Ether documentation

version 0.4.0

Islam Amer

Ed Bartosh

Anton Beresin

March 10, 2011

Contents

Ether's documentation!

Overview

This project implements commit/post-receive hooks for various version control systems.

We use AMQP for handling the events. The format of messages is described in *Format of AMQP Messages*.

System Architecture

Format of AMQP Messages

The format of AMQP messages is loosely based on github payload spec.

Routing Key

Payload

Modules and classes

Publishers

AMQP

AMQP sender API.

class ether.publishers.amqp.**BasicAMQPPublisher** (*config*)
Base abstract class for AMQP publishers.

send_payload (*payload*)
Send payload to the server. Abstract method. Has to be implemented in derived classes.

Parameters: **payload** (*dictionary*) -- data to be sent

class ether.publishers.amqp.**BlockingAMQPPublisher** (*config*)
Blocking (synchronous) publisher. Code is borrowed from Pika Blocking demo_send example_blocking:

send_payload (*payload*)
Send payload to the server using blocking approach.

Parameters: **payload** (*dictionary*) -- data to be sent

class ether.publishers.amqp.**AsyncAMQPPublisher** (*config*)
Asynchronous Publisher. Code is borrowed from Pika Asynchronous demo_send example_async: Methods are placed in the same order as they're called by pika

on_channel_open (*channel*)
Callback. Called when channel has opened.

Parameters: **channel** (*object*) -- channel object

on_connected (*connection*)
Callback. Called when we are fully connected to RabbitMQ.

Parameters: **connection** (*object*) -- connection object

send_payload (*payload*)
Send payload to the server setting up chain of callbacks: on_connected -> on_channel_open -> on_queue_declared. (see above)

Parameters: **payload** (*dictionary*) -- data to be sent

log

class ether.publishers.log.**FileLogger** (*filename*)

send_payload (*payload*)

Consumers

AMQP

AMQP Listener.

class ether.consumers.amqp.**BaseAMQPConsumer** (*config*)
Base abstract class for AMQP consumers.

process_payload (*payload*, *routing_key=None*)
Process payload. Abstract method. Has to be implemented in derived classes.

Parameters:
- **payload** -- received payload
- **routing_key** (*string*) -- AMQP routing key

Returns: result code

Return type: int

receive_payload (*channel*, *method*, *header*, *body*)
Recieve payload from the server. Abstract method. Has to be implemented in derived classes.

Parameters: **body** (*dictionary*) -- data received

class ether.consumers.amqp.**AsyncAMQPConsumer** (*config*)
Asynchronous consumer.

consume ()
Start the IO event loop so we can communicate with RabbitMQ.

on_channel_open (*channel*)
Step #3: Called when our channel has opened.

Parameters: **channel** (*object*) -- channel object

on_connected (*connection*)
Step #2: Called when we are fully connected to RabbitMQ.

Parameters: **connection** (*object*) -- connection object

on_queue_bound (*frame*)
Step #6: Called when the queue has been bound to the exchange.

Parameters: **_frame** (*object*) -- response from broker

on_queue_declared (*frame*)
Step #5: Called when Queue has been declared.
frame is the response from RabbitMQ

setup_connection ()
Step #1: Connect to RabbitMQ.

Hooks

SVN hooks APIs.

class ether.hooks.svn.**SvnHook** (*sender*, *config*)
SVN hooks API. Gets hook data, creates hook payload and calls sender.send_payload.

postcommit (*repos*, *rev*)
Postcommit hook.

exception ether.hooks.svn.**SvnHookError**
Custom exception.

ether.hooks.svn.**get_repo_url** (*paths*, *config*)
Get svn repo url from the list of paths. :param paths: list of changed paths :ptype paths: list :param config: configuration object which maps repository paths to urls :ptype config: list of tuples (<path>, <url>) :returns: repository url :rtype: string

GIT hooks APIs.

Loosely modelled after the excellent contrib hook /usr/share/doc/git/contrib/hooks/post-receive-email

class ether.hooks.git.**GitHook** (*sender*)
GIT hook.

postreceive (*commits*)
Postcommit hook.

Indices and tables

- *genindex*
- *modindex*
- *search*

Development Practices

General Ideas

Code in this project should satisfy following requirements:

- PEP8 compliant
- Covered with testcases not less than 90%
- Covered with documentation
- Reviewed

Code Quality

We use pylint with default settings to check the code compliance:

```
pylint <your-file>
```

Code rated < 9 is **NOT** acceptable.

Make sure you run unit tests before committing any changes.

Make sure that all the test cases pass and code coverage is more than 90%.

If you fix a defect, create a test case to avoid regressions in future.

Documenting Your Code

For documenting we use sphinx. You can find project documentation in docs/ directory of the project.

Here are recomendations of documenting API:

If you document your functions this way:

```
def foo(param1, param2):
    """Description of foo.

    :param param1: param1 description
    :type param1: string

    :param param2: param2 description
    :type param2: list

    :returns: tuple (<category name>, <list of items>)
    :rtype: tuple

    """

    # some code here

    return (category, items)
```

you'll have this nice documentation generated by *sphinx*:

docfunc.**foo** (*param1*, *param2*)
Description of foo.

Parameters:
- **param1** (*string*) -- param1 description
- **param2** (*list*) -- param2 description

Returns: tuple (<category name>, <list of items>)

Return type: tuple

The same is for classes:

```
class Foo(object):
    """Class description."""

    #: class attribute description.
    classattr = None

    def method(self, param1, param2):
        """
        Method description.
        Could take several lines.

        :param param1: param1 description
        :type param1: string
        :param param2: param2 description
        :type param2: list

        """

        pass
```

class docclass.**Foo**
Class description.

classattr
class attribute description.

method (*param1*, *param2*)
Method description. Could take several lines.

Parameters:
- **param1** (*string*) -- param1 description
- **param2** (*list*) -- param2 description

Some useful links related to sphinx:

- http://docutils.sourceforge.net/docs/user/rst/quickref.html
- http://sphinx.pocoo.org/markup/code.html
- http://sphinx.pocoo.org/ext/autodoc.html
- http://packages.python.org/an_example_pypi_project/sphinx.html

Branching Policy

Recommended way is to have separate git branch for each task. After code is ready branch can be rebased against master and provided for review.

Please note that **no code should go to master without review**. It's reviewer task to review the code, disscuss it with developer, then merge it to master and tag it if needed.

Building Debian Package

To build packages, use:

```
git-buildpackage -rfakeroot -uc -us -sa -D
```

Build Debian package is a recommended way to check your work, because project documentation is re-generated and unit tests are run during package building.

Indices and tables

- *genindex*
- *modindex*
- *search*

Index

Python Module Index

d

docclass

e

ether

ether.consumers.amqp

ether.hooks.git

ether.hooks.svn

ether.publishers.amqp

ether.publishers.log