gordon-dns

Release 0.0.1.dev10

Apr 05, 2023

Contents

Service to consume hostname-related events from a pub/sub and add, update, & delete records for a 3rd party DNS provider.

Release v0.0.1.dev10 (*What's new?*).

Warning: This is still in the planning phase and under active development. Gordon should not be used in production, yet.

CHAPTER 1

Requirements

For the initial release, the following will be supported:

- Python 3.6
- Google Cloud Platform

Support for other Python versions and cloud providers may be added.

CHAPTER 2

Development

For development and running tests, your system must have all supported versions of Python installed. We suggest using pyenv.

2.1 Setup

```
$ git clone git@github.com:spotify/gordon.git && cd gordon
# make a virtualenv
(env) $ pip install -r dev-requirements.txt
```

2.2 Running tests

To run the entire test suite:

```
# outside of the virtualenv
# if tox is not yet installed
$ pip install tox
$ tox
```

If you want to run the test suite for a specific version of Python:

```
# outside of the virtualenv
$ tox -e py36
```

To run an individual test, call `pytest` directly:

```
# inside virtualenv
(env) $ pytest tests/test_foo.py
```

2.3 Build docs

To generate documentation:

```
(env) $ pip install -r docs-requirements.txt
(env) $ cd docs && make html  # builds HTML files into _build/html/
(env) $ cd _build/html
(env) $ python -m http.server $PORT
```

Then navigate to `localhost:$PORT`!

To watch for changes and automatically reload in the browser:

```
(env) $ cd docs
(env) $ make livehtml  # default port 8888
# to change port
(env) $ make livehtml PORT=8080
```

CHAPTER 3

Code of Conduct

This project adheres to the Open Code of Conduct. By participating, you are expected to honor this code.

CHAPTER 4

User's Guide

4.1 Configuring the Gordon Service

Main module to run the Gordon service.

The service expects a `gordon.toml` and/or a `gordon-user.toml` file for configuration in the current working directory, or in a provided root directory.

Any configuration defined in `gordon-user.toml` overwrites those in `gordon.toml`.

Example:

```
$ python gordon/main.py
$ python gordon/main.py -c /etc/default/
$ python gordon/main.py --config-root /etc/default/
```

4.1.1 Example Configuration

An example of a `gordon.toml` file:

```
# Gordon Core Config
[core]
plugins = ["foo.plugin"]
debug = false
metrics = "ffwd"

[core.route]
consume = "enrich"
enrich = "publish"
publish = "cleanup"

[core.logging]
level = "info"
handlers = ["syslog"]
```

```
fmt = "%(created)f %(levelno)d %(message)s"
date_fmt = "%Y-%m-%dT%H:%M:%S"
address = ["10.99.0.1", "514"]


# Plugin Config
["foo"]
# global config to the general "foo" package
bar = baz

["foo.plugin"]
# specific plugin config within "foo" package
baz = bla
```

You may choose to have a `gordon-user.toml` file for development. All tables are deep merged into `gordon.toml`, to limit the amount of config duplication needed. For example, you can override `core.debug` without having to redeclare which plugins you'd like to run.

```
[core]
debug = true

[core.logging]
level = "debug"
handlers = ["stream"]
```

4.1.2 Supported Configuration

The following sections are supported:

core

plugins=LIST-OF-STRINGS
: Plugins that the Gordon service needs to load. If a plugin is not listed, Gordon will skip it even if there's configuration.

  The strings must match the plugin's config key. See the plugin's documentation for config key names.

debug=true|false
: Whether or not to run the Gordon service in `debug` mode.

  If `true`, Gordon will continue running even if installed & configured plugins can not be loaded. Plugin exceptions will be logged as warnings with tracebacks.

  If `false`, Gordon will exit out if it can't load one or more plugins.

metrics=STR
: The metrics provider to use. Depending on the provider, more configuration may be needed. See provider implementation for details.

core.logging

level=info(default)|debug|warning|error|critical
: Any log level that is supported by the Python standard `logging` library.

handlers=LIST-OF-STRINGS
: `handlers` support any of the following handlers: `stream`, `syslog`, and `stackdriver`. Multiple handlers are supported. Defaults to `syslog` if none are defined.

