# Vumi Go Documentation

***Release 0.5.1a***

**Praekelt Foundation and individual contributors**

November 25, 2015

Contents

Vumi Go is a hosted version of Vumi. Where Vumi gives you the tools to build large scale messaging applications, Vumi Go provides you with a working environment that is already integrated into numerous countries.

Vumi Go offers the following on top of Vumi:

- Managing contact information and groups of contacts.
- The ability to store all messages sent and received.
- A set of applications ready for you to use.
- A sandboxed Javascript environment for you to use to develop (and host) your own applications that will run on our scalable infrastructure.

This documentation is mostly geared towards beginning developers interested in building Javascript based applications that can be hosted on Vumi Go and which can interact with a user base via SMS, USSD and other messaging channels.

CHAPTER 1

# Sample Applications

A number of example applications have been built that highlight different bits of functionality.

**Ushahidi**

This is an application that exposes the functionality of Ushahidi over USSD. It allows submission of reports via a USSD menu and uses Google Maps' API for geolocation based on raw address input.

**source** https://github.com/smn/go-ushahidi

**demo** Available in South Africa on `*120*8864*1087#`, interacts with http://vumi.crowdmap.com/

**Google Maps Directions**

This application allows people to receive directions from Google Maps' API. It asks where the user currently is and where they want to go. Using Google's API a geolocation is done on the raw input and the directions are sent to the user via SMS.

**source** https://github.com/smn/go-google-maps

**demo** Available in South Africa on `*120*8864*1105#`.

CHAPTER 2

# Available Resources for the Javascript Sandbox

The Javascript sandbox provides an isolated environment within which a developer's (your!) application code is run.

It talks to the outside work via what are called "Resources". These resources expose core Vumi functionality inside the Sandbox in a controlled fashion.

Applications in Vumi Go's Javascript sandbox have the following resources available:

**config**

This provides access to the `config` variable as stored in the Vumi Go UI.

**class** `go.apps.jsbox.vumi_app.`**`ConversationConfigResource`**(*name*, *app_worker*, *config*)
Bases: `vxsandbox.resources.utils.SandboxResource`

Resource that provides access to conversation config.

**outbound**

This provides access to outbound messaging from the Javascript sandbox to the end user.

**class** `go.apps.jsbox.outbound.`**`GoOutboundResource`**(*name*, *app_worker*, *config*)
Bases: `vxsandbox.resources.utils.SandboxResource`

Resource that provides outbound message support for Go.

Includes support for replying, replying to groups and sending messages via given tags.

Configuration options:

**Parameters** **`allowed_helper_metadata`** (*list*) – List of helper_metadata fields that may be set by sandboxed applications.

**`handle_reply_to`**(*api*, *command*)
Sends a reply to the individual who sent a received message.

**Command fields:**

- `content`: The body of the reply message.
- `in_reply_to`: The `message id` of the message being replied to.
- `continue_session`: Whether to continue the session (if any). Defaults to `true`.
- `helper_metadata`: An object of additional helper metadata fields to include in the reply.

**Reply fields:**

- `success`: `true` if the operation was successful, otherwise `false`.

Example:

```
        api.request(
            'outbound.reply_to',
            {content: 'Welcome!',
             in_reply_to: '06233d4eede945a3803bf9f3b78069ec'},
            function(reply) { api.log_info('Reply sent: ' +
                                           reply.success); });
```

**handle_reply_to_group**(*api*, *command*)

Sends a reply to the group from which a received message was sent.

**Command fields:**

- `content`: The body of the reply message.
- `in_reply_to`: The `message id` of the message being replied to.
- `continue_session`: Whether to continue the session (if any). Defaults to `true`.
- `helper_metadata`: An object of additional helper metadata fields to include in the reply.

**Reply fields:**

- `success`: `true` if the operation was successful, otherwise `false`.

Example:

```
        api.request(
            'outbound.reply_to_group',
            {content: 'Welcome!',
             in_reply_to: '06233d4eede945a3803bf9f3b78069ec'},
            function(reply) { api.log_info('Reply to group sent: ' +
                                           reply.success); });
```

**handle_send_to_endpoint**(*api*, *command*)

Sends a message to a specified endpoint.

**Command fields:**

- `content`: The body of the reply message.
- `to_addr`: The address of the recipient (e.g. an MSISDN).
- `endpoint`: The name of the endpoint to send the message via.
- `helper_metadata`: An object of additional helper metadata fields to include in the message being sent.

**Reply fields:**

- `success`: `true` if the operation was successful, otherwise `false`.

Example:

```
        api.request(
            'outbound.send_to_endpoint',
            {content: 'Welcome!', to_addr: '+27831234567',
             endpoint: 'sms'},
            function(reply) { api.log_info('Message sent: ' +
                                           reply.success); });
```

**handle_send_to_tag**(**args*, ***kwargs*)

Sends a message to a specified tag.

