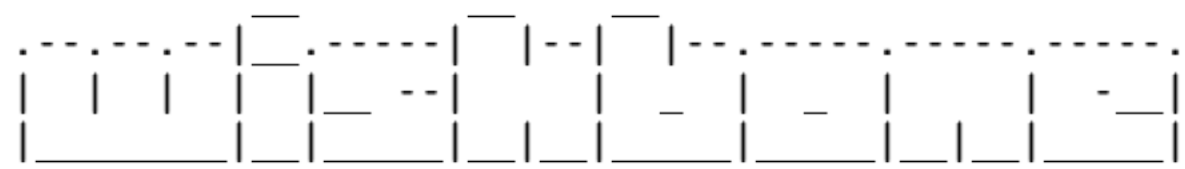
Wishbone Documentation

Release 3.1.4

Jelle Smet

Jul 28, 2018

Contents



A pragmatic framework to build reactive event processing services.

Wishbone is a **Python** framework to build reactive event processing services by combining and connecting modules into a processing pipeline through which events flow, modify and trigger interactions with remote services.

The framework can be used to implement a wide area of solutions such as mashup enablers, ETL servers, stream processing servers, webhook services , ChatOps services, bots, Cloud-based integrations and all kinds of event driven automation.

CHAPTER 1

Why?

The goal of the project is to provide a complete, expressive and ops friendly programming framework which removes a maximum of (boring) boilerplate without sacrificing flexibility.

CHAPTER 2

When?

Wishbone will probably be useful to you when tackling the:

> "*If this event happens I want to trigger that action . . .* " - kind of problems.

2.1 Installation

Note: Wishbone is developed for python 3.6+

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.

Installation sources:

2.1.1 Python

Note: You really should install Wishbone inside virtualenv so you do not mess with your OS Python packages.

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

```
$ pip install wishbone
```

2.1.2 Source

Note: You really should install Wishbone inside virtualenv so you do not mess with your OS Python packages.

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

CI builds can be seen here: https://travis-ci.org/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
```

2.1.3 Docker

Note: The Wishbone containers are big. Any help reducing the size is highly appreciated.

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

The docker files necessary to build Wishbone containers can be found here.

```
$ docker pull smetj/wishbone
$ docker images
REPOSITORY                                          TAG                 IMAGE ID
↪        CREATED              SIZE
smetj/wishbone                                      base_python
↪72c6cc524d53        26 minutes ago       778 MB
smetj/wishbone                                      develop
↪81e425eb6784        5 minutes ago       806 MB
```

- The `smetj/wishbone:base_python` container is a Python3.6 based container containing the necessary dependencies to install Wishbone.
- The `develop` tag tracks the Wishbone develop branch.
- The `master` tag tracks the Wishbone master branch.

The container entrypoint is pointing to the wishbone executable:

```
$ docker run -t -i smetj/wishbone:develop
usage: wishbone [-h] {start,stop,list,show} ...
wishbone: error: the following arguments are required: command
```

The following commands runs a Wishbone container:

```
$ docker run --volume ${PWD}/bootstrap.yaml:/tmp/bootstrap.yaml smetj/
↪wishbone:develop start --config /tmp/bootstrap.yaml
```

Installing additional modules

To install additional Wishbone modules inside the Docker container you will have to build a new container.

```
FROM            smetj/wishbone:develop
MAINTAINER      Jelle Smet
RUN             /opt/python/bin/pip3 install --process-dependency-link https://github.
↪com/smetj/wishbone-input-httpserver/archive/wishbone3.zip
```

Building the container:

```
$ docker run -t -i smetj/wishbone:http list
```

Running the container:

```
$ docker run -t -i smetj/wishbone:http list
           __        __    __
.--.--.--|__.-----|  |--|  |--.-----.-----.-----.
|  |  |  |  |__ --|     |  _  |  _  |     |  -__|
|_____|__|_____|__|__|_____|_____|__|__|_____|
                                  version 3.0.0

+------------------+----------------+----------+----------------+---------+-----------
↪-------------------------------------------------------------+
| Namespace        | Component type | Category | Name           | Version |␣
↪Description                                                  |
+------------------+----------------+----------+----------------+---------+-----------
↪-------------------------------------------------------------+
|                  |                |          |                |         |␣
↪                                                             |
| wishbone         | protocol       | decode   | dummy          |   3.0.0 | A dummy␣
↪decoder.                                                     |
|                  |                |          | json           |   3.0.0 | Decode␣
↪JSON data into a Python data structure.                      |
|                  |                |          | msgpack        |   3.0.0 | Decode␣
↪MSGpack data into a Python data structure.                   |
|                  |                |          | plain          |   3.0.0 | Decode␣
↪plaintext using the defined charset.                         |
|                  |                |          |                |         |␣
↪                                                             |
```

(continues on next page)

(continued from previous page)

```
|                   |                | encode    | dummy           |   3.0.0 | A dummy␣
↪encoder.                                                                  |
|                   |                |           | json            |   3.0.0 | Encode␣
↪data into JSON format.                                                    |
|                   |                |           | msgpack         |   3.0.0 | Encode␣
↪data into msgpack format.                                                 |
|                   |                |           |                 |         |          ␣
↪                                                                          |

...snip ...

| wishbone_contrib | module         | input     | httpserver      |   1.1.0 | Receive␣
↪events over HTTP.                                                         |
|                   |                |           |                 |         |          ␣
↪                                                                          |
+------------------+----------------+-----------+-----------------+---------+-----------
↪-------------------------------------------------------------------+
```

2.2 Components

A Wishbone service consists out of a combination of different components. Wishbone has 3 component types:

2.2.1 Modules

Modules are isolated pieces of code which do not directly invoke each others functionality. They merely act upon the messages coming in to its queues and submit messages into another queue for the next module to process. Modules run as greenthreads.

Wishbone comes with a set of builtin modules. Besides these, there's a collection of external modules available which are developed and released seperately from Wishbone itself.

Wishbone has following module types:

Input

Note: Input modules either take events from the outside world or generate events.

Input module properties:

- They have a *protocol decoder method* mapped to `wishbone.module.InputModule.decode()` in order to convert the incoming data into a workable datastructure.
- `wishbone.actorconfig.ActorConfig.protocol_function` determines whether *`wishbone.module.InputModule.generateEvent()`* either expects events from the outside world to be Wishbone events or regular data.
- Contextual data about the incoming event can/should be stored under `tmp.<module name>`.
- Should always have an `destination` and `native_events` parameter.

- Should use `wishbone.actor.Actor.generateEvent()` to generate the event in which to store the incoming data. It takes care of how the event is created in relation to the obligatory `destination` and `native_events` parameters.
- If you're setting a default decoder function make sure you use *wishbone.module.InputModule.setDecoder()* as this method will prevent overwrite any user defined decoder set via `wishbone.actorconfig.ActorConfig`.

The builtin Wishbone Input modules:

Name	Description
`wishbone.module.input.cron`	Generates an event at the defined time.
`wishbone.module.input.generator`	Generates an event at the chosen interval.
`wishbone.module.input.inotify`	Monitors one or more paths for inotify events.

`Input` modules must base *wishbone.module.InputModule*:

class wishbone.module.**InputModule**(*config*)
Bases: `wishbone.actor.Actor`

generateEvent(*data={}*, *destination=None*)
Generates a new event.

This function can get overridden by `wishbone.module.InputModule._generateNativeEvent`.

The provided `data` will be traversed in search of valid templates which then will be rendered.

Parameters

- **data** – The payload to add to the event.
- **destination** – The destination key to write the data to

Returns An event containing `data` as a payload.

Return type wishbone.event.Event

getDecoder()
Returns a new instance of the `handler()` method of the decoder set by `self.setDecoder()`. Each concurrent incoming data stream should use its own instance of the decoder otherwise they end up overwriting each other's content.

loop()
The global lock for this module.

Returns True when module is in running mode. False if not.

Return type bool

postHook()
Is executed when module exits.

preHook()
Is executed when module starts. Can be overriden by the user.

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

Don't not trap errors here. When <function> fails then the event will be submitted to the "failed" queue, If <function> succeeds to the success queue.

Registering `function` to consume `queue` will also apply all the registered module functions against the events consumed from it.

Parameters

- **`function`** – The function which processes events
- **`queue`** – The name of the queue from which `function` will process the events.

Returns None

`renderEventKwargs`(*event*, *queue=None*)
Renders kwargs using the content of `event` and stores the result under `event.kwargs`.

Parameters

- **`event`** – An Event instance
- **`queue`** – The queue name so `RenderKwargs` can store the results in the correct queue context.

Returns The provided event instance.

Return type `wishbone.event.Event`

`renderKwargs`()
Renders kwargs without making use of event content. This is typically used when initiliazing a module and render the defined kwargs which do not need a event data for rendering.

Returns None

`sendToBackground`(*function*, **args*, ***kwargs*)
Executes a function and sends it to the background. Such a function should never exit until `self.loop` returns `False`. This *method* wraps `function` again in a loop as long `self.loop` returns `False` so that `function` is restarted and an error is logged.

Parameters

- **`function`** – The function which has to be executed.
- **`*args`** – Variable length argument list.
- **`**kwargs`** – Arbitrary keyword arguments.

`setDecoder`(*name*, **args*, ***kwargs*)
Sets the decoder with name <name> unless there's already a decoder defined via `actorconfig.ActorConfig`.

Parameters

- **`name`** – The entrypoint name of the decoder to initialize
- **`*args`** – Variable length argument list.
- **`**kwargs`** – Arbitrary keyword arguments.

`start`()
Starts the module.

Returns None

`stop`()
Makes `self.loop` return `False` and handles shutdown of of the registered background jobs.

`submit`(*event*, *queue*)
Submits <event> to the queue with name <queue>.

Parameters

- **event** – An event instance.
- **queue** – The name of the queue

Returns None

Output

Note: Output modules submit data to external services.

Output module properties:

- They have a *protocol encoder method* mapped to `wishbone.module.OutputModule.encode()` in order to convert the desired `wishbone.event.Event` payload into the desired format prior to submitting it to the external service.
- Should **always** provide a `selection`, `payload`, `native_events` and `parallel_streams` module parameter. If `payload` is not `None`, then it takes precendence over `selection`. `Selection` defines the event key to submit whilst template comes up with a string to submit.‘‘payload‘‘ usually makes no sense with bulk events.
- Should use *`wishbone.module.OutputModule.getDataToSubmit()`* to retrieve the actual data to submit to the external service.This automatically takes care of bulk events.
- Through inheriting *`wishbone.module.OutputModule`* *Output* modules override `wishbone.actor.Actor._consumer()` with their own version which executes the registered `function` in parallel greenthreads by using a threadpool. The module's `parallel_streams` parameter defines the size of the pool and therefor the number of parallel greenthreads submitting the event data externally.It depends on the nature of your output protocol whether this makes sense.Normally you shouldn't really bother with this as long a Gevent's monkey patching works on the code you're using to speak to the remote service.
- If you're setting a default encoder function make sure you use *`wishbone.module.OutputModule.setEncoder()`* as this method will prevent overwrite any user defined encoder set via `wishbone.actorconfig.ActorConfig`.

Warning: Be aware that if `parallel_streams` is larger than 1, the equal amount of events will be processed concurrently by the function registered with `wishbone.actor.Actor.registerConsumer()` to consume the queue. Within that function do **NOT** change shared (module) variables but only use local (to the function) ones.

The builtin Wishbone Output modules:

Name	Description
`wishbone.module.output.null`	Purges events.
`wishbone.module.output.stdout`	Prints event data to STDOUT.
`wishbone.module.output.syslog`	Submits event data to syslog.

`Output` modules must base *`wishbone.module.OutputModule`*:

class wishbone.module.**OutputModule**(*config*)
Bases: `wishbone.actor.Actor`

generateEvent(*data={}*, *destination=None*)
Generates a new event.

This function can get overridden by `wishbone.module.InputModule._generateNativeEvent`.

The provided `data` will be traversed in search of valid templates which then will be rendered.

Parameters

- **data** – The payload to add to the event.
- **destination** – The destination key to write the data to

Returns An event containing `data` as a payload.

Return type wishbone.event.Event

getDataToSubmit(*event*)
Derives the data to submit from `event` taking into account `native_events`, `payload` and `selection` module parameters.

Parameters **event** – The event to extract data from.

Returns The data to submit.

Return type dict/str/...

loop()
The global lock for this module.

Returns True when module is in running mode. False if not.

Return type bool

postHook()
Is executed when module exits.

preHook()
Is executed when module starts. Can be overriden by the user.

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

Don't not trap errors here. When <function> fails then the event will be submitted to the "failed" queue, If <function> succeeds to the success queue.

Registering `function` to consume `queue` will also apply all the registered module functions against the events consumed from it.

Parameters

- **function** – The function which processes events
- **queue** – The name of the queue from which `function` will process the events.

Returns None

renderEventKwargs(*event*, *queue=None*)
Renders kwargs using the content of `event` and stores the result under `event.kwargs`.

Parameters

- **event** – An Event instance
- **queue** – The queue name so `RenderKwargs` can store the results in the correct queue context.

Returns The provided event instance.

Return type `wishbone.event.Event`

renderKwargs()
Renders kwargs without making use of event content. This is typically used when initiliazing a module and render the defined kwargs which do not need a event data for rendering.

Returns None

sendToBackground(*function*, **args*, ***kwargs*)
Executes a function and sends it to the background. Such a function should never exit until `self.loop` returns `False`. This *method* wraps `function` again in a loop as long `self.loop` returns `False` so that `function` is restarted and an error is logged.

Parameters

- **function** – The function which has to be executed.
- ***args** – Variable length argument list.
- ****kwargs** – Arbitrary keyword arguments.

setEncoder(*name*, **args*, ***kwargs*)
Sets the encoder with name <name> unless there's already an encoder defined via `actorconfig.ActorConfig`.

Parameters

- **name** – The name of the encoder to initialize
- ***args** – Variable length argument list.
- ****kwargs** – Arbitrary keyword arguments.

Returns

True if the encoder is set, False when an encoder was already set via `actorconfig.ActorConfig`

Return type Bool

start()
Starts the module.

Returns None

stop()
Makes `self.loop` return `False` and handles shutdown of of the registered background jobs.

submit(*event*, *queue*)
Submits <event> to the queue with name <queue>.

Parameters

- **event** – An event instance.
- **queue** – The name of the queue

Returns None

Flow

Note: Flow modules apply logic of some sort to decide which queue to submit the event to without altering the event's payload.

Flow modules select the outgoing queue to which incoming events are submitted based on certain conditions. For example, Wishbone queues can only be connected 1 queue.

If you need a *1-to-many* or a *many-to-1* queue connection then you can use the `wishbone.module.fanout.Fanout` or `wishbone.module.funnel.Funnel` respectively.

Some of the characteristics of *flow* modules are:

- They do not alter the content of events flowing through except optionally setting some contextual data.

The builtin flow modules are:

Name	Description
`wishbone.module.flow.acknowledge`	Forwards or drops events by acknowleding values.
`wishbone.module.flow.count`	Pass or drop events based on the number of times an event value occurs.
`wishbone.module.flow.fanout`	Forward each incoming message to all connected queues.
`wishbone.module.fresh.Fresh`	Generates a new event unless an event came through in the last x time.
`wishbone.module.funnel.Funnel`	Funnel multiple incoming queues to 1 outgoing queue.
`wishbone.module.flow.queueselect`	Submits message to the queue defined by a rendered template.
`wishbone.module.flow.roundrobin`	Round-robins incoming events to all connected queues.
`wishbone.module.flow.switch`	Switch outgoing queues while forwarding events.

Process

Note: Process modules process and therefor modify events in one way or another.

Process modules usually aren't very cooperative in the Gevent sense of doing things since they aren't supposed to do any IO.

The builtin Wishbone Output modules:

Name	Description
`wishbone.module.process.modify`	Modify and manipulate datastructures.
`wishbone.module.process.pack`	Packs multiple events into a bulk event.
`wishbone.module.process.template`	Renders Jinja2 templates.
`wishbone.module.process.unpack`	Unpacks bulk events into single events.

`Process` modules must base *`wishbone.module.ProcessModule`*:

class wishbone.module.**ProcessModule**(*config*)

Bases: `wishbone.actor.Actor`

generateEvent(*data={}*, *destination=None*)
Generates a new event.

This function can get overridden by `wishbone.module.InputModule._generateNativeEvent`.

The provided `data` will be traversed in search of valid templates which then will be rendered.

Parameters

- **data** – The payload to add to the event.
- **destination** – The destination key to write the data to

Returns An event containing `data` as a payload.

Return type wishbone.event.Event

loop()
The global lock for this module.

Returns True when module is in running mode. False if not.

Return type bool

postHook()
Is executed when module exits.

preHook()
Is executed when module starts. Can be overriden by the user.

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

Don't not trap errors here. When <function> fails then the event will be submitted to the "failed" queue, If <function> succeeds to the success queue.

Registering `function` to consume `queue` will also apply all the registered module functions against the events consumed from it.

Parameters

- **function** – The function which processes events
- **queue** – The name of the queue from which `function` will process the events.

Returns None

renderEventKwargs(*event*, *queue=None*)
Renders kwargs using the content of `event` and stores the result under `event.kwargs`.

Parameters

- **event** – An Event instance
- **queue** – The queue name so `RenderKwargs` can store the results in the correct queue context.

Returns The provided event instance.

Return type `wishbone.event.Event`

renderKwargs()
Renders kwargs without making use of event content. This is typically used when initiliazing a module and render the defined kwargs which do not need a event data for rendering.

Returns None

sendToBackground (*function*, **args*, ***kwargs*)
Executes a function and sends it to the background. Such a function should never exit until `self.loop` returns `False`. This *method* wraps `function` again in a loop as long `self.loop` returns `False` so that `function` is restarted and an error is logged.

Parameters

- **function** – The function which has to be executed.
- ***args** – Variable length argument list.
- ****kwargs** – Arbitrary keyword arguments.

start()
Starts the module.

Returns None

stop()
Makes `self.loop` return `False` and handles shutdown of of the registered background jobs.

submit (*event*, *queue*)
Submits <event> to the queue with name <queue>.

Parameters

- **event** – An event instance.
- **queue** – The name of the queue

Returns None

Characteristics

- If a queue is not connected to another queue then the messages submitted to it are dropped. This is by design to prevent queues from filling up.
- Each module has by default a `_success` and `_failed` queue to which a copy of passing events is submitted if it has been processed successfully or not.
- Each module has by default a `_logs` and `_metrics` queue to which logs and metrics are submitted respectively.

Module configuration

A module has an arbirary number of parameters but always needs to accept `wishbone.actorconfig.ActorConfig` which passes Wishbone specific the characteristics to it:

```
from wishbone.module.generator import Generator
from wishbone.actor import ActorConfig

actor_config = ActorConfig(
    name='generator'
    size=100
    frequency=1,
    template_functions={},
    description="This is a fizzbuzz exaple"
)
test_event = Generator(actor_config, payload="test")

test_event.pool.queue.outbox.disableFallThrough()
test_event.start()
```

(continues on next page)

(continued from previous page)

```
event = getter(test_event.pool.queue.outbox)
assert event.get() == "test"
```

2.2.2 Functions

Functions are small pieces of reusable code which can be applied to different parts of a Wishbone setup. They are initialized initialized in the *module_functions* or *template_functions* section of the bootstrap file.

There are 2 types functions:

Template Functions

Template functions return data which can be used inside a template.

Wishbone makes use of Jinja2 for all its templates. Template functions are functions which can be executed inside templates in order to render data.

Note: When bootstrapping a server the following template functions are **always** available:

- strftime (wishbone.function.template.strftime)
- epoch (wishbone.function.template.epoch)
- env (wishbone.function.template.environment)
- version (wishbone.function.template.)

Characteristics:

- Template functions are functions which are added to the `Jinja2` list of global functions.
- Template functions are classes which base `wishbone.function.template.TemplateFunction`.
- Template functions **must** have a `get()` method which provides the desired data.

Wishbone comes by default with following builtin template functions:

Name	Description
wishbone.function.template.choice	Returns a random element from the provided array.
wishbone.function.template.cycle	Cycles through the provided array returning the next element.
wishbone.function.template.environment	Returns environment variables.
wishbone.function.template.epoch	Returns epoch with sub second accuracy as a float.
wishbone.function.template.pid	Returns the PID of the current process.
wishbone.function.template.random_bool	Randomly returns True or False
wishbone.function.template.random_integer	Returns a random integer.
wishbone.function.template.random_uuid	Returns a uuid value.
wishbone.function.template.random_word	Returns a random word.
wishbone.function.template.regex	Regex matching on a string.
wishbone.function.template.strftime	Returns a formatted version of an epoch timestamp.
wishbone.function.template.version	Returns the version of the desired module.

See following examples:

- *Using a template function.*
- *Creating a template function.*

Module Functions

Module functions are functions in a module which are automatically applied to events when they are consumed from a queue.

Multiple module functions can be chained in order to reach the desired effect. Module function modify events in one way or another.

Characteristics:

- Module functions are applied to events and modify them.
- Module functions are executed when events are consumed from a queue.
- Module functions are only applied to queue which are consumed by a registered function by using `wishbone.actor.Actor.registerConsumer()`.
- When a function returns an error it is logged and skipped and the rest of the module functions will be applied.

Wishbone comes by default with following builtin module functions:

Name	Description
wishbone.function.module.append	Adds a value to an existing list.
wishbone.function.module.lowercase	Puts the desired field in lowercase.
wishbone.function.module.set	Sets a field to the desired value.
wishbone.function.module.uppercase	Puts the desired field in uppercase.

See following examples:

- *Using a module function.*

- *Creating a module function.*

2.2.3 Protocols

Protocol components can be plugged into either *input* or *output* modules and are responsible for converting incoming and outgoing data.

There are 2 types of protocol modules:

Decode

Decode modules can only be used by *input* modules. They are reponsible for converting the incoming data format into a format Wisbone can work with.

Some characteristings:

- Decoder modules should base `wishbone.protocol.Decode`

Wishbone comes with following protocol decoders:

Name	Description
`wishbone.protocol.decode.plain`	Decode plaintext using the defined charset.
`wishbone.protocol.decode.json`	Decode JSON data into a Python data structure.
`wishbone.protocol.decode.msgpack`	Decode MSGpack data into a Python data structure.

See following examples:

- *Using a protocol decoder.*

Encode

Encode modules can only be used by *output* modules. They are reponsible for converting the Wishbone internal format into an appropriate outgoing data format.

Some characteristings:

- Encoder modules should base `wishbone.protocol.Encode`

Wishbone comes with following protocol decoders:

Name	Description
`wishbone.protocol.encode.json`	Encode data into JSON format.
`wishbone.protocol.encode.msgpack`	Encode data into msgpack format.

See following examples:

- *Using a protocol encoder.*

Components are referred to with a unique name written in dotted format:

```
<namespace>.<component type>.<category>.<name>
```

`namespace` is a logical grouping for components. The `wishbone` namespace indicates the component is an one part of the default Wishbone installation or the Wishbone project. If you develop additional components outside of the Wishbone project itself, it is advised to do so in its dedicated namespace.

`component type`, `category` and `name` further categorize the the components into logical groupings.

Each component has an entrypoint so it can be referred to from a bootstrap file or referred to using `wishbone.componentmanager.ComponentManager.getComponentByName()`. The default Wishbone entrypoints are defined in its setup.py file. A component entrypoint is the same as the component name.

An overview of available components can be viewed by using the `list` command:

Tip: By default, the Wishbone executable includes the `wishbone_contrib` and `wishbone_external` into its searchpath when searching for available modules.

```
$ wishbone list
          __        __    __
.--.--.--|__.-----|  |--|  |--.-----.-----.-----.
|  |  |  |  |__ --|     |  _  |  _  |     |  -__|
|_____|__|__|_____|__|__|_____|_____|__|__|_____|
                                  version 3.0.0

Build composable event pipeline servers with minimal effort.


