Battlesnake Documentation

Release 0.1

Greg Taylor

Sep 27, 2017

Contents

Battlesnake is a Softcode supplement/replacement bot for BattletechMUX.

The typical BattletechMUX requires numerous complex systems to function. For example:

- Mech ref libararies
- Economic simulations
- Stores for purchasing/selling commodities and parts
- Player stat tracking
- Bulletin board systems

While these have all been successfully built and maintained in softcode, maintenance and future expansion can be very slow. Softcode is a poor choice for larger systems.

There are also things that can't be done in-game without the help of hardcode modifications or logfile workarounds:

- Sending emails
- Communication with arbitrary databases/data stores
- Web-based character creation
- Integration with messenger services
- Utilization of social media

Battlesnake aims to supplement or replace large chunks of softcode in your game, while also opening up any external services you'd like to use.

CHAPTER 1

Learning more

Project Status: Early development

License: Battlesnake is licensed under the BSD License.

- Source repository: https://github.com/gtaylor/btmux_battlesnake
- Issue tracker: https://github.com/gtaylor/btmux_battlesnake/issues
- Live support is available on the *BTMux* channel on the Frontier.

CHAPTER 2

Documentation

How it works

Battlesnake is powered by Python and Twisted. The bot connects to your game over telnet, just like a user would. It sets some semi-random tokens on its player object that allows your softcode to communicate with it via `@pemit`.

Your softcoded commands end up being mostly for gathering up the relevant data to `@pemit` to Battlesnake. If the commands need to send a reply back out to a player, the bot does so with `@pemit` as well.

What is Battlesnake ideally suited for?

Battlesnake is best used for larger, more complicated systems that would be easier maintained in Python than softcode. Battlesnake may be paired with your choice of database or data store, with softcode commands used to feed game state data to Battlesnake. The bot can also pull things from the game on its own if you show it how to.

Example usage cases

A few ideas for neat things Battlesnake can be used for:

- Web-based character creation
- Adding an HTTP API for your website's use
- Tweeting/SMS'ing/emailing certain events to your playerbase
- Very detailed player stat tracking (perhaps shown on your website)
- Stuffing arbitrary bits of data into a full-fledged database
- External AI
- Web-integrated bulletin board systems
- Economic simulations
- Research/industrial/assembly systems

Installation

Warning: Battlesnake currently requires a game database with a very specific set of objects/functions/parents. If you don't have a DB from Kelvin, you'll want to find him on the Frontier's BTMux channel and ask. We'll eventually have one up for download once this stabilizes.

Battlesnake is developed on GitHub in a git repository. We don't do any point releases at this time, as things are still evolving, and the expected audience for this software is pretty niche.

The first thing to do is retrieve a clone of the repository:

```
git clone https://github.com/gtaylor/btmux_battlesnake.git
```

This will leave you with a `btmux_battlensake` directory. `cd` into it:

```
cd btmux_battlesnake
```

Now install the requirements via `pip`, preferrably within a virtualenv (you are using virtualenv, right?):

```
pip install -r requirements.txt
```

If you want to be able to generate documentation locally or run the test suite, install the developer dependencies:

```
pip install -r requirements_dev.txt
```

Now copy the starter config file and adjust the `hostname`, `port`, `username`, and `password` values:

```
cp config/battlesnake.cfg.dist battlesnake.cfg
vim battlesnake.cfg (Or whatever your preferred editor is)
```

You are now ready to run the bot (from within `btmux_battlesnake`):

```
twistd -n battlesnake
```

This defaults to using the `battlesnake.cfg` file in your current directory, but you can run multiple bots or use an alternative location with the `-c` flag:

```
twistd -n battlesnake -c battlesnake.cfg
```

When in doubt, check out the help listing:

```
twistd -n battlesnake -h
```

Tip: You will not be able to run battlesnake unless your current directory is `btmux_battlesnake` (or whatever you have renamed it to). This is a limitation of Twisted's plugin system.

For more details on settings, check out *Settings*.

Getting started using the Bot

At this point, we assume that your bot is connected to your game. The first thing you'll want to do is `examine` your bot's player object. We'll also assume that its name is `Battlesnake`. You should see three attributes like these:

```
BATTLESNAKE_PREFIX.D: @G$>
BATTLESNAKE_KWARG_DELIMITER.D: &R^
BATTLESNAKE_LIST_DELIMITER.D: #E$
```

Tip: Note these values, as we'll be using them in the examples below.

You'll also want to note your bot's dbref. We'll use the dbref `#123` as a placeholder.