**Command fields:**

- `content`: The body of the reply message.
- `to_addr`: The address of the recipient (e.g. an MSISDN).
- `tagpool`: The name of the tagpool to send the message via.
- `tag`: The name of the tag (within the tagpool) to send the message from. Your Go user account must have the tag acquired.

**Reply fields:**

- `success`: `true` if the operation was successful, otherwise `false`.

Example:

```
api.request(
    'outbound.send_to_tag',
    {content: 'Welcome!', to_addr: '+27831234567',
     tagpool: 'vumi_long', tag: 'default10001'},
    function(reply) { api.log_info('Message sent: ' +
                                   reply.success); });
```

**metrics**

This provides access to the metrics aggregation system inside Vumi. Metrics that are fired here are aggregated and available for displaying in a dashboard. The backend for this is Graphite.

**class** `go.apps.jsbox.metrics.`**`MetricsResource`**(*name*, *app_worker*, *config*)
Bases: `vxsandbox.resources.utils.SandboxResource`

Resource that provides metric storing.

**`handle_fire`**(*api*, *command*)
Fire a metric value.

**http**

This enables you to make outbound HTTP calls. GET, POST, PUT and DELETE are available.

**class** `vumi.application.sandbox.`**`HttpClientResource`**(*name*, *app_worker*, *config*)
Bases: `vumi.application.sandbox.SandboxResource`

Resource that allows making HTTP calls to outside services.

All command on this resource share a common set of command and response fields:

**Command fields:**

- `url`: The URL to request
- **`verify_options`: A list of options to verify when doing** an HTTPS request. Possible string values are `VERIFY_NONE`, `VERIFY_PEER`, `VERIFY_CLIENT_ONCE` and `VERIFY_FAIL_IF_NO_PEER_CERT`. Specifying multiple values results in passing along a reduced `OR` value (e.g. VERIFY_PEER | VERIFY_FAIL_IF_NO_PEER_CERT)
- **`headers`: A dictionary of keys for the header name and a list** of values to provide as header values.
- `data`: The payload to submit as part of the request.
- **`files`: A dictionary, submitted as multipart/form-data** in the request:

```
[{
    "field name": {
        "file_name": "the file name",
```

```
                    "content_type": "content-type",
                    "data": "data to submit, encoded as base64",
                }
            }, ...]
```

The `data` field in the dictionary will be base64 decoded before the HTTP request is made.

**Success reply fields:**

- `success`: Set to `true`
- `body`: The response body
- `code`: The HTTP response code

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure

Example:

```
api.request(
    'http.get',
    {url: 'http://foo/'},
    function(reply) { api.log_info(reply.body); });
```

**agent_class**
alias of `Agent`

**handle_delete**(*api*, *command*)
Make an HTTP DELETE request.

See `HttpResource` for details.

**handle_get**(*api*, *command*)
Make an HTTP GET request.

See `HttpResource` for details.

**handle_head**(*api*, *command*)
Make an HTTP HEAD request.

See `HttpResource` for details.

**handle_patch**(*api*, *command*)
Make an HTTP PATCH request.

See `HttpResource` for details.

**handle_post**(*api*, *command*)
Make an HTTP POST request.

See `HttpResource` for details.

**handle_put**(*api*, *command*)
Make an HTTP PUT request.

See `HttpResource` for details.

**contacts**

This resource provides access to the contact database stored in Vumi Go. It allows you to create, delete and update contact information.

**class** `go.apps.jsbox.contacts.`**`ContactsResource`**(*name*, *app_worker*, *config*)

Bases: `vxsandbox.resources.utils.SandboxResource`

Sandbox resource for accessing, creating and modifying contacts for a Go application.

See `go.vumitools.contact.Contact` for a look at the Contact model and its fields.

**`handle_get`**(**args*, ***kwargs*)

Accepts a delivery class and address and returns a contact's data, as well as the success flag of the operation (can be `true` or `false`).

**Command fields:**

- `delivery_class`: the type of channel used for the passed in address. Can be one of the following types: `sms`, `ussd`, `twitter`, `gtalk`, `mxit`, `wechat`.
- `addr`: The address to use to lookup of the contact. For example, if `sms` was the delivery class, the address would look something like `+27731112233`

**Success reply fields:**

- `success`: set to `true`
- `contact`: An object containing the contact's data. Looks something like this:

```
{
    'key': 'f953710a2472447591bd59e906dc2c26',
    'surname': 'Person',
    'user_account': 'test-0-user',
    'bbm_pin': null,
    'msisdn': '+27831234567',
    'created_at': '2013-04-24 14:01:41.803693',
    'gtalk_id': null,
    'dob': null,
    'groups': ['group-a', 'group-b'],
    'facebook_id': null,
    '$VERSION': null,
    'twitter_handle': null,
    'mxit_id': null,
    'wechat_id': null,
    'email_address': null,
    'name': 'A Random'
}
```

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure

Example:

```
api.request(
    'contacts.get',
    {delivery_class: 'sms', addr: '+27731112233'},
    function(reply) { api.log_info(reply.contact.name); });
```

**`handle_get_by_key`**(**args*, ***kwargs*)

Retrieve a contact object by key

**Command fields:**

- `key`: The key identifying an existing contact.

