Wishbone Documentation

Release 2.2.0

Jelle Smet

December 01, 2016

Contents

A Python framework to build event stream processing servers

https://github.com/smetj/wishbone

CHAPTER 1

What?

Wishbone is a Python framework for building servers to read, process and write infinite event streams by combining and connecting modules into a processing pipeline through which structured data flows, changes, triggers logic and interacts with external services.

Wishbone can be used to implement solutions for a wide spectrum of tasks from building mashup enablers and ETL servers to CEP and stream processing servers.

Wishbone comes with a set of useful builtin event and *lookup* modules with many more external modules available and ready to be used.

The goal of the project is to provide a simple and pleasant yet solid and flexible framework which provides the user a toolbox to be creative building custom solutions with minimal effort and development time.

CHAPTER 2

How?

Servers can be created directly in Python or by bootstrapping an instance using a YAML file directly from CLI.

> *The following "hello world" example creates a server which continuously prints "Hello world!" to STDOUT.*

For this we connect wishbone.input.testevent to wishbone.output.stdout:

2.1 In Python

```
from wishbone.module.testevent import TestEvent
from wishbone.module.stdout import STDOUT
from wishbone.router import Default
from wishbone.actor import ActorConfig

input_config = ActorConfig("input")
output_config = ActorConfig("output")

router = Default()
router.registerModule(TestEvent, input_config, {"message": "Hello world!"})
router.registerModule(STDOUT, output_config)
router.connectQueue("input.outbox", "output.inbox")
router.start()
try:
    router.block()
except KeyboardInterrupt:
    router.stop()
```

2.2 Using a bootstrap file

```
modules:
  input:
    module: wishbone.input.testevent
    arguments:
      message : Hello World!

  stdout:
    module: wishbone.output.stdout
```

```
routingtable:
  - input.outbox            -> stdout.inbox
```

The server can be started and stopped using the wishbone CLI:

```
$ wishbone debug --config hello_world.yaml
```

2.2.1 Installation

Wishbone works on python 2.7+ and PyPy 2.3.1+

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.

Python

To install the latest stable release from pypi you can use *pip*:

```
$ pip install wishbone
```

or use *easy_install*:

```
$ easy_install wishbone
```

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
```

Execute tests

```
$ python setup.py test
```

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
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
smetj/wishbone      development         e4acd2360be8        40 minutes ago      932.3 MB
smetj/wishbone      2.1.0               9bb81f6baa3a        4 months ago        756.2 MB
```

The container entrypoint is pointing to the wishbone 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 --volume ${PWD}/bootstrap.yaml:/tmp/bootstrap.yaml smetj/wishbone:2.1.0 debug --config /
```

The default containers don't contain bootstrap files. The idea is that you have to mount the bootstrap file into the container and refer to it using the **–config** argument.

Docker build file

This example build file creates a Wishbone Docker container:

```
FROM            centos:centos7
MAINTAINER      Jelle Smet
EXPOSE          19283
RUN             yum install -y wget automake autoconf make file libtool gcc  gcc-c++ python-dev bzip2
RUN             wget -qO- https://bitbucket.org/squeaky/portable-pypy/downloads/pypy-4.0.1-linux_x86_
RUN             wget -O /tmp/get-pip.py https://bootstrap.pypa.io/get-pip.py
RUN             /opt/pypy-4.0.1-linux_x86_64-portable/bin/pypy /tmp/get-pip.py
RUN             /opt/pypy-4.0.1-linux_x86_64-portable/bin/pip install cython
RUN             /opt/pypy-4.0.1-linux_x86_64-portable/bin/pip install --process-dependency-link https
ENTRYPOINT      ["/opt/pypy-4.0.1-linux_x86_64-portable/bin/wishbone"]
```

2.2.2 Event Modules

> *Event modules are building blocks which perform 1 clearly defined action to the events passing through them.*

Event modules are functionally isolated blocks of code which do not directly invoke each other's functionality. They only interact by passing *wishbone.event.Event* instances to each other's `wishbone.Queue` queues from where the incoming events are consumed and processed by a function registered by *wishbone.Actor.registerConsumer()*.

Event modules run as Gevent greenlets in the background and are created by inheriting *wishbone.Actor* as a baseclass and live inside a *wishbone.router.Default* instance.

Modules typically have, but are not limited to an **inbox**, **outbox**, **success** and **failed** queue.

A queue can only be connected to 1 single queue.

Note: If you need to have *"one to many"* or *"many to one"* connections then you can use the `wishbone.module.Fanout` and `wishbone.module.Fanout` modules.

The *wishbone.Actor* baseclass must be initialized by passing a `wishbone.actor.ActorConfig` instance which controls the behavior of the module instance.

class `wishbone.`**`Actor`**(*config*)

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

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

`loop`()
The global lock for this module

`metricProducer`()
A greenthread which collects the queue metrics at the defined interval.

`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.

`sendToBackground`(*function*, **args*, ***kwargs*)
Executes a function and sends it to the background.

Background tasks are usually running indefinately. When such a background task generates an error, it is automatically restarted and an error is logged.

`start`()
Starts the module.

`stop`()
Stops the loop lock and waits until all registered consumers have exit otherwise kills them.

`submit`(*event*, *queue*)
A convenience function which submits <event> to <queue>.

Warning: When a queue is not connected any message submitted to it will be dropped. This is by design to ensure queues do not fill up without ever being consumed.

Module types

Wishbone has 6 different module types builtin.

Input modules

Input modules read or accept data from the outside world into the Wishbone framework.

Features:

- Creates *wishbone.event.Event* instances from data.

- Place events into its **outbox** queue.

Output modules

Output modules write data to the outside world.

Features:

- *output* modules should have a *selection* parameter defaulting to *'@data'* which defines the part of the event which is actually submitted to the external service.
- Should inspect each incoming event whether it is of type `wishbone.event.Event` or `wishbone.event.Bulk` and handle bulk events accordingly.

Flow modules

Flow modules apply routing logic to passing messages by altering the destination queue of the event based on certain properties

Features:

- Do not alter events in transit.

Encode modules

Encode modules convert the event instance into the requested format

Features:

- Typically have an *inbox* and *outbox* queue.

Decode modules

Decode modules convert structured data format into the internal event respresentation

Features:

- Typically have an *inbox* and *outbox* queue.

Function modules

Function modules modify events during transit

Features:

- Typically have an *inbox* and *outbox* queue.

Builtin Modules

Input modules

wishbone.input.cron

class wishbone.module.cron.**Cron**(*actor_config*, *cron='*/10 * * * *'*, *payload='wishbone'*, *field='@data'*)

Generates an event at the defined time

Generates an event with the defined payload at the chosen time. Time is in crontab format.

Parameters

- **cron** (–) –

  The cron expression.

- **payload** (–) –

  The content of <field>.

- **field** (–) –

  The location to write <payload> to.

Queues:

•outbox

Outgoing messges

---

wishbone.input.dictgenerator

class wishbone.module.dictgenerator.**DictGenerator**(*actor_config*, *keys=[]*, *randomize_keys=True*, *num_values=False*, *num_values_min=0*, *num_values_max=1*, *min_elements=1*, *max_elements=1*, *interval=1*)

Generates random dictionaries.

This module allows you to generate an stream of dictionaries.

Parameters

- **keys** (–) –

  If provided, documents are created using the provided keys to which random values will be assigned.

- **randomize_keys** (–) –

  Randomizes the keys. Otherwise keys are sequential numbers.

- **num_values** (–) –

  If true values will be numeric and randomized.

- **num_values_max** (–) –

  The maximum of a value when they are numeric.

- **min_elements** (–) –

  The minimum number of elements per dictionary.

- **max_elements** (–) –

  The maximum number of elements per dictionary.

- **interval** (–) –

The time in seconds to sleep between each message.

Queues:

•**outbox**

Outgoing messges

---

wishbone.input.testevent

class wishbone.module.testevent.**TestEvent**(*actor_config*, *interval=1*, *message='test'*, *numbered=False*, *additional_values={}*)

Generates a test event at the chosen interval.

The data field of the test event contains the string "test".

Parameters

- **interval** (–) –

  The interval in seconds between each generated event.
  A value of 0 means as fast as possible.

- **message** (–) –

  The content of the test message.

- **numbered** (–) –

  When true, appends a sequential number to the end.

- **additional_values** (–) –

  A dictionary of key/value to add to the event.

Queues:

•**outbox**

Contains the generated events.

Output modules

wishbone.output.null

class wishbone.module.null.**Null**(*actor_config*)

Purges incoming events.

Purges incoming events.

Parameters:

n/a

Queues:

•**inbox**

incoming events

---

wishbone.output.stdout

class wishbone.module.stdout.**STDOUT**(*actor_config*, *selection='@data'*, *counter=False*, *prefix=''*, *pid=False*, *foreground_color='WHITE'*, *background_color='RESET'*, *color_style='NORMAL'*)

Prints incoming events to STDOUT.

Prints incoming events to STDOUT. When <complete> is True, the complete event including headers is printed to STDOUT.

You can optionally define the colors used.

Parameters

- **selection** (–) –

  The part of the event to submit externally.
  Use an empty string to refer to the complete event.

- **counter** (–) –

  Puts an incremental number for each event in front
  of each event.

- **prefix** (–) –

  Puts the prefix in front of each printed event.

- **pid** (–) –

  Includes the pid of the process producing the output.

- **foreground_color** (–) –

  The foreground color.
  Valid values: BLACK, RED, GREEN, YELLOW, BLUE, MAGENTA, CYAN, WHITE

- **background_color** (–) –

  The background color.
  Valid values: BLACK, RED, GREEN, YELLOW, BLUE, MAGENTA, CYAN, WHITE, RESET

- **color_style** (–) –

  The coloring style to use
  Valid values: DIM, NORMAL, BRIGHT

Queues:

•inbox

Incoming events.

---

wishbone.output.syslog

class wishbone.module.wbsyslog.**Syslog**(*actor_config*, *selection='@data'*, *level=5*, *ident='sphinx-build'*)

Writes log events to syslog.

Logevents have following format:

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

The first value corresponds to the syslog severity level.

Parameters

- **selection** (–) –

  The part of the event to submit externally.
  Use an empty string to refer to the complete event.

- **level** (–) –

  The loglevel.

- **ident** (–) –

  The syslog id string.
  If not provided the script name is used.

Queues:

•inbox

incoming events

Flow modules

wishbone.function.acknowledge
class wishbone.module.acknowledge.**Acknowledge**(*actor_config*, *ack_id=None*)

Lets events pass or not based on some event value present or not in a lookup table.

This module stores a value <ack_id> from passing events in a list and only let's events go through for which the <ack_id> value is not in the list.

<ack_id> can be removed from the list by sending the event into the <acknowledge> queue.

<ack_id> should some unique identifier to make sure that any following <modules are not processing events with the same datastructure.

Typically, downstream modules's <successful> and/or <failed> queues are sending events to the <acknowledge> queue.

Parameters **ack_id** (–) –

A value stored somewhere in the event which then acts as the ack_id. It possibly makes only sense to define an EventLookup value here.

Queues:

•inbox

Incoming events

•outbox

Outgoing events

•acknowledge

Acknowledge events

•dropped

Where events go to when unacknowledged

wishbone.function.deserialize

class wishbone.module.deserialize.**Deserialize**(*actor_config*, *source='@data'*, *destination='@data'*)

Deserializes Bulk events or arrays.

When incoming data is a Bulk object the content will be forwarded as single events again.

When the incoming data is a single and <source> is a list/array a new event is created from each element of the array.

Parameters

- **source** (–) –

  The source of the array.
  (Ignored when incoming type is Bulk)

- **destination** (–) –

  The destination key to store the array item.
  (Ignored when incoming type is Bulk)

Queues:

•inbox

Incoming messages

•outbox

Outgoing messges

---

wishbone.flow.fanout

class wishbone.module.fanout.**Fanout**(*actor_config*)

Forward each incoming message to all connected queues.

Forward each incoming message to all connected queues.

Parameters **n/a** –

Queues:

•inbox:

Incoming messages

---

wishbone.flow.fresh

class wishbone.module.fresh.**Fresh**(*actor_config*, *timeout_payload='timeout'*, *recovery_payload='recovery'*, *timeout=60*, *repeat_interval=60*)

Generates a new event unless an event came through in the last x time.

This module forwards events without modifying them. If an event has been forwarded it resets the timeout counter back to <timeout>. If the timeout counter reaches zero because no messages have passed through, an event with <timeout_payload> is generated and submitted to the module's <timeout> queue. When the a timeout_payload has been sent and the event stream recovers, a new event with <recovery_payload> is generated and submitted to the <timeout> queue.

Parameters

- **timeout_payload** (–) –

The data a timeout event contains.

- **recovery_payload** (–) –

  The data a recovery event contains

- **timeout** (–) –

  The max time in seconds allowed to not to receive events.

- **repeat_interval** (–) –

  The interval time to resend the <payload> event in case <timeout> has expired and

Queues:

•inbox

Incoming events.

•inbox

Incoming events.

•timeout

timeout and recovery events.

---

wishbone.flow.funnel

class wishbone.module.funnel.**Funnel**(*actor_config*)

Funnel multiple incoming queues to 1 outgoing queue.

Funnel multiple incoming queues to 1 outgoing queue.

Parameters **n/a** –

Queues:

•outbox:

Outgoing messages

---

wishbone.flow.roundrobin

class wishbone.module.roundrobin.**RoundRobin**(*actor_config*, *randomize=False*)

Round-robins incoming events to all connected queues.

Create a "1 to n" relationship between queues. Events arriving in inbox are then submitted in a roundrobin (or randomized) fashion to the connected queues. The outbox queue is non existent.

Parameters **randomize** (–) –

Randomizes the queue selection instead of going round-robin over all queues.

Queues:

•inbox

Incoming events

---

wishbone.flow.switch

class wishbone.module.switch.**Switch**(*actor_config*, *outgoing='outbox'*)

Switch outgoing queues while forwarding events.

Forwards events to the desired outgoing queue based on the value of <outgoing>.

The value of <outgoing> can be dynamically set in 2 ways:

- Using a lookup value.
- By sending an event to the <switch> queue with the value of <outgoing> stored under *@data*.

Parameters **outgoing** (–) –

The name of the queue to submit incoming events to.

Queues:

- **inbox**

  incoming events

- **switch**

  incoming events to alter outgoing queue.

- **outbox**

  outgoing events

- **<connected_queue_1>**

  outgoing events

- **<connected_queue_n>**

  outgoing events

---

wishbone.flow.tippingbucket

class wishbone.module.tippingbucket.**TippingBucket**(*actor_config*, *bucket_size=100*, *bucket_age=10*, *aggregation_key='default'*)

Aggregates multiple events into bulk.

Aggregates multiple incoming events into bulk usually prior to submitting to an output module.

Flushing the buffer can be done in various ways:

- The age of the bucket exceeds <bucket_age>.
- The size of the bucket reaches <bucket_size>.
- Any event arrives in queue <flush>.

Parameters

- **bucket_size** (–) –

  The maximum amount of events per bucket.

- **bucket_age** (–) –

  The maximum age in seconds before a bucket is closed and forwarded. This actually corresponds the time since the first

event was added to the bucket.

- **aggregation_key** (–) –

  Groups events with key <aggregation_key> into the same buckets.

Queues:

•inbox

Incoming events

•outbox

Outgoing bulk events

•flush

Flushes the buffer on incoming events despite the bulk being full (bucket_size) or expired (bucket_age).

---

wishbone.flow.ttl

class wishbone.module.ttl.**TTL**(*actor_config*, *ttl=1*)

Allows messages to pass a maximum number of times.

When a message has traveled through this module more than <ttl> times it will be submitted to the <ttl_exceeded> queue.

Parameters:

•ttl(int)(1)

The maximum number of times an event is allowed to travel through.

Queues:

•inbox

Incoming events.

•outbox

Outgoing events.

•ttl_exceeded

Events which passed the module more than <ttl> times.

Function modules

wishbone.function.modify

class wishbone.module.modify.**Modify**(*actor_config*, *expressions=[]*)

Modify and manipulate datastructures.

This module modifies the data of an event using a **sequential** list of expressions.

Expressions are dictionaries containing 1 item. The key is a string and the value is a list of parameters accepted by the expression.

For example:

```
{"set": ["hi", "@data.one"]}
```

Sets the value *"hi"* to key *"@data.one"*.

In the the YAML formatted bootstrap file that would look like:

```
module: wishbone.function.modify
arguments:
  expressions:
    - set: [ "hi", "@data.one" ]
```

Valid expressions are:

- **add_item**:

```
add_item: [<item>, <key>]
```

Adds <*item*> to the list stored under <*key*>.

- **copy**:

```
copy: [<source_key>, <destination_key>, <default_value>]
```

Copies <*source_key*> to <*destination_key*> and overwrites <*destination_key*> when it exists. If <source_key> does not exist, <default_value> is taken instead.

- **del_item**:

```
del_item: [<key>, <item>]
```

Deletes first occurance of <*item*> from the list stored under <*source_key*>.

- **delete**:

```
delete: [<key>]
```

Deletes <*key*> from the event.

- **extract**:

```
extract: [<destination>, <regex>, <source>]
```

Makes use of Python re module to extract named groups from <*source*> using <*regex*> and add the resulting matches to <*destination*>.

The following example would extract the words "one" and "two" from "@data.test" and add the to @data.extract:

expression:

```
extract: ["@data.extract", '(?P<first>.*?);(?P<second>.*)', "@data.test"]
```

result:

```
{"@data":{"test:"one;two", extract:{"first": "one", "second": "two"}}}
```

- **join**:

```
join: [<array>, <join>, <destination>]
```

Joins an array into a string using the <join> value.

- **lowercase**:

```
lowercase: [<key>]
```

Turns the string stored under <*key*> to lowercase.

•**merge**:

```
merge: [<object_one>, <object_two>, <destination>]
```

Merges 2 arrays into <destination>

•**replace**:

```
replace: [<regex>, <value>, <key>]
```

replaces every occurance of <regex> of the value stored in <key> with <value>

•**set**:

```
set: [<value>, <key>]
```

Sets <*value*> to the event <*key*>.

•**uppercase**:

```
uppercase: [<key>]
```

Turns the string stored under <*key*> to uppercase.

•**template**:

```
template: [<destination_key>, <template>, <source_key>]
```

Uses the dictionary stored in <*source_key*> to complete <*template*> and stores the results into key <*destination_key*>. The templating language used is Python's builtin string format one.

•**time**:

```
time: [<destination_key>, <format>]
```

Modifies the <*@timestamp*> value according the the <*format*> specification and stores it into <*destination_key*>. See http://crsmithdev.com/arrow/#format for the format.

Parameters **expressions** (–) –

A list of expressions to apply.

Queues:

•**inbox:**

Incoming messages

•**outbox:**

Outgoing modified messages

Encode modules

wishbone.encode.humanlogformat

class wishbone.module.humanlogformat.**HumanLogFormat**(*actor_config*, *colorize=True*, *ident=None*)

Converts the internal log format into human readable form.

Logs are formated from the internal wishbone format into a more pleasing human readable format suited for STDOUT or a logfile.

Internal Wishbone format:

(6, 1367682301.430527, 3342, 'Router', 'Received SIGINT. Shutting down.')

Sample output format:

2013-08-04T19:54:43 pid-3342 informational dictgenerator: Initiated 2013-08-04T19:54:43 pid-3342 informational metrics_null: Started

Parameters **n/a** –

Queues:

•inbox

Incoming messages

•outbox

Outgoing messges

---

wishbone.encode.json

class wishbone.module.jsonencode.**JSONEncode**(*actor_config*, *source='@data'*, *destination='@data'*)

Converts Python dict data structures to JSON strings.

Encodes Python data structures to JSON.

Parameters

- **source** (–) –

  The data to convert.

- **destination** (–) –

  The location to write the JSON string to.

Queues:

•inbox

Incoming messages

•outbox

Outgoing messges

Decode modules

wishbone.decode.json

class wishbone.module.jsondecode.**JSONDecode**(*actor_config*, *source='@data'*, *destination='@data'*, *str=True*)

Decodes JSON data to Python data objects.

Decodes the payload or complete events from JSON format.

Parameters

- **source** (–) –

  The source of the event to decode.
  Use an empty string to refer to the complete event.

- **destination** (–) –

  The destination key to store the Python <dict>.
  Use an empty string to refer to the complete event.

- **unicode** (–) –

  When True, converts strings to unicode otherwise regular string.

Queues:

•inbox

Incoming messages

•outbox

Outgoing messges

External Modules

There's a list of module which are developed and maintained outside the Wishbone framework itself.

These modules are released as seprate Python modules and can be installed by doing something similar as

