abaco Documentation

Joe Stubbs,KeDarius Whitley

Oct 28, 2019

Introduction:

CHAPTER 1

Welcome to Abaco

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.

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*.

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 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.

2.1 Account Creation and Software Installation

2.1.1 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 .

2.1.2 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 .

2.1.3 Install the TACC Cloud Python SDK

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

```
$ pip3 install agavepy
```

Attention: While `agavepy` works with both Python 2 and 3 we strongly recommend using Python 3.

2.2 Working with TACC OAuth

Authentication and authorization to the TACC Cloud 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 TACC Cloud while keeping your access credentials and digital assets secure. This is covered in great detail in our Developer Documentation but some key concepts will be highlighted here, interleaved with Python code.

2.2.1 Create an OAuth Client

The first step is to create an OAuth client. This is a one-time set up step, much like creating a TACC account. To do it, we will use the TACC Cloud API Python SDK. First, import the `Agave` class and create python object called `ag` that points to the TACC Cloud API server using your TACC username and password. Do so by typing the following in a Python shell:

```
>>> from agavepy.agave import Agave
>>> ag = Agave(api_server='https://api.tacc.utexas.edu',
...            username='your username',
...            password='your password')
```

Once the object is instantiated, interact with it according to the API documentation and your specific usage needs. For example, to create a new OAuth client we type the following:

```
>>> ag.clients.create(body={'clientName': 'enter a client name'})
```

You should see a response like:

```
{'_links': {'self': {'href': 'https://api.tacc.utexas.edu/clients/v2/abaco_quickstart
↪'},
 'subscriber': {'href': 'https://api.tacc.utexas.edu/profiles/v2/apitest'},
 'subscriptions': {'href': 'https://api.tacc.utexas.edu/clients/v2/abaco_quickstart/
↪subscriptions/'}},
 'callbackUrl': '',
 'consumerKey': 'pYV81QNBxkqeC6Nms3XBzk9UJuca',
 'consumerSecret': 'Oug0gdLa3a_Xt37_fwxO6ZGNffUa',
 'description': '',
 'name': 'abaco_quickstart',
 'tier': 'Unlimited'}
```

Record the consumerKey and consumerSecret in a secure place; you will use them over and over to generate Oauth tokens, which are temporary credentials that you can use in place of putting your real credentials into code that is scripting against the TACC APIs.

2.2.2 Reuse an Existing Oauth Client

Once you generate an OAuth client, you can re-use its key and secret:

```
>>> from agavepy.agave import Agave
>>> ag = Agave(api_server='https://api.tacc.utexas.edu',
...            username='your username', password='your password',
...            client_name='my_client',
...            api_key='pYV81QNBxkqeC6Nms3XBzk9UJuca',
...            api_secret='Oug0gdLa3a_Xt37_fwxO6ZGNffUa')
```

2.2.3 Generate a Token

With the `ag` object instantiated and an OAuth client created, we are ready to generate an OAuth token:

```
>>> ag.token.create()
Out[1]: 'c21199177da6dd4d14d659399a933f5'
```

Note that the token is automatically stored on the `ag` object for you. You are now ready to check your access to the TACC Cloud APIs.

2.2.4 Check Access to the TACC Cloud APIs

The Agave object `ag` should now be configured to talk to all TACC Cloud APIs on your behalf. We can check that our client is configured properly by making any API call. Here's an example: Let's retrieve the current user's **profile**.

```
>>> ag.profiles.get()
Out[1]:
{'email': 'aci-cic@tacc.utexas.edu',
 'first_name': 'API',
```

(continues on next page)

(continued from previous page)

```
 'full_name': 'API Test',
 'last_name': 'Test',
 'mobile_phone': '',
 'phone': '',
 'status': '',
 'uid': 834517,
 'username': 'apitest'}
```

2.3 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.

2.3.1 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 `agavepy`.

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 agavepy.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)
```

2.3.2 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 `agavepy` library in our image:

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

(note: there is a `abacosample` image that contains Python and the agavepy library; see *Abaco Samples* for more details)

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 agavepy
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
```

2.3.3 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 `Agave` client object we created above (see *Working with TACC OAuth*).