**Success reply fields:**

- `success`: set to `true`
- `contact`: An object containing the contact's data

**Failure reply fields:**

- `success`: set to `false`
- `reason`: A string describing the reason for the failure

Examples:

Retrieve a contact which is known to have key '391cea45-ae51-441c-b972-5de765c7a0dc'.

```
api.request(
    'contacts.get_by_key', {
        key: '391cea45-ae51-441c-b972-5de765c7a0dc',
    },
    function(reply) { api.log_info(reply.contact); });
```

**handle_get_or_create**(**args*, ***kwargs*)

Similar to `handle_get()`, but creates the contact if it does not yet exist.

**Success reply fields:**

- `success`: set to `true`
- `contact`: An object containing the contact's data
- `created`: `true` if a new contact was created, otherwise `false`

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure

**handle_new**(**args*, ***kwargs*)

Creates a new contacts with the given fields of an existing contact.

**Command fields:**

- `contact`: The contact data to initialise the new contact with.

**Success reply fields:**

- `success`: set to `true`
- `contact`: An object containing the contact's data.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure

Example:

```
api.request(
    'contacts.new',
    {contact: {surname: 'Jones', extra: {location: 'CPT'}}},
    function(reply) { api.log_info(reply.success); });
```

**handle_save**(**args*, ***kwargs*)

Saves a contact's data, overwriting the contact's previous data. Use with care. This operation only works for existing contacts. For creating new contacts, use `handle_new()`.

**Command fields:**

- `contact`: The contact's data. **Note**: `key` must be a field in the contact data in order identify the contact.

**Success reply fields:**

- `success`: set to `true`
- `contact`: An object containing the contact's data.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure

Example:

```
api.request(
    'contacts.save', {
        contact: {
            'key': 'f953710a2472447591bd59e906dc2c26',
            'surname': 'Person',
            'user_account': 'test-0-user',
            'msisdn': '+27831234567',
            'groups': ['group-a', 'group-b'],
            'name': 'A Random'
        }
    },
    function(reply) { api.log_info(reply.success); });
```

**handle_search**(**args*, ***kwargs*)

Search for contacts

**Command fields:**

- `query`: The Lucene search query to perform.
- **`max_keys`: If present, a non-negative number that specifies** the maximum number of keys to return in the result. By default keys for all matching contacts are returned.

**Success reply fields:**

- `success`: set to `true`
- `keys`: A list of keys for matching contacts.

Note: If no matches are found `keys` will be an empty list.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure

Examples:

Searching on a single contact field:

```
api.request(
    'contacts.search', {
         query: 'name:"My Name"',
    },
    function(reply) { api.log_info(reply.keys); });
```

**handle_update**(**args*, ***kwargs*)
Updates the given fields of an existing contact.

**Note**: All subfields of a Dynamic field such as `extra` and `subscription` are overwritten if specified as one of the fields to be updated.

**Command fields:**

- `key`: The contacts key
- `fields`: The contact fields to be updated

**Success reply fields:**

- `success`: set to `true`
- `contact`: An object containing the contact's data.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure

Example:

```
api.request(
    'contacts.update', {
        key: 'f953710a2472447591bd59e906dc2c26',
        fields: {surname: 'Jones', extra: {location: 'CPT'}}},
    function(reply) { api.log_info(reply.success); });
```

**handle_update_extras**(*api*, *command*)
Updates subfields of an existing contact's `extra` field.

**Command field:**

- `key`: The contact's key
- `fields`: The extra fields to be updated

**Success reply fields:**

- `success`: set to `true`
- `contact`: An object containing the contact's data.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure

Example:

```
api.request(
    'contacts.update_extras', {
        key: f953710a2472447591bd59e906dc2c26',
        fields: {location: 'CPT', beer: 'Whale Tail Ale'}},
    function(reply) { api.log_info(reply.success); });
```

**handle_update_subscriptions**(*api*, *command*)
Updates subfields of an existing contact's `subscription` field.

**Command field:**

- `key`: The contact's key

- `fields`: The subscription fields to be updated

**Success reply fields:**

- `success`: set to `true`
- `contact`: An object containing the contact's data.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure

Example:

```
api.request(
    'contacts.update_subscriptions', {
        key: f953710a2472447591bd59e906dc2c26',
        fields: {a: 'one', b: 'two'}},
    function(reply) { api.log_info(reply.success); });
```

**groups**

This resource provides access to the groups stored in Vumi Go. It allows you to find, create and update group information and retrieve their member counts.