> **Note:** If `stackdriver` is selected, `ulogger[stackdriver]` needs to be installed as its dependencies are not installed by default.

Other key-value pairs as supported by ulogger will be passed into the configured handlers. For example:

```
[core.logging]
level = "info"
handlers = ["syslog"]
address = ["10.99.0.1", "514"]
fmt = "%(created)f %(levelno)d %(message)s"
date_fmt = "%Y-%m-%dT%H:%M:%S"
```

core.route

A table of key-value pairs of phases used to indicate the route the a message should take. All keys should correspond to either the *start_phase* attribute of a runnable plugin or the *phase* of a message handling plugin. Values may only correspond to *phase* of a message handling plugin.

```
[core.route]
start_phase = "phase2"
phase2 = "phase3"
```

4.2 Gordon's Plugin System

Module for loading plugins distributed via third-party packages.

Plugin discovery is done via `entry_points` defined in a package's `setup.py`, registered under `'gordon.plugins'`. For example:

```
# setup.py
from setuptools import setup

setup(
    name=NAME,
    # snip
    entry_points={
        'gordon.plugins': [
            'gcp.gpubsub = gordon_gcp.gpubsub:EventClient',
            'gcp.gce.a = gordon_gcp.gce.a:ReferenceSourceClient',
            'gcp.gce.b = gordon_gcp.gce.b:ReferenceSourceClient',
            'gcp.gdns = gordon_gcp.gdns:DNSProviderClient',
        ],
    },
    # snip
)
```

Plugins are initialized with any config defined in `gordon-user.toml` and `gordon.toml`. See *Configuring the Gordon Service* for more details.

Once a plugin is found, the loader looks up its configuration via the same key defined in its entry point, e.g. `gcp.gpubsub`.

The value of the entry point (e.g. `gordon_gcp.gpubsub:EventClient`) must point to a class. The plugin class is instantiated with its config.

A plugin will not have access to another plugin's configuration. For example, the `gcp.gpusub` will not have access to the configuration for `gcp.gdns`.

See *Gordon's Plugin System* for details on how to write a plugin for Gordon.

4.2.1 Writing a Plugin

Todo: Add documentation once interfaces are firmed up

4.3 API Reference

4.3.1 main

Main module to run the Gordon service.

The service expects a `gordon.toml` and/or a `gordon-user.toml` file for configuration in the current working directory, or in a provided root directory.

Any configuration defined in `gordon-user.toml` overwrites those in `gordon.toml`.

Example:

```
$ python gordon/main.py
$ python gordon/main.py -c /etc/default/
$ python gordon/main.py --config-root /etc/default/
```

`gordon.main.`**`setup`**(*config_root=''*)

Service configuration and logging setup.

Configuration defined in `gordon-user.toml` will overwrite `gordon.toml`.

Parameters **`config_root`** (*str*) – Where configuration should load from, defaults to current working directory.

Returns A dict for Gordon service configuration.

4.3.2 router

Core message routing logic for the plugins within Gordon Service.

Messages received on the success channel will be routed to the next designated plugin phase. For example, a message that has a `consume` phase will be routed to the installed enricher provider (or publisher provider if no enricher provider is installed).

If a message fails its next phase, its phase will be updated to `drop` and routed to the event consumer provider for cleanup.

Attention: The *GordonRouter* only supports the following two phase routes:

1. consume -> enrich -> publish -> done
2. consume -> publish -> done

Future releases may support more configurable phase routings.

class gordon.router.**GordonRouter**(*phase_route*, *success_channel*, *error_channel*, *plugins*, *metrics*)

Route messages to the appropriate plugin destination.

Attention: *error_channel* is currently not used in this router, and may be removed entirely from all interface definitions.

Parameters

- **phase_route** (*dict(str, str)*) – The route messages should follow.
- **success_channel** (*asyncio.Queue*) – A sink for successfully processed *gordon.interfaces.IEventMessage* s.
- **error_channel** (*asyncio.Queue*) – A sink for *gordon.interfaces.IEventMessage* s that were not processed due to problems.
- **plugins** (*list*) – Instantiated message handling plugins.
- **metrics** (*obj*) – Implemented `IMetricRelay` interface to emit metrics.

