Wishbone Documentation

Release 0.5.0

Jelle Smet

August 28, 2014
# Contents

1 Creating a server in Python  
2 Bootstrapping server from CLI  
   2.1 Installation  
   2.2 Introduction  
   2.3 Wishbone modules  
   2.4 Router  
   2.5 Builtin Modules  
   2.6 Bootstraping on CLI  
   2.7 Patterns and best practices  
   2.8 Components  
3 Indices and tables
https://github.com/smetj/wishbone

A Python library and CLI tool to build and manage event pipeline servers with minimal effort.
Creating a server in Python

>>> from wishbone.router import Default
>>> from wishbone.module import TestEvent
>>> from wishbone.module import RoundRobin
>>> from wishbone.module import STDOUT

>>> router = Default()
>>> router.registerModule(TestEvent, "input", interval=1)
>>> router.registerModule(RoundRobin, "mixing")
>>> router.registerModule(STDOUT, "output1", prefix="I am number one: ")
>>> router.registerModule(STDOUT, "output2", prefix="I am number two: ")

>>> router.connect("input.outbox", "mixing.inbox")
>>> router.connect("mixing.one", "output1.inbox")
>>> router.connect("mixing.two", "output2.inbox")

>>> router.start()
>>> router.block()
I am number one: test
I am number two: test
I am number one: test
I am number two: test
I am number one: test
I am number two: test
I am number one: test
I am number two: test
I am number one: test
I am number two: test
In this example we initialize wishbone.router.Default to create a simple setup in which we connect wishbone.module.TestEvent, which does nothing more than generating the word “test” every second, to wishbone.module.RoundRobin which on its turn “roundrobins” the incoming events to two wishbone.module.STDOUT instances which print all incoming events to STDOUT.
Bootstrapping server from CLI

Wishbone comes with a CLI tool to bootstrap servers using a YAML formatted config file. The bootstrap file describes the modules to initialize and how the modules should be connected to each other.

Following bootstrap file creates exactly the same setup as shown in the above example:

```yaml
---
modules:
  input:
    module: wishbone.input.testevent
  mixing:
    module: wishbone.flow.roundrobin
  output1:
    module: wishbone.output.stdout
    arguments:
      prefix: "I am number one: "
  output2:
    module: wishbone.output.stdout
    arguments:
      prefix: "I am number two: "
routingtable:
  - input.outbox -> mixing.inbox
  - mixing.one -> output1.inbox
  - mixing.two -> output2.inbox
...```

Bootstrapping the environment is just a matter of invoking the `wishbone` executable using the `--config` parameter pointing to the bootstrap file.