**class** `go.apps.jsbox.contacts.`**`GroupsResource`**(*name*, *app_worker*, *config*)

Bases: `vxsandbox.resources.utils.SandboxResource`

Sandbox resource for accessing, creating and modifying groups for a Go application.

See `go.vumitools.contact.ContactGroup` for a look at the Contact model and its fields.

**`handle_count_members`**(**args*, ***kwargs*)

Count the number of members in a group.

**Command fields:**

- `key`: The key of the group to retrieve

**Success reply fields:**

- `success`: set to `true`
- `group`: A dictionary with the group's data.
- `count`: The number of members in this group.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure

Example:

```
api.request(
    'groups.count_members', {
         key: 'a-key'
    },
    function(reply) { api.log_info(reply.group); });
```

**`handle_get`**(**args*, ***kwargs*)

Get a group by its key

**Command fields:**

- `key`: The key of the group to retrieve

**Success reply fields:**

- `success`: set to `true`
- `group`: A dictionary with the group's data.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure

Example:

```
api.request(
    'groups.get', {
        key: 'a-key',
    },
    function(reply) { api.log_info(reply.group); });
```

**handle_get_by_name**(**args*, ***kwargs*)

Get a group by its name

**Command fields:**

- `name`: The key of the group to retrieve

**Success reply fields:**

- `success`: set to `true`
- `group`: A dictionary with the group's data.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure

**Note: If more than 1 matching groups are found a Failure reply is** returned.

Example:

```
api.request(
    'groups.get_by_name', {
        name: 'My Group',
    },
    function(reply) { api.log_info(reply.group); });
```

**handle_get_or_create_by_name**(**args*, ***kwargs*)

Get or create a group by its name

**Command fields:**

- `name`: The name of the group to get or create

**Success reply fields:**

- `success`: set to `true`
- `group`: A dictionary with the group's data.
- `created`: A boolean, `True` if created, `False` if not.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure

Example:

```
api.request(
    'groups.get_or_create_by_name', {
        name: 'My Group',
    },
    function(reply) { api.log_info(reply.group); });
```

**handle_list**(**args*, ***kwargs*)

List all known groups

Command fields: None

**Success reply fields:**

- `success`: set to `true`
- `groups`: A list of dictionaries with group data

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure

Example:

```
api.request(
    'groups.list', {},
    function(reply) { api.log_info(reply.groups); });
```

**handle_search**(**args*, ***kwargs*)

Search for groups

**Command fields:**

- `query`: The Lucene search query to perform.

**Success reply fields:**

- `success`: set to `true`
- `groups`: An list of dictionaries with group information.

Note: If no matches are found `groups` will be an empty list.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure

Example:

```
api.request(
    'groups.search', {
        query: 'name:"My Group"',
    },
    function(reply) { api.log_info(reply.groups); });
```

**handle_update**(**args*, ***kwargs*)

Update a group's name or query.

**Command fields:**

- `key`: The key of the group to retrieve
- `name`: The new name
- `query`: The query to store, defaults to `None`.

**Note: If a `query` is provided the group is treated as a** "smart" group.

**Success reply fields:**

- `success`: set to `true`
- `group`: A dictionary with the group's updated data.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure

Example:

```
api.request(
    'groups.update', {
         key: 'a-key',
         name: 'My New Group',
         query: 'name:foo*'
    },
    function(reply) { api.log_info(reply.group); });
```

**log**

Provides logging facilities for your application. These logs are available for viewing in the UI. The 1000 most recent log entries are stored.

**class** `go.apps.jsbox.log.`**`GoLoggingResource`**(*name*, *app_worker*, *config*)
Bases: `vxsandbox.resources.logging.LoggingResource`

Resource that allows a sandbox to log messages.

Messages are logged both via Twisted's logging framework and to a per-conversation log store in Redis.

**kv**

Provides access to a Redis backed key value store. GET, SET and INCR operations are available. There is a limit of 10000 keys per user.

**class** `vumi.application.sandbox.`**`RedisResource`**(*name*, *app_worker*, *config*)
Bases: `vumi.application.sandbox.SandboxResource`

Resource that provides access to a simple key-value store.

Configuration options:

**Parameters**

- **`redis_manager`** (*dict*) – Redis manager configuration options.
- **`keys_per_user_soft`** (*int*) – Maximum number of keys each user may make use of in redis before usage warnings are logged. (default: 80% of hard limit).
- **`keys_per_user_hard`** (*int*) – Maximum number of keys each user may make use of in redis (default: 100). Falls back to keys_per_user.
- **`keys_per_user`** (*int*) – Synonym for *keys_per_user_hard*. Deprecated.

**handle_delete**(**args*, ***kwargs*)
Delete a key.

**Command fields:**

- `key`: The key to delete.

