# Brewtils Documentation

***Release 2.3.2***

**Logan Asher Jones**

**Mar 07, 2018**

# Contents

Contents:

CHAPTER 1

# Brewtils

Brewtils is the Python library for interfacing with Beergarden systems. If you are planning on writing beer-garden plugins, this is the correct library for you. In addition to writing plugins, it provides simple ways to query the API and is officially supported by the beer-garden team.

## 1.1 Features

Brewtils helps you interact with beer-garden.

- Easy way to create beer-garden plugins
- Full support of the entire Beer-Garden API
- Officially supported by the beer-garden team

## 1.2 Installation

To install brewtils, run this command in your terminal:

```
$ pip install brewtils
```

Or add it to your `requirements.txt`

```
$ cat brewtils >> requirements.txt
$ pip install -r requirements.txt
```

## 1.3 Quick Start

You can create your own beer-garden plugins without much problem at all. To start, we'll create the obligatory hello-world plugin. Creating a plugin is as simple as:

```
from brewtils.decorators import system, parameter, command
from brewtils.plugin import RemotePlugin

@system
class HelloWorld(object):

    @parameter(key="message", description="The message to echo", type="String")
    def say_hello(self, message="World!"):
        print("Hello, %s!" % message)
        return "Hello, %s!" % message

if __name__ == "__main__":
    client = HelloWorld()
    plugin = RemotePlugin(client,
                          name="hello",
                          version="0.0.1",
                          bg_host='127.0.0.1',
                          bg_port=2337)
    plugin.run()
```

Assuming you have a Beer Garden running on port 2337 on localhost, running this will register and start your plugin! You now have your first plugin running in beer-garden. Let's use another part of the `brewtils` library to exercise your plugin from python.

The `SystemClient` is designed to help you interact with registered Systems as if they were native Python objects.

```
from brewtils.rest.system_client import SystemClient

hello_client = SystemClient('localhost', 2337, 'hello')

request = hello_client.say_hello(message="from system client")

print(request.status) # 'SUCCESS'
print(request.output) # Hello, from system client!
```

In the background, the `SystemClient` has executed an HTTP POST with the payload required to get beer-garden to execute your command. The `SystemClient` is how most people interact with beer-garden when they are in the context of python and want to be making requests.

Of course, the rest of the API is accessible through the `brewtils` package. The `EasyClient` provides simple convenient methods to call the API and auto-serialize the responses. Suppose you want to get a list of all the commands on all systems:

```
from brewtils.rest.easy_client import EasyClient

client = EasyClient('localhost', 2337)

systems = client.find_systems()

for system in systems:
    for command in system.commands:
        print(command.name)
```

This is just a small taste of what is possible with the `EasyClient`. Feel free to explore all the methods that are exposed.

For more detailed information and better walkthroughs, checkout the full documentation!

## 1.4 Documentation

- Full Beer Garden documentation is available at https://beer-garden.io
- Brewtils Documentation is available at https://brewtils.readthedocs.io

CHAPTER 2

# Installation

## 2.1 Stable release

To install Brewtils, run this command in your terminal:

```
$ pip install brewtils
```

This is the preferred method to install Brewtils, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

## 2.2 From sources

The sources for Brewtils can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git@github.com/beer-garden/brewtils.git
```

Or download the tarball:

```
$ curl  -OL https://github.com/beer-garden/brewtils/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

# Usage

CHAPTER 4

# Brewtils

Brewtils is the Python library for interfacing with Beergarden systems. If you are planning on writing beer-garden plugins, this is the correct library for you. In addition to writing plugins, it provides simple ways to query the API and is officially supported by the beer-garden team.

## 4.1 Features

Brewtils helps you interact with beer-garden.

- Easy way to create beer-garden plugins
- Full support of the entire Beer-Garden API
- Officially supported by the beer-garden team

## 4.2 Installation

To install brewtils, run this command in your terminal:

```
$ pip install brewtils
```

Or add it to your `requirements.txt`

```
$ cat brewtils >> requirements.txt
$ pip install -r requirements.txt
```

## 4.3 Quick Start

You can create your own beer-garden plugins without much problem at all. To start, we'll create the obligatory hello-world plugin. Creating a plugin is as simple as:

```
from brewtils.decorators import system, parameter, command
from brewtils.plugin import RemotePlugin

@system
class HelloWorld(object):

    @parameter(key="message", description="The message to echo", type="String")
    def say_hello(self, message="World!"):
        print("Hello, %s!" % message)
        return "Hello, %s!" % message

if __name__ == "__main__":
    client = HelloWorld()
    plugin = RemotePlugin(client,
                          name="hello",
                          version="0.0.1",
                          bg_host='127.0.0.1',
                          bg_port=2337)
    plugin.run()
```

Assuming you have a Beer Garden running on port 2337 on localhost, running this will register and start your plugin! You now have your first plugin running in beer-garden. Let's use another part of the `brewtils` library to exercise your plugin from python.

The `SystemClient` is designed to help you interact with registered Systems as if they were native Python objects.

```
from brewtils.rest.system_client import SystemClient

hello_client = SystemClient('localhost', 2337, 'hello')

request = hello_client.say_hello(message="from system client")

print(request.status) # 'SUCCESS'
print(request.output) # Hello, from system client!
```

In the background, the `SystemClient` has executed an HTTP POST with the payload required to get beer-garden to execute your command. The `SystemClient` is how most people interact with beer-garden when they are in the context of python and want to be making requests.

Of course, the rest of the API is accessible through the `brewtils` package. The `EasyClient` provides simple convenient methods to call the API and auto-serialize the responses. Suppose you want to get a list of all the commands on all systems:

```
from brewtils.rest.easy_client import EasyClient

client = EasyClient('localhost', 2337)

systems = client.find_systems()

for system in systems:
    for command in system.commands:
        print(command.name)
```

This is just a small taste of what is possible with the `EasyClient`. Feel free to explore all the methods that are exposed.

For more detailed information and better walkthroughs, checkout the full documentation!

## 4.4 Documentation

- Full Beer Garden documentation is available at https://beer-garden.io
- Brewtils Documentation is available at https://brewtils.readthedocs.io

CHAPTER 5

# brewtils

## 5.1 brewtils package

### 5.1.1 Subpackages

#### brewtils.log package

#### Module contents

Brewtils Logging Utilities

This module is for setting up your plugins logging correctly.

#### Example

In order to use this, you should simply call `setup_logger` in the same file where you initialize your plugin sometime before you initialize your Plugin object.

```
host = 'localhost'
port = 2337
ssl_enabled = False
system_name = 'my_system'

setup_logger(bg_host=host, bg_port=port, system_name=system_name, ssl_enabled=ssl_
↪enabled)
plugin = Plugin(my_client, bg_host=host, bg_port=port, ssl_enabled=ssl_enabled,
                name=system_name, version="0.0.1")
plugin.run()
```

brewtils.log.**convert_logging_config**(*logging_config*)

Converts a LoggingConfig object into a python logging configuration

The python logging configuration that is returned can be passed to *logging.config.dictConfig*

**Parameters** **logging_config** –

**Returns** Python logging configuration

brewtils.log.**get_python_logging_config**(*bg_host*, *bg_port*, *system_name*, *ca_cert=None*, *client_cert=None*, *ssl_enabled=None*)

Returns a dictionary for the python logging configuration

**Parameters**

- **bg_host** (*str*) – Hostname of a beer-garden
- **bg_port** (*int*) – Port beer-garden is listening on
- **system_name** (*str*) – The system
- **ca_cert** – Certificate that issued the server certificate used by the beer-garden server
- **client_cert** – Certificate used by the server making the connection to beer-garden
- **ssl_enabled** (*bool*) – Whether to use SSL for beer-garden communication

**Returns** Python logging configuration

brewtils.log.**setup_logger**(*bg_host*, *bg_port*, *system_name*, *ca_cert=None*, *client_cert=None*, *ssl_enabled=None*)

Configures python logging module to use logging specified in beer-garden API.