```
[smetj@indigo ~]$ wishbone debug --config simple.yaml --id docker
2014-08-06T23:17:41 wishbone[6609]: debug metrics_funnel: Started with max queue size of 100 events and metrics interval of 1 seconds.
2014-08-06T23:17:41 wishbone[6609]: debug logs_funnel: Started with max queue size of 100 events and metrics interval of 1 seconds.
2014-08-06T23:17:41 wishbone[6609]: debug mixing: Started with max queue size of 100 events and metrics interval of 1 seconds.
2014-08-06T23:17:41 wishbone[6609]: debug input: Started with max queue size of 100 events and metrics interval of 1 seconds.
2014-08-06T23:17:41 wishbone[6609]: debug output1: Started with max queue size of 100 events and metrics interval of 1 seconds.
2014-08-06T23:17:41 wishbone[6609]: debug output2: Started with max queue size of 100 events and metrics interval of 1 seconds.
```
2014-08-06T23:17:41 wishbone[6609]: debug log_format: Started with max queue size of 100 events and metrics interval of 1 seconds.
2014-08-06T23:17:41 wishbone[6609]: debug syslog: Started with max queue size of 100 events and metrics interval of 1 seconds.
2014-08-06T23:17:41 wishbone[6609]: debug syslog: preHook() found, executing
I am number one: test
I am number two: test
I am number one: test
I am number two: test
I am number one: test
I am number two: test
I am number one: test
I am number two: test
2014-08-06T23:18:09 wishbone[6609]: debug syslog: postHook() found, executing
2014-08-06T23:18:09 wishbone[6609]: debug syslog: postHook() found, executing
2014-08-06T23:18:09 wishbone[6609]: informational input: Stopped producing events.
[smetj@indigo ~]$}

Contents:

2.1 Installation

Wishbone works on python 2.7+ and PyPy 2.3.1+

2.1.1 Versioning

- Wishbone uses Semantic Versioning.
- Each release is tagged in Github with the release number.
- The master branch contains the latest stable release.
- The development branch is where all development is done. There is no the code works.
- Other branches are special test cases or experiments.

2.1.2 Wishbone

Pypi

To install the latest stable release from https://pypi.python.org/pypi/wishbone use pip.

$ pip install wishbone

From source

Wishbone’ source can be downloaded from http://github.com/smetj/wishbone

Stable

Install the latest stable release from the master branch.
$ git clone https://github.com/smetj/wishbone
$ cd wishbone
$ cd checkout master #just in case your repo is in another branch
$ sudo python setup.py install

### Development

Install the latest development release from the development branch.

$ git clone https://github.com/smetj/wishbone.git
$ cd wishbone
$ git checkout develop
$ sudo python setup.py install

### Docker

Wishbone is also available as a Docker container.

Pull the smetj/wishbone repository from https://registry.hub.docker.com/u/smetj/wishbone into your Docker environment:

$ docker pull smetj/wishbone
$ docker images

<table>
<thead>
<tr>
<th>REPOSITORY</th>
<th>TAG</th>
<th>IMAGE ID</th>
<th>CREATED</th>
<th>VIRTUAL SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>smetj/wishbone</td>
<td>latest</td>
<td>e4acd2360be8</td>
<td>40 minutes ago</td>
<td>932.3 MB</td>
</tr>
<tr>
<td>smetj/wishbone</td>
<td>alertmachine</td>
<td>9bb81f6baa3a</td>
<td>4 months ago</td>
<td>756.2 MB</td>
</tr>
</tbody>
</table>

The container is prepared to be run as an executable:

$ docker run -i -t smetj/wishbone:latest

Usage: wishbone [-h] {show,stop,list,start,kill,debug} ...

wishbone: error: too few arguments

The following commands runs a Wishbone container:

$ docker run --privileged=true -t -i --volume /bootstrap:/bootstrap smetj/wishbone:1.0.0 debug --config /bootstrap/simple.yaml

The idea is that the Docker host has a directory called “/bootstrap” which contains all the Wishbone bootstrap files. The above command mounts the host’s /bootstrap directory to the container’s mountpoint called “/bootstrap”. Once done you can point the --config parameter to the mountpoint and load the bootstrap files stored on the host.

### Verify installation

Once installed you should have the wishbone executable available in your search path:

$ wishbone --help

Usage: wishbone [-h] {start,debug,stop,kill,list,show} ...

Wishbone bootstrap server.

Positional arguments:

{start,debug,stop,kill,list,show}

Optional arguments:

-h, --help show this help message and exit

2.1. Installation 7
2.2 Introduction

Wishbone is a Python library to create IO driven event processing servers by defining a pipeline of inputs and outputs with a number of intermediate processing stages in between through which events travel.

Wishbone comes with the necessary tools and modules to bootstrap servers from CLI and have them running as a permanent solution in a minimum of time.

2.2.1 Modules and Queues

Modules are isolated blocks of code (greenlets) each with their own specific functionality. They are created by inheriting wishbone.Actor as a baseclass. Modules cannot (and are not supposed to) directly invoke each others functionality. Their only means of interaction is by passing events to each other’s wishbone.Queue queues. Modules typically have, but are not limited to, an inbox, outbox, successful and failed queue.

2.2.2 Router

The wishbone.router.Default plays an important role. It’s job is to hold the initialized module instances and to organize the queue connections between the different modules.

Modules are registered using wishbone.router.Default.registerModule(). The router takes care of the proper startup wishbone.router.Default.start() and shutdown wishbone.router.Default.stop() sequence of the registered modules.

The router automatically connects each module’s metrics and logs queue to a wishbone.module.Funnel instance for your convenience. This allows the user to further organize log and metric processing by connecting other modules to one of these instances.

Queues are connected to each other using wishbone.router.Default.connect(). Queues can only have a “1 to 1” relationship. If you require a “1 to N” you will have to use one of the builtin flow modules.

2.2.3 Events

Wishbone events are simple data structures:

```
{ "header":{}, "data": object }
```

Input modules are responsible for creating events using this format.

2.2.4 Gevent

Wishbone uses Gevent. The modules run as cooperatively scheduled greenlets while taking advantage of the cooperative socket support for network IO. This makes Wishbone servers cope well with IO intensive tasks.

2.3 Wishbone modules

2.3.1 Introduction

Modules are isolated blocks of code (greenlets) each with their own specific functionality. They are created by inheriting wishbone.Actor as a baseclass. Modules cannot (and are not supposed to) directly invoke each others
functionality. Their only means of interaction is by passing events to each other’s `wishbone.Queue` queues. Modules typically have, but are not limited to, an `inbox`, `outbox`, `successful` and `failed` queue.

A module’s queues always live inside `wishbone.QueuePool` which, besides offering some convenience functions, is nothing more than a container to centralize all the module’s queues. Typically, modules consume the events entering the `inbox` queue and apply to them to some pre-registered method. Registering a method to consume all events of a queue is done using `wishbone.Actor.registerConsumer`. The registered method is then responsible to submit the event to another queue of choice, typically but not necessarily `outbox`.

### 2.3.2 Module categories

Modules are stored into a hierarchical name space. The name of a module consists out of:

```
<category name> . <group name> . <module name>
```

Wishbone comes with a set of builtin modules which are an integral part of the Wishbone framework.

External modules are installed as regular Python modules and should create an entrypoint in the `wishbone.contrib` namespace.

[https://github.com/smetj/wishboneModules](https://github.com/smetj/wishboneModules) is a repository containing additional modules.

You can list all available modules using the `list` command:

```
$ wishbone list
```

```
+----------+----------+----------------+---------+-------------------------------------------------------------+
| Category | Group    | Module          | Version | Description                                                 |
+----------+----------+----------------+---------+-------------------------------------------------------------+
| wishbone | flow     | funnel          | 0.5.0   | Funnel multiple incoming queues to 1 outgoing queue.         |
|          |          | fanout          | 0.5.0   | Funnel multiple incoming queues to 1 outgoing queue.         |
|          |          | roundrobin      | 0.5.0   | Round-robins incoming events to all connected queues.       |
| encode   | humanlogformat |              | 0.5.0   | Formats Wishbone log events.                                |
|          | msgpack   |              | 0.5.0   | Encodes events to MSGPack format.                           |
|          | graphite  |              | 0.5.0   | Converts the internal metric format to Graphite format.     |
| decode   | msgpack   |              | 0.5.0   | Decodes events from MSGPack format.                         |
| function | header    |              | 0.5.0   | Adds information to event headers.                          |
| input    | amqp      |              | 0.5.0   | Consumes messages from AMQP.                                |
|          | testevent |              | 0.5.0   | Generates a test event at the chosen interval.               |
|          | tcp       |              | 0.5.0   | A Wishbone input module which listens on a TCP socket.      |
|          | subscriber|              | 0.5.0   | Subscribes to one or more ZeroMQ publishers.                |
|          | dictgenerator |            | 0.5.0   | Generates random dictionaries.                              |
|          | disk      |              | 0.5.0   | Reads messages from a disk buffer.                          |
| output   | publisher |              | 0.5.0   | Publishes data to one or more ZeroMQ receivers.             |
+----------+----------+----------------+---------+-------------------------------------------------------------+

Build event pipeline servers with minimal effort.
To read the help and module instructions use the **show** command:

```bash
$ wishbone show --module wishbone.input.testevent
```

---

Build event pipeline servers with minimal effort.

---

### wishbone.input.testevent

**Version: 0.5.0**

Generates a test event at the chosen interval.

---

Events have following format:

```json
{ "header":{}, "data":"test" }
```

**Parameters:**

- **name**(str)
  
  The name of the module.

- **size**(int)
  
  The default max length of each queue.

