TapisActorsDocumentation

Release 0.1

Texas Advanced Computing Center

Sep 22, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION TO TAPIS ACTORS

1.1 What is Tapis?

Tapis is TACC's Application Programming Interface

- Science-as-a-Service platform
- Web services that provide access to TACC resources
- Can provide access to other resources
- Supports common file and compute job operations
- Used by TACC Portals

More information about Tapis and its current and future capabilities is available at: The Tapis Project

1.2 What is Tapis Actors?

Tapis Actors is a form of *serverless computing* like AWS Lambda, Firebase Functions, or OpenFaas, but are tailored to support the needs of research computing.

1.3 What is the use case?

I need X to happen when Y occurs.

- I need to *load a JSON file into a database* when *it is uploaded to my $WORK*
- I need to *send an email to my supervisor* when *this analysis detects something interesting*
- I need to *launch the next stage in workflow* when *the current stage completes successfully*
- I need to *generate and email a report* when *the time is 1:00 PM each day*
- I need to *compute a value N* when *given an input P*

The Tapis Actors API lets you deploy functions or services that accomplish these (and many other) use cases.

Tapis Actors also lets you:

- **Avoid lockin**
  - Built as Docker containers
  - Underlying platform is free and open-source

- **Deploy any code**
  - Write new function code in any language
  - Bring in legacy software and binaries
- **Run at any scale**
  - Automatically scale up when needed
  - Scale to zero when not

Tapis Actors can run standalone, be composed into complex workflows, or integrated with external third-party platforms. They serve as connecting threads amongst the complex systems in our commercial and research computing ecosystems.

1.4 What is an Actor?

For our purposes, an *Actor* is a container-based function-as-a-service deployed on a software platform called Abaco, where it will follow the *actor model* of concurrent computation.

In response to a message it receives, an Actor can

- Make local decisions
- Create more actors
- Send more messages
- Determine how to respond to the next message received.

Actors may modify their own private state, but can only affect each other indirectly through messaging. The *actor model* is characterized by concurrency of computation within and among actors, dynamic actor creation, requirement for actor addresses in messages, and interaction only through direct message passing.

In the Tapis Actors implementation, each actor registered in the system is associated with a Docker image. Actor containers are executed in response to messages posted to their inbox, which itself is given by a URI exposed via the system. In the process of executing each actor container state, logs and execution statistics are collected.

Typically, functions performed by actors are quick and require little processing power. Use cases with more substantial run times or resource requirements are usually best addressed using Tapis Apps.

1.5 How Does an Actor Work?

The function an actor performs is specified as the **default command** in a Docker container. Code for this function is written based around a handful of core assumptions:

1. The *message* is passed via an environment variable `MSG`
2. Supplemental environment variables can be specified when the actor is deployed
3. Parameters can be provided alongside the message, which are passed as additional environment variables
4. The execution environment is read-only and unpriveleged
5. Inbound network connections are disallowed
6. Outbound network connections are unrestricted
7. The execution environment is destroyed when the function has completed
8. `STDERR` and `STDOUT` are captured by the Abaco platform for later review

Depending on the configuration of your specific Tapis Actors *tenant*, the following additional assumptions may apply. For the `tacc.prod` tenant running at TACC, they absolutely do.

1. A Tapis access token is available via an environment variable
2. The TACC `$WORK` filesystem is mounted and writeable at `/work`
3. Code will run as your TACC-default user and group ID

1.5.1 Workflow

The workflow for bworking with Actors will be covered in detail in this tutorial, but briefly, is as follows:

- Write code and package into a Docker container
- Push the container to a public container registry (DockerHub)
- Register an actor to use the container
- Send a message to the actor
- Verify execution by inspecting the logs
- (Optional) Update container or actor
- (Optional) Share the actor with other users
- (Optional) Delete the actor

1.6 Learn More

For a full reference guide to actors, see the Tapis Actors online documentation.

CHAPTER

TWO

SET UP YOUR ENVIRONMENT

Building, deploying, managing, and using Tapis Actors is comprehensively supported via the Tapis CLI and a local installation of Docker. In this section, we will cover preparation of your working environment.

2.1 Prerequisites

You will need the following to work with Tapis Actors:

1. A TACC account
2. A DockerHub account
3. A terminal emulator and/or SSH client

2.2 Local Development Environment

Many people develop and use Actors right on their local laptop computer using the Tapis CLI and Docker Desktop

1. Install Docker: Mac | Ubuntu | Windows
2. Ensure Docker is installed and active: `docker images list`
3. Log into DockerHub: `docker login`
4. Install the Tapis CLI: `pip3 install tapis-cli` (**Tapis CLI is Python3-only**)
5. Check that the CLI is available: `tapis -h`

Note: You can also install the latest Tapis CLI from source but we recommend using the version available on PyPi.

2.2.1 Configure the Tapis CLI

Tapis CLI must be configured to talk to the Tapis APIs and DockerHub. The `auth init` command is an interactive workflow that guides you through that process.

```
$ tapis auth init --interactive

 Configure Tapis API access:
 ===========================
 +---------------+---------------------------------------+-------------------------------
↪--------+
```

(continues on next page)