This method will overwrite your current logging configuration, so only call it if you want beer-garden's logging configuration.

**Parameters**

- **bg_host** (*str*) – Hostname of a beer-garden
- **bg_port** (*int*) – Port beer-garden is listening on
- **system_name** (*str*) – The system
- **ca_cert** – Certificate that issued the server certificate used by the beer-garden server
- **client_cert** – Certificate used by the server making the connection to beer-garden
- **ssl_enabled** (*bool*) – Whether to use SSL for beer-garden communication

**Returns**

## brewtils.rest package

## Submodules

## brewtils.rest.client module

**class** brewtils.rest.client.**BrewmasterRestClient**(**args*, ***kwargs*)

Bases: *brewtils.rest.client.RestClient*

**class** brewtils.rest.client.**RestClient**(*host*, *port*, *ssl_enabled=False*, *api_version=None*, *logger=None*, *ca_cert=None*, *client_cert=None*, *url_prefix=None*, *ca_verify=True*)

Bases: object

Simple Rest Client for communicating to with beer-garden.

The is the low-level client responsible for making the actual REST calls. Other clients (e.g. *brewtils.rest.easy_client.EasyClient*) build on this by providing useful abstractions.

**Parameters**

- **host** – beer-garden REST API hostname.
- **port** – beer-garden REST API port.
- **ssl_enabled** – Flag indicating whether to use HTTPS when communicating with beer-garden.
- **api_version** – The beer-garden REST API version. Will default to the latest version.
- **logger** – The logger to use. If None one will be created.
- **ca_cert** – beer-garden REST API server CA certificate.
- **client_cert** – The client certificate to use when making requests.
- **url_prefix** – beer-garden REST API Url Prefix.
- **ca_verify** – Flag indicating whether to verify server certificate when making a request.

**JSON_HEADERS = {'Accept': 'text/plain', 'Content-type': 'application/json'}**

**LATEST_VERSION = 1**

**delete_queue**(*queue_name*)
Performs a DELETE on a specific Queue URL

**Returns** Response to the request

**delete_queues**()
Performs a DELETE on the Queues URL

**Returns** Response to the request

**delete_system**(*system_id*)
Performs a DELETE on a System URL

**Parameters** **system_id** – The ID of the system to remove

**Returns** Response to the request

**get_command**(*command_id*)
Performs a GET on the Command URL

**Parameters** **command_id** – ID of command

**Returns** Response to the request

**get_commands**()
Performs a GET on the Commands URL

**get_logging_config**(***kwargs*)
Perform a GET to the logging config URL

**Parameters** **kwargs** – Parameters to be used in the GET request

**Returns** The request response

**get_queues**()
Performs a GET on the Queues URL

**Returns** Response to the request

**get_request**(*request_id*)
Performs a GET on the Request URL

**Parameters** **request_id** – ID of request

**Returns** Response to the request

**get_requests**(***kwargs*)
Performs a GET on the Requests URL

**Parameters** **kwargs** – Parameters to be used in the GET request

**Returns** Response to the request

**get_system**(*system_id*, ***kwargs*)
Performs a GET on the System URL

**Parameters**

- **system_id** – ID of system
- **kwargs** – Parameters to be used in the GET request

**Returns** Response to the request

**get_systems**(***kwargs*)
Perform a GET on the System collection URL

**Parameters** **kwargs** – Parameters to be used in the GET request

**Returns** The request response

**get_version**(***kwargs*)
Perform a GET to the version URL

**Parameters** **kwargs** – Parameters to be used in the GET request

**Returns** The request response

**patch_instance**(*instance_id*, *payload*)
Performs a PATCH on the instance URL

**Parameters**

- **instance_id** – ID of instance
- **payload** – The update specification

**Returns** Response

**patch_request**(*request_id*, *payload*)
Performs a PATCH on the Request URL

**Parameters**

- **request_id** – ID of request
- **payload** – New request definition

**Returns** Response to the request

**patch_system**(*system_id*, *payload*)
Performs a PATCH on a System URL

**Parameters**

- **system_id** – ID of system
- **payload** – The update specification

**Returns** Response

**post_event**(*payload*, *publishers=None*)
Performs a POST on the event URL

**Parameters**

- **payload** – New event definition
- **publishers** – Array of publishers to use

**Returns** Response to the request

**post_requests**(*payload*)
Performs a POST on the Request URL

**Parameters** **payload** – New request definition

**Returns** Response to the request

**post_systems**(*payload*)
Performs a POST on the System URL

**Parameters** **payload** – New request definition

**Returns** Response to the request

### brewtils.rest.easy_client module

**class** brewtils.rest.easy_client.**BrewmasterEasyClient**(**args*, ***kwargs*)
Bases: *brewtils.rest.easy_client.EasyClient*

**class** brewtils.rest.easy_client.**EasyClient**(*host*, *port*, *ssl_enabled=False*, *api_version=None*, *ca_cert=None*, *client_cert=None*, *parser=None*, *logger=None*, *url_prefix=None*, *ca_verify=True*)
Bases: object

Client for communicating with beer-garden.

This class provides nice wrappers around the functionality provided by a *brewtils.rest.client.RestClient*

**Parameters**

- **host** – beer-garden REST API hostname.
- **port** – beer-garden REST API port.
- **ssl_enabled** – Flag indicating whether to use HTTPS when communicating with beer-garden.
- **api_version** – The beer-garden REST API version. Will default to the latest version.
- **ca_cert** – beer-garden REST API server CA certificate.
- **client_cert** – The client certificate to use when making requests.
- **parser** – The parser to use. If None will default to an instance of BrewmasterSchemaParser.
- **logger** – The logger to use. If None one will be created.
- **url_prefix** – beer-garden REST API URL Prefix.
- **ca_verify** – Flag indicating whether to verify server certificate when making a request.

**clear_all_queues**()
Cancel and clear all messages from all queues

**Returns** The response

**clear_queue**(*queue_name*)
Cancel and clear all messages from a queue

**Returns** The response

**create_request**(*request*)
Create a new request.

**Parameters** **request** – The request to create

**Returns** The response

**create_system**(*system*)
Create a new system by POSTing to a BREWMASTER server.

**Parameters** **system** – The system to create

**Returns** The system creation response

**find_requests**(***kwargs*)
Find requests using keyword arguments as search parameters.

**Parameters** **kwargs** – Search parameters

**Returns** A list of request instances satisfying the given search parameters

**find_systems**(***kwargs*)
Find systems using keyword arguments as search parameters.

**Parameters** **kwargs** – Search parameters

**Returns** A list of system instances satisfying the given search parameters

**find_unique_request**(***kwargs*)
Find a unique request using keyword arguments as search parameters.

---

**Note:** If 'id' is present in kwargs then all other parameters will be ignored.

---

**Parameters** **kwargs** – Search parameters

**Returns** One request instance

**find_unique_system**(***kwargs*)
Find a unique system using keyword arguments as search parameters.

**Parameters** **kwargs** – Search parameters

**Returns** One system instance

**get_logging_config**(*system_name*)
Get the logging configuration for a particular system.

**Parameters** **system_name** – Name of system

**Returns** LoggingConfig object

**get_queues**()
Retrieve all queue information

**Returns** The response

**get_version**(***kwargs*)

**initialize_instance**(*instance_id*)
Start an instance by PATCHing to a BREWMASTER server.

**Parameters** **instance_id** – The ID of the instance to start

**Returns** The start response

**instance_heartbeat**(*instance_id*)
Send heartbeat to BREWMASTER for health and status purposes

**Parameters** **instance_id** – The ID of the instance

**Returns** The response

**publish_event**(**args*, ***kwargs*)
Publish a new event.

**Parameters**

- **args** – The Event to create
- **_publishers** – Optional list of specific publishers. If None all publishers will be used.
- **kwargs** – If no Event is given in the *args, on will be constructed from the kwargs

