Junebug Documentation

Release 0.1.39

Praekelt Foundation

Jan 18, 2018

Contents

Junebug is a messaging gateway which is intended to provide the means to connect to mobile network infrastructure to enable the managing of text messages via a RESTful HTTP interface.

CHAPTER 1

Contents

1.1 Installation

Junebug requires Python (version 2.7) to be installed. This installation method also requires pip. Both of these must be installed before following the installation steps below.

Junebug also needs Redis and RabbitMQ. On Debian-based systems, one can install them using:

```
$ apt-get install redis-server rabbitmq-server
```

The Python cryptography library that Junebug depends on requires that the SSL and FFI library headers be installed. On Debian-based systems, one can install these using:

```
$ apt-get install libssl-dev libffi-dev
```

Junebug can be then installed using:

```
$ pip install junebug
```

1.2 Getting started

This guides assumed you have already followed the *installation guide*.

If you prefer a video tutorial, you can start by watching our demo.

First, make sure you have Redis and RabbitMQ running. On a Debian-based system, one can run them with:

```
$ service redis-server start
$ service rabbitmq-server start
```

We can now launch Junebug:

```
$ jb -p 8000
```

This starts the Junebug HTTP API running on port 8000. At the moment it won't have any transports running, so let's create one using the API:

```
$ curl -X POST \
       -d '{
        "type": "telnet",
        "label": "My First Channel",
        "mo_url": "http://requestb.in/pzvivfpz",
        "config": {"twisted_endpoint": "tcp:9001"}
        }' \
       http://localhost:8000/channels/
```

Here, we tell Junebug to send all mobile-originating messages received by this channel to *mo_url*. We use a requestb.in url so that we can inspect the messages.

This creates a simple telnet transport that listens on port 9001. You should get a response like:


```
{
  "status": 200,
  "code": "OK",
  "description": "channel created",
  "result": {
    "status": {
      "status": null,
      "inbound_message_rate": 0,
      "outbound_message_rate": 0,
      "submitted_event_rate": 0,
      "rejected_event_rate": 0,
      "delivery_succeeded_rate": 0,
      "delivery_pending_rate": 0,
      "delivery_failed_rate": 0,
      "components": {},
    },
    "mo_url": "http://requestb.in/pzvivfpz",
    "label": "My First Channel",
    "type": "telnet",
    "config": {"twisted_endpoint": "tcp:9001"},
    "id": "bc5f2e63-7f53-4996-816d-4f89f45a5842"
  }
}
```


With the telnet transport running, we can now connect to the telnet transport:

```
$ telnet localhost 9001
```

If we take a look at the requestbin we gave as the *mo_url* when creating the channel, we should see something like this:


```
{
  "channel_data": {"session_event": "new"},
  "from": "127.0.0.1:53378",
  "channel_id": "bc5f2e63-7f53-4996-816d-4f89f45a5842",
  "timestamp": "2015-10-06 14:16:34.578820",
  "content": null,
  "to": "0.0.0.0:9001",
  "reply_to": null,
```

```
  "group": null,
  "message_id": "35f3336d4a1a46c7b40cd172a41c510d"
}
```


This message was sent to the channel when we connected to the telnet transport, and is equivalent to starting a session for a session-based transport type like USSD.

Now, lets send a message to the telnet transport via junebug:

```
$ curl -X POST \
    -d '{
      "to": "127.0.0.1:53378",
      "content": "hello"
    }' \
    localhost:8000/channels/bc5f2e63-7f53-4996-816d-4f89f45a5842/messages/
```

Here, we sent a message to the address *127.0.0.1:53378*. We should see a *hello* message appear in our telnet client.

We should also get the message details returned as the response to our request


```
{
  "status": 200,
  "code": "OK",
  "description": "message sent",
  "result": {
    "channel_data": {},
    "from": null,
    "channel_id": "bc5f2e63-7f53-4996-816d-4f89f45a5842",
    "timestamp": "2017-11-16 07:22:25.162779",
    "content": "hello",
    "to": "127.0.0.1:53378",
    "reply_to": null,
    "group": null,
    "message_id": "5ee304925efd4afcaa6211a9a578a9f1"
  }
}
```


We can use this message ID to get the details of our sent message:

```
$ curl localhost:8000/channels/bc5f2e63-7f53-4996-816d-4f89f45a5842/messages/
↪5ee304925efd4afcaa6211a9a578a9f1
```

Which should give us a response similar to


```
{
  "status": 200,
  "code": "OK",
  "description": "message status",
  "result": {
    "events": [
      {
        "channel_id": "bc5f2e63-7f53-4996-816d-4f89f45a5842",
        "event_type": "submitted",
        "timestamp": "2017-11-16 07:22:25.169550",
        "event_details": {},
        "message_id": "5ee304925efd4afcaa6211a9a578a9f1"
      }
    ],
```

```
    "last_event_timestamp": "2017-11-16 07:22:25.169550",
    "id": "5ee304925efd4afcaa6211a9a578a9f1",
    "last_event_type": "submitted"
  }
}
```

Which tells us that our message was successfully submitted.

Now, lets try receive a message via junebug by entering a message in our telnet client (followed by a new line):

```
> Hi there
```

If we take a look at our requestbin url, we should see a new request:

```
{
  "channel_data": {"session_event": "resume"},
  "from": "127.0.0.1:53378",
  "channel_id": "bc5f2e63-7f53-4996-816d-4f89f45a5842",
  "timestamp": "2015-10-06 14:30:51.876897",
  "content": "Hi there",
  "to": "0.0.0.0:9001",
  "reply_to": null,
  "message_id": "22c9cd74c5ff42d9b8e1a538e2a17175"
}
```

Now, lets send a reply to this message by referencing its *message_id*:

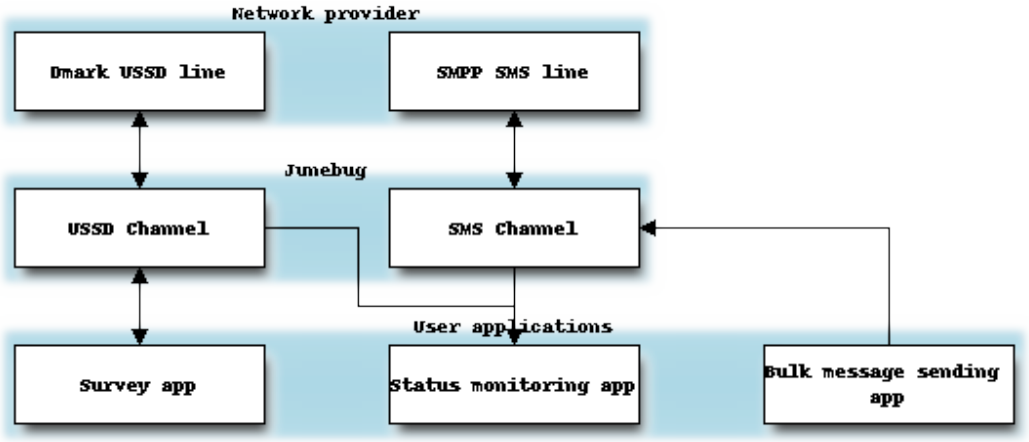
```
$ curl -X POST \
    -d '{
      "reply_to": "22c9cd74c5ff42d9b8e1a538e2a17175",
      "content": "hello again"
    }' \
    localhost:8000/channels/bc5f2e63-7f53-4996-816d-4f89f45a5842/messages/
```

We should see *hello again* appear in our telnet client.