(continued from previous page)

```
|      Name     |              Description              |                   URL          ␣
↪        |
+---------------+---------------------------------------+--------------------------------
↪--------+
|      3dem     |              3dem Tenant              |         https://api.3dem.org/ ␣
↪        |
|     a2cps     |   Acute to Chronic Pain Signatures    |         https://api.a2cps.org/␣
↪        |
|     bridge    |                Bridge                 |     https://api.bridge.tacc.
↪cloud/     |
|   designsafe  |              DesignSafe               |    https://agave.designsafe-ci.
↪org/    |
|  iplantc.org  |         CyVerse Science APIs          |       https://agave.iplantc.
↪org/       |
|      irec     |              iReceptor                | https://irec.tenants.prod.tacc.
↪cloud/  |
|    portals    |            Portals Tenant             |  https://portals-api.tacc.
↪utexas.edu/  |
|      sd2e     |              SD2E Tenant              |         https://api.sd2e.org/ ␣
↪        |
|      sgci     | Science Gateways Community Institute |        https://sgci.tacc.cloud/
↪        |
|   tacc.prod   |                TACC                   |      https://api.tacc.utexas.
↪edu/      |
| vdjserver.org |              VDJ Server               | https://vdj-agave-api.tacc.
↪utexas.edu/ |
+---------------+---------------------------------------+--------------------------------
↪--------+
Enter a tenant name [tacc.prod]: tacc.prod
Verify SSL connections [Y/n]: Y
tacc.prod username: vaughn
tacc.prod password for vaughn: <password>

Container registry access:
--------------------------
Registry Url [https://index.docker.io]:
Registry Username [mwvaughn]: mwvaughn
Registry Password []: <password>
Registry Namespace [sd2e]: mwvaughn
+--------------------+----------------------------------+
| Field              | Value                            |
+--------------------+----------------------------------+
| tenant_id          | tacc.prod                        |
| username           | vaughn                           |
| api_key            | k_Mt8PGe_e1T4fSFpnvfSkVOyIQa     |
| access_token       | cbbb521f1f4df4ad278d6dbf30168812 |
| expires_at         | Wed Aug 25 13:21:00 2021         |
| verify             | True                             |
| registry_url       | https://index.docker.io          |
| registry_username  | mwvaughn                         |
| registry_password  | p******d                        |
| registry_namespace | mwvaughn                         |
```

(continues on next page)

(continued from previous page)

```
 +--------------------+----------------------------------+
```

Now, confirm that Tapis CLI is properly configured by retrieving your user data from the `tapis profiles` service.

```
$ tapis profiles show me
+--------------+------------------------+
| Field        | Value                  |
+--------------+------------------------+
| first_name   | Matthew                |
| last_name    | Vaughn                 |
| email        | vaughn@tacc.utexas.edu |
| mobile_phone |                        |
| phone        |                        |
| username     | vaughn                 |
+--------------+------------------------+
```

2.3 Using a VM

If you find that your local system does not support the Tapis CLI or Docker, it is possible to use a virtual machine. Please do feel free to reach out to us at TACC for assistance.

CHAPTER

THREE

BUILD HELLO WORLD ACTOR

Let us build our hello-world actor!

3.1 Create a New Actor

The function of an actor is exposed as the default command in a Docker container. Here, we will create an actor from an existing Docker container image called **tacc/hello-world:1.0** available on Docker Hub. The default command for this container simply prints the message "Hello, World" or the message sent to it, which will be captured in the actor logs.

Create the actor as:

```
$ tapis actors create --repo tacc/hello-world:1.0 \
                      -n hello-world-actor \
                      -d "Test actor that says Hello, World"
+----------------+-----------------------------+
| Field          | Value                       |
+----------------+-----------------------------+
| id             | NN5N0kGDvZQpA               |
| name           | hello-world-actor           |
| owner          | taccuser                    |
| image          | tacc/hello-world:1.0        |
| lastUpdateTime | 2021-07-14T22:25:06.171534  |
| status         | SUBMITTED                   |
| cronOn         | False                       |
+----------------+-----------------------------+
```

The `--repo` flag points to the Docker Hub repo on which this actor is based, the `-n` flag and `-d` flag attach a human-readable name and description to the actor.

The resulting actor is assigned an id: `NN5N0kGDvZQpA`. The actor id can be queried by:

```
$ tapis actors show NN5N0kGDvZQpA
+-----------------+-----------------------------------+
| Field           | Value                             |
+-----------------+-----------------------------------+
| id              | NN5N0kGDvZQpA                     |
| name            | hello-world-actor                 |
| description     | Test actor that says Hello, World |
| owner           | taccuser                          |
| image           | tacc/hello-world:1.0              |
```

(continues on next page)

(continued from previous page)

```
| createTime      | 2021-09-21T20:05:10.738Z          |
| lastUpdateTime  | 2021-09-21T20:05:10.738Z          |
| gid             | 859336                            |
| link            |                                   |
| privileged      | False                             |
| queue           | default                           |
| stateless       | True                              |
| status          | READY                             |
| statusMessage   |                                   |
| token           | True                              |
| uid             | 859336                            |
| useContainerUid | False                             |
| webhook         |                                   |
| cronOn          | False                             |
| cronSchedule    | None                              |
+-----------------+-----------------------------------+
```

Above, you can see the plain text name, description that were passed on the command line. In addition, you can see the "status" of the actor is "READY", meaning it is ready to receive and act on messages. Finally, you can list all actors visible to you with:

```
$ tapis actors list
+---------------+------------------+----------+------------------------------+-----------
↪-----------------+--------+-------+
| id            | name             | owner    | image                        |␣
↪lastUpdateTime             | status | cronOn|
+---------------+------------------+----------+------------------------------+-----------
↪-----------------+--------+-------+
| NN5N0kGDvZQpA | hello-word-actor  | taccuser | tacc/hello-world:1.0         | 2021-07-
↪14T22:25:06.171Z   | READY  | False |
+---------------+------------------+----------+------------------------------+-----------
↪-----------------+--------+-------+
```

3.2 Submit a Message to the Actor

Next, let's craft a simple message to send to the reactor. Messages can be plain text or in JSON format. When using the python actor libraries as in the example above, JSON-formatted messages are made available as python dictionaries.

```
# Submit the message to the actor
$ tapis actors submit -m "Hello, World" NN5N0kGDvZQpA
+-------------+---------------+
|  Field      | Value         |
+-------------+---------------+
| executionId | N4xQ5WM5Np1X0 |
| msg         | Hello, World  |
+-------------+---------------+
```

The id of the actor (`N4xQ5WM5Np1X0`) was used on the command line to specify which actor should receive the message. In response, an "execution id" (`N4xQ5WM5Np1X0`) is returned. An execution is a specific instance of an actor. List all the executions for a given actor as:

```
$ tapis actors execs list NN5N0kGDvZQpA
+---------------+----------+
| executionId   | status   |
+---------------+----------+
| N4xQ5WM5Np1X0 | COMPLETE |
+---------------+----------+
```