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

```
>>> from agavepy.agave import Agave
>>> ag = Agave(api_server='https://api.tacc.utexas.edu', token='<access_token>')
```

(continues on next page)

(continued from previous page)

```
>>> my_actor = {"image": "user/my_actor", "name": "word_counter", "description":
↪"Actor that counts words."}
>>> ag.actors.add(body=my_actor)
```

You should see a response like this:

```
{'_links': {'executions': 'https://api.tacc.utexas.edu/actors/v2/O08Nzb3mRA7Bz/
↪executions',
'owner': 'https://api.tacc.utexas.edu/profiles/v2/jstubbs',
'self': 'https://api.tacc.utexas.edu/actors/v2/O08Nzb3mRA7Bz'},
'createTime': '2018-07-03 22:41:29.563024',
'defaultEnvironment': {},
'description': 'Actor that counts words.',
'id': 'O08Nzb3mRA7Bz',
'image': 'abacosamples/wc',
'lastUpdateTime': '2018-07-03 22:41:29.563024',
'mounts': [],
'name': 'word_counter',
'owner': 'jstubbs',
'privileged': False,
'state': {},
'stateless': False,
'status': 'SUBMITTED',
'statusMessage': '',
'type': 'none',
'useContainerUid': False}
```

Notes:

- Abaco assigned an id to the actor (in this case `O08Nzb3mRA7Bz`) 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:

```
>>> ag.actors.get(actorId='O08Nzb3mRA7Bz')
```

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

2.3.4 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, Executions, and Logs* section.

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

```
>>> ag.actors.sendMessage(actorId='O08Nzb3mRA7Bz',
                          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://api.tacc.utexas.edu/actors/v2/O08Nzb3mRA7Bz/messages
↪',
  'owner': 'https://api.tacc.utexas.edu/profiles/v2/jstubbs',
  'self': 'https://api.tacc.utexas.edu/actors/v2/O08Nzb3mRA7Bz/executions/
↪kA1P1m8NkkolK'},
 '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:

```
>>> ag.actors.getExecution(actorId='O08Nzb3mRA7Bz', executionId='kA1P1m8NkkolK')
```

The response will be similar to the following:

```
{'_links': {'logs': 'https://api.tacc.utexas.edu/actors/v2/TACC-PROD_O08Nzb3mRA7Bz/
↪executions/kA1P1m8NkkolK/logs',
  'owner': 'https://api.tacc.utexas.edu/profiles/v2/jstubbs',
  'self': 'https://api.tacc.utexas.edu/actors/v2/TACC-PROD_O08Nzb3mRA7Bz/executions/
↪kA1P1m8NkkolK'},
 'actorId': 'O08Nzb3mRA7Bz',
 'apiServer': 'https://api.tacc.utexas.edu',
 'cpu': 0,
 'executor': 'jstubbs',
 'exitCode': 1,
 'finalState': {'Dead': False,
  'Error': '',
  'ExitCode': 1,
  'FinishedAt': '2018-07-03T22:56:30.605256563Z',
  'OOMKilled': False,
  'Paused': False,
  'Pid': 0,
  'Restarting': False,
  'Running': False,
  'StartedAt': '2018-07-03T22:56:30.474917256Z',
  'Status': 'exited'},
 'id': 'kA1P1m8NkkolK',
 'io': 0,
 'messageReceivedTime': '2018-07-03 22:56:29.075122',
 'runtime': 1,
 'startTime': '2018-07-03 22:56:29.558470',
 'status': 'COMPLETE',
 'workerId': 'e7B3JXDNxM6M0'}
```

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

2.3.5 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`:

```
>>> ag.actors.getExecutionLogs(actorId='O08Nzb3mRA7Bz', executionId='kA1P1m8NkkolK')
```

The response should be similar to the following:

```
{'_links': {'execution': 'https://api.tacc.utexas.edu/actors/v2/6PlMbDLa4zlON/
↪executions/kGQk6RRJQBL3',
  'owner': 'https://api.tacc.utexas.edu/profiles/v2/jstubbs',
  'self': 'https://api.tacc.utexas.edu/actors/v2/6PlMbDLa4zlON/executions/
↪kGQk6RRJQBL3/logs'},
 'logs': 'Number of words is: 5\n'}
```

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

2.3.6 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.

CHAPTER 3

Overview

The Technical Guide for Abaco provides a more detailed reference to Abaco's advanced features.

- *Actor Registration*: Complete reference for actor registration and management.
- *Messages, Executions, and Logs*: Covers the different types of messages that can be sent to an Actor.
- *Abaco Context & Container Runtime*: Full details regarding the context injected into every Abaco actor.
- *Actor State*: Working with the State API to store state between actor executions.
- *Actor Sharing and Nonces*: Sharing actors with other users and using nonces to execute actors.
- *API Reference*: Complete HTTP API reference.

CHAPTER 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 complete list of properties 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 values can also be overridden when passing a message to the reactor in the query parameters (see *Messages, Executions, and Logs*).
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.
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.
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 Actors*).
use_container_uid	Run the actor using the UID/GID set in the Docker image. *Note*: Setting this parameter to True requires elevated permissions.

4.1 Notes

- 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.

4.2 Examples

4.2.1 curl

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 "Authorization: Bearer $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://api.tacc.utexas.edu/actors/v2
```