- **frequency**(int)
  
  The frequency in seconds to generate metrics.

- **interval**(float)(1)
  
  The interval in seconds between each generated event.
  
  A value of 0 means as fast as possible.

- **message**(string)("test")
  
  The content of the test message.

- **numbered**(bool)
  
  When true, appends a sequential number to the end.
2.3.3 Module groups

Wishbone modules are divided into 6 group types depending on their functionality:

**input modules**

Input modules take input from the outside into the Wishbone framework. They are responsible of accepting data and converting that data into the appropriate Wishbone data format. Input modules typically have a `wishbone.Queue` named `output`.

**output modules**

Output modules are responsible for submitting Wishbone event data to the outside world. Output modules typically have a `wishbone.Queue` named `input`.

**flow modules**

Flow modules do not change data but they decide on the flow of events in the pipeline.

**function modules**

Function modules can have a wide range of functionalities but they take events in, change them and send events out. Function modules have at least 1 incoming and 1 outgoing queue.

**encode modules**

Encode a data format into another or into the internal metric/log format.

**decode modules**

Decode a data format into another or into the internal metric/log format.

2.3.4 Important properties and behavior

**successful and failed queues**

Each module has a `successful` and `failed` queue. Whenever a registered method (see `wishbone.Actor.registerConsumer`) fails to process an event, the framework will submit the event into the `failed` queue. Therefore it is important not to trap exceptions in the registered consumer methods but rather rely upon the fact Wishbone will trap that exception and submits it to the `failed` queue from which it can be further processed by another module.

2.3. Wishbone modules
An example which takes advantage of this behavior might be connecting the failed queue of the wishbone.module.TCPOut module to the inbox queue of the wishbone.module.DiskOut module.

On the other side, each time a registered consumer method successfully processes an event, it will automatically be submitted to the successful queue, from where it can be further processed by another module when desired.

An example which takes advantage of this behavior might be connecting the successful queue of the wishbone.module.TCPOut module to the acknowledgment queue of the wishbone.module.AMQPOut module.

It’s up to the method which has been registered to consume a queue to submit the event to another queue such as outbox from where it can be routed to the next module for further processing.

**metrics and logs queues**

Each module has a metrics and logs queue which hold metric and log events respectively, ready to be consumed by another module. If these queues aren’t connected to other queues then that data will be dropped.

**Queues drop data by default**

When a queue is not connected it will drop all messages submitted to it. The moment a queue is connected to another queue, messages are kept and transported.

**Module default parameters**

The wishbone.Actor baseclass must be initialized with at least 3 parameters:

- **name**: The name of the module.
- **size**: The size of each of the module queues.
- **frequency**: The frequency at which metrics are generated.

**2.3.5 Events**

Events are very simple `<type 'dict'>` data structures which contain a **header** and a **data** key.

The **header** is again a `<type 'dict'>` while, **data** can be any type of object.

```python
    { "header":{}, "data": object }
```

**2.3.6 Example**

Consider the following example module which reverses the content of incoming events and optionally converts the first letter into a capital.

```python
from wishbone import Actor

class ReverseData(Actor):
    '''**Sample module which reverses incoming events.**

Parameters:

- name (str): The instance name.
```
- capitalize(bool): When True capitalizes the first letter. Default: True

Queues:
- inbox: Incoming events.
- outbox: Outgoing events.

```python
def __init__(self, name, size=100, frequency=1, capitalize=False):
    Actor.__init__(self, name, size, frequency)
    self.pool.createQueue("inbox")
    self.pool.createQueue("outbox")
    self.registerConsumer(self.consume, "inbox")

    self.capitalize = capitalize

def preHook(self):
    self.logging.info("Hello from preHook")

def postHook(self):
    self.logging.info("Goodbye from postHook")

def consume(self, event):
    data = event["data"]
    data = [::-1]

    if self.capitalize:
        data = data.title()
    event["data"] = data

    self.submit(event, self.queuepool.queue.outbox)
```