Show detailed information for the execution with:

```
$ tapis actors execs show -v NN5N0kGDvZQpA N4xQ5WM5Np1X0
 {
     "actorId": "NN5N0kGDvZQpA",
     "apiServer": "https://api.tacc.utexas.edu",
     "cpu": 121748743,
     "exitCode": 0,
     "finalState": {
         "Dead": false,
         "Error": "",
         "ExitCode": 0,
         "FinishedAt": "2021-07-14T22:32:45.602Z",
         "OOMKilled": false,
         "Paused": false,
         "Pid": 0,
         "Restarting": false,
         "Running": false,
         "StartedAt": "2021-07-14T22:32:45.223Z",
         "Status": "exited"
     },
     "id": "N4xQ5WM5Np1X0",
     "io": 176,
     "messageReceivedTime": "2021-07-14T22:32:37.051Z",
     "runtime": 1,
     "startTime": "2021-07-14T22:32:44.752Z",
     "status": "COMPLETE",
     "workerId": "JABKl4BeDwXJD",
     "_links": {
         "logs": "https://api.tacc.utexas.edu/actors/v2/NN5N0kGDvZQpA/executions/
↪N4xQ5WM5Np1X0/logs",
         "owner": "https://api.tacc.utexas.edu/profiles/v2/sgopal",
         "self": "https://api.tacc.utexas.edu/actors/v2/NN5N0kGDvZQpA/executions/
↪N4xQ5WM5Np1X0"
     }
 }
```

We can see here that the above execution has already completed.

3.3 Check the Logs for an Execution

An execution's logs will contain whatever was printed to STDOUT / STDERR by the actor. In our demo actor, we just expect the actor to print the message passed to it.

```
$ tapis actors execs logs NN5N0kGDvZQpA N4xQ5WM5Np1X0
Logs for execution N4xQ5WM5Np1X0
 Actor received message: Hello, World
```

In a normal scenario, the actor would then act on the contents of a message to, e.g., kick off a job, perform some data management, send messages to other actors, or more.

3.4 Run Synchronously

The previous message submission (with `tapis actors submit`) was an *asynchronous* run, meaning the command prompt detached from the process after it was submitted to the actor. In that case, it was up to us to check the execution to see if it had completed and manually print the logs.

There is also a mode to run actors *synchronously* using `tapis actors run`, meaning the command line stays attached to the process awaiting a response after sending a message to the actor.

3.5 Delete and Update an Actor

Actors can be deleted with the following:

```
$ tapis actors delete NN5N0kGDvZQpA
+----------+--------------------+
| Field    | Value              |
+----------+--------------------+
| deleted  | ['NN5N0kGDvZQpA'] |
| messages | []                 |
+----------+--------------------+
```

This will delete the actor and any associated executions. Actors can also be updated with the `tapis actors update` command to make changes once created.

Need help? Ask your questions using the [TACCSTER 2021 Slack Workspace] using the #tutorial-automating-work-at-tacc-with-tapis-actors channel.

CHAPTER

FOUR

SLACKBOT ACTOR

In this section, we will create an actor that sends a message to a Slack Channel using a webhook provided by Slack. A pre-requisite requirement is an active webhook for your Slack workspace.

4.1 Writing Customized Actors

Recall that Tapis Actors are essentially bits of code wrapped in lightweight environments called containers. When we deployed the `hello-world` Actor in the previous section, we used a pre-built Docker container (`tacc/hello-world:1.0`). However, a more common use case is to write custom code that runs whenever the Actor executes. To do this, we must first build a Docker image using our custom `actor.py`, push the Docker image to a public image registry (DockerHub), and instruct Tapis to use our custom image when executing the container.

If you are interested in learning more about Docker containers, please attend our deep dive Docker container session tomorrow (Friday Sept. 24th), where we will cover custom image creation, common use cases, and more. The documentation for this session can be found here .

4.2 Initialize a New Actor `slackbot-actor`

```
$ mkdir slackbot_actor
$ cd slackbot_actor
$ touch Dockerfile actor.py
$ find -L .
.
./Dockerfile
./actor.py
```

4.3 Edit the Actor Source Code in `actor.py`

An example of a functional actor that says sends a slack message is:

```
# actor.py
"""Forward message from Actor inbox to Slack"""

from agavepy.actors import get_context
import requests
import os
```

(continues on next page)

(continued from previous page)

```
import simplejson as json


def post_to_slack(message: str):
    """Forward string type `message` to Slack channel"""

    webhook_url = os.environ.get('SLACK_WEBHOOK')
    print("Actor sending message to Slack: {0}".format(message))

    response_from_slack = requests.post(
        webhook_url, data=json.dumps({'text': message}),
        headers={"Content-type": "application/json"})
    print("Response from Slack:")
    print(response_from_slack)


def main():
    """Main entrypoint"""
    context = get_context()
    message = context['raw_message']
    post_to_slack(message)


if __name__ == '__main__':
    main()
```

Here inject the necessary environment variable `SLACK_WEBHOOK` set from `tapis actors create` command. How do we get the webhook into our Actor? We don't. Instead of embedding it in the underlying container, it is defined as an environment variable in the Actor. We must set `SLACK_WEBHOOK` while creating the actor. This can be done using Tapis CLI. In the `tapis actors create` command, we can pass it as an environment variables via the -e flag.

