

# grind

0.2.0

a log analysis and data aggregation tool

Computer Science BSc. graduation project submission by Ahmad Amireh,  
#0078401 with the supervision and management of Dr. Baddar, Sherenaz.

# Table of Contents

<b>What it is</b>	<b>4</b>
<b>Features</b>	<b>4</b>
<b>License</b>	<b>5</b>
<b>Getting started</b>	<b>6</b>
<b>Grind overview</b>	<b>7</b>
<i>grind terminology &amp; entities</i>	7
<i>Structure</i>	10
<i>Watchers: viewing your data</i>	12
<i>Keepers: archiving your data</i>	12
<b>Installing grind</b>	<b>13</b>
<i>The core</i>	13
<i>Watchers</i>	15
<i>Keepers</i>	16
<b>Configuring grind</b>	<b>18</b>
<i>The core</i>	18
<i>Watchers</i>	19
<i>Keepers</i>	19
<b>Testing</b>	<b>20</b>
<i>Simulating feeds</i>	20
<b>Tutorial 1: Birth of the chimp</b>	<b>21</b>
<i>Receiving messages</i>	21
<i>Formatting data into something useful</i>	22
<i>Modeling the formatted data into views</i>	25
<i>Joining the party</i>	26
<b>Tutorial 2: Where no monkey is safe</b>	<b>28</b>
<i>The format</i>	28
<i>The klass</i>	29
<i>Modeling the data... Views revisited</i>	30
<b>Appendix A: Some Lua to get you running with grind</b>	<b>33</b>
<i>Lua variables and data-types</i>	33
<b>Appendix B: grind internals</b>	<b>36</b>
<i>Composite signature formats</i>	36
<i>Transport format</i>	36
<i>Labels are unique</i>	37



# What it is

grind is a tool that helps you extract information stored in your application log files, and gives you the ability to reformat that data or view a subset of it at any time.

Log files could get messy, and when your software matures and gets deployed on production servers it becomes a daunting task to find out what's going on. Even if there's only a single rotating log file, grind saves your time by making it easy to reach any kind information you're after.

## Features

- View and analyze logs from different applications across multiple servers from a single, unified interface
- Utmost flexibility thanks to grind being completely [PCRE](#)-driven
- Easy to configure; grind uses [Lua](#), internally and for configuration, with super-clean syntax
- Archiving of formatted data for offline querying or history backtracking
- Bandwidth-efficient; nothing is transmitted except for what you're currently viewing
- Filtering system that allows you to control exactly what you want to receive at any time
- Horizontal and vertical parsing; pieces of data can be extracted not only from a single message, but from several ones too
- Completely free and open source!
- Ready examples for popular applications like Apache httpd, Ruby on Rails, Ogre3D, and CEGUI

# License

The MIT License - Copyright (c) 2011-2012 Ahmad Amireh <[kandie@mxvt.net](mailto:kandie@mxvt.net)>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

# Getting started

First you need to get grind [installed](#) and [configured](#). After that, you should take a look at its [structure](#) to get briefed on some terminology and how things work. Afterwards, go through the [tutorials](#) for a good look at using grind's features, or if you're really impatient, check out the [5-minute](#) tutorial.

# Grind overview

In this article you will get to see how grind is structured and how it works.

## grind terminology & entities

There are 3 entities in grind which you need to learn about and configure: application groups, classes, and views.

### Application Groups

Application groups in grind are an abstraction for any number of applications that might differ in their logging format or source (aka *feed*) while still manageable as a single entity.

Say you're developing a web service and you have a web server (like httpd or nginx) that constantly logs page access information and so on, and you also have a web application (like Rails or a PHP framework) that handles requests. These applications might log to different files, and more importantly, using entirely different formats. Defining an application group will allow you extract and format information from both logs intermittently and seamlessly.