```
$ pip install wishbone_input_amqp
```

Input modules

- wishbone_input_amqp
- wishbone_input_disk
- wishbone_input_gearman
- wishbone_input_httpclient
- wishbone_input_httpserver
- wishbone_input_irc
- wishbone_input_livestatus
- wishbone_input_namedpipe
- wishbone_input_tcp
- wishbone_input_twitter
- wishbone_input_udp
- wishbone_input_zmqpull
- wishbone_input_zmqtopic

Output modules

- wishbone_output_amqp
- wishbone_output_disk
- wishbone_output_elasticsearch
- wishbone_output_email
- wishbone_output_file
- wishbone_output_http
- wishbone_output_sse
- wishbone_output_tcp
- wishbone_output_udp
- wishbone_output_uds
- wishbone_output_zmqpush
- wishbone_output_zmqtopic

Function modules

- wishbone_function_template

Flow modules

- wishbone_flow_jq
- wishbone_flow_jsonvalidate
- wishbone_flow_match

Encode modules

- wishbone_encode_flatten
- wishbone_encode_graphite
- wishbone_encode_influxdb
- wishbone_encode_msgpack

Decode modules

- wishbone_decode_msgpack

Logs

Each event module has a `wishbone.Logging` instance called `self.logging` which can be used to generate logs. The user is responsible for connecting the necessary modules to collect and process the log events from each of the modules' `logs` queue.

```
Log({'message': 'Received stop. Initiating shutdown.', 'module': 'metrics_graphite', 'pid': 18179, '
```

class `wishbone.event.`**`Log`**(*time*, *level*, *pid*, *module*, *message*)
A Wishbone log object

Note: If you are bootstrapping a server from CLI then the `logs` queue of each module will be connected to `stdout` or `syslog` depending on the startup mode.

Warning: If no module is consuming the logs from the module's `logs` queue, adding new logs events to the full queue will cause the event to be dropped.

Metrics

Each Wishbone module collects statistics of its queues. These metric events are submitted to the modules' `metrics` queue. The user is responsible for connecting the necessary modules to collect and process the metric events from each of the modules' `metrics` queue.

Metrics are generated at the interval determined by the `wishbone.actor.ActorConfig` instance passed to the module.

class `wishbone.event.`**`Metric`**(*time*, *type*, *source*, *name*, *value*, *unit*, *tags*)
A Wishbone metric object

```
Metric({'tags': (), 'unit': '', 'value': 0, 'name': 'module.input.queue.failed.size', 'source': 'serv
```

2.2.3 Lookup modules

Lookup modules provide functions which return values used to initialize event modules with dynamic values as parameter values. Lookup modules not process events.

Lookup module functionality can be used when writing servers directly in Python but are especially useful when creating servers using bootstrap files. Wishbone modules are initialized using the variables defined in the bootstrap file. Instead of defining static values, you can define a lookup function which returns a value by executing some predefined function. See the UpLook library for more information. Lookup functions are defined and initialized in the lookups section of the bootstrap file.

Builtin lookup modules

Wishbone comes with a set of builtin lookup modules:

Builtin Modules

wishbone.lookup.choice

class `wishbone.lookup.choice.`**`Choice`**(*array*)
Returns a random element from the provided array.

This function returns a random element from the provided array.

- Parameters to initialize the function:
  - values(list)(None): An array of elements to choose from
- Parameters to call the function:

  None

---

wishbone.lookup.cycle

class wishbone.lookup.cycle.**Cycle**(*values*)

Cycles through the provided array returning the next element.

This function rotates through the elements in the provided array always returning the next element. The order is fixed and when the end is reached the first element is returned again.

- Parameters to initialize the function:
  - values(list)(None): An array of elements to cycle through/
- Parameters to call the function:

  None

---

wishbone.lookup.etcd

class wishbone.lookup.etcd.**ETCD**(*base='/v2/keys'*)

Returns a value from etcd.

Returns a value from an etcd instance.

- Parameters to initialize the function:
  - base(str)("/v2/keys"): The base part of the endpoint.
- Parameters to call the function:
  - key(str)(): The name of the key to request.

---

wishbone.lookup.event

class wishbone.lookup.event.**EventLookup**

Returns the requested event header value.

- Parameters to initialize the function:

  None
- Parameters to call the function:

When calling the function a variable reference can be used similar to:

~~headerlookup("modulename.header.variablename","unknown")

Keep in mind you always have to use a dynamic lookup function (double tilde). You can provide a default value in case <variablename> does not exist in the header of namespace <modulename>.

---

wishbone.lookup.pid

class wishbone.lookup.pid.**PID**

Returns the PID of the current process.

•Parameters to initialize the function:

None

•Parameters to call the function:

None

---

wishbone.lookup.random_bool

class wishbone.lookup.random_bool.**RandomBool**

Randomly returns True or False

•Parameters to initialize the function:

None

•Parameters to call the function:

None

---

wishbone.lookup.random_integer

class wishbone.lookup.random_integer.**RandomInteger**(*minimum=0*, *maximum=0*)

Returns a random integer.

Returns a random integer between <min> and <max>.

•Parameters to initialize the function:

–minimum(int)(0): The minimum value

–maximum(int)(0): The maximum value

•Parameters to call the function:

None

---

wishbone.lookup.random_uuid

class wishbone.lookup.random_uuid.**RandomUUID**

Returns a uuid value.

This function returns a uuid value.

•Parameters to initialize the function:

None

•Parameters to call the function:

None

---

wishbone.lookup.word

class wishbone.lookup.random_word.**RandomWord**(*filename=None*)

Returns a random word.

This function returns a random word from a wordlist. Each line is a new word.

•Parameters to initialize the function:

– **filename(str)(None): When value is None, a buildin wordlist is used.** If not a filename of a wordlist is expected.

•Parameters to call the function:

None

Importing modules

Lookup modules are referenced in the boostrap file using a dotted name which is the entrypoint for a class.

```
wishbone.lookup.randominteger
```

Lookup modules are classes

```
from random import randint

class RandomInteger(Lookup):

    def __init__(self, minimum=0, maximum=0):

        self.minimum = minimum
        self.maximum = maximum

    def lookup(self):

        return randint(self.minimum, self.maximum)
```

Writing a custom lookup function is really simple. You just have to provide a class with a *lookup* method which can (optionally) be called using the arguments values provided in the bootstrap file.

```
---
lookups:
  gimme_a_number:
    module: wishbone.lookup.random_integer
    arguments:
```

```
        minimum: 0
        maximum: 99

modules:
  input:
    module: wishbone.input.testevent
    arguments:
      message : ~~gimme_a_number()
      interval: 1

  stdout:
    module: wishbone.output.stdout

routingtable:
  - input.outbox            -> stdout.inbox
...
```

2.2.4 Events

wishbone.event.Event object instances are used to store and transport structured data between module queues.

The input modules should initialize a *wishbone.event.Event* instance to encapsulate the data they receive or generate.

wishbone.event.Event is a simple class used for data representation including some convenience functions for data manipulation.

Examples

```
>>> from wishbone.event import Event
>>> e = Event("hi")
>>> e.dump()
{'@timestamp': '2015-12-13T10:45:35.442088+01:00', '@version': 1, '@data': 'hi'}
>>>
```

Initializing the Event objects assigns the data you pass to *@data*.

```
>>> e = Event("hi")
>>> e.get()
'hi'
>>>
```

By default, the get method returns *@data*.