4.2.2 Python

To register the same actor using the agavepy library, we use the `actors.add()` method and pass the same arguments through the *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 agavepy.agave import Agave
>>> ag = Agave(api_server='https://api.tacc.utexas.edu', token='<access_token>')
>>> actor = {"image": "abacosamples/test",
             "name": "test",
             "description": "My test actor using the abacosamples image registered␣
↪using agavepy.",
             "default_environment":{"key1": "value1", "key2": "value2"} }
>>> ag.actors.add(body=actor)
```

CHAPTER 5

Abaco Context & Container Runtime

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

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.

5.1.1 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 *Actor State* for more details.
- The "executor" of the actor may be different from the owner; see *Actor Sharing and Nonces* for more details.

5.1.2 Access from Python

The `agavepy.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 `agavepy.Agave` object.
- `update_state(val)` - Atomically, update the actor's state to the value `val`.

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.

5.2.1 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 *Actor Registration*).

5.2.2 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.

CHAPTER 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.

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.

6.1.1 Sending a message

cURL

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

```
$ curl -H "Authorization: Bearer $TOKEN" \
-d "message=<your content here>" \
https://api.tacc.utexas.edu/actors/v2/<actor_id>/messages
```

Python

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

```
ag.actors.sendMessage(actorId='<actor_id>',
                      body={'message':'<your content here>'})
```

Results

These calls result in a JSON list similar to the following:

```
{'message': 'The request was successful',
 'result': {'_links': {'messages': 'https://api.tacc.utexas.edu/actors/v2/
↪R0y3eYbWmgEwo/messages',
   'owner': 'https://api.tacc.utexas.edu/profiles/v2/apitest',
   'self': 'https://api.tacc.utexas.edu/actors/v2/R0y3eYbWmgEwo/executions/
↪00wLaDX53WBAr'},
  'executionId': '00wLaDX53WBAr',
  'msg': '<your content here>'},
 'status': 'success',
 'version': '0.11.0'}
```

6.1.2 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 "Authorization: Bearer $TOKEN" \
https://api.tacc.utexas.edu/actors/v2/<actor_id>/messages
```

Python

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

```
ag.actors.getMessages(actorId='<actor_id>')
```

Results

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

```
{'message': 'The request was successful',
 'result': {'_links': {'owner': 'https://api.tacc.utexas.edu/profiles/v2/cgarcia',
   'self': 'https://api.tacc.utexas.edu/actors/v2/R4OR3KzGbRQmW/messages'},
  'messages': 12},
 'status': 'success',
 'version': '0.11.0'}
```

6.1.3 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 `AgavePy` in order to keep code in one script.

Python with AgavePy

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

```
from agavepy.agave import Agave
ag = Agave(api_server='https://api.tacc.utexas.edu',
           username='<username>', password='<password>',
           client_name='JPEG_classifier',
           api_key='<api_key>',
           api_secret='<api_secret>')

ag.get_access_token()
ag = Agave(api_server='https://api.tacc.utexas.edu/', token=ag.token)
```

Creating actor with the TensorFlow image classifier docker image:

```
my_actor = {'image': 'notchristiangarcia/bin_classifier',
            'name': 'JPEG_classifier',
            'description': 'Labels a read in binary image'}
actor_data = ag.actors.add(body=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 `application/octet-stream` header:

```
result = ag.actors.sendMessage(actorId=actor_data['id'],
                               body={'binary': binary_image},
                               headers={'Content-Type': 'application/octet-stream'})
```

The following returns information pertaining to the execution:

```
execution = ag.actors.getExecution(actorId=actor_data['id'],
                                   executionId = result['executionId'])
```

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

```
exec_info = requests.get('{}/actors/v2/{}/executions/{}'.format(url, actor_id, exec_
↪id),
                         headers={'Authorization': 'Bearer {}'.format(token)})
```

6.1.4 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 `AgavePy` 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 `/actor/<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 "Authorization: Bearer $TOKEN" \
-d "message=<your content here>" \
https://api.tacc.utexas.edu/actors/v2/<actor_id>/executions/<execution_id>/results
```

6.1.5 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 "Authorization: Bearer $TOKEN" \
-d "message=<your content here>" \
https://api.tacc.utexas.edu/actors/v2/<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).