Those are the basics for sending and receiving messages via junebug. Take a look at junebug's *HTTP API documentation* to see how else one can interact with junebug, and junebug's *CLI* and *config* references for more on how junebug can be configured.

1.2.1 Infrastructure Diagram

This diagram is an example configuration of how all the parts of Junebug fit together in a typical setup.



This diagram details a simple application that uses Junebug. It has two lines. The first line is a USSD line which the users will use to answer survey questions. The second is an SMS line, which is used for bulk message sending to prompt the users to dial the USSD line when a new survey is available.

Each of these lines is connected to a Junebug channel.

The USSD channel sends all of its incoming messages to an application which knows what to do with the messages, and can generate appropriate responses. In this case, the application will store the user's answer, and send the user the next question in the survey.

The SMS channel receives messages that it must send out on its messages endpoint. These messages are generated by the bulk send application, which notifies the users when a new survey is available.

Both of the channels send their status events to a status monitoring app, which sends out emails to the correct people when there is something wrong with either of the channels.

1.3 Command-line Reference

Junebug. A system for managing text messaging transports via a RESTful HTTP interface

```
usage: jb [-h] [--config CONFIG_FILENAME] [--interface INTERFACE]
          [--port PORT] [--log-file LOGFILE] [--sentry-dsn SENTRY_DSN]
          [--redis-host REDIS_HOST] [--redis-port REDIS_PORT]
          [--redis-db REDIS_DB] [--redis-password REDIS_PASS]
          [--amqp-host AMQP_HOST] [--amqp-vhost AMQP_VHOST]
          [--amqp-port AMQP_PORT] [--amqp-user AMQP_USER]
          [--amqp-password AMQP_PASS]
          [--inbound-message-ttl INBOUND_MESSAGE_TTL]
          [--outbound-message-ttl OUTBOUND_MESSAGE_TTL]
          [--allow-expired-replies] [--channels CHANNELS]
          [--replace-channels REPLACE_CHANNELS] [--routers ROUTERS]
          [--replace-routers REPLACE_ROUTERS] [--plugin PLUGINS]
          [--metric-window METRIC_WINDOW] [--logging-path LOGGING_PATH]
          [--log-rotate-size LOG_ROTATE_SIZE] [--max-log-files MAX_LOG_FILES]
          [--max-logs MAX_LOGS]
          [--rabbitmq-management-interface RABBITMQ_MANAGEMENT_INTERFACE]
```

1.4 Named Arguments

--config, -c Path to config file. Optional. Command line options override config options

--interface, -i The interface to expose the API on. Defaults to "localhost"

--port, -p The port to expose the API on, defaults to "8080"

--log-file, -l The file to log to. Defaults to not logging to a file

--sentry-dsn, -sd The DSN to log exceptions to. Defaults to not logging

--redis-host, -redish The hostname of the redis instance. Defaults to "localhost"

--redis-port, -redisp The port of the redis instance. Defaults to "6379"

--redis-db, -redisdb The database to use for the redis instance. Defaults to "0"

--redis-password, -redispass The password to use for the redis instance. Defaults to "None"

--amqp-host, -amqph The hostname of the amqp endpoint. Defaults to "127.0.0.1"

--amqp-vhost, -amqpvh The amqp vhost. Defaults to "/"

--amqp-port, -amqpp The port of the amqp endpoint. Defaults to "5672"

--amqp-user, -amqpu The username to use for the amqp auth. Defaults to "guest"

--amqp-password, -amqppass The password to use for the amqp auth. Defaults to "guest"

--inbound-message-ttl, -ittl The maximum time allowed to reply to a message (in seconds).Defaults to 600 seconds (10 minutes).

--outbound-message-ttl, -ottl The maximum time allowed for events to arrive for messages (in seconds). Defaults to 172800 seconds (2 days)

--allow-expired-replies, -aer If enabled messages with a reply_to that arrive for which the original inbound cannot be found (possible of the TTL expiring) are sent as normal outbound messages.

--channels, -ch Add a mapping to the list of channels, in the format "channel_type:python_class".

--replace-channels, -rch If True, replaces the default channels with 'channels'. If False, adds 'channels' to the list of default channels. Defaults to False.

--routers Add a mapping to the list of routers, in the format "router_type:python_class".

--replace-routers If True, replaces the default routers with 'routers'. If False, adds 'routers' to the list of default routers. Defaults to False.

--plugin, -pl Add a plugins to the list of plugins, as a json blob of the plugin config. Must contain a 'type' key, with the full python class path of the plugin

--metric-window, -mw The size of each bucket (in seconds) to use for metrics. Defaults to 10 seconds.

--logging-path, -lp The path to place log files for each channel. Defaults to 'logs/'

--log-rotate-size, -lrs The maximum size (in bytes) for each log file before it gets rotated. Defaults to 1000000.

--max-log-files, -mlf the maximum number of log files to keep before deleting old files. defaults to 5. 0 is unlimited.

--max-logs, -ml the maximum number of log entries to to allow to be fetched through the API. Defaults to 100.

--rabbitmq-management-interface, -rmi This should be the url string of the rabbitmq management interface. If set, the health of each individual queue will be checked. This is only available for RabbitMQ

We also have the following environment variables:

JUNEBUG_REACTOR Choose which twisted reactor to use for Junebug. Can be one of SELECT, POLL, KQUEUE, WFMO, IOCP or EPOLL.

JUNEBUG_DISABLE_LOGGING Set to true to disable logging to the command line for Junebug.

1.5 Config File Reference

A yaml config file can be given using the *-c command line option*. Note that command line options override config fields.

1.5.1 Fields

interface Interface to expose the API on

type: *str*
default: `'localhost'`

port Port to expose the API on

type: *int*
default: `8080`

logfile File to log to or 'None' for no logging

type: *str*
default: `None`

sentry_dsn DSN to send exceptions

type: *str*
default: `None`

redis Config to use for redis connection

type: *dict*
default: `{'db': 0, 'host': 'localhost', 'password': None, 'port': 6379}`

amqp Config to use for amqp connection

type: *dict*
default:

```
{'db': 0,
 'hostname': '127.0.0.1',
 'password': 'guest',
 'port': 5672,
 'username': 'guest',
 'vhost': '/'}
```

inbound_message_ttl Maximum time (in seconds) allowed to reply to messages

type: *int*
default: `600`

outbound_message_ttl Maximum time (in seconds) allowed for events to arrive for messages

type: *int*
default: `172800`

allow_expired_replies If 'True' messages with a reply_to that arrive for which the original inbound cannot be found (possible of the TTL expiring) are sent as normal outbound messages.

type: *bool*
default: `False`

channels Mapping between channel types and python classes.

type: *dict*
default: `{}`

replace_channels If 'True', replaces the default channels with 'channels'. If 'False', 'channels' is added to the default channels.

type: *bool*
default: `False`

routers Mapping between router types and python classes.

type: *dict*
default: `{}`

replace_routers If 'True', replaces the default routers with 'routers'. If 'False','routers' is added to the default routers.

type: *bool*
default: `False`

plugins A list of dictionaries describing all of the enabled plugins. Each item should have a 'type' key, with the full python class name of the plugin.

type: *list*
default: `[]`

metric_window The size of the buckets (in seconds) used for metrics.

type: *float*
default: `10.0`

logging_path The path to place log files in.

type: *str*
default: `'logs/'`

log_rotate_size The maximum size (in bytes) of a log file before it gets rotated.

type: *int*
default: `1000000`

max_log_files The maximum amount of log files allowed before old files start to get deleted. 0 is unlimited.

type: *int*
default: `5`

max_logs The maximum amount of logs that is allowed to be retrieved via the API.

type: *int*

default: `100`

rabbitmq_management_interface This should be the url string of the rabbitmq management interface.If set, the health of each individual queue will be checked. This is only available for RabbitMQ

type: *str*
default: `None`

1.6 Design

An HTTP API for managing text-based messaging channels such as SMS, USSD, XMPP, Twitter, IRC.