Available components:
+-----------+----------------+----------+----------------+---------+------------------
↪-------------------------------------------------------+
| Namespace | Component type | Category | Name           | Version | Description     ␣
↪                                                       |
+-----------+----------------+----------+----------------+---------+------------------
↪-------------------------------------------------------+
|           |                |          |                |         |                 ␣
↪                                                       |
| wishbone  | protocol       | decode   | dummy          |   3.0.0 | A dummy decoder.␣
↪                                                       |
|           |                |          | json           |   3.0.0 | Decode JSON data␣
↪into a Python data structure.                          |
|           |                |          | msgpack        |   3.0.0 | Decode MSGpack␣
↪data into a Python data structure.                       |
|           |                |          | plain          |   3.0.0 | Decode text data␣
↪into a Python data structure.                          |
|           |                |          |                |         |                 ␣
↪                                                       |
|           |                | encode   | dummy          |   3.0.0 | A dummy encoder.␣
↪                                                       |
|           |                |          | json           |   3.0.0 | Encode data into␣
↪JSON format.                                           |
|           |                |          | msgpack        |   3.0.0 | Encode data into␣
↪msgpack format.                                        |
|           |                |          |                |         |                 ␣
↪                                                       |
|           | function       | module   | append         |   3.0.0 | Adds <data> to␣
↪the array <destination>.                                 |
|           |                |          | lowercase      |   3.0.0 | Puts the desired␣
↪field in lowercase.                                    |
|           |                |          | set            |   3.0.0 | Sets a field to␣
↪the desired value.                                      |
|           |                |          | uppercase      |   3.0.0 | Puts the desired␣
↪field in uppercase.                                    |
|           |                |          |                |         |                 ␣
↪                                                       |
```

(continues on next page)

(continued from previous page)

```
|           |                | template | choice          |   3.0.0 | Returns a random␣
↪element from the provided array.                                    |
|           |                |          | cycle           |   3.0.0 | Cycles through␣
↪the provided array returning the next element.                      |
|           |                |          | epoch           |   3.0.0 | Returns epoch␣
↪with sub second accuracy as a float.                                |
|           |                |          | pid             |   3.0.0 | Returns the PID␣
↪of the current process.                                             |
|           |                |          | random_bool     |   3.0.0 | Randomly returns␣
↪True or False                                                       |
|           |                |          | random_integer  |   3.0.0 | Returns a random␣
↪integer.                                                            |
|           |                |          | random_uuid     |   3.0.0 | Returns a uuid␣
↪value.                                                              |
|           |                |          | random_word     |   3.0.0 | Returns a random␣
↪word.                                                               |
|           |                |          | regex           |   3.0.0 | Regex matching␣
↪on a string.                                                        |
|           |                |          | strftime        |   3.0.0 | Returns a␣
↪formatted version of an epoch timestamp.                            |
|           |                |          |                 |         |                                                                      ␣
↪                                                                    |
|           | module         | flow     | acknowledge     |   3.0.0 | Forwards or␣
↪drops events by acknowleding values.                                |
|           |                |          | count           |   3.0.0 | Pass or drop␣
↪events based on the number of times an event value occurs.          |
|           |                |          | fanout          |   3.0.0 | Forward each␣
↪incoming message to all connected queues.                           |
|           |                |          | fresh           |   3.0.0 | Generates a new␣
↪event unless an event came through in the last x time.              |
|           |                |          | funnel          |   3.0.0 | Funnel multiple␣
↪incoming queues to 1 outgoing queue.                                |
|           |                |          | queueselect     |   3.0.0 | Submits message␣
↪to the queue defined by a rendered template.                        |
|           |                |          | roundrobin      |   3.0.0 | Round-robins␣
↪incoming events to all connected queues.                            |
|           |                |          | switch          |   3.0.0 | Switch outgoing␣
↪queues while forwarding events.                                     |
|           |                |          |                 |         |                                                                      ␣
↪                                                                    |
|           |                | input    | cron            |   3.0.0 | Generates an␣
↪event at the defined time                                           |
|           |                |          | generator       |   3.0.0 | Generates an␣
↪event at the chosen interval.                                       |
|           |                |          | inotify         |   3.0.0 | Monitors one or␣
↪more paths for inotify events.                                      |
|           |                |          |                 |         |                                                                      ␣
↪                                                                    |
|           |                | output   | null            |   3.0.0 | Purges incoming␣
↪events.                                                             |
|           |                |          | stdout          |   3.0.0 | Prints incoming␣
↪events to STDOUT.                                                   |
|           |                |          | syslog          |   3.0.0 | Writes log␣
↪events to syslog.                                                   |
|           |                |          |                 |         |                                                                      ␣
↪                                                                    |
|           |                | process  | modify          |   3.0.0 | Modify and␣
↪manipulate datastructures.                                          |
```

(continues on next page)

(continued from previous page)

```
|           |                |          | pack           |   3.0.0 | Packs multiple␣
↪events into a bulk event.                                  |
|           |                |          | template       |   3.0.0 | Renders Jinja2␣
↪templates.                                                 |
|           |                |          | unpack         |   3.0.0 | Unpacks bulk␣
↪events into single events.                                 |
|           |                |          |                |         |                ␣
↪                                                           |
+-----------+----------------+----------+----------------+---------+-----------------
↪-----------------------------------------------------------+
```

2.3 Bootstrap CLI

An important aspect of Wishbone is the ability to bootstrap a server on CLI.

To bootstrap you need the following items:

2.3.1 Bootstrap File

A boostrap file is written in YAML syntax and it should adhere this schema.

It consists out of 5 sections:

1. **protocols** section:

   This section contains the protocols to initialize.Both protocol instances for *input* and *output* modules should be initialized in this section.It's not necessary to use all the initialized instances. This section is optional.

   A sample extract:

   ```
   protocols:
       json_encode:
           protocol: wishbone.protocol.encode.json
           arguments:
               sort_keys: true
       msgpack_decode:
           protocol: wishbone.protocol.decode.msgpack
   ```

   - The `protocol` value is the *entrypoint* value.
   - `arguments` is optional.

2. **module_functions** section:

   This section initializes the *module functions*. It is not necessary to use all the initialized functions.This section is optional.

   A sample extract:

   ```
   module_functions:
     tagit:
         function: wishbone.function.module.append
         arguments:
             data: you_are_tagged
             destination: tags
   ```

- The `function` value is the *entrypoint* name.
- `arguments is optional`

3. **template_functions** section:

   This section initializes the *template functions*. It is not necessary to use all the initialized functions.This section is optional.

   A sample extract:

```
template_functions:
  gimmeNumber:
    function: wishbone.function.template.choice
    arguments:
      array:
        - one
        - two
        - three
```

   - The `function` value is the *entrypoint* name.
   - `arguments is optional`

4. **modules** section:

   This section initializes *modules*.It is not necessary to connect a module to another module in the *routingtable* section. Otherwise this section is mandatory.

   A sample extract:

```
modules:
  input:
    module: wishbone.module.input.generator
    arguments:
      interval: 1
      payload: hello

  output:
    module: wishbone.module.output.stdout
    arguments:
      prefix: '{{ data }} is the prefix '
      selection: '.'
```

   - The `module` value is the entrypoint name.
   - `arguments` is optional.

5. **routingtable** section:

   The routing table section defines all the connections between the module queues therefor defining the event flow and order the events are passing through modules.

   The entries should have following format:

```
source_module_instance_name.queue_name -> destination_module_instance_name.
↪queue_name
```

   A sample extract:

```
routingtable:
  - input.outbox            -> jsondecode.inbox
  - jsondecode.outbox       -> match.inbox
  - match.email             -> email.inbox
  - match.pagerduty         -> pagerduty.inbox
  - match.mattermost        -> mattermost.inbox
  - match.jira              -> jira.inbox
  - match.msteams           -> msteams.inbox
```

- The routing table is obligatory
- The routing table contains '->' indicating the relation between the source queue and the destination queue.

A complete example can be seen in the *examples* section.

2.3.2 Wishbone executable

The `wishbone` executable takes care of many aspects of setting up your service. It accepts following commands:

- **start**

```
$ wishbone start --help
usage: wishbone start [-h] [--config CONFIG] [--frequency FREQUENCY] [--graph]
                      [--graph_include_sys] [--identification IDENTIFICATION]
                      [--instances INSTANCES] [--log_level LOG_LEVEL] [--fork]
                      [--nocolor] [--pid PID] [--profile]
                      [--queue_size QUEUE_SIZE]

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.
  --frequency FREQUENCY
                        The metric frequency.
  --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.
  --identification IDENTIFICATION
                        An identifier string for generated logs.
  --instances INSTANCES
                        The number of parallel Wishbone instances to
                        bootstrap.
  --loglevel  LOG_LEVEL
                        The maximum loglevel.
  --fork                When defined forks Wishbone to background and INFO
                        logs are written to STDOUT.
  --nocolor             When defined does not print colored output to stdout.
  --pid PID             The pidfile to use.
  --profile             When enabled profiles the process and dumps a Chrome
                        developer tools profile file in the current directory.
  --queue-size QUEUE_SIZE
                        The queue size to use.
```

- **list**

```
$ wishbone list --help
usage: wishbone list [-h] [--namespace NAMESPACE]

Lists the available modules.

optional arguments:
  -h, --help            show this help message and exit
  --namespace NAMESPACE
                        The component namespace to query.
```

- **stop**

```
$ wishbone stop --help
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.
```

- **show**

```
$ wishbone show --help
usage: wishbone show [-h] (--docs DOCS | --code CODE)

Shows information about a component.

optional arguments:
  -h, --help   show this help message and exit
  --docs DOCS  Shows the documentation of the component.
  --code CODE  Shows the code of the refered component.
```

Behind the scenes, the Wishbone bootstrap process automatically makes a couple of configurations:

- All the `_metrics` queues of all modules are connected a `wishbone.module.flow.funnel` instance called `_metrics` from where the user can optionally connect modules for further metric processing.
- All the `_logs` queues of all modules are connected a `wishbone.module.flow.funnel` instance called `_logs`. The `_logs` module instance is then connected to a `wishbone.module.flow.queueselect` instance called `_logs_filter` in order to filter out the logs according to the `--log_level` value.
- If the `Wishbone` server is started with `--fork` then `_logs_filter.pass` is connected to a `wishbone.module.output.syslog` instance called `_logs_syslog` which has the effect all modules logs are written to syslog.
- If the `Wishbone` server is started without `--fork` then `_logs_filter.pass` is connected to a `wishbone.module.output.stdout` instances called `_logs_stdout` which has the effect all modules logs are written to stdout.

The following bootstrap file:

Listing 1: hello_world.yaml

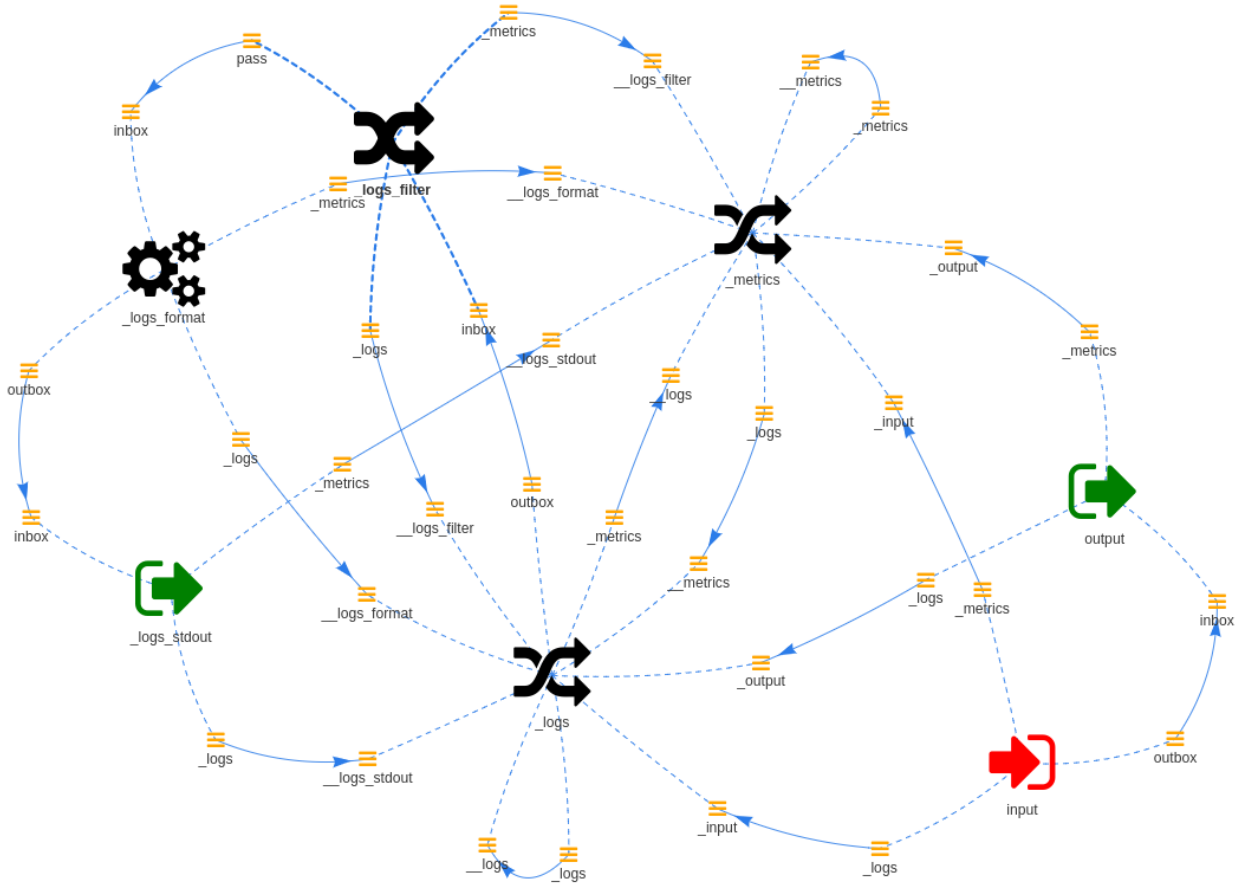
```
modules:
  input:
    module: wishbone.module.input.generator
    arguments:
```

(continues on next page)

(continued from previous page)

```
      payload: hello world

  output:
    module: wishbone.module.output.stdout

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

Generates following setup when run in foreground:



This includes the user defined `input.outbox -> output.inbox` connections including the auto-generated *metrics* and *logs* modules.

