Dawn of the Titans Documentation

Release 1.0dev

Gregory Taylor

October 02, 2013

CONTENTS

Dawn of the Titans (dott) is a custom codebase for a space-based MUD. It is being crafted specifically for a single game, but the code is posted here to allow for contributions and forks for other games.

CHAPTER

ONE

GENERAL INFORMATION

1.1 Introduction

Dawn of the Titans (dott) is a MUD server that I (gtaylor) am working on for the space-themed game I've been kicking around in my head for a long time. It is very much a "work on it when I feel like it" project, any may never fully come to fruition as a running game.

The text-based medium is still special for me, and I think using projects such as these in order to learn new things and hone existing skills is great. Scratching multiple itches with one... stick?

1.1.1 Under the hood

dott is powered by Python, and is built on the excellent Twisted framework. CouchDB is used for data storage, but the game itself is entirely memory resident (so we're not really using any of CouchDB's unique capabilities).

1.1.2 Nifty features

- A two-process proxy and MUD server architecture that allows connections to be maintained while the MUD server is down or rebooting.
- An extremely simple JSON DB structure (CouchDB speaks JSON).
- The great async capabilities that Twisted brings to the table.

1.1.3 Where to go for getting help, reporting bugs, and sharing ideas

> **Warning:** Before going into any further detail, it is important to note that this codebase exists for the very selfish purpose of running a single, specific game on it. I (gtaylor) am not writing this as a general-purpose codebase. I am solely focused on my eventual game, so please do not take it personally if I am not interested in your idea! But please do fork away and make it your own.

The best way to get help, report bugs, or share ideas is our issue tracker. We make a conscious effort to watch and respond to things there.

1.1.4 License

dott is covered under the extremely permissive BSD License. Just please leave the copyright and attribution notices be and I'll be happy.

CHAPTER

TWO

DEVELOPER DOCUMENTATION

2.1 Installing dott

2.1.1 Setting up virtualenv + virtualenvwrapper

You'll want to install all of your pre-reqs in a virtual environment. This keeps a clone of your Python interpreter sandboxed elsewhere, allowing you to install specific versions of support packages that might be different from those found in your system's global Python install.

Install and set up the following two utilities for doing this:

- virtualenv (Shouldn't really require any setup)
- virtualenvwrapper (Requires some manual work, no Windows support)

Create your virtualenv:

```
mkdir ~/.virtualenvs
mkvirtualenv dott
```

2.1.2 Install prereqs

Install pip in your newly created `dott` virtualenv, and have it install everything in the `requirements.txt` file:

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

You'll also need to install CouchDB. The instructions to do so vary, but most Linux distributions have this packaged already. Once you have CouchDB set up, Futon is pretty helpful for browsing/editing the contents of your various DBs (they'll be created at the game's first boot).

2.1.3 Clone the repository

Your environment is ready, time to get a copy of the code (if you haven't already). You'll want to pull either from the main gtaylor/dott repository, or your own fork (if you've forked the project):

```
git clone git://github.com/gtaylor/dott.git
```

2.1.4 Amazon Simple Email Service

dott does not use a traditional SMTP connection. It'd be possible to write an SMTP backend, and we'd welcome the contribution for our developers, but I (gtaylor) don't want to spend the time on it myself.

You'll need an Amazon AWS account, and you'll need to sign up for SES. Signup is free, and I'll be shocked if you manage to even ring up a few pennies worth of usage. Sign up for both of these, and get your AWS API keys. You'll need them for the next step.

Once you're signed up, copy/paste the following into a Python module then run it:

```
import boto

# Substitute these two values with your own.
conn = boto.connect_ses('ACCESS_KEY', 'SECRET_KEY')
# A verification email will be sent to whatever address you want to send
# as, so you'll need to be able to check the inbox.
conn.verify_email_address('your.email@somewhere.com')
```

Once you receive the verification email for whatever address you state that you'll be sending as, follow the link in the email and that specific address will be verified as a valid "Send As" value for SES.