### Signature formats

Application groups define a number of signature formats: regular expressions that identify messages coming from certain feeds. Since grind has no idea how the messages look like, and more importantly, where they start and end, signature formats need to specify that. An example taken from Apache httpd 2.2 access logs:

```
127.0.0.1 - - [10/Jul/2012:16:36:42 +0300] "GET / HTTP/1.1" 304 -
```

All log entries in that file begin with an IP, followed by - - and then a timestamp between [ ]. A signature format would match this "signature" of log messages to tell grind that this is an access log message, beginning with that match and ends when another one is found.

## Message formats

Application groups also define one or more message formats which extract meta-data contained in any log entry, such as the logging context, an application name, etc.

Following our example above, we can define a message format that extracts the HTTP method used for the request, the URL, and HTTP version. If the format fails to capture this information, it will simply be ignored for this entry.

Message formats are directly used by classes, the second (of three) entities in grind.

## Klasses

Klasses (with a 'k' for implementation limitations) represent a classification of *messages* that belong to certain message formats.

For example, you might be interested in all the errors that are being logged; to retrieve only those messages, you can define a klass that will reject any message that is not an error.

Sometimes for certain applications, only one klass will make sense, and for others, as many as 10 could be defined, and some of them could belong to even multiple message formats!

A good way to think of klasses is: "what kind of data do I want?".

Finally, a klass belongs to an application group and can contain many views as we will see below.

## Views

Views are the last of grind entities and they're similar to what views are in the MVC model; they define how the data, or a subset of it, is represented.

Take a look at the following log entry, taken also from an access log for http, but this time with a different, more complex format:



```
mxvt.net:80 178.77.158.31, 10.240.82.194 T-7FrX8AAAEAAHZl7a0AAAAF
- - [12/Jul/2012:12:40:13 +0000] "GET
/assets/image.png HTTP/1.1" 200 452 "http://www.google.com"
"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.11
(KHTML, like Gecko) Chrome/20.0.1132.57 Safari/536.11"
```

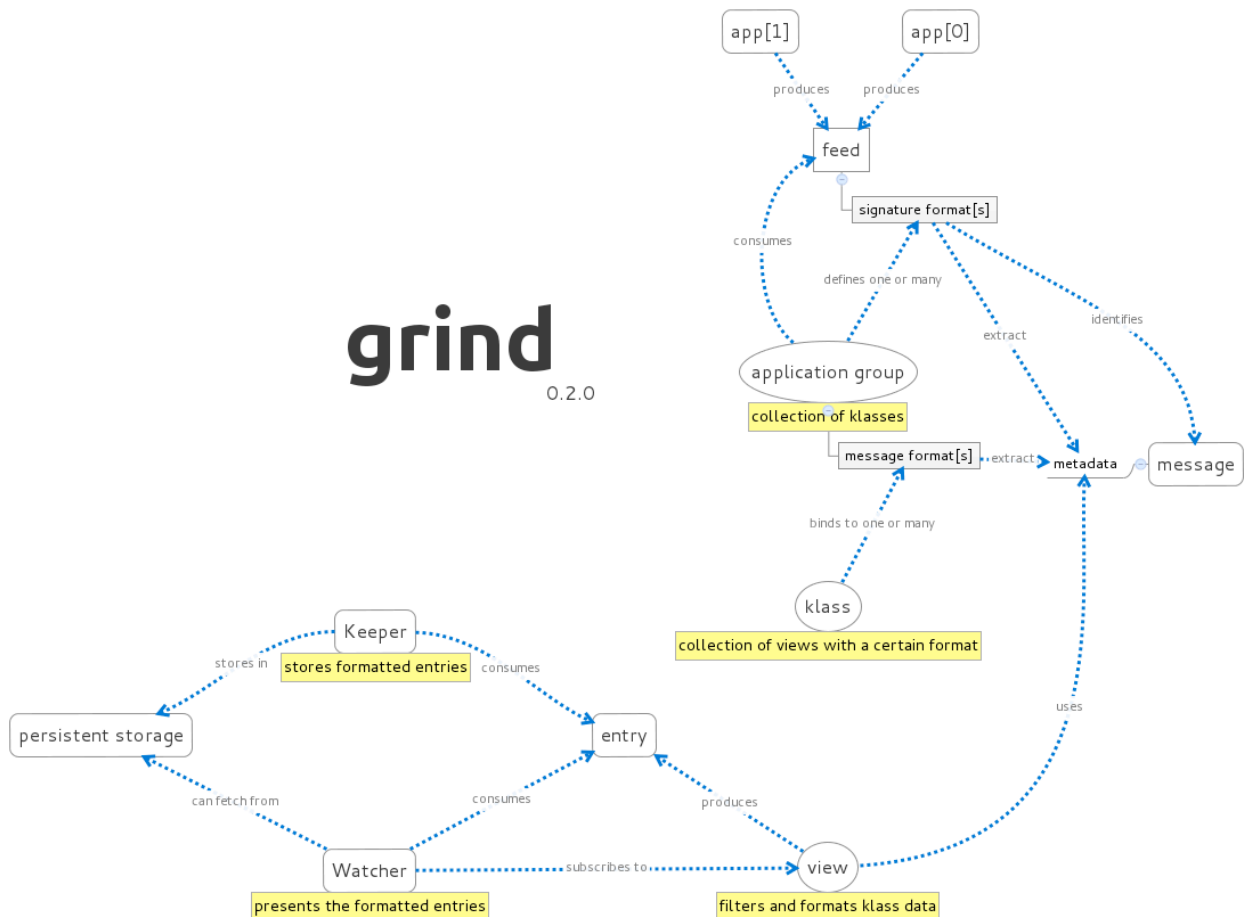
That certainly looks different, and much longer, than the one we saw in the first section. Well, this entry contains a number of fields:

- the host address and port
- an httpd virtual host address(es)
- a request UUID
- a timestamp
- the message, which contains: the HTTP method, URL, HTTP version, HTTP RC, size, referrer, and agent!

That might be too much information to look at at any one time. Might be you're only interested in the URLs and response codes for those hits, and that's where views kick in: you can customize the fields that are shown per view and you can also re-format the data in any way you like.

# Structure

This is how the system looks like, its entities, and the interaction between them:



## The pipeline in English

1. an application produces a log entry which is relayed to grind via some means (`syslog`, `tail -F | netcat`, etc.)
2. grind routes the entry to the application group's handler at the specified port
3. the handler attempts to identify the message using one of the signature formats
4. the metadata from the signature format is extracted (ie, timestamp)
5. every message format defined is asked whether they match this entry

6. classes bound to all matching formats are asked whether they apply to this entry
7. the views of all applicable classes are invoked to do their formatting
8. any formatted-entry produced by a view is relayed to all subscribed Watchers
9. Watcher represents the data via its interface
10. any attached Keeper will store the formatted entry in its storage engine

*Watchers and Keepers are explained in the next section.*

## Watchers: viewing your data

grind and its entities, referred to as the core, do not by themselves output anything. That responsibility is delegated to separate, stand-alone components called the Watchers.

Technically, this separation was made in order to decouple the functionality from how the output is viewed. This flexibility allows us to "watch" grind in different ways:

- from the terminal, using an ncurses-based Watcher
- from the browser, using a web-based Watcher
- from your desktop, using a Qt-based cross-platform Watcher

As of this writing, only a web Watcher is implemented using Ruby, HTML5 WebSockets, and JavaScript.

## Keepers: archiving your data

Most of the time you won't be able to process the data in real-time due to the amount of traffic, the Watcher might not be able to keep up (as is the case with the web watcher, because DOM manipulation is so slow). Or maybe what you're looking for has already been logged and you want to look it up. For this reason, Keepers can be used to store grind's output in a database engine. The Watcher you're using can connect to a Keeper's archive and allow you to dig up any information using queries and regular expressions.

As of this writing, only a MongoDB Keeper is implemented using Ruby and Sinatra.

# Installing grind

To set up grind, you should begin by building the core, then getting a Watcher set up, and optionally a Keeper.

## The core

The core is composed of C++ source files and Lua scripts. CMake is used for building the source.

As of this writing there are no packaged builds and you will have to do this manually.

