A look behind Garcon

Introducing a lightweight library for AWS SWF.

Michael Ortali Follow
Mar 5, 2015 · 6 min read

We've recently open-sourced Garcon, a lightweight library that helps reduce the lengthy task of writing jobs with Amazon Simple Workflow to a few minutes. The library is available in Python 2.7 and 3.4. In this article, we give you a look behind Garcon.

Amazon SWF

Amazon Simple WorkFlow Service (SWF) makes it easy to build applications that coordinate units of work across distributed components. The service is based on the notion of *workflow*, which is a set of coordinated *activities* that carry out an objective.

For instance, *booking an online ticket* requires: creating an invoice, checking availability and reserving the seat, verifying user information and processing payment; when all of them are successful, sending out a confirmation email to the user and marking the invoice as paid. Some activities can be run in parallel if needed (sending email, updating invoice state.)

SWF works with *tasks* (small logical units of work), *activities* (runs specific list of tasks to perform), *deciders* (manage and orchestrate activities) and *executions* (start a workflow.) Except for start and completion of a workflow, each activity has a well-defined predecessor and successor.

Workflow executions always happen within a domain, which defines its scope and restrains the activity tasks it can run. For each domain, the SWF management console provides the history of all executions (active and closed), which includes the list of events and activities launched.

Entirely distributed and stateless, activities and tasks can be written in any language and distributed across many systems (on AWS and/or on premise). Parts of your workflow can be written in Java, while some others can be in go or Python, which allows you to use the language that performs best for a specific task. For a given workflow, you can also have as many activity workers and deciders as you need.

To use Amazon SWF, you can use any SDKs (provide high level access to SWF), and for Java/Ruby developers, Flow framework.

Workflows

Building workflows using boto (AWS Python SDK) directly can take some time. You have at least 2 main components to create: one or more activities and a decider. Both are workers which continuously poll information from SWF.

The activity

It executes a list of tasks which are small discrete logical units (going back to our earlier example of *booking a plane ticket*: an activity can process two tasks such as verifying user information and processing the payment.)

The activity marks itself as completed or failed. If an activity takes longer to execute it can send a heartbeat to SWF to avoid timing out. Activities can take an input and return a value which will be recorded as the activity response.

The decider

It schedules the different activities. If written manually, you need to find the next activities to execute which involves reading the execution event list (as shown in the example). If you add more features such as the ability to retry on activity failure, and/or running activities in parallel, your code gets quickly more complex.

The decider can send additional information to the activity (as an input), and it can retrieve the response of each activity regardless of success or failure.

Stepping back

That's how we've started to build Garcon, building a system that facilitates communication between the different components and taking in consideration the high-level requirements:

- Activities within a workflow can depend on other activities to be completed.
- Activities can require data as an input and can return a result.
- Tasks should be discrete, reusable components.

Code Sample

Before going onto the details, let's take a quick look at Garcon's implementation of Serial Activity Execution:

```
from garcon import activity
from garcon import runner


domain = 'dev'
name = 'boto_tutorial'
create = activity.create(domain, name)

a_tasks = create(
    name='a_tasks',
    run=runner.Sync(
        lambda context, activity: dict(result='Now don't be givin him sambuca!'))

b_tasks = create(
    name='b_tasks',
    requires=[a_tasks],
    run=runner.Sync(
        lambda context, activity: print(context)))

c_tasks = create(
    name='c_tasks',
    requires=[b_tasks],
    run=runner.Sync(
        lambda context, activity: print(context)))
```

gistfile1.py hosted with ❤ by GitHub view raw

By way of comparison, check out the equivalent implementation using only boto.

> **Notes:** Executing this code shows that the activity "*a_tasks*" returns a dictionary which hydrates the execution context. When the activity "*b_tasks*" is executed, the context

> passed for its execution contains the key/value previously passed as an output.

More examples (including runners) are available online.

Execution context

In Garcon, each execution has an execution context.

The execution context is created at the initialization of the workflow, and is progressively hydrated by the response of each activity. When the decider is ready to schedule a new activity, it looks in the event history, gets all the activity responses, merges them into one and schedule the activity with the appropriate context (if you know SWF, we use the *input* field.)