Any design goals marked in italics are goals that are not yet implemented in the code base.

1.6.1 Features

Junebug is a system for managing text messaging transports via a RESTful HTTP interface that supports:

- Creating, introspecting, updating and deleting transports
- Sending and receiving text messages
- Receiving status updates on text messages sent
- Monitoring transport health and performance
- Retrieving recent transport logs for debugging transport issues.

1.6.2 Design principles

Junebug aims to satisfy the following broad criteria:

- Avoid replication of work to integrate with aggregators and MNOs (and to maintain these connections).
- Maximum simplicity (narrow scope, minimal useful feature set).
- Provide a common interface to diverse connection protocols.
- Handle buffering of messages and events in both inbound and outbound directions.
- Minimum dependencies.
- Easy to install.

1.6.3 Design goals

Outbound message sending

- Simple universal HTTP API for sending.
- Each channel is independent and cannot block or cause others to fail.
- Does not handle routing. Channel to send to is specified when making the HTTP API request.
- *Configurable rate-limiting on each connection.*

- Queues for each connection.
- *Multiple priority levels supported for each connection. We need at least two priority levels: "bulk" and "real-time".*

Delivery and status reports for outbound messages

- Can specify an HTTP URL with each outgoing request, as the callback for delivery status reports for that message.
- *Callback URL supports template variables (like Kannel does).*

Channel creation and management

- CRUD operations on channels.
- Possible to list channels.
- *Possible to query the queue length.*
- Possible to query the message sending, delivery and failure rates.
- *Possible to cancel the sending of all queued messages.*

Inbound messages

- Able to specify a URL, per channel
- *Inbound message URL supports configurable (substituted) parameters.*
- *Retries delivery on failures*

Logging

Accessing detailed log messages is a vital requirement for debugging issues with connections to third parties.

- Be able to access PDUs or similar detailed channel message logs for recent messages.
- Logs should be retrievable by associated channel id.
- *Logs should be retrievable by associated message id.*

1.6.4 Relevant prior works

Other systems that have acted as inspirations or components of Junebug:

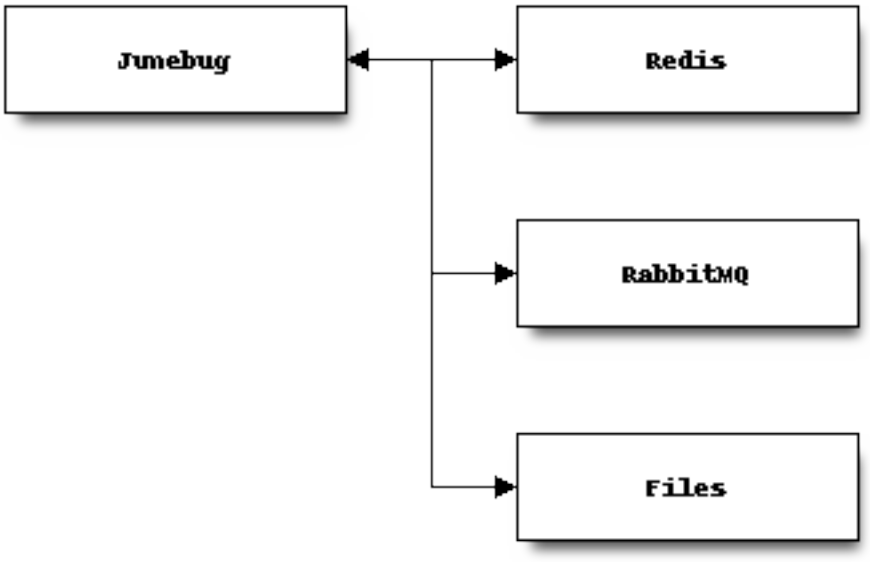
- Vumi and Vumi Go
- Kannel
- CloudHopper

1.6.5 Glossary

channel a single connection to a provider

channel type connections are instances of channel types. Example: Twilio, SMPP 3.4, etc.

1.6.6 System Architecture



Junebug relies on Redis for temporary storage, RabbitMQ for message queueing, and files on disc for logs. This setup was chosen so that Junebug would be easy to setup and start using, and not require a large, complicated, multi-system setup.

There are downsides to this approach, however, Junebug is currently restricted to a single threaded process.

1.7 Internal Architecture

Internal Junebug is structured as a set of services that live within a single process.

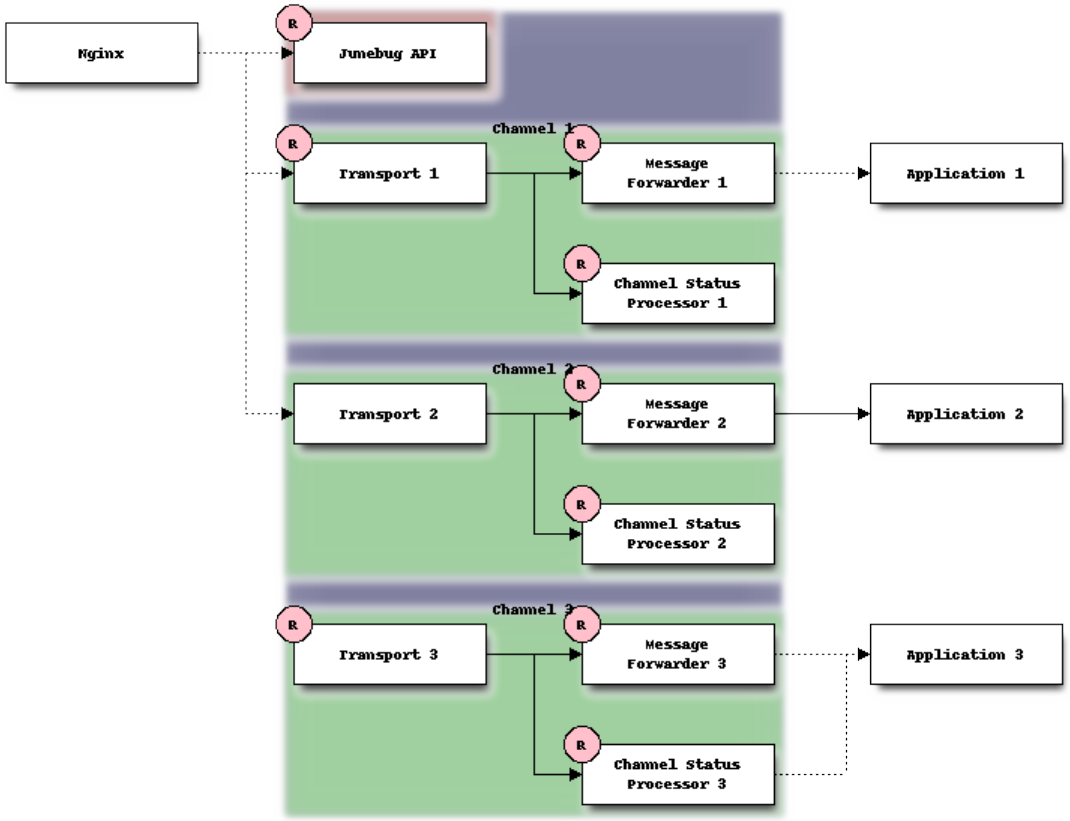
Each Junebug channel creates three services – a transport, a message forwarder and a status receiver.