**Returns** The response

**remove_system**(***kwargs*)
Remove a specific system using keyword arguments as search parameters.

**Parameters** **kwargs** – Search parameters

**Returns** The response

**update_instance_status**(*instance_id*, *new_status*)
Update an instance by PATCHing to a BREWMASTER server.

**Parameters**

- **instance_id** – The ID of the instance to start
- **new_status** – The updated status

**Returns** The start response

**update_request**(*request_id*, *status=None*, *output=None*, *error_class=None*)
Set various fields on a request with a PATCH

**Parameters**

- **request_id** – The ID of the request to update
- **status** – The new status
- **output** – The new output
- **error_class** – The new error class

**Returns** The response

**update_system**(*system_id*, *new_commands=None*, ***kwargs*)
Update a system with a PATCH

**Parameters**

- **system_id** – The ID of the system to update
- **new_commands** – The new commands

**Keyword Arguments**

- *metadata* (dict) The updated metadata for the system
- *description* (str) The updated description for the system
- *display_name* (str) The updated display_name for the system
- *icon_name* (str) The updated icon_name for the system

**Returns** The response

### brewtils.rest.system_client module

**class** brewtils.rest.system_client.**BrewmasterSystemClient**(**args*, ***kwargs*)
Bases: *brewtils.rest.system_client.SystemClient*

**class** brewtils.rest.system_client.**SystemClient**(*host*, *port*, *system_name*, *version_constraint='latest'*, *default_instance='default'*, *always_update=False*, *timeout=None*, *max_delay=30*, *api_version=None*, *ssl_enabled=False*, *ca_cert=None*, *blocking=True*, *max_concurrent=None*, *client_cert=None*, *url_prefix=None*, *ca_verify=True*)
Bases: object

High-level client for generating requests for a beer-garden System.

**SystemClient creation:** This class is intended to be the main way to create beer-garden requests. Create an instance with beer-garden connection information (optionally including a url_prefix) and a system name:

```
client = SystemClient(host, port, 'example_system', ssl_enabled=True, url_
↪prefix=None)
```

Pass additional keyword arguments for more granularity:

**version_constraint:** Allows specifying a particular system version. Can be a version literal ('1.0.0') or the special value 'latest.' Using 'latest' will allow the the SystemClient to retry a request if it fails due to a missing system (see Creating Requests).

**default_instance:** The instance name to use when creating a request if no other instance name is specified. Since each request must be addressed to a specific instance this is a convenience to prevent needing to specify the 'default' instance for each request.

**always_update:** Always attempt to reload the system definition before making a request. This is useful to ensure Requests are always made against the latest version of the system. If not set the System definition will be loaded once (upon making the first request) and then only reloaded if a Request fails.

**Loading the System:** The System definition is lazily loaded, so nothing happens until the first attempt to send a Request. At that point the SystemClient will query beer-garden to get a system definition that matches the system_name and version_constraint. If no matching system can be found a BrewmasterFetchError will be raised. If always_update was set to True this will happen before making each request, not just the first.

**Making a Request:** The standard way to create and send requests is by calling object attributes:

```
request = client.example_command(param_1='example_param')
```

In the normal case this will block until the request completes. Request completion is determined by periodically polling beer-garden to check the Request status. The time between polling requests starts at 0.5s and doubles each time the request has still not completed, up to max_delay. If a timeout was specified and the Request has not completed within that time a `BrewmasterTimeoutError` will be raised.

It is also possible to create the SystemClient in non-blocking mode by specifying blocking=False. In this case the request creation will immediately return a Future and will spawn a separate thread to poll for Request completion. The max_concurrent parameter is used to control the maximum threads available for polling.

```
# Create a SystemClient with blocking=False
client = SystemClient(host, port, 'example_system', ssl_enabled=True,␣
↪blocking=False)

# Create and send 5 requests without waiting for request completion
futures = [client.example_command(param_1=number) for number in range(5)]

# Now wait on all requests to complete
concurrent.futures.wait(futures)
```

If the request creation process fails (e.g. the command failed validation) and version_constraint is 'latest' then the SystemClient will check to see if a different version is available, and if so it will attempt to make the request on that version. This is so users of the SystemClient that don't necessarily care about the target system version don't need to be restarted if the target system is updated.

**Tweaking beer-garden Request Parameters:** There are several parameters that control how beer-garden routes / processes a request. To denote these as intended for beer-garden itself (rather than a parameter to be passed to the Plugin) prepend a leading underscore to the argument name.

Sending to another instance:

```
request = client.example_command(_instance_name='instance_2', param_1=
↪'example_param')
```

Request with a comment:

```
request = client.example_command(_comment='I'm a beer-garden comment!',
                                 param_1='example_param')
```

Without the leading underscore the arguments would be treated the same as param_1 - another parameter to be passed to the plugin.

**Parameters**

- **host** – beer-garden REST API hostname.
- **port** – beer-garden REST API port.
- **system_name** – The name of the system to use.
- **version_constraint** – The system version to use. Can be specific or 'latest'.
- **default_instance** – The instance to use if not specified when creating a request.
- **always_update** – Should check for a newer System version before each request.
- **timeout** – Length of time to wait for a request to complete. 'None' means wait forever.
- **max_delay** – Maximum time to wait between checking the status of a created request.

- **api_version** – beer-garden API version.
- **ssl_enabled** – Flag indicating whether to use HTTPS when communicating with beer-garden.
- **ca_cert** – beer-garden REST API server CA certificate.
- **blocking** – Block after request creation until the request completes.
- **max_concurrent** – Maximum number of concurrent requests allowed.
- **client_cert** – The client certificate to use when making requests.
- **url_prefix** – beer-garden REST API URL Prefix.
- **ca_verify** – Flag indicating whether to verify server certificate when making a request.

**create_bg_request**(*command_name*, ***kwargs*)

Create a callable that will execute a beer-garden request when called.

Normally you interact with the SystemClient by accessing attributes, but there could be certain cases where you want to create a request without sending it.

Example:

```
client = SystemClient(host, port, 'system', blocking=False)
requests = []

# No arguments
requests.append(client.create_bg_request('command_1'))

# arg_1 will be passed as a parameter
requests.append(client.create_bg_request('command_2', arg_1='Hi!'))

futures = [request() for request in requests]    # Calling creates and sends␣
↪the request
concurrent.futures.wait(futures)                 # Wait for all the futures to␣
↪complete
```

**Parameters**

- **command_name** – The name of the command that will be sent.
- **kwargs** – Additional arguments to pass to send_bg_request.

**Raises** **AttributeError** – The system does not have a command with the given command_name.

**Returns** A partial that will create and execute a beer-garden request when called.

**load_bg_system**()

Query beer-garden for a System definition

This method will make the query to beer-garden for a System matching the name and version constraints specified during SystemClient instance creation.

If this method completes successfully the SystemClient will be ready to create and send Requests.

**Raises** ***BrewmasterFetchError*** – If unable to find a matching System

**Returns** None

**send_bg_request**(***kwargs*)
Actually create a Request and send it to beer-garden

**Note:** This method is intended for advanced use only, mainly cases where you're using the SystemClient without a predefined System. It assumes that everything needed to construct the request is being passed in kwargs. If this doesn't sound like what you want you should check out create_bg_request.

**Parameters** **kwargs** – All necessary request parameters, including beer-garden internal parameters

**Raises** ***BrewmasterValidationError*** – If the Request creation failed validation on the server

**Returns** If the SystemClient was created with blocking=True a completed request object, otherwise a Future that will return the Request when it completes.

#### Module contents

brewtils.rest.**normalize_url_prefix**(*url_prefix*)

### 5.1.2 Submodules

### 5.1.3 brewtils.choices module

**class** brewtils.choices.**FunctionTransformer**
Bases: lark.tree.Transformer

**static arg_pair**(*s*)

**static func**(*s*)

**func_args**
alias of __builtin__.list

**static reference**(*s*)

**static url**(*s*)

**url_args**
alias of __builtin__.list

brewtils.choices.**parse**(*input_string*, *parse_as=None*)
Attempt to parse a string into a choices dictionary.