**Reply fields:**

- `success`: `true` if the operation was successful, otherwise `false`.

Example:

```
api.request(
    'kv.delete',
    {key: 'foo'},
    function(reply) {
        api.log_info('Value deleted: ' +
                     reply.success);
    }
);
```

**handle_get**(**args*, ***kwargs*)
Retrieve the value of a key.

**Command fields:**

- `key`: The key whose value should be retrieved.

**Reply fields:**

- `success`: `true` if the operation was successful, otherwise `false`.
- `value`: The value retrieved.

Example:

```
api.request(
    'kv.get',
    {key: 'foo'},
    function(reply) {
        api.log_info(
            'Value retrieved: ' +
            JSON.stringify(reply.value));
    }
);
```

**handle_incr**(**args*, ***kwargs*)
Atomically increment the value of an integer key.

The current value of the key must be an integer. If the key does not exist, it is set to zero.

**Command fields:**

- `key`: The key to delete.
- `amount`: The integer amount to increment the key by. Defaults to 1.

**Reply fields:**

- `success`: `true` if the operation was successful, otherwise `false`.
- `value`: The new value of the key.

Example:

```
        api.request(
            'kv.incr',
            {key: 'foo',
             amount: 3},
            function(reply) {
                api.log_info('New value: ' +
                             reply.value);
            }
        );
```

**handle_set**(**args*, ***kwargs*)
Set the value of a key.

**Command fields:**

- `key`: The key whose value should be set.
- `value`: The value to store. May be any JSON serializable object.
- `seconds`: Lifetime of the key in seconds. The default `null` indicates that the key should not expire.

**Reply fields:**

- `success`: `true` if the operation was successful, otherwise `false`.

Example:

```
        api.request(
            'kv.set',
            {key: 'foo',
             value: {x: '42'}},
            function(reply) { api.log_info('Value store: ' +
                                           reply.success); });
```

**messagestore**

Provides access to the stats in the messagestore. Specifically counts of messages sent, received, unique "from_addr"s and "to_addr"s and calculated throughput of any conversation linked to a Vumi Go account.

**class** `go.apps.jsbox.message_store.`**MessageStoreResource**(*name*, *app_worker*, *config*)
Bases: `vxsandbox.resources.utils.SandboxResource`

**handle_count_inbound_uniques**(**args*, ***kwargs*)
Count from how many unique "from_addr"s messages were received.

If no conversation is specified then the current application's conversation is used.

**Command fields:**

- `conversation_key`: The key of the conversation to use. This is optional, if not specified the application's own conversation is used.

**Success reply fields:**

- `success`: set to `true`
- `count`: the number of unique "from_addr"s messages were sent.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure.

**handle_count_outbound_uniques**(**args*, ***kwargs*)
Count to how many unique "to_addr"s messages were sent.

If no conversation is specified then the current application's conversation is used.

**Command fields:**

- `conversation_key`: The key of the conversation to use. This is optional, if not specified the application's own conversation is used.

**Success reply fields:**

- `success`: set to `true`
- `count`: the number of unique "to_addrs"s messages were sent.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure.

**handle_count_replies**(**args*, ***kwargs*)
Count how many messages were received in the conversation.

If no conversation is specified then the current application's conversation is used.

**Command fields:**

- `conversation_key`: The key of the conversation to use. This is optional, if not specified the application's own conversation is used.

**Success reply fields:**

- `success`: set to `true`
- `count`: the number of messages received

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure.

**handle_count_sent_messages**(**args*, ***kwargs*)
Count how many messages were sent in the conversation.

If no conversation is specified then the current application's conversation is used.

**Command fields:**

- `conversation_key`: The key of the conversation to use. This is optional, if not specified the application's own conversation is used.

**Success reply fields:**

- `success`: set to `true`
- `count`: the number of messages sent

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure.

**handle_inbound_throughput**(**args*, ***kwargs*)
Count how many messages a minute were received.

If no conversation is specified then the current application's conversation is used.

**Command fields:**

- `conversation_key`: The key of the conversation to use. This is optional, if not specified the application's own conversation is used.
- `sample_time`: How far to look back to calculate the throughput. Defaults to 300 seconds (5 minutes)

**Success reply fields:**

- `success`: set to `true`
- `throughput`: how many inbound messages per minute the conversation has done on average.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure.

**handle_outbound_throughput**(**args*, ***kwargs*)
Count how many messages a minute were sent.

If no conversation is specified then the current application's conversation is used.

**Command fields:**

- `conversation_key`: The key of the conversation to use. This is optional, if not specified the application's own conversation is used.
- `sample_time`: How far to look back to calculate the throughput. Defaults to 300 seconds (5 minutes)

**Success reply fields:**

- `success`: set to `true`
- `throughput`: how many outbound messages per minute the conversation has done on average.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure.