```
>>> e = Event({"one": {"two": hi}})
>>> e.get('@data.one.two')
'hi'
>>>
```

Nested dictionaries can be accessed in dotted format.

```
>>> e = Event('hi')
>>> e.set("howdy", "one.two.three")
>>> e.get('one')
{'two': {'three': 'howdy'}}
>>>
```

New 'root' keys can be created outside @data. Setting nested dictionary values can be done using dotted format.

```
>>> e = Event('hello')
>>> e.dump(complete=True)
{'@timestamp': '2015-12-13T11:10:45.862036+01:00', '@tmp': {},
'@version': 1, '@data': 'hello', '@errors': {}}
>>>>
```

- *@tmp* is where modules can optionally store temporary data which is not part of the data model but contain useful information for other modules.
- The *@errors* key is where modules store exceptions related information.

class `wishbone.event.`**Event**(*data=None*)

The Wishbone event object representation.

A class object containing the event data being passed from one Wishbone module to the other.

clone()

Returns a cloned version of the event using deepcopy.

copy(*source*, *destination*)

Copies the source key to the destination key.

Parameters

- **source** (*str*) – The name of the source key.
- **destination** (*str*) – The name of the destination key.

deepish_copy(*org*)

much, much faster than deepcopy, for a dict of the simple python types.

Blatantly ripped off from https://writeonly.wordpress.com/2009/05/07 /deepcopy-is-a-pig-for-simple-data/

delete(*key=None*)

Deletes a key.

Parameters **key** (*str*) – The name of the key to delete

dict_merge(*dct*, *merge_dct*)

Recursive dict merge. Inspired by :meth:`dict.update()`, instead of updating only top-level keys, dict_merge recurses down into dicts nested to an arbitrary depth, updating keys. The `merge_dct` is merged into `dct`.

Stolen from https://gist.github.com/angstwad/bf22d1822c38a92ec0a9

Parameters

- **dct** – dict onto which the merge is executed
- **merge_dct** – dct merged into dct

Returns None

dump(*complete=False*, *convert_timestamp=True*)

Dumps the content of the event.

Parameters

- **complete** (*bool*) – Determines whether to include @tmp and @errors.
- **convert_timestamp** (*bool*) – When True converts <Arrow> object to iso8601 string.

Returns The content of the event.

Return type dict

get (*key='@data'*)
Returns the value of <key>.

Parameters **key** (*str*) – The name of the key to read.

Returns The value of <key>

has (*key='@data'*)
Returns a boot indicating the event has <key>

Parameters **key** (*str*) – The name of the key to check

Returns Bool

raw (*complete=False*, *convert_timestamp=True*)
Dumps the content of the event.

Parameters

- **complete** (*bool*) – Determines whether to include @tmp and @errors.
- **convert_timestamp** (*bool*) – When True converts <Arrow> object to iso8601 string.

Returns The content of the event.

Return type dict

set (*value*, *key='@data'*)
Sets the value of <key>.

Parameters

- **value** – The value to set.
- **key** (*str*) – The name of the key to assign <value> to.

Bulk Events

By default events flow one by one from one module the other. *wishbone.event.Bulk* instances contain multiple *wishbone.event.Event* events and therefor can deliver multiple events at once to a module.

The modules receiving *wishbone.event.Bulk* events need to know how to process them. By convention, output modules should know how to deal with *Bulk* events.

The builtin module *wishbone.module.tippingbucket.TippingBucket* is an example of a module collecting multiple events into a Bulk event and forwarding it based on one or more conditions.

class wishbone.event.**Bulk** (*max_size=None*, *delimiter='n'*)

append (*event*)
Appends an event to the bulk object.

dump ()
Returns an iterator returning all contained events

dumpFieldAsList (*field='@data'*)
Returns a list containing a specific field of each stored event. Events with a missing field are skipped.

dumpFieldAsString (*field='@data'*)
Returns a string joining <field> of each event with <self.delimiter>. Events with a missing field are skipped.

size()
Returns the number of elements stored in the bulk.

2.2.5 Router

A router is responsible for initializing and holding all event modules, connecting their queues and organize the event flow between them.

The router object plays a central role in the Wishbone setup. When bootstrapping an instances using the CLI tooling you are not really exposed to it (though the routing table might give a hint), however if you create a server directly in Python you will always first have to initialize a `Router` instance:

```
input_config = ActorConfig("input")
output_config = ActorConfig("output")

router = Default()
router.registerModule(TestEvent, input_config, {"message": "Hello world!"})
router.registerModule(STDOUT, output_config)
router.connectQueue("input.outbox", "output.inbox")
router.start()
router.block()
```

Multiple routers can be initialized which run inside the same `process`.

```
from wishbone.actor import ActorConfig
from wishbone.router import Default
from wishbone.module.testevent import TestEvent
from wishbone.module.stdout import STDOUT

input_config = ActorConfig("input")
output_config = ActorConfig("output")

router_1 = Default()
router_1.registerModule(TestEvent, input_config, {"message": "Hello world from router instance 1!"})
router_1.registerModule(STDOUT, output_config)
router_1.connectQueue("input.outbox", "output.inbox")
router_1.start()

router_2 = Default()
router_2.registerModule(TestEvent, input_config, {"message": "Hello world from router instance 2!"})
router_2.registerModule(STDOUT, output_config)
router_2.connectQueue("input.outbox", "output.inbox")
router_2.start()

router_1.block()
router_2.block()
```

```
$ python 2_routers.py
Hello world from router instance 1!
Hello world from router instance 2!
Hello world from router instance 1!
Hello world from router instance 2!
Hello world from router instance 1!
Hello world from router instance 2!
```

Note: If desired, multiple `Router` instances can each run into their own `Process` using *gipc*. This is what

`wishbone.bootstrap.Dispatch` does.

Wishbone comes with the default *`wishbone.router.Default`* router implementation.

class `wishbone.router.`**Default**(*config=None*, *size=100*, *frequency=1*, *identification='wishbone'*, *graph=False*, *graph_include_sys=False*)

The default Wishbone router.

A Wishbone router is responsible for shoveling the messages from one module to the other.

Parameters

- **config** (*obj*) – The router setup configuration.
- **size** (*int*) – The size of all queues.
- **frequency** (*int) (1*) – The frequency at which metrics are produced.
- **identification** (*wishbone*) – A string identifying this instance in logging.

block()

Blocks until stop() is called and the shutdown process ended.

connectQueue(*source*, *destination*)

Connects one queue to the other.

For convenience, the syntax of the queues is <modulename>.<queuename> For example:

stdout.inbox

Parameters

- **source** (*str*) – The source queue in <module.queue_name> syntax
- **destination** (*str*) – The destination queue in <module.queue_name> syntax

getChildren(*module*)

Returns all the connected child modules

Parameters **module** (*str*) – The name of the module.

Returns A list of module names.

Return type list

registerModule(*module*, *actor_config*, *arguments={}*)

Initializes the wishbone module module.

Parameters

- **module** (Actor) – A Wishbone module object (not initialized)
- **actor_config** (*ActorConfig*) – The module's actor configuration
- **arguments** (*dict*) – The parameters to initialize the module.

start()

Starts all registered modules.

stop()

Stops all running modules.

2.2.6 Bootstrap

Boostrapping a Wishbone server on CLI is done using a YAML config file which describes the modules to initialize and their connections.

Example:

```
$ wishbone start --config /etc/wishbone_bootstrap.yaml --pid /var/run/wishbone.pid
```

Wishbone CLI commands

start

Starts Wishbone in the background

Example:

```
$ wishbone start --config /etc/wishbone_bootstrap.yaml --pid /var/run/wishbone.pid
```

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.