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.

6.2.1 Access execution data

cURL

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

```
$ curl -H "Authorization: Bearer $TOKEN" \
https://api.tacc.utexas.edu/actors/v2/<actor_id>/executions/<execution_id>
```

Python

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

```
ag.actors.getExecution(actorId='<actor_id>',
                       executionId='<execution_id>')
```

Results

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

```
{'message': 'Actor execution retrieved successfully.',
 'result': {'_links': {'logs': 'https://api.tacc.utexas.edu/actors/v2/R0y3eYbWmgEwo/
↪executions/00wLaDX53WBAr/logs',
   'owner': 'https://api.tacc.utexas.edu/profiles/v2/apitest',
   'self': 'https://api.tacc.utexas.edu/actors/v2/R0y3eYbWmgEwo/executions/
↪00wLaDX53WBAr'},
  'actorId': 'R0y3eYbWmgEwo',
  'apiServer': 'https://api.tacc.utexas.edu',
```

(continues on next page)

(continued from previous page)


```
  'cpu': 7638363913,
  'executor': 'apitest',
  'exitCode': 1,
  'finalState': {'Dead': False,
   'Error': '',
   'ExitCode': 1,
   'FinishedAt': '2019-02-21T17:32:18.56680737Z',
   'OOMKilled': False,
   'Paused': False,
   'Pid': 0,
   'Restarting': False,
   'Running': False,
   'StartedAt': '2019-02-21T17:32:14.893485694Z',
   'Status': 'exited'},
  'id': '00wLaDX53WBAr',
  'io': 124776656,
  'messageReceivedTime': '2019-02-21 17:31:24.300900',
  'runtime': 11,
  'startTime': '2019-02-21 17:32:12.798836',
  'status': 'COMPLETE',
  'workerId': 'oQpeybmGRVNyB'},
 'status': 'success',
 'version': '0.11.0'}
```


6.2.2 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 "Authorization: Bearer $TOKEN" \
https://api.tacc.utexas.edu/actors/v2/<actor_id>/executions
```

Python

To list executions with `AgavePy` the following is done:

```
ag.actors.listExecutions(actorId='<actor_id>')
```

Results

Calling the list of executions should result in something similar to:


```
{'message': 'Actor execution retrieved successfully.',
 'result': {'_links': {'logs': 'https://api.tacc.utexas.edu/actors/v2/R4OR3KzGbRQmW/
↪executions/YqM3RPRoWqz3g/logs',
   'owner': 'https://api.tacc.utexas.edu/profiles/v2/apitest',
   'self': 'https://api.tacc.utexas.edu/actors/v2/R4OR3KzGbRQmW/executions/
↪YqM3RPRoWqz3g'},
```


(continues on next page)

(continued from previous page)

```
  'actorId': 'R4OR3KzGbRQmW',
  'apiServer': 'https://api.tacc.utexas.edu',
  'cpu': 0,
  'executor': 'apitest',
  'id': 'YqM3RPRoWqz3g',
  'io': 0,
  'messageReceivedTime': '2019-02-22 01:01:50.546993',
  'runtime': 0,
  'startTime': 'None',
  'status': 'SUBMITTED'},
 'status': 'success',
 'version': '0.11.0'}
```

6.2.3 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 `AgavePy`, you would get the message context from within the actor and then get it's `raw_message` field.

