Garcon, Documentation

Release 0.0.4

Michael Ortali

Jan 06, 2018

Contents

Lightweight library for AWS SWF.

CHAPTER 1

Requirements

- Python 2.7, 3.4 (tested)
- Boto 2.34.0 (tested)

CHAPTER 2

Goal

The goal of this library is to allow the creation of Amazon Simple Workflow without the need to worry about the orchestration of the different activities and building out the different workers. This framework aims to help simple workflows. If you have a more complex case, you might want to use directly boto.

CHAPTER 3

Code sample

The code sample shows a workflow that has 4 activities. It starts with activity_1, which after being completed schedule activity_2 and activity_3 to be ran in parallel. The workflow ends after the completion of activity_4 which requires activity_2 and activity_3 to be completed:

```
from __future__ import print_function

from garcon import activity
from garcon import runner


domain = 'dev'
create = activity.create(domain)

test_activity_1 = create(
    name='activity_1',
    tasks=runner.Sync(
        lambda activity, context: print('activity_1')))

test_activity_2 = create(
    name='activity_2',
    requires=[test_activity_1],
    run=runner.Async(
        lambda activity, context: print('activity_2_task_1'),
        lambda activity, context: print('activity_2_task_2')))

test_activity_3 = create(
    name='activity_3',
    requires=[test_activity_1],
    run=runner.Sync(
        lambda activity, context: print('activity_3')))

test_activity_4 = create(
    name='activity_4',
    requires=[test_activity_3, test_activity_2],
    run=runner.Sync(
```

```
        lambda activity, context: print('activity_4')))
```

CHAPTER 4

Documentation

4.1 User's guide

4.1.1 Introduction

Garcon is a Python library for Amazon SWF, originally built at The Orchard.

The goal of this library is to allow the creation of workflows using SWF without the need to worry about the orchestration of the different activities, and build out the complex different workers.

Main Features:

- Simple: when you write a flow, the deciders and the activity workers are automatically generated. No extra work is required.
- Retry mechanisms: if an activity has failed, you can set a maximum of retries. It ends up very useful when you work with external APIs.
- Scalable timeouts: all the timeout are calculated and consider other running workflows.
- *Activity Generators*: some workflows requires more than one instance of a specific activity.

4.1.2 Code Sample

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

```
from garcon import activity
from garcon import runners


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)))
```

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

Note: 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. Same observation can be done in "c_tasks".

All activities are running in series. More examples (including runners) are available online.

4.1.3 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.

Note: Be aware that SWF has a limit on the number of events the history can hold, always make sure the number of activities spawned by the generator will allow enough room.

Example:

```
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(
        lambda context, activity: print('activity_1')))

test_activity_2 = create(
    name='activity_2',
    requires=[test_activity_1],
    generators=[
        country_generator],
    retry=3,
    tasks=runner.Sync(
        unstable_country_task.fill(country_id='generator.country_id')))

test_activity_3 = create(
    name='activity_3',
    requires=[test_activity_2],
    tasks=runner.Sync(
        lambda context, activity: print('end of flow')))
```

Note: Generators attribute 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.

4.2 Api

4.2.1 Activity

Activities are self generated classes to which you can pass an identifier, and a list of tasks to perform. The activities are in between the decider and the tasks.

For ease, two types of task runners are available: Sync and Async. If you need something more specific, you should either create your own runner, or you should create a main task that will then split the work.

Create an activity:

```
from garcon import activity

# First step is to create the workflow on a specific domain.
create = activity.create('domain')

initial_activity = create(
    # Name of your activity
```

```
    name='activity_name',

    # List of tasks to run (here we use the Sync runner)
    run=runner.Sync(task1),

    # No requires since it's the first one. Later in your flow, if you have
    # a dependency, just use the variable that contains the activity.
    requires=[],

    # If the activity fails, number of times you want to retry.
    retry=0,

    # If you want to run the activity `n` times, you can use a generator.
    generator=[generator_name])
```

class garcon.activity.**Activity**(***kwargs*)

Bases: boto.swf.layer2.ActivityWorker, *garcon.log.GarconLogger*

execute_activity(*context*)

Execute the runner.