**Parameters**

- **input_string** – The string to parse
- **parse_as** – String specifying how to parse *input_string*. Valid values are 'func' or 'url'. Will try all valid values if None.

**Returns** A dictionary containing the results of the parse

**Raises** **lark.common.ParseError** – The parser was not able to find a valid parsing of *input_string*

### 5.1.4 brewtils.decorators module

brewtils.decorators.**system**(*cls*)
Class decorator that marks a class as a beer-garden System

Creates a `_commands` property on the class that holds all registered commands.

**Parameters** **cls** – The class to decorated

**Returns** The decorated class

brewtils.decorators.**parameter**(*_wrapped=None*, *key=None*, *type=None*, *multi=None*, *display_name=None*, *optional=None*, *default=None*, *description=None*, *choices=None*, *nullable=None*, *maximum=None*, *minimum=None*, *regex=None*, *is_kwarg=None*, *model=None*, *form_input_type=None*)
Decorator that enables Parameter specifications for a beer-garden Command

This decorator is intended to be used when more specification is desired for a Parameter.

For example:

```
@parameter(key="message", description="Message to echo", optional=True, type=
↪"String",
           default="Hello, World!")
def echo(self, message):
    return message
```

**Parameters**

- **_wrapped** – The function to decorate. This is handled as a positional argument and shouldn't be explicitly set.
- **key** – String specifying the parameter identifier. Must match an argument name of the decorated function.
- **type** – String indicating the type to use for this parameter.
- **multi** – Boolean indicating if this parameter is a multi. See documentation for discussion of what this means.
- **display_name** – String that will be displayed as a label in the user interface.
- **optional** – Boolean indicating if this parameter must be specified.
- **default** – The value this parameter will be assigned if not overridden when creating a request.
- **description** – An additional string that will be displayed in the user interface.
- **choices** – List or dictionary specifying allowed values. See documentation for more information.
- **nullable** – Boolean indicating if this parameter is allowed to be null.
- **maximum** – Integer indicating the maximum value of the parameter.
- **minimum** – Integer indicating the minimum value of the parameter.
- **regex** – String describing a regular expression constraint on the parameter.
- **is_kwarg** – Boolean indicating if this parameter is meant to be part of the decorated function's kwargs.

- **model** – Class to be used as a model for this parameter. Must be a Python type object, not an instance.
- **form_input_type** – Only used for string fields. Changes the form input field (e.g. textarea)

**Returns** The decorated function.

brewtils.decorators.**command**(*_wrapped=None*, *command_type='ACTION'*, *output_type='STRING'*, *schema=None*, *form=None*, *template=None*, *icon_name=None*, *description=None*)

Decorator that marks a function as a beer-garden command

For example:

```
@command(output_type='JSON')
def echo_json(self, message):
    return message
```

**Parameters**

- **_wrapped** – The function to decorate. This is handled as a positional argument and shouldn't be explicitly set.
- **command_type** – The command type. Valid options are Command.COMMAND_TYPES.
- **output_type** – The output type. Valid options are Command.OUTPUT_TYPES.
- **schema** – A custom schema definition.
- **form** – A custom form definition.
- **template** – A custom template definition.
- **icon_name** – The icon name. Should be either a FontAwesome or a Glyphicon name.
- **description** – The command description. Will override the function's docstring.

**Returns** The decorated function.

brewtils.decorators.**command_registrar**(*cls*)

Class decorator that marks a class as a beer-garden System

Creates a _commands property on the class that holds all registered commands.

**Parameters** **cls** – The class to decorated

**Returns** The decorated class

brewtils.decorators.**plugin_param**(*_wrapped=None*, *key=None*, *type=None*, *multi=None*, *display_name=None*, *optional=None*, *default=None*, *description=None*, *choices=None*, *nullable=None*, *maximum=None*, *minimum=None*, *regex=None*, *is_kwarg=None*, *model=None*, *form_input_type=None*)

Decorator that enables Parameter specifications for a beer-garden Command

This decorator is intended to be used when more specification is desired for a Parameter.

For example:

```
@parameter(key="message", description="Message to echo", optional=True, type=
↪"String",
           default="Hello, World!")
def echo(self, message):
    return message
```

**Parameters**

- **_wrapped** – The function to decorate. This is handled as a positional argument and shouldn't be explicitly set.
- **key** – String specifying the parameter identifier. Must match an argument name of the decorated function.
- **type** – String indicating the type to use for this parameter.
- **multi** – Boolean indicating if this parameter is a multi. See documentation for discussion of what this means.
- **display_name** – String that will be displayed as a label in the user interface.
- **optional** – Boolean indicating if this parameter must be specified.
- **default** – The value this parameter will be assigned if not overridden when creating a request.
- **description** – An additional string that will be displayed in the user interface.
- **choices** – List or dictionary specifying allowed values. See documentation for more information.
- **nullable** – Boolean indicating if this parameter is allowed to be null.
- **maximum** – Integer indicating the maximum value of the parameter.
- **minimum** – Integer indicating the minimum value of the parameter.
- **regex** – String describing a regular expression constraint on the parameter.
- **is_kwarg** – Boolean indicating if this parameter is meant to be part of the decorated function's kwargs.
- **model** – Class to be used as a model for this parameter. Must be a Python type object, not an instance.
- **form_input_type** – Only used for string fields. Changes the form input field (e.g. textarea)

**Returns** The decorated function.

brewtils.decorators.**register**(*_wrapped=None*, *command_type='ACTION'*, *output_type='STRING'*, *schema=None*, *form=None*, *template=None*, *icon_name=None*, *description=None*)

Decorator that marks a function as a beer-garden command

For example:

```
@command(output_type='JSON')
def echo_json(self, message):
    return message
```

**Parameters**

- **_wrapped** – The function to decorate. This is handled as a positional argument and shouldn't be explicitly set.
- **command_type** – The command type. Valid options are Command.COMMAND_TYPES.
- **output_type** – The output type. Valid options are Command.OUTPUT_TYPES.
- **schema** – A custom schema definition.
- **form** – A custom form definition.
- **template** – A custom template definition.
- **icon_name** – The icon name. Should be either a FontAwesome or a Glyphicon name.
- **description** – The command description. Will override the function's docstring.

