MockingBOT

User Documentation and Guides

October 1, 2013

Introduction

MockingBOT is a bot protocol developed for the MacGRID initiative, funded by McMaster University and organized by Dr. David Harris Smith. It is developed using the .NET Framework and C#, and comes in two packages that work independently from each other. There’s a ‘client’, a Windows Form Application that runs as a separate program on your computer. In addition, there’s a ‘region module’, a .dll file that can be integrated into any OpenSim installation and runs automatically upon loading the sim. The difference between those will be covered in the section ‘Client vs. Module’.

In this guide, I will cover how to make use of both methods, and the basic code base behind them in order to facilitate developing them further. Note that I designed MockingBOT to be used by any layperson who might be interested in the MacGRID initiative. While OSSLNPC exists for this purpose, I wanted something that was much more open, easy to use, and included a far greater degree of functionality.

The name is an amalgamation of ‘bot’ and ‘mockingbird’. The main purpose of MockingBOT is to store information and to iterate on it in new, non-linear ways that seem natural and original, much like mockingbirds do with song.

Module vs. Client

The client and module have very different ways of working, even though they are effectively the same protocol. That is, bots are effectively the same using either method.

The first, and most basic factor, is the requirements needed for each. While both systems are fairly compact and require very little to run, actually operating the bots requires a couple things for each.

The region module runs on its own. Once you place the .dll file into the OpenSim installation folder (opensim-x.x.x/bin), OpenSim will load the module automatically when it next boots up the sim. No extra steps are needed, though do note that the simulator needs to be rebooted in order to load the module. However, the main implication of this is that you need to own the simulator, or otherwise have access to its installation. If you wish to take a bot into another grid, you must use the client.

At present, the bots and all of their settings are loaded from an .xml file named ‘botsettings.xml’, which is to be included in the same folder as the module. In the future, once a MockingBOT Module is installed, anyone can create or edit bots within the sim (as of yet, without restriction). The .xml file may be edited manually, or using the client. This will be covered later. All that’s important is that this .xml file contains all the settings for every bot, and is used by both the client and the module.

Once running, bots using the module effectively operate like OSSLNPCs, with AI powered by the MockingBOT protocol, but with some large advantages besides that. OSSLNPCs are fairly limited in terms of functionality. MockingBOTs are capable of receiving and responding to IMs, offering to teleport other avatars, and accessing APIs outside of OpenSim – they are able to tweet, retrieve weather information, and so on. This is not something OSSLNPCs are designed to do, at this moment.

The client has all the same advantages. Bots do not, however, operate like NPCs. They are effectively avatars, able to do anything a human character could do. In order to use the client, you need only run the program – you do not need any special access to the grid. This approach was designed for the Second Life environment, which had no in-built support for NPCs, and did not give out access to server software to users. The main requirement is that the bot be logged in using a user account on whatever grid it wants to access. Using the client, you log it in using a username and password, like any viewer – though, unlike typical Second Life/OpenSim viewers, the MockingBOT client does not insist on using an outrageous amount of resources. In fact, 30k of memory are all it requires, as a high estimate. That said, it makes no attempt to render the scene or show you what’s happening in the grid. If you want to see how your bot is doing, you must log in using a viewer and check it out in-world. The client is designed to log in the bot, and then run silently.

Furthermore, while the region module will only work for OpenSim installations, the client can be used in Second Life or OpenSim with no extra work.

In terms of operability, the region module gives the bots greater functionality, because bots running from it are able to access virtually all information in the grid. It also runs multiple bots in one process, which means that they’re able to communicate and share information. While the implementation isn’t nearly as cool as it sounds, they *are* able to answer questions collaboratively, and if they don’t know an answer to a user’s question, they can refer the user to another bot that does. This isn’t necessarily a huge upgrade, but if you’re using multiple bots, consider the region module for that reason. The client only operates one bot at a time, though you can switch between any of the saved bots that have been programmed into the settings, and also program multiple bots using the one program. If you want to operate multiple bots using the client method, you just need to run multiple clients – which, considering one client uses 30k of memory, is no hardship.

Finally, while I will obviously claim MockingBOT to be stable/infallible/generally perfect in every conceivable way, one thing to consider is the off-chance of it failing. This isn’t a problem for the client: worst case scenario, it will simply crash, the bot will log out, and you can easily reload it without any further difficulty. Same as if your word processor crashes. This is because it runs separately and independently of anything else.