Parameters **context** (*dict*) – The flow context.

hydrate(*data*)

Hydrate the task with information provided.

Parameters **data** (*dict*) – the data to use (if defined.)

instances(*context*)

Get all instances for one activity based on the current context.

There are two scenarios: when the activity worker has a generator and when it does not. When it doesn't (the most simple case), there will always be one instance returned.

Generators will however consume the context to calculate how many instances of the activity are needed – and it will generate them (regardless of their state.)

Parameters **context** (*dict*) – the current context.

Returns

all the instances of the activity (for a current workflow execution.)

Return type *list*

poll_for_activity(**args*, ***kwargs*)

Runs Activity Poll.

If a SWF throttling exception is raised during a poll, the poll will be retried up to 5 times using exponential backoff algorithm.

Upgrading to boto3 would make this retry logic redundant.

Parameters **identity** (*str*) – Identity of the worker making the request, which is recorded in the ActivityTaskStarted event in the AWS console. This enables diagnostic tracing when problems arise.

run(*identity=None*)

Activity Runner.

Information is being pulled down from SWF and it checks if the Activity can be ran. As part of the information provided, the input of the previous activity is consumed (context).

Parameters **activity_id** (*str*) – Identity of the worker making the request, which is recorded in the ActivityTaskStarted event in the AWS console. This enables diagnostic tracing when problems arise.

task_list = None

version = '1.0'

class garcon.activity.**ActivityInstance**(*activity_worker*, *local_context=None*, *execution_context=None*)

activity_name
Return the activity name of the worker.

create_execution_input()
Create the input of the activity from the context.

AWS has a limit on the number of characters that can be used (32k). If you use the *task.decorate*, the data sent to the activity is optimized to match the values of the context as well as the execution context.

Returns the input to send to the activity.

Return type dict

heartbeat_timeout
Return the heartbeat in seconds.

This heartbeat corresponds on when an activity needs to send a signal to swf that it is still running. This will set the value when the activity is scheduled.

Returns Task list timeout.

Return type int

id
Generate the id of the activity.

The id is crutial (not just important): it allows to indentify the state the activity instance in the event history (if it has failed, been executed, or marked as completed.)

Returns

composed of the activity name (task list), and the activity id.

Return type str

retry
Return the number of retries allowed (matches the worker.)

runner
Shortcut to get access to the runner.

Raises runner.RunnerMissing – an activity should always have a runner, if the runner is missing an exception is raised (we will not be able to calculate values such as timeouts without a runner.)

Returns the activity runner.

Return type Runner

schedule_to_close
Return the schedule to close timeout.

The schedule to close timeout is a simple calculation that defines when an activity (from the moment it has been scheduled) should end. It is a calculation between the schedule to start timeout and the activity timeout.

Returns Schedule to close timeout.

Return type int

schedule_to_start
Return the schedule to start timeout.

The schedule to start timeout assumes that only one activity worker is available (since swf does not provide a count of available workers). So if the default value is 5 minutes, and you have 10 instances: the schedule to start will be 50 minutes for all instances.

Returns Schedule to start timeout.

Return type int

timeout
Return the timeout in seconds.

This timeout corresponds on when the activity has started and when we assume the activity has ended (which corresponds in boto to start_to_close_timeout.)

Returns Task list timeout.

Return type int

exception garcon.activity.**ActivityInstanceNotReadyException**
Bases: exceptions.Exception

Exception when an activity instance is not ready.

Activity instances that are considered not ready are instances that have not completed.

class garcon.activity.**ActivityState**(*activity_id*)
Provides information about a specific activity instance state (if the instance is already scheduled, has failed, or has been completed.) Along with the default values, this class also provides additional metadata such as the result of an activity instance.

add_state(*state*)
Add a state in the activity execution.

Parameters **state** (*int*) – the state of the activity to add (see activity.py)

get_last_state()
Get the last state of the activity execution.

Returns the state of the activity (see: activity.py)

Return type int

ready
Check if an activity is ready.

result
Get the result.

set_result(*result*)
Set the result of the activity.