4.4 Write the Dockerfile

The first step to building our customized Actor is to build a customized Docker image. To do this, we write a Dockerfile, which is a list of instructions for building our customized environment. The below Dockerfile constructs an image containing a Python3 runtime environment, Python package dependencies installed by pip, and our `actor.py` script. The pip-installed requirements are agavepy, requests simplejson python libraries, which are available through PyPi.

```
# pull base image, which provides a Python3 runtime
FROM python:3.6

# install package dependencies using pip
RUN pip3 install agavepy simplejson requests

# add our custom Python script
ADD actor.py /actor.py

# command to run the python script
CMD ["python", "/actor.py"]
```

4.5 Build Docker Container

We can now use our Dockerfile to build a custom Docker image:

Note: In the below command, make sure to replace `taccuser` with your DockerHub username.

```
# Build and tag the image
$ docker build -t taccuser/slackbot-actor:1.1 .
Sending build context to Docker daemon  4.096kB
Step 1/5 : FROM python:3.6
...
Successfully built b0a76425e8b3
Successfully tagged taccuser/slackbot-actor:1.1

# Push the tagged image to Docker Hub
$ docker push taccuser/slackbot-actor:1.1
The push refers to repository [docker.io/taccuser/slackbot-actor]
...
1.1: digest: sha256:67cc6f6f00589d9ae83b99d779e4893a25e103d07e4f660c14d9a0ee06a9ddaf␣
↪size: 1995
```

4.6 Create the Actor

We pass the `SLACK_WEBHOOK` as an environment variable during the time of actor creation.

```
$ tapis actors create --repo taccuser/slackbot-actor:1.1 \
                      -n slackbot-actor \
                      -d "Send a message containing text to Slack channel" \
                      -e SLACK_WEBHOOK="https://hooks.slack.com/services/$
↪{XXXsecretXtokenXXX}"


+----------------+----------------------------+
| Field          | Value                      |
+----------------+----------------------------+
| id             | ww15Ex5oLxJ6b              |
| name           | slackbot-actor             |
| owner          | taccuser                   |
| image          | taccuser/slackbot-actor:1.1|
| lastUpdateTime | 2021-08-24T14:31:58.248860 |
| status         | SUBMITTED                  |
+----------------+----------------------------+
```

```
$ tapis actors show ww15Ex5oLxJ6b
+-----------------+-------------------------------------------------+
| Field           | Value                                           |
+-----------------+-------------------------------------------------+
| id              | ww15Ex5oLxJ6b                                   |
| name            | slackbot-actor                                  |
```

(continues on next page)

(continued from previous page)

```
| description     | Send a message containing text to Slack channel |
| owner           | taccuser                                        |
| image           | taccuser/slackbot-actor:1.1                     |
| createTime      | 2021-09-21T20:27:05.613Z                        |
| lastUpdateTime  | 2021-09-21T20:27:05.613Z                        |
| gid             | 859336                                          |
| link            |                                                 |
| privileged      | False                                           |
| queue           | default                                         |
| stateless       | True                                            |
| status          | READY                                           |
| statusMessage   |                                                 |
| token           | True                                            |
| uid             | 859336                                          |
| useContainerUid | False                                           |
| webhook         |                                                 |
| cronOn          | False                                           |
| cronSchedule    | None                                            |
+-----------------+-------------------------------------------------+
```

we can see the "status" of the actor is "READY", meaning it is ready to receive and act on messages.

Finally, you can list all actors visible to you with:

```
$ tapis actors list

+--------------+--------------+----------+---------------------------+--------------
↪-------------+--------+
| ww15Ex5oLxJ6b | slackbot-actor| taccuser | taccuser/slackbot-actor:1.1 | 2021-08-
↪25T14:04:42.819Z   | READY  |
+--------------+--------------+----------+---------------------------+--------------
↪-------------+--------+
```

4.7 Submit a Message to the Actor

```
# Submit the message to the actor
$ tapis actors submit -m "Hello, Slack!" ww15Ex5oLxJ6b
+-------------+---------------+
|  Field      | Value         |
+-------------+---------------+
| executionId | EjO6yw03GKRmR |
| msg         | Hello, Slack  |
+-------------+---------------+
```

Let us grab the executionId from here to track the progress of the actor.

4.8 List Executions of Actor

The above execution has already completed. Show detailed information for the execution with:

```
$ tapis actors execs show ww15Ex5oLxJ6b EjO6yw03GKRmR
+-----------+-----------------------------+
| Field     | Value                       |
+-----------+-----------------------------+
| actorId   | ww15Ex5oLxJ6b               |
| apiServer | https://api.tacc.utexas.edu |
| id        | EjO6yw03GKRmR               |
| status    | COMPLETE                    |
| workerId  | EbQByMAXeMVPa               |
+-----------+-----------------------------+
```

4.9 Check the Logs for an Execution

In our slackbot-actor, we expect the actor to print the message passed to it and notify on the slack channel.

```
$ tapis actors execs logs ww15Ex5oLxJ6b EjO6yw03GKRmR
Logs for execution EjO6yw03GKRmR
 Actor sending message to Slack: Hello, Slack!
```

Finally check your Slack channel to find your message!

CHAPTER

FIVE

SEND A MESSAGE BETWEEN ACTORS

5.1 Introduction

While standalone Actors are useful, one can also network multiple Actors to generate more complex workflows. Actors have the ability to send messages to other Actors, allowing developers to chain together workflow steps, each contained in a separate Actor.

In this section of the tutorial, we will deploy a simple `upstream-messenger` Actor that sends a message to the `hello-world-actor` that we deployed in the previous section. We will demonstrate that the downstream `hello-world-actor` runs after it receives the message from `upstream-messenger`.

5.2 Create a New Actor Named `upstream-messenger`

First, let's write the code that our new Actor will run on execution. Instead of manually writing these files, we can simply adapt one of the provided Actor templates. To view all available Actor templates, issue:

```
$ tapis actors init -L
+--------------------+--------------------+--------------------------------------------
↪--------------+----------+
| id                 | name               | description                                ␣
↪              | level    |
+--------------------+--------------------+--------------------------------------------
↪--------------+----------+
| default            | Default            | Basic code and configuration skeleton      ␣
↪              | beginner |
| echo               | Echo               | Echo message                               ␣
↪              | beginner |
| hello_world        | Hello World        | Say Hello, World!                          ␣
↪              | beginner |
| sd2e_base          | sd2e_base          | Default reactor context for docker://sd2e/
↪reactors:python3 | beginner |
| tacc_reactors_base | tacc_reactors_base | Default actor context for docker://sd2e/
↪reactors:python3   | beginner |
+--------------------+--------------------+--------------------------------------------
↪--------------+----------+
```