2.4 Python

The most prominent Python parts invovled in writing a Wishbone server are:

`wishbone.event.Event` Encapsulates the data traveling between the modules.

`wishbone.actor.Actor` A baseclass for all module types.

`wishbone.actorconfig.ActorConfig` Determines the characteristics of the actor based module.

`wishbone.router.default.Default` Holds all module instances and organises the stream of events between them

wishbone.componentmanager.ComponentManager A convenience function to easily load components by entry point.

Consider the following "hello world" example which demonstrates how to setup a Wishbone instance directly from Python:

```
from wishbone.actorconfig import ActorConfig
from wishbone.router.default import Default
from wishbone.componentmanager import ComponentManager


def main():

    router = Default()

    f = ComponentManager().getComponentByName("wishbone.function.module.append")

    f_instance = f(
        data="you_are_tagged",
        destination="tags"
    )

    router.registerModule(
        module="wishbone.module.input.generator",
        actor_config=ActorConfig(
            name='input'
        ),
        arguments={
            "payload"
        }
    )

    router.registerModule(
        module="wishbone.module.output.stdout",
        actor_config=ActorConfig(
            name='output',
            module_functions={
                "inbox": [
                    f_instance
                ]
            }
        ),
        arguments={
            "selection": None
        }
    )

    router.connectQueue('input.outbox', 'output.inbox')
    router.start()
    router.block()


if __name__ == '__main__':
    main()
```

2.5 Examples & Recipes

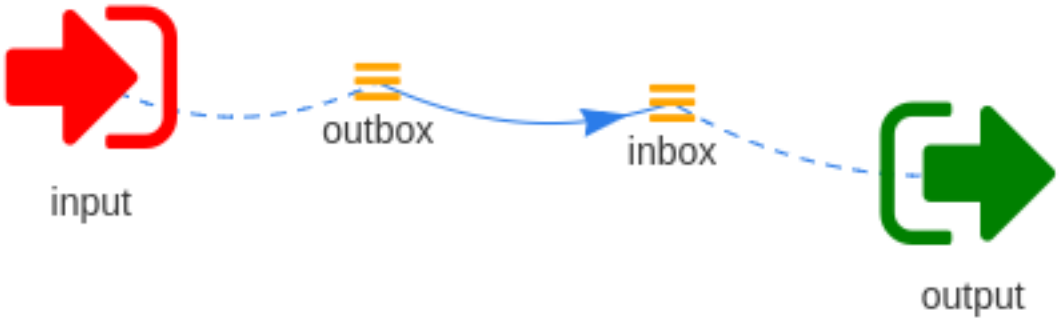
2.5.1 Hello World

The obligatory *hello world* example:

Listing 2: hello_world.yaml

```
modules:
  input:
    module: wishbone.module.input.generator
    arguments:
      payload: hello world

  output:
    module: wishbone.module.output.stdout

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

The setup diagram:



Executing the server:

```
$ wishbone start --config hello_world.yaml
Instance started in foreground with pid 21669
{'cloned': False, 'bulk': False, 'data': 'hello world', 'errors': {}, 'tags': [],
↪'timestamp': 1510156238.5204272, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': '69f85535-2502-4eee-b58c-55f21293057f'}
{'cloned': False, 'bulk': False, 'data': 'hello world', 'errors': {}, 'tags': [],
↪'timestamp': 1510156239.5209413, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': '5049c94c-bc57-4581-a972-85a330f190f6'}
{'cloned': False, 'bulk': False, 'data': 'hello world', 'errors': {}, 'tags': [],
↪'timestamp': 1510156240.522063, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid':
↪'e17a8fff-a99b-481c-91c6-3be62c0b015f'}
{'cloned': False, 'bulk': False, 'data': 'hello world', 'errors': {}, 'tags': [],
↪'timestamp': 1510156241.5230553, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': '317835eb-7cc5-4cbb-8778-eca8bdf7e280'}
{'cloned': False, 'bulk': False, 'data': 'hello world', 'errors': {}, 'tags': [],
↪'timestamp': 1510156242.5241177, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': '9b73cdb1-54fb-434c-9114-d698afb936db'}
{'cloned': False, 'bulk': False, 'data': 'hello world', 'errors': {}, 'tags': [],
↪'timestamp': 1510156243.5250602, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': '7b2a65fa-30aa-4fab-9c7c-058f1ddcd92c'}
```

(continues on next page)

(continued from previous page)

```
{'cloned': False, 'bulk': False, 'data': 'hello world', 'errors': {}, 'tags': [],
↪'timestamp': 1510156244.5259635, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': 'eb1f5177-33ac-4daa-b41d-f0a82b2b2375'}
{'cloned': False, 'bulk': False, 'data': 'hello world', 'errors': {}, 'tags': [],
↪'timestamp': 1510156245.5269322, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': '91c4e3e4-b8ce-40b8-81a7-d34581b8bd79'}
{'cloned': False, 'bulk': False, 'data': 'hello world', 'errors': {}, 'tags': [],
↪'timestamp': 1510156246.527863, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid':
↪'ba920b77-7477-424b-b4e6-a7075764d55f'}
{'cloned': False, 'bulk': False, 'data': 'hello world', 'errors': {}, 'tags': [],
↪'timestamp': 1510156247.5287836, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': '75f0df63-fdec-415d-abcc-b2f1dbaa4b6f'}
^C2017-11-08T15:50:48.4685+00:00 wishbone[21669] informational input: Received stop.␣
↪Initiating shutdown.
2017-11-08T15:50:48.4688+00:00 wishbone[21669] informational output: Received stop.␣
↪Initiating shutdown.
$
```

2.5.2 Creating a template function

Creating your own template function is easy.

In this example we will write a *fizzbuzz* template function which returns the system's uptime in seconds.

- Your class must base `wishbone.function.template.TemplateFunction`
- Your class must have a `get()` method which actually returns the desired data.
- Write a terse docstring as this will be used when issuing `wishbone show --docs wishbone_external.function.template.uptime`.
- Install your template function along a similar entrypoint in `setup.py`:

```
entry_points={
  'wishbone_external.function.template': [
      'uptime = wishbone_external.function.template.uptime:Uptime'
  ]
}
```

Create a class

```
from wishbone.function.template import TemplateFunction
from uptime import uptime


class Uptime(TemplateFunction):
    '''
    Returns the uptime in seconds of the system.

    A Wishbone template function which returns the system's uptime in
    seconds.

    Args:
        None
    '''

    def get(self):
```

(continues on next page)

(continued from previous page)

```
        '''
        The function mapped to the template function.

        Args:
            None

        Returns:
            float: Uptime in seconds.
        '''

        return uptime()
```

2.5.3 Creating a module function

Creating a module function is just a matter of creating a simple class.

In this example we will create a module function which calculates the grand total of an itemized bill.

Some key points of a module function:

- Your class must base `wishbone.function.module.ModuleFunction`
- Your class must have a `do()` method which accepts the event and returns it modified.
- Write a terse docstring as this will be used when issuing `wishbone show --docs wishbone_external.function.module.grandtotal`.
- Install your template function along a similar entrypoint in `setup.py`:

```
entry_points={
  'wishbone_external.module.grandtotal': [
      'grandtotal = wishbone_external.module.grantotal:GrandTotal'
  ]
}
```

```
from wishbone.function.module import ModuleFunction


class GrandTotal(ModuleFunction):
    '''
    Calculates the grand total of all articles.

    A Wishbone module function which calculates the grand total of all the
    article items stored under ``articles``.

    Args:
        source (str): The source field where the articles are stored
        destination (str): The destination field where to write the total.
    '''

    def __init__(self, source='data.articles', destination='data.total'):

        self.source = source
        self.destination = destination

    def do(self, event):
        '''
```

(continues on next page)

(continued from previous page)

```
        The function mapped to the module function.

        Args:
            event (wishbone.event.Event): The Wishbone event.

        Returns:
            wishbone.event.Event: The modified event.
        '''

        total = 0
        for article, price in event.get(self.source).items():
            total += int(price)

        event.set(total, self.destination)
        return event
```

The following bootstrap YAML file demonstrates how the `grandtotal` module can be used:

```
module_functions:
  make_grand_total:
    function: wishbone_external.function.module.grandtotal

template_functions:
  get_price:
    function: wishbone.function.template.random_integer
    arguments:
      minimum: 1
      maximum: 100

modules:
  input:
    module: wishbone.module.input.generator
    arguments:
      payload:
        articles:
          article_1: "{{ get_price() }}"
          article_2: "{{ get_price() }}"
          article_3: "{{ get_price() }}"
          article_4: "{{ get_price() }}"
          article_5: "{{ get_price() }}"

  output:
    module: wishbone.module.output.stdout
    functions:
      inbox:
        - make_grand_total
    arguments:
        selection: .

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

The output looks like:

```
$ wishbone start --config module_function_grandtotal.yaml --no-fork
Instance started in foreground with pid 29585
2017-10-29T19:56:51.7004+00:00 wishbone[29585] debug input: Connected queue input._
↪logs to _logs._input
```

(continues on next page)

(continued from previous page)

```
2017-10-29T19:56:51.7006+00:00 wishbone[29585] debug input: Connected queue input._
↪metrics to _metrics._input
2017-10-29T19:56:51.7007+00:00 wishbone[29585] debug input: Connected queue input.
↪outbox to output.inbox
2017-10-29T19:56:51.7009+00:00 wishbone[29585] debug input: preHook() found, executing
2017-10-29T19:56:51.7010+00:00 wishbone[29585] debug input: Started with max queue␣
↪size of 100 events and metrics interval of 10 seconds.
2017-10-29T19:56:51.7011+00:00 wishbone[29585] debug output: Connected queue output._
↪logs to _logs._output
2017-10-29T19:56:51.7013+00:00 wishbone[29585] debug output: Connected queue output._
↪metrics to _metrics._output
2017-10-29T19:56:51.7014+00:00 wishbone[29585] debug output: preHook() found,␣
↪executing
2017-10-29T19:56:51.7015+00:00 wishbone[29585] debug output: Started with max queue␣
↪size of 100 events and metrics interval of 10 seconds.
2017-10-29T19:56:51.7016+00:00 wishbone[29585] debug output: Function 'consume' has␣
↪been registered to consume queue 'inbox'
{'cloned': False, 'bulk': False, 'data': {'articles': {'article_1': '39', 'article_2
↪': '35', 'article_3': '64', 'article_4': '44', 'article_5': '71'}, 'total': 253},
↪'errors': {}, 'tags': [], 'timestamp': 1509307012.7014496, 'tmp': {}, 'ttl': 253,
↪'uuid_previous': [], 'uuid': 'b42ab53f-9f41-4ad4-814e-2c227537e4fe'}
{'cloned': False, 'bulk': False, 'data': {'articles': {'article_1': '26', 'article_2
↪': '95', 'article_3': '58', 'article_4': '10', 'article_5': '72'}, 'total': 261},
↪'errors': {}, 'tags': [], 'timestamp': 1509307013.702464, 'tmp': {}, 'ttl': 253,
↪'uuid_previous': [], 'uuid': '94a854a6-8400-4a36-b790-070ee0bd5c2c'}
{'cloned': False, 'bulk': False, 'data': {'articles': {'article_1': '36', 'article_2
↪': '10', 'article_3': '96', 'article_4': '89', 'article_5': '82'}, 'total': 313},
↪'errors': {}, 'tags': [], 'timestamp': 1509307014.7034726, 'tmp': {}, 'ttl': 253,
↪'uuid_previous': [], 'uuid': '020e5aed-50fd-46f9-a7a4-495b8a474984'}
```

2.5.4 Creating a module

Contents

The following example module evaluates whether an event containing an integer value is between a *minimum* and a

maximum. Depending on whether the value is higher or lower the event will be routed to the appropriate queue.