**handle_progress_status**(**args*, ***kwargs*)
Accepts a conversation_key and retrieves the progress_status breakdown for that conversation.

If no conversation is specified then the current application's conversation is used.

**Command fields:**

- `conversation_key`: The key of the conversation to use. This is optional, if not specified the application's own conversation is used.

**Success reply fields:**

- `success`: set to `true`
- `progress_status`: A dictionary with a break down of the conversations progress status:

```
            {
                'ack': 1,
                'delivery_report': 0,
                'delivery_report_delivered': 0,
                'delivery_report_failed': 0,
                'delivery_report_pending': 0,
                'nack': 0,
                'sent': 1,
            }
```

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure.

**optout**

Provides access to the opt-out status of contacts. Allows one to check, count and change the opt-out status of `address_type` and `address_value` pairs.

---

**Note:** This resource needs to be enabled on a per-account basis. By default it is disabled for all accounts.

---

**class** `go.apps.jsbox.opt_out.`**`OptOutResource`**(*name*, *app_worker*, *config*)

Bases: `vxsandbox.resources.utils.SandboxResource`

**`handle_cancel_optout`**(*api*, *command*)

Cancel an opt-out, effectively opting an address_type & address_value combination back in.

**Command fields:**

- `address_type`: the type of address cancel the opt-out for.
- `address_value`: the value of the `address_type` to cancel the opt-out for.

**Success reply fields:**

- `success`: set to `true`
- `opted_out`: set to `false`

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure.

**`handle_count`**(**args*, ***kwargs*)

Return a count of however many opt-outs there are

Command fields: None

**Success reply fields:**

- `success`: set to `true`
- `count`: an Integer.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure.

**`handle_optout`**(*api*, *command*)

Opt out an address_type, address_value combination

**Command fields:**

- `address_type`: the type of address to opt-out. At the moment only `msisdn` is used.
- `address_value`: the value of the `address_type` to opt-out.
- `message_id` the message_id of the message that triggered the opt-out, for auditing purposes.

**Success reply fields:**

- `success`: set to `true`
- `opted_out`: set to `true`
- `created_at`: the timestamp of the opt-out
- `message_id`: the message_id of the message that triggered the opt-out.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure.

**`handle_status`**(*api*, *command*)

Accepts an address_type and address_value and retrieves the opt-out entry for it.

**Command fields:**

- `address_type`: the type of address to check for opt-out status on. At the moment only `msisdn` is used.
- `address_value`: the value of the `address_type` to check. At the moment this would be a normalized msisdn.

**Success reply fields:**

- `success`: set to `true`
- `opted_out`: set to `true` or `false`
- `created_at`: the timestamp of the opt-out (if opted out)
- `message_id`: the message_id of the message that triggered the opt-out.

**Failure reply fields:**

- `success`: set to `false`
- `reason`: Reason for the failure.

CHAPTER 3

# Example Code for Javascript Sandbox features

The way the Javascript sandbox talks to the outside world can sometimes be a bit un-intuitive when first starting with Vumi Go development.

Here are some code samples to give you a head start:

## 3.1 Using the Key-Value store:

The KV store allows for GET, SET and INCR operations. These are namespaced to your account unless you explicitly namespace it differently per conversation. This allows you to share counters across different applications.

If you're looking for DECR just use INCR with a negative value.

https://github.com/smn/go-kv-store

## 3.2 Firing Events from an Application

Metrics are crucial for any application. Our metrics backend is powered by Graphite and your application can send metrics to Graphite for aggregation.

The metrics are not yet visible within the UI but hopefully will be graphed there soon.

https://github.com/smn/go-events-firing

## 3.3 Using the HTTP API

The HTTP API allows for interacting with 3rd party applications that are not hosted on our platform. It allows for both streaming of messages in real time or using HTTP POST to forward message (and message events) to a remote URL.

**Django backend for HTTP forwarding setup:** https://github.com/smn/go-heroku

**Node.js backend for consuming the Streaming API:** https://github.com/smn/go-heroku-streaming

CHAPTER 4

# Vumi Go's HTTP API

The API allows for sending & receiving Vumi messages via HTTP. These messages are plain JSON strings. Three types of messages are available:

- Inbound and outbound user messages (e.g. SMSes, USSD responses, Twitter messages)
- Events (e.g. delivery reports, acknowledgements)
- Metrics (values recorded at a specific time)

Inbound user messages and events can be received via streaming HTTP or can be pushed to a third party URL via HTTP POST. Outbound messages and metrics can be pushed to Vumi Go via HTTP PUT.

Each HTTP api is bound to a conversation which stores all of the messages sent and received. HTTP Basic auth is used for authentication, the username is the Vumi Go account key and the password is an access token that is stored in the conversation. In order to connect three keys are required:

1. The account key
2. The accesss token
3. The conversation key

## 4.1 Inbound and Outbound User Messages

This is the format for messages being sent to, or received from, a person.