A crude botinfo example

Now choose an object to put a command on. This could be in your master room or in your current location. Here's the attribute we'll set:

```
BOTINFO.C: $botinfo:think [u(#47/SET_BOT_REGISTERS.F)][u(#47/SENDPACKET.F,botinfo)]
```

Breaking this down, `SET_BOT_REGISTERS.F` sets some %q registers that `SENDPACKET.F` uses to form a `pemit()` call to a connected bot. The `botinfo` command is being sent to the bot.

You'll probably want to set your bot `WIZARD` first, then try running your new `botinfo` command within your MUX. If everything is set up correctly, you should see a response with some info about the bot.

Moving on from here

The next step is to read over the *Battlesnake protocol* and start thinking big!

Battlesnake protocol

Battlesnake communicates over `@pemit`, using strings broken up with multi-character delimiters for various purposes. While the protocol is crude and makes some assumptions, it is reliable enough for heavy usage.

Inbound vs Outbound commands

Battlesnake has a notion of *inbound* and *outboud* commands. Outbound commands are those performed by the bot, sending text to the game. Inbound commands are `@pemit` strings asking the bot to do something, typically from your softcode.

This document will mostly focus on inbound commands, as that is where most of the challenge is.

A high level overview of inbound command syntax

An simple inbound command `@pemit` syntax example:

```
<prefix_str><command_name><kwarg_delim><invoker_dbref>
```

Breaking this down by component:

<prefix_str> A randomized multi-character string that is set on the bot's player object when it connects. If the bot sees this at the beginning of a line of input, it knows to look **command_name** up in its command table.

<command_name> This is the command name that Battlesnake will look up in its internal command table. For example, *send_email*.

<kwarg_delim> This is another randomly generated multi-character string that is used to separate bits of input to send to the bot. Almost all data (save for the invoker's dbref) is in key=value form, separated by this delimiter.

<invoker_dbref> This is the object on the MUX that is sending the command. The most common use for this is to give the bot a way to reply to the invoker.

Here's an example inbound command with no additional data:

```
PREFIXSTRsend_emailKWDELIM#212
```

<prefix_str> is PREFIXSTR, <command_name> is send_email, <kwarg_delim> is KWDELIM, and <invoker_dbref> is #212.

Sending key/value data with inbound commands

We know how to send inbound commands now, but this isn't too useful unless we can also send in arbitrary bits of data. Expanding on our previous example, here's how that works:

```
<prefix_str><command_name><kwarg_delim><invoker_dbref><kwarg_delim>key1=value1<kwarg_
↪delim>key2=value2
```

As you can see, **kwarg_delim** is used to split up key/value pairs. On the Battlesnake side, we convert these into Python dicts. Here's how the command parser would split this up:

```
{'key1': 'value1', 'key2': 'value2'}
```

But what if we omit a value for a key?

```
<kwarg_delim>key1=<kwarg_delim>
{'key1': ''}
```

Sending lists values

Sometimes we'll need to send lists instead of individual values:

```
<kwarg_delim>key=item1<list_delim>item2
```

We've introduced a new delimiter, **list_delim**. Much like **prefix_str** and **kwarg_delim**, this is a randomly generated multi-character delimiter. The presence of a list delimiter in a kwarg's value causes it to be converted to a list in Battlesnake. Let's say we do something like this (omitted invoker/command name/prefix for brevity):

```
<kwarg_delim>key=item1<list_delim>item2<list_delim>item3
```

Within Battlesnake, this would be interpreted as:

```
{'key': ['item1', 'item2', 'item3']}
```

You can combine regular string values and list values without issue:

```
<kwarg_delim>key1=value1<kwarg_delim>key2=item1<list_delim>item2<list_delim>item3
```

In Python land, this would be interpreted as:

```
{
    'key1': 'value1',
    'key2': ['item1', 'item2', 'item3']
}
```

Protocol limitations

The Battlesnake protocol only knows of two different data types: strings and lists. Your logic on the Python side will need to know how to treat the data being passed in. If you need ints, you'll need to cast them and potentially handle errors.

While the delimiter characters should be random enough to avoid collisions, if your softcode were to generate values that matched one of the delimiters, kwarg pairs could be discarded. The likelihood of this happening is incredibly low, though, unless your data is sufficiently random and large.

Triggers

Battlesnake's triggers work much like a traditional MU* client's in that a key phrase sets off an action. The primary use for triggers in Battlesnake is for data collection, though it can be used for a number of other purposes.