```
$ wishbone start -h
usage: wishbone start [-h] [--config CONFIG] [--instances INSTANCES]
                      [--pid PID] [--queue-size QUEUE_SIZE]
                      [--frequency FREQUENCY] [--id IDENTIFICATION]
                      [--module_path MODULE_PATH] [--graph]

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 IDENTIFICATION   An identification string.
  --module_path MODULE_PATH
                        A comma separated list of directories to search and
                        find Wishbone modules.
```

debug

Starts Wishbone in foreground writing all logs to stdout.

Example:

```
$ wishbone debug --config /etc/wishbone_bootstrap.yaml --graph
```

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

```
$ wishbone debug --help
usage: wishbone debug [-h] [--config CONFIG] [--instances INSTANCES]
                      [--queue-size QUEUE_SIZE] [--frequency FREQUENCY]
                      [--id IDENTIFICATION] [--module_path MODULE_PATH]
                      [--graph] [--graph_include_sys] [--profile]

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 IDENTIFICATION   An identification string.
  --module_path MODULE_PATH
                        A comma separated list of directories to search and
                        find Wishbone modules.
  --graph               When enabled starts a webserver on 8088 showing a
                        graph of connected modules and queues.
  --graph_include_sys   When enabled includes logs and metrics related queues
                        modules and queues to graph layout.
  --profile             When enabled profiles the process and dumps a profile
                        file in the current directory. The profile file can be
                        loaded in Chrome developer tools.
```

stop

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

Example:

```
$ wishbone stop --pid /var/run/wishbone.pid
```

```
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

Kills Wishbone using the provided pid file

Example:

```
$ wishbone kill --pid /var/run/wishbone.pid
```

Warning: Sends a SIGKILL to the pids in the pidfile.

```
$ 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.

Example:

```
$ wishbone list
```

```
$ wishbone list
          __        __    __
.--.--.--|__.-----|  |--|  |--.-----.-----.-----.
|  |  |  |  |__ --|     |  _  |  _  |     |  -__|
|________|__|_____|__|__|_____|_____|__|__|_____|
                                  version 1.2.0

Build composable event pipeline servers with minimal effort.

Available modules:
+----------+----------+-----------------+---------+-----------------------------------------------
| Category | Group    | Module          | Version | Description
+----------+----------+-----------------+---------+-----------------------------------------------
|          |          |                 |         |
| wishbone | flow     | fanout          |   1.2.0 | Forward each incoming message to all connected que
|          |          | funnel          |   1.2.0 | Funnel multiple incoming queues to 1 outgoing queu
|          |          | match           |   1.2.0 | Pattern matching on a key/value document stream.
|          |          | roundrobin      |   1.2.0 | Round-robins incoming events to all connected queu
|          |          | ttl             |   1.2.0 | Allows messages to pass a maximum number of times
|          |          |                 |         |
|          | encode   | graphite        |   1.2.0 | Converts the internal metric format to Graphite fo
|          |          | humanlogformat  |   1.2.0 | Formats Wishbone log events.
|          |          | influxdb        |   1.2.0 | Converts the internal metric format to InfluxDB li
|          |          | json            |   1.2.0 | Encodes Python data objects to JSON strings.
|          |          | msgpack         |   1.2.0 | Encodes Python objects to MSGPack format.
|          |          |                 |         |
|          | decode   | json            |   1.2.0 | Decodes JSON data to Python data objects.
|          |          | msgpack         |   1.2.0 | Decodes MSGPack data into Python objects.
|          |          |                 |         |
|          | function | jsonvalidate    |   1.2.0 | Validates JSON data against JSON-schema.
|          |          | loglevelfilter  |   1.2.0 | Filters log events based on loglevel.
|          |          | modify          |   1.2.0 | Modify and manipulate datastructures.
|          |          | template        |   1.2.0 | A Wishbone module which generates a text from a di
|          |          |                 |         |
```

```
|           | input    | amqp           |   1.2.0 | Consumes messages from AMQP.
|           |          | dictgenerator  |   1.2.0 | Generates random dictionaries.
|           |          | disk           |   1.2.0 | Reads messages from a disk buffer.
|           |          | gearman        |   1.2.0 | Consumes events/jobs from  Gearmand.
|           |          | httpclient     |   1.2.0 | A HTTP client doing http requests to pull data in
|           |          | httpserver     |   1.2.0 | Receive events over HTTP.
|           |          | namedpipe      |   1.2.0 | Takes data in from a named pipe..
|           |          | tcp            |   1.2.0 | A TCP server.
|           |          | testevent      |   1.2.0 | Generates a test event at the chosen interval.
|           |          | udp            |   1.2.0 | A UDP server.
|           |          | zeromq_pull    |   1.2.0 | Pulls events from one or more ZeroMQ push modules
|           |          | zeromq_topic   |   1.2.0 | Subscribes to one or more ZeroMQ Topic publish mod
|           |          |                |         |
|           | output   | amqp           |   1.2.0 | Produces messages to AMQP.
|           |          | disk           |   1.2.0 | Writes complete messages to a disk buffer.
|           |          | elasticsearch  |   1.2.0 | Submit data to Elasticsearch.
|           |          | email          |   1.2.0 | Sends out incoming events as email.
|           |          | file           |   1.2.0 | Writes events to a file
|           |          | http           |   1.2.0 | Posts data to the requested URL
|           |          | null           |   1.2.0 | Purges incoming events.
|           |          | sse            |   1.2.0 | A server sent events module.
|           |          | stdout         |   1.2.0 | Prints incoming events to STDOUT.
|           |          | syslog         |   1.2.0 | Writes log events to syslog.
|           |          | tcp            |   1.2.0 | A TCP client which writes data to a TCP socket.
|           |          | udp            |   1.2.0 | A UDP client which writes data to an UDP socket.
|           |          | uds            |   1.2.0 | Writes events to a Unix Domain Socket.
|           |          | zeromq_push    |   1.2.0 | Pushes events out to one or more ZeroMQ pull modul
|           |          | zeromq_topic   |   1.2.0 | Publishes data to one or more ZeroMQ Topic subscri
|           |          |                |         |
+-----------+----------+----------------+---------+-----------------------------------------------
```

show

Displays the docstring of the requested module.

Example:

```
$ wishbone show --module wishbone.flow.fanout
```

```
$ wishbone show --module wishbone.flow.fanout


          __        __    __
.--.--.--|__.-----|  |--|  |--.-----.-----.-----.
|  |  |  |  |__ --|     |  _  |  _  |     |  -__|
|_____|__|__|_____|__|__|_____|_____|__|__|_____|
                                  version 2.1.0

Build composable event pipeline servers with minimal effort.


====================
wishbone.flow.fanout
====================

Version: 2.1.0

Forward each incoming message to all connected queues.
```

```
--------------------------------------------------


    Forward each incoming message to all connected queues.

    Parameters:

        n/a


    Queues:

        inbox
         |  Outgoing events.
```

Bootstrap Files

> *Bootstrap files are YAML formatted configuration files describing the modules to initialize and how they are connected used to start a Wishbone server on command line.*

A bootstrap file contains the configuration of all the lookup functions and modules to initialize and how they are connected to each other.

Bootstrap files are loaded using the *–config* option:

```
$ wishbone start --config static_dynamic.yaml
```

Example bootstrap file

```
---
lookups:
  randomword:
    module: wishbone.lookup.randomword

modules:
  dynamic:
    description: Generating an event per second just for fun.
    module: wishbone.input.testevent
    arguments:
      message: ~~randomword()
      interval: 1

  screen:
    module: wishbone.output.stdout

routingtable:
  - dynamic.outbox        -> screen.inbox
...
```

Bootstrap files consist out of 3 sections:

lookups

See lookup functions for more information.

The *lookups* section has following format in JSON-schema:

```
{
  "lookups": {
    "patternProperties": {
      ".*": {
        "additionalProperties": false,
        "properties": {
          "arguments": {
            "type": "object"
          },
          "module": {
            "type": "string"
          }
        },
        "required": [
          "module"
        ],
        "type": "object"
      }
    },
    "type": "object"
  }
}
```

The name of the key (line 3) defines the name of the function to use in the *module section* (line 11).

Parameters:

module

The name of the module to load in dotted format.

If the lookup name is *wishbone.lookup.randomword* then the following Ptyhon code should be able to execute successfully:

```
from wishbone.lookup import randomword
```

arguments

An optional dictionary of arguments used to initialize the lookup module.

modules

The *modules* section has following format in JSON-schema:

```
{
  "patternProperties": {
    ".*": {
      "additionalProperties": false,
      "properties": {
        "arguments": {
          "type": "object"
        },
        "description": {
          "type": "string"
        },
        "module": {
          "type": "string"
        }
      },
```

```
      "required": [
        "module"
      ],
      "type": "object"
    }
  },
  "type": "object"
}
```

The name of the key (line 7) defines the name of the module instance.

Parameters:

description

An optional description explaining what the module does. The descriptoin appears in the module and queue graph. See –monitor option.

arguments

An optional dictionary of keyword arguments used to initialize the module.

routingtable

The *routing* section has following format in JSON-schema:

```
{
  "routingtable": {
    "type": "array"
  }
}
```

The routing table connects the queues of the different module instances into the desired flow.

The section consists out of a list of entries containing a source queue, a separator -> and a destination queue.

Logs

The bootstrap process is responsible for loading and initializing the event modules automatically connects the `logs` queue of each module to a *`wishbone.module.funnel.Funnel`* instance named `_logs`. This module then effectively receives all log events.

If the user decides not to connect the `_logs.outbox` queue to another module then the bootstrap process will automatically initialize additional modules to send these to either SYSLOG or STDOUT depending on it's started to run in the background (–start) or foreground (–debug) respectively.

If you would like to forward the log events of all the modules to a module of your choice, you can achieve this by connecting the `_logs.outbox` queue to the desired module.

Metrics

The bootstrap process is responsible for loading and initializing the event modules automatically connects the `metrics` queue of each module to a *`wishbone.module.funnel.Funnel`* instance named `_metrics`. This module then effectively receives all metric events.

If the user decides not to connect the `_metrics.outbox` queue to another module all metrics will simply be dropped.

By default the `_metrics.outbox` queue is not connected to another module (in contrary to `_logs.outbox`) therefor all metric data is lost by default. If however you would like to process the Wishbone metrics externally you can hook up the necessary modules to **_metrics.outbox** to achieve the desired result.

Note: The `--frequency` parameter determines the rate metrics are collected.

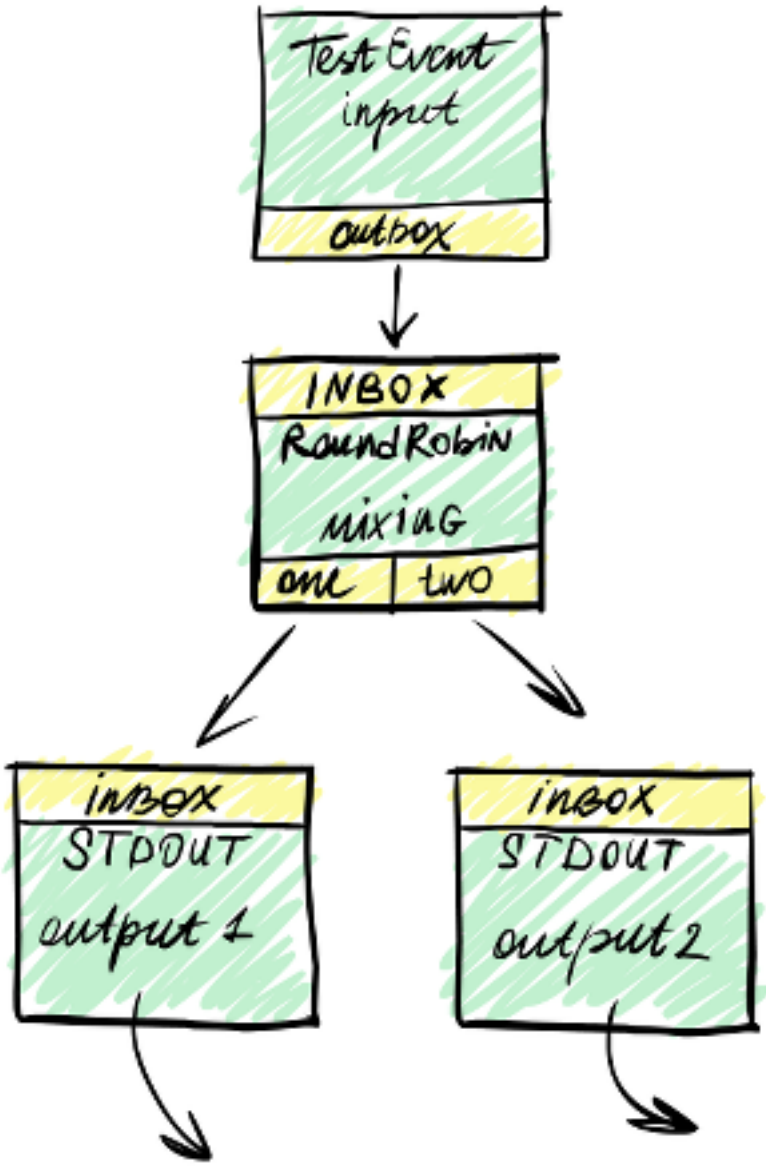
For example you can forward the Wishbone metrics to Graphite by chaining wishbone.encode.graphite (converts *wishbone.event.Metric* into a Graphite format) and wishbone.output.tcp (submits the Graphite data over TCP to Graphite).

2.2.7 Examples

Event pipeline

The following bootstrap file uses the *wishbone.module.testevent.TestEvent* module to generate a message every second. To provide the content of the message is uses the lookup module *wishbone.lookup.random_word.RandomWord* to generate the content of the message.

The generated event is then passed onto the flow module *wishbone.module.roundrobin.RoundRobin* which *"roundrobins"* the event over 2 *wishbone.module.stdout.STDOUT* output modules which print the incoming event to STDOUT.




```
lookups:
  randomword:
    module: wishbone.lookup.randomword
    arguments:
      interval: 1

modules:
  input:
    module: wishbone.input.testevent
    description: I generate a random word.
    arguments:
      message: ~~randomword()

  mixing:
    module: wishbone.flow.roundrobin
    description: I roundrobin incoming messages
```

```
  output1:
    module: wishbone.output.stdout
    description: I write incoming messages to stdout.
    arguments:
      prefix: "I am output #1: "

  output2:
    module: wishbone.output.stdout
    description: I write incoming messages to stdout.
    arguments:
      prefix: "I am output #2: "

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

The server can be bootstrapped on CLI by issuing following command:

```
$ wishbone debug --config bootstrap.yaml
```

Writing a module

Wishbone is all about making the development of custom modules as simple as possible by taking care of the things which distract you from writing the actual code you want to be working on.

Introduction

The following example module evaluates whether an integer is between a *minimum* and a *maximum* value. Depending whether that's a case the event will be submitted to the appropriate queue.

```
#!/usr/bin/env python

from wishbone import Actor


class BiggerAndSmaller(Actor):

    '''**Checks whether an integer is between min and max.**

    Checks whether an integer is between the defined minimum and maximum
    values.  When the value is inside the scope it is submitted to the
    *inside* queue otherwise it is submitted to the *outside* queue.


    Parameters:

        - selection(str)('@data')
           |  The value

        - min(int)(1)
           |  The minimum integer value.

        - max(int)(100)
           |  The maximum integer value.

```

```
26      Queues:
27
28          - inbox
29             |  Incoming messages
30
31          - inside
32             |  Values are inside the <min> and <max> values.
33
34          - outside
35             |  Values are outside the <min> and <max> values.
36      '''
37
38      def __init__(self, actor_config, selection='@data', min=1, max=100):
39          Actor.__init__(self, actor_config)
40
41          self.pool.createQueue("inbox")
42          self.pool.createQueue("inside")
43          self.pool.createQueue("outside")
44          self.registerConsumer(self.consume, "inbox")
45
46      def consume(self, event):
47
48          if not isinstance(event.data, int):
49              raise TypeError("Event data is not type integer")
50
51          if event.get(self.kwargs.selection) >= self.kwargs.min and event.data <= self.kwargs.max:
52              self.submit(event, self.pool.queue.inside)
53          else:
54              self.submit(event, self.pool.queue.outside)
55
```

Document the module (line 8-36)

The docstring (line 8-36) contains the module's description. It's encouraged to document your module similarly. The content of the docstring can be accessed on CLI using the show command.

Inherit baseclass (line 3, 6)

A Wishbone module is created by making a class which uses *wishbone.Actor* as a base class.

This baseclass must be initialized (line 35) with the *actor_config* value provided when initializing `(line 34)` the module.

The *actor_config* value is a `wishbone.actor.ActorConfig` instance containing the configuration specific to the modules behavior inside the Wishbone framework. This instance is automatically created and provided by the framework so it's not of any concern when developing the module.

Creating the required queues

All the module's queues are stored in `wishbone.pool` which is an instance of `wishbone.queue.QueuePool`.

Besides for the default *failed* and *success* queues, it's left up to the developer to create the required queues. Creating queues is done by invoking the `wishbone.queue.QueuePool.createQueue` (line 37, 38, 39).

Registering a function

For this specific module, we're expecting that data events arrive to the **inbox** queue. Wishbone takes care of draining that queue and applying the events to the function responsible for processing the event.

You need to have such a function, otherwise events will not get consumed for the queue and the therefor the queue will just fill up without anything happening.

Registering such a function is done with *wishbone.Actor.registerConsumer()* (line 40). The function consuming the events (line 42) must have 1 parameter to receive the events(`wishbone.Event`).

Submitting the event to a queue

After processing the event it must be submitted to the relevant queue so it can be forwarded to the next module.

Submitting an event to a queue is done with *wishbone.Actor.submit()* (line 48, 50).

Dealing with exceptions while processing events

If an *exception* occurs within the registered function (we deliberately invoke one on line 44) then Wishbone will automatically submit the event to the module's default **failed** queue.

Make the module importable by Wishbone

Wishbone uses Python's setuptools entrypoint definitions to load modules. These are defined in the module's setup.py file.

Example:

```
entry_points={
    'mymodule.flow': [
        'biggersmaller = biggerandsmaller:BiggerAndSmaller'
    ]
}
```

This entrypoint definition allows Wishbone to import the module using *mymodule.flow.biggersmaller* in the boostrap file.

2.2.8 Miscellaneous

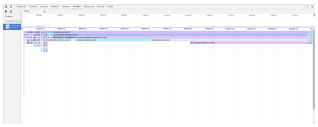
Profiling

If you want to profile a Wishbone server to locate a performance issue or identify a bottleneck you can start the server in *debug* mode using the *–profile* option.

```
$ wishbone debug --config test.yaml --profile
```

Pressing CTRL+C will stop the server and dump the profile file named *"wishbone_<pid>_.cpuprofile"* in the current working directory.

The profile file can be loaded into Chrome's "Developer Tools" for further analysis.

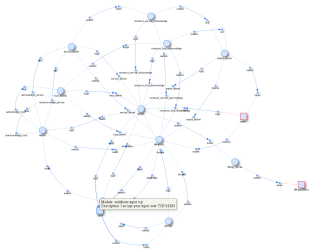


Graph topology

If you have a complex bootstrap file it can be difficult to get a good insight into the data flow. To get a graphical representation of the loaded bootstrap file you can start wishbone using the *–graph* switch.

```
$ wishbone debug --config test.yaml --graph
```

This will start a webserver which listens on port 8088. Visiting the url with your browser produces a graph showing all the loaded modules including the connected queues.



Caveats

Wishbone comes with a couple of caveats you need to keep in mind:

- **Storing data in** *wishbone.event.Event*:

  You can store whatever Python object type in *wishbone.event.Event* but when using `wishbone.Event.clone()` keep in mind that only a naive copy is done which works fine for default types but probably not for more complex objects. It does not behave as a deepcopy().Preferably you should only store JSON style data in Events.

- **Queues which are not connected**:

  When a queue is not connected to another queue then submitting a message into it will result into the message being dropped.This is by design to ensure queues do not fill up without ever being consumed.

- **IO-bound VS CPU-bound workload**:

  Since Wishbone heavily leans on the Gevent library it lends itself best for IO bound workloads.If you have a CPU intensive task, a good idea might be to decouple the IO part from the CPU-intensive part of the chain by running multiple Wishbone processes and pass messages from one to the other.

S

T