2.1.5 Configuration

cd into the newly created `dott` directory, and take a look at `settings.py`. You don't want to change any of the values in there. Instead, create a new file called `local_settings.py` and copy/paste the setting you'd like to override into said new file.

You'll need to change the following values at minimum:

- `SECRET_KEY` (used for hashing account passwords)
- `AWS_ACCESS_KEY_ID` (get this from your AWS account web dashboard)
- `AWS_SECRET_ACCESS_KEY` (get this from your AWS account web dashboard)
- `SERVER_EMAIL_FROM` (must be the email address you verified with SES)

2.1.6 Starting the game

dott is split into two different processes:

- Telnet proxy (`proxy.tac`)
- MUD Server (`server.tac`)

The proxy server accepts telnet connections and handles user authentication and registration. It deals with all of the boring protocol-related stuff to get players connected and whatnot. It also maintains everyone's connection if the MUD server goes down, allowing for seamless reboots without most players ever noticing.

The MUD server handles all of the game-related stuff. It holds the world, tracks NPCs, handles combat, and does all of the fun stuff. The proxy just pipes input into the MUD server, and spits back out whatever the MUD server says.

You'll need to start the two pieces separately:

```
twistd --pidfile=proxy.pid -ny proxy.tac
twistd --pidfile=server.pid -ny server.tac
```

After you run both of these, you should be able to connect to port 4000 and create a character.

Note: You may stop/start the two processes independently of one another, and normal operation will resume once they are both running at the same time. It's perfectly acceptable to restart server.tac, and is in fact how we handle code reloading.

2.2 Running Unit Tests

Unit tests are the preferred means of developing new features, and testing existing functionality for regressions. While we do have and do accept some changes without unit tests, we strongly prefer them.

If you look around the `dott` directory structure, you will see files named in the pattern of `tests.py`. These contain unit tests. They are scattered around the codebase with the code that they test. For example, in `src/server/objects/`, you'll see a `in_memory_store.py` module, and the unit tests for it at `tests.py`.

2.2.1 Before you begin

Unit tests are ran via `nose`, which is a unit testing module. You can install that like this:

```
easy_install nose
```

Or if you have pip:

```
pip install nose
```

2.2.2 Running all unit tests

To run the entire suite, first make sure that you are in the top-level `dott` directory. Then do this:

```
PYTHONPATH=. nosetests -s
```

Note: This will run the entire unit test suite. The -s captures stdout output, which is needed to show `print` statements, if you're using them while developing (they shouldn't be in your final commit/pull request).

2.2.3 Getting more specific with test selection

If you'd only like to run a certain test module, simply provide its name:

```
PYTHONPATH=. nosetests -s src/server/objects/tests.py
```

Since tests are grouped by `TestCase` classes, each of these modules will probably contain a handful of different `TestCase` sub-classes for the various API calls. For example, the `src/server/objects/tests.py` module has `InMemoryObjectStoreTests`. Let's say we only want to run the tests in `InMemoryObjectStoreTests`:

```
PYTHONPATH=. nosetests -s src/server/objects/tests.py:InMemoryObjectStoreTests
```

Now we're narrowed down to just the `InMemoryObjectStoreTests` TestCase object. But what if we just want to run one of the unit test methods on that class? For example, the `test_create_room()` method:

```
PYTHONPATH=. nosetests -s src/server/objects/tests.py:InMemoryObjectStoreTests.test_create_room
```

We're now only running that specific unit test. This is useful when developing methods, or tracking a specific regression.

CHAPTER

THREE

ADMINISTRATOR DOCUMENTATION

3.1 Basic Building

Note: This is all preliminary, and only partially implemented. What follows is best described as a scratch pad, that may eventually become a building guide.