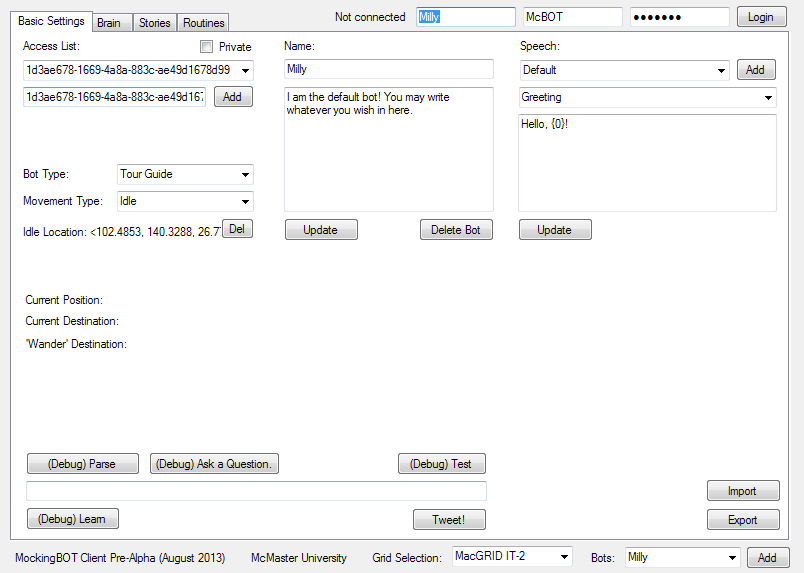
However, the region module is integrated with the simulator. In addition to requiring some extra programming to make it work in the same process as the sim, it acts as an additional point of failure for it. This is as much a problem for any module on OpenSim, so don’t think too much of it; it’s just something to note. Of course, if there are any problems running the module in this way, it’s advised you contact the developer immediately and share as much information as you can regarding the situation. MockingBOT has been tested rather heavily, but, there are always bugs in every piece of software (OpenSim is no exception – Second Life itself, doubly so).

To summarize:

|  |  |  |
| --- | --- | --- |
| **Client**  Requirements  A working user account in any grid  Advantages  Programmable on-the-fly  Runs independently from the simulator, and thus doesn’t act as a failure point.  Disadvantages  While you can program multiple bots from one client, each client operates one bot – you can easily run multiple clients, however.  Bots are unable to recognize each other as bots, unlike the module, and thus have no interaction with each other. But, this is not heavily utilized in the module, either. |  | **Module**  Requirements  Access to the OpenSim server installation  Rebooting the sim to load the module  Advantages  Greater support for multiple bots, and functionality to let them interact with other.  No external client required.  Disadvantages  Loads settings saved through the client – no programming on-the-fly (though you can still teach bots in-world using text chat and save/load the settings file manually)  Integrated into the simulator, and thus acts as an additional failure point in the event of a crash. |

Either way, it is best to learn the client, since this is the most effective way to program and create bots at the moment. The module is there to allow bots to run automatically without having to run an external program.

Programming – Basic Settings



**Access List** This is for controlling who can control your bot. The long strings of letters and numbers are actually UUIDs, which is unique for every avatar. To find it, go into the grid, and check the person’s profile.

If you don’t want to use this feature, you can simply not check the ‘Private’ box. Then, the bot will never check this list.

**Name** This is the name of the NPC created in OpenSim if using the region module. This is not the case when logging in with a user account – in that case, this can only be used for organizational purposes, in telling various bots apart in-client.

**Description** Purely for organizational purposes.

**Delete Bot** Self-explanatory – will delete this bot entirely. This is permanent.

**Bot Type** There are two major types that you need to worry about here. The first is **Tour Guide**. Tour guides will automatically greet people nearby and ask if they would like a tour.

**NPC**s, on the other hand, do not use this behaviour. They are general, all-purpose bots.

**HitchBOT** is a proof-of-concept type that will probably not be used for any functions.

**Movement Type** There are four types of movement available.

**Wander** allows the bot to move around aimlessly, without restriction.

**Idle** forces the bot to return to their idle location whenever they’re not engaged with someone else, which means following or giving tours.

**Wander and Idle** allows the bot to wander aimlessly, but only around their idle location. Good for being more natural, but anchored to the place where they’ll be most useful to users.

**Routine** lets the bot use a programmed routine. Good for scripting NPCs.

**Idle Location** This is set in-world using **!setidle**. Depending on the Movement Type, the bot will gravitate to this spot. You can use this to set up a welcome area, information desk, etc.