4.3.3 plugins_loader

Module for loading plugins distributed via third-party packages.

Plugin discovery is done via `entry_points` defined in a package's `setup.py`, registered under `'gordon.plugins'`. For example:

```
# setup.py
from setuptools import setup

setup(
    name=NAME,
    # snip
    entry_points={
        'gordon.plugins': [
            'gcp.gpubsub = gordon_gcp.gpubsub:EventClient',
            'gcp.gce.a = gordon_gcp.gce.a:ReferenceSourceClient',
            'gcp.gce.b = gordon_gcp.gce.b:ReferenceSourceClient',
            'gcp.gdns = gordon_gcp.gdns:DNSProviderClient',
        ],
    },
    # snip
)
```

Plugins are initialized with any config defined in `gordon-user.toml` and `gordon.toml`. See *Configuring the Gordon Service* for more details.

Once a plugin is found, the loader looks up its configuration via the same key defined in its entry point, e.g. `gcp.gpubsub`.

The value of the entry point (e.g. `gordon_gcp.gpubsub:EventClient`) must point to a class. The plugin class is instantiated with its config.

A plugin will not have access to another plugin's configuration. For example, the `gcp.gpusub` will not have access to the configuration for `gcp.gdns`.

See *Gordon's Plugin System* for details on how to write a plugin for Gordon.

`gordon.plugins_loader.`**`load_plugins`**(*config*, *plugin_kwargs*)
Discover and instantiate plugins.

Parameters

- **`config`** (*dict*) – loaded configuration for the Gordon service.
- **`plugin_kwargs`** (*dict*) – keyword arguments to give to plugins during instantiation.

Returns list of names of plugins, list of instantiated plugin objects, and any errors encountered while loading/instantiating plugins. A tuple of three empty lists is returned if there are no plugins found or activated in gordon config.

Return type Tuple of 3 lists

4.3.4 interfaces

`interface` `gordon.interfaces.`**`IEventMessage`**
A discrete unit of work for Gordon to process.

Gordon expects plugins to return or accept objects that implement this interface in order to route them to other plugins, and handle retries or cleanup in case of errors.

`msg_id`
Identifier for the event message instance.

`phase`
Variable phase of the event message.

`__init__`(*msg_id*, *data*, *history_log*, *phase=None*)
Initialize an EventMessage.

Parameters

- **`msg_id`** (*str*) – Unique message identifier.
- **`data`** (*dict*) – Data required to update DNS records.
- **`history_log`** (*list*) – Log of actions performed on message.
- **`phase`** (*str*) – Current phase.

`append_to_history`(*entry*, *plugin_phase*)
Append entry to the IEventMessage's history_log.

Parameters

- **`entry`** (*str*) – Information to append to log.
- **`plugin_phase`** (*str*) – Phase of plugin that created the log entry message.

`update_phase`(*new_phase*)
Update the phase of a message to new phase.

Parameters **new_phase** (*str*) – Phase to update the message to.

interface gordon.interfaces.**IRunnable**
Extends: *gordon.interfaces.IGenericPlugin*

Runnable plugin to produce event messages for Gordon to process.

The plugin also has the ability to send `gordon.interfaces EventMessage` objects to both success and error channels. At least one runnable plugin is required to run Gordon.

start_phase
Starting phase for event messages.

__init__(*config*, *success_channel*, *error_channel*, *metrics*, ***kwargs*)
Initialize a runnable plugin.

Parameters

- **config** (*dict*) – Plugin-specific configuration.
- **success_channel** (*asyncio.Queue*) – A sink for successfully processed IEventMessages.
- **error_channel** (*asyncio.Queue*) – A sink for IEventMessages that were not processed due to problems.
- **metrics** (*obj*) – Optional obj used to emit metrics.

run()
Begin consuming messages using the provided event loop.

interface gordon.interfaces.**IMessageHandler**
Extends: *gordon.interfaces.IGenericPlugin*

Plugin which performs some operation on an event message.