**Returns** The decorated function.

### 5.1.5 brewtils.errors module

Module containing all of the BREWMASTER error definitions

**exception** brewtils.errors.**AckAndContinueException**
Bases: exceptions.Exception

**exception** brewtils.errors.**AckAndDieException**
Bases: exceptions.Exception

**exception** brewtils.errors.**BGConflictError**
Bases: *brewtils.errors.BrewmasterRestError*

Error indicating a 409 was raised on the server

**exception** brewtils.errors.**BGNotFoundError**
Bases: *brewtils.errors.BrewmasterRestError*

Error Indicating a 404 was raised on the server

**exception** brewtils.errors.**BrewmasterConnectionError**
Bases: *brewtils.errors.BrewmasterRestError*

Error indicating a connection error while performing a request

**exception** brewtils.errors.**BrewmasterDeleteError**
Bases: *brewtils.errors.BrewmasterRestError*

Error Indicating a server Error occurred performing a DELETE

**exception** brewtils.errors.**BrewmasterFetchError**
Bases: *brewtils.errors.BrewmasterRestError*

Error Indicating a server Error occurred performing a GET

**exception** brewtils.errors.**BrewmasterModelError**
Bases: exceptions.Exception

Wrapper Error for All BrewmasterModelErrors

**exception** brewtils.errors.**BrewmasterModelValidationError**
Bases: *brewtils.errors.BrewmasterModelError*

Error to indicate an invalid Brewmaster Model

**exception** brewtils.errors.**BrewmasterRestError**
Bases: exceptions.Exception

Wrapper Error to Wrap more specific BREWMASTER Rest Errors

**exception** brewtils.errors.**BrewmasterSaveError**
Bases: *brewtils.errors.BrewmasterRestError*

Error Indicating a server Error occurred performing a POST/PUT

**exception** brewtils.errors.**BrewmasterTimeoutError**
Bases: *brewtils.errors.BrewmasterRestError*

Error Indicating a Timeout was reached while performing a request

**exception** brewtils.errors.**BrewmasterValidationError**
Bases: *brewtils.errors.BrewmasterRestError*

Error Indicating a client (400) Error occurred performing a POST/PUT

**exception** brewtils.errors.**DiscardMessageException**
Bases: exceptions.Exception

Raising an instance will result in a message not being requeued

**exception** brewtils.errors.**NoAckAndDieException**
Bases: exceptions.Exception

**exception** brewtils.errors.**PluginError**
Bases: exceptions.Exception

Generic error class

**exception** brewtils.errors.**PluginParamError**
Bases: *brewtils.errors.PluginError*

Error used when plugins have illegal parameters

**exception** brewtils.errors.**PluginValidationError**
Bases: *brewtils.errors.PluginError*

Plugin could not be validated successfully

**exception** brewtils.errors.**RepublishRequestException**(*request*, *headers*)
Bases: exceptions.Exception

Republish to the end of the message queue

**Parameters**

- **request** (*brewtils.models.Request*) – The Request to republish
- **headers** – A dictionary of headers to be used by *brewtils.request_consumer.RequestConsumer*

**exception** brewtils.errors.**RequestProcessingError**
Bases: *brewtils.errors.AckAndContinueException*

**exception** brewtils.errors.**RequestStatusTransitionError**
Bases: *brewtils.errors.BrewmasterModelValidationError*

Error to indicate an updated status was not a valid transition

### 5.1.6 brewtils.models module

**class** brewtils.models.**Choices**(*type=None*, *display=None*, *value=None*, *strict=None*, *details=None*)

Bases: object

**DISPLAYS = ('select', 'typeahead')**

**TYPES = ('static', 'url', 'command')**

**schema = 'ChoicesSchema'**

**class** brewtils.models.**Command**(*name*, *description=None*, *id=None*, *parameters=None*, *command_type=None*, *output_type=None*, *schema=None*, *form=None*, *template=None*, *icon_name=None*, *system=None*)

Bases: object

**COMMAND_TYPES = ('ACTION', 'INFO', 'EPHEMERAL')**

**OUTPUT_TYPES = ('STRING', 'JSON', 'XML', 'HTML')**

**get_parameter_by_key**(*key*)

Given a Key, it will return the parameter (or None) with that key

**Parameters** **key** –

**Return parameter**

**has_different_parameters**(*parameters*)

Given a set of parameters, determines if the parameters provided differ from the parameters already defined on this command.

**Parameters** **parameters** –

**Return boolean**

**parameter_keys**()

Convenience Method for returning all the keys of this command's parameters.

**Return list_of_parameters**

**schema = 'CommandSchema'**

**class** brewtils.models.**Event**(*name=None*, *payload=None*, *error=None*, *metadata=None*, *timestamp=None*)

Bases: object

**schema = 'EventSchema'**

**class** brewtils.models.**Events**

Bases: enum.Enum

**ALL_QUEUES_CLEARED = 15**

**BARTENDER_STARTED = 3**

**BARTENDER_STOPPED = 4**

**BREWVIEW_STARTED = 1**

**BREWVIEW_STOPPED = 2**

**INSTANCE_INITIALIZED = 8**

**INSTANCE_STARTED = 9**

**INSTANCE_STOPPED = 10**

**QUEUE_CLEARED = 14**

**REQUEST_COMPLETED = 7**

**REQUEST_CREATED = 5**

**REQUEST_STARTED = 6**

**SYSTEM_CREATED = 11**

**SYSTEM_REMOVED = 13**

**SYSTEM_UPDATED = 12**

**class** brewtils.models.**Instance**(*name=None*, *description=None*, *id=None*, *status=None*, *status_info=None*, *queue_type=None*, *queue_info=None*, *icon_name=None*, *metadata=None*)

Bases: object

**INSTANCE_STATUSES = set(['UNKNOWN', 'DEAD', 'PAUSED', 'RUNNING', 'STOPPED', 'INITIALIZ**

**schema = 'InstanceSchema'**

**class** brewtils.models.**LoggingConfig**(*level=None*, *handlers=None*, *formatters=None*, *loggers=None*)

Bases: object

**DEFAULT_FORMAT = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'**

**DEFAULT_HANDLER = {'class': 'logging.StreamHandler', 'formatter': 'default', 'stream**

**LEVELS = ('DEBUG', 'INFO', 'WARN', 'ERROR')**

**SUPPORTED_HANDLERS = ('stdout', 'file', 'logstash')**

**formatter_names**

**get_plugin_log_config**(***kwargs*)

Get a specific plugin logging configuration.

It is possible for different systems to have different logging configurations. This method will create the correct plugin logging configuration and return it. If a specific logger is not found for a system, then the current logging configuration will be returned.

**Parameters** **kwargs** – Identifying information for a system (i.e. system_name)

**Returns**

**handler_names**

**schema = 'LoggingConfigSchema'**

**class** brewtils.models.**Parameter**(*key*, *type=None*, *multi=None*, *display_name=None*, *optional=None*, *default=None*, *description=None*, *choices=None*, *parameters=None*, *nullable=None*, *maximum=None*, *minimum=None*, *regex=None*, *form_input_type=None*)

Bases: object

**FORM_INPUT_TYPES = ('textarea',)**

**TYPES = ('String', 'Integer', 'Float', 'Boolean', 'Any', 'Dictionary', 'Date', 'DateTi**

**is_different**(*other*)

**schema = 'ParameterSchema'**

**class** brewtils.models.**PatchOperation**(*operation=None*, *path=None*, *value=None*)
Bases: object

**schema = 'PatchSchema'**

**class** brewtils.models.**Queue**(*name=None*, *system=None*, *version=None*, *instance=None*, *system_id=None*, *display=None*, *size=None*)
Bases: object

**schema = 'QueueSchema'**

**class** brewtils.models.**Request**(*system=None*, *system_version=None*, *instance_name=None*, *command=None*, *id=None*, *parent=None*, *children=None*, *parameters=None*, *comment=None*, *output=None*, *output_type=None*, *status=None*, *command_type=None*, *created_at=None*, *error_class=None*, *metadata=None*, *updated_at=None*)
Bases: object

**COMMAND_TYPES = ('ACTION', 'INFO', 'EPHEMERAL')**

**COMPLETED_STATUSES = ('CANCELED', 'SUCCESS', 'ERROR')**

**OUTPUT_TYPES = ('STRING', 'JSON', 'XML', 'HTML')**

**STATUS_LIST = ('CREATED', 'RECEIVED', 'IN_PROGRESS', 'CANCELED', 'SUCCESS', 'ERROR')**

**is_ephemeral**

**schema = 'RequestSchema'**

**status**

**class** brewtils.models.**System**(*name=None*, *description=None*, *version=None*, *id=None*, *max_instances=None*, *instances=None*, *commands=None*, *icon_name=None*, *display_name=None*, *metadata=None*)
Bases: object

**get_command_by_name**(*command_name*)
Retrieve a particular command from the system

**Parameters** **command_name** – Name of the command to retrieve

**Returns** The command object. None if the given command name does not exist in this system.

**get_instance**(*name*)
Get an instance that currently exists in the system

**Parameters** **name** – The name of the instance to search

**Returns** The instance with the given name exists for this system, None otherwise

**has_different_commands**(*commands*)
Check if a set of commands is different than the current commands

**Parameters** **commands** – The set commands to compare against the current set

**Returns** True if the sets are different, False if the sets are the same

**has_instance**(*name*)
Determine if an instance currently exists in the system

**Parameters** **name** – The name of the instance to search

**Returns** True if an instance with the given name exists for this system, False otherwise.

**instance_names**

```
schema = 'SystemSchema'
```

### 5.1.7 brewtils.plugin module

**class** brewtils.plugin.**PluginBase**(*client*, *bg_host=None*, *bg_port=None*, *ssl_enabled=None*, *ca_cert=None*, *client_cert=None*, *system=None*, *name=None*, *description=None*, *version=None*, *icon_name=None*, *instance_name=None*, *logger=None*, *parser=None*, *multithreaded=None*, *metadata=None*, *max_concurrent=None*, *bg_url_prefix=None*, ***kwargs*)

Bases: `object`

A beer-garden Plugin.