### C++ dependencies:

Make sure you got [CMake](#) installed. Afterwards, you need to get the following development libraries set up:

- [gcc 4.6.2+](#) or any fully C++0x-compliant compiler
- [boost](#) 1.46 or higher (the used parts of boost are: filesystem, thread, date\_time, system, and regex)
- [log4cpp](#)
- [Lua](#) development headers

**Note:** for the remainder of this article, the directory which contains the grind repository will be referred to as \$GRIND.

## Building the C++ core

Fire up a terminal and go to \$GRIND, and create a directory called `build` which will contain CMake's temporary build files:

```
cd $GRIND; mkdir build && cd build;

ccmake ..
```

Now you are faced with the ncurses CMake interface. Type `c` to configure the package (look for the dependencies). If the configuration fails, make sure you haven't missed any of the dependencies above. When configured, type `g` to generate the build script and `ccmake` will now exit.

```
make && sudo make install
```

Will build the grind C++ core, and the Lua bindings which will be installed to `/usr/local/lib` (or whatever `INSTALL_PREFIX` you've chosen). A configuration script will also be installed to `/etc/grind/config.lua`, we will get to that later.

### About the bindings

While the Lua bindings wrapper is already generated for you, if you want to generate it yourself you need [SWIG](#) and a Python interpreter (2 or 3, doesn't matter.) Once that is done, go to the build directory and type:

```
make -B lua_grind_SWIG.
```

## Preparing the Lua core

Make sure you have [Lua 5.1](#) installed, as well as the [luarocks](#) package manager. When `luarocks` is set up, you can use it to install the Lua dependencies:

- `lrexlib-pcre`
- `dkjson`
- `luafilesystem`
- `lua_cliargs`
- `luasocket` - optional, needed for some tests and helper tools

### For Debian and Ubuntu users

I've had to do `ldconfig` after `make install` for Lua to find the bindings, you might have to do so as well.

Now you should have grind set up, but it still needs [configuring](#). You should probably configure the grind core before setting up a Watcher.

Running grind is as simple as:

grind

## Watchers

Watchers are the interface for grind. A watcher allows you to connect to a grind instance and display entries that you're interested in.

### The Ruby HTML5 watcher

This is a web UI. It uses [EventMachine](#) with HTML5 WebSockets for communication between the JavaScript and the grind instance. It also uses [Sinatra](#) for the actual web application. Stylesheets are written using [SASS](#), and [jQuery](#) is also used.

You will need Ruby 1.9.2+ and the following gems:

- `sinatra`
- `sinatra-content-for`
- `sass`
- `json`
- `yajl`
- `eventmachine`
- `em-websocket`

You also need a websocket-enabled browser.

Running the Watcher is done in two steps. First, run the internal Watcher server:

```
cd grind/watchers/ruby
ruby server.rb
```

And then the web server:

```
cd grind/watchers/ruby
ruby app.rb
```

### Running the Watcher under PhusionPassenger and httpd

Simply define a virtual host entry in your Apache httpd config file that contains something like this:

```
<VirtualHost *:80>
  ServerName grind.mydomain.com
  DocumentRoot /path/to/grind/watchers/ruby/public
  PassengerAppRoot /path/to/grind/watchers/ruby
  <Directory /path/to/grind/watchers/ruby/public>
    AllowOverride all
    Options -MultiViews
  </Directory>
</VirtualHost>
```

Note that you still have to make sure `server.rb` is running.

Grats! With the core and a Watcher set up, you are now ready to use grind.

The next step is optional, you only need to do it if you want to archive the data (which you should, really.)

## Keepers

Keepers store and archive the entries produced by grind in a database for offline viewing, querying, or even stat aggregation. Right now there's only one Keeper implemented that uses MongoDB and is written in Ruby.

### The MongoDB Ruby Keeper

Just as with the Ruby Watcher above, you will need Ruby 1.9.2+ with the following gems:

- json
- yajl
- mongo
- eventmachine
- sinatra



The Keeper, by default, connects to MongoDB in Single mode and writes to a database called grind. The Keeper's web API is accessible via the port 11146. For more info about Keepers, see [this article](#).