The Gordon core router will use its *phase_route* to direct messages produced by any runnable plugins the appropriate message handling plugins, identified by their phase attribute. At least one message handling plugin is required to run Gordon.

phase
Plugin phase

__init__(*config*, *metrics*, ***kwargs*)
Initialize a message handler.

Parameters

- **config** (*dict*) – Plugin-specific configuration.
- **metrics** (*obj*) – Obj used to emit metrics.

handle_message(*event_message*)
Perform some operation on or triggered by an event message.

Parameters **event_message** (*IEventMessage*) – Message on which to operate.

interface gordon.interfaces.**IGenericPlugin**
Do not implement this interface directly.

Use *gordon.interfaces.IRunnable*, or *gordon.interfaces.IMessageHandler* instead.

shutdown()
Perform any actions required to gracefully shutdown plugin.

interface gordon.interfaces.**IMetricRelay**
Manage Gordon metrics.

incr(*metric_name*, *value=1*, *context=None*, ***kwargs*)
Increase a metric by 1 or a given amount.

Parameters

- **metric_name** (*str*) – Identifier of the metric.
- **value** (*int*) – (optional) Value with which to increase the metric.
- **context** (*dict*) – (optional) Additional key-value pairs which further describe the metric, for example: {'remote-host': '1.2.3.4'}

timer(*metric_name*, *context=None*, ***kwargs*)
Get a timer object which implements ITimer.

Parameters

- **metric_name** (*str*) – Identifier of the metric.
- **context** (*dict*) – (optional) Additional key-value pairs which further describe the metric, for example: {'unit': 'seconds'}

set(*metric_name*, *value*, *context=None*, ***kwargs*)
Set a metric to a given value.

Parameters

- **metric_name** (*str*) – Identifier of the metric.
- **value** (*number*) – The value of the metric.
- **context** (*dict*) – (optional) Additional key-value pairs which further describe the metric, for example: {'app-version': '1.5.3'}

cleanup(***kwargs*)
Perform cleanup tasks related to metrics handling.

4.4 Metrics Implementations

4.4.1 ffwd

Gordon ships with a simple ffwd metrics implementation, which can be enabled via configuration. This module contains the SimpleFfwdRelay, and all required classes that it uses to send messsages to the ffwd daemon via UDP.

The *SimpleFfwdRelay* requires no configuration, but can be customized. The defaults that may be overridden are shown below.

```
[ffwd]
# to identify the service creating metrics
key = 'gordon-service'

# the address of the ffwd daemon (see: UDPClient)
ip = "127.0.0.9"
port = 19000

# a scaling factor for timing (see: FfwdTimer)
time_unit = 1E9
```

class gordon.metrics.ffwd.**SimpleFfwdRelay**(*config*)
Metrics relay which sends to FFWD immediately.

The relay does no client-side aggregation and metrics are emitted immediately. The relay uses a combination of the key and attributes fields to semantically identify metrics in ffwd.

Parameters **config** (*dict*) – Configuration with optional keys described above.

cleanup()
Not used.

incr(*metric_name*, *value=1*, *context=None*, ***kwargs*)
Increase a metric by 1 or a given amount.

Parameters

- **metric_name** (*str*) – Identifier of the metric.
- **value** (*int*) – (optional) Value with which to increase the metric (default: 1).
- **context** (*dict*) – (optional) Additional key-value pairs which further describe the metric, for example: {'remote-host': '1.2.3.4'}

set(*metric_name*, *value*, *context=None*, ***kwargs*)
Set a metric to a given value.

Parameters

- **metric_name** (*str*) – Identifier of the metric.
- **value** (*number*) – The value of the metric.
- **context** (*dict*) – (optional) Additional key-value pairs which further describe the metric, for example: {'app-version': '1.5.3'}

timer(*metric_name*, *context=None*, ***kwargs*)
Create a FfwdTimer.

Parameters

- **metric_name** (*str*) – Identifier of the metric.
- **context** (*dict*) – (optional) Additional key-value pairs which further describe the metric, for example: {'unit': 'seconds'}

class gordon.metrics.ffwd.**FfwdTimer**(*metric*, *udp_client*, *time_unit=None*)
Timer which sends UDP messages to FFWD on completion.