A `Trigger` consists of a regular expression that determines what to look for, and a function to run when a match is made. The neat thing about Python regular expressions is that we can use regex groups to break the matches up into useful pieces. For example, if a trigger has the following regex:

```
line_regex = re.compile(r'(?P<talker>.*) says "[Hh]ello"')
```

Our `run` method on the `Trigger` can address each group individually:

```
talkers_name = re_match.group("talker")
response = "Why hello there, {talkers_name}.".format(talkers_name=talkers_name)
mux_commands.say(protocol, response)
```

In the case of the previous example, anyone saying "Hello" to the bot would be greeted back:

```
You say "Hello"
Battlesnake says "Why hello there, Kelvin McCorvin."
```

Note how the original speaker's name comes back.

Common usage cases

We'll outline a few common usage cases for Triggers, for the sake of providing some ideas. These aren't the only uses, though.

Retrieving data

If you have a large amount of data to feed your bot on a regular basis, you can use triggers and *Timers* in conjunction with one another to do so. Perhaps you write a softcode command that your bot executes with a timer, picking the output up with a trigger. Or maybe your game has a task scheduling system interally which can be used to `@pemit` a string matching one of your triggers in order to feed data in.

Event monitoring

Triggers can be used to watch for certain events. Perhaps you join your bot to the **MUXConnections** channel and set up a trigger to note when players log in. Or maybe you park your bot in an Observation Lounge and use triggers to record shot stats or respond to base capture emits.

Timers

Battlesnake's timers work much like a traditional MU* client's in that they are used to perform an action every so often. There are two kinds of timers:

- `battlesnake.core.timers.IntervalTimer`
- `battlesnake.core.timers.CronTimer`

Timers of the `IntervalTimer` type are executed on intervals measured in seconds. For example, "Do something every 30 seconds".

Timers of the `CronTimer` type execute tasks at pre-set times during the day. For example, "Do something on the 30th minute of every hour of the day". Use these when you want to get very specific with execution times.

Common usage cases

We'll outline a few common usage cases for Timers, for the sake of providing some ideas. These aren't the only uses, though.

Retrieving data

Battlesnake has to either retrieve or be fed data from your game in order to stay in sync with what is going on. Timers are a great way to achieve that.

Some example ways to do this:

- Run the `battlesnake.outbound_commands.mux_commands.think()` command to retrieve specific values of interest.
- Run various softcode commands in your game whose output is picked up by *Triggers* within Battlesnake. This is good for when you have a very large volume of data to retrieve (which may be a lot more involved to pick up via `think`.

Economy/weather ticks

Economy and weather simulations are some of the more bulky, nasty bits of softcode in your typical BTMUX. These are excellent candidates to move into Battlesnake. Timers are important pieces of both Economy and weather-related things.

Stat collection

If you are tracking certain time-series data points (like # of users connected), timers are a great way to make sure you are getting points the correct interval.

Settings

When Battlesnake is started, the path to your config file is passed in via the `-c` flag. Default values are pulled from `battlesnake/config/configspec.cfg`. Any of the values detailed below may be overridden in your config file.

The name in brackets in each section below is the section name in the config file.

[mux]

`hostname` The hostname or IP address of your game. **Must be overridden in your config file**.

`port` The port to connect to. **Must be overridden in your config file**.

[account]

`username` (default: Battlesnake) The player username to connect as.

`password` The player's password. **Must be overridden in your config file**.

[bot]

`response_watcher_expire_check_interval` (default: 1.0) Sets the interval (seconds) for how often to check for stale response watchers to purge.

`enable_hudinfo` (default: False) If True, generate and send a HUDINFO key. This will allow you to start using HUDINFO commands.

`extra_services` (default: []) A list of Python paths to loader functions that return a Service. If you only have one item to add to the list, make sure there is a trailing comma or you'll get a validation error. A comma causes our config system to convert the string to a list.

`plugins` (default: 'battlesnake.plugins.example_plugin.plugin.ExamplePlugin','battlesnake.plugins.nat_idler.plugin.NatIdlerP
A comma-separated list of BattlesnakePlugin sub-classes to register. Plugins can contain *Triggers*, *Timers*, and commands.

Plugins

[nat_idler]

`keepalive_interval` (default: 30.0) Sets the interval (seconds) at which the bot sends an IDLE command to the MUX. This is useful to prevent timeouts over NATs.

[unit_spawning]

`unit_parent_dbref` (default: #66) Your mech/unit parent's dbref.

[ai]

`ai_parent_dbref` (default: #69) Your AI parent's dbref.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search