Each template is a project directory for a different type of Actor. For this Actor, let's use the `default` template:

```
$ tapis actors init --template default --actor-name upstream-messenger
+-------+-------------------------------------------------------------------+
| stage | message                                                           |
+-------+-------------------------------------------------------------------+
| setup | Project path: ./upstream_messenger                                |
| setup | CookieCutter variable name=upstream-messenger                     |
| setup | CookieCutter variable project_slug=upstream_messenger             |
| setup | CookieCutter variable docker_namespace=taccuser                       |
| setup | CookieCutter variable docker_registry=https://index.docker.io |
| clone | Project path: ./upstream_messenger                                |
+-------+-------------------------------------------------------------------+
$ cd upstream_messenger
$ find -L .
.
./requirements.txt
./Dockerfile
./project.ini
./message.jsonschema
./default.py
./.gitignore
./secrets.jsonsample
./config.yml
```

We see that the `tapis actors init` command has initialized an Actor project directory for us, and it already contains many files that we could have written by hand such as the `Dockerfile` or the Python source code in `default.py`.

The only file that was not provided by the template is `secrets.json`. Let's make an empty one now:

```
echo '{}' > secrets.json
```

5.3 Edit Actor Source

The Actor we just created doesn't do much; it just says "hello world," like the `hello-world-actor` we deployed previously. Let's change its behavior so it does something more interesting, like message another Actor. We will use the Python API command sendMessage to implement this. Using your favorite text editor, edit the `default.py` script so it looks like:

```
import os
from agavepy.actors import get_context, get_client


def main():
    """Main entrypoint"""
    context = get_context()
    m = context['raw_message']
    print("Actor received message: {}".format(m))

    # Get an active Tapis client
    client = get_client()

    # Pull in the downstream Actor ID from the environment
```

(continues on next page)

(continued from previous page)

```
    downstream_actor_id = context['DOWNSTREAM_ACTOR_ID']
    # alternatively:
    # downstream_actor_id = os.environ['DOWNSTREAM_ACTOR_ID']

    # Using our Tapis client, send a message to the downstream Actor
    message = 'greetings, hello-world-actor!'
    print("Sending message '{}' to {}".format(message, downstream_actor_id))
    response = client.actors.sendMessage(actorId=downstream_actor_id, body={"message":␣
↪message})
    print("Successfully triggered execution '{}' on actor '{}'".format(response[
↪'executionId'], downstream_actor_id))


if __name__ == '__main__':
    main()
```

All we've done is add a block of code that calls the Tapis/Agave API so that it sends a message to another Actor. Notice that we are mimicking the CLI workflow from before:

Action	CLI	Python API
Get an authenticated Tapis client	tapis auth init	client = get_client()
Using the client, send message to an Actor	tapis actors submit	client.actors.sendMessage()
Using the client, submit HPC job to Tapis Application	tapis jobs submit	client.jobs.submit()

In fact, the CLI is making the same calls to the Python API under the hood!

Notice that we haven't actually defined **which** Actor ID we want to send the message to. Per best practice, we've chosen not to "hard code" the Actor ID into `default.py`, but rather read it from the Actor environment, which we access via `context['DOWNSTREAM_ACTOR_ID']` or alternatively `os.environ['DOWNSTREAM_ACTOR_ID']`. To set the `DOWNSTREAM_ACTOR_ID`, we need only define it in the Actor environment when we deploy in the next step. The downstream Actor is the `hello-world-actor` we deployed previously, and we can retrieve its ID using the CLI:

```
$ tapis actors list
+---------------+--------------------+-------+--------------------------------+-----------
↪---------------+--------+--------+
| id            | name               | owner | image                          |␣
↪lastUpdateTime          | status | cronOn |
+---------------+--------------------+-------+--------------------------------+-----------
↪---------------+--------+--------+
```

(continues on next page)

(continued from previous page)

```
| MqqbarbazBB8x | hello-world-actor  | eho   | tacc/hello-world:latest       | 2021-08-
↪24T19:13:44.036Z | READY  | False  |
+---------------+--------------------+-------+-------------------------------+----------
↪---------------+--------+--------+
```

We will need this Actor ID (`MqqbarbazBB8x` in my case, yours will be different) when we deploy in the next section.

5.4 Deploy Actor

Our new `upstream-messenger` Actor is now ready to deploy. Just like before, we want to:

1. Build the Docker image
2. Push the Docker image
3. Register the Docker image as a new Actor

Remember to replace the `DOWNSTREAM_ACTOR_ID` with the appropriate Actor ID from above, and the placeholder `taccuser` with your DockerHub username.

```
$ docker build -t taccuser/upstream-messenger:0.0.1 .
$ docker push taccuser/upstream-messenger:0.0.1
$ tapis actors create --repo taccuser/upstream-messenger:0.0.1 \
                      -n upstream-messenger \
                      -d "Sends message to another actor" \
                      -e DOWNSTREAM_ACTOR_ID=MqqbarbazBB8x
+----------------+-----------------------------------+
| Field          | Value                             |
+----------------+-----------------------------------+
| id             | MDfoobar7AOwx                     |
| name           | upstream-messenger                |
| owner          | taccuser                          |
| image          | taccuser/upstream-messenger:0.0.1 |
| lastUpdateTime | 2021-08-26T20:33:20.320620        |
| status         | SUBMITTED                         |
| cronOn         | False                             |
+----------------+-----------------------------------+
```

If deployment was successful, we should now see our new Actor:

```
$ tapis actors list
+---------------+--------------------+-------+-----------------------------------+-------
↪--------------------+--------+--------+
| id            | name               | owner | image                             |␣
↪lastUpdateTime           | status | cronOn |
+---------------+--------------------+-------+-----------------------------------+-------
↪--------------------+--------+--------+
| MqqbarbazBB8x | hello-world-actor  | eho   | tacc/hello-world:latest           | 2021-
↪08-24T19:13:44.036Z | READY  | False  |
| MDfoobar7AOwx | upstream-messenger | eho   | taccuser/upstream-messenger:0.0.1 | 2021-
↪08-24T20:23:07.619Z | READY  | False  |
+---------------+--------------------+-------+-----------------------------------+-------
↪--------------------+--------+--------+
```