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

from wishbone.module import FlowModule


class HigherLower(FlowModule):
    '''
    **Checks whether an integer is higher or lower than the defined value.**

    Checks whether an event value is higher, lower or equal to the defined baseline.
    Depending on the outcome, the event will be submitted to the appropirate queue.

    Parameters::

        - base(int)(100)
           |  The value to compare against.

        - value(int)(100)
           |  The value to compare.

    Queues::

        - inbox
           |  Incoming messages

        - higher
           |  Events with a higher value than ``value`` are submitted to this
           |  queue.

        - lower
           |  Events with a lower value than ``value`` are submitted to this
           |  queue.

        - equal
           |  Events with an equal value to ``value`` are submitted to this
           |  queue.
    '''

    def __init__(self, actor_config, base=100, value=100):
        FlowModule.__init__(self, actor_config)

        self.pool.createQueue("inbox")
        self.pool.createQueue("higher")
        self.pool.createQueue("lower")
        self.pool.createQueue("equal")
        self.registerConsumer(self.consume, "inbox")

    def consume(self, event):

        if not isinstance(event.data, int):
            raise TypeError("Event data is not type integer")

        if event.kwargs.value > event.kwargs.base:
            self.submit(event, self.pool.queue.higher)
        elif event.kwargs.value < event.kwargs.base:
            self.submit(event, self.pool.queue.lower)
```

(continues on next page)

(continued from previous page)

```
        else:
            self.submit(event, self.pool.queue.equal)
```

Document the module

The docstring (line 6-29) contains the module's description. It's encouraged to document your module in a similar fashion. The content of the docstring can be accessed on CLI using the *wishbone show* command.

```
$ wishbone show --docs wishbone_contrib.module.flow.higherlower
```

Base the correct class

A module should base (line 5) one of the four *four modules types.*

Since this example module is applying logic of some sort to its incoming events to decide which queue to submit the event to without actually modifying its payload we choose type *flow module*.

The first parameter of a Wishbone module must **always** be `actor_config` which on its turn is used to initialize the base class (line 32).

The `actor_config` parameter is a `wishbone.actorconfig.ActorConfig` instance which configures the module's behavior within the Wishbone framework.

Creating queues

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

Besides for the default `_failed` and `_success` queues, it's left up to the developer to make sure the necessary queues are created.

Creating queues is done by invoking the `wishbone.queue.QueuePool.createQueue` (line 43-46). In the case of this specific *flow module* we will create `inbox`, `higher`, `lower`, `equal`.

Registering a function

The modules incoming events are to its `inbox` queue. We need to register a function which takes care of processing the events in the `inbox` queue. Once we have registered such a function, Wishbone will take care of draining the queue and applying the registered `function` to all its events.

Registering such a function is done by applying `wishbone.actor.Actor.registerConsumer()` (line 47). The function should have 1 parameter accepting the `wishbone.event.Event` instances.

Handling dynamic parameter values

This is an important topic. Somehow the module needs to know where in the event it can find the integer value to work with.

There are 2 different approaches to this:

Define the field as a parameter

In this case we make the field where to find the integer configurable. The module's parameters could look like this:

```
def __init__(self, actor_config, base=100, field="data"):
```

The `consume` function then could look something like this:

```
if event.get(self.kwargs.field) > event.kwargs.base:
    self.submit(event, self.pool.queue.higher)
elif event.get(self.kwargs.field) < event.kwargs.base:
    self.submit(event, self.pool.queue.lower)
else:
    self.submit(event, self.pool.queue.equal)
```

Define the value as a template value

This is the technique we use in this example

We can also pass a template as a parameter which fetches the desired value from the event. Each time an event enters a registered version, then Wishbone stores a rendered version of `self.kwargs` (the modules parameters) under `event.kwargs` using the content of the event itself.

So let's say that incoming events have following `wishbone.event.Event` format:

```
{
"bulk": false,
"cloned": false,
"data": 99,
"errors": {},
"tags": [],
"timestamp": 1515239271.0001013,
"tmp": {},
"ttl": 254,
"uuid": "1bb5301c-36d6-4a6e-b039-c310eb9a4d85",
"uuid_previous": []
}
```

We can have the bootstrap file initialize the module as such:

```
evaluate:
    module: wishbone_contrib.module.flow.higherlower
    arguments:
        base: 50
        value: '{{data}}'
```

Wishbone resolves the (Jinja2) template `{{data}}` then into the desired value and store it under `event.kwargs.value`. Hence we can do:

```
if event.kwargs.value > event.kwargs.base:
```

Submitting an 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 should be done by using `wishbone.actor.Actor.submit()` (line 55, 57, 59).

Dealing with errors

If an `exception` occurs inside the registered function then Wishbone will automatically submit the event to the module's default `_failed` queue. Therefor it is important to allow errors to raise. On the contrary, when the event has been handled without exceptions then it is also submitted to the modules `_success` queue.

Taking advantage of this behavior is useful to setup error handling constructions.

Provide an entrypoint

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

Example:

```
entry_points={
    'wishbone_contrib.module.flow': [
        'higherlower = higherlower:HigherLower'
    ]
}
```

This entrypoint definition allows Wishbone to import the module using `wishbone_contrib.module.flow.higherlower` in the boostrap file.

2.5.5 Using a template function

This example explains how to use a template function to feed module parameters a *dynamic* value. In this example we initialize `wishbone.function.template.choice` by setting its `wishbone.function.template.choice.Choice.array` parameter.

Coded in Python

```
from wishbone.actorconfig import ActorConfig
from wishbone.router.default import Default
from wishbone.componentmanager import ComponentManager


def main():

    router = Default()

    f = ComponentManager().getComponentByName("wishbone.function.template.choice")
    f_instance = f(["one", "two", "three"])

    router.registerModule(
        module="wishbone.module.input.generator",
        actor_config=ActorConfig(
            name='input',
            template_functions={
                "gimmeNumber": f_instance
            }
        ),
        arguments={
            "payload": "The value '{{gimmeNumber()}}' is chosen."
        }
    )
```

(continues on next page)

(continued from previous page)

```
    router.registerModule(
        module="wishbone.module.output.stdout",
        actor_config=ActorConfig(
            name='output'
        )
    )

    router.connectQueue('input.outbox', 'output.inbox')
    router.start()
    router.block()


if __name__ == '__main__':
    main()
```

Bootstrap File

The following bootstrap file does exactly the same as the above `python` version:

```
---
template_functions:
  gimmeNumber:
    function: wishbone.function.template.choice
    arguments:
      array:
        - one
        - two
        - three

modules:
  input:
    module: wishbone.module.input.generator
    arguments:
      payload: The value '{{gimmeNumber()}}' is chosen.

  output:
    module: wishbone.module.output.stdout

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

Server output:

The server can be started on CLI using the `wishbone` executable:

```
$ wishbone start --config boostrap.yaml --nofork
Instance started in foreground with pid 32206
2017-10-27T10:58:57.6725+00:00 wishbone[32206] debug input: Connected queue input._
↪logs to _logs._input
2017-10-27T10:58:57.6727+00:00 wishbone[32206] debug input: Connected queue input._
↪metrics to _metrics._input
2017-10-27T10:58:57.6728+00:00 wishbone[32206] debug input: Connected queue input.
↪outbox to output.inbox
2017-10-27T10:58:57.6729+00:00 wishbone[32206] debug input: preHook() found, executing
2017-10-27T10:58:57.6731+00:00 wishbone[32206] debug input: Started with max queue␣
↪size of 100 events and metrics interval of 10 seconds.
```

(continues on next page)

(continued from previous page)

```
2017-10-27T10:58:57.6732+00:00 wishbone[32206] debug output: Connected queue output._
↪logs to _logs._output
2017-10-27T10:58:57.6733+00:00 wishbone[32206] debug output: Connected queue output._
↪metrics to _metrics._output
2017-10-27T10:58:57.6734+00:00 wishbone[32206] debug output: preHook() found,␣
↪executing
2017-10-27T10:58:57.6736+00:00 wishbone[32206] debug output: Started with max queue␣
↪size of 100 events and metrics interval of 10 seconds.
2017-10-27T10:58:57.6737+00:00 wishbone[32206] debug output: Function 'consume' has␣
↪been registered to consume queue 'inbox'
The value 'one' is chosen.
The value 'three' is chosen.
The value 'three' is chosen.
The value 'two' is chosen.
```

2.5.6 Using a module function

This example explains how to use a module function adding a tag the events passing through.

Coded in Python

```
from wishbone.actorconfig import ActorConfig
from wishbone.router.default import Default
from wishbone.componentmanager import ComponentManager


def main():

    router = Default()

    f = ComponentManager().getComponentByName("wishbone.function.module.append")
    f_instance = f(
        data="you_are_tagged",
        destination="tags"
    )

    router.registerModule(
        module="wishbone.module.input.generator",
        actor_config=ActorConfig(
            name='input'
        )
    )

    router.registerModule(
        module="wishbone.module.output.stdout",
        actor_config=ActorConfig(
            name='output',
            module_functions={
                "inbox": [
                    f_instance
                ]
            }
        ),
        arguments={
            "selection": "."
```

(continues on next page)

(continued from previous page)

```
        }
    )

    router.connectQueue('input.outbox', 'output.inbox')
    router.start()
    router.block()


if __name__ == '__main__':
    main()
```

Bootstrap File

The following bootstrap file does exactly the same as the above `python` version:

```
---
module_functions:
  tagit:
    function: wishbone.function.module.append
    arguments:
        data: you_are_tagged
        destination: tag

modules:
  input:
    module: wishbone.module.input.generator

  output:
    module: wishbone.module.output.stdout
    module_functions:
        - tagit
    arguments:
        selection: .

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

Server output:

The server can be started on CLI using the `wishbone` executable:

```
$ wishbone start --config boostrap.yaml --nofork
Instance started in foreground with pid 16695
2017-10-29T17:40:30.1223+00:00 wishbone[16695] debug input: Connected queue input._
↪logs to _logs._input
2017-10-29T17:40:30.1224+00:00 wishbone[16695] debug input: Connected queue input._
↪metrics to _metrics._input
2017-10-29T17:40:30.1226+00:00 wishbone[16695] debug input: Connected queue input.
↪outbox to output.inbox
2017-10-29T17:40:30.1227+00:00 wishbone[16695] debug input: preHook() found, executing
2017-10-29T17:40:30.1229+00:00 wishbone[16695] debug input: Started with max queue␣
↪size of 100 events and metrics interval of 10 seconds.
2017-10-29T17:40:30.1230+00:00 wishbone[16695] debug output: Connected queue output._
↪logs to _logs._output
2017-10-29T17:40:30.1231+00:00 wishbone[16695] debug output: Connected queue output._
↪metrics to _metrics._output
```

(continues on next page)

(continued from previous page)