This class represents a beer-garden Plugin - a continuously-running process that can receive and process Requests.

To work, a Plugin needs a Client instance - an instance of a class defining which Requests this plugin can accept and process. The easiest way to define a `Client` is by annotating a class with the `@system` decorator.

When creating a Plugin you can pass certain keyword arguments to let the Plugin know how to communicate with the beer-garden instance. These are:

- `bg_host`
- `bg_port`
- `ssl_enabled`
- `ca_cert`
- `client_cert`
- `bg_url_prefix`

A Plugin also needs some identifying data. You can either pass parameters to the Plugin or pass a fully defined System object (but not both). Note that some fields are optional:

```
PluginBase(name="Test", version="1.0.0", instance_name="default", description="A␣
↪Test")
```

or:

```
the_system = System(name="Test",
                    version="1.0.0",
                    instance_name="default,
                    description="A Test")
PluginBase(system=the_system)
```

If passing parameters directly note that these fields are required:

**name** Environment variable `BG_NAME` will be used if not specified

**version** Environment variable `BG_VERSION` will be used if not specified

**instance_name** Environment variable `BG_INSTANCE_NAME` will be used if not specified. 'default' will be used if not specified and loading from environment variable was unsuccessful

And these fields are optional:

- description (Will use docstring summary line from Client if not specified)
- icon_name

- metadata
- display_name

Plugins service requests using a `concurrent.futures.ThreadPoolExecutor`. The maximum number of threads available is controlled by the max_concurrent argument (the 'multithreaded' argument has been deprecated).

**Warning:** The default value for `max_concurrent` is 1. This means that a Plugin that invokes a Command on itself in the course of processing a Request will deadlock! If you intend to do this, please set `max_concurrent` to a value that makes sense and be aware that Requests are processed in separate thread contexts!

**Parameters**

- **client** – Instance of a class annotated with @system.
- **bg_host** (*str*) – Hostname of a beer-garden.
- **bg_port** (*int*) – Port beer-garden is listening on.
- **ssl_enabled** (*bool*) – Whether to use SSL for beer-garden communication.
- **ca_cert** – Certificate that issued the server certificate used by the beer-garden server.
- **client_cert** – Certificate used by the server making the connection to beer-garden.
- **system** – The system definition.
- **name** – The system name.
- **description** – The system description.
- **version** – The system version.
- **icon_name** – The system icon name.
- **instance_name** (*str*) – The name of the instance.
- **logger** (`logging.Logger`.) – A logger that will be used by the Plugin.
- **parser** (*brewtils.schema_parser.SchemaParser*.) – The parser to use when communicating with beer-garden.
- **multithreaded** (*bool*) – DEPRECATED Process requests in a separate thread.
- **worker_shutdown_timeout** (*int*) – Time to wait during shutdown to finish processing.
- **metadata** (*dict*) – Metadata specific to this plugin.
- **max_concurrent** (*int*) – Maximum number of requests to process concurrently.
- **bg_url_prefix** (*str*) – URL Prefix beer-garden is on.
- **display_name** (*str*) – The display name to use for the system.
- **max_attempts** (*int*) – Number of times to attempt updating the request before giving up (default -1 aka never).
- **max_timeout** (*int*) – Maximum amount of time to wait before retrying to update a request.
- **starting_timeout** (*int*) – Initial time to wait before the first retry.
- **max_instances** (*int*) – Maximum number of instances allowed for the system.

- **ca_verify** (*bool*) – Verify server certificate when making a request.

**process_admin_message**(*message*, *headers*)

**process_message**(*target*, *request*, *headers*)
Process a message. Intended to be run on an Executor.

**Parameters**

- **target** – The object to invoke received commands on. (self or self.client)
- **request** – The parsed Request object
- **headers** – Dictionary of headers from the *brewtils.request_consumer.RequestConsumer*

**Returns** None

**process_request_message**(*message*, *headers*)
Processes a message from a RequestConsumer

**Parameters**

- **message** – A valid string-representation of a *brewtils.models.Request*
- **headers** – A dictionary of headers from the *brewtils.request_consumer.RequestConsumer*

**Returns** A *concurrent.futures.Future*

**run**()

**class** brewtils.plugin.**RemotePlugin**(*client*, *bg_host=None*, *bg_port=None*, *ssl_enabled=None*, *ca_cert=None*, *client_cert=None*, *system=None*, *name=None*, *description=None*, *version=None*, *icon_name=None*, *instance_name=None*, *logger=None*, *parser=None*, *multithreaded=None*, *metadata=None*, *max_concurrent=None*, *bg_url_prefix=None*, ***kwargs*)
Bases: *brewtils.plugin.PluginBase*

### 5.1.8 brewtils.request_consumer module

**class** brewtils.request_consumer.**RequestConsumer**(*amqp_url*, *queue_name*, *on_message_callback*, *panic_event*, *logger=None*, *thread_name=None*, ***kwargs*)
Bases: threading.Thread

Consumer that will handle unexpected interactions with RabbitMQ.

If RabbitMQ closes the connection, it will reopen it. You should look at the output, as there are limited reasons why the connection may be closed, which usually are tied to permission related issues or socket timeouts.

If the channel is closed, it will indicate a problem with one of the commands that were issued and that should surface in the output as well.

**Parameters**

- **amqp_url** (*str*) – The AMQP url to connection with
- **queue_name** (*str*) – The name of the queue to connect to
- **on_message_callback** (func) – The function called to invoke message processing. Must return a Future.

- **panic_event** (*event*) – An event to be set in the event of a catastrophic failure
- **logger** (*logging.Logger*) – A configured logger
- **thread_name** (*str*) – The name to use for this thread
- **max_connect_retries** (*int*) – Number of connection retry attempts before failure. Default -1 (retry forever).
- **max_connect_backoff** (*int*) – Maximum amount of time to wait between connection retry attempts. Default 30.
- **max_concurrent** (*int*) – Maximum number of requests to process concurrently

**close_channel**()
Call to close the channel cleanly by issuing the Channel.Close RPC command.

**close_connection**()
This method closes the connection to RabbitMQ.

**on_cancelok**(*unused_frame*)
Invoked when the queueing service acknowledges cancellation.

This method is invoked by pika when RabbitMQ acknowledges the cancellation of a consumer. At this point we will close the channel. This will invoke the on_channel_closed method once the channel has been closed, which will in-turn close the connection.

**Parameters** **unused_frame** (*pika.frame.Method*) – The Basic.CancelOK frame

**on_channel_closed**(*channel*, *reply_code*, *reply_text*)
Invoekd when the queueing service unexpectedly closes the channle.

Invoked by pika when RabbitMQ unexpectedly closes the channel. Channels are usually closed if you attempt to do something that violates the protocol, such as re-declare an exchange or queue with different parameters. In this case, we'll close the connection to shutdown the object.

**Parameters**

- **channel** (*pika.channel.Channel*) – The closed channel
- **reply_code** (*int*) – The numeric reason the channel was closed
- **reply_text** (*str*) – The text reason the channel was closed

**on_channel_open**(*channel*)
This method is invoked by pika when the channel has been opened. The channel object is passed in so we can make use of it.

The exchange / queue binding should have already been set up so just start consuming.