(continues on next page)

(continued from previous page)

```
$ tapis actors show -v MDfoobar7AOwx
{
   "id": "MDfoobar7AOwx",
   "name": "upstream-messenger",
   "description": "Sends message to another actor",
   "owner": "eho",
   "image": "enho/upstream-messenger:0.0.1",
   "createTime": "2021-09-21T20:35:33.39Z0",
   "lastUpdateTime": "2021-09-21T20:35:33.39Z0",
   "defaultEnvironment": {
      "DOWNSTREAM_ACTOR_ID": "MqqbarbazBB8x"
   },
   "gid": 859336,
   "hints": [],
   "link": "",
   "mounts": [],
   "privileged": false,
   "queue": "default",
   "stateless": true,
   "status": "READY",
   "statusMessage": " ",
   "token": true,
   "uid": 859336,
   "useContainerUid": false,
   "webhook": "",
   "cronOn": false,
   "cronSchedule": null,
   "cronNextEx": null,
   "_links": {
      "executions": "https://api.tacc.utexas.edu/actors/v2/MDfoobar7AOwx/executions",
      "owner": "https://api.tacc.utexas.edu/profiles/v2/eho",
      "self": "https://api.tacc.utexas.edu/actors/v2/MDfoobar7AOwx"
   }
}
```

5.4.1 Send Message to `upstream-messenger` Using CLI

Once the `upsteam_messenger` Actor is READY, we can trigger a new execution by sending it a message:

```
$ tapis actors submit -m 'hello, upstream-messenger!' MDfoobar7AOwx
+-------------+----------------------------+
| Field       | Value                      |
+-------------+----------------------------+
| executionId | MDanexec7AOwx              |
| msg         | hello, upstream-messenger! |
+-------------+----------------------------+
```

As usual, we check the status of the execution, and show the logs when it finishes:

```
$ tapis actors execs show MDfoobar7AOwx MDanexec7AOwx
+-----------+----------------------------+
```

(continues on next page)

(continued from previous page)

```
| Field     | Value                        |
+-----------+------------------------------+
| actorId   | MDfoobar7AOwx                |
| apiServer | https://api.tacc.utexas.edu  |
| id        | MDanexec7AOwx                |
| status    | COMPLETE                     |
| workerId  | wZvworker1KmQ                |
+-----------+------------------------------+
$ tapis actors execs logs MDfoobar7AOwx MDanexec7AOwx
Actor received message: hello, upstream-messenger!
Sending message 'greetings, hello-world-actor!' to MqqbarbazBB8x
Successfully triggered execution '5P7foobarrrA6' on actor 'MqqbarbazBB8x'
```

5.4.2 Check Execution of Downstream `hello-world-actor`

The goal of this tutorial was to send a message to `upstream-messenger` and have it trigger an execution on `hello-world-actor`. Let's check the status of the execution and inspect the logs:

```
$ tapis actors execs show MqqbarbazBB8x 5P7foobarrrA6
+-----------+------------------------------+
| Field     | Value                        |
+-----------+------------------------------+
| actorId   | MqqbarbazBB8x                |
| apiServer | https://api.tacc.utexas.edu  |
| id        | 5P7foobarrrA6                |
| status    | COMPLETE                     |
| workerId  | DJPworkerzKlN                |
+-----------+------------------------------+
$ tapis actors execs logs MqqbarbazBB8x 5P7foobarrrA6
Logs for execution 5P7foobarrrA6
 Actor received message: hello, hello-world-actor!
```

5.4.3 Conclusion

Congratulations! We have successfully deployed a workflow that sends a message between two Actors. Of course, real-world multi-Actor workflows will send much more useful information than "hello, world." In practice, messages contain file paths, names of analyses to run, and other metadata. It is also possible for one Actor to send messages to multiple other Actors, allowing for a single action such as a file upload to trigger many downstream processes, such as file management, running analyses, logging, and more.

CHAPTER

SIX

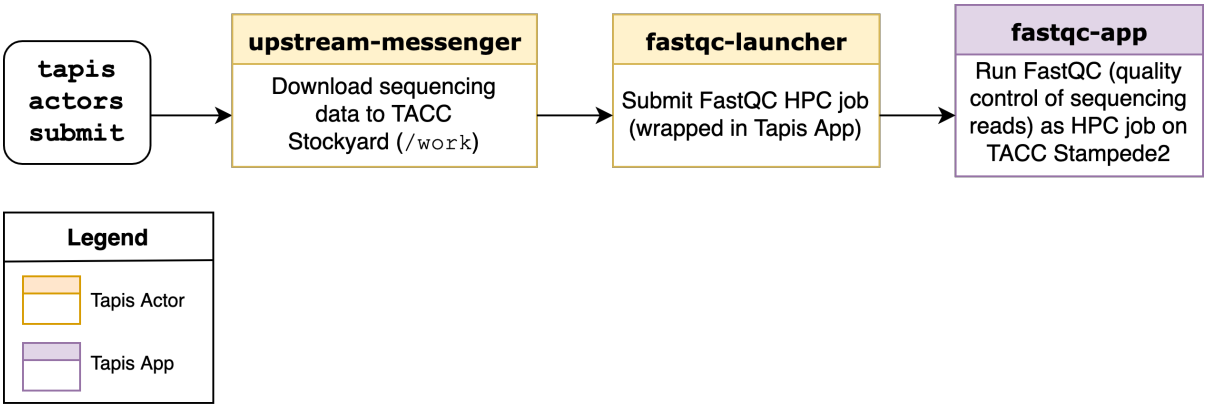
DEPLOYING A SEQUENCING PIPELINE

6.1 Introduction

Thus far, we have demonstrated several useful features of actors:

- Deployment, execution, and handling using Tapis CLI
- Logging to 3rd party services (Slack in our example)
- Inter-Actor messaging

In this section of the tutorial, we will demonstrate this toolbox in a realistic example: a basic pipeline for validating DNA sequencing data.