User messages are JSON objects of the following format:

```
{
    "message_id": "59b37288d8d94e42ab804158bdbf53e5",
    "in_reply_to": null,
    "session_event": null,
    "to_addr": "1234",
    "to_addr_type": "msisdn",
    "from_addr": "+27761234567",
    "from_addr_type": "msisdn",
    "content": "This is an incoming SMS!",
    "transport_name": "smpp_transport",
    "transport_type": "sms",
    "transport_metadata": {
        // this is a dictionary containing
        // transport specific data
    },
    "helper_metadata": {
```

```
        // this is a dictionary containing
        // application specific data
    }
}
```

A reply to this message would put the value of the "message_id" in the "in_reply_to" field so as to link the two.

The *from_addr_type* and *to_addr_type* fields describe the type of address declared in *from_addr* and *to_addr*. The default for *to_addr_type* is *msisdn*, and the default for *from_addr_type* is *null*, which is used to mark that the type is unspecified. The other valid values are *irc_nickname*, *twitter_handle*, *gtalk_id*, *jabber_id*, *mxit_id*, and *wechat_id*.

The "session_event" field is used for transports that are session oriented, primarily USSD. This field will be either "null", "new", "resume" or "close". There are no guarantees that these will be set for USSD as it depends on the networks whether or not these values are available. If replying to a message in USSD session then set the "session_event" to "resume" if you are expecting a reply back from the user or to "close" if the message you are sending is the last message and the session is to be closed.

The go-heroku application is an example app that uses the HTTP API to receive and send messages.

A Python client for the HTTP API is available at https://github.com/praekelt/go-http-api. It can be installed with `pip install go-http`.

## 4.2 Sending Messages

```
$ curl -X PUT \
       --user '<account-key>:<access-token>' \
       --data '{"in_reply_to": "59b37288d8d94e42ab804158bdbf53e5", \
                "to_addr": "+27761234567", \
                "to_addr_type": "msisdn", \
                "content": "This is an outgoing SMS!"}' \
       http://go.vumi.org/api/v1/go/http_api_nostream/<conversation-key>/messages.json \
       -vvv
```

The UI expects you to specify an access token. All requests to the API require you to use your account key as the username and the token as the password.

The response to the PUT request is the complete Vumi Go user message and includes the generated Vumi `message_id` which should be stored if you wish to be able to associate events with the message later.

If a message is sent to a recipient that has opted out, the response will be an HTTP 400 error, with the body detailing that the recipient has opted out. Messages sent as a reply will still go through to an opted out recipient. The following is an example response of the error returned by the API:

```
{
    "success": false,
    "reason": "Recipient with msisdn +12345 has opted out"
}
```

This behaviour can be overridden by setting the *disable_optout* flag in the account to *True*. Ask a Vumi Go admin if you need to have optouts disabled.

## 4.3 Receiving User Messages

Vumi Go will forward any inbound messages to your application via an HTTP POST. Please specify the URL in the Go UI. You can include a username and password in the URL and use HTTPS if you require authentication.

There is a separate URL for receiving events.

## 4.4 Events

This is the format for events. Each event is associated with an outbound user message.

Events are JSON messages with the following format:

```
{
    "message_type": "event",
    "event_id": "b04ec322fc1c4819bc3f28e6e0c69de6",
    "event_type": "ack",
    "user_message_id": "60c48289d8d94e42ab804159acce42d4",
    "helper_metadata": {
        // this is a dictionary containing
        // application specific data
    },
    "timestamp": "2014-10-28 16:19:37.485612",
    "sent_message_id": "external-id",
}
```

The `event_id` unique id for this event.

The `user_message_id` is the id of the outbound message the event is for (this should be returned to you when you post the message to the HTTP API).

The `event_type` is the type of event and can be either `ack`, `nack` or `delivery_report`.

An `ack` indicates that the outbound message was succesfully sent to a third party (e.g. a cellphone network operator) for sending. A `nack` indicates that the message was not successfully sent to a third party and should be resent. The reason the message could not be sent will be given in the `nack_reason` field. Every outbound message should receive either an `ack` or a `nack` event.

A `delivery_report` indicates whether a message has successfully reached it's final destination (e.g. a cellphone). Delivery reports are only available for some SMS channels. The delivery status will be given in the `delivery_status` field and can be one of `pending` (SMS is still waiting to be delivered to the cellphone), `failed` (the cellphone operator has given up attempting to deliver the SMS) or `delivered` (the SMS was successfully delivered to the cellphone).

---

**Note:** The meaning of delivery statuses can vary subtly between cellphone operators and should not be relied upon without careful testing of your specific use case.