- The ReverseData class inherits the `wishbone.Actor` base class[4].
- The `wishbone.Actor` base class is initialized with name, size and frequency parameter [23].
- Two queues, inbox and outbox are created [24][25].
- The `consume` method [36] is registered to consume each event from the `inbox` queue using the `wishbone.Actor.registerConsumer` method[26].
- The `preHook` [30] method is executed before starting the registered consumer methods while the the `postHook` [33] method is executed before shutdown. They are invoked automatically by the Wishbone framework.
- Logging is done by simply invoking the appropriate `wishbone.Logging` functions.
- The registered consumer method is responsible for adding the (changed) event to the appropriate queue. [46]
## 2.4 Router

The `wishbone.router.Default` router is used to initialize the modules and to organize the event stream between them.

Modules are registered using `wishbone.router.Default.registerModule()`. The router takes care of the proper startup `wishbone.router.Default.start()` and shutdown `wishbone.router.Default.stop()` sequence of all the modules.

Queues are connected to each other using `wishbone.router.Default.connect()`. Queues can only have a “1 to 1” relationship. If you require a “1 to N” or similar scenario you might have to use one of the builtin flow modules.

The router also takes care of the logs and metrics produced by the modules. The `wishbone.router.Default` router automatically connects the `metrics` and `logs` queues of each module to a `wishbone.module.Funnel` module which can optionally be connected to another module for further handling.

### 2.4.1 Logging

Each module has an instance of `wishbone.Logging`. This class offers the methods to generate logging with the appropriate level.

When modules are initialized in a router then the router it will automatically connect the `logs` queue to a `wishbone.module.Funnel` instance called `logs_funnel`. This module receives the logs from all modules. It is up to the user to decide which other module(s) to connect to the `outbox` of the `logs_funnel` to further process the logs.

Logs events are tuples have following format:

(6, 1367682301.430527, 'Router', 'Received SIGINT. Shutting down.')

Typically these log events are send to the `wishbone.module.Syslog`.

### 2.4.2 Metrics

Each module processes metrics of its queues. Metrics are generated at the defined `<frequency>`.

When modules are initialized in a router then the router it will automatically connect the `metrics` queue to a `wishbone.module.Funnel` instance called `metrics_funnel`. This module receives the metrics from all modules. It is up to the user to decide which other module(s) to connect to the `outbox` of the `metrics_funnel` to further process the metrics.

For each queue we have following metrics:

- `dropped_rate`
- `dropped_total`
- `in_rate`
- `int_total`
- `out_rate`
- `out_total`
- `size`
Typically a `wishbone.module.Graphite` instance is connected to the `metrics_funnel` module which in conjunction with `wishbone.module.TCPOut` sends the metrics to Graphite.

Format

Wishbone represents metrics into a fixed data structure:

(time, type, source, name, value, unit, (tag1, tag2))

It is a tuple containing a number of fields:

- timestamp A timestamp of the metric in unix time.
- type A free to choose description of the type of the metric
- source The originating source of the metric
- name The name of the metric
- value The metric value
- unit The value units
- tags A tuple of tags

For example:

(1381002603.726132, ‘wishbone’, ‘hostname’, ‘queue.outbox.in_rate’, 0, '', (“production”,monitored))