```
from agavepy.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 `AgavePy` 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 agavepy.actors import get_binary_message

bin_message = get_binary_message()
```

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.

6.3.1 Retrieving an executions logs

cURL

To call the `log` endpoint using cURL, do the following:

```
$ curl -H "Authorization: Bearer $TOKEN" \
https://api.tacc.utexas.edu/actors/v2/<actor_id>/executions/<execution_id>/logs
```

Python

To call the `log` endpoint using `AgavePy` and Python, do the following:

```
ag.actors.getExecutionLogs(actorId='<actor_id>',
                           executionId='<executionId>')
```

Results

This would result in data similar to the following:

```
{'message': 'Logs retrieved successfully.',
 'result': {'_links': {'execution': 'https://api.tacc.utexas.edu/actors/v2/
↪qgKRpNKxg0DME/executions/qgmq08wKARlg3',
   'owner': 'https://api.tacc.utexas.edu/profiles/v2/apitest',
   'self': 'https://api.tacc.utexas.edu/actors/v2/qgKRpNKxg0DME/executions/
↪qgmq08wKARlg3/logs'},
  'logs': '<command line output here>'},
 'status': 'success',
 'version': '0.11.0'}
```

CHAPTER 7

Actor State

In this section we describe the state that can persist through Abaco actor container executions.

7.1 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 `/actors/v2/{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.

7.1.1 Notes

- 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.

7.2 Utilizing State in Actors to Accomplish Something

WIP

7.3 Examples

7.3.1 curl

Here are some examples interacting with state using curl.

Registering an actor specifying statefulness: `stateless=false`.

```
$curl -H "$header" \
-X POST \
-d "image=abacosamples/test&stateless=false" \
https://api.tacc.utexas.edu/actors/v2
```

POSTing a state to a particular actor; keep in mind we must indicate in the header that we are passing content type `application/json`.

```
$curl -H "$header" \
-H "Content-Type: application/json" \
-d '{"some variable": "value", "another variable": "value2"}' \
https://api.tacc.utexas.edu/actors/v2/<actor_id>/state
```

GETting information about a particular actor's state.

```
$curl -H "$header" \
https://api.tacc.utexas.edu/actors/v2/<actor_id>/state
```

7.3.2 Python

Here are some examples interacting with state using Python. The `agavepy.actors` module provides access to an actor's environment data in native Python objects.

Registering an actor specifying statefulness: `stateless=false`.

```
>>> from agavepy.agave import Agave
>>> ag = Agave(api_server='https://api.tacc.utexas.edu', token='<access_token>')
>>> actor = {"image": "abacosamples/test",
    "stateless": "False"}
>>> ag.actors.add(body=actor)
```

POSTing a state to a particular actor; again keep in mind we must pass in JSON serializable data.

```
>>> from agavepy.actors import update_state
>>> state = {"some variable": "value", "another variable": "value2"}
>>> update_state(state)
```

GETting information about a particular actor's state. This function returns a Python dictionary with many fields one of which is state.

```
>>> from agavepy.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': {}}
```

7.4 Additional Work

- Create a pipeline between worker and actor to exchange state without HTTP latency. (Not worker->server->actor->server)
- Develop 'stateful' actors that can execute in parallel (utilizing CRDT data-types)

CHAPTER 8

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.

8.1 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.

8.1.1 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 "Authorization: Bearer $TOKEN" \
-d "user=jdoe&level=READ" \
https://api.tacc.utexas.edu/actors/v2/<actor_id>/permissions
```

Example response:

```
{
  "message": "Permission added successfully.",
  "result": {
```

(continues on next page)

(continued from previous page)

```
    "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 "Authorization: Bearer $TOKEN" \
https://api.tacc.utexas.edu/actors/v2/<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*

8.2 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.

8.2.1 cURL

The following grants *READ* permission to all API users:

```
$ curl -H "Authorization: Bearer $TOKEN" \
-d "user=ABACO_WORLD&level=READ" \
https://api.tacc.utexas.edu/actors/v2/<actor_id>/permissions
```

8.3 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.

8.3.1 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 "Authorization: Bearer $TOKEN" \
-d "maxUses=5&level=READ" \
https://api.tacc.utexas.edu/actors/v2/<actor_id>/nonces
```

A typical response:


```
{
  "message": "Actor nonce created successfully.",
  "result": {
    "_links": {
      "actor": "https://api.tacc.utexas.edu/actors/v2/rNjQG5BBJoxO1",
      "owner": "https://api.tacc.utexas.edu/profiles/v2/testuser",
      "self": "https://api.tacc.utexas.edu/actors/v2/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,
    "roles": [
      "Internal/everyone",
      "Internal/AGAVEDEV_testuser_postman-test-client-1497902074_PRODUCTION",
      "Internal/AGAVEDEV_testuser_postman-test-client-1494517466_PRODUCTION",
   ]
  },
  "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*.

8.3.2 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://api.tacc.utexas.edu/actors/v2/<actor_id>/messages?x-nonce=TACC_vr9rMO6Zy0yHz
```

The response will be exactly the same as if issuing the request with an access token.