We will leverage a few additional Actor capabilities beyond what we have covered previously:

- **Actors can interact with TACC filesystems such as `/work` using the Tapis API**
  - We won't cover this in depth today, but if you're interested in Tapis Files, please consult the Tapis documentation
- Actors can submit HPC jobs, by submitting to Tapis Applications

Reusing as much of our previous work as possible, pipeline deployment will consist of the following steps:

- Deploy a Tapis App `eho-fastqc-0.11.9` that runs the FastQC tool as HPC job
- Deploy new actor `fastqc-launcher`
- Change `upstream-messenger` so it uploads a fastq data file from the web to TACC Stockyard filesystem
- Re-deploy `upstream-messenger` so it sends the path to data file to the `fastqc-launcher`

Since deploying Tapis Apps is outside the scope of this tutorial, I have already completed the first step. If you are interested in learning how to deploy Tapis Applications, I encourage you to consult one of our in-depth Tapis tutorials .

6.2 Deploy a New Actor `fastqc-launcher`

Here, we will create an Actor that launches an submits an HPC job to a Tapis App. Recall that Tapis Apps are wrappers around computationally intensive processes. They are similar to Actors in that they can be managed using the Tapis API, but unlike Actors, they are HPC jobs under the hood.

Similar to the `upstream-messenger`, this new Actor (which we will call `fastqc-launcher`) leverages the active Tapis client to interact with the Tapis ecosystem. In this case, we will submit to a Tapis Application instead of messaging another Actor. We instantiate a new Actor as before:

```
tapis actors init --template default --actor-name fastqc-launcher
cd fastqc_launcher
echo '{}' > secrets.json
```

We edit the Actor source code in `default.py` so it resembles:

```
import os
from agavepy.actors import get_context, get_client


def main():
   context = get_context()
   fastq_uri = context['raw_message']
   print("Actor received message: {}".format(fastq_uri))

   # Usually, one would perform some input validation before submitting
   # a job to a Tapis App. Here, we simply validate that the path looks
   # like a Tapis/Agave URI
   assert fastq_uri.startswith('agave://')

   # Get an active Tapis client
   client = get_client()

   # Using our Tapis client, submit a job to Tapis App eho-fastqc-0.11.9
   body = {
      "name": "fastqc-test",
      "appId": "eho-fastqc-0.11.9",
      "archive": False,
      "inputs": {
         "fastq": "agave://eho.work.storage/{}".format(os.path.basename(fastq_uri))
      }
   }
   response = client.jobs.submit(body=body)
   print("Successfully submitted job {} to Tapis App {}".format(response['id'], response[
↪'appId']))


if __name__ == '__main__':
    main()
```

We can deploy this new Actor as usual, by building, pushing, and registering the custom Docker image as a new Actor:

```
$ docker build -t taccuser/fastqc-launcher:0.0.1 .
$ docker push taccuser/fastqc-launcher:0.0.1
$ tapis actors create --repo taccuser/fastqc-launcher:0.0.1 \
                      -n fastqc-launcher \
                      -d "Submits job to FastQC Tapis App"
```

6.3 Edit `upstream-messenger` Source

Using your favorite text editor, edit the `default.py` for `upstream-messenger` so it looks like:

```
import os
from agavepy.actors import get_context, get_client
import requests


def main():
    """Main entrypoint"""
    context = get_context()
    m = context['raw_message']
    print("Actor received message: {}".format(m))

    # Get an active Tapis client
    client = get_client()

    # Pull in the downstream Actor ID from the environment
    downstream_actor_id = context['DOWNSTREAM_ACTOR_ID']
    # alternatively:
    # downstream_actor_id = os.environ['DOWNSTREAM_ACTOR_ID']

    # Using our Tapis client,
    # upload our fastq file to TACC Stockyard using
    url = "https://raw.githubusercontent.com/eho-tacc/fastqc_app/main/tests/data_R1_001.
↪fastq"
    systemId = 'eho.work.storage'
    files_resp = client.files.importData(
        fileName='example.fastq',
        filePath='/',
        systemId=systemId, urlToIngest=url)

    # Using our Tapis client, send the message containing file path
    # to the downstream Actor
    message = "agave://{}/{}".format(systemId, files_resp['path'])
    print("Sending message '{}' to {}".format(message, downstream_actor_id))
    response = client.actors.sendMessage(actorId=downstream_actor_id, body={"message":␣
↪message})
    print("Successfully triggered execution '{}' on actor '{}'".format(response[
↪'executionId'], downstream_actor_id))
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    main()
```

6.4 Re-deploy Actor upstream-messenger

Our Actor upstream-messenger is still configured to send messages to hello-world-actor. We would instead like it to send messages to our new actor fastqc-launcher, so we must update it with a new DOWNSTREAM_ACTOR_ID. Instead of deleting and deploying a new Actor, we can instead:

- Build and push an updated Docker image
- Update the DOWNSTREAM_ACTOR_ID variable using tapis actors update

```
$ docker build -t enho/upstream-messenger:0.0.2 .
$ docker push enho/upstream-messenger:0.0.2
$ tapis actors update --repo taccuser/upstream-messenger:0.0.2 \
                      -e DOWNSTREAM_ACTOR_ID=$FASTQC_LAUNCHER_ID \
                      MDfoobar7AOwx
+----------------+-----------------------------------+
| Field          | Value                             |
+----------------+-----------------------------------+
| id             | MDfoobar7AOwx                     |
| name           | upstream-messenger                |
| owner          | taccuser                          |
| image          | taccuser/upstream-messenger:0.0.2 |
| lastUpdateTime | 2021-08-26T20:33:20.320620        |
| status         | SUBMITTED                         |
| cronOn         | False                             |
+----------------+-----------------------------------+
```

6.5 Test the Pipeline

6.5.1 Send Message to upstream-messenger Using CLI

Once the upsteam_messenger Actor is READY, we can trigger a new execution by sending it a message:

```
$ tapis actors submit -m 'hello, FastQC pipeline!' MDfoobar7AOwx
+-------------+----------------------------+
| Field       | Value                      |
+-------------+----------------------------+
| executionId | MDanexec7AOwx              |
| msg         | hello, FastQC pipeline!    |
+-------------+----------------------------+
```