2.5 Builtin Modules

2.5.1 Input modules

`wishbone.input.amqp`

`wishbone.input.dictgenerator`

`wishbone.input.disk`

`wishbone.input.httpclient`
Wishbone Documentation, Release 0.5.0

wishbone.input.httpserver

wishbone.input.namedpipe

wishbone.input.pull

wishbone.input.tcp

wishbone.input.testevent

wishbone.input.topic

wishbone.input.udp

2.5.2 Output modules

wishbone.output.amqp

wishbone.output.disk

wishbone.output.null

wishbone.output.topic

wishbone.output.stdout

wishbone.output.syslog

wishbone.output.tcp

Chapter 2. Bootstrapping server from CLI
2.5. Builtin Modules

wishbone.output.push

wishbone.output.udp

wishbone.output.UDSOut

2.5.3 Flow modules

wishbone.flow.funnel

wishbone.flow.fanout

wishbone.flow.roundrobin

2.5.4 Function modules

wishbone.function.header

wishbone.function.template

2.5.5 Encode modules

wishbone.encode.graphite

wishbone.encode.humanlogformat

wishbone.encode.msgpack

wishbone.encode.json

2.5.6 Decode modules

wishbone.decode.msgpack
Wishbone Documentation, Release 0.5.0

wishbone.decode.json

2.6 Bootstraping on CLI

Wishbone comes with a CLI tool to easily bootstrap a server using a YAML formatted config file. Following file creates exactly the same environment as the above example:

```yaml
---
modules:
  input:
    module: wishbone.input.testevent

  mixing:
    module: wishbone.flow.roundrobin

  output1:
    module: wishbone.output.stdout
    arguments:
      prefix: "I am number one: "

  output2:
    module: wishbone.output.stdout
    arguments:
      prefix: "I am number two: "

routingtable:
  - input.outbox -> mixing.inbox
  - mixing.one -> output1.inbox
  - mixing.two -> output2.inbox
```

Bootstrapping the environment is just a matter of invoking the `wishbone` executable with the `--config` parameter pointing to the bootstrap file.

2.6.1 Available commands

**start**

The `start` command detaches the Wishbone server from console and runs it in the background. This implies that logs are written to syslog unless specifically defined otherwise. Metrics are written to Null unless specifically defined otherwise.

The `pidfile` contains the pids of the control process and all child processes. When stopping a Wishbone instance make sure you point to the pid file used to start the Wishbone instance.

```
[smetj@indigo ~]$ wishbone start -h
```

```

Starts a Wishbone instance and detaches to the background. Logs are written to syslog.
```

optional arguments:
- `h`, `--help` show this help message and exit
- `--config CONFIG` The Wishbone bootstrap file to load.
--instances INSTANCES
   The number of parallel Wishbone instances to bootstrap.
--pid PID
   The pidfile to use.
--queue-size QUEUE_SIZE
   The queue size to use.
--frequency FREQUENCY
   The metric frequency.
--id IDENT
   An identification string.

**debug**

The debug command does pretty much the same as start just that it keeps the Wishbone instance in the foreground without detaching it. Logs are written to STDOUT. The running instance can be stopped gracefully with CTRL+C

```
[smetj@indigo ~]$ wishbone debug -h
usage: wishbone debug [-h] [--config CONFIG] [--instances INSTANCES]
   [--queue-size QUEUE_SIZE] [--frequency FREQUENCY]
   [--id IDENT]
Starts a Wishbone instance in foreground and writes logs to STDOUT.
```

optional arguments:
- $h, --help
  show this help message and exit
--config CONFIG
   The Wishbone bootstrap file to load.
--instances INSTANCES
   The number of parallel Wishbone instances to bootstrap.
--queue-size QUEUE_SIZE
   The queue size to use.
--frequency FREQUENCY
   The metric frequency.
--id IDENT
   An identification string.

**stop**

Stops the Wishbone instance gracefully by sending SIGINT to all processes.

```
smetj@indigo ~]$ wishbone stop -h
usage: wishbone stop [-h] [--pid PID]
```

Tries to gracefully stop the Wishbone instance.

optional arguments:
- $h, --help
  show this help message and exit
--pid PID
   The pidfile to use.
kill

** Use with caution, sends SIGKILL to the pids in the pidfile. **

[smetj@indigo ~]$ wishbone kill -h
usage: wishbone kill [-h] [--pid PID]

Kills the Wishbone processes immediately.

optional arguments:
- h, --help  show this help message and exit
--pid PID    The pidfile to use.

list

Lists all installed Wishbone modules, given that they have the correct entry-points.

[smetj@indigo ~]$ wishbone list

| .--.--.--|__.-----| |--| |--.-----.-----.-----.| | | | |__ --| | _ | _ | | -__| |________|__|_____|__|__|_____|_____|__|__|_____|

version 0.5.0

Build event pipeline servers with minimal effort.