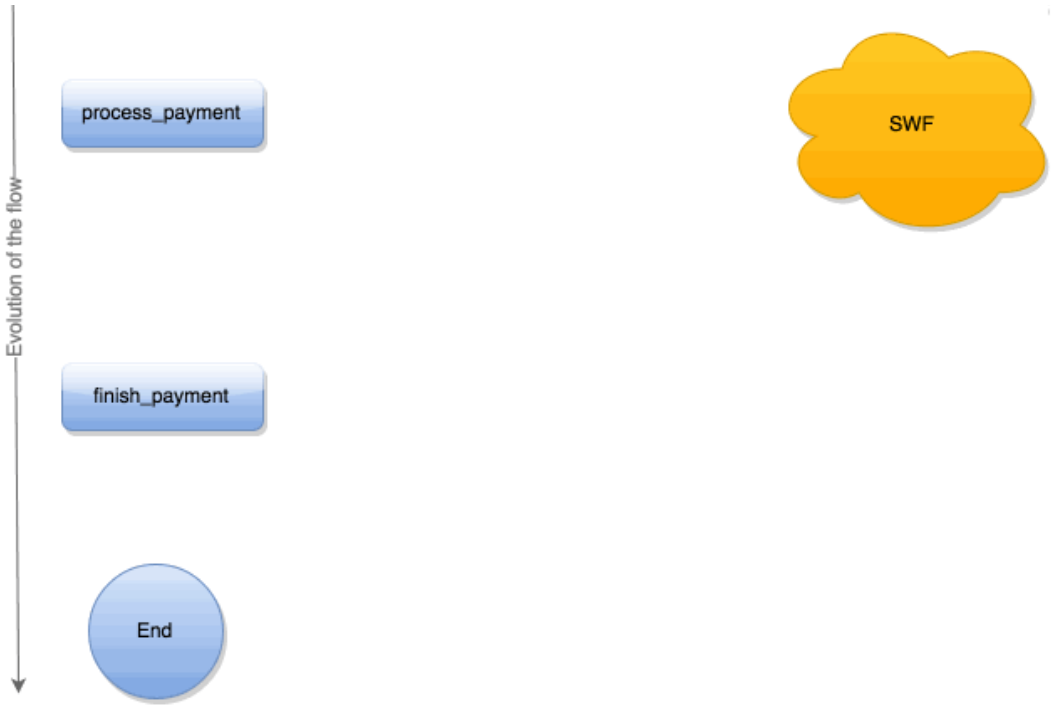
> **Note**: If you use *task.decorate* (see below) along with the *.fill* method, the decider will pluck all the information needed from the context and only send those to the activity. If you don't: the full context is sent to the activity. Remember: SWF has a limit on input / result of 32k characters.

```
1  import boto.swf.layer2 as swf
2
3  from garcon import activity
4  from garcon import runner
5  from garcon import task
6
7  domain = 'dev'
8  name = 'context_example'
9  create = activity.create(domain, name)
10
11
12 @task.decorate()
13 def bootstrap():
14     # The bootstrap is just here to provide more information
15     # in the context which we can use later on. It's for the
16     # purpose of this demo.
17     return dict(user_id='<user id>', email='<email id>')
18
19
20 @task.decorate()
21 def process_payment_for_user(user_id):
22     # Process the payment for a specific user
23     return dict(payment_id='<payment id>')
```

```python

@task.decorate()
def send_email_to_user(user_id, email, payment_id):
    # Send some information to the user about the payment id
    print('Email sent to {email} ({user_id}) about {payment_id}'.format(
        user_id=user_id,
        payment_id=payment_id,
        email=email))
    return


initialize = create(
    name='initialize',
    tasks=runner.Sync(
        bootstrap.fill()))

process = create(
    name='process_payment',
    requires=[initialize],
    tasks=runner.Sync(
        # The bootstrap returns `user_id` in the context. The value will be
        # captured from the context then sent to the activity.
        process_payment_for_user.fill(user_id='user_id')))

finish_payment = create(
    name='finish_payment',
    requires=[process],
    tasks=runner.Sync(send_email_to_user.fill(
        # Same as above except this time, we're trying to get more information
        # from the context.
        user_id='user_id',
        payment_id='payment_id',
        email='email')))
```

flow.py hosted with ❤️ by GitHub view raw

Will be executed as follows:



This graph illustrates information sent to activities and returned by activities. The decider is not represented here (to keep this illustration easy to understand.)

All values sent and returned from the activities can be tracked directly in the SWF console. Each activity call displays an *input* field and a *result* field.