Transports are Vumi transport workers that send and receive SMSes, USSD requests or other text messages from external service providers such as MNOs (mobile network operators) or aggregators (re-sellers of connections to multiple MNOs).

Message Forwarders receive inbound (MO, mobile originated) messages from transports and relay to applications either via HTTP or AMQP depending on the channel configuration.

Channel Status Processors receive status events from transports and store them in redis and forward them on via HTTP to interested applications.

Junebug uses Redis to store configuration and temporary state such as channel status. Services that use Redis are marked with an "R" in the diagram below. Some types of transports will also make use of Redis.



1.8 HTTP API

Junebug's HTTP API.

1.8.1 HTTP API endpoints

Channels

`GET /channels/`
List all channels

`POST /channels/`
Create a channel.

Parameters

- **`type`** (*`str`*) – channel type (e.g. `smpp`, `twitter`, `xmpp`)
- **`label`** (*`str`*) – user-defined label
- **`config`** (*`dict`*) – channel connection settings (a blob to pass to the channel type implementor)
- **`metadata`** (*`dict`*) – user-defined data blob (used to record user-specified information about the channel, e.g. the channel owner)

- **status_url** (*str*) – URL that junebug should send *status events* to. May be `null` if not desired. Not supported by every channel.
- **mo_url** (*str*) – URL to call on incoming messages (mobile originated) from this channel. One or both of mo_url or amqp_queue must be specified. If both are specified, messages will be sent to both.
- **mo_url_auth_token** (*str*) – The token to use for authentication if the mo_url requires token auth.
- **amqp_queue** (*str*) – AMQP queue to repost messages onto for mobile originated messages. One or both of mo_url or amqp_queue must be specified. If both are specified, messages are sent to both. See *AMQP integration* for more details.
- **rate_limit_count** (*int*) – Number of incoming messages to allow in a given time window. Not yet implemented. See `rate_limit_window`.
- **rate_limit_window** (*int*) – Size of throttling window in seconds. Not yet implemented.
- **character_limit** (*int*) – Maximum number of characters allowed per message.

Returns:

Parameters

- **status** (*int*) – HTTP status code (201 on success).
- **code** (*str*) – HTTP status string.
- **description** (*str*) – Description of result (`"channel created"` on success).
- **result** (*dict*) – The channel created.

GET /channels/(**channel_id:** *str*)

Return the channel configuration and a nested *status* object.

The status object takes the following form:

Parameters

- **status** (*str*) – The worst case of each *component's status level*. For example if the `redis` component's status is `degraded` and the `amqp` component's status is `down`, this field's value will will be *down*.
- **components** (*dict*) – An object showing the most recent status event for each component.
- **inbound_message_rate** (*float*) – The inbound messages per second for the channel.
- **outbound_message_rate** (*float*) – The outbound messages per second for the channel.
- **submitted_event_rate** (*float*) – The submitted events per second for the channel.
- **rejected_event_rate** (*float*) – The rejected events per second for the channel.
- **delivery_succeeded_rate** (*float*) – The delivery succeeded events per second for the channel.
- **delivery_failed_rate** (*float*) – The delivery failed events per second for the channel.
- **delivery_pending_rate** (*float*) – The delivery pending events per second for the channel.

Example response:


```
{
  "status": 200,
  "code": "OK",
  "description": "channel found",
  "result": {
    "id": "89b71dfe-afd8-4e0d-9290-bba791458627",
    "type": "smpp",
    "label": "An SMPP Transport",
    "config": {
      "system_id": "secret_id",
      "password": "secret_password"
    },
    "metadata": {
      "owned_by": "user-5"
    },
    "status_url": "http://example.com/user-5/status",
    "mo_url": "http://example.com/user-5/mo",
    "rate_limit_count": 500,
    "rate_limit_window": 10,
    "character_limit": null,
    "status": {
       "status": "ok",
       "components": {
          "smpp": {
             "component": "smpp",
             "channel_id": "89b71dfe-afd8-4e0d-9290-bba791458627",
             "status": "ok",
             "details": {},
             "message": "Successfully bound",
             "type": "bound"
          }
      },
      "inbound_message_rate": 1.75,
      "outbound_message_rate": 7.11,
      "submitted_event_rate": 6.2,
      "rejected_event_rate": 2.13,
      "delivery_succeeded_rate": 5.44,
      "delivery_failed_rate": 1.27,
      "delivery_pending_rate": 4.32
    }
  }
}
```


POST /channels/(**channel_id:** *str*)

Modify a channel's configuration.

Accepts the same parameters as *POST /channels/*. Only the parameters provided are updated. Others retain their original values.

DELETE /channels/(**channel_id:** *str*)

Stops a channel and deletes it's configuration.

POST /channels/(**channel_id:** *str*)**/restart**

Restart a channel.

Channel Logs

GET /channels/(**channel_id:** *str*)**/logs**
Get the most recent logs for a specific channel.

Query Parameters

- **n** (*int*) – Optional. The number of logs to fetch. If not supplied, then the configured maximum number of logs are returned. If this number is greater than the configured maximum logs value, then only the configured maximum number of logs will be returned.

The response is a list of logs, with each log taking the following form:

Parameters

- **logger** (*str*) – The logger that created the log, usually the channel id.
- **level** (*int*) – The level of the logs. Corresponds to the levels found in the python module `logging`.
- **timestamp** (*float*) – Timestamp, in the format of seconds since the epoch.
- **message** (*str*) – The message of the log.

In the case of an exception, there will be an exception object, with the following parameters:

Parameters

- **class** (*str*) – The class of the exception.
- **instance** (*str*) – The specific instance of the exception.
- **stack** (*list*) – A list of strings representing the traceback of the error.

Example Request:

```
GET /channels/123-456-7a90/logs?n=2 HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
{
  "status": 200,
  "code": "OK",
  "description": "logs retrieved",
  "result": [
      {
          "logger": "123-456-7a90",
          "level": 40,
          "timestamp": 987654321.0,
          "message": "Last log for the channel",
          "exception": {
              "class": "ValueError",
              "instance": "ValueError(\"Bad value\",)",
              "stack": [
                  "..."
              ]
          }
      },
      {
          "logger": "123-456-7a90",
          "level": 20,
```

```
            "timestamp": 987654320.0,
            "message": "Second last log for the channel"
        }
    ]
}
```

Channel Messages

POST /channels/(**channel_id:** *str*)**/messages/**

Send an outbound (mobile terminated) message.

Parameters

- **to** (*str*) – the address (e.g. MSISDN) to send the message too. If Junebug is configured with `allow_expired_replies` The `to` parameter is used as a fallback in case the value of the `reply_to` parameter does not resolve to an inbound message.
- **from** (*str*) – the address the message is from. May be `null` if the channel only supports a single from address.
- **group** (*str*) – If supported by the channel, the group to send the messages to. Not required, and may be `null`
- **reply_to** (*str*) – the uuid of the message being replied to if this is a response to a previous message. Important for session-based transports like USSD. Optional. Can be combined with `to` and `from` if Junebug is configured with `allow_expired_replies`. If that is the case the `to` and `from` values will be used as a fallback in case the value of the `reply_to` parameter does not resolve to an inbound message. The default settings allow 10 minutes to reply to a message, after which an error will be returned.
- **content** (*str*) – The text content of the message. Required.
- **event_url** (*str*) – URL to call for status events (e.g. acknowledgements and delivery reports) related to this message. The default settings allow 2 days for events to arrive, after which they will no longer be forwarded.
- **event_auth_token** (*str*) – The token to use for authentication if the event_url requires token auth.
- **priority** (*int*) – Delivery priority from 1 to 5. Higher priority messages are delivered first. If omitted, priority is 1. Not yet implemented.
- **channel_data** (*dict*) – Additional data that is passed to the channel to interpret. E.g. `continue_session` for USSD, `direct_message` or `tweet` for Twitter.