Available modules:

<table>
<thead>
<tr>
<th>Category</th>
<th>Group</th>
<th>Module</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wishbone</td>
<td>flow</td>
<td>funnel</td>
<td>0.5.0</td>
<td>Funnel multiple incoming queues to 1 outgoing queue.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fanout</td>
<td>0.5.0</td>
<td>Funnel multiple incoming queues to 1 outgoing queue.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>roundrobin</td>
<td>0.5.0</td>
<td>Round-robins incoming events to all connected queues.</td>
</tr>
<tr>
<td>encode</td>
<td>humanlogformat</td>
<td>0.5.0</td>
<td>Formats Wishbone log events.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>msgpack</td>
<td>0.5.0</td>
<td>Encodes events to MSGPack format.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>graphite</td>
<td>0.5.0</td>
<td>Converts the internal metric format to Graphite format.</td>
<td></td>
</tr>
<tr>
<td>decode</td>
<td>msgpack</td>
<td>0.5.0</td>
<td>Decodes events from MSGPack format.</td>
<td></td>
</tr>
<tr>
<td>function</td>
<td>header</td>
<td>0.5.0</td>
<td>Adds information to event headers.</td>
<td></td>
</tr>
<tr>
<td>input</td>
<td>amqp</td>
<td>0.5.0</td>
<td>Consumes messages from AMQP.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>testevent</td>
<td>0.5.0</td>
<td>Generates a test event at the chosen interval.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tcp</td>
<td>0.5.0</td>
<td>A Wishbone input module which listens on a TCP socket.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>subscriber</td>
<td>0.5.0</td>
<td>Subscribes to one or more ZeroMQ publishers.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dictgenerator</td>
<td>0.5.0</td>
<td>Generates random dictionaries.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>disk</td>
<td>0.5.0</td>
<td>Reads messages from a disk buffer.</td>
<td></td>
</tr>
<tr>
<td>output</td>
<td>publisher</td>
<td>0.5.0</td>
<td>Publishes data to one or more ZeroMQ receivers.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>null</td>
<td>0.5.0</td>
<td>Purges incoming events.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>amqp</td>
<td>0.5.0</td>
<td>Produces messages to AMQP.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>stdout</td>
<td>0.5.0</td>
<td>Prints incoming events to STDOUT.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tcp</td>
<td>0.5.0</td>
<td>A Wishbone output module which writes data to a TCP socket.</td>
<td></td>
</tr>
</tbody>
</table>
show

Displays the docstring of the requested module.

[smetj@indigo ~]$ wishbone show --module wishbone.flow.fanout

Funnel multiple incoming queues to 1 outgoing queue.

Parameters:

- name(str)
  | The name of the module.

- size(int)
  | The default max length of each queue.

- frequency(int)
  | The frequency in seconds to generate metrics.

- dupe(bool)(False)
  | Determines whether we send references to the original event to all destination or an actual copy.

Queues:

  outbox
  | Outgoing events.
2.7 Patterns and best practices

This section discusses some common usage patterns and best practices. Although most if not all are strictly spoken not required, they might be helpful in building efficient Wishbone solutions.

2.7.1 Event headers

Write data to headers

In its bare minimum, an event has following layout:

```json
{ "header":{}, "data": object }
```

Whenever a module writes data into the header section of the event, it should store this under the `<self.name>` key, which is unique anyway within a router instance.

The `<self.name>` value of a module is determined when registering the module using `wishbone.router.Default.registerModule()`.

So when registering a module like this:

```python
router = Default(interval=1)
router.registerModule(STDOUT, "on_screen")
```

Then any information this module instance wants to write into the header should look like:

```json
{ "header":{"on_screen":{"one":1, "two":2}}, "data": object }
```

Retrieving data from headers

When a module requires specific information from the header which has been stored by another it should be possible to initiate the module using a parameter named “key”. You should not hard code the name of the header key into the module because if someone registers a module with another name, your code will not work anymore.

Consider following example module:

```python
class Output (Actor):
    def __init__(self, name, key=None):
        Actor.__init__(self, name)
        self.name=name
        if key == None:
            self.key=self.name
        else:
            self.key=key

    def consume(self, event):
        print(event["header"][self.key]["one"])
```

2.7.2 Writing output modules

Handle failed and successful events

Output modules are responsible of delivering messages to the outside world. Obviously, we want this to happen as reliable as possible. Whenever the function registered with `wishbone.Actor.registerConsumer` fails to
submit the event to the outside world and because of that raises an exception, then Wishbone will submit the event to the module’s **failed** queue.

