapis.io/v3/actors/<actor_id>/permissions
```

Example response:

```
{
  "message": "Permission added successfully.",
  "result": {
    "jdoe": "READ",
    "testuser": "UPDATE"
  },
  "status": "success",
  "version": "1.0.0"
}
```

We can list all permissions associated with an actor at any time using a GET request:

```
$ curl -H "X-Tapis-Token: $TOKEN" \
https://tacc.tapis.io/v3/actors/<actor_id>/permissions
```

Example response:

```
{
  "message": "Permissions retrieved successfully.",
  "result": {
    "jdoe": "READ",
    "jsmith": "EXECUTE",
    "testuser": "UPDATE"
  },
  "status": "success",
  "version": "1.0.0"
}
```

Note: To remove a user's permission, POST to the permission endpoint and set *level=NONE*

11.9.2 Public Actors

At times, it can be useful to grant **all** API users access to an actor. To enable this, Abaco recognizes the special ABACO_WORLD user. Granting a permission to the ABACO_WORLD user will effectively grant the permission to all API users.

11.9.3 cURL

The following grants *READ* permission to all API users:

```
$ curl -H "X-Tapis-Token: $TOKEN" \
-d "user=ABACO_WORLD&level=READ" \
https://tacc.tapis.io/v3/actors/<actor_id>/permissions
```

11.9.4 Nonces

Abaco provides a capability referred to as actor *nonces* to ease integration with third-party systems leveraging different authentication mechanisms. An actor *nonce* can be used in place of the typical TACC API access token (bearer token). However, unlike an access token which can be used for any actor the user has access, a nonce can only be used for a specific actor.

Creating Nonces

API users create nonces using the nonces endpoint associated with an actor. Nonces can be limited to a specific permission level (e.g., *READ* only), and can have a finite number of uses or an unlimited number.

The following example uses curl to create a nonce with *READ* level permission and with 5 uses.

```
$ curl -H "X-Tapis-Token: $TOKEN" \
-d "maxUses=5&level=READ" \
https://tacc.tapis.io/v3/actors/<actor_id>/nonces
```

A typical response:

```
{
  "message": "Actor nonce created successfully.",
  "result": {
    "_links": {
      "actor": "https://tacc.tapis.io/v3/actors/rNjQG5BBJoxO1",
      "owner": "https://tacc.tapis.io/profiles/v3/testuser",
      "self": "https://tacc.tapis.io/v3/actors/rNjQG5BBJoxO1/nonces/DEV_qBMrvO6Zy0yQz"
    },
    "actorId": "rNjQG5BBJoxO1",
    "apiServer": "http://172.17.0.1:8000",
    "createTime": "2019-06-18 12:20:53.087704",
    "currentUses": 0,
    "description": "",
    "id": "TACC_qBMrvO6Zy0yQz",
    "lastUseTime": "None",
    "level": "READ",
    "maxUses": 5,
    "owner": "testuser",
    "remainingUses": 5,
```

(continues on next page)

(continued from previous page)

```
  },
  "status": "success",
  "version": "1.0.0"
}
```

The *id* of the nonce (in the above example, *TACC_qBMrvO6Zy0yQz*) can be used to access the actor in place of the access token.

Note: Roles are used throughout the TACC API's to grant users with specific privileges (e.g., administrative access to certain APIs). The roles of the API user generating the nonce are captured at the time the nonce is created; when using a nonce, a request will have permissions granted via those roles. Most users will not need to worry about TACC API roles.

To create a nonce with unlimited uses, set *maxUses=-1*.

Redeeming Nonces

To use a nonce in place of an access token, simply form the request as normal and add the query paramter *x-nonce=<nonce_id>*.

For example

```
$ curl -X POST -d "message=<your content here>" \
https://tacc.tapis.io/v3/actors/<actor_id>/messages?x-nonce=TACC_vr9rMO6Zy0yHz
```

The response will be exactly the same as if issuing the request with an access token.

11.10 Networks of Actors

Working with individual, isolated actors can augment an existing application with a lot of additional functionality, but the full power of Abaco's actor-based system is realized when many actors coordinate together to solve a common problem. Actor coordination introduces new challenges that the system designer must address, and Abaco provides features specifically designed to address these challenges.

11.10.1 Actor Aliases

An *alias* is a user-defined name for an actor that is managed independently of the actor itself. Put simply, an alias maps a name to an actor id, and Abaco will replace a reference to an alias in any request with the actor id defined by the alias at the time. Aliases are useful for insulating an actor from changes to another actor to which it will send messages.

For example, if actor A sends messages to actor B, the user can create an alias for actor B and configure A to send messages to that alias. In the future, if changes need to be made to actor B or if messages from actor A need to be routed to a different actor, the alias value can be updated without any code changes needed on the part of actor A.

Creating and managing aliases is done via the `/aliases` collection.

cURL

To create an alias, make a POST request passing the alias and actor id. For example, suppose we have an actor that counts the words sent in a message. We might create an alias for it with the following:

```
$ curl -H "X-Tapis-Token: $TOKEN" \
-d "alias=counter&actorId=6PlMbDLa4zlON" \
https://tacc.tapis.io/v3/actors/aliases
```

Example response:

```
{
  "message": "Actor alias created successfully.",
  "result": {
    "_links": {
      "actor": "https://tacc.tapis.io/v3/actors/6PlMbDLa4zlON",
      "owner": "https://tacc.tapis.io/profiles/v3/jstubbs",
      "self": "https://tacc.tapis.io/v3/actors/aliases/counter"
    },
    "actorId": "6PlMbDLa4zlON",
    "alias": "counter",
    "owner": "apitest"
  },
  "status": "success",
  "version": "1.1.0"
}
```

With the alias `counter` created, we can now use it in place of the actor id in any Abaco request. For example, we can get the actor's details:

```
$ curl -H "X-Tapis-Token: $TOKEN" \
https://tacc.tapis.io/v3/actors/counter
```

The response returned is identical to that returned when the actor id is used.

11.10.2 Nonces Attached to Aliases

Important: Support for Nonces attached to aliases was added in version 1.1.0.

Important: The nonces attached to aliases feature was updated in version 1.5.0, so that 1) `UPDATE` permission on the underlying actor id is required and 2) It is no longer possible to create an alias nonce for permission level `UPDATE`.

Nonces can be created for aliases in much the same way as creating nonces for a specific actor id - instead of using the `/nonces` endpoint associated with the actor id, use the `/nonces` endpoint associated with the alias instead. The POST message payload is the same. For example:

```
$ curl -H "X-Tapis-Token: $TOKEN" \
-d "maxUses=5&level=READ" \
https://tacc.tapis.io/v3/actors/aliases/counter/nonces
```

will create a nonce associated with the `counter` alias.

Note: Listing, creating and deleting nonces associated with an alias requires the analagous permission for both the alias **and** the associated actor.

11.10.3 Actor Events, Links and WebHooks

Important: Support for Actor events, links and webhooks was added in version 1.2.0.

Abaco captures certain events pertaining to the evolution of the system runtime and provides mechanisms for users to consume these events in actors as well as in external systems.

First, Abaco provides a facility to automatically send a message to a specified actor whenever certain events occur. This mechanism is called an actor *link*: if actor A is registered with a `link` property specifying actor B, then Abaco will automatically send actor B a message whenever any of the recognized events occurs.

Second, an actor can be registered with a `webhook` property: a single string representing a URL to send an HTTP POST request to. The Abaco events subsystem will send a POST request **exactly once** to the specified URL whenever a recognized event occurs.

Webhooks and event messages are guaranteed to be delivered in order relative to the order the events occurred for the specific actor. Since there is no total ordering on events across different actors, there is no analagous order guarantee.

Links or Webhooks - Which to use?

In both cases, the details of the event are described in a JSON message (sent to an actor in the case of a link, and sent in the POST payload in the case of a webhook).

However, the actor link is far more general and flexible since the user can define arbitrary logic to handle the event. Even when the ultimate goal is a webhook, the user may opt for defining a link to an actor that performs the webhook. This approach enables users to customtize the webhook processing in various ways, including retry logic, authentication, etc. In fact, the `abacosamples/webhook` image provides a webhook dispatcher built to parse the Abaco events message with many configurable options.

Use of an actor's `webhook` property is really intended for simple use cases or situations missed or dropped events will not cause a major issue.

Adding a Link

Registering an actor with a link (or updating an exisitng actor to add a link property) follows the same semantics as defined in the registration section; simply add the `link` attribute in the payload. For example, the following request creates an actor with a link to actor id `6PlMbDLa4zlON`.

```
$ curl -H "X-Tapis-Token: $TOKEN" \
-H "Content-Type: application/json" \
-d '{"image": "abacosamples/test", "name": "test", "link": "6PlMbDLa4zlON",
↪"description": "My test actor using the abacosamples image.", "default_environment":
↪{"key1": "value1", "key2": "value2"} }' \
https://tacc.tapis.io/v3/actors
```

It is also possible to link an actor to an alias: just pass `link=<the_alias>` in the registration payload.

Note: Setting a link attribute requires `EXECUTE` permission for the associated actor.

Note: Defining a link property that would result in a cycle of linked actors is not permitted, as this would result in infinite messages. In particular, an actor cannot link to itself.

Adding a WebHook

Registering an actor with a webhook is accomplished similarly by setting the `webhook` property in the actor registration (POST) or update (PUT) payload. For example, the following request creates an actor with a webhook set to the requestbin at `https://eniih104j4tan.x.pipedream.net`.

```
$ curl -H "X-Tapis-Token: $TOKEN" \
-H "Content-Type: application/json" \
-d '{"image": "abacosamples/test", "name": "test", "webhook": "https://eniih104j4tan.
↪x.pipedream.net", }' \
https://tacc.tapis.io/v3/actors
```

Events and Event Message Format

Whenever a supported event occurs, Abaco sends a JSON message to the linked actor or webhook with data about the event. The included data depends on the event type, as documented below.

In the case of a linked actor, all the typical context variables, as documented in context, will be injected as usual, excepted where noted below. In this case, note that there are details about two actors: the actor for which the event occurred and the linked actor itself (which are always different, as self-links are not permitted). The former is described in the message itself with variables such as `actor_id`, `tenant_id`, etc., while the latter is described using the special reserved Abaco variables, e.g., `_abaco_actor_id`, etc.

Variable Name	Description	Event Type
actor_id	The id of the actor for which the event occurred.	all types
tenant_id	The id of the tenant of the actor for which the event occurred.	all types
actor_dbid	The internal id of the actor for which the event occurred.	all types
event_type	The event type associated with the event. (see table below)	all types
event_time_utc	The time of the event, in UTC, as a float.	all types
event_time_display	The time of the event, as a string, formatted for display.	all types
_abaco_link	The actor id of the linked actor (the actor receiving the event message)	all types
_abaco_username	'Abaco Event'	all types
status_message	A message indicating details about the error status.	ACTOR_ERROR
execution_id	The id of the completed execution.	EXECUTION_COMPLETE
exit_code	The exit code of the completed execution.	EXECUTION_COMPLETE
status	The final status of the completed execution.	EXECUTION_COMPLETE

The following table lists all events by their `event_type` value and a brief description. Additional event types may be added in subsequent releases.

Event type	Description
ACTOR_READY	The actor is ready to accept messages.
ACTOR_ERROR	The actor is in error status and requires manual intervention.
EXECUTION_COMPLETE	An actor execution has just completed.

11.11 Autoscaling Actors

The Abaco platform has an optional autoscaler subsystem for automatically managing the pool of workers associated with the registered actors. In general, the autoscaler ignores actors that are registered with `stateless: False`, as it assumes these actors must process their message queues synchronously. For *stateless* actors without custom configurations, the austocaling algorithm is as follows:

1. Every 5 seconds, check the length of the actor's message queue.
2. If the queue length is greater than 0, and the actor's worker pool is less than the maximum workers per actor, start a new worker.
3. If the queue length is 0, reduce the actor's worker pool until: a) the worker pool size becomes 0 or b) the actor receives a message.

In particular, the worker pool associated with an actor with 0 messages in its message queue will be reduced to 0 to free up resources on the Abaco compute cluster.

11.11.1 Official "sync" Hint

Important: Support for actor hints and the official "sync" hint was added in version 1.4.0.

For some use cases, reducing an actor's worker pool to 0 as soon as its message queue is empty is not desirable. Starting up a worker takes significant time, typically on the order of 10 seconds or more, depending on configuration options for the actor, and adding this overhead to actors that have low latency requirements can be a serious issue. In particular, actors that will respond to "synchronous messages" (i.e., `_abaco_synchronous=true`) have low latency requirements to respond within the HTTP timeout window.

For this reason, starting in version 1.4.0, Abaco recognizes an "official" actor hint, `sync`. When registered with the `sync` hint, the Abaco autoscaler will leave at least one worker in the actor's worker pool up to a configurable period of idle time (specific to the Abaco tenant). For the Abaco public tenant, this period is 60 minutes.

The `hints` attribute for an actor is saved at registration time. In the following example, we register an actor with the `sync` hint using curl:

```
$ curl -H "X-Tapis-Token: $TOKEN" \
-H "Content-type: application/json" \
-d '{"image": "abacosamples/wc", "hints": ["sync"]}' \
https://tacc.tapis.io/v3/actors
```

11.12 API Reference

The following table lists the public endpoints within the Abaco API.

GET	POST	PUT	DELETE	Endpoint	Description
X				/v3/actors/utilization	Get high-level usage stats.
X	X			/v3/actors	List/create actors.
X	X			/v3/actors/aliases	List/create aliases.
X			X	/v3/actors/aliases/{alias}	List/delete an alias.
X		X	X	/v3/actors/{actor_id}	List/update/delete an actor.
X	X			/v3/actors/{actor_id}/messages	Get number messages/send message
X	X			/v3/actors/{actor_id}/nonces	List/create actor nonces.
X			X	/v3/actors/{actor_id}/nonces/{nonce_id}	Get nonce details/delete nonce.
X	X			/v3/actors/{actor_id}/state	Retrieve/update actor state.
X	X			/v3/actors/{actor_id}/workers	List/create actor workers.
X			X	/v3/actors/{actor_id}/workers/{worker_id}	Get worker details/delete worker
X	X			/v3/actors/{actor_id}/permissions	List/update actor permissions.
X				/v3/actors/{actor_id}/executions	Retrieve execution details.
			X	/v3/actors/{actor_id}/executions/{eid}	Halt running execution.
X				/v3/actors/{actor_id}/executions/{eid}/logs	Retrieve execution logs.
X				/v3/actors/{actor_id}/executions/{eid}/results	Retrieve execution results.
X				/v3/actors/search/{database}	Searches specified database
X				/metrics	

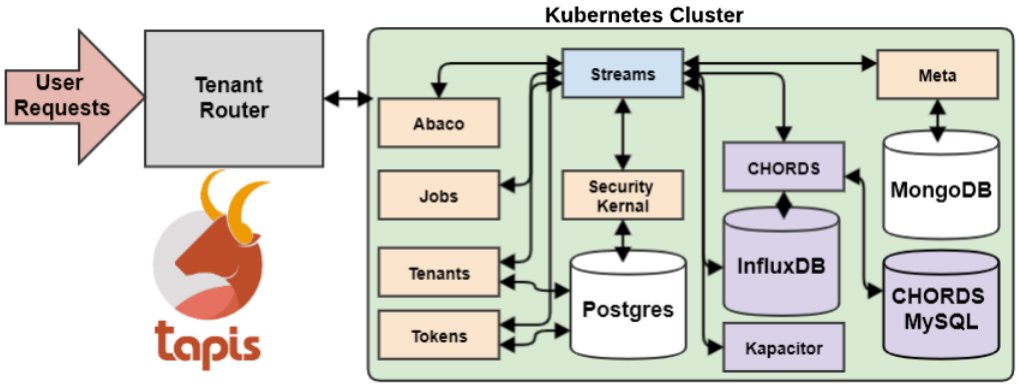
CHAPTER 12

Security

Coming soon

CHAPTER 13

Streams



13.1 Projects

Projects are defined at a top level in the hierarchy of Streams resources. A user registers a project by providing metadata information such as the principal Investigator, project URL, funding resource, etc. A list of authorized users can be added to various project roles to have a controlled access over the project resources. When a project is first registered, a collection is created in the back-end MongoDB. User permissions to access this collection are then set up in the security kernel. Every request to access the project resource or documents within (i.e sites, instruments, variables) goes through a security kernel check and only the authorized user requests are allowed to be processed.

13.1.1 Create Project

With PySDK:

```
$ t.streams.create_project(project_name='tapis_demo_project_testuser6',description=
↪'test project', owner='testuser6', pi='testuser6', funding_resource='tapis',␣
↪project_url='test.tacc.utexas.edu', project_id='tapis_demo_project_testuser6',
↪active=True)
```

With CURL:

```
$ curl -v -X POST -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d '{
↪"project_name": "tapis_demo_project_testuser6",
                                                      "project_id":"tapis_demo_
↪project_testuser6",
                                                      "owner": "testuser6",
                                                      "pi": "testuser6",
                                                      "description": "test project",
                                                      "funding_resource": "tapis",
                                                      "project_url": "test.tacc.
↪utexas.edu",
                                                      "active": "True"}' $BASE_URL/
↪v3/streams/projects
```

The response will look something like the following:

```
active: True
description: test project
funding_resource: tapis
owner: testuser6
permissions:
users: ['testuser6']
pi: testuser6
project_id: tapis_demo_project_testuser6
project_name: tapis_demo_project_testuser6
project_url: test.tacc.utexas.edu
```

13.1.2 List Projects

With PySDK:

```
$ t.streams.list_projects()
```

With CURL:

```
$ curl -H "X-Tapis-Token:$jwt" $BASE_URL/v3/streams/projects
```

The response will look something like the following:

```
[
 active: True
 description: project for early adopters demo
 funding_resource: tapis
 owner: testuser6
 permissions:
 users: ['testuser6']
```

(continues on next page)

(continued from previous page)

```
 pi: ajamthe
 project_id: wq_demo_project12
 project_name: wq_demo_project12
 project_url: test.tacc.utexas.edu,

 active: True
 description: test project
 funding_resource: tapis
 owner: testuser6
 permissions:
 users: ['testuser6']
 pi: testuser6
 project_id: tapis_demo_project_testuser6
 project_name: tapis_demo_project_testuser6
 project_url: test.tacc.utexas.edu,
]
```

13.1.3 Get Project Details

With PySDK:

Note: project_uuid is same as project_id, used in project creation.

```
$ t.streams.get_project(project_uuid='tapis_demo_project_testuser6')
```

With CURL:

```
$ curl -H "X-Tapis-Token:$jwt" $BASE_URL/v3/streams/projects/tapis_demo_project_
↪testuser6
```

The response will look something like the following:

```
active: True
description: project for early demo
funding_resource: tapis
owner: testuser6
permissions:
users: ['testuser6']
pi: testuser6
project_id: tapis_demo_project_testuser6
project_name: tapis_demo_project_testuser6
project_url: test.tacc.utexas.edu
```

13.1.4 Update Project

With PySDK:

```
$ t.streams.update_project(project_uuid='tapis_demo_project_testuser6', project_name=
↪'tapis_demo_project_testuser6', pi='testuser6', owner='testuser6', description=
↪'changed description',project_url='tapis_demo_project.tacc.utexas.edu')
```

With CURL:

```
$ curl -v -X PUT -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d '{
↪"project_name": "tapis_demo_project_testuser6",
                                                    "project_uuid":"tapis_demo_
↪project_testuser6",
                                                    "owner": "testuser6",
                                                    "pi": "testuser6",
                                                    "description": "changed␣
↪description",
                                                    "funding_resource": "tapis",
                                                    "project_url": "tapis_demo_
↪project.tacc.utexas.edu",
                                                    "active": "True"}' $BASE_URL/
↪v3/streams/projects/tapis_demo_project_testuser6
```

The response will look something like the following:

```
active: True
description: changed description
funding_resource: tapis
last_updated: 2020-07-20 17:34:58.848079
owner: testuser6
permissions:
users: ['testuser6']
pi: testuser6
project_id: tapis_demo_project_testuser6
project_name: tapis_demo_project_testuser6
project_url: tapis_demo_project.tacc.utexas.edu
```

13.1.5 Delete Project

With PySDK:

```
$ t.streams.delete_project(project_uuid='tapis_demo_project_testuser6')
```

With CURL:

```
$ curl -X DELETE -H "X-tapis-token:$jwt" $BASE_URL/v3/streams/projects/tapis_demo_
↪project_testuser6
```

The response will look something like the following:

```
active: True
description: project for early adopters demo
funding_resource: tapis
last_updated: 2020-12-04 15:06:41.460343
owner: testuser6
permissions:
users: ['testuser6']
pi: testuser6
```

(continues on next page)

(continued from previous page)

```
project_id: tapis_demo_project_testuser6
project_name: tapis_demo_project_testuser6
project_url: test.tacc.utexas.edu
tapis_deleted: True
```

13.2 Sites

Site is a geographical location that may hold one or more instruments. Sites are next in the streams hierarchy and they inherit permissions from the projects. Project owners can create sites by providing the geographical information such as latitude, longitude and elevation of the site or GeoJSON encoded spatial information. This spatial information is useful when searching sites or data based on location. In the back-end database a site is represented as a JSON document within the project collection. Site permissions are inherited from the project.

13.2.1 Create Site

With PySDK:

```
$ t.streams.create_site(project_uuid='tapis_demo_project_testuser6',site_name='tapis_
↪demo_site', site_id='tapis_demo_site', latitude=50, longitude = 10, elevation=2,
↪description='test_site')
```

With CURL:

```
$  curl -X POST -H "Content-Type:application/json" -H "X-Tapis-Token:$jwt" --data '{
↪"site_name":"tapis_demo_site","latitude":50,"longitude":10,"elevation":2,"site_id":
↪"tapis_demo_site", "description":"test_site"}' $BASE_URL/v3/streams/projects/tapis_
↪demo_project_testuser6/sites
```

The response will look something like the following:

```
chords_id: 27
created_at: 2020-06-08 18:27:12.416134
description: test_site
elevation: 2
latitude: 50
location:
coordinates: [10.0, 50.0]
type: Point
longitude: 10
site_id: tapis_demo_site
site_name: tapis_demo_site
```

13.2.2 List Sites

With PySDK:

```
$ t.streams.list_sites(project_uuid='tapis_demo_project_testuser6')
```

With CURL:

```
$ curl -H "X-Tapis-Token:$jwt"  $BASE_URL/v3/streams/projects/tapis_demo_project_
↪testuser6/sites
```

The response will look something like the following:

```
[
 chords_id: 13
 created_at: 2020-07-20 19:00:55.220397
 description: demo site
 elevation: 1
 latitude: 1.0
 location:
 coordinates: [2.0, 1.0]
 type: Point
 longitude: 2
 site_id: demo_site
 site_name: demo_site,

 chords_id: 12
 created_at: 2020-07-20 18:15:25.404740
 description: test_site
 elevation: 2
 latitude: 50
 location:
 coordinates: [10.0, 50.0]
 type: Point
 longitude: 10
 site_id: tapis_demo_site
 site_name: tapis_demo_site]
```

13.2.3 Get Site Details

With PySDK:

```
$ t.streams.get_site(project_uuid='tapis_demo_project_testuser6', site_id='tapis_demo_
↪site1')
```

With CURL:

```
$ curl -H "X-Tapis-Token:$jwt"  $BASE_URL/v3/streams/projects/tapis_demo_project_
↪testuser6/sites/tapis_demo_site
```

The response will look something like the following:

$ t.streams.get_site(project_uuid='tapis_demo_project_testuser6', site_id='tapis_demo_site')

```
chords_id: 12
created_at: 2020-07-20 18:15:25.404740
description: test_site
elevation: 2
latitude: 50
location:
coordinates: [10.0, 50.0]
type: Point
longitude: 10
site_id: tapis_demo_site
site_name: tapis_demo_site
```

13.2.4 Update Site

With CURL:

```
$ curl -X PUT -H "Content-Type:application/json"  -H "X-Tapis-Token:$jwt" -d '{
↪"project_id": "tapis_demo_project_testuser6","site_name":"tapis_demo_site","latitude
↪":10, "longitude":80, "elevation":2, "description":"test site changed"}' $BASE_URL/
↪v3/streams/projects/tapis_demo_project_testuser6/sites/tapis_demo_site
```

With PySDK

```
$ t.streams.update_site(project_uuid='tapis_demo_project_testuser6',site_name='tapis_
↪demo_site', site_id='tapis_demo_site', latitude=10, longitude = 80, elevation=2,
↪description='test_site changed')
```

The response will look something like the following:

```
chords_id: 4
created_at: 2020-08-10 19:36:48.649316
description: test_site changed
elevation: 2
last_updated: 2020-08-10 19:37:20.115021
latitude: 10
location:
coordinates: [80.0, 10.0]
type: Point
longitude: 80
site_id: tapis_demo_site
site_name: tapis_demo_site
```

13.2.5 Delete Site

With CURL:

```
$ curl -X DELETE -H "X-Tapis-Token:$jwt"  $BASE_URL/v3/streams/projects/tapis_demo_
↪project_testuser6/sites/tapis_demo_site
```

With PySDK

```
$ t.streams.delete_site(project_uuid='tapis_demo_project_testuser6', site_id='tapis_
↪demo_site')
```

13.3 Instruments

Instruments are physical entities that may have one or more embedded sensors to sense various parameters such as temperature, relative humidity, specific conductivity, etc. These sensors referred to as variables in Streams API generate measurements, which are stored in the influxDB along with a ISO8601 timestamp. Instruments are associated with specific sites and projects. Information about the instruments such as site and project ids, name and description of the instrument, etc. are stored in the mongoDB sites JSON document.

13.3.1 Create Instrument

With PySDK

```
$ t.streams.create_instrument(project_uuid='tapis_demo_project_testuser6',topic_
↪category_id ='2',site_id='tapis_demo_site',  inst_name='tapis_demo_instrument',inst_
↪description='demo instrument', inst_id='tapis_demo_instrument')
```

With CURL:

```
$ curl -v -X POST -H "Content-Type:application/json" -H "X-Tapis-Token:$jwt" --data '{
↪"project_uuid":"tapis_demo_project_testuser6","topic_category_id":"2","site_id":
↪"tapis_demo_site","inst_name":"tapis_demo_instrument","inst_description":"demo␣
↪instrument", "inst_id":"tapis_demo_instrument"}'  $BASE_URL/v3/streams/projects/
↪tapis_demo_project_testuser6/sites/tapis_demo_site/instruments
```

The response will look something like the following:

```
chords_id: 10
created_at: 2020-07-20 20:09:11.990814
inst_description: demo instrument
inst_id: tapis_demo_instrument
inst_name: tapis_demo_instrument
topic_category_id: 2
```

13.3.2 List Instruments

With PySDK

```
$ t.streams.list_instruments(project_uuid='tapis_demo_project_testuser6', site_id=
↪'tapis_demo_site')
```

With CURL:

```
$ curl -H "X-Tapis-Token:$jwt"  $BASE_URL/v3/streams/projects/tapis_demo_project_
↪testuser6/sites/tapis_demo_site/instruments
```

The response will look something like the following:

```
[
 chords_id: 10
 created_at: 2020-07-20 20:09:11.990814
 inst_description: demo instrument
 inst_id: tapis_demo_instrument
 inst_name: tapis_demo_instrument
 topic_category_id: 2,

 chords_id: 11
 created_at: 2020-07-20 20:14:20.512383
 inst_description: demo instrument
 inst_id: tapis_demo_instrument
 inst_name: tapis_demo_instrument1
 project_uuid: tapis_demo_project_testuser6
 site_id: tapis_demo_site
 topic_category_id: 2,

 chords_id: 12
 created_at: 2020-07-20 20:20:45.171473
 inst_description: demo instrument
 inst_id: demo_instrument
 inst_name: demo_instrument
 topic_category_id: 2,

 chords_id: 13
 created_at: 2020-07-20 20:21:52.842495
 inst_description: demo instrument
 inst_id: demo_instrument_aj
 inst_name: demo_instrument_aj
 topic_category_id: 2]
```

13.3.3 Get instrument Details

With PySDK

```
$ t.streams.list_instruments(project_uuid='tapis_demo_project_testuser6', site_id=
↪'tapis_demo_site',inst_id='demo_instrument')
```

With CURL:

```
$ curl -H "X-Tapis-Token:$jwt"  $BASE_URL/v3/streams/projects/tapis_demo_project_
↪testuser6/sites/tapis_demo_site/instruments/demo_instrument
```

The response will look something like the following:

```
chords_id: 12
created_at: 2020-07-20 20:20:45.171473
```

(continues on next page)

(continued from previous page)

```
inst_description: demo instrument
inst_id: demo_instrument
inst_name: demo_instrument
topic_category_id: 2
```

13.3.4 Update Instrument

With PySDK

```
$ t.streams.update_instrument(inst_id= 'Ohio_River_Robert_C_Byrd_Locks', project_uuid=
↪'wq_demo_tapis_streams_proj2020-08-26T08:41:11.813391', site_id='wq_demo_site',␣
↪inst_name='test', inst_description='test')
```

With CURL:

```
$ curl -X PUT -H "X-Tapis-token:$jwt" -H "Content-Type:application/json" --data '{
↪"inst_id": "Ohio_River_Robert_C_Byrd_Locks",
"site_id": "wq_demo_site", "inst_name": "UpdatedNAME","inst_description": "updated␣
↪descript"}'
$BASE_URL/v3/streams/projects/wq_demo_tapis_streams_proj2020-08-26T08:41:11.813391/
↪sites/wq_demo_site/instruments/Ohio_River_Robert_C_Byrd_Locks'
```

The response will look something like the following:

```
chords_id: 6
inst_description: test
inst_id: Ohio_River_Robert_C_Byrd_Locks
inst_name: test
site_chords_id: 7
updated_at: 2020-08-26 18:40:07.534077
variables: [
chords_id: 21
shortname: temp
updated_at: 2020-08-26 16:15:49.835211
var_id: temp
var_name: temperature,
chords_id: 22
shortname: bat
updated_at: 2020-08-26 16:15:50.349601
var_id: batv
var_name: battery,
chords_id: 23
shortname: spc
updated_at: 2020-08-26 16:15:50.749192
var_id: spc
var_name: specific_conductivity,
chords_id: 24
shortname: turb
updated_at: 2020-08-26 16:15:51.158687
var_id: turb
var_name: turbidity,
```

(continues on next page)

(continued from previous page)

```
chords_id: 25
shortname: ph
updated_at: 2020-08-26 16:15:51.588573
var_id: ph
var_name: ph_level]
```

13.3.5 Delete Instrument

With PySDK

```
$ t.streams.delete_instrument(inst_id= 'tapis_demo_instrument', project_uuid='tapis_
↪demo_project_testuser6_3', site_id='tapis_demo_site')
```

With CURL:

```
$ curl -X DELETE -H "X-Tapis-token:$jwt" $BASE_URL/v3/streams/projects/tapis_demo_
↪project_testuser6_3/sites/tapis_demo_site/instruments/tapis_demo_instrument
```

13.4 Variables

13.4.1 Create Variables

With PySDK

```
$ t.streams.create_variable(project_uuid='tapis_demo_project_testuser6', topic_
↪category_id='2', site_id='tapis_demo_site', inst_id='demo_instrument', var_name=
↪'battery', shortname='bat', var_id='batv')
```

With CURL:

```
$ curl -v -X POST -H "Content-Type:application/json" -H "X-Tapis-Token:$jwt" --data '{
↪"project_uuid":"tapis_demo_project_testuser6", "topic_category_id":"2","site_id":
↪"tapis_demo_site", "inst_id":"demo_instrument", "var_name":"battery", "shortname":
↪"bat", "var_id":"batv"}'  $BASE_URL/v3/streams/projects/tapis_demo_project_
↪testuser6/sites/tapis_demo_site/instruments/demo_instrument/variables
```

The response will look something like the following:

```
chords_id: 39
shortname: bat
updated_at: 2020-07-20 21:51:38.712035
var_id: batv
var_name: battery
```

13.4.2 List Variables

With PySDK

```
$ t.streams.list_variables(project_uuid='tapis_demo_project_testuser6',site_id='tapis_
↪demo_site', inst_id='demo_instrument')
```

With CURL:

```
$ curl -H "Content-Type:application/json" -H "X-Tapis-Token:$jwt"  $BASE_URL/v3/
↪streams/projects/tapis_demo_project_testuser6/sites/tapis_demo_site/instruments/
↪demo_instrument/variables
```

The response will look something like the following:

```
[
 chords_id: 38
 shortname: bat
 updated_at: 2020-07-20 21:50:46.382558
 var_id: batv
 var_name: battery,

 chords_id: 39
 shortname: bat
 updated_at: 2020-07-20 21:51:38.712035
 var_id: batv
 var_name: battery,

 chords_id: 40
 inst_id: demo_instrument_1
 project_uuid: tapis_demo_project_testuser6
 shortname: bat
 site_id: tapis_demo_site
 topic_category_id: 2
 updated_at: 2020-07-20 21:56:45.555381
 var_id: batv
 var_name: battery]
```

13.4.3 Get Variable Details

With PySDK

```
$ t.streams.get_variable(project_uuid='tapis_demo_project_testuser6_1', site_id=
↪'tapis_site_final', inst_id='tapis_inst_final', var_id='batv')
```

With CURL:

```
$ curl -H "X-Tapis-Token:$jwt"  $BASE_URL/v3/streams/projects/tapis_demo_project_
↪testuser6_1/sites/tapis_site_final/instruments/tapis_inst_final/variables/batv
```

The response will look something like the following:

```
[
chords_id: 21
shortname: bat
updated_at: 2020-08-18 20:46:11.673033
var_id: batv
var_name: battery]
```

13.4.4 Update Variable

With PySDK

```
$ t.streams.update_variable(var_name='"updated_temp', var_id='temp', shortname='temp_
↪updated', project_uuid='wq_demo_tapis_streams_proj2020-08-25T16:21:30.113392', site_
↪id='wq_demo_site',inst_id='Ohio_River_Robert_C_Byrd_Locks')
```

With CURL:

```
$ curl -X PUT -H "X-Tapis-token:$jwt" -H "Content-type:application/json"  --data '{
↪"var_name": "updated_temp","var_id": "temp","shortname":"temp_updated"}' $BASE_URL/
↪v3/streams/projects/wq_demo_tapis_streams_proj2020-08-25T16:21:30.113392/sites/wq_
↪demo_site/instruments/Ohio_River_Robert_C_Byrd_Locks/variables/temp
```

The response will look something like the following:

```
chords_id: 16
inst_chords_id: 5
shortname: temp_updated
site_chords_id: 6
updated_at: 2020-08-27 14:36:04.271154
var_id: temp
var_name: "updated_temp
```

13.4.5 Delete Variable

With PySDK

```
$ t.streams.delete_variable( var_id='139', project_uuid='tapis_demo_instrument', site_
↪id='tapis_demo_site',inst_id='tapis_demo_instrument')
```

With CURL:

```
$ curl -v -X DELETE  -H "Content-Type:application/json" -H "X-Tapis-Token:$jwt"
↪$BASE_URL/v3/streams/projects/tapis_demo_project_testuser6_3/sites/tapis_demo_site/
↪instruments/tapis_demo_instrument/variables/batv
```

The response will look something like the following:

```
inst_chords_id: 24
updated_at: 2020-12-03 02:52:27.437378
var_id: 139
```

13.5 Measurements

13.5.1 Create Measurements

With PySDK

```
$ t.streams.create_measurement(inst_id='demo_instrument',vars=[{"var_id": "batv",
↪"value": 10}],datetime='2020-07-20T22:19:25Z')
```

With CURL:

```
$ curl -v -X POST -H "Content-Type:application/json" -H "X-Tapis-Token:$jwt" --data '{
↪"inst_id":"demo_instrument", "datetime":"2020-07-20T23:19:25Z", "vars":[{"var_id":
↪"batv", "value": 10}]}'  $BASE_URL/v3/streams/measurements
```

The response will look something like the following:

```
{'message': 'Measurements Saved',
'result': [],
'status': 'success',
'version': 'dev'}
```

13.5.2 List Measurements

With PySDK