Activity ID	Name	Version	Status
finish_payment-1	finish_payment	1.0	Completed
process_payment-1	process_payment	1.0	Completed
initialize-1	initialize	1.0	Completed

finish_payment [with Activity ID finish_payment-1] selected

Control:	*Not Specified*
Decision Task Completed Event Id:	16
Heartbeat Timeout:	10 minutes
Input:	{"user_id": "<user id>", "payment_id": "<payment id>", "email": "<email id>"}

Run of an execution.

Task Runners

Garcon provides two task runners:

- *Sync*: all tasks are launched in series. The task's response hydrates the local context, which is then passed to the next task. The last task defines the response of the activity.
- *Async*: each task will be ran asynchronously, each consumes the activity context . Each task response will be combined, and they define the response activity.

In this system, collisions can easily happen: so it's always good to namespace your responses as much as possible. Other point: always avoid returning the entire execution context in an activity's response.

Handling failures

In a distributed architecture activities might fail for various reasons (network, resource timing out, etc) and cause failures. If this is the case, you may benefit from the retry param.

This flag will look in the event history for activity failures and, until the count has been reached, the activity will keep trying to execute. One of our examples has random failures, so your activity creation will look similar to this:

```
1  test_activity_3 = create(
2      name='activity_3',
3      retry=10,
4      requires=[test_activity_1],
5      run=runner.Sync(activity_failure))
```

gistfile1.py hosted with ❤ by GitHub view raw

Activity Generators

Generators spawn one or more instances of an activity based on values provided in the context.

One of our use case includes a job that calls an API each day to get metrics for all the countries in the world. If the API fails for one country, the entire activity fails — retrying

it means we will have to restart the entire list of countries.

Instead of having one activity to do all calls, it's a lot more robust to have one activity per country and have a retry mechanism applied to it. Failures will only be contained for one country that has failed instead of all.

```
import boto.swf.layer2 as swf

from garcon import activity
from garcon import runner
from garcon import task
import random


domain = 'dev'
name = 'country_flow'
create = activity.create(domain, name)


def country_generator(context):
    # We limit this so you can more easily see the failures / retries.
    total_countries = 6
    for country_id in range(1, total_countries):
        yield {'generator.country_id': country_id}


@task.decorate()
def unstable_country_task(activity, country_id):
    num = int(random.random() * 4)
    base = 'activity_2_country_id_{country_id}'.format(
        country_id=country_id)

    if num == 3:
        # Make the task randomly fail.
        print(base, 'has failed')
        raise Exception('fails')

    print(base, 'has succeeded')


test_activity_1 = create(
    name='activity_1',
    tasks=runner.Sync(
```

```
38          lambda context, activity: print('activity_1')))
39
40  test_activity_2 = create(
41      name='activity_2',
42      requires=[test_activity_1],
43      generators=[
44          country_generator],
45      retry=3,
46      tasks=runner.Sync(
47          unstable_country_task.fill(country_id='generator.country_id')))
48
49  test_activity_3 = create(
50      name='activity_3',
51      requires=[test_activity_2],
52      tasks=runner.Sync(
53          lambda context, activity: print('end of flow')))
```

gistfile1.py hosted with ❤️ by GitHub view raw

Example output:

```
activity_1
activity_2_country_id_1 has succeeded
activity_2_country_id_2 has succeeded
activity_2_country_id_4 has succeeded
activity_2_country_id_3 has failed
activity_2_country_id_5 has succeeded
activity_2_country_id_3 has succeeded
end of flow
```

Note: *generators* takes a list of generators. If you have a flow that has a date range, list of countries, you can create activities that corresponds to one day and one specific countries. If you have 10 days in your range and 20 countries, you will run 200 activities.

Workflow registration

One of the requirements to execute a workflow is the registration of the domain, workflow type and activity types. In order to make this part easier, when you launch the decider, all required information will be pre-filled so the execution can begin right away.

Feedback / Contributions

Garcon is at a very early stage.

We're currently using it for a few lightweight production processes (such as pulling data from Redshift, computing it and putting it into Dynamodb.)

If you have feedback, questions, or simply want to contribute, we'd love to hear from you.

Michael
Currently listening *Breaking Your Locks*, *Myback*

Amazon Web Services Python Tech

About Write Help Legal

Get the Medium app