Gestalt

Release 20.1.1

Jan 14, 2020

Contents

gestalt is a Python application framework for building distributed systems

CHAPTER 1

User Guide

This section of the documentation provides user focused information such as installing and quickly using this package.

1.1 Install Guide

Note: It is best practice to install run Python projects in a virtual environment, which can be created and activated as follows using Python 3.6+.

```
$ python -m venv venv
$ source venv/bin/activate
(venv) $ pip install gestalt
```

The simplest way to install Gestalt is using Pip.

```
$ pip install gestalt
```

This will install `gestalt` and all of its dependencies.

1.2 API Reference

The API Reference provides API-level documentation.

A record of significant changes can be found in the change log.

1.2.1 Change Log

Notable changes to this project will be documented in this file.

Version History

20.1.0

- Update package to support Python3.8
- Use yarl package for URLs.
- Add linting to improve code sustainment.
- Fix bug in stream protocols that affected msg_len in scenarios where messages were fragmented.

19.9.2

- Fix markdown displayed on PyPI.

19.9.1

- Adopted CalVer for package versioning.
- Initial functionality release.
- Clean up type annotation to remove all Mypy errors.
- Add Mypy type check to CI.
- Improve unit test code coverage.

0.2.0

- Added basic functionality.

0.0.1

- Project created.

1.3 Report Bugs

Report bugs at the issue tracker.

Please include:

- Operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

CHAPTER 2

Developers Guide

2.1 Contributing Guide

Contributions are welcome and greatly appreciated!

2.1.1 Workflow

A bug-fix or enhancement is delivered using a pull request. A good pull request should cover one bug-fix or enhancement feature. This ensures the change set is easier to review and less likely to need major re-work or even be rejected.

The workflow that developers typically use to fix a bug or add enhancements is as follows.

- Fork the `gestalt` repo into your account.
- Obtain the source by cloning it onto your development machine.

```
$ git clone git@github.com:your_name_here/gestalt.git
$ cd gestalt
```

- Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

  Now you can make your changes locally.

- Familiarize yourself with the developer convenience rules in the Makefile.

```
$ make help
```

- Create and activate a Python virtual environment for local development.

```
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $ pip install pip -U
(venv) $
```

Note: (venv) is used to indicate when the commands should be run within the virtual environment containing the development dependencies.

- Install the gestalt package as an editable install along with development extras and optional extras.

```
(venv) $ pip install -e .[develop,amq,protobuf,msgpack,avro,brotli,snappy,yaml]
```

- Develop fix or enhancement:
  - Make a fix or enhancement (e.g. modify a class, method, function, module, etc).
  - Update an existing unit test or create a new unit test module to verify the change works as expected.
  - Run the test suite.

    Some tests require optional dependencies. If the dependencies are not installed then tests requiring these dependencies are skipped.

    Some tests require RabbitMQ to be running with default user/password (guest/guest) and port (5672). The easiest way to get it running is to use Docker, such as:

    ```
    $ docker pull rabbitmq:3-management
    $ docker run -d -p 5672:5672 -p 15672:15672 rabbitmq:3-management
    ```

    Then execute the test suite.

    ```
    (venv) $ make test
    ```

    See the *Testing* section for more information on testing.
  - Check code coverage of the area of code being modified.

    ```
    (venv) $ make check-coverage
    ```

    Review the output produced in `docs/source/_static/coverage/index.html`. Add additional test steps, where practical, to improve coverage.
  - The change should be style compliant. Apply style formatter.

    ```
    (venv) $ make style
    ```

    See the *Code Style* section for more information.
  - The change should pass pylint static checking.

    ```
    (venv) $ make check-lint
    ```
  - The change should include type annotations where appropriate. Perform type annotations check.

    ```
    (venv) $ make check-types
    ```

    See the *Type Annotations* section for more information.
  - Fix any errors or regressions.

- The docs and the change log should be updated for anything but trivial bug fixes. Perform docs check.

```
(venv) $ make docs
```

  See the *Documentation* section for more information.

- Commit and push changes to your fork.

```
$ git add .
$ git commit -m "A detailed description of the changes."
$ git push origin name-of-your-bugfix-or-feature
```

  A pull request should preferably only have one commit upon the current master HEAD, (via rebases and squash).