This method sometimes throws an exception: an activity id can only have one result.

Parameters **result** (*dict*) – Result of the activity.

wait()
Wait until ready.

class garcon.activity.**ActivityWorker**(*flow*, *activities=None*)

run()
Run the activities.

class garcon.activity.**ExternalActivity**(*timeout=None*, *heartbeat=None*)
Bases: *garcon.activity.Activity*

External activity

One of the main advantages of SWF is the ability to write a workflow that has activities written in any languages. The external activity class allows to write the workflow in Garcon and benefit from some features (timeout calculation among other things, sending context data.)

run()
Run the external activity.

This activity is handled outside, so the run method should remain unimplemented and return False (so the run loop stops.)

garcon.activity.**count_activity_failures**(*states*)
Count the number of times an activity has failed.

Parameters **states** (*dict*) – list of activity states.

Returns The number of times an activity has failed.

Return type int

garcon.activity.**create**(*domain*, *name*, *version='1.0'*, *on_exception=None*)
Helper method to create Activities.

The helper method simplifies the creation of an activity by setting the domain, the task list, and the activity dependencies (what other activities) need to be completed before this one can run.

Note: The task list is generated based on the domain and the name of the activity. Always make sure your activity name is unique.

Parameters

- **domain** (*str*) – the domain name.
- **name** (*str*) – name of the activity.
- **version** (*str*) – activity version.
- **on_exception** (*callable*) – the error handler.

Returns activity generator.

Return type callable

garcon.activity.**find_activities**(*flow*, *context*)
Retrieves all the activities from a flow.

Parameters **flow** (*module*) – the flow module.

Returns All the activity instances for the flow.

Return type *list*

garcon.activity.**find_available_activities**(*flow*, *history*, *context*)
Find all available activity instances of a flow.

The history contains all the information of our activities (their state). This method focuses on finding all the activities that need to run.

Parameters

- **flow** (*module*) – the flow module.
- **history** (*dict*) – the history information.
- **context** (*dict*) – from the context find the available activities.

garcon.activity.**find_uncomplete_activities**(*flow*, *history*, *context*)
Find uncomplete activity instances.

Uncomplete activities are all the activities that are not marked as completed.

Parameters

- **flow** (*module*) – the flow module.
- **history** (*dict*) – the history information.
- **context** (*dict*) – from the context find the available activities.

Yields *activity* – The available activity.

garcon.activity.**find_workflow_activities**(*flow*)
Retrieves all the activities from a flow

Parameters **flow** (*module*) – the flow module.

Returns all the activities.

Return type *list*

garcon.activity.**worker_runner**(*worker*)
Run indefinitely the worker.

Parameters **worker** (*object*) – the Activity worker.

4.2.2 Decider Worker

The decider worker is focused on orchestrating which activity needs to be executed and when based on the flow procided.

class garcon.decider.**DeciderWorker**(*flow*, *register=True*)
Bases: boto.swf.layer2.Decider, *garcon.log.GarconLogger*

create_decisions_from_flow(*decisions*, *activity_states*, *context*)
Create the decisions from the flow.

Simple flows don't need a custom decider, since all the requirements can be provided at the activity level. Discovery of the next activity to schedule is thus very straightforward.

Parameters

- **decisions** (*Layer1Decisions*) – the layer decision for swf.
- **activity_states** (*dict*) – all the state activities.
- **context** (*dict*) – the context of the activities.

delegate_decisions(*decisions*, *decider*, *history*, *context*)
Delegate the decisions.

For more complex flows (the ones that have, for instance, optional activities), you can write your own decider. The decider receives a method *schedule* which schedule the activity if not scheduled yet, and if scheduled, returns its result.

Parameters

- **decisions** (*Layer1Decisions*) – the layer decision for swf.
- **decider** (*callable*) – the decider (it needs to have schedule)
- **history** (*dict*) – all the state activities and its history.
- **context** (*dict*) – the context of the activities.

get_activity_states(*history*)
Get the activity states from the history.

From the full history extract the different activity states. Those states contain

Parameters **history** (*list*) – the full history.