```
2017-10-29T17:40:30.1232+00:00 wishbone[16695] debug output: preHook() found,␣
↪executing
2017-10-29T17:40:30.1234+00:00 wishbone[16695] debug output: Started with max queue␣
↪size of 100 events and metrics interval of 10 seconds.
2017-10-29T17:40:30.1235+00:00 wishbone[16695] debug output: Function 'consume' has␣
↪been registered to consume queue 'inbox'
{'cloned': False, 'bulk': False, 'data': 'test', 'errors': {}, 'tags': ['you_are_
↪tagged'], 'timestamp': 1509298831.1225557, 'tmp': {}, 'ttl': 253, 'uuid_previous':␣
↪[], 'uuid': '8d1489f7-7d55-4a26-8114-69c68c7b5ecf'}
{'cloned': False, 'bulk': False, 'data': 'test', 'errors': {}, 'tags': ['you_are_
↪tagged'], 'timestamp': 1509298832.124007, 'tmp': {}, 'ttl': 253, 'uuid_previous':␣
↪[], 'uuid': '854f31a4-cf96-446e-9712-a4e3d5a8b38b'}
{'cloned': False, 'bulk': False, 'data': 'test', 'errors': {}, 'tags': ['you_are_
↪tagged'], 'timestamp': 1509298833.1251073, 'tmp': {}, 'ttl': 253, 'uuid_previous':␣
↪[], 'uuid': '76fec0c3-0690-4683-90aa-ae5d7c5b6b34'}
{'cloned': False, 'bulk': False, 'data': 'test', 'errors': {}, 'tags': ['you_are_
↪tagged'], 'timestamp': 1509298834.1261678, 'tmp': {}, 'ttl': 253, 'uuid_previous':␣
↪[], 'uuid': 'a50af14d-cc7c-4449-864b-92a86d727de0'}
{'cloned': False, 'bulk': False, 'data': 'test', 'errors': {}, 'tags': ['you_are_
↪tagged'], 'timestamp': 1509298835.1271603, 'tmp': {}, 'ttl': 253, 'uuid_previous':␣
↪[], 'uuid': '4bcfba25-e700-484f-8fee-73ac77597e3f'}
{'cloned': False, 'bulk': False, 'data': 'test', 'errors': {}, 'tags': ['you_are_
↪tagged'], 'timestamp': 1509298836.1281745, 'tmp': {}, 'ttl': 253, 'uuid_previous':␣
↪[], 'uuid': '5cb0f80e-742a-47fa-a971-f10744467358'}
```

2.5.7 Using a protocol decoder

```
from wishbone.actorconfig import ActorConfig
from wishbone.router.default import Default
from wishbone.componentmanager import ComponentManager


def main():

    c = ComponentManager()
    protocol = c.getComponentByName("wishbone.protocol.decode.json")()

    router = Default()

    router.registerModule(
        module="wishbone.module.input.generator",
        actor_config=ActorConfig(
            name='input',
            protocol=protocol
        ),
        arguments={
            "payload": '{"one": 1}'
        }
    )

    router.registerModule(
        module="wishbone.module.output.stdout",
        actor_config=ActorConfig(
            name='output',
        ),
```

(continues on next page)

(continued from previous page)

```
        arguments={
            "selection": "."
        }
    )

    router.connectQueue('input.outbox', 'output.inbox')
    router.start()
    router.block()


if __name__ == '__main__':
    main()
```

The equivalent using a bootstrap file:

```
protocols:
  json:
    protocol: wishbone.protocol.decode.json

modules:
  input:
    module: wishbone.module.input.generator
    protocol: json
    arguments:
      payload: '{"one": 1}'

  output:
    module: wishbone.module.output.stdout
    arguments:
      selection: .

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

The output:

```
$ wishbone start --config demo_decode.yaml --nofork

Instance started in foreground with pid 8899
2017-11-01T13:16:59.6693+00:00 wishbone[8899] debug _logs: Connected queue _logs._
↪logs to _logs.__logs
2017-11-01T13:16:59.6695+00:00 wishbone[8899] debug _logs: Connected queue _logs._
↪metrics to _metrics.__logs
2017-11-01T13:16:59.6697+00:00 wishbone[8899] debug _logs: Module instance '_logs'␣
↪has no queue '__metrics' so auto created.
2017-11-01T13:16:59.6698+00:00 wishbone[8899] debug _logs: Module instance '_logs'␣
↪has no queue '_input' so auto created.
2017-11-01T13:16:59.6699+00:00 wishbone[8899] debug _logs: Module instance '_logs'␣
↪has no queue '_output' so auto created.
... snip ...
{'cloned': False, 'bulk': False, 'data': {'one': 1}, 'errors': {}, 'tags': [],
↪'timestamp': 1509542220.6696804, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': '4e754cec-402f-48b6-8a25-af3afeeb65fb'}
{'cloned': False, 'bulk': False, 'data': {'one': 1}, 'errors': {}, 'tags': [],
↪'timestamp': 1509542221.670773, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid':
↪'7cc500bc-750f-476a-b7b3-4d1adb522218'}
{'cloned': False, 'bulk': False, 'data': {'one': 1}, 'errors': {}, 'tags': [],
↪'timestamp': 1509542222.6718802, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': 'ede9fc76-f5d7-4102-95ac-c7a3aacebfd7'}
```

(continues on next page)

(continued from previous page)

```
{'cloned': False, 'bulk': False, 'data': {'one': 1}, 'errors': {}, 'tags': [],
↪'timestamp': 1509542223.672989, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid':
↪'06291f44-10ba-4194-8e9b-4e6817fae5d2'}
{'cloned': False, 'bulk': False, 'data': {'one': 1}, 'errors': {}, 'tags': [],
↪'timestamp': 1509542224.6740425, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': '294d24c7-e713-4e8b-be88-c14322917e96'}
{'cloned': False, 'bulk': False, 'data': {'one': 1}, 'errors': {}, 'tags': [],
↪'timestamp': 1509542225.6750607, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': '8493d02a-2f55-468e-900c-a5286e842f7a'}
{'cloned': False, 'bulk': False, 'data': {'one': 1}, 'errors': {}, 'tags': [],
↪'timestamp': 1509542226.6760375, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': '369eebe5-2bb1-4c71-ba73-c3be78915db2'}
```

2.5.8 Using a protocol encoder

```
from wishbone.actorconfig import ActorConfig
from wishbone.router.default import Default
from wishbone.componentmanager import ComponentManager


def main():

    c = ComponentManager()
    protocol = c.getComponentByName("wishbone.protocol.encode.json")()

    router = Default()

    router.registerModule(
        module="wishbone.module.input.generator",
        actor_config=ActorConfig(
            name='input',
        ),
        arguments={
            "payload": {"one": 1, "two": 2}
        }
    )

    router.registerModule(
        module="wishbone.module.output.stdout",
        actor_config=ActorConfig(
            name='output',
            protocol=protocol
        ),
    )

    router.connectQueue('input.outbox', 'output.inbox')
    router.start()
    router.block()


if __name__ == '__main__':
    main()
```

The equivalent using a bootstrap file:

```
protocols:
  json:
    protocol: wishbone.protocol.encode.json

modules:
  input:
    module: wishbone.module.input.generator
    arguments:
      payload:
        one: 1
        two: 2

  output:
    module: wishbone.module.output.stdout
    protocol: json

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

The output:

```
$ wishbone start --config demo_decode.yaml --nofork

Instance started in foreground with pid 8899
2017-11-01T13:16:59.6693+00:00 wishbone[8899] debug _logs: Connected queue _logs._
↪logs to _logs.__logs
2017-11-01T13:16:59.6695+00:00 wishbone[8899] debug _logs: Connected queue _logs._
↪metrics to _metrics.__logs
2017-11-01T13:16:59.6697+00:00 wishbone[8899] debug _logs: Module instance '_logs'␣
↪has no queue '__metrics' so auto created.
2017-11-01T13:16:59.6698+00:00 wishbone[8899] debug _logs: Module instance '_logs'␣
↪has no queue '_input' so auto created.
2017-11-01T13:16:59.6699+00:00 wishbone[8899] debug _logs: Module instance '_logs'␣
↪has no queue '_output' so auto created.
... snip ...
{"one": 1, "two": 2}
{"one": 1, "two": 2}
{"one": 1, "two": 2}
{"one": 1, "two": 2}
```

2.5.9 Read data using inotify

Some modules need read data from disk. Wishbone comes with the `wishbone.module.input.inotify` which can be setup to listen for filesystem changes required to reload your data from disk.

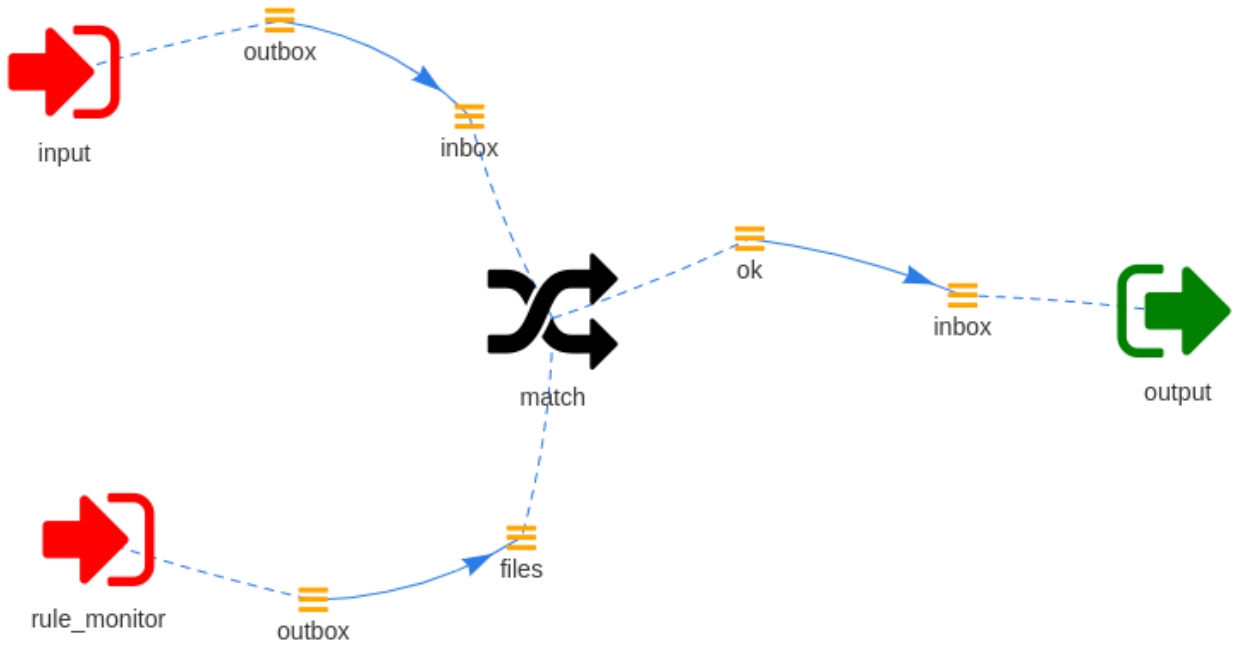
The `wishbone.module.queueselect.QueueSelect` is a module which can read its conditions from disk. Obviously when these rules change they have to be reloaded. Instead of having to build the file monitoring/reload functionality its easier to foresee a queue which can process the events indicating files have changed.

The following yaml bootstrap file demonstrates how `wishbone.module.input.inotify` can be used to feed a module changes to files so it can respond to that accordingly:

Note: Obviously the module receiving the inotify events needs to have specific support for that.

```
modules:
  input:
    module: wishbone.module.input.generator
    arguments:
        payload: ok

  rule_monitor:
    module: wishbone.module.input.inotify
    arguments:
      paths:
        "/var/tmp/rules": [
            "IN_CREATE",
            "IN_CLOSE_WRITE",
            "IN_DELETE"
        ]
      glob_pattern: "*.yaml"

  match:
    module: wishbone.module.flow.queueselect
    arguments:
      templates:
        - name: test
          queue: >
            {{ 'ok' if data == 'ok' }}

  output:
    module: wishbone.module.output.stdout

routingtable:
  - input.outbox        -> match.inbox
  - match.ok            -> output.inbox

  - rule_monitor.outbox -> match.files
```

The bootstrap file translates into following diagram:



2.5.10 Handling logs and metrics

When bootstrapping an instance using the `wishbone` executable and `wishbone.config.configfile.ConfigFile` is used to generate a `router` configuration.

Review the `wishbone.config.configfile.ConfigFile` docstring to read about the parts which are configured automatically.

Shipping metrics

Wishbone metric events are just plain Wishbone events.