Example request:

```
{
  "to": "+26612345678",
  "from": "8110",
  "reply_to": "uuid-1234",
  "event_url": "http://example.com/events/msg-1234",
  "content": "Hello world!",
  "priority": 1,
  "channel_data": {
    "continue_session": true,
  }
}
```

Example response:

```
{
  "status": 201,
  "code": "created",
  "description": "message submitted",
  "result": {
    "message_id": "message-uuid-1234"
  }
}
```

GET /channels/(**channel_id:** *str*)**/messages/**
msg_id: *str* Retrieve a message's status.

Example response:

```
{
  "status": 200,
  "code": "OK",
  "description": "message status",
  "result": {
    "id": "msg-uuid-1234",
    "last_event_type": "ack",
    "last_event_timestamp": "2015-06-15 13:00:00",
    "events": [
        "...array of all events; formatted like events..."
    ]
  }
}
```

Routers

GET /routers/
Get a list of routers.

Example response:

```
{
  "status": 200,
  "code": "OK",
  "description": "routers retrieved",
  "results": [
    "89b9e287-f437-4f71-afcf-3d581716a221",
    "512cb98c-39f0-49b2-9938-8bb2ab9da704"
  ]
}
```

POST /routers/
Create a new router.

Parameters

- **type** (*str*) – the type of router to create. Required.
- **label** (*str*) – user-defined label. Ignored by the router, but can be used to store an application specific label, e.g. the name of the router that you want to appear on the front end. Not required.

- **config** (*dict*) – the config to send to the router type to create the new router. This config differs per router type. Required.
- **metadata** (*dict*) – user-defined data blob. Ignored by the router, but can be used to store application specific information, eg. the owner of the router. Not required.

Returns:

Parameters

- **status** (*int*) – HTTP status code (201 on success).
- **code** (*str*) – HTTP status string.
- **description** (*str*) – Description of result (`"router created"` on success).
- **result** (*dict*) – The router created.

Example request:

```
{
  "type": "from_address",
  "label": "SMS longcode 12345",
  "config": {
    "channel": "65227a53-b785-4679-a8e6-b53115b7995a"
  },
  "metadata": {
    "owner": 7
  }
}
```

Example response:

```
{
  "status": 201,
  "code": "Created",
  "description": "router created",
  "result": {
    "id": "512cb98c-39f0-49b2-9938-8bb2ab9da704",
    "type": "from_address",
    "label": "SMS longcode 12345",
    "config": {
      "channel": "65227a53-b785-4679-a8e6-b53115b7995a"
    },
    "metadata": {
      "owner": 7
    },
    "status": {
      "inbound_message_rate": 1.75,
      "outbound_message_rate": 7.11,
      "submitted_event_rate": 6.2,
      "rejected_event_rate": 2.13,
      "delivery_succeeded_rate": 5.44,
      "delivery_failed_rate": 1.27,
      "delivery_pending_rate": 4.32
    }
  }
}
```

GET /routers/(router_id: *str*)

Get the configuration and status information for a router. Returns in the same format as *creating a router*.

PUT /routers/(**router_id:** *str*)
Replace the router's configuration with the one provided. Takes and returns the same parameters as *creating a router*.

PATCH /routers/(**router_id:** *str*)
Replace parts of the router's configuration with the parts provided. Takes and returns the same parameters as *creating a router*, except no parameters are required.

DELETE /routers/(**router_id:** *str*)
Stops a router and deletes its configuration.

Router Destinations

GET /routers/(**router_id:** *str*)**/destinations/**
Get a list of destinations for the specified router

Example response:

```
{
    "status": 200,
    "code": "OK",
    "description": "destinations retrieved",
    "result": [
      "53ed5492-48a1-4aec-9d64-b9080893cb4a",
      "7b23a20f-9330-4a4e-8bd1-4819470ffa31"
    ]
}
```

POST /routers/(**router_id:** *str*)**/destinations/**
Create a new destination for a router.

Parameters

- **label** (*str*) – user-defined label. Ignored by the destination, but can be used to store an application specific label, e.g. the name of the destination that you want to appear on the front end. Not required.
- **config** (*dict*) – The configuration for this destination. Configuration is specific to the type of router. Required.
- **metadata** (*dict*) – user-defined data blob. Ignored by the destination, but can be used to store application specific information, eg. the owner of the destination. Not required.
- **mo_url** (*str*) – The url to send inbound (mobile originated) to. None, one, or both of mo_url and amqp_queue may be specified. If none are specified, messages are ignored. If both are specified, messages are sent to both. Optional.
- **mo_url_token** (*str*) – The token to use for authentication if the mo_url requires token auth. Optional.
- **amqp_queue** (*str*) – The queue to place messages on for this destination. Optional.
- **character_limit** (*int*) – Maximum number of characters allowed per message.

Returns:

Parameters

- **status** (*int*) – HTTP status code (201 on success)
- **code** (*str*) – HTTP status string

- **description** (*str*) – Description of result ("destination created" on success)
- **result** (*dict*) – The destination created.

Example request:

```
{
  "label": "*123*4567*1# subcode",
  "config": {
    "regular_expression": "^\*123\*4567\*1#$"
  },
  "metadata": {
    "owner": 7
  },
  "mo_url": "https://www.example.org/messages",
  "mo_url_token": "my-secret-token",
  "amqp_queue": "subcode_1_queue",
  "character_limit": 140
}
```

Example response:

```
{
  "status": 201,
  "code": "Created",
  "description": "destination created",
  "result": {
    "id": "7b23a20f-9330-4a4e-8bd1-4819470ffa31",
    "label": "*123*4567*1# subcode",
    "config": {
      "regular_expression": "^\*123\*4567\*1#$"
    },
    "metadata": {
      "owner": 7
    },
    "mo_url": "https://www.example.org/messages",
    "mo_url_token": "my-secret-token",
    "amqp_queue": "subcode_1_queue",
    "character_limit": 140,
    "status": {
      "inbound_message_rate": 1.75,
      "outbound_message_rate": 7.11,
      "submitted_event_rate": 6.2,
      "rejected_event_rate": 2.13,
      "delivery_succeeded_rate": 5.44,
      "delivery_failed_rate": 1.27,
      "delivery_pending_rate": 4.32
    }
  }
}
```

GET /routers/(**router_id:** *str*)**/destinations/**
destination_id: *str* Get the configuration and status information for a destination. Returns in the same format as *creating a destination*.

PUT /routers/(**router_id:** *str*)**/destinations/**
destination_id: *str* Replace the destination's configuration with the one provided. Takes and returns the same parameters as *creating a destination*.

PATCH /routers/(**router_id:** *str*)**/destinations/**

destination_id: *str* Replace parts of the destination's configuration with the parts provided. Takes and returns the same parameters as *creating a destination*, except no parameters are required.

DELETE /routers/(**router_id:** *str*)**/destinations/**
destination_id: *str* Stops a destination and deletes its configuration.

Router logs

GET /routers/(**router_id:** *str*)**/logs**
Get the latest logs for a router. Takes and returns the same parameters as *channel logs*.