3.1.1 Digging Rooms

- @dig <room-name>

3.1.2 Exits

- @open <alias/dir> <exit-name>[=<dest-dbref>]
- @link <exit-dbref>=<dest-dbref>
- @unlink <exit-dbref>

3.1.3 Things

- @create <thing-name>

3.1.4 Zones

@zone <obj-dbref> = <zone-dbref>

3.1.5 General object commands

- @name <obj-dbref> = <new-name>
- @nuke <dbref>
- @desc <dbref>=<description>
- @teleport <target-dbref>=<dest-dbref>
- @alias <dbref>=<primary alias> [<alias2>]

@alias/del <dbref>=<alias-to-delete> @alias/add <dbref>=<alias-to-add>

- @parent <dbref>=<parent-class>

3.2 Building in Spaaaacee....

Note: This is all preliminary, and only partially implemented. What follows is best described as a scratch pad, that may eventually become a space building guide.

3.2.1 Creating Solar Systems

Create a solar system:

- @dig/teleport Test Solar System
- @parent here=src.game.parents.space.solar_system.SolarSystemObject

Create two test planets:

- @create Test Planet 1
- @parent Test Planet 1=src.game.parents.space.solar_system.PlanetObject
- @create Test Planet 2
- @parent Test Planet 2=src.game.parents.space.solar_system.PlanetObject

Enter your first test planet so our new ship will be created here:

- enter test planet 1

Create a basic shuttle:

- @create Test Trafficker
- enter Test Trafficker
- @parent here=src.game.parents.space.ships.shuttles.trafficker.TraffickerSpaceShipObject
- @create Cockpit
- enter Cockpit
- @parent here=src.game.parents.space.ships.shuttles.trafficker.TraffickerSpaceShipBridgeObject

You now have a basic space ship. Let's create a Hangar on Test Planet 1:

- @dig/teleport Test Planet 1 Hangar
- @parent here=src.game.parents.space.hangar.RoomHangarObject

Now find the dbref of Test Planet 1 so we can set the hangar's zone. You'll want to use your dbref in place of `#36` below:

- @find Test Planet 1 Hangar
- @zone here=#36

You now have a dockable location. Find your Cockpit object and teleport to it (use your dbref in place of `#35` below):

- @find Cockpit
- @tel #35

- dock

You can now dock at your location like this:

- dock 36

You can also take off like this:

- launch

To warp to the other planet, look at the warp list and warp to it:

- warp
- warp 33

CHAPTER

FOUR

SCRATCH PAD

4.1 Scribbles

This is a random assortment of various scribbles. Concepts graduate from here, moving into their own relevant documents over time.

4.1.1 Universe Arrangement

The universe can be of any general shape, with the new player starting area in the center. The inner core of the universe will be safe, in that you can't attack other players in it. As one moves further outward, it becomes open season for everyone.

4.1.2 System Security

For now, we'll just have 'law-less' and 'policed' security states. In law-less systems, anyone can shoot anyone. In 'policed' systems, you can only shoot NPCs.

4.1.3 Mining

Make this more fun than EVE and other games. Mining will be done by firing oneself or remote-controlled robots at an asteroid. The player then works through a randomly generated area, finding ores and potentially fighting weird life forms.

Issues to work through:

- If the player is remote-controlling a robot, need some way to alert the player of things going on with their ship. IE: Someone warps in, or someone starts firing on them.
- Facilities for taking control of objects needs to be absolutely stout. Make sure to handle damage to their dormant player object.

4.1.4 Manufacturing

This can probably be a little more 'boring', at least initially. Assembly lines can be rented or installed in larger ships or bases. I like EVE's blueprint system for this. BPOs, BPCs. Make these relatively easy to obtain. BPOs for all of the common stuff. BPCs for everything else.

CHAPTER

FIVE

PLAYER DOCUMENTATION

The topics here will eventually be kept in a dedicated player guide. Things outlined here are considered our current approaches.

5.1 A Spacefarer's guide to ships

This guide is meant for all aspiring or current

5.1.1 Ship Classes

There are many different sizes of ships, typically called a *ship class*. While a ship's class does not limit it to certain roles, the class and size of a ship strongly influence what said ship can be used for.