Running the Keeper is done by:

```
cd grind/keepers/mongo && ruby keeper.rb
```

# Configuring grind

## The core

By default the core configuration script is installed to `/etc/grind/config.lua` and it contains some initializing routines along with some settings you need to configure manually. The configuration part is outlined below (and it is in the beginning of that script):

```
-- This needs to point to the directory that contains grind.lua,  
-- by default it is in the scripts/ directory within the grind  
-- repository root.
```

```
local root = "/home/kandie/Workspace/Projects/grind/scripts"
```

```
local cfg = {  
  kernel = {  
    -- the ethernet interface to use for accepting feeders  
    feeder_interface = "127.0.0.1",  
    -- the ethernet interface to use for accepting watchers  
    watcher_interface = "127.0.0.1",  
    -- the port to use for watcher connections  
    watcher_port = "11142"  
  }  
}
```

Optionally, if the script couldn't be found in `/etc` then it's looked for in:

- `/usr/share/grind/config.lua`
- `/usr/local/share/grind/config.lua`

It can also be overridden by running `grind` with the `-c` option:

```
grind -c /path/to/config.lua
```

That's it, the rest of the configuration is the actual scripts you need to define for your application groups, classes, and views as explained in [this article](#) and in the [tutorials](#).

# Watchers

## The Ruby HTML5 Watcher

This watcher can be configured from the interface itself, or directly in the file called `config.json` found in the root directory where the other scripts lie (like `app.rb` or `server.rb`). Primarily, the watcher needs to know two pieces of information:

- the address & port of the grind instance to connect to (the `watcher_interface` and `watcher_port` explained in the core configuration section)
- the address & port of the WebSocket server (the `server.rb` script) which, by default, are set to `127.0.0.1` and port **8181**

# Keepers

## The MongoDB Ruby Keeper

Not yet configurable.

# Testing

## Simulating feeds

The process I follow for testing group, klass, and view definitions is outlined below:

I save the samples (or the needed subset of them) I'd like to test under `test/fixture/$group_name/$sample`, and then I use the script found in `test/unit/feed.lua` to relay the content within that file as a feed to grind.

### An example

For an application group labelled 'tutorial' that's bound to port 11150 and with the messages stored in `/tmp/tutorial.txt`:

```
cd /path/to/grind/test/unit
./feed.lua -i /tmp/tutorial.txt 11150
```

# Tutorial 1: Birth of the chimp

Hi, this is the introductory tutorial for [grind](#).

grind likes log messages, and it promises that it can do quite a lot with them, but only if you tell it a little about the data you're archiving.

In this tutorial, we will be using a hypothetical game as an example to learn how to configure grind and see what it can do with our data. Pendulum is the name of the game, and it's supposed to be a multiplayer deathmatch between two behemoths who control monkeys of all colors. The game client receives events from a server and plays them out, and we will be analyzing its [log file](#).

I suggest that you keep [this log file](#) open in a separate tab so you can check out the relevant part each section is talking about.

Ready? Let's get kicking!

## Receiving messages

Our first task is to relay the feed to grind. The feed is simply the stream of messages your applications are logging. We will begin by defining the group and reserving a port for it:

```
grind.define_group("pendulum", 11142)
```