Parameters

- **metric** (*dict*) – Dict representation of the metric to send.
- **udp_client** (UDPClient) – A metric sending client.
- **time_unit** (*number*) – (optional) Scale time unit for use with time.perf_counter(), for example: 1E9 to send nanoseconds.

start()
Start timer.

stop()
Stop timer.

class gordon.metrics.ffwd.**UDPClient**(*ip=None*, *port=None*, *loop=None*)
Client for sending UDP datagrams.

Parameters

- **ip** (*str*) – (optional) Destination IP address (default: 127.0.0.1).
- **port** (*int*) – (optional) Destination port (default: 9000).
- **loop** (*asyncio.AbstractEventLoop impl*) – (optional) Event loop.

send(*metric*)
Transform metric to JSON bytestring and send to server.

Parameters **metric** (*dict*) – Complete metric to send as JSON.

class gordon.metrics.ffwd.**UDPClientProtocol**(*message*)
Protocol for sending one-off messages via UDP.

Parameters **message** (*bytes*) – Message for ffwd agent.

connection_made(*transport*)
Create connection, use to send message and close.

Parameters **transport** (*asyncio.DatagramTransport*) – Transport used for sending.

CHAPTER 5

Project Information

5.1 License and Credits

`gordon` is licensed under the Apache 2.0 license. The full license text can be also found in the source code repository.

5.2 How to Contribute

Every open source project lives from the generous help by contributors that sacrifice their time and `gordon` is no different.

This project adheres to the Open Code of Conduct. By participating, you are expected to honor this code. If the core project maintainers/owners feel that this Code of Conduct has been violated, we reserve the right to take appropriate action, including but not limited to: private or public reprimand; temporary or permanent ban from the project; request for public apology.

5.2.1 Communication/Support

Feel free to drop by the Spotify FOSS Slack organization in the #gordon channel.

5.2.2 Contributor Guidelines/Requirements

Contributors should expect a response within one week of an issue being opened or a pull request being submitted. More time should be allowed around holidays. Feel free to ping your issue or PR if you have not heard a timely response.

Submitting Bugs

Before submitting, users/contributors should do the following:

- **Basic troubleshooting:**
  - Make sure you're on the latest supported version. The problem may be solved already in a later release.
  - Try older versions. If you're on the latest version, try rolling back a few minor versions. This will help maintainers narrow down the issue.
  - Try the same for dependency versions - up/downgrading versions.
- Search the project's issues to make sure it's not already known, or if there is already an outstanding pull request to fix it.
- If you don't find a pre-existing issue, check the discussion on Slack. There may be some discussion history, and if not, you can ask for help in figuring out if it's a bug or not.

What to include in a bug report:

- What version of Python is being used? i.e. 2.7.13, 3.6.2, PyPy 2.0
- What operating system are you on? i.e. Ubuntu 14.04, RHEL 7.4
- What version(s) of the software are you using?
- How can the developers recreate the bug? Steps to reproduce or a simple base case that causes the bug is extremely helpful.

Contributing Patches

No contribution is too small. We welcome fixes for typos and grammar bloopers just as much as feature additions and fixes for code bloopers!

- Check the outstanding issues and pull requests first to see if development is not already being done for what you which to change/add/fix.
- If an issue has the `available` label on it, it's up for grabs for anyone to work on. If you wish to work on it, just comment on the ticket so we can remove the `available` label.
- Do not break backwards compatibility.
- Once any feedback is addressed, please comment on the pull request with a short note, so we know that you're done.
- Write good commit messages.

Workflow

- This project follows the gitflow branching model. Please name your branch accordingly.
- Always make a new branch for your work, no matter how small. Name the branch a short clue to the problem you're trying to fix or feature you're adding.
- Ideally, a branch should map to a pull request. It is possible to have multiple pull requests on one branch, but is discouraged for simplicity.
- Do not submit unrelated changes on the same branch/pull request.
- Multiple commits on a branch/pull request is fine, but all should be atomic, and relevant to the goal of the PR. Code changes for a bug fix, plus additional tests (or fixes to tests) and documentation should all be in one commit.
- Pull requests should be rebased off of the `develop` branch.

- To finish and merge a release branch, project maintainers should first create a PR to merge the branch into `develop`. Then, they should merge the release branch into `master` locally and push to master afterwards.
- Bugfixes meant for a specific release branch should be merged into that branch through PRs.