A newer player will most likely start with Fighters, working their way up to whatever ship class feels most comfortable.

Tip: Bigger does not always mean better for every situation.

Fighters

Fighters are small, single-pilot ships that require no crew. While some fighter class ships may be able to carry more than one person, this is the solo pilot's bread and butter.

Frigates

Frigates are the smallest vessel that is typically manned by a crew, instead of a single pilot. Frigates are small and simple enough to be easy and efficient to operate without help or escort.

Cruisers

Cruisers are the next step up from frigates, trading speed and maneuverability for more defensive and/or offensive capabilities. Cruisers typically benefit from having others aboard.

Cruisers may have a difficult time fending off Fighters without Fighter or Frigate support of their own.

Battlecruisers

Battlecruisers are the size of a Cruiser, but typically carry Battleship weaponry. This allows for a cheap way to attack larger targets without having to rely exclusively on the more expensive and cumbersom Battleships.

Battlecruisers tend to be about as durable as a Cruiser. Due to their larger weapons (meant for larger targets), Battlecruisers may struggle to fend off Fighters and Frigates effectively.

Battleships

Battleships are large, lumbering mammoths, often bristling with lots of nasty weaponry. These ships' sole purpose is to deal out as much pain as possible, meanwhile absorbing as much retaliation as possible.

Battleships are almost never seen alone, and require the support of smaller ships. Battleships are particularly ill-equipped to defend themselves from large groups of Fighters and Frigates.

Capital Ships

Capital Ships are the largest ships to grace the space lanes. These range in purpose from mass cargo freighters, to factory ships, to Carriers and Dreadnaughts.

Capital Ships are typically very specialized, extremely expensive, and reliant upon fleet support. Capitals may often field and maintain a complement of Fighters.

5.1.2 Basic ship operation

Assumptions

This guide assumes that the reader either has a ship, or is interested in learning more about how ships work before purchasing one. We assume that the reader's ship is landed in some hangar (whether it be on a planet, in a station, or inside of another ship).

Entering a landed ship

Before following any of the instructions in this guide, you'll want to enter your ship. You can do so using the **enter** command from within a hangar.

Note: Entry to ships is restricted based on identity and/or corporate affiliation. Make sure you're entering your own ship.

Depending on the size of the ship you are entering, you will either hop straight into the cockpit, or find yourself in one of the ship's rooms (probably the airlock). Fighters and some Frigates are small enough to only have a single "section", the cockpit. Larger ships have multiple sections/rooms within, so you'll need to find the *Bridge*. A ship's Bridge is functionally equivalent to a smaller ship's Cockpit, it's just much bigger and can hold quite a few people.

Tip: The only place a ship may be controlled from is its Cockpit or Bridge.

Inspecting your ship

The best overall indicator of your ship's status and general health can be found via the **status** command. This will show you a quick, high level view of things such as:

- Where your ship is.
- Armor and Shield levels.
- Any special conditions being applied to the ship (jammed, warp scrambled, etc.)
- A very brief weapons/module readout.

> **Warning:** Show some example command output here.

You'll want to consult your status readout during combat to help keep track of how you're holding up.

Launching from a Hangar

Your ship should be sitting in a hangar of some sort at this point. To take off and get into space, you'll want to use the **launch** command.

After a short delay, you'll find your ship floating out in space by whatever planet/station/ship you launched from.

Note: Many ship commands aren't available while landed/docked.

Sensors and Scanners

Whether you are in a single-seat Fighter or a massive Carrier, your only reliable view of the outside world comes courtesy of your ship's sensors. To see what other ships or objects are near your ship, you'll use the **contacts** command. Only ships that are in the same location as you will be shown with this readout.

> **Warning:** Show some example command output here.

Some basic details on what each object is are returned, along with its unique numerical ID. You will use this numerical ID with any command that interacts with objects in space.

