Clojurecraft Documentation

Release 0.0.1

Steve Losh

July 30, 2015

Contents

Clojurecraft is a framework for writing Minecraft bots in Clojure.

It's definitely not ready for actual use yet, but if you want to poke around it has some very basic functionality. Just remember that *everything* is subject to change.

Check out the Quick Start Guide if you want to poke around.

- Mercurial Repo: http://bitbucket.org/sjl/clojurecraft/
- Git Repo: http://github.com/sjl/clojurecraft/
- Documentation: http://clojurecraft.rtfd.org/
- Issue Tracker: http://github.com/sjl/clojurecraft/issues/
- Mailing List: clojurecraft@librelist.com (send a blank email to subscribe)

CHAPTER 1

Table of Contents

1.1 Quick Start

Clone down the repo with Mercurial or Git:

```
hg clone http://bitbucket.org/sjl/clojurecraft
git clone http://github.com/sjl/clojurecraft.git
```

Or if you don't want to bother you can just download and extract a tarball:

```
wget https://bitbucket.org/sjl/clojurecraft/get/tip.tar.gz -O clojurecraft.tar.gz
tar xzf clojurecraft.tar.gz
```

Grab the dependencies using Leiningen 2 and fire up a REPL:

```
cd clojurecraft
lein deps
lein repl
```

Now you'll need to open another terminal window to download and run the vanilla server:

```
cd path/to/clojurecraft
bundled/bootstrap.sh
bundled/runserver.sh
```

Wait for the server to finish loading (it'll say "Done") and then connect to it with a normal Minecraft client so you can watch your bot.

Now you can go back to your REPL and get started. Import the things you'll need:

```
(require '(clojurecraft [core :as core] [actions :as actions]))
```

Create a bot connected to your local server:

```
(def bot (core/connect core/minecraft-local "desired_username"))
```

Right now Clojurecraft doesn't support authentication, so it's turned off on the bundled server and you can choose any username you like. You can pass `nil` instead of a username to get a random one.

Give your bot a little time to connect. You should see it appear in the world through your Minecraft client.

Once your bot is in the world you're all set to play around. At the moment the only action implemented is basic movement. Move your bot around with `actions/move`:

```
(actions/perform! (actions/move bot 2 0 1))
```

The numbers are the x, y, and z distance you wish to move. For now you can't use the `y` argument – you must always pass `0`.

If the bot doesn't appear to move, you may have tried to make an illegal move (like moving into a block). Try some other numbers. Check the server output for more information if something goes wrong.

Now try jumping:

```
(actions/perform! (actions/jump bot))
```

Clojurecraft isn't stable and is evolving quickly, but you can check out these docs to read about some of the design decisions. As soon as you see a `v1.0.0` tag in the repo you'll know I've started caring about backwards compatibility.

1.2 Basic Concepts

The basic flow of creating a Minecraft bot with Clojurecraft looks like this:

- Connect to a Minecraft server to get a `Bot` object.
- Define and add events handlers to the bot.
- Define and add loops to the bot.
- Let the bot do its thing.
- Disconnect the bot from the server.

You can also manually tell a bot to perform actions from a REPL.

1.2.1 Connecting

You can connect to a Minecraft server with the `connect` function:

```
(clojurecraft.core/connect {:name "hostname" :port INT} "username")
```

You can also pass `nil` as a username to get a random string of letters.

When you connect to a server you get a `Bot` object back. Once you've got a bot you can query it for data about its world, tell it to perform actions, and add event handlers and loops.

1.2.2 Event Handlers

Event handlers are functions that let your bot react to things that happen in the world. Check out the Event Handlers page for more information.

1.2.3 Loops

Loops are functions that run every `N` milliseconds and let your bot query the world and perform actions. Check out the Loops page for more information.

1.2.4 Disconnecting

To disconnect a bot from the server you simply call the `disconnect` function:

```
(clojurecraft.core/disconnect bot)
```

1.3 Data Structures

World data is shared between all the bots you create in a single process. This helps keep memory usage down by not storing duplicate copies of chunk and entity information for each bot in the same world.

1.3.1 Worlds

`World` objects have several pieces of data. Please read the Transactions section to learn why the data is structured the way it is.

`(:time world)` is a ref containing the current world time.

`(:entities world)` is a ref containing a map of entity IDs to `Entity` refs.

`(:chunks world)` is a ref containing a map of chunk coordinates ([x y z] vectors) to `Chunk` refs.

1.3.2 Locations

`Location` objects represent the locations of entities in the world. They have the following pieces of data:

- `(:x location)`
- `(:y location)`
- `(:z location)`
- `(:yaw location)`
- `(:pitch location)`
- `(:stance location)`
- `(:onground location)`

1.3.3 Entities

`Entity` objects represent a single entity in the world. One of these is your bot's player.