GET /routers/(**router_id:** *str*)**/destinations/**
destination_id: *str***/logs** Get the latest logs for a destination. Takes and returns the same parameters as *channel logs*.

Router Destination Messages

POST /routers/(**router_id:** *str*)**/destinations/**
destination_id: *str***/messages** Send a message from a destination of a router. Takes and returns the same parameters as *sending a channel message*. Messages can also be sent through the *sending a channel message* endpoint.

GET /routers/(**router_id:** *str*)**/destinations/**
destination_id: *str***/messages/message-id:** *str* Get the status of a message. Takes and returns the same parameters as *getting the status of a channel message*. Message statuses can also be found at the *getting the status of a channel message* endpoint.

1.8.2 Events

Events `POST`ed to the `event_url` specified in *`POST /channels/(channel_id:str)/messages/`* have the following format:

POST /event/url

Parameters

- **event_type** (*str*) – The type of the event. See the list of event types below.
- **message_id** (*str*) – The UUID of the message the event is for.
- **channel_id** (*str*) – The UUID of the channel the event occurred for.
- **timestamp** (*str*) – The timestamp at which the event occurred.
- **event_details** (*dict*) – Details specific to the event type.

Events are posted to the message's `event_url` after the message is submitted to the provider, and when delivery reports are received. The default settings allow events to arrive for up to 2 days; any further events will not be forwarded.

Request example:

```
{
  "event_type": "submitted",
  "message_id": "msg-uuid-1234",
  "channel_id": "channel-uuid-5678",
  "timestamp": "2015-06-15 13:00:00",
  "event_details": {
```

```
      "...detail specific to the channel implementation..."
  }
}
```

Event types

Sent when the message is submitted to the provider:

- `submitted`: message successfully sent to the provider.
- `rejected`: message rejected by the channel.

Sent later when (or if) delivery reports are received:

- `delivery_succeeded`: provider confirmed that the message was delivered.
- `delivery_failed`: provider declared that message delivery failed.
- `delivery_pending`: provider is still attempting to deliver the message.

1.8.3 Inbound (Mobile Originated) Messages

Inbound messages that are `POST`ed to the `mo_url` specified in *`POST /channels/`* have the following format:

`POST /mobile_originated/url`

Parameters

- **`to`** (*`str`*) – The address that the message was sent to.
- **`from`** (*`str`*) – The address that the message was sent from.
- **`group`** (*`str`*) – If the transport supports groups, the group that the message was sent in.
- **`message_id`** (*`str`*) – The string representation of the UUID of the message.
- **`channel_id`** (*`str`*) – The string representation of the UUID of the channel that the message came in on.
- **`timestamp`** (*`str`*) – The timestamp of when the message arrived at the channel, in the format `"%Y-%m-%d %H:%M:%S.%f`.
- **`reply_to`** (*`str`*) – If this message is a reply of an outbound message, the string representation of the UUID of the outbound message.
- **`content`** (*`str`*) – The text content of the message.
- **`channel_data`** (*`dict`*) – Any channel implementation specific data. The contents of this differs between channel implementations.

Request example:

```
{
    "to": "+27821234567",
    "from": "12345",
    "group": null,
    "message_id": "35f3336d4a1a46c7b40cd172a41c510d"
    "channel_id": "bc5f2e63-7f53-4996-816d-4f89f45a5842",
    "timestamp": "2015-10-06 14:16:34.578820",
    "reply_to": null,
    "content": "Test message",
```

```
    "channel_data": {
        "session_event": "new"
    },
}
```

1.8.4 Status events

Status events POSTed to the `status_url` specified in *POST /channels/* have the following format:

POST /status/url

Parameters

- **component** (*str*) – The *component* relevant to this status event.
- **channel_id** (*str*) – The UUID of the channel the status event occurred for.
- **status** (*str*) – The *status level* this event was categorised under.
- **type** (*str*) – A programmatically usable string value describing the reason for the status event.
- **message** (*str*) – A human-readable string value describing the reason for the status event.
- **details** (*dict*) – Details specific to this event intended to be used for debugging purposes. For example, if the event was related to a component establishing a connection, the host and port are possible fields.

Request Example:

```
{
    "status": "down",
    "component": "smpp",
    "channel_id": "channel-uuid-5678",
    "type": "connection_lost",
    "message": "Connection lost",
    "details": {}
}
```

Components

Each status event published by a channel describes a component used as part of the channel's operation. For example, an smpp channel type will have a `redis` component describing its redis connection, an `amqp` component describing its amqp connection and an `smpp` component describing events specific to the SMPP protocol (for example, connections, binds, throttling).

Status levels

A status event can be categorised under one of the following levels:

- `ok`: The component is operational.
- `degraded`: The component is operational, but there is an issue which may affect the operation of the component. For example, the component may be throttled.
- `down`: The component is not operational as a result of the issue described by the event.

1.8.5 Health

`GET /health/`

Provides HTTP GET access to test or verify the health of the system.

If the `rabbitmq_management_interface` config item is set it will also query the RabbitMQ Management interface to check the health of each queue. This is only available for RabbitMQ.

Returns:

param int status

HTTP status code.

- `200`: Everything is healthy.
- `500`: There are queues stuck.

param str code HTTP status string.

param str description

Description of result

- `"health ok"`: Everything is healthy and `rabbitmq_management_interface` is not set.
- `"queues ok"`: Everything is healthy and `rabbitmq_management_interface` is set.
- `"queues stuck"`: There are queues stuck and `rabbitmq_management_interface` is set.

param dict result A list of queues with details (Only if `rabbitmq_management_interface` is set).

Response Example without ‘‘rabbitmq_management_interface‘‘:

```
{
  "status": 200,
  "code": "OK",
  "description": "health ok"
}
```

Response Example with ‘‘rabbitmq_management_interface‘‘:

```
{
  "status": 200,
  "code": "OK",
  "description": "queues ok",
  "result": [
      {
          "stuck": false,
          "rate": 1.3,
          "messages": 4583,
          "name": "b4fda175-011f-40bd-91da-5c88789e1e2a.inbound"
      },
      {
          "stuck": false,
          "rate": 1.6,
          "messages": 43,
          "name": "b4fda175-011f-40bd-91da-5c88789e1e2a.outbound"
```

```
        }
    ]
}
```

1.9 AMQP integration

Should you choose to use the AMQP queues to handle messaging with Junebug, when you specify the `amqp_queue` parameter when you configure the *channel*, the usage is as follows:

- Inbound *(mobile originated)* messages are sent to the **routing key** of `{amqp_queue}.inbound`
- Events are sent to the **routing key** of `{amqp_queue}.event`
- Outbound *(mobile terminated)* messages will be fetched from **queue** `{amqp_queue}.outbound`

Remember to bind the **routing key** to your desired **queue**, and that the Exchange name defaults to `vumi`, else it you will not recieve the messages. Please have a look at https://www.rabbitmq.com/tutorials/amqp-concepts.html for more information.

The data sent over AMQP is the standard Vumi data format.

An example of the data format is:

```
{
    "transport_name": "2427d857-688d-4cee-88d9-8e0e32dfdc13",
    "from_addr_type": null,
    "group": null,
    "from_addr": "127.0.0.1:46419",
    "timestamp": "2016-03-18 11:49:36.830534",
    "in_reply_to": null,
    "provider": null,
    "to_addr": "0.0.0.0:9001",
    "routing_metadata": {},
    "message_id": "a5d2800751f54b55a622e3965e1b71ec",
    "content": "a",
    "to_addr_type": null,
    "message_version": "20110921",
    "transport_type": "telnet",
    "helper_metadata": {
        "session_event": "resume"
    },
    "transport_metadata": {},
    "session_event": "resume",
    "message_type": "user_message"
}
```

1.10 Plugin Reference

A Junebug plugin is a class that has specific methods that get called during specific events that happen within Junebug. Plugins provide a way to hook into Junebug and add extra functionality.