**Speech** This is for the generic dialogue the bot uses for several functions, all of which you can see in the dropdown menu. A full explanation exists in the section ‘Programming – Speech’.

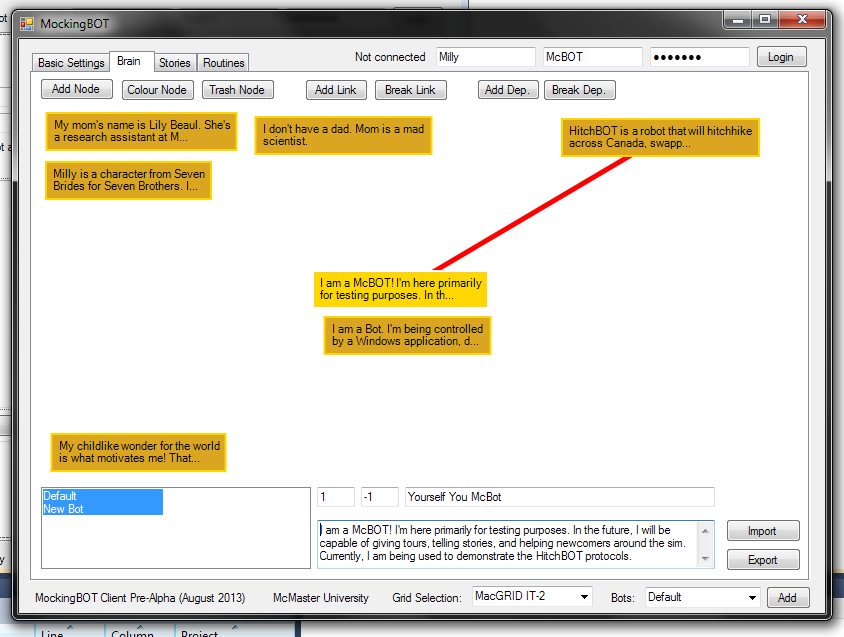
The following are available on every tab, not just Basic Settings:

**Import/Export** This allows you to select which .xml file to save and load. You are asked to load a file when you first load the client, but you can use this to manage multiple settings.

**Grid Selection** This lets you decide what grid to log the bot onto.

**Bots** This is a list of all the bots you have to work with, each with their own speech settings, allowed dialogue nodes, tours, etc. You can add as many as you like.

Programming – Dialogue Nodes



MockingBOT’s main purpose is to store and recite information for users. They are, in short, information kiosks. To this end, you can easily write pieces of information, or ‘nodes’, into its mind map. Each node contains keywords and a string of information for the bot to recite, ranging from asingle sentence to multiple paragraphs.

In the client, merely open the tab called ‘Brain’. Along the top of this tab, you’ll see a row a buttons: Add Node, Colour Node, Trash Node, Add Link, Break Link, Add Dependency, and Break Dependency. These are simple to use, so long as you know how MockingBOT reads the information in these nodes.

**Add Node** Does exactly what it sounds like. Click it, and then click anywhere in the mind map. You will get an empty node, ready to be filled in.

**Colour Node** Lets you change the colour of the selected node through a simple palette menu.

**Trash Node** Deletes the currently selected node. Will ask for confirmation.

**Add Link** Select one node, then the other, to link them together. When the MockingBOT reads one node, it will automatically read any that it’s linked to, provided the person is not currently aware of that piece of information.

**Break Link** Breaks off any a link with another node. Select one, then the other that it’s linked to.

**Add Dep.** Adds a dependency, which is effectively a one-way link. Works just like ‘Add Link’. It indicates that one node of information is dependent on another, in order to be understood fully by the user. If the user is not aware of that piece of information, the bot tells them about it right away. This can help them cover highly complex subject matter.

**Break Dep.** Break the dependency, just like ‘Break Link’ above.

You can scroll the mind map by holding the Right Mouse Button and moving the mouse. The limit on space is a hard value set by the computer (32,767 pixels wide and tall), and is more than you’ll ever need.

In the lower-left, all the bots saved in settings are listed. The ones that are permitted to use the selected node are highlighted. By clicking on names, you can highlight them, which indicates that they are now allowed to read the node (and vice versa). You use Ctrl and Shift to select multiple ones. This way, certain pieces of information can be reserved for specific bots. If you are using the region module version, then if a bot does not have access to information that the user is asking them about, they can refer them to one that does. This can be used to help differentiate characters – a pedestrian NPC can be ignorant of things that a physician NPC is not.