CHAPTER 9

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.

9.1 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.

9.1.1 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 "Authorization: Bearer $TOKEN" \
-d "alias=counter&actorId=6PlMbDLa4zlON" \
https://api.tacc.utexas.edu/actors/v2/aliases
```

Example response:

```
{
  "message": "Actor alias created successfully.",
  "result": {
    "_links": {
      "actor": "https://api.tacc.utexas.edu/actors/v2/6PlMbDLa4zlON",
      "owner": "https://api.tacc.utexas.edu/profiles/v2/jstubbs",
      "self": "https://api.tacc.utexas.edu/actors/v2/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 "Authorization: Bearer $TOKEN" \
https://api.tacc.utexas.edu/actors/v2/counter
```

The response returned is identical to that returned when the actor id is used.

9.1.2 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 "Authorization: Bearer $TOKEN" \
-d "maxUses=5&level=READ" \
https://api.tacc.utexas.edu/actors/v2/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.

9.2 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.

9.2.1 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.

9.2.2 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 *Actor 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 "Authorization: Bearer $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://api.tacc.utexas.edu/actors/v2
```

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.

9.2.3 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 "Authorization: Bearer $TOKEN" \
-H "Content-Type: application/json" \
-d '{"image": "abacosamples/test", "name": "test", "webhook": "https://eniih104j4tan.
↪x.pipedream.net", }' \
https://api.tacc.utexas.edu/actors/v2
```

9.2.4 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 *Abaco Context & Container Runtime*, 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.

CHAPTER 10

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.

10.1 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 "Authorization: Bearer $TOKEN" \
-H "Content-type: application/json" \
-d '{"image": "abacosamples/wc", "hints": ["sync"]}' \
https://api.tacc.utexas.edu/actors/v2
```

CHAPTER 11

API Reference

The following table lists the public endpoints within the Abaco API.

GET	POST	PUT	DELETE	Endpoint	Description
X				/actors/v2/utilization	Get high-level usage stats.
X	X			/actors/v2	List/create actors.
X	X			/actors/v2/aliases	List/create aliases.
X			X	/actors/v2/aliases/{alias}	List/delete an alias.
X		X	X	/actors/v2/{actor_id}	List/update/delete an actor.
X	X			/actors/v2/{actor_id}/messages	Get number messages/send message
X	X			/actors/v2/{actor_id}/nonces	List/create actor nonces.
X			X	/actors/v2/{actor_id}/nonces/{nonce_id}	Get nonce details/delete nonce.
X	X			/actors/v2/{actor_id}/state	Retrieve/update actor state.
X	X			/actors/v2/{actor_id}/workers	List/create actor workers.
X			X	/actors/v2/{actor_id}/workers/{worker_id}	Get worker details/delete worker
X	X			/actors/v2/{actor_id}/permissions	List/update actor permissions.
X				/actors/v2/{actor_id}/executions	Retrieve execution details.
			X	/actors/v2/{actor_id}/executions/{eid}	Halt running execution.
X				/actors/v2/{actor_id}/executions/{eid}/logs	Retrieve execution logs.
X				/actors/v2/{actor_id}/executions/{eid}/results	Retrieve execution results.
X				/metrics	

CHAPTER 12

Abaco Samples

In order to simplify the creation of Abaco actors, the Abaco team is developing a suite of Docker images that provide code examples and convenience utilities. This growing catalogue of public example images is available on the public Docker Hub within the abacosamples Docker organization.

CHAPTER 13

Reactor Recipes

Coming soon... some effective patterns for event-driven programming with Abaco.

CHAPTER 14

Overview

In this section we cover additional tools and resources for working with the Abaco Platform.

CHAPTER 15

Abaco CLI

The Abaco CLI is a command line toolkit for developing, managing and using Abaco Actors. The CLI can be installed directly from its github repository, https://github.com/TACC-Cloud/abaco-cli. Please follow the instructions found on the project's README.

CHAPTER 16

Using Abaco from the TACC Cloud JupyterHub

Coming soon... executing functions in parallel on Abaco from the TACC Cloud Jupyter Hub.