1.10.1 Installing plugins

The *CLI* and *config* references describe how you can add plugins to Junebug.

1.10.2 Methods

The following methods are available to plugins to perform actions according to events that happen within Junebug:

class junebug.plugin.**JunebugPlugin**
Base class for all Junebug plugins

channel_started(*channel*)
Called whenever a channel is started. Should be implemented by the plugin. Can return a deferred.

Parameters **channel** (Channel) – The channel that has been started.

channel_stopped(*channel*)
Called whenever a channel is stopped. Should be implemented by the plugin. Can return a deferred.

Parameters **channel** (Channel) – The channel that has been stopped.

start_plugin(*config*, *junebug_config*)
Can be overridden with any required startup code for the plugin. Can return a deferred.

Parameters

- **config** (*dictionary*) – The config specific to the plugin.
- **junebug_config** (JunebugConfig) – The config that Junebug was started with.

stop_plugin()
Can be overridden with any required shutdown code for the plugin. Can return a deferred.

1.10.3 Creating Plugins

In order to create a plugin, create a python class that inherits from *junebug.plugin.JunebugPlugin*, and implements any of the methods. Then add the plugin to Junebug through either the CLI or a config file, as described in *Installing plugins*.

1.10.4 Bundled plugins

Nginx

Maintains configuration for an nginx virtual host for Junebug to expose http-based channels, reloading nginx whenever a new http-based channel is added.

The plugin looks for `web_path` and `web_port` config fields in each channel config given to *`POST /channels/`*. `web_port` determines the internal tcp port for the server that nginx should proxy requests to. `web_path` determines the path to expose the http-channel on (e.g. `'/foo/bar'`) for the vhost.

The plugin will also look for a `public_http` object in each added channel's config. `public_http.web_path` and `public_http.web_port` override `web_path` and `web_port` respectively. Additionally, one can disable the plugin for a particular channel by setting `public_http.enabled` to `false`.

Config options:

vhost_file The file to write the junebug nginx vhost file to

type: *str*
default: `'/etc/nginx/sites-enabled/junebug.conf'`

locations_dir The directory to write location block config files to

type: *str*
default: `'/etc/nginx/includes/junebug/'`

server_name Server name to use for nginx vhost

type: *str*
default: `None`

vhost_template Path to the template file to use for the vhost config

type: *str*
default: `'/home/docs/checkouts/readthedocs.org/user_builds/junebug/envs/latest/local/lib/python2.7/site-packages/junebug-0.1.39-py2.7.egg/junebug/plugins/nginx/vhost.template'`

location_template Path to the template file to use for each channel's location config

type: *str*
default: `'/home/docs/checkouts/readthedocs.org/user_builds/junebug/envs/latest/local/lib/python2.7/site-packages/junebug-0.1.39-py2.7.egg/junebug/plugins/nginx/location.template'`

Note: The plugin needs permissions to write the vhost and location config files it maintains. See the config options for the default paths that the plugin will write these files to.

1.11 Routers

Note: Not yet implemented

1.11.1 Introduction

Routers in Junebug allow you to use a single channel with multiple applications.

This can be useful in cases where, for example with an SMPP transport, you have a single bind with a provider, but multiple SMS lines.

Or if you have a USSD line, and you want the subcodes to go to different applications.

The routers are desgined in such a way that you can do both one to many, and many to one routing, for example where you have multiple MNOs that you have a separate channel for each, but you want all messages to go to a single application.

See also:

Routers How to use routers with the http API

Config File Reference How to add new router types via a config file

Command-line Reference How to add new router types via command line arguments

1.11.2 Built in router types

The following routers are available with any default setup and installation of Junebug.

From address router

The `from_address` router type routes inbound messages based on regex rules on the from address.

The config for the router takes the following parameters:

channel *(str)* The channel ID of the channel whose messages you want to route. This channel may not have an `amqp_queue` parameter specified. Required.

The config for each of the router destinations takes the following parameters:

regular_expression *(str)* The regular expression to match the from address on. Any inbound messages with a from address that matches this regular expression will be sent to this destination.

default *(bool)* Whether or not this destination is the default destination. Only one destination may be the default destination. Any messages that don't match any of the configured destinations will be sent to the default destination. If no default destination is configured, then non-matching messages will be dropped. Optional, defaults to false.

1.12 Release Notes

1.12.1 v0.1.39

– Fixes

- View messages on destination now fetches from the correct place.
- Disabled channel messages endpoint if there is no destination configured.

1.12.2 v0.1.38

– Features

- Endpoint to create and view messages directly on a routers destination.

1.12.3 v0.1.37

– Features

- Added a log endpoint for routers

1.12.4 v0.1.36

- **Fixes**
  - Validation on router PATCH api endpoint.

1.12.5 v0.1.35

- **Fixes**
  - Store outbound messages that the from address router is forwarding, so that we can forward the events to the correct place.

1.12.6 v0.1.34

8 January 2018

- **Fixes**
  - Add handling for outbound messages from destinations on the from address router

1.12.7 v0.1.33

4 January 2018

– Features

- Added a base router class and a new from address router.

1.12.8 v0.1.32

13 December 2017

- **Documentation**
  - Fixed the mo_url_auth_token field description for channels
- **Fixes**
  - Fixed character limit validation for router destinations

1.12.9 v0.1.31

12 December 2017

- **Features**
  - Added endpoints for creating, modifying, deleting and listing router destinations.
- **Documentation**

  - Remove not yet implemented note for router destinations.

1.12.10 v0.1.30

12 December 2017

- **Documentation**
  - Add format for inbound messages send from Junebug.
- **Fixes**
  - Add handling for invalid JSON in the request body

1.12.11 v0.1.29

6 December 2017

- **Fixes**
  - Fix the name clash of validate_config for creating routers
  - Fix the TestRouter so that it can be used as an actual router for testing

1.12.12 v0.1.28

5 December 2017

- **Features**
  - Added the endpoints for creating, modifying, and deleting routers.
- **Documentation**
  - Removed not yet implemented from Router endpoints.

1.12.13 v0.1.27

27 November 2017

- **Features**
  - Added a config option for RabbitMQ Management Interface to see status of each queue with the health check.
- **Fixes**
  - Change created API endpoint statuses to 201 to match documentation
- **Documentation**
  - Use older version of docutils in documentation build, since txJSON-RPC does not work with the newer version.
  - Create minimal sphinx argparse implementation to use until official sphinx-argparse is made to work with readthedocs again
  - Bring release notes up to date
  - Update design notes to mark what has been implemented and what is yet to be implemented

  - Bring getting started section up to date with what the current API actually returns
  - Adding section in getting started to show getting the related events for a message
  - Update HTTP API documentation to what the API actually looks like
  - Update Readme to give more information to someone who wants to work on the project.