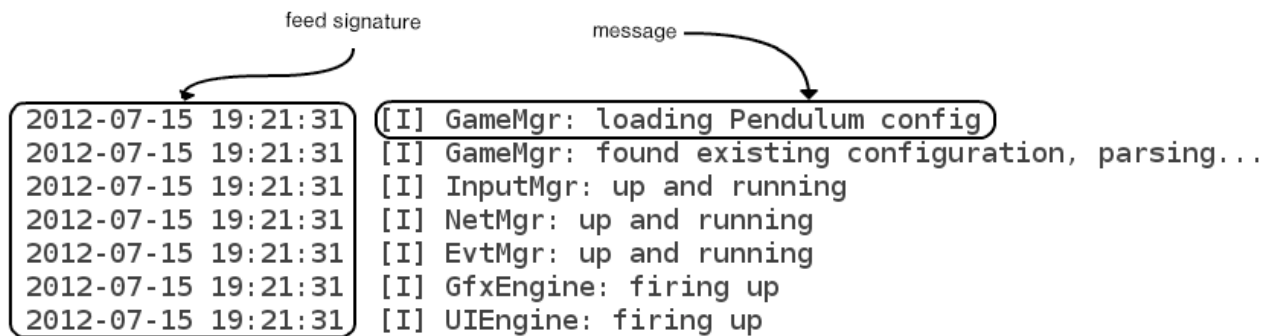
### Feeds and application groups

Each application group is mapped to a port where all the feeds it's interested in should be forwarded to. In this tutorial, we will be relaying a single feed manually using the helper explained [here](#).

## Configuring the feed

grind has no idea how the log messages look like, and more importantly, where they start and where they end. This is the most important step to get right for proper handling; we need to define a signature format [1] that can be used as a delimiter between log messages to tell where one starts and ends.

Looking at several messages we can see that the timestamp is always there. We can use that as our signature format:



The signature format is really just a regular expression that needs to match the unique part. Here's what it looks like for our game:

```
(\d{4}-\d{2}-\d{2}\s\d{2}:\d{2}:\d{2}) (view in PCREck)
```

Defining the signature in grind is as follows:

```
grind.define_signature("pendulum", [[(\d{4}-\d{2}-\d{2}\s\d{2}:\d{2}:\d{2})]])
```

We've also captured a piece of information from the signature: the timestamp. All captures from feed signatures are automatically passed to our class formatters as we will see later on.

With the group and its feed signature defined, grind will now be able to capture all the game messages on port 11142!

## Formatting data into something useful

Receiving messages properly is hardly an achievement, but it's a first step. Now that we have captured the log entries along with some metadata (the timestamp), let's do something with them. In the case of our game, we could be interested in a number of things:

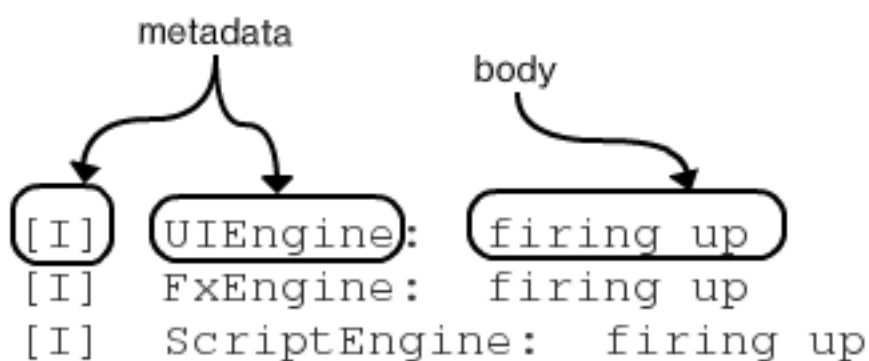
- what happened in a game session; were the engines and managers properly set up? were there any system errors?

- what happened in a battle; which units were spawned, who killed what, and who won?

Classifying data in such a way is what classes are for. Classes in grind (with a 'k' for implementation limitations) represent subsets of the log feed.

## Message formats

Message formats give you a chance to extract application-specific metadata like the logging context, module name, etc.:



When a matching format has been found for a message, eligible classes (and their views) will receive the metadata captured from both the signature and message formats along with the remainder of the message (from now on referred to as the body.)

## Classes & formats

A class must *bind to* one or more formats to indicate which messages it applies to. Keep in mind that this is different from the signature format; that was used merely to delimit messages and is applicable only at the application group level.

## Klass #1 - Sessions

Earlier in this section, we wanted to see all the messages related to a client game session. We begin by defining a format, calling it the `sessions` format:

```
grind.define_format("pendulum", "sessions", [==(\\[A-Z\\]) (\\w+):
(\\.*) ]==])
```

[Here](#) to view the expression in PCREck.