Scanning ships/objects in space

To get more details on something that neighbors you in space, use the **scan** command with the ship/object's unique ID number. For example, if a Fighter in your location has the ID `1835`, I could scan it like this:

```
scan 1835
```

This will return a readout very similar to your own `status` command. The big difference is that some of the information will be less specific, or omitted altogether.

Note: Any time you scan another ship, its sensors may notice. This depends on your scanner and your target's sensors. Know that some pilots take scanning as an aggressive action.

Navigating Solar Systems

Your ship should now be outside of whatever planet/station/ship it launched from. You are now flying around in a *Solar System*. You can see which Solar System you're in via the **status** command. Solar Systems are inter-connected by *Jump Gates*, which we'll cover in more detail later.

Solar Systems are partitioned into *Locations*. For example, "Near Station J12", "Near the Sol Jump Gate", or "In Asteroid Belt 12". These are roughly equivalent to a traditional MUD's Rooms. If Location is to Room, Solar System is to a traditional MUD's Areas or Zones.

Ships can only interact with other ships that are in the same location. For example, it's not possible to fire at a ship that is near the "Sol Jump Gate" from your position in "Asteroid Belt 12".

Warping to Locations

Almost every ship in the game is equipped with a Faster-Than-Light warp drive. These drives are the way you get from Location to Location within a Solar System. To see your current location, you use the **status**, or the **warp** command with no arguments.

To see a full list of locations that you may warp to, use the **warp** command with no arguments.

> **Warning:** Show some example command output here.

Note that each Location has a unique ID number, just like ships do. You'll use this number with the **warp** command to warp around the system. For example, let's say we have a Location called "Asteroid Belt 12" that has a unique identifier of `1832`:

```
warp 1832
```

This would fire up our warp drive, and after a brief delay, we'd appear in the Asteroid Belt.

Traveling to Different Systems

To travel to different systems, you'll first need to **warp** to a *Jump Gate*. Jump Gates are large, stationary mass accelerators that fling you to distant locations.

Once you have picked out and warped to a Jump Gate, you'll actually jump through it with the **jump** command. Note that it is not necessary to specify any arguments with the **jump** command, as we can infer that you moved to a specific Jump Gate by warping there.

After a short delay, you will find yourself far from where you started. You are then free to **warp** around your new Solar System, or **jump** back to your previous location.

Docking/Landing

It is very unwise to leave your ship floating out in space when you retire for the night. It is unlikely that your neighbors will resist the temptation to destroy your ship (with your character in it).

You may dock or land your ship on Planets, Stations, or other Ships. Your options will vary based on who you are, your corporate affiliation, and your personal and corporate standings.

The easiest way to find likely places to dock is to look at your **warp** list for Locations that look like Planets or Stations. Warp to either of these, then use the **dock** command with no arguments to see a list of docking/landing options for your current Location. Like almost everything else, each location has a unique ID, which you can use with the **dock** command:

```
dock 1293
```

After a short delay, your ship ends up in whatever hangar you selected.

Note: Your docking options may vary based on your identity and corporate affiliation.

5.1.3 Command Reference

Ship info

- **status**: Ship status display.
- **modules**: Ship module display.
- **weapons**: Weapon display.

Movement

- **land or dock**: Lists all landable/dockable places.
- **land <dest> or dock <dest>**: Lands/docks somewhere.
- **warp**: Lists all warp points.
- **warp <dest>**: Warps to a point in the system. Gates, planets, etc.
- **jump <gate>**: Jumps through a warp gate or wormhole.

Sensors

- **contacts**: Lists other Ships and objects in your Location.
- **scan <id>**: Shows details on another Ship or object in your Location.

Combat

- **fire <Weap#> <ID>**: Starts firing one of your weapons at the specified target.
- **fire <ID>**: Fires all weapons at target.
- **fire s**: Stops firing all weapons.

Modules

- **activate <Module#>**: Activates a ship module.
- **deactivate <Module#>**: De-activates a ship module.
- **activate all**: Activates all ship modules.
- **deactivate all**: De-activates all ship modules.