1.12.14 v0.1.26

07 September 2017

- **Fixes**
  - Also catch RequestTransmissionFailed for HTTP requests. This should fix the issue where messages or events would stop being processed in the case of an HTTP error

1.12.15 v0.1.25

28 August 2017

- **Features**
  - Update to vumi 0.6.18

1.12.16 v0.1.24

31 July 2017

- **Features**
  - Allow event auth token to be specified when sending a message
- **Documentation**
  - Add documentation mentioning specifying an auth token for the mo_url and for the event URL

1.12.17 v0.1.23

28 July 2017

- **Features**
  - Allow specifying all of to, from, and reply to when sending a message, defaulting to using reply_to if specified.
  - If allow_expired_replies is specified in the configuration, then if to, from, and reply_to is specified, and reply_to does not resolve to a valid message in the message store, then we will fall back to to and from for creating the message.

1.12.18 v0.1.22

24 July 2017

- **Features**
  - Allow token and basic auth for sending of event messages

1.12.19 v0.1.21

21 July 2017

- **Features**
  - Upgrade to vumi 0.6.17

1.12.20 v0.1.20

18 July 2017

- **Fixes**
  - Allow any 6.x.x version for Raven (Sentry)

1.12.21 v0.1.19

17 July 2017

- **Features**
  - Expose 'group' attribute from vumi message in message payload
- **Documentation**
  - Add newline in cli reference so that documentation renders correctly
  - Fix example response for message creation
  - Update documentation for reply_to change

1.12.22 v0.1.18

12 July 2017

- **Fixes**
  - Change to setup.py to allow Junebug to be installable on python 3

1.12.23 v0.1.17

10 July 2017

- **Features**
  - Display more information on HTTP failure when logging failure
  - Allow an auth token to be specified for inbound (mobile originated) messages being sent over HTTP
- **Fixes**
  - Also catch CancelledError for HTTP timeouts

1.12.24 v0.1.16

7 June 2017

- **Features**
  - Upgrade to pypy 5.7.1
  - Add ability to log exceptions to Sentry

1.12.25 v0.1.15

29 May 2017

- **Features**
  - Upgrade vumi to 0.6.16

1.12.26 v0.1.14

31 March 2017

- **Fixes**
  - Fix tests for new Twisted error output

1.12.27 v0.1.13

Skipped

1.12.28 v0.1.12

31 March 2017

- **Features**
  - Upgrade vumi to 0.6.14

1.12.29 v0.1.11

10 February 2017

- **Fixes**
  - Trap ConnectionRefusedError that can happen when trying to relay a message to an event_url of mo_url.

1.12.30 v0.1.10

06 February 2017

- **Fixes**
  - Make Junebug gracefully handle timeouts and connection failure for events and messages posted to URL endpoints.

1.12.31 v0.1.9

02 February 2017

- **Fixes**
  - Allow one to set the `status_url` and the `mo_url` for a channel to `None` to disable pushing of status events and messages to these URLs.

1.12.32 v0.1.8

18 January 2017

- **Fixes**
  - Change the default smpp channel type from the depricated SmppTransport (SmppTransceiverTransportWithOldConfig), to the new SmppTransceiverTransport.

1.12.33 v0.1.7

10 January 2017

- **Features**
  - Update the minimum version of vumi to get the latest version of the SMPP transport, which allows us to set the keys of the data coding mapping to strings. This allows us to use the data coding mapping setting in Junebug, since in JSON we cannot have integers as keys in an object.

1.12.34 v0.1.6

3 October 2016

- **Fixes**
  - Fix the teardown of the MessageForwardingWorker so that if it didn't start up properly, it would still teardown properly.
  - Handling for 301 redirect responses improved by providing the URL to be redirected to in the body as well as the Location header.
  - We no longer crash if we get an event without the user_message_id field. Instead, we just don't store that event.
- **Features**
  - Update channel config error responses with the field that is causing the issue.
  - Set a minimum twisted version that we support (15.3.0), and ensure that we're testing against it in our travis tests.
  - The logging service now creates the logging directory if it doesn't exist and if we have permissions. Previously we would give an error if the directory didn't exist.
- **Documentation**
  - Added instructions to install libssl-dev and libffi-dev to the installation instructions.
  - Added documentation and diagrams for the internal architecture of Junebug.

1.12.35 v0.1.5

19 April 2016

- **Fixes**
  - Have nginx plugin add a leading slash to location paths if necessary.

1.12.36 v0.1.4

12 April 2016

- **Fixes**
  - Fix nginx plugin to properly support reading of web_path and web_port configuration.
  - Add endpoint for restarting channels.
  - Automate deploys.

1.12.37 v0.1.3

5 April 2016

- **Fixes**
  - Reload nginx when nginx plugin starts so that the vhost file is loaded straight away if the nginx plugin is active.

1.12.38 v0.1.2

5 April 2016

- **Fixes**
  - Added manifest file to fix nginx plugin template files that were missing from the built Junebug packages.
- **Features**
  - Added environment variable for selecting reactor
- **Documentation**
  - Extended AMQP documentation

1.12.39 v0.1.1

1 March 2016

- **Fixes**
  - Junebug now works with PyPy again
  - Fixed sending messages over AMQP

1.12.40 v0.1.0

18 December 2015

- **Fixes**
  - Fixed config file loading
- **Features**
  - We can now get message and event rates on a GET request to the channel endpoint
  - Can now get the last N logs for each channel
  - Can send and receive messages to and from AMQP queues as well as HTTP
  - Dockerfile for creating docker containers
- **Documentation**
  - Add documentation for message and event rates
  - Add documentation for getting a list of logs for a channel
  - Add a changelog to the documentation
  - Update documentation to be ready for v0.1.0 release
  - Remove Alpha version warning

1.12.41 v0.0.5

9 November 2015

- **Fixes**
  - When Junebug is started up, all previously created channels are now started
- **Features**
  - Send errors replies for messages whose length is greater than the configured character limit for the channel
  - Ability to add additional channel types through config
  - Get a message status and list of events for that message through an API endpoint
  - Have channel statuses POSTed to the configured URL on status change
  - Show the latest channel status event for each component and the overall status sumary with a GET request to the specific channel endpoint.
  - Add infrastructure for Junebug Plugins
  - Add Nginx Junebug Plugin that automatically updates the nginx config when it is required for HTTP based channels
  - Add SMPP and Dmark USSD channel types to the default list of channel types, as we now support those channels fully
- **Documentation**
  - Add getting started documentation
  - Updates for health events documentation
  - Add documentation for plugins

  - Add documentation for the Nginx plugin

1.12.42 v0.0.4

23 September 2015

- **Fixes**
  - Ignore events without an associated event forwarding URL, instead of logging an error.
  - Fix race condition where an event could come in before the message is stored, leading to the event not being forwarded because no URL was found

1.12.43 v0.0.3

23 September 2015

- **Fixes**
  - Remove channel from channel list when it is deleted
- **Features**
  - Ability to specify the config in a file along with through the command line arguments
  - Ability to forward MO messages to a configured URL
  - Ability to reply to MO messages
  - Ability to forward message events to a per-message configured URL
- **Documentation**
  - Add documentation about configurable TTLs for inbound and outbound messages

1.12.44 v0.0.2

9 September 2015

- **Fixes**
  - Collection API endpoints now all end in a /
  - Channels are now only started/stopped once instead of twice
- **Features**
  - Ability to send a MT message through an API endpoint
  - Ability to get a list of channels through an API endpoint
  - Ability to delete a channel through an API endpoint

1.12.45 v0.0.1

1 September 2015

- **Features:**
  - API endpoint structure

  - API endpoint validation
  - Health endpoint
  - `jb` command line script
  - Ability to create, get, and modify channels
- **Documentation:**
  - API endpoint documentation
  - Installation documentation
  - Run command documentation

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

HTTP Routing Table

/channels

/event

/health

/mobile_originated

/routers

/status

Index