Returns

list of all the activities and their state. It only contains activities that have been scheduled with AWS.

Return type dict

get_history(*poll*)
Get all the history.

The full history needs to be recovered from SWF to make sure that all the activities have been properly scheduled. With boto, only the last 100 events are provided, this methods retrieves all events.

Parameters **poll** (*object*) – The poll object (see AWS SWF for details.)

Returns All the events.

Return type *list*

register()
Register the Workflow on SWF.

To work, SWF needs to have pre-registered the domain, the workflow, and the different activities, this method takes care of this part.

run(*identity=None*)
Run the decider.

The decider defines which task needs to be launched and when based on the list of events provided. It looks at the list of all the available activities, and launch the ones that:

- are not been scheduled yet.
- have all the dependencies resolved.

If the decider is not able to find an uncompleted activity, the workflow can safely mark its execution as complete.

Parameters **identity** (*str*) – Identity of the worker making the request, which is recorded in the DecisionTaskStarted event in the AWS console. This enables diagnostic tracing when problems arise.

Returns

Always return true, so any loop on run can act as a long running process.

Return type boolean

class garcon.decider.**ScheduleContext**

The schedule context keeps track of all the current scheduling progress – which allows to easy determinate if there are more decisions to be taken or if the execution can be closed.

mark_uncompleted()

Mark the scheduling as completed.

When a scheduling is completed, it means all the activities have been properly scheduled and they have all completed.

garcon.decider.**ensure_requirements**(*requires*)

Ensure scheduling meets requirements.

Verify the state of the requirements to make sure the activity can be scheduled.

Parameters **requires** (list) – list of all requirements.

Throws:

ActivityInstanceNotReadyException: if one of the activity in the requirements is not ready.

garcon.decider.**schedule**(*decisions*, *schedule_context*, *history*, *context*, *schedule_id*, *current_activity*, *requires=None*, *input=None*, *version='1.0'*)

Schedule an activity.

Scheduling an activity requires all the requirements to be completed (all activities should be marked as completed). The scheduler also mixes the input with the full execution context to send the data to the activity.

Parameters

- **decisions** (*Layer1Decisions*) – the layer decision for swf.
- **schedule_context** (*dict*) – information about the schedule.
- **history** (*dict*) – history of the execution.
- **context** (*dict*) – context of the execution.
- **schedule_id** (*str*) – the id of the activity to schedule.
- **current_activity** (Activity) – the activity to run.
- **requires** (list) – list of all requirements.
- **input** (*dict*) – additional input for the context.

Throws:

ActivityInstanceNotReadyException: if one of the activity in the requirements is not ready.

Returns the state of the schedule (contains the response).

Return type State

garcon.decider.**schedule_activity_task**(*decisions*, *instance*, *version='1.0'*, *id=None*)

Schedule an activity task.

Parameters

- **decisions** (*Layer1Decisions*) – the layer decision for swf.
- **instance** (ActivityInstance) – the activity instance to schedule.

- **version** (*str*) – the version of the activity instance.
- **id** (*str*) – optional id of the activity instance.

garcon.event.**activity_states_from_events**(*events*)
Get activity states from a list of events.

The workflow events contains the different states of our activities. This method consumes the logs, and regenerates a dictionnary with the list of all the activities and their states.

Note: Please note: from the list of events, only activities that have been registered are accessible. For all the others that have not yet started, they won't be part of this list.

Parameters **events** (*dict*) – list of all the events.

Returns the activities and their state.

Return type *dict*

garcon.event.**get_current_context**(*events*)
Get the current context from the list of events.

Each activity returns bits of information that needs to be provided to the next activities.

Parameters **events** (list) – List of events.

Returns The current context.

Return type dict

Garcon logger module

class garcon.log.**GarconLogger**
This class is meant to be extended to get the Garcon logger feature The logger injects the execution context into the logger name.

This is used by the Activity class in Garcon and allows you to log from the activity object. Typically, you can log from a Garcon task and it will prefix your log messages with execution context information (domain, workflow_id, run_id).