**Parameters** **channel** (*pika.channel.Channel*) – The channel object

**on_connection_closed**(*connection*, *reply_code*, *reply_text*)
Invoked when the connection is closed.

This method is invoked by pika when the connection to RabbitMQ is closed unexpectedly. Since it is unexpected, we will reconnect to RabbitMQ if it disconnects.

**Parameters**

- **connection** (*pika.connection.Connection*) – the closed connection object
- **reply_code** (*int*) – The server provided reply_code if given
- **reply_text** (*basestring*) – The server provided reply_text if given

**on_connection_open**(*unused_connection*)
Invoked when the connection has been established.

This method is called by pika once the connection to RabbitMQ has been established. It passes the handle to the connection object in case we need it, but in this case, we'll just mark it unused.

**on_consumer_cancelled**(*method_frame*)
Invoked by pika when RabbitMQ sends a Basic.Cancel for a consumer receiving messages.

**Parameters** **method_frame** (*pika.frame.Method*) – The Basic.Cancel frame

**on_message**(*channel*, *basic_deliver*, *properties*, *body*)
Invoked when a message is delivered from the queueing service.

Invoked by pika when a message is delivered from RabbitMQ. The channel is passed for your convenience. The basic_deliver object that is passed in carries the exchange, routing key, delivery tag and a redelivered flag for the message. the properties passed in is an instance of BasicProperties with the message properties and the body is the message that was sent.

**Parameters**

- **channel** (*pika.channel.Channel*) – The channel object
- **basic_deliver** (*pika.Spec.Basic.Deliver*) – basic_deliver method
- **properties** (*pika.Spec.BasicProperties*) – properties
- **body** (*str|unicode*) – The message body

**on_message_callback_complete**(*basic_deliver*, *future*)
Invoked when the future returned by _on_message_callback completes.

**Parameters**

- **basic_deliver** (*pika.Spec.Basic.Deliver*) – basic_deliver method
- **future** (*concurrent.futures.Future*) – Completed future

**Returns** None

**open_channel**()
Opens a channel on the queueing service.

Open a new channel with RabbitMQ by issuing the Channel.Open RPC command. When RabbitMQ responds that the channel is open, the on_channel_open callback will be invoked by pika.

**open_connection**()
Opens a connection to the queueing service.

This method connects to RabbitMQ, returning the connection handle. When the connection is established, the on_connection_open method will be invoked by pika.

**Return type** pika.SelectConnection

**reconnect**()
Will be invoked by the IOLoop timer if the connection is closed.

**run**()
Run the example consumer.

Creates a connection to the queueing service, then starts the IOLoop. The IOLoop will block and allow the SelectConnection to operate.

**Returns**

**start_consuming**()
Begin consuming messages from the queueing service.

This method sets up the consumer by first calling add_on_cancel_callback so that the object is notified if RabbitMQ cancels the consumer. It then issues the Basic.Consume RPC command which returns the consumer tag that is used to uniquely identify the consumer with RabbitMQ. We keep the value to use it when we want to cancel consuming. The on_message method is passed in as a callback pika will invoke when a message is fully received.

**stop**()
Cleanly shutdown the connection.

Assumes the stop_consuming method has already been called. When the queueing service acknowledges the closure, the connection is closed which will end the RequestConsumer.

**Returns**

**stop_consuming**()
Stop consuming by sending the Basic.Cancel RPC command.

### 5.1.9 brewtils.schema_parser module

**class** brewtils.schema_parser.**BrewmasterSchemaParser**
Bases: *brewtils.schema_parser.SchemaParser*

**class** brewtils.schema_parser.**SchemaParser**
Bases: object

Serialize and deserialize Brewtils models

**logger = <logging.Logger object>**

**classmethod parse_command**(*command*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a command model object

**Parameters**

- **command** – The raw input
- **from_string** – True if 'command' is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** A Command object

**classmethod parse_event**(*event*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to an event model object

**Parameters**

- **event** – The raw input
- **from_string** – True if 'event' is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** An Event object

**classmethod parse_instance**(*instance*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to an instance model object

**Parameters**

- **instance** – The raw input

- **from_string** – True if 'instance' is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** An Instance object

**classmethod parse_logging_config**(*logging_config*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a logging config model object

Note: for our logging_config, many is _always_ set to False. We will always return a dict from this method.

**Parameters**

- **logging_config** – The raw input
- **from_string** – True if 'logging_config' is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** A LoggingConfig object

**classmethod parse_parameter**(*parameter*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a parameter model object

**Parameters**

- **parameter** – The raw input
- **from_string** – True if 'parameter' is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** An Parameter object

**classmethod parse_patch**(*patch*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a patch model object

Note: for our patches, many is _always_ set to True. We will always return a list from this method.

**Parameters**

- **patch** – The raw input
- **from_string** – True if 'patch' is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** A PatchOperation object

**classmethod parse_queue**(*queue*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a queue model object

**Parameters**

- **queue** – The raw input
- **from_string** – True if 'event' is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** A Queue object

**classmethod parse_request**(*request*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a request model object

**Parameters**

- **request** – The raw input
- **from_string** – True if 'request' is a JSON string, False if a dictionary

- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** A Request object

**classmethod parse_system**(*system*, *from_string=False*, ***kwargs*)
Convert raw JSON string or dictionary to a system model object

**Parameters**

- **system** – The raw input
- **from_string** – True if 'system' is a JSON string, False if a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** A System object

**classmethod serialize_command**(*command*, *to_string=True*, ***kwargs*)
Convert a command model into serialized form

**Parameters**

- **command** – The command object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** Serialized representation of command

**classmethod serialize_event**(*event*, *to_string=True*, ***kwargs*)
Convert a logging config model into serialized form

**Parameters**

- **event** – The event object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** Serialized representation of event

**classmethod serialize_instance**(*instance*, *to_string=True*, ***kwargs*)
Convert an instance model into serialized form

**Parameters**

- **instance** – The instance object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** Serialized representation of instance

**classmethod serialize_logging_config**(*logging_config*, *to_string=True*, ***kwargs*)
Convert a logging config model into serialize form

**Parameters**

- **logging_config** – The logging config object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** Serialized representation of logging config

**classmethod serialize_parameter**(*parameter*, *to_string=True*, ***kwargs*)
Convert a parameter model into serialized form

**Parameters**

- **parameter** – The parameter object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** Serialized representation of parameter

**classmethod serialize_patch**(*patch*, *to_string=True*, ***kwargs*)
Convert a patch model into serialized form

**Parameters**

- **patch** – The patch object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** Serialized representation of patch

**classmethod serialize_queue**(*queue*, *to_string=True*, ***kwargs*)
Convert a queue model into serialized form

**Parameters**

- **queue** – The queue object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** Serialized representation of queue

**classmethod serialize_request**(*request*, *to_string=True*, ***kwargs*)
Convert a request model into serialized form

**Parameters**

- **request** – The request object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** Serialized representation of request

**classmethod serialize_system**(*system*, *to_string=True*, *include_commands=True*, ***kwargs*)
Convert a system model into serialized form

**Parameters**

- **system** – The system object(s) to be serialized
- **to_string** – True to generate a JSON-formatted string, False to generate a dictionary
- **include_commands** – True if the system's command list should be included
- **kwargs** – Additional parameters to be passed to the Schema (e.g. many=True)