`(:eid entity)` is the ID of the entity.

`(:loc entity)` is a `Location` object representing the location of the entity in the world.

`(:despawned entity)` is a boolean that indicates whether the entity has despawned. You should never need to read this, but please read the Transactions section for the reason why it's included.

`(:velocity entity)` is the y velocity of the entity. Only exists for bots, and you should never need to touch it.

1.3.4 Chunks

A chunk has four arrays representing the data for blocks in the chunk. You shouldn't need to access chunk data directly – there are helper functions in `clojurecraft.chunks` that will look up block objects for you.

1.3.5 Blocks

1.3.6 Bots

`Bot` objects are your gateway to observing and interacting with the world.

`(:world bot)` is a `World` object representing the bot's world.

`(:player bot)` is a ref containing the `Entity` representing the bot. This is just a shortcut so you don't have to pull it out of the `:entities` map in the bot's world all the time.

1.4 Transactions

There are two main types of data you'll want to observe with your bots: chunks and entities. A `World` object contains two maps: one of entities and one of chunks.

Each of these maps is a ref, and each of the entries in each map is also a ref. This may seem excessive – why not simply make each map a ref *or* each entry a ref?

1.4.1 Top-Level Refs

The maps themselves clearly need to be refs so that multiple bots sharing the same world can update them safely.

1.4.2 Entry Refs

To understand the reason for making each entity a ref consider a bot with the following goal:

"Find all the hostile mobs. Pick the one with the lowest health and attack it."

Now imagine that during the process of picking a mob to kill the bot received an update about one of the peaceful entities.

If the entries of the map were not themselves refs the bot would *have* to synchronize on the entire map. This peaceful entity update would cause a retry of the transaction even though the bot doesn't care about peaceful entities at all!

Making each entity its own ref means we can do the following:

- **Inside of a dosync:**
  - Find all the hostile mobs.
  - `ensure` on all of them.
  - Perform our calculations.

This lets us ignore updates to peaceful mobs, but retry when a hostile mob is updated (perhaps someone else has killed one). Keeping the "find mobs" step inside the dosync ensures that if a mob despawns we will be looking at an accurate list the next time we retry.

Note that if a new hostile mob is spawned it will not cause a retry. If you are performing an action that needs perfectly accurate data you can always synchronize on the maps themselves, but be aware that this will probably not be very performant.

1.4.3 Entity Despawns

This also reveals the reason for the `:despawned` entry in an `Entity` object: if we simply removed the entity from the map when it despawned any transactions depending on that entity wouldn't be restarted. Setting the `:despawned` value on an entity modifies it and triggers appropriate retries.

1.5 Actions

Actions are functions that take a `Bot` object and some arguments, and return a map representing the action.

They return a map instead of actually performing the action to make it easier to unit test your event handlers and loops. This also makes them pure functions that can be retried in transactions.

1.5.1 Performing Actions

If you want to make your bot perform an action immediately you should use `perform!` to make it happen:

```
(clojurecraft.actions/perform! (clojurecraft.actions/jump bot))
```

1.5.2 Available Actions

There are a number of actions your bots can perform. More will be added in the future.

chat

`(clojurecraft.actions/chat bot message)`

Tells the bot to send a chat message.

Note: you *will* get kicked from the server if your message is too long. Here's the formula for the vanilla Minecraft server:

```
(defn too-long? [bot message]
  (> (+ 3 (count (:username bot)) (count message))
     100))
```

It's up to *you* to avoid sending messages that are too long. This action doesn't handle it because there are multiple options you might want:

- Ignore messages that are too long.
- Split them into multiple messages.

You might also be writing a bot for a modded server that allows longer messages.

jump

```
(clojurecraft.actions/jump bot)
```

Tells the bot to jump, if possible.

look-to

```
(clojurecraft.actions/look-to bot pitch)
```

Changes the position of the bot's head. `pitch` can be anywhere from `-90` to `90`.

- `-90`: looking straight up.
- `0`: looking straight ahead.
- `90`: looking straight down.

look-up

```
(clojurecraft.actions/look-up bot)
```

Changes the position of the bot's head to look straight up.

Exactly equivalent to `(clojurecraft.actions/look-to bot -90.0)`.

look-down

```
(clojurecraft.actions/look-down bot)
```

Changes the position of the bot's head to look straight down.

Exactly equivalent to `(clojurecraft.actions/look-to bot 90.0)`.

look-straight

```
(clojurecraft.actions/look-straight bot)
```

Changes the position of the bot's head to look straight ahead.

Exactly equivalent to `(clojurecraft.actions/look-to bot 0.0)`.

turn-to

```
(clojurecraft.actions/turn-to bot yaw)
```

Changes the direction the bot is looking. `yaw` can be anywhere from `0` to `359`.