On the lower right are the textboxes where all the magic happens. The thin one on top is where you write keywords. These are what the bot searches for when looking for information. If a user asks a question like ‘what is love’, the bot will look for any node with the keywords ‘what’ and ‘love’. You list keywords with a space in-between. Do not worry about capitals or punctuation – the bot will simply throw those out.

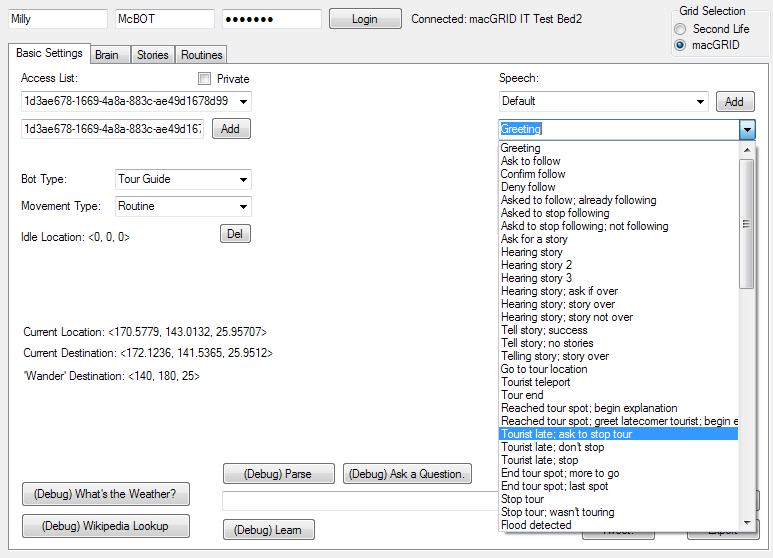
You can group keywords together, so that both need to be present in a question before the bot will acknowledge either. You do this by separating the keywords with a hyphen. For example, ‘are-you’.

Below that is the textbox where you write the actual information that the bot will read. This can be as much or as little as you want.

The main purpose of this is to organize information in ways that users can sort through a question-and-answer process. Think of how you’re going to structure a conversation someone might have with the bot. You can change the bot’s greetings (covered in the section ‘Programming - Speech’) to tell the user specifically what sort of things that can tell them about. You could likewise for the speech string ‘Doesn’t have response’, which is used for when the bot has no answer to a question. There could be a node specifically to answer the question ‘what can you tell me about’.

Links and dependencies are what makes MockingBOT more than just a glorified search engine. Consider this document, and how often I have to write ‘this will be covered in another section’. In a linear format, things need to be organized in this way. Unfortunately, I don’t know exactly what you need to know or how you learn, and I can’t structure this document to cater to that. MockingBOT isn’t that complicated, but obviously it’s difficult to teach something with a lot of information, all of which is interconnected, that can’t be put into such a simple and organized fashion. MockingBOT doesn’t just pull out specific pieces of information, but can be used to teach people based on what they’re actually looking to know and understand.

Programming – Speech



Dialogue nodes are one major part of how bots speak. The other is speech strings, which are available to be edited in the Basic Settings tab. These are the generic responses the bot gives for various situations, as shown in the list available to you – for example, the first one is for a basic greeting. There is another one for greeting someone that is already known.

One you select one, the textbox below will show the content of this generic response. The easy explanation is that you can type anything in this textbox, press ‘Update’, and this will become the new bit of text the bot will recite when this response is called. There are a few more nuances, however.

For example, sometimes the bot will need to access a variable. The most obvious example is the ‘Greeting’, where the bot uses the person’s name. This is all handled in the actual code. To actually tell the bot where to use this variable, you must use the syntax ‘{0}’. This will be replaced by the name of the person being greeted. You could, of course, choose not to use this – there’s no problem if you ignore the variables altogether. It just helps make the bot more responsive.