After defining the format, we need to write an extractor to map all the captured metadata to the entry object we'll be using in our classes and views:

```
grind.define_extractor("pendulum", "sessions",  
  -- notice how the timestamp is passed from the signature format  
  function(context, module, body, timestamp)  
    return { -- this is accessible using entry.meta  
      timestamp = timestamp,  
      context = context,  
      module = module  
    }, body -- and this using entry.body  
  end)
```

Nothing is now left except to actually define the class, which in this case is trivial:

```
grind.define_class("pendulum", { "sessions" }, "sessions")
```

The second argument to `grind.define_class` is a list of the message formats this class applies to. The third is the label of the class.



# Modeling the formatted data into views

Views in grind let you customize the fields that are shown at any one time. Since our format has only 4 fields, having multiple views is hardly useful. We will get to see the use of views in the next part of the tutorial.

For the `sessions` class, we will define a simple view that shows all the fields:

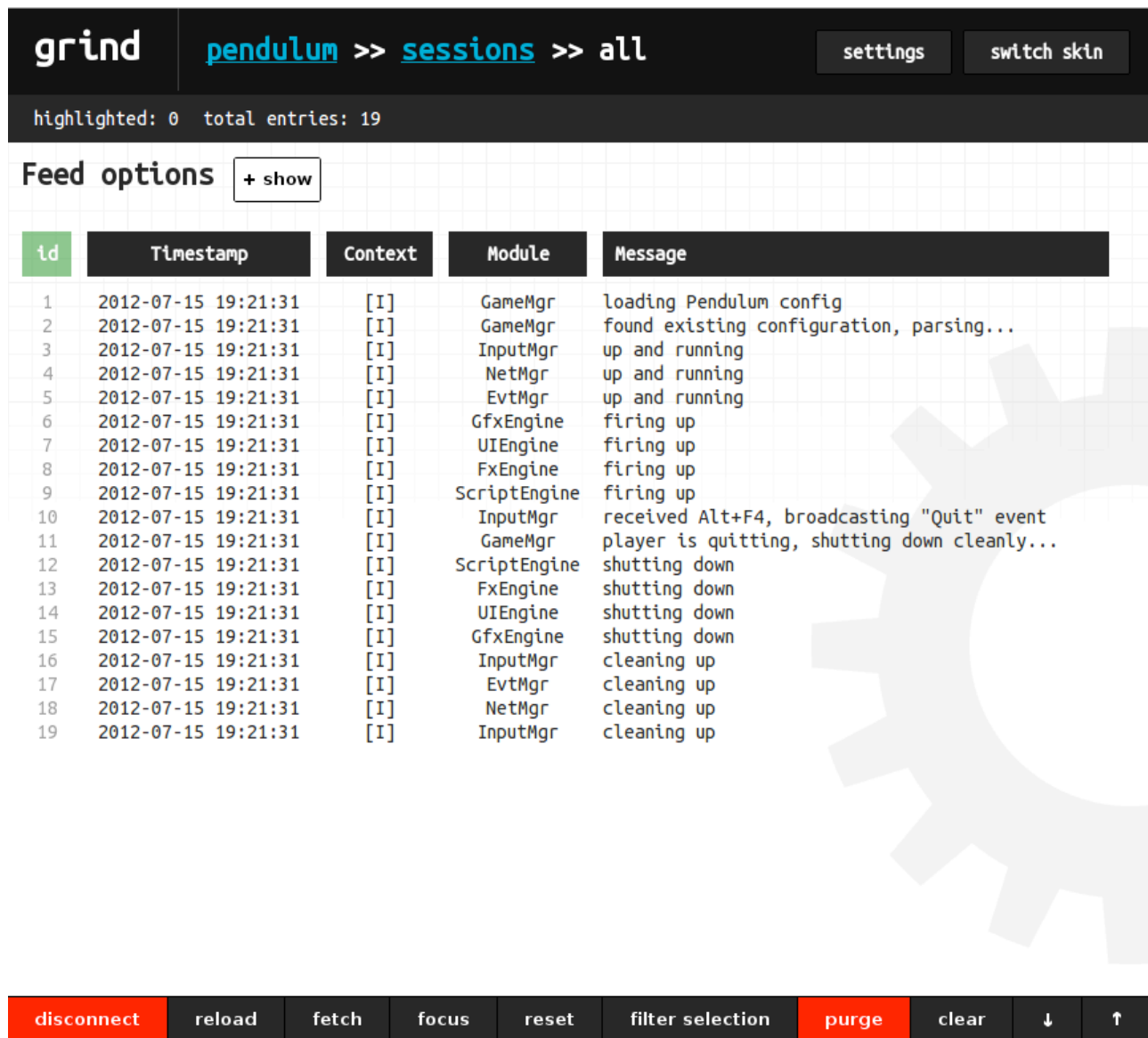
```
grind.define_view("pendulum", "sessions", "all",
  { "Timestamp", "Context", "Module", "Message" },
  function(fmt, ctx, entry)
    return true, {
      Timestamp = entry.meta.timestamp,
      Context = entry.meta.context,
      Module = entry.meta.module,
      Message = entry.body
    }
  end)
```