**Returns** Serialized representation of system

### 5.1.10 brewtils.schemas module

**class** brewtils.schemas.**BaseSchema**(*strict=True*, ***kwargs*)
Bases: marshmallow.schema.Schema

**classmethod get_attribute_names**()

**make_object**(*data*)

**opts = <marshmallow.schema.SchemaOpts object>**

**class** brewtils.schemas.**ChoicesSchema**(*strict=True*, ***kwargs*)
Bases: *brewtils.schemas.BaseSchema*

**opts = <marshmallow.schema.SchemaOpts object>**

**class** brewtils.schemas.**CommandSchema**(*strict=True*, ***kwargs*)
Bases: *brewtils.schemas.BaseSchema*

**opts = <marshmallow.schema.SchemaOpts object>**

**class** brewtils.schemas.**DateTime**(*format='epoch'*, ***kwargs*)
Bases: marshmallow.fields.DateTime

Class that adds methods for (de)serializing DateTime fields as an epoch

**static from_epoch**(*epoch*)

**static to_epoch**(*dt*, *localtime=False*)

**class** brewtils.schemas.**EventSchema**(*strict=True*, ***kwargs*)
Bases: *brewtils.schemas.BaseSchema*

**opts = <marshmallow.schema.SchemaOpts object>**

**class** brewtils.schemas.**InstanceSchema**(*strict=True*, ***kwargs*)
Bases: *brewtils.schemas.BaseSchema*

**opts = <marshmallow.schema.SchemaOpts object>**

**class** brewtils.schemas.**LoggingConfigSchema**(*strict=True*, ***kwargs*)
Bases: *brewtils.schemas.BaseSchema*

**opts = <marshmallow.schema.SchemaOpts object>**

**class** brewtils.schemas.**ParameterSchema**(*strict=True*, ***kwargs*)
Bases: *brewtils.schemas.BaseSchema*

**opts = <marshmallow.schema.SchemaOpts object>**

**class** brewtils.schemas.**PatchSchema**(*strict=True*, ***kwargs*)
Bases: *brewtils.schemas.BaseSchema*

**opts = <marshmallow.schema.SchemaOpts object>**

**unwrap_envelope**(*data*, *many*)

**wrap_envelope**(*data*, *many*)

**class** brewtils.schemas.**QueueSchema**(*strict=True*, ***kwargs*)
Bases: *brewtils.schemas.BaseSchema*

**opts = <marshmallow.schema.SchemaOpts object>**

**class** brewtils.schemas.**RequestSchema**(*strict=True*, ***kwargs*)
Bases: *brewtils.schemas.BaseSchema*

**opts = <marshmallow.schema.SchemaOpts object>**

**class** brewtils.schemas.**StatusInfoSchema**(*strict=True*, ***kwargs*)
Bases: *brewtils.schemas.BaseSchema*

**opts = <marshmallow.schema.SchemaOpts object>**

**class** brewtils.schemas.**SystemSchema**(*strict=True*, ***kwargs*)
Bases: *brewtils.schemas.BaseSchema*

**opts = <marshmallow.schema.SchemaOpts object>**

### 5.1.11 brewtils.stoppable_thread module

**class** brewtils.stoppable_thread.**StoppableThread**(***kwargs*)
Bases: threading.Thread

Thread class with a stop() method. The thread itself has to check regularly for the stopped() condition.

**stop**()
Sets the stop event

**stopped**()
Determines if stop has been called yet.

**wait**(*timeout=None*)
Delegate wait call to threading.Event

### 5.1.12 Module contents

brewtils.**get_bg_connection_parameters**(***kwargs*)
Parse the keyword arguments, search in the arguments, and environment for the values

**Parameters** **kwargs** –

**Returns**

brewtils.**get_bool_from_kwargs_and_env**(*key*, *env_name*, ***kwargs*)
Gets a boolean value defaults to True

brewtils.**get_easy_client**(***kwargs*)
Initialize an EasyClient using Environment variables as default values

**Parameters** **kwargs** – Options for configuring the EasyClient

**Returns** An EasyClient

brewtils.**get_from_kwargs_or_env**(*key*, *env_names*, *default*, ***kwargs*)
Get a value from the kwargs provided or environment

**Parameters**

- **key** – Key to search in the keyword args
- **env_names** – Environment names to search
- **default** – The default if it is not found elsewhere
- **kwargs** – Keyword Arguments

**Returns**

CHAPTER 6

# Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

## 6.1 Types of Contributions

### 6.1.1 Report Bugs

Report bugs at https://github.com/beer-garden/brewtils/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

### 6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

### 6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

### 6.1.4 Write Documentation

Brewtils could always use more documentation, whether as part of the official Brewtils docs, in docstrings, or even on the web in blog posts, articles, and such.

### 6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/beer-garden/brewtils/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

## 6.2 Get Started!

Ready to contribute? Here's how to set up `brewtils` for local development.

1. Fork the `brewtils` repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/brewtils.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv brewtils
$ cd brewtils/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 brewtils test
$ nosetests
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

## 6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.5, and 3.6. Check https://travis-ci.org/beer-garden/brewtils/pull_requests and make sure that the tests pass for all supported Python versions.

## 6.4 Tips

To run a subset of tests:

```
$ nosetests test/models_test.py:SystemTest.test_instance_names
```

CHAPTER 7

# Credits

## 7.1 Development Leads

- Logan Asher Jones <loganasherjones@gmail.com>
- Matt Patrick

## 7.2 Contributors

None yet. Why not be the first?

CHAPTER 8

# Brewtils Changelog

## 8.1 2.3.2

Date: 3/7/18

### 8.1.1 Bug Fixes

- Fixed issue with multi-instance remote plugins failing to initialize

## 8.2 2.3.1

Date: 2/22/18

### 8.2.1 New Features

- Added `description` keyword argument to `@command` decorator

## 8.3 2.3.0

Date: 1/26/18

### 8.3.1 New Features

- Added methods for interacting with the Queue API to RestClient and EasyClient
- Clients and Plugins can now be configured to skip server certificate verification when making HTTPS requests

- Timestamps now have true millisecond precision on platforms that support it
- Added `form_input_type` to Parameter model
- Plugins can now be stopped correctly by calling their `_stop` method
- Added Event model

### 8.3.2 Bug Fixes

- Plugins now additionally look for `ca_cert` and `client_cert` in `BG_CA_CERT` and `BG_CLIENT_CERT`

### 8.3.3 Other Changes

- Better data integrity by only allowing certain Request status transitions

## 8.4 2.2.1

Date: 1/11/18

### 8.4.1 Bug Fixes

- Nested requests that reference a different beer-garden no longer fail

## 8.5 2.2.0

Date: 10/23/17

### 8.5.1 New Features

- Command descriptions can now be changed without updating the System version
- Standardized Remote Plugin logging configuration
- Added domain-specific language for dynamic choices configuration
- Added `metadata` field to Instance model

### 8.5.2 Bug Fixes

- Removed some default values from model `__init__` functions
- System descriptors (description, display name, icon name, metadata) now always updated during startup
- Requests with output type 'JSON' will now have JSON error messages

### 8.5.3 Other changes

- Added license file

## 8.6 2.1.1

Date: 8/25/17

### 8.6.1 New Features

- Added `updated_at` field to `Request` model
- `SystemClient` now allows specifying a `client_cert`
- `RestClient` now reuses the same session for subsequent connections
- `SystemClient` can now make non-blocking requests
- `RestClient` and `EasyClient` now support PATCHing a `System`

### 8.6.2 Deprecations / Removals

- `multithreaded` argument to `PluginBase` has been superseded by `max_concurrent`
- These decorators are now deprecated - `@command_registrar`, instead use `@system` - `@plugin_param`, instead use `@parameter` - `@register`, instead use `@command`
- These classes are now deprecated - `BrewmasterSchemaParser`, instead use `SchemaParser` - `BrewmasterRestClient`, instead use `RestClient` - `BrewmasterEasyClient`, instead use `EasyClient` - `BrewmasterSystemClient`, instead use `SystemClient`

### 8.6.3 Bug Fixes

- Reworked message processing to remove the possibility of a failed request being stuck in `IN_PROGRESS`
- Correctly handle custom form definitions with a top-level array
- Smarter reconnect logic when the RabbitMQ connection fails

### 8.6.4 Other changes

- Removed dependency on `pyopenssl` so there's need to compile any Python extensions
- Request processing now occurs inside of a `ThreadPoolExecutor` thread
- Better serialization handling for epoch fields

# Python Module Index

## b

# Index

## Q



## R



## S



## T

## U



## W