Requirements: Your loggers need to be set up so there is at least one of them with a name prefixed with LOGGER_PREFIX. The Garcon logger will inherit the handlers from that logger.

The formatter for your handler(s) must use the logger name. Formatter Example:

```
%(asctime)s - %(name)s - %(levelname)s - %(message)s
```

This formatter will generate a log message as follow: '2015-01-15 - garcon.[domain].[workflow_id].[run_id] - [level] - [message]'

logger
Return the appropriate logger. Default to LOGGER_PREFIX if no logger name was set.

Returns a logger object

Return type logging.Logger

set_log_context(*execution_context*)
Set a logger name with execution context passed in.

Parameters **execution_context** (*dict*) – execution context information

unset_log_context()
Unset the logger name.

garcon.log.**get_logger_namespace**(*execution_context*)
Return the logger namespace for a given execution context.

Parameters **execution_context** (*dict*) – execution context information

4.2.3 Task runners

The task runners are responsible for running all the tasks (either in series or in parallel). There's only one task runner per activity.

class garcon.runner.**Async**(**args*, ***kwargs*)
Bases: *garcon.runner.BaseRunner*

execute(*activity*, *context*)

class garcon.runner.**BaseRunner**(**args*)

execute(*activity*, *context*)
Execution of the tasks.

heartbeat(*context*)
Calculate and return the heartbeat for an activity.

The heartbeat represents when an actvitity should be sending a signal to SWF that it has not completed yet. The heartbeat is sent everytime a new task is going to be launched.

Similar to the *BaseRunner.timeout*, the heartbeat is pessimistic, it looks at the largest heartbeat and set it up.

Returns

The heartbeat timeout (boto requires the timeout to be a string not a regular number.)

Return type str

requirements(*context*)
Find all the requirements from the list of tasks and return it.

If a task does not use the *task.decorate*, no assumptions can be made on which values from the context will be used, and it will raise an exception.

Raise:

NoRequirementFound: The exception when no requirements have been mentioned in at least one or more tasks.

Returns the list of the required values from the context.

Return type set

timeout(*context*)
Calculate and return the timeout for an activity.

The calculation of the timeout is pessimistic: it takes the worse case scenario (even for asynchronous task lists, it supposes there is only one thread completed at a time.)

Returns

The timeout (boto requires the timeout to be a string and not a regular number.)

Return type str

class garcon.runner.**External**(*timeout=None*, *heartbeat=None*)
Bases: *garcon.runner.BaseRunner*

exception garcon.runner.**NoRunnerRequirementsFound**
Bases: exceptions.Exception

exception garcon.runner.**RunnerMissing**
Bases: exceptions.Exception

class garcon.runner.**Sync**(**args*)
Bases: *garcon.runner.BaseRunner*

execute(*activity*, *context*)

4.2.4 Task

Tasks are small discrete applications that are meant to perform a defined action within an activity. An activity can have more than one task, they can run in series or in parallel.

Tasks can add values to the context by returning a dictionary that contains the informations to add (useful if you need to pass information from one task – in an activity, to another activity's task.)

Note: If you need a task runner that is not covered by the two scenarios below, you may need to just have a main task, and have this task split the work the way you want.

---

garcon.task.**contextify**(*fn*)
Decorator to take values from the context and apply them to fn.

The goal of this decorator is to allow methods to be called with different values from the same context. For instance: if you need to increase the throughtput of two different dynamodb tables, you will need to pass a table name, table index, and the new throughtput.

If you have more than one table, it gets difficult to manage. With this decorator, it's a little easier:

```
@contextify
def increase_dynamodb_throughtput(
        activity, context, table_name=None, table_index=None,
        table_throughtput=None):
    print(table_name)
activity_task = increase_dynamodb_throughtput.fill(
    table_name='dynamodb.table_name1',
    table_index='dynamodb.table_index1',
    table_throughtput='dynamodb.table_throughtput1')
context = dict(
    'dynamodb.table_name1': 'table_name',
    'dynamodb.table_index1': 'index',
    'dynamodb.table_throughtput1': 'throughtput1')
activity_task(..., context) # shows table_name
```

garcon.task.**decorate**(*timeout=None*, *heartbeat=None*, *enable_contextify=True*)
Generic task decorator for tasks.