Here's an explanation of the arguments to `grind.define_view`:

1. the name of the application group
2. the name of the class this view belongs to
3. the view's label
4. a list of the fields (the structure) of the view's output
5. the output function which receives:
  1. the format that is used for the current entry
  2. a context variable (explained later)
  3. the formatted entry produced by the message format extractor

# Joining the party

Congratulations! You have now set up your first application group, its signature format, a message format that extracts the logging context, the module, and the message, a klass that does nothing, and a "raw" view. Let's take a look at the result:



The screenshot shows the 'grind' application interface. At the top, there's a header with 'grind' on the left, a breadcrumb path 'pendulum >> sessions >> all' in the center, and 'settings' and 'switch skin' buttons on the right. Below the header, it says 'highlighted: 0 total entries: 19'. The main content area is titled 'Feed options' with a '+ show' button. Below this is a table with 5 columns: 'id', 'Timestamp', 'Context', 'Module', and 'Message'. The table contains 19 rows of log data. At the bottom, there's a row of buttons: 'disconnect', 'reload', 'fetch', 'focus', 'reset', 'filter selection', 'purge', 'clear', and two arrow buttons (down and up).

id	Timestamp	Context	Module	Message
1	2012-07-15 19:21:31	[I]	GameMgr	loading Pendulum config
2	2012-07-15 19:21:31	[I]	GameMgr	found existing configuration, parsing...
3	2012-07-15 19:21:31	[I]	InputMgr	up and running
4	2012-07-15 19:21:31	[I]	NetMgr	up and running
5	2012-07-15 19:21:31	[I]	EvtMgr	up and running
6	2012-07-15 19:21:31	[I]	GfxEngine	firing up
7	2012-07-15 19:21:31	[I]	UIEngine	firing up
8	2012-07-15 19:21:31	[I]	FxEngine	firing up
9	2012-07-15 19:21:31	[I]	ScriptEngine	firing up
10	2012-07-15 19:21:31	[I]	InputMgr	received Alt+F4, broadcasting "Quit" event
11	2012-07-15 19:21:31	[I]	GameMgr	player is quitting, shutting down cleanly...
12	2012-07-15 19:21:31	[I]	ScriptEngine	shutting down
13	2012-07-15 19:21:31	[I]	FxEngine	shutting down
14	2012-07-15 19:21:31	[I]	UIEngine	shutting down
15	2012-07-15 19:21:31	[I]	GfxEngine	shutting down
16	2012-07-15 19:21:31	[I]	InputMgr	cleaning up
17	2012-07-15 19:21:31	[I]	EvtMgr	cleaning up
18	2012-07-15 19:21:31	[I]	NetMgr	cleaning up
19	2012-07-15 19:21:31	[I]	InputMgr	cleaning up

Even though the result is hot, it does seem like a lot of trouble for what it actually is... I mean, this is only a table representation of the raw data! Well, we had to walk through this so we can soon tap into the more powerful features grind offers.

In the [next tutorial](#), we will be analyzing our battlefield where, rumor has it, no monkey is safe.

[1]: The terms *signature format* and *feed signature* are used interchangeably throughout grind's documentation and they have similar meaning.

# Tutorial 2: Where no monkey is safe