```
$ t.streams.list_measurements(inst_id='demo_instrument',start_date='2020-05-
↪08T00:00:00Z',end_date='2020-07-21T22:19:25Z', format='csv',project_uuid='tapis_
↪demo_project_testuser6',site_id='tapis_demo_site')
```

With CURL:

```
$ curl -H "X-Tapis-Token:$jwt"  $BASE_URL/v3/streams/measurements/demo_instrument
```

The response will look something like the following:

```
b'time,batv\n2020-07-20T22:19:25Z,10.0\n2020-07-20T23:19:25Z,10.0\n'
```

13.6 Channels

13.6.1 Create Channels

With PySDK

```
$ t.streams.create_channels(channel_id="demo.tapis.channel", channel_name='demo.tapis.
↪channel', template_id="demo_channel_template",triggers_with_actions=[{"inst_ids":[
↪"demo_instrument"],"condition":{"key":"demo_instrument.batv","operator":">", "val
↪":20},"action":{"method":"ACTOR","actor_id" :"XXXX","message":"Instrument: demo_
↪instrument exceeded threshold", "abaco_base_url":"https://api.tacc.utexas.edu",
↪"nonces":"XXXX-YYYY-ZZZZ" }}])
```

With CURL:

```
$ curl -v -X POST -H "Content-Type:application/json" -H "X-Tapis-Token:$jwt" --data '{
↪"channel_id":"demo.tapis.channel","channel_name":"demo.tapis.channel_1","template_id
↪":"demo_channel_template","triggers_with_actions":[{"inst_ids":["demo_instrument"],
↪"condition":{"key":"demo_instrument.batv","operator":">", "val":"20"}, "action":{
↪"method":"ACTOR","actor_id" :"XXXX","message":"Instrument: demo_instrument batv␣
↪exceeded threshold", "abaco_base_url":"https://api.tacc.utexas.edu","nonces":"XXXX-
↪YYYY-ZZZZ"}}]}'  $BASE_URL/v3/streams/channels
```

The response will look something like the following:

```
channel_id: demo.tapis.channel
channel_name: demo.tapis.channel
create_time: 2020-07-21 03:02:51.755215
last_updated: 2020-07-21 03:02:51.755227
permissions:
users: ['testuser6']
status: ACTIVE
template_id: demo_channel_template
triggers_with_actions: [
action:
abaco_base_url: https://api.tacc.utexas.edu
actor_id: XXXX
message: Instrument: demo_instrument exceeded threshold
method: ACTOR
nonces: XXXX-YYYY-ZZZZ
condition:
key: demo_instrument.batv
operator: >
val: 20
inst_ids: ['demo_instrument']]
```

13.6.2 List Channels

With PySDK

```
$ t.streams.list_channels()
```

With CURL:

```
$ curl -H "X-Tapis-Token:$jwt"  $BASE_URL/v3/streams/channels
```

The response will look something like the following:

```
{'message': 'Channels found',
 'result': [],
 'status': 'success',
 'version': 'dev'}
```

13.6.3 Get Channel Details

With PySDK

```
$ t.streams.get_channel(channel_id='demo.tapis.channel')
```

With CURL:

```
$ curl -H "X-Tapis-Token:$jwt"  $BASE_URL/v3/streams/channels/demo.tapis.channel
```

The response will look something like the following:

```
channel_id: demo.tapis.channel
channel_name: demo.tapis.channel
create_time: 2020-07-21 03:02:51.755215
last_updated: 2020-07-21 03:02:51.755227
permissions:
users: ['testuser6']
status: ACTIVE
template_id: demo_channel_template
triggers_with_actions: [
action:
abaco_base_url: https://api.tacc.utexas.edu
actor_id: XXXX
message: Instrument: demo_instrument exceeded threshold
method: ACTOR
nonces: XXXX-YYYY-ZZZZ
condition:
    key: demo_instrument.batv
    operator: >
    val: 20
    inst_ids: ['demo_instrument']]
```

13.6.4 Update Channels:

With PySDK

```
$ t.streams.update_channel(channel_id="test1", channel_name='demo.wq.channel',␣
↪template_id="demo_channel_template",triggers_with_actions=[{"inst_ids":[
"Ohio_River_Robert_C_Byrd_Locks"],"condition":{"key":"Ohio_River_Robert_C_Byrd_Locks.
↪temp","operator":">", "val":30},
"action":{"method":"ACTOR","actor_id" :"XXXX","message":"Instrument: Ohio_River_
↪Robert_C_Byrd_Locks  exceeded threshold",
"abaco_base_url":"https://api.tacc.utexas.edu","nonces":"XXXX-YYYY-ZZZZ" }}])
```

With CURL:

```
$ curl -X PUT -H "X-Tapis-Token:$jwt" -H "Content-Type:application/json" $BASE_URL/v3/
↪streams/channels/test1 -d '{"channel_id": "test1","channel_name":"demo.wq.channel",
↪"template_id": "demo_channel_template",
"triggers_with_actions": [{"inst_ids": ["Ohio_River_Robert_C_Byrd_Locks" ],
"condition": {"key": "Ohio_River_Robert_C_Byrd_Locks.temp","operator": ">","val": "40
↪" } }]}'
```

The response will look something like the following:

```
channel_id: test1
channel_name: demo.wq.channel
create_time: 2020-08-18 20:51:41.350377
last_updated: 2020-08-18 21:57:42.174860
permissions:
users: ['testuser2']
status: ACTIVE
template_id: demo_channel_template
triggers_with_actions: [
action:
abaco_base_url: https://api.tacc.utexas.edu
actor_id: XXXX
message: Instrument: Ohio_River_Robert_C_Byrd_Locks  exceeded threshold
method: ACTOR
nonces: XXXX-YYYY-ZZZZ
condition:
key: Ohio_River_Robert_C_Byrd_Locks.temp
operator: >
val: 30
inst_ids: ['Ohio_River_Robert_C_Byrd_Locks']]
```

13.6.5 Update Channels Status

With PySDK

```
$ t.streams.update_status(channel_id='demo.tapis.channel', status='INACTIVE')
```

With CURL:

```
$ curl -X POST -H "Content-Type:application/json" -H "X-Tapis-Token:$jwt" -d '{"status
↪":"INACTIVE"}' $BASE_URL/v3/streams/channels/demo.tapis.channel
```

The response will look something like the following:

```
channel_id: demo.tapis.channel
channel_name: demo.tapis.channel
create_time: 2020-07-21 03:02:51.755215
last_updated: 2020-07-22 18:09:19.940080
permissions:
users: ['testuser6']
status: INACTIVE
template_id: demo_channel_template
triggers_with_actions: [
action:
abaco_base_url: https://api.tacc.utexas.edu
actor_id: XXXX
message: Instrument: demo_instrument exceeded threshold
method: ACTOR
nonces: XXXX-YYYY-ZZZZ
condition:
key: demo_instrument.batv
operator: >
val: 90
inst_ids: ['demo_instrument']]
```

13.7 Templates

13.7.1 Create Template

With PySDK

```
$ t.streams.create_template(template_id='test_template_for_tutorial', type='stream',
        script=' var crit lambda \n var channel_id string\n stream\n    |from()\n
↪    .measurement(\'tsdata\')\n        '
                ' .groupBy(\'var\')\n   |alert()\n       '
                ' .id(channel_id +  \' {{ .Name }}/{{ .Group }}/{{.TaskName}}/{{index .
↪Tags \"var\" }}\')\n         .crit(crit)\n    .noRecoveries()\n       '
                '  .message(\'{{.ID}} is {{ .Level}} at time: {{.Time}} as value: {{
↪index .Fields \"value\" }} exceeded the threshold\')\n       '
                ' .details(\'\')\n         .post()\n         .endpoint(\'api-alert\
↪')\n     .captureResponse()\n    |httpOut(\'msg\')', _tapis_debug=True)
```

The response will look something like the following:

```
create_time: 2020-07-22 15:30:58.244391
last_updated: 2020-07-22 15:30:58.244407
permissions:
users: ['testuser6']
script:  var crit lambda
 var channel_id string
 stream
    |from()
        .measurement('tsdata')
         .groupBy('var')
   |alert()
```

(continues on next page)

(continued from previous page)

```
        .id(channel_id +  ' {{ .Name }}/{{ .Group }}/{{.TaskName}}/{{index .Tags "var
↪" }}')
         .crit(crit)
    .noRecoveries()
        .message('{{.ID}} is {{ .Level}} at time: {{.Time}} as value: {{ index .
↪Fields "value" }} exceeded the threshold')
        .details('')
         .post()
         .endpoint('api-alert')
     .captureResponse()
    |httpOut('msg')
template_id: test_template_for_tutorial
type: stream
```

13.7.2 List Templates

With PySDK

```
$ t.streams.list_templates()
```

With CURL:

```
$ curl -H "X-Tapis-Token:$jwt"  $BASE_URL/v3/streams/templates
```

The response will look something like the following:

13.7.3 Get Template Details

With PySDK

```
$ t.streams.get_template(template_id='test_template_for_tutorial')
```

With CURL:

```
$ curl  -H "X-Tapis-Token:$jwt" $BASE_URL/v3/streams/templates/test_template_for_
↪tutorial
```

The response will look something like the following:

```
create_time: 2020-07-22 15:30:58.244391
last_updated: 2020-07-22 15:30:58.244407
permissions:
users: ['testuser6']
script:  var crit lambda
 var channel_id string
 stream
```

(continues on next page)

(continued from previous page)

```
    |from()
        .measurement('tsdata')
         .groupBy('var')
   |alert()
        .id(channel_id +  ' {{ .Name }}/{{ .Group }}/{{.TaskName}}/{{index .Tags "var
↪" }}')
         .crit(crit)
    .noRecoveries()
        .message('{{.ID}} is {{ .Level}} at time: {{.Time}} as value: {{ index .
↪Fields "value" }} exceeded the threshold')
        .details('')
         .post()
         .endpoint('api-alert')
     .captureResponse()
    |httpOut('msg')
template_id: test_template_for_tutorial
type: stream
```

13.7.4 Update Template

With PySDK

```
t.streams.update_template(template_id='test_template_for_tutorial', type='stream',
        script=' var period=5s\n var every=0s\n var crit lambda \n var channel_id␣
↪string\n stream\n    |from()\n        .measurement(\'tsdata\')\n        '
               ' .groupBy(\'var\')\n   |alert()\n       '
               ' .id(channel_id +  \' {{ .Name }}/{{ .Group }}/{{.TaskName}}/{{index .
↪Tags \"var\" }}\')\n         .crit(crit)\n    .noRecoveries()\n      '
               '  .message(\'{{.ID}} is {{ .Level}} at time: {{.Time}} as value: {{␣
↪index .Fields \"value\" }} exceeded the threshold\')\n       '
               ' .details(\'\')\n         .post()\n         .endpoint(\'api-alert\
↪')\n     .captureResponse()\n    |httpOut(\'msg\')', _tapis_debug=True)
```

The response will look something like the following:

```
create_time: 2020-08-19 19:48:59.177935
last_updated: 2020-08-19 19:50:00.102827
permissions:
users: ['testuser2']
script:  var period=5s
 var every=0s
 var crit lambda
 var channel_id string
 stream
    |from()
        .measurement('tsdata')
         .groupBy('var')
   |alert()
        .id(channel_id +  ' {{ .Name }}/{{ .Group }}/{{.TaskName}}/{{index .Tags "var
↪" }}')
         .crit(crit)
```

(continues on next page)

(continued from previous page)

```
    .noRecoveries()
        .message('{{.ID}} is {{ .Level}} at time: {{.Time}} as value: {{ index .
↪Fields "value" }} exceeded the threshold')
        .details('')
         .post()
         .endpoint('api-alert')
     .captureResponse()
    |httpOut('msg')
template_id: test_template_update
type: stream
```

13.8 Alerts

13.8.1 List Alerts

With PySDK

```
$ t.streams.list_alerts(channel_id='demo_wq_channel2020-06-19T17_34_46.425419')
```

With CURL:

```
$ curl  -H "X-Tapis-Token:$jwt" $BASE_URL/v3/streams/channels/demo_wq_channel2020-06-
↪19T17_34_46.425419/alerts
```

The response will look something like the following:

```
alerts: [
    actor_id: XXXX
    alert_id: 70fa63b4-c6b1-45a4-91a8-f4e9803ec898
    channel_id: demo_wq_channel2020-06-19T17_34_46.425419
    channel_name: demo.wq.channel
    create_time: 2020-06-19 20:51:44.390887
    execution_id: 7mBGaJbD4q0M1
    message: demo_wq_channel2020-06-19T17_34_46.425419 tsdata/var=11/demo_wq_
↪channel2020-06-19T17_34_46.425419/11 is CRITICAL at time: 2020-06-19 20:51:43.
↪229988 +0000 UTC as value: 150 exceeded the threshold,
    actor_id: XXXX
    alert_id: c16ab843-8417-4af0-a06c-ce1e4e7e4816
    channel_id: demo_wq_channel2020-06-19T17_34_46.425419
    channel_name: demo.wq.channel
    create_time: 2020-06-19 20:51:21.138143
    execution_id: ByOkp5W8Jxkqj
    message: demo_wq_channel2020-06-19T17_34_46.425419 tsdata/var=11/demo_wq_
↪channel2020-06-19T17_34_46.425419/11 is CRITICAL at time: 2020-06-19 20:51:20.
↪114319 +0000 UTC as value: 150 exceeded the threshold,
    actor_id: XXXX
    alert_id: 4c4b7e70-a034-419b-be8c-2c337803e5d4
    channel_id: demo_wq_channel2020-06-19T17_34_46.425419
    channel_name: demo.wq.channel
    create_time: 2020-06-19 20:51:10.454269
```

(continues on next page)

(continued from previous page)