Code

- See docs on how to setup your environment for development.
- Code should follow the Google Python Style Guide.
- **Documentation is not optional.**
  - Docstrings are required for public API functions, methods, etc. Any additions/changes to the API functions should be noted in their docstrings (i.e. "added in 2.5")
  - If it's a new feature, or a big change to a current feature, consider adding additional prose documentation, including useful code snippets.
- **Tests aren't optional.**
  - Any bug fix should have a test case that invokes the bug.
  - Any new feature should have test coverage hitting at least $PERCENTAGE.
  - Make sure your tests pass on our CI. You will not get any feedback until it's green, unless you ask for help.
  - Write asserts as "expected == actual" to avoid any confusion.
  - Add good docstrings for test cases.

Github Labels

The super secret decoder ring for the labels applied to issues and pull requests.

Triage Status

- `needs triaging`: a new issue or pull request that needs to be triaged by the goalie
- `no repro`: a filed (closed) bug that can not be reproduced - issue can be reopened and commented upon for more information
- `won't fix`: a filed issue deemed not relevant to the project or otherwise already answered elsewhere (i.e. questions that were answered via linking to documentation or stack overflow, or is about GCP products/something we don't own)
- `duplicate`: a duplicate issue or pull request
- `waiting for author`: issue/PR has questions or requests feedback, and is awaiting the other for a response/update

Development Status

To be prefixed with `Status:`, e.g. `Status: abandoned`.

- `abandoned`: issue or PR is stale or otherwise abandoned

- `available`: bug/feature has been confirmed, and is available for anyone to work on (but won't be worked on by maintainers)
- `blocked`: issue/PR is blocked (reason should be commented)
- `completed`: issue has been addressed (PR should be linked)
- `wip`: issue is currently being worked on
- `on hold`: issue/PR has development on it, but is currently on hold (reason should be commented)
- `pending`: the issue has been triaged, and is pending prioritization for development by maintainers
- `review needed`: awaiting a review from project maintainers

Types

To be prefixed with `Type:` e.g. `Type:  bug`.

- `bug`: a bug confirmed via triage
- `feature`: a feature request/idea/proposal
- `improvement`: an improvement on existing features
- `maintenance`: a task for required maintenance (e.g. update a dependency for security patches)
- `extension`: issues, feature requests, or PRs that support other services/libraries separate from core

5.2.3 Local Development Environment

TODO

5.3 Changelog

5.3.1 0.0.1.dev10 (2020-09-23)

Changed

- Include metric type in attributes when sending metrics.

5.3.2 0.0.1.dev9 (2019-02-28)

Changed

- Deep merge user config file.

5.3.3 0.0.1.dev8 (2018-09-07)

Changed

- Remove mutation of the context passed to ffwd plugin.

5.3.4 0.0.1.dev7 (2018-06-21)

Fixed

- Add support for ulogger configuration.

5.3.5 0.0.1.dev6 (2018-06-20)

Fixed

- Fix routing for handling more than one message at a time.
- Improve warning log messages when loading plugin phase route.

5.3.6 0.0.1.dev5 (2018-06-18)

Fixed

- Provide router setup with correct number of arguments.

5.3.7 0.0.1.dev4 (2018-06-18)

Added

- Add logging-based default metrics plugin.
- Emit basic metrics from core router.
- Add a basic graceful shutdown mechanism.

5.3.8 0.0.1.dev3 (2018-06-14)

Added

- Add `IRunnable`, `IMessageHandler`.
- Add route configuration requirement.

Changed

- Require `IEventMessage` to have `phase` and `msg_id`.

Removed

- Remove `IEventConsumerClient`, `IEnricherClient`, `IPublisherClient`.

5.3.9 0.0.1.dev2 (2018-06-13)

Added

- Add logic to start installed + configured plugins.
- Add initial routing logic for event messages.
- Add interface definitions for a metrics plugin.
- Add FFWD-compatible metrics plugin.
- Enable plugin loader to load metrics plugin.

Fixed

- Load config only for active plugins.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

Symbols

A

C

D

F

G

H

I

L

M

P

R

S

T

U