```
{'cloned': False, 'bulk': False, 'data': {'time': 1511013563.2159407, 'type':
↪'wishbone', 'source': 'indigo', 'name': 'module.input.queue._metrics.size', 'value
↪': 0, 'unit': '', 'tags': ()}, 'errors': {}, 'tags': [], 'timestamp': 1511013563.
↪2159421, 'tmp': {}, 'ttl': 252, 'uuid_previous': [], 'uuid': '76b331e8-d088-4002-
↪b772-0df613c6f757'}
{'cloned': False, 'bulk': False, 'data': {'time': 1511013563.2159586, 'type':
↪'wishbone', 'source': 'indigo', 'name': 'module.input.queue._metrics.in_total',
↪'value': 35, 'unit': '', 'tags': ()}, 'errors': {}, 'tags': [], 'timestamp':␣
↪1511013563.2159598, 'tmp': {}, 'ttl': 252, 'uuid_previous': [], 'uuid': 'ccabd8ac-
↪420f-4e6d-a89a-b53351e887e1'}
{'cloned': False, 'bulk': False, 'data': {'time': 1511013563.215973, 'type': 'wishbone
↪', 'source': 'indigo', 'name': 'module.input.queue._metrics.out_total', 'value': 35,
↪ 'unit': '', 'tags': ()}, 'errors': {}, 'tags': [], 'timestamp': 1511013563.215974,
↪'tmp': {}, 'ttl': 252, 'uuid_previous': [], 'uuid': '16717619-c525-4e33-9f7d-
↪35bdf42819be'}
{'cloned': False, 'bulk': False, 'data': {'time': 1511013563.2160392, 'type':
↪'wishbone', 'source': 'indigo', 'name': 'module.input.queue._metrics.in_rate',
↪'value': 3.4852626619479707, 'unit': '', 'tags': ()}, 'errors': {}, 'tags': [],
↪'timestamp': 1511013563.2160406, 'tmp': {}, 'ttl': 252, 'uuid_previous': [], 'uuid
↪': '2d80683d-100b-45d9-b8a5-68cff227bbae'}
```

(continues on next page)

(continued from previous page)

```
{'cloned': False, 'bulk': False, 'data': {'time': 1511013563.216054, 'type': 'wishbone
↪', 'source': 'indigo', 'name': 'module.input.queue._metrics.out_rate', 'value': 3.
↪485262248221764, 'unit': '', 'tags': ()}, 'errors': {}, 'tags': [], 'timestamp':␣
↪1511013563.2160554, 'tmp': {}, 'ttl': 252, 'uuid_previous': [], 'uuid': 'c3e3c916-
↪4b6d-4df4-845a-f2f71097d339'}
{'cloned': False, 'bulk': False, 'data': {'time': 1511013563.2160676, 'type':
↪'wishbone', 'source': 'indigo', 'name': 'module.input.queue._metrics.dropped_total',
↪ 'value': 0, 'unit': '', 'tags': ()}, 'errors': {}, 'tags': [], 'timestamp':␣
↪1511013563.216069, 'tmp': {}, 'ttl': 252, 'uuid_previous': [], 'uuid': '69cf9aeb-
↪32ef-4b61-80e2-f45cdfe0ccb5'}
```

If we want to send Wishbone internal metrics to Graphite we use `wishbone.module.process.template` module in order to convert the above JSON into the desired Graphite format.

Note: The `wishbone.module.output.tcp` is an external module which has to be installed separately

```
modules:
  input:
    module: wishbone.module.input.generator
    arguments:
      payload: hello world

  output:
    module: wishbone.module.output.stdout

  metrics_graphite:
    module: wishbone.module.process.template
    arguments:
      templates:
        graphite: 'wishbone.{{data.name}} {{data.value}} {{data.time}}'

  metrics_pack:
    module: wishbone.module.process.pack
    arguments:
      bucket_size: 1500


  metrics_out:
    module: wishbone_external.module.output.tcp
    arguments:
      selection: graphite
      host: graphite-host.some.domain
      port: 2013

routingtable:
  - input.outbox -> output.inbox

  - _metrics.outbox         -> metrics_graphite.inbox
  - metrics_graphite.outbox -> metrics_pack.inbox
  - metrics_pack.outbox     -> metrics_out.inbox
```

- The `metrics_graphite` module instance *assembles* the fields of the events containing the metrics into a format Graphite understands.
- The `wishbone_external.module.output.tcp` opens and closes a connection per event. This is not very efficient hence we put a `wishbone.module.process.pack` module in front of the output in order to

submit buckets of 1500 metrics per connection.

Shipping logs

Instead of sending formatted logs to STDOUT or SYSLOG you might want to ship the Wishbone log events in JSON format to STDOUT.

You could do that using following bootstrap file:

```
protocols:
  json:
    protocol: wishbone.protocol.encode.json

modules:
  input:
    module: wishbone.module.input.generator
    arguments:
      payload: hello world

  output:
    module: wishbone.module.output.stdout

  logs_out:
    protocol: json
    module: wishbone.module.output.stdout
    arguments:
      selection: data

routingtable:
  - input.outbox -> output.inbox

  - _logs.outbox -> logs_out.inbox
```

Starting the Wishbone instance in foreground would give following result:

```
$ wishbone start --config hello_world_logs.yaml
Instance started in foreground with pid 11126
{"time": 1511022472.4646914, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs", "message": "Connected␣
↪queue _logs._logs to _logs.__logs"}
{"time": 1511022472.4647346, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs", "message": "Connected␣
↪queue _logs._metrics to _metrics.__logs"}
{"time": 1511022472.4647586, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs", "message": "Module␣
↪instance '_logs' has no queue '__logs_filter' so auto created."}
{"time": 1511022472.464825, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs", "message": "Module␣
↪instance '_logs' has no queue '__metrics' so auto created."}
{"time": 1511022472.4648945, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs", "message": "Module␣
↪instance '_logs' has no queue '_input' so auto created."}
{"time": 1511022472.464962, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs", "message": "Module␣
↪instance '_logs' has no queue '_output' so auto created."}
{"time": 1511022472.4650266, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs", "message": "Module␣
↪instance '_logs' has no queue '_logs_out' so auto created."}
```

(continues on next page)

(continued from previous page)


```
{"time": 1511022472.4651015, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs", "message": "Connected␣
↪queue _logs.outbox to logs_out.inbox"}
{"time": 1511022472.465119, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs", "message": "Following␣
↪template functions are available: strftime, epoch, version"}
{"time": 1511022472.4651282, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs", "message": "preHook()␣
↪found, executing"}
{"time": 1511022472.4651651, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs", "message": "Started with␣
↪max queue size of 100 events and metrics interval of 10 seconds."}
{"time": 1511022472.4688632, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs", "message": "Function
↪'consume' has been registered to consume queue '__logs'"}
{"time": 1511022472.46477, "identification": "wishbone", "event_id": null, "level": 7,
↪ "txt_level": "debug", "pid": 11126, "module": "_logs_filter", "message":
↪"Connected queue _logs_filter._logs to _logs.__logs_filter"}
{"time": 1511022472.464803, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs_filter", "message":
↪"Connected queue _logs_filter._metrics to _metrics.__logs_filter"}
{"time": 1511022472.4651802, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs_filter", "message":
↪"Following template functions are available: strftime, epoch, version"}
{"time": 1511022472.4651895, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs_filter", "message":
↪"preHook() found, executing"}
{"time": 1511022472.4652004, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs_filter", "message": "Module␣
↪has no preHook() method set."}
{"time": 1511022472.4652123, "identification": "wishbone", "event_id": null, "level":␣
↪7, "txt_level": "debug", "pid": 11126, "module": "_logs_filter", "message":
↪"Started with max queue size of 100 events and metrics interval of 10 seconds."}
```


2.5.11 A dead man's switch

The following Wishbone bootstrap file triggers an action in the case an event is expected but not received within a certain time window. This principle is called a dead man's switch.

For this setup we will make use of `wishbone.module.flow.fresh`.

Note: The `wishbone_contrib.module.input.httpserver` module is an external module and should be installed separately.

Consider following bootstrap file:

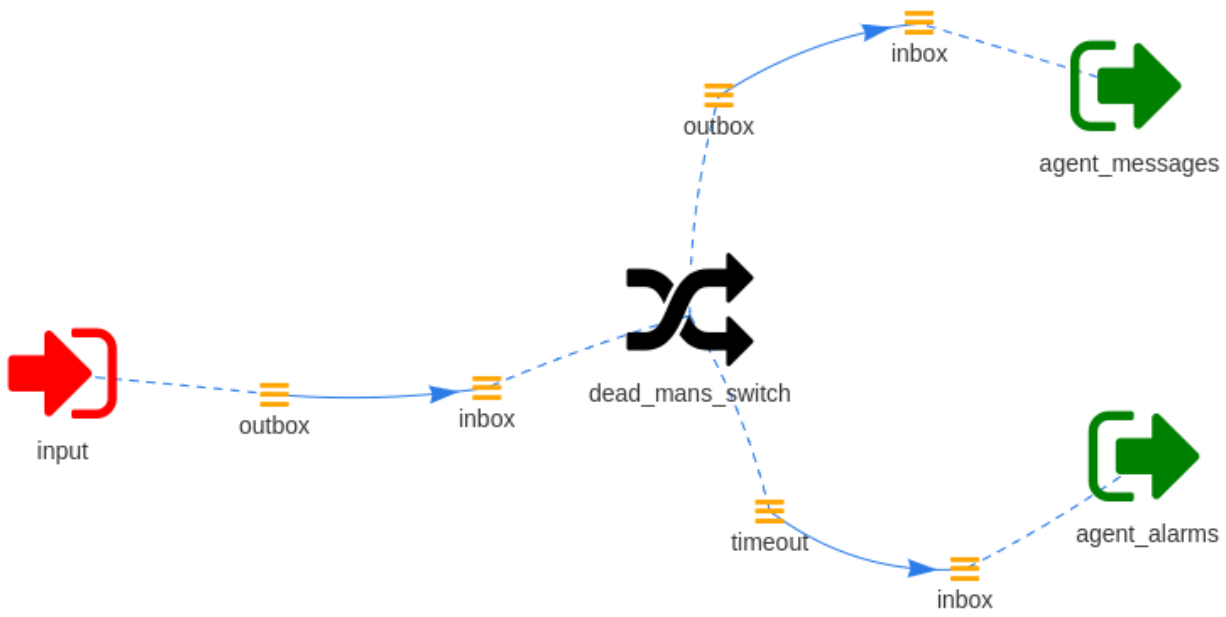
```
modules:
  input:
    module: wishbone_contrib.module.input.httpserver

  dead_mans_switch:
    module: wishbone.module.flow.fresh
    arguments:
      timeout: 10
```

(continues on next page)

(continued from previous page)

```
      timeout_payload:
        message: We didn't recieve the expected keepalive signal from agent X.
      recovery_payload:
        message: Agent X says hello.

  agent_messages:
    module: wishbone.module.output.stdout
    arguments:
      prefix: 'Agent message :'
      colorize: true
      foreground_color: GREEN
      payload: '{{data}}'

  agent_alarms:
    module: wishbone.module.output.stdout
    arguments:
      prefix: 'Agent status :'
      colorize: true
      foreground_color: RED
      payload: '{{data}}'


routingtable:
  - input.outbox             -> dead_mans_switch.inbox
  - dead_mans_switch.outbox  -> agent_messages.inbox
  - dead_mans_switch.timeout -> agent_alarms.inbox
```

The diagram of this bootstrap file:



The output looks like:

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

Instance started in foreground with pid 11971
```

(continues on next page)

(continued from previous page)

```
2017-11-19T15:58:10.4449+00:00 wishbone[11971] informational input: Serving on 0.0.0.
↪0:19283 with a connection poolsize of 1000.
Agent message :hello
2017-11-19T15:58:16.3481+00:00 None[11971] informational input: 127.0.0.1 - - [2017-
↪11-19 16:58:16] "PUT / HTTP/1.1" 200 103 0.000461
Agent message :hello
2017-11-19T15:58:19.3404+00:00 None[11971] informational input: 127.0.0.1 - - [2017-
↪11-19 16:58:19] "PUT / HTTP/1.1" 200 103 0.000323
Agent status :We didn't recieve the expected keepalive signal from agent X.
2017-11-19T15:58:29.4267+00:00 wishbone[11971] informational dead_mans_switch:␣
↪Timeout of 10 seconds expired.  Generated timeout event.
```

Whenever data is submitted into Wishbone's webserver `echo hello|curl -XPUT -d @- http://localhost:19283/` the timeout window resets and the message gets submitted to the `agent_messages` module instance.

When data is not submitted withint the predefined window of 10s then an internal event is generated and send to the `agent_alarms` output module.

2.5.12 Passing variables to Docker

`wishbone.function.template.environment` is a template function to access and use environment variables in your bootstrap file. When using the `wishbone` executable then the `env()` template function is loaded automatically.

This might be practical when using the containerized version of Wishbone.