On the contrary, whenever the function registered with `wishbone.Actor.registerConsumer` exits successfully the event is submitted to the module’s **successful** queue.

It is up the user to connect these queues to another queue in order come to the desired strategy. Whenever these queues remain unconnected, all messages submitted to them are discarded.

Some practical examples:

- After submitting an event successfully over TCP to the outside world, it is submitted to the **successful** queue. This queue is on its turn connected to the AMQP **acknowledge** queue to ascertain it is acknowledged from AMQP.

- After submitting an event over TCP failed, the event is submitted to the **failed** queue from where it is forwarded to another module which writes the event to disk.

## 2.8 Components

```python
class wishbone.Actor(name, size=100, frequency=1)
```

- **connect(source, instance, destination)**
  
  Connects the `<source>` queue to the `<destination>` queue. In fact, the source queue overwrites the destination queue.

- **flushQueuesToDisk()**
  
  Writes whatever event in the queue to disk for later retrieval.

- **getChildren(queue=None)**
  
  Returns the queue name `<queue>` is connected to.

- **loop()**
  
  The global lock for this module

- **readQueuesFromDisk()**
  
  Reads events from disk into the queue.

- **registerAdminCallback(function)**
  
  Registers `<function>` to respond to the incoming admin messages.

- **registerConsumer(function, queue)**
  
  Registers `<function>` to process all events in `<queue>`

  Do not trap errors. When `<function>` fails then the event will be submitted to the “failed” queue, If `<function>` succeeds to the success queue.

- **start()**
  
  Starts the module.

- **stop()**
  
  Stops the loop lock and waits until all registered consumers have exit.

- **submit(event, queue)**
  
  A convenience function which submits `<event>` to `<queue>` and deals with QueueFull and the module lock set to False.

```python
class wishbone.Queue(max_size=1)
```

A queue used to organize communication messaging between Wishbone Actors.
Parameters:

• **max_size (int): The max number of elements in the queue.** Default: 1

When a queue is created, it will drop all messages. This is by design. When `<disableFallThrough()>` is called, the queue will keep submitted messages. The motivation for this is that when is queue is not connected to any consumer it would just sit there filled and possibly blocking the chain.

The `<stats()>` function will reveal whether any events have disappeared via this queue.

clean()

Deletes the content of the queue.

dump()

Dumps the queue as a generator and cleans it when done.

empty()

Returns True when queue and unacknowledged is empty otherwise False.

get()

Gets an element from the queue.

size()

Returns the length of the queue.

stats()

Returns statistics of the queue.

waitUntilContent()

Blocks until at least 1 slot is taken.

waitUntilEmpty()

Blocks until the queue is completely empty.

waitUntilFree()

Blocks until at least 1 slot it free.

waitUntilFull()

Blocks until the queue is completely full.

class wishbone.QueuePool(size)

createQueue(name)

Creates a Queue.

getQueue(name)

Convenience function which returns the queue instance.

hasQueue(name)

Returns trie when queue with <name> exists.

join()

Blocks until all queues in the pool are empty.

listQueues(names=False, default=True)

returns the list of queue names from the queuepool.

class wishbone.Logging(name, q)

Generates Wishbone formatted log messages following the Syslog priority definition.

alert(message)

Generates a log message with priority alert(1).
**crit**(message)
Generates a log message with priority critical(2).

**critical**(message)
Generates a log message with priority critical(2).

**debug**(message)
Generates a log message with priority debug(7).

**emerg**(message)
Generates a log message with priority emergency(0).

**emergency**(message)
Generates a log message with priority emergency(0).

**err**(message)
Generates a log message with priority error(3).

**error**(message)
Generates a log message with priority error(3).

**info**(message)
Generates a log message with priority informational(6).

**informational**(message)
Generates a log message with priority informational(6).

**notice**(message)
Generates a log message with priority notice(5).

**warn**(message)
Generates a log message with priority warning(4).

**warning**(message)
Generates a log message with priority warning(4).
Indices and tables

- genindex
- search