As usual, we check the status of the execution, and show the logs when it finishes:

```
$ tapis actors execs show MDfoobar7AOwx MDanexec7AOwx
+-----------+----------------------------+
| Field     | Value                      |
```

(continues on next page)

(continued from previous page)

```
+-----------+-----------------------------+
| actorId   | MDfoobar7AOwx               |
| apiServer | https://api.tacc.utexas.edu |
| id        | MDanexec7AOwx               |
| status    | COMPLETE                    |
| workerId  | wZvworker1KmQ               |
+-----------+-----------------------------+
$ tapis actors execs logs MDfoobar7AOwx MDanexec7AOwx
Actor received message: hello, FastQC pipeline!
Sending message 'greetings, hello-world-actor!' to MqqbarbazBB8x
Successfully triggered execution '5P7foobarrrA6' on actor 'MqqbarbazBB8x'
```

6.5.2 Check File Upload

Using the Tapis CLI, we can check that the `upstream-messenger` created the expected file:

```
$ tapis files show agave://eho.work.storage/example.fastq
+--------------+--------------------------------+
| Field        | Value                          |
+--------------+--------------------------------+
| name         | example.fastq                  |
| path         | /work/06634/eho/example.fastq  |
| lastModified | 20 seconds ago                 |
| length       | 64431                          |
| permissions  | READ_WRITE                     |
| mimeType     | application/octet-stream       |
| type         | file                           |
+--------------+--------------------------------+
```

6.5.3 Check Execution of Downstream `fastqc-launcher`

Let's check the status of the execution and inspect the logs:

```
$ tapis actors execs logs MqqbarbazBB8x wKoAJD5NykAKN
Logs for execution wKoAJD5NykAKN
Actor received message: agave://eho.work.storage/example.fastq
Successfully submitted job 6c9d5842-XXXX-XXXX-XXXX-f07b1f73b948-007 to Tapis App eho-
↪fastqc-0.11.9
```

6.5.4 Check Job Submitted to FastQC Tapis App

Finally, let's check that our test job using the FastQC Tapis App successfully finished. We can check the status of this job using the CLI:

```
$ tapis jobs show 6c9d5842-XXXX-XXXX-XXXX-f07b1f73b948-007
+--------------------+------------------------------------------------------------------------
↪--------------+
| Field              | Value                                                                  ␣
↪              |
```

(continues on next page)

(continued from previous page)

```
+--------------------+-------------------------------------------------------------------------
↪--------------+
| accepted           | 2021-09-22T21:39:06.147Z                                                ␣
↪              |
| appId              | eho-fastqc-0.11.9                                                       ␣
↪              |
| appUuid            | 4765625137153839596-XXXXXXXX-0001-005                                   ␣
↪              |
| archive            | False                                                                   ␣
↪              |
| archivePath        | eho/archive/jobs/job-XXXXXXXX-XXXX-XXXX-afbf-f07b1f73b948-007           ␣
↪              |
| archiveSystem      | None                                                                    ␣
↪              |
| blockedCount       | 0                                                                       ␣
↪              |
| created            | 2021-09-22T21:39:06.152Z                                                ␣
↪              |
| ended              | 4 minutes ago                                                           ␣
↪              |
| failedStatusChecks | 0                                                                       ␣
↪              |
| id                 | 6c9d5842-3066-XXXX-XXXX-f07b1f73b948-007                                ␣
↪              |
| lastStatusCheck    | 4 minutes ago                                                           ␣
↪              |
| lastStatusMessage  | Transitioning from status CLEANING_UP to FINISHED in phase␣
↪ARCHIVING.          |
| lastUpdated        | 2021-09-22T21:40:52.194Z                                                ␣
↪              |
| maxHours           | 0.5                                                                     ␣
↪              |
| memoryPerNode      | 1.0                                                                     ␣
↪              |
| name               | fastqc-test                                                             ␣
↪              |
| nodeCount          | 1                                                                       ␣
↪              |
| owner              | eho                                                                     ␣
↪              |
| processorsPerNode  | 1                                                                       ␣
↪              |
| remoteEnded        | 4 minutes ago                                                           ␣
↪              |
| remoteJobId        | 8493758                                                                 ␣
↪              |
| remoteOutcome      | FINISHED                                                                ␣
↪              |
| remoteQueue        | normal                                                                  ␣
↪              |
| remoteStarted      | 2021-09-22T21:39:42.702Z                                                ␣
↪              |
```

(continues on next page)

(continued from previous page)

```
| remoteStatusChecks | 2                                                                                ␣
↪                |
| remoteSubmitted    | 5 minutes ago                                                                    ␣
↪                |
| schedulerJobId     | None                                                                             ␣
↪                |
| status             | FINISHED                                                                         ␣
↪                |
| submitRetries      | 0                                                                                ␣
↪                |
| systemId           | eho.stampede2.execution                                                          ␣
↪                |
| tenantId           | tacc.prod                                                                        ␣
↪                |
| tenantQueue        | aloe.jobq.tacc.prod.submit.DefaultQueue                                          ␣
↪                |
| visible            | True                                                                             ␣
↪                |
| workPath           | /scratch/06634/eho/eho/job-XXXXXXXX-XXXX-XXXX-XXXX-f07b1f73b948-
↪007-fastqc-test |
+--------------------+-------------------------------------------------------------------------
↪--------------+
```

We can also download and inspect the job outputs:

```
$ tapis jobs outputs download 6c9d5842-XXXX-XXXX-XXXX-f07b1f73b948-007
+-------------+-------+
| Field       | Value |
+-------------+-------+
| downloaded  | 11    |
| skipped     | 0     |
| messages    | 5     |
| elapsed_sec | 59    |
+-------------+-------+
$ cd 6c9d5842-3066-XXXX-XXXX-f07b1f73b948-007
$ cat ./*.err
# ...
Started analysis of example.fastq
$ cat ./*.out
Analysis complete for example.fastq
```

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search