```
    execution_id: jboJWNqRKAA6V
    message: demo_wq_channel2020-06-19T17_34_46.425419 tsdata/var=11/demo_wq_
↪channel2020-06-19T17_34_46.425419/11 is CRITICAL at time: 2020-06-19 20:51:09.
↪862752 +0000 UTC as value: 150 exceeded the threshold]
    num_of_alerts: 3
]
```

13.9 Roles

Streams service uses **roles** to manage permissions on the streams resources. CRUD operations on Streams resources such as Sites, Instruments and Variables can be performed by authorized users having a specific role on the Project. Similarly CRUD operations on Channels and Templates can be done by authorized users having specific roles. Streams service supports three types of roles: *admin*, *manager* and *user*.

Admin has elevated privileges. An *admin* can create, update, or delete any of the Streams resources.

Manager can perform all read and write operations on Streams resources, with an exception of deleting them.

User can only perform read operations on the resources and are not authorized to write or delete the resources.

Table 1 below summarizes the authorized actions with respect to user roles.

Role	Request permitted
admin	GET, PUT, POST, DELETE
manager	GET, PUT, POST
user	GET

When a user creates project, channel or template, an admin role of the form: **streams_projects_$project-oid_admin**, **streams_channels_$channel-oid_admin** or **streams_templates_$template-oid_admin**, respectively is created in the Security Kernel and is assigned to the requesting (JWT) user. Oid is the unique object id generated by the backend MongoDB for each of the Project, Channel or Template.

Admins can further grant roles such as **admin**, **manager** or **user** for other users listed on the project. To perform CRUD operations on Projects, Sites, Instruments and Variables, users must have appropriate role on the Project. To perform CRUD operation on either Channels and Templates, users must have role associated with each of the resources.

13.9.1 List Roles

To get the list of user roles on a project, channel or template, the requesting(JWT) user must provide following three parameters:

1) **resource_type** : project/channel/template
2) **resource_id**: project_id/channel_id/template_id
3) **user**: username for whom roles are to be checked

In order to list the user roles on a resource (Project, Channel, Template) the requesting(JWT) user must have one of the three roles (admin, manager, user) on it.

With PySDK

```
$ t.streams.list_roles(resource_id=<resource_id>, user=<username>,resource_type=
↪'project')
$ t.streams.list_roles(resource_id=<resource_id>, user=<username>,resource_type=
↪'channel')
$ t.streams.list_roles(resource_id=<resource_id>, user=<username>,resource_type=
↪'template')
```

With CURL:

```
$ curl -H "X-Tapis-Token:$jwt" {BASE_URL}/v3/streams/roles?user={userid}&resource_
↪type={project/channel/template}&resource_id={project_id/channel_id/template_id}
```

The response will look like the following with the Python Client:

```
result: ['admin']
```

There are three possible responses depending on if the requesting(JWT) user and user specified in query parameters are same or different.

Case I: When requesting(JWT) user and user specified in the query parameters are same and both have role on the project/channel/template The result will include all the roles for the user in query parameters for the given resource_id


```
{
 "message": "Roles found",
 "result": [
    "admin"
],
"status": "success",
"version": "dev"
}
```


Case II: When requesting(JWT) user and user specified in the query parameters are different and JWT user does not have any role on the project/channel/template


```
{
   "message": "User not authorized to access roles",
   "result": "",
   "status": "success",
   "version": "dev"
}
```


Case III: When requesting(JWT) user and user specified in the query parameters are different. JWT user has role on the project and user in query parameter does not have role on the project/channel/template


```
{
   "message": "Roles not found",
   "result": "",
   "status": "success",
   "version": "dev"
}
```

13.9.2 Grant Roles

Roles can be granted by Project/Channel/Template *admins* or *managers* so that users can perform CRUD operations on the Streams resources.

Table 2 below shows that *admin* can grant any of the three roles to other users. Same or lower level permissions can be granted by *admins* and *managers*. Self role granting is not permitted.

Managers can only grant *manager* and *user* to other users.

Users do **not** have privileges to grant roles.

- Roles of the requesting(JWT) user are first checked by querying SK.
- If the username provided in the request body is the same as the JWT user, then self role granting is not permitted.
- If the JWT user and user provided in the request body are different, then existing roles for the username provided in the request body are retrieved and if the user already has the role, JWT user user is asking for no action is taken.
- If the role does not exist then JWT user roles are retrieved and compared with the rolename provided in the request body. Role is granted only if the JWT user has **same** or **higher** roles than the role name specified in the request body (*admin* role has highest rank, followed by *manager* and then *user*). Otherwise an error message saying, *User not authorized to grant role* is given in the response.

Role	Grant
admin	admin, manager, user
manager	manager, user
user	cannot grant roles

With PySDK

```
$ t.streams.grant_role(resource_id=<resource_id>, user=<user>,resource_type='project/
↪channel/template',role_name='admin/manager/user')
```

With CURL:

```
$ curl -X POST -H "X-Tapis-Token:$jwt" {BASE_URL}/v3/streams/roles

Request body: { "user":"user_id",
                "resource_type":"project/channel/template",
                "resource_id":"project_uuid/channel_id/template_id",
                "role_name": "admin/manager/user"
               }
```

The response will vary based on following cases:

Case I: If the username provided in the request body is the same as the JWT user, then self role granting is not permitted.

With PySDK

```
$ t.streams.grant_role(resource_id='test_proj', user='testuser2',resource_type=
↪'project',role_name='manager')
```

```
{'message': 'Cannot grant role for self',
 'metadata': {},
 'result': '',
```

(continues on next page)

(continued from previous page)

```
 'status': 'error',
 'version': 'dev'}
```

Case II: If the JWT user and username provided in the request body are different, then existing roles for the username provided in the request body are retrieved and if the user already has the role JWT user user is asking for, no action is taken.

With PySDK

```
$ t.streams.grant_role(resource_id='test_proj', user='testuser6',resource_type=
↪'project',role_name='manager')
```

```
{
    "message": "Role already exists",
    "metadata": {},
    "result": [
        "manager"
    ],
    "status": "success",
    "version": "dev"
}
```

Case III: If the role does not exist then JWT user roles are retrieved and compared with the rolename provided in the request body. Role is granted only if the JWT user has **same** or **higher*** roles than the role name specified in the request body. Otherwise an error message saying, "User not authorized to grant role" is given in the response.

For example testuser4 has ***manager** role on the project and the request is to grant testuser5 **admin*** role, then the request will not be fulfilled.

```
$ t.streams.grant_role(resource_id='test_proj', user='testuser5',resource_type=
↪'project',role_name='admin')
```

```
{
   "message": "Role admin cannot be granted",
   "result": "",
   "status": "error",
   "version": "dev"
}
```

If the requesting (JWT) user only has a **user*** role, then no role can be granted to other users, and the response will be following

```
{
   "message": "Role manager cannot be granted",
   "result": "",
   "status": "error",
   "version": "dev"
}
```

Case IV: If the requesting (JWT) user has no role on the project/channel/template, then the user is not authorized to grant any roles

```
{
   "message": "User not authorized to grant role",
   "result": "",
   "status": "error",
```

(continues on next page)

(continued from previous page)

```
    "version": "dev"
}
```

13.9.3 Revoke Roles

Users in **admin** role are capable of revoking any of the three roles: **admin**, **manager** and **user** for other users. Self role revoking is not permitted. Users in *manager* and *user* role are not capable of revoking roles.

With PySDK

```
$ permitted_client.streams.revoke_role(resource_id='test_proj', user='testuser6',
↪resource_type='project',role_name='manager')
```

With CURL:

```
$ curl -X POST -H "X-Tapis-Token:$jwt" {BASE_URL}/v3/streams/revokeRole

Request body: { "user":"user_id",
                "resource_type":"project/channel/template",
                "resource_id":"project_uuid/channel_id/template_id",
                "role_name": "admin/manager/user"
               }
```

The response will be following:

```
{'message': 'Role manager successfully deleted for user testuser6',
'metadata': {},
'result': '',
'status': 'success',
'version': 'dev'}
```

Responses will vary based on following cases:

Case I: If the requesting (JWT) user has a **manager** or **user** role

```
{
    "message": "User not authorized to revoke role",
    "result": "",
    "status": "error",
    "version": "dev"
}
```

Case II: If the JWT user is trying to revoke self role

```
{
    "message": "Cannot delete role for self",
    "result": "",
    "status": "error",
    "version": "dev"
}
```