Here is a list of all the responses that have variables attached to them:

**Greet (0)** - Name

**Point out tourist spot (32)** - Position ("X, Y, Z") (Only one variable)

**Tell where going (66)** - Destination, distance in miles, distance in grid coordinates, destination

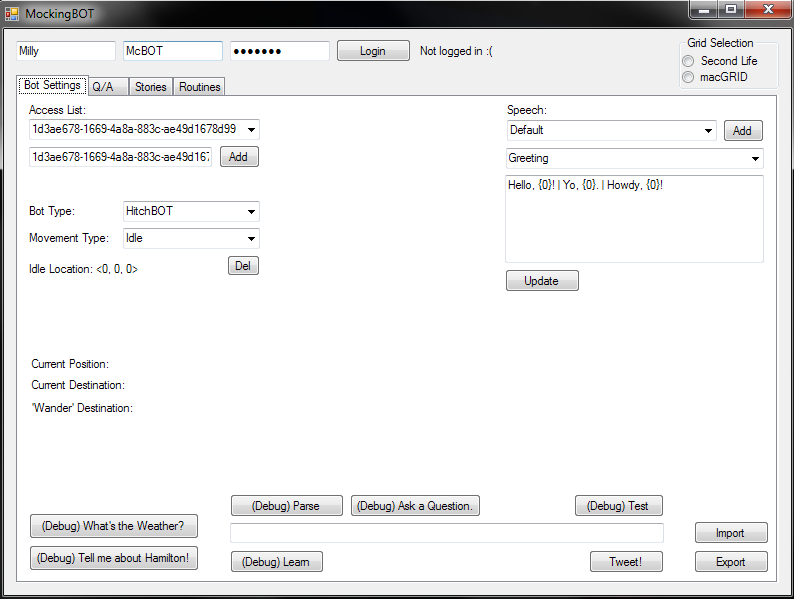
**Tell where going; city only (67)** - Destination

**Note city (68)** – Current city

**Greeting a known person (69)** - Name

**Too long to tweet (76)** – Length of tweet, how many characters over

**Talk about weather (77)** - Weather data



In cases where there are multiple variables, you can call the others using {1}, {2}, etc. The variables listed above are in the correct order for this.

You can also enter multiple possible responses, and the bot will pick a random one to use. You do this by separating responses with a vertical line ( | ). You can use this to make bots seem more natural – repeated dialogue can be rather jarring.

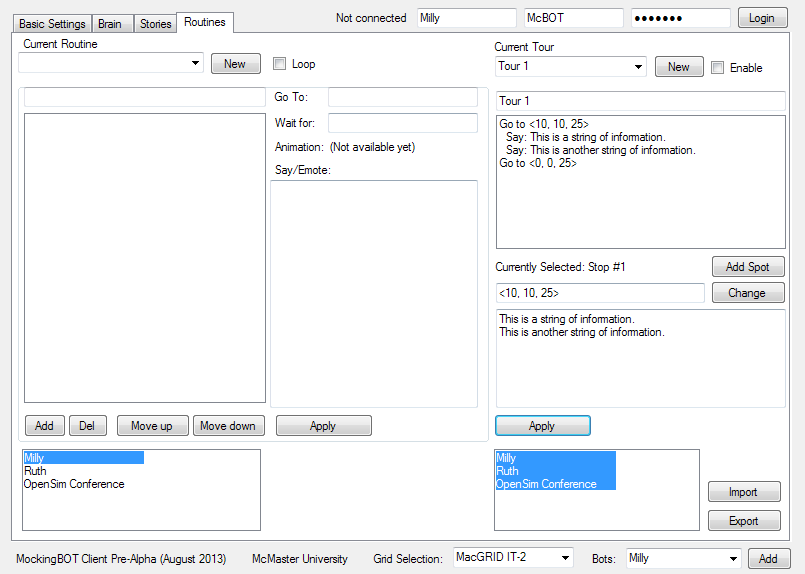
Programming – Tours

Tours are one of the main features of the bot, allowing them to actively show users around the region. The basic flow of this is as follows:

1. The bot asks to give a tour and the user accepts, or the user asks.
2. The bot looks through the tours on its list of allowed tours.
3. It takes a random tour from that list and compiles the list of destinations and strings of information to recite.
4. It goes off to the first location on that list on its own.
5. It waits at that spot for the tourist to show up. If they take too long, the bot offers to teleport them to its location.
   * If denied, the bot asks if the user would like the tour to stop. If they hear no response or the user says ‘no’, the tour ends. Else, the bot continues to wait and offer teleports on an interval.
6. When the tourist shows up, the bot recites information from a list, tied to that location.
7. The bot then asks if there are any questions or if it should proceed, which allows for the dialogue nodes to be used to program answers in freeform conversation.
8. After the user asks to proceed or enough time passes with no questions, the bot repeats the process from Step 4 until all the locations have been covered.