- `0`: looking in the -z direction.
- `90`: looking in the -x direction.
- `180`: looking in the +z direction.
- `270`: looking in the +x direction.

A few notes:

- Changing your bot's yaw by too large an increment at once seems to be handled weirdly by the vanilla client. It will turn partway toward the destination yaw, pause, and then turn the rest of the way.
- Changing the direction the bot is looking affects the *head* of your bot, not necessarily the *body*. The body will rotate as much as is necessary to prevent an Exorcist-style head-spin. I don't know of a way to force the body to face a certain direction at the moment, but I'll keep looking.

turn-north

```
(clojurecraft.actions/turn-north bot)
```

Changes the direction the bot is looking to north.

Exactly equivalent to `(clojurecraft.actions/turn-to bot 90.0)`.

turn-south

```
(clojurecraft.actions/turn-south bot)
```

Changes the direction the bot is looking to south.

Exactly equivalent to `(clojurecraft.actions/turn-to bot 270.0)`.

turn-east

```
(clojurecraft.actions/turn-east bot)
```

Changes the direction the bot is looking to east.

Exactly equivalent to `(clojurecraft.actions/turn-to bot 180.0)`.

turn-west

```
(clojurecraft.actions/turn-west bot)
```

Changes the direction the bot is looking to west.

Exactly equivalent to `(clojurecraft.actions/turn-to bot 0.0)`.

move

```
(clojurecraft.actions/move bot x y z)
```

The `move` action adjusts the location of the bot. This lets it move around the world.

Right now you can't really use the `y` argument. Use `clojurecraft.actions/jump` instead.

This action is fairly low level. Expect to see some fun path-finding algorithms/libraries in the future that will remove the need to call this directly.

respawn

```
(clojurecraft.actions/respawn bot)
```

The `respawn` action tells your bot to respawn. Only send this if your bot has died, because I'm not sure what the vanilla server will do otherwise.

1.6 Events

One of the main ways your Clojurecraft bots will interact with the world is through event handlers.

Event handlers are functions you create that respond to events that happen in the Minecraft world. They return a list of Actions that you want your bot to perform.

Event handlers are pure functions that should take the bot as their first argument. Their other arguments will depend on the particular handler.

1.6.1 Creating and Registering Event Handlers

The first thing you need to do is create an event handling function:

```
(defn jump-on-chat [bot message]
  [(clojurecraft.actions/jump bot)])
```

Then register the handler for the action:

```
(clojurecraft.events/add-handler bot :chat #'jump-on-chat)
```

Notice that you don't pass the function directly to the `add-handler` function. You pass a symbol to the function. This is two extra characters to type, but it means you can redefine the function in the REPL and your changes will take effect immediately in all of the currently running bots.

1.6.2 Available Events

You can register handlers for the following events.

`:chat`

```
(defn chat-handler [bot message]
  [... actions ...])
```

Chat events are fired when a chat message arrives.

A few things to note when writing bots for the vanilla server:

First, a "chat message" includes things like "foo joined/left the game.". You can parse these and take appropriate action if you like.

The other important point is that *you will receive your own messages*. If you fire a chat action you'll get a message back for it! A helpful function to use might be something like this:

```
(defn message-is-own? [bot message]
  (clojure.contrib.string/substring? (str "<" (:username bot) ">")
                                     message))
```

`:dead`

```
(defn dead-handler [bot]
  [... actions ...])
```

This event is fired when your bot dies.

You'll probably want to respawn when this happens:

```
(defn dead-handler [bot]
  [(clojurecraft.actions/chat bot "WHY YOU DO THIS?")
   (clojurecraft.actions/respawn bot)])
```

1.7 Loops

Another way your bots will interact with the world is through loops.

Loops are functions that repeatedly run with a delay in between each run. They return a list of Actions that you want your bot to perform, just like event handlers.

Loops are pure functions that should take a single argument: the bot.

1.7.1 Adding Loops

To add a loop to your bot, you first need to create the loop function:

```
(defn jump [bot]
  [(clojurecraft.actions/jump bot)])
```

Now you can add it to the bot:

```
(clojurecraft.loops/add-loop bot #'jump 3000 :jump-loop)
```

The first argument to the `add-loop` function is your bot.

Next is a symbol to your loop function. The reason for passing a symbol is the same as the reason you pass a symbol to event handlers.

Next is the number of milliseconds you want to wait in between each run of the loop function.

Finally you must pass a "loop ID" keyword. It can be anything you like, but it must be unique for each loop added to a given bot. This is what you'll use to remove the loop from the bot later.

This example adds a loop to the bot that will make it jump every three seconds.

1.7.2 Removing Loops

Removing a loop from a bot is as simple as calling `remove-loop` with the bot and the loop ID you used when adding the loop:

```
(clojurecraft.loops/remove-loop bot :jump-loop)
```