Parameters

- **timeout** (*int*) – The timeout of the task (see timeout).
- **heartbeat** (*int*) – The heartbeat timeout.

- **contextify** (*boolean*) – If the task can be contextified (see contextify).

Returns The wrapper.

Return type callable

garcon.task.**fill_function_call**(*fn*, *requirements*, *activity*, *context*)
Fill a function calls from values from the context to the variable.

Parameters

- **fn** (*callable*) – the function to call.
- **requirements** (*dict*) – the requirements. The key represent the variable name and the value represents where the value is in the context.
- **activity** (ActivityWorker) – the current activity worker.
- **context** (*dict*) – the current context.

Returns The arguments to call the method with.

Return type dict

garcon.task.**flatten**(*callables*, *context=None*)
Flatten the tasks.

The task list is a mix of tasks and generators. The task generators are consuming the context and spawning new tasks. This method flattens everything into one list.

Parameters **callables** (list) – list of callables (including tasks and generators.)

Yields *callable* – one of the task.

garcon.task.**is_task_list**(*fn*)
Check if a function is a task list.

Returns if a function is a task list.

Return type boolean

garcon.task.**list**(*fn*)
Wrapper for a callable to define a task generator.

Generators are used to check values in the context and schedule different tasks based on it. Note: depending on the tasks returned by the generator, the timout values will be calculated differently.

For instance:

```
@task.list
def create_client(context):
    yield create_user.fill(
        username='context.username',
        email='context.email')
    if context.get('context.credit_card'):
        yield create_credit_card.fill(
            username='context.username',
            credit_card='context.credit_card')
    yield send_email.fill(email='context.email')
```

garcon.task.**namespace_result**(*dictionary*, *namespace*)
Namespace the response

This method takes the keys in the map and add a prefix to all the keys (the namespace):

```
resp = dict(key='value', index='storage')
namespace_response(resp, 'namespace')
# Returns: {'namespace.index': 'storage', 'namespace.key': 'value'}
```

Parameters

- **dictionary** (*dict*) – The dictionary to update.
- **map** (*dict*) – The keys to update.

Returns the updated dictionary

Return type Dict

garcon.task.**timeout**(*time*, *heartbeat=None*)

Wrapper for a task to define its timeout.

Parameters

- **time** (*int*) – the timeout in seconds
- **heartbeat** (*int*) – the heartbeat timeout (in seconds too.)

4.2.5 Utils

garcon.utils.**create_dictionary_key**(*dictionary*)

Create a key that represents the content of the dictionary.

Parameters **dictionary** (*dict*) – the dictionary to use.

Returns the key that represents the content of the dictionary.

Return type str

garcon.utils.**non_throttle_error**(*swf_response_error*)

Activity Runner.

Determine whether SWF Exception was a throttle or a different error.

Parameters **error** – boto.exception.SWFResponseError instance.

Returns True if SWFResponseError was a throttle, False otherwise.

Return type bool

garcon.utils.**throttle_backoff_handler**(*details*)

Callback to be used when a throttle backoff is invoked.

For more details see: https://github.com/litl/backoff/#event-handlers

Parameters **dictionary** (*dict*) – Details of the backoff invocation. Valid keys include:

target: reference to the function or method being invoked. args: positional arguments to func. kwargs: keyword arguments to func. tries: number of invocation tries so far. wait: seconds to wait (on_backoff handler only). value: value triggering backoff (on_predicate decorator only).

4.3 Release notes

4.3.1 What's new in Garcon 0.0.4

May 12th, 2015

External Activities

SWF allows any activities to be written in any language (allowing the service to be fully polyglot). This change makes it easier for Garcon to handle and work with external activities.

To define an external activity: simply set the flag *external* on the activity creation, and define (it's mandatory) the *timeout* and, if needed, the *heartbeat* timeout.

Change: *#52 <https://github.com/xethorn/garcon/pull/52/>`*

CHAPTER 5

Licence

Python Module Index

g

Index

L

M

N

P

R

S

T

U

V

W