---

## 4.5 Receiving Events

Vumi Go will forward any events to your application via an HTTP POST. Please specify the URL in the Go UI. You can include a username and password in the URL and use HTTPS if you require authentication.

This is a separate URL to the one for receiving user messages.

## 4.6 Publishing Metrics

You are able to publish metrics to Vumi Go via the HTTP APIs metrics endpoint. These metrics are able to be displayed in the Vumi GO UI using the dashboards.

How these dashboards are configured is explained in *Vumi Go Dashboards*.

```
PUT http://go.vumi.org/api/v1/go/http_api_nostream/<conversation-key>/metrics.json
```

An example using curl from the commandline:

```
$ curl -X PUT \
    --user '<account-key>:<access-token>' \
    --data '[["total_pings", 1200, "MAX"]]' \
    https://go.vumi.org/api/v1/go/http_api_nostream/<conversation-key>/metrics.json \
    -vvv
```

CHAPTER 5

# Vumi Go Dashboards

Vumi has growing support for dashboards. These dashboards are backed by Graphite and Vumi applies some bucketing of metrics before publishing to Graphite.

**Note:** Graphite itself is not directly accessible for 3rd party applications.

## 5.1 Conversation Types supporting Dashboards

The visualization of dashboards is currently only supported by the Javascript sandbox conversation type.

The publishing of metrics to a dashboard is supported by the *Vumi Go's HTTP API* and the Javascript application conversation types.

Metrics can be shared between conversations if they share the same metrics store value.

## 5.2 Dashboard Setup

Dashboards are set up by creating a Javascript application type and ensuring there is a `reports` section that has a JSON dashboard description.

The report defines:

1. The layout
2. The widgets
3. The metrics

Here is a sample reports.json file from go-events-firing-via-http

```
{
  "layout": [
    {
      "type": "diamondash.widgets.lvalue.LValueWidget",
      "time_range": "1d",
      "name": "Last Ping Count",
      "target": {
        "metric_type": "account",
        "store": "default",
        "name": "total_pings",
        "aggregator": "max"
```

```
      }
    },
    "new_row",
    {
      "type": "diamondash.widgets.graph.GraphWidget",
      "name": "Ping Counts",
      "width": 6,
      "time_range": "1d",
      "bucket_size": "1h",
      "metrics": [
        {
          "name": "Pings",
          "target": {
            "metric_type": "account",
            "store": "default",
            "name": "total_pings",
            "aggregator": "max"
          }
        }
      ]
    }
  ]
}
```

Once a `reports` section is created the Vumi Go UI will read the definition and render the dashboard widgets for you in the Reports page.

## 5.3 Available Widgets

The following widgets are available:

### 5.3.1 LValueWidget

Displays two values of a metrics, the current value and the value of the metric a configurable time ago. The time difference between the two values is determined by the `time_range` value.

```
{
  "type": "diamondash.widgets.lvalue.LValueWidget",
  "time_range": "1d",
  "name": "Last Ping Count",
  "target": {
    "metric_type": "account",
    "store": "default",
    "name": "total_pings",
    "aggregator": "max"
  }
}
```

### 5.3.2 GraphWidget

Displays a line graph. Multiple metrics can be rendered on the same graph.

```
{
  "type": "diamondash.widgets.graph.GraphWidget",
  "name": "Ping Counts",
  "width": 6,
  "time_range": "1d",
  "bucket_size": "1h",
  "metrics": [
    {
      "name": "Pings",
      "target": {
        "metric_type": "account",
        "store": "default",
        "name": "total_pings",
        "aggregator": "max"
      }
    }
  ]
}
```

### 5.3.3 HistogramWidget

Display a histogram of the metrics.

```
{
    "type": "diamondash.widgets.histogram.HistogramWidget",
    "name": "Total Pings (Histogram)",
    "target": {
      "metric_type": "account",
      "store": "default",
      "name": "total_pings",
      "aggregator": "max"
    },
    "time_range": "1h",
    "bucket_size": "5m",
    "width": 6
}
```

### 5.3.4 PieWidget

Display a pie chart of metric values.

```
{
    "type": "diamondash.widgets.pie.PieWidget",
    "name": "Total Pings (Pie)",
    "time_range": "1d",
    "width": 6,
    "metrics": [
      {
        "name": "Total Pings",
        "target": {
          "metric_type": "account",
          "store": "default",
          "name": "total_pings",
          "aggregator": "max"
        }
      }
```

```
    ]
}
```

For developers:

CHAPTER 6

# Vumi Go Architecture

## 6.1 Notes

- indicates that a component requires requests to be authenticated (i.e. it is intended to be exposed as a public service).
- Public services are expected to use Nginx internal redirects to return results of specific requests to internal services as needed.

CHAPTER 7

# Vumi Go Messaging Architecture

CHAPTER 8

# Indices and tables

- genindex
- modindex
- search

## A



## C



## G



## H

## M



## O



## R