- Submit a pull request through the service website (i.e. Github).
- Check automated continuous integration steps all pass. Fix any problems if necessary and update the pull request.

2.2 Testing

The Gestalt project implements a regression test suite that assists applying enhancements by identifying capability regressions early.

Developers implementing fixes or enhancements should run the tests to ensure they have not broken existing functionality. The Gestalt project provides some convenience tools so this testing step can be performed quickly.

2.2.1 Test Support

Some of the tests require an instance of RabbitMQ to be running. The easiest way to do this is using Docker. Pull the current version of the RabbitMQ container image.

During development we may want to observe the internal state of the RabbitMQ service so we'll run a RabbitMQ container that includes the management plugin. The RabbitMQ container listens on the default port of 5672 and the management service is available on the standard management port of 15672. The default username and password for the management plugin is guest / guest.

```
$ docker pull rabbitmq:3-management
```

Then run the RabbitMQ service.

```
$ docker run -d --hostname my-rabbit --name rabbitmq -p 5672:5672 -p 15672:15672␣
↪rabbitmq:3-management
```

Once the RabbitMQ service is running we can run the integration tests. You can access the RabbitMQ management interface by visiting http://<container-ip>:15672 in a browser and log in using the default username and password of guest, guest.

2.2.2 Run Tests

Use the Makefile convenience rules to run the tests.

```
(venv) $ make test
```

To run tests verbosely use:

```
(venv) $ make test-verbose
```

Alternatively, you may want to run the tests suite directly. The following steps assume you are running in a virtual environment in which the `gestalt` package has been installed (such as with `pip install -e .`). This avoids needing to explicitly set the `PYTHONPATH` environment variable so that the `gestalt` package can be found.

```
(venv) $ cd tests
(venv) $ python -m unittest
```

Individual unit tests can be run also.

```
(venv) $ python -m test_version.VersionTestCase.test_version
```

2.3 Coverage

The `coverage` tool can be run to collect code test coverage metrics.

Use the Makefile convenience rule to run the tests.

```
(venv) $ make check-coverage
```

The test code coverage report can be found here

2.4 Code Style

Adopting a consistent code style assists with maintenance. This project uses the code style formatter called Black. A Makefile convenience rule to enforce code style compliance is available.

```
(venv) $ make style
```

To simply check if the style formatting would make any changes use the `style.check` rule.

```
(venv) $ make check-style
```

2.5 Type Annotations

The code base contains type annotations to provide helpful type information that can improve code maintenance.

Use the Makefile convenience rule to check no issues are reported.

```
(venv) $ make check-types
```

2.6 Documentation

To rebuild this project's documentation, developers should use the Makefile in the top level directory. It performs a number of steps to create a new set of sphinx html content.

```
(venv) $ make docs
```

To quickly check consistency of ReStructuredText files use the dummy run which does not actually generate HTML content.

```
(venv) $ make check-docs
```

To quickly view the HTML rendered docs, start a simple web server and open a browser to http://127.0.0.1:8000/.

```
(venv) $ make serve-docs
```

2.7 Release Process

The following steps are used to make a new software release.

The steps assume they are executed from within a development virtual environment.

- Check that the package version label in `__init__.py` is correct.
- Create and push a repo tag to Github. As a convention use the package version number (e.g. YY.MM.MICRO) as the tag.

```
$ git checkout master
$ git tag YY.MM.MICRO -m "A meaningful release tag comment"
$ git tag  # check release tag is in list
$ git push --tags origin master
```

  - This will trigger Github to create a release at:

```
https://github.com/{username}/gestalt/releases/{tag}
```

- Create the release distribution. This project produces an artefact called a pure Python wheel. The wheel file will be created in the `dist` directory.

```
(venv) $ make dist
```

- Test the release distribution. This involves creating a virtual environment, installing the distribution into it and running project tests against the installed distribution. These steps have been captured for convenience in a Makefile rule.

```
(venv) $ make dist-test
```

- Upload the release to PyPI using

```
(venv) $ make dist-upload
```

  The package should now be available at https://pypi.org/project/gestalt/

CHAPTER 3

Quick Start

Gestalt is available on PyPI and can be installed with pip.

```
$ pip install gestalt
```

After installing Gestalt you can use it like any other Python module.

Here is a simple example:

```
import gestalt
# Fill this section in with the common use-case.
```