The main components of the tour are a) the speech strings used for general dialogue b) the list of locations to visit and c) the list of information strings for each location. All of these are available to be edited in the client, but more importantly, in-world.

To edit the tour in-client, simply open the ‘Tours’ tab.



Here you can select a tour to edit, set whether or not it’s enabled at all, change its name, see a full list of steps, add a new spot to go to, change its coordinates, and write the strings of information available at that spot. You can also set it to be enabled for specific bots, and not others.

The aim is to make it easy to edit existing tours. However, if you want to build a tour from scratch, you will find that while all the tools are there, it is difficult to tell exactly where you’re sending the bot based purely on the coordinates. The best way to find where you want bots to go is to program it in-world. This will update the client automatically.

To do this, you have to load a bot up in a grid. You can program it using IMs or Local Chat (after asking it to follow you around).

To begin, use the command !learntour. From now on, every string it hears (except for commands starting with an exclamation mark) will be remembered as something to say at this location, so be careful (though you can always edit mistakes in-client).

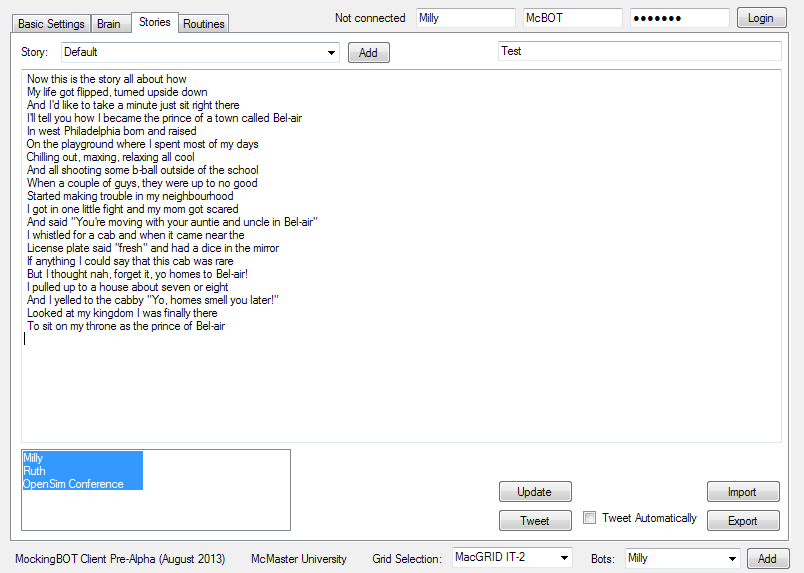
Simply go to a location in the region, and say !tourhere. The bot should respond with a generic speech string, to the effect of nodding and *writing down the coordinates of where you’re standing*. This adds a new location to its list of places to go. At this point, you can describe the location, effectively telling it what to say when it reaches this location during the tour.

You can repeat this process as many times as you want, until the end of the tour. At this point, say !stoptour, and the bot will return to its ordinary operations. The tour will be available in the client to be changed as much as you like.

(above) The same process is used for routines

Programming – Stories

Stories are of limited use right now – currently, if a bot is set to follow a user, they will ask for stories, and sometimes tell them in return. After a user tells them a story, the bot uses if they may share it – if so, the story is saved to memory, and can be recalled later. This can be decent for NPCs, and giving each bot more personality. You can ask a bot to tell a story even if they’re not following you.



In-client, this is a very simple process. You can select a story from the top-left menu, or create a new one. From there, it’s a simple matter of writing the story in the big, blank box in the middle of the screen. Take care to divide it up into paragraphs properly, as the bot breaks it apart using line breaks and tells the story one paragraph at a time. Most grids cap the length of a chat message at 1000 characters. This paragraph (starting from under the image above) is 500 characters.

In general, breaking apart a story by short lines is the best strategy. This is how users online tend to talk – in short bursts, rather than large paragraphs.

Once that’s done, you can dictate which bot can tell the story (default being all of them). Lastly, you *must click ‘Update’* *below*.

You can tweet the story, in which case the bot will break it down into multiple, 140-character bytes. There is an option to make it tweet stories automatically once a user tells it to them. This is not advised for use on a grid – it was intended for HitchBOT, which receives stories via text messaging, and doesn’t receive a terribly high volume. This limits the amount of tweets it would send. A bot on-grid could be hearing much longer stories, from a high volume of users. If you feel this won’t be the case, feel free to switch this on for your bots.