Consider following bootstrap file:

```
modules:
  input:
    module: wishbone.module.input.generator
    arguments:
      payload: '{{env("message")}}'

  output:
    module: wishbone.module.output.stdout

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

We can bootstrap a Wishbone container using following command:

```
$ docker run -t -i --env message="hello world" -v $(pwd)/hello_world.yaml:/tmp/
↪bootstrap.yaml docker.io/smetj/wishbone:develop start --config /tmp/bootstrap.yaml
Instance started in foreground with pid 1
{'cloned': False, 'bulk': False, 'data': 'hello world', 'errors': {}, 'tags': [],
↪'timestamp': 1511299095.2465549, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': '2e6f6a44-50ef-4517-a727-0f3e0af0e6ab'}
{'cloned': False, 'bulk': False, 'data': 'hello world', 'errors': {}, 'tags': [],
↪'timestamp': 1511299096.2474735, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': '5c18eb80-5529-4f01-aa33-8f7286bc4769'}
{'cloned': False, 'bulk': False, 'data': 'hello world', 'errors': {}, 'tags': [],
↪'timestamp': 1511299097.2487144, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': '39edee41-bba1-4b81-9251-98b411f09918'}
{'cloned': False, 'bulk': False, 'data': 'hello world', 'errors': {}, 'tags': [],
↪'timestamp': 1511299098.2498908, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': '52f9709f-de1d-467d-8f53-f9c311e2bcc9'}
```

(continues on next page)

(continued from previous page)

```
{'cloned': False, 'bulk': False, 'data': 'hello world', 'errors': {}, 'tags': [],
↪'timestamp': 1511299099.2510643, 'tmp': {}, 'ttl': 253, 'uuid_previous': [], 'uuid
↪': '9443e1d0-dddc-41a8-bd8c-be291881876c'}
```

2.5.13 HTTP Fizzbuzz Example

This *example* Wishbone server accepts *JSON* data over http on the `/colors` endpoint and replies to the client with the defined `response` for it. The `categorize` module instance validates whether the value of `color` is either *red, green or blue* and forwards the event to the `requestbin` module instance if so. If not, the complete event is printed to STDOUT. The `requestbin` module submits the event to the defined `url` extended by the *requestbin_id* value defined by the user. After submitting the event successfully to the defined `url`, the complete event is printed to STDOUT.

Depending on the modules chosen you

Server

```
$ wishbone start --config fizzbuzz.yaml --nofork
Instance started in foreground with pid 25260
... snip ...
2017-09-30T14:18:46.7928+00:00 wishbone[25260] informational input: Serving on 0.0.0.
↪0:19283 with a connection poolsize of 1000.
```

Bootstrap file

```
---
protocols:
  json_decode:
    protocol: wishbone.protocol.decode.json
  json_encode:
    protocol: wishbone.protocol.encode.json

modules:
  input:
    module: wishbone_contrib.module.input.httpserver
    protocol: json_decode
    arguments:
      resource:
        colors:
          users: []
          tokens: []
          response: Hi '{{tmp.input.env.http_user_agent}}' on '{{tmp.input.env.remote_
↪addr}}'. Your id is '{{uuid}}'. Thank you for choosing Wishbone ;)'

  categorize:
    module: wishbone.module.flow.queueselect
    arguments:
      templates:
        - name: primary
          queue: >
            {{ 'primary' if data.color in ("red", "green", "blue") else 'not_primary'␣
↪}}
```

(continues on next page)

(continued from previous page)

```
          payload:
            greeting: Hello
            message: '{{data.color}} is an awesome choice'

  funnel:
    module: wishbone.module.flow.funnel

  requestbin:
    protocol: json_encode
    module: wishbone.module.output.http
    arguments:
      method: PUT
      url: 'https://requestb.in/{{data.requestbin_id}}'
      selection: tmp.categorize.payload

  stdout:
    module: wishbone.module.output.stdout
    protocol: json_encode
    arguments:
      selection: .

routingtable:
  - input.colors           -> categorize.inbox

  - categorize.primary     -> requestbin.inbox
  - categorize.not_primary -> funnel.not_primary

  - requestbin.success     -> funnel.requestbin

  - funnel.outbox          -> stdout.inbox
...
```

Client

```
$ curl -d '{"color":"red", "requestbin_id": "abcdefg"}' http://localhost:19283/colors
Hi 'curl/7.53.1' on '127.0.0.1'. Your id is 'd805df4c-816e-4af2-bb32-8454cae366aa'.
```

Server STDOUT after submitting event

```
{
  "cloned": true,
  "bulk": false,
  "data": {
    "color": "red",
    "requestbin_id": "abcdefg"
  },
  "errors": {},
  "tags": [],
  "timestamp": 1506791239.4684186,
  "tmp": {
    "input": {
      "remote_addr": "127.0.0.1",
      "request_method": "POST",
```

(continues on next page)

(continued from previous page)


```
      "user_agent": "curl/7.53.1",
      "queue": "colors",
      "username": "",
      "response": "Hi 'curl/7.53.1' on '127.0.0.1'. Your id is 'd805df4c-816e-4af2-
↪bb32-8454cae366aa'. Thank you for choosing Wishbone ;)"
    },
    "categorize": {
      "original_event_id": "94ff6c3b-3c83-41c5-b5b7-091f244e85a5",
      "queue": "primary",
      "payload": {
        "greeting": "Hello",
        "message": "red is an awesome choice"
      }
    },
    "requestbin": {
      "server_response": "ok",
      "status_code": 200,
      "url": "https://requestb.in/abcdefg",
      "method": "PUT",
      "useragent": "wishbone.module.output.http/3.0.0"
    }
  },
  "ttl": 251,
  "uuid_previous": [
    "94ff6c3b-3c83-41c5-b5b7-091f244e85a5"
  ],
  "uuid": "d805df4c-816e-4af2-bb32-8454cae366aa"
}
```


2.6 Miscellaneous

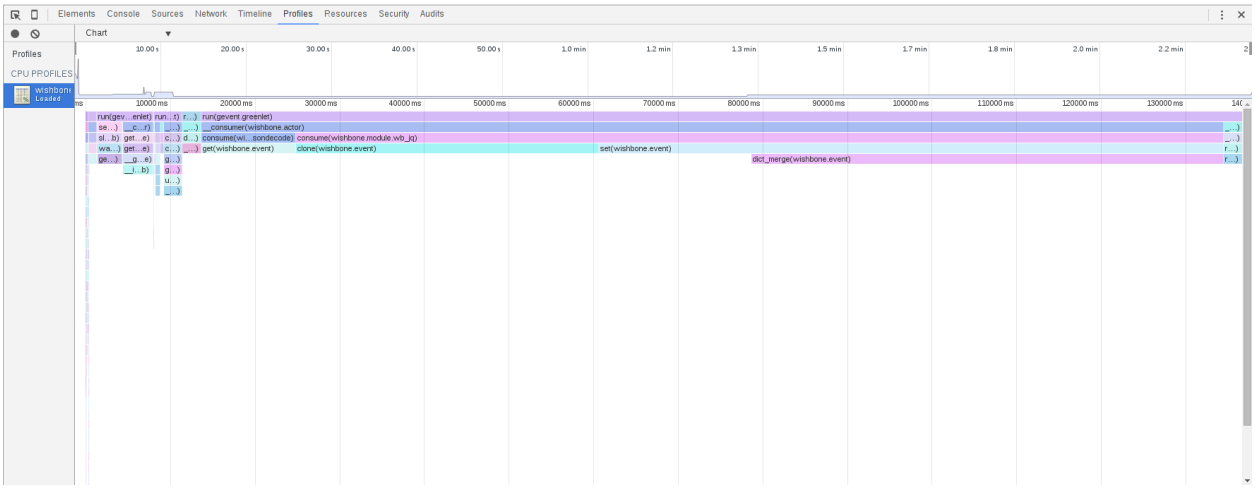
2.6.1 Profiling

You can profile a Wishbone server in order to locate performance issues by starting the `wishbone` executable using the `--profile` option.

```
$ wishbone start --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 directly into Chrome's "Developer Tools" for further analysis:



The *javascript* profiler is somewhat hidden in chrome. To open:

Options -> More Tools -> Developer Tools

Once *developer tools* is open select:

Options -> More Tools -> JavaScript profiler

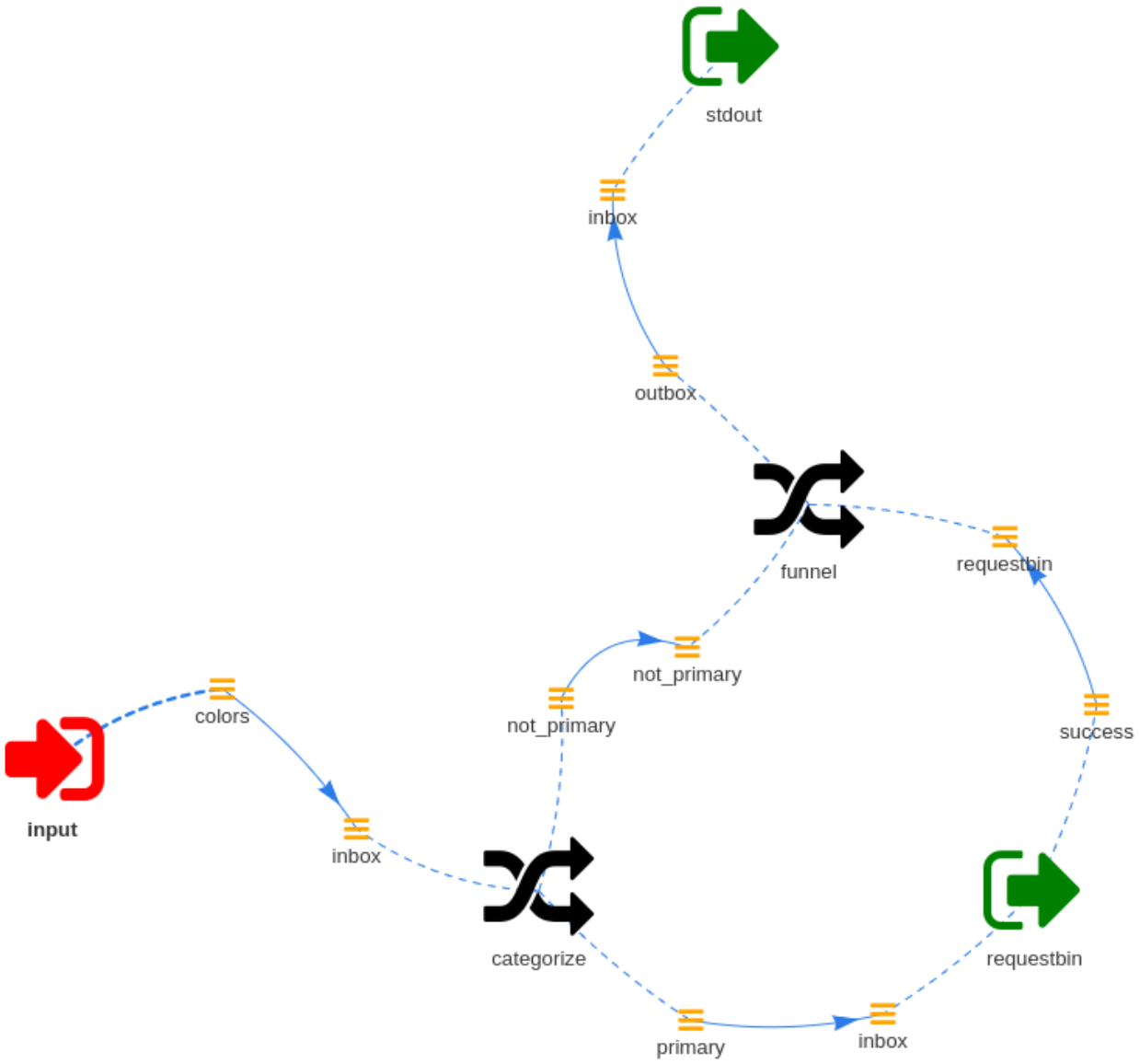
2.6.2 Topology

Sometimes it's useful to have a graphical representation of the bootstrap file showing the toplogy of the connected modules and queues.

For this you can start wishbone using the `--graph` switch.

```
$ wishbone start --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.



2.6.3 Caveats

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

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

  While you could store whatever Python object type in `wishbone.event.Event` it's really designed to store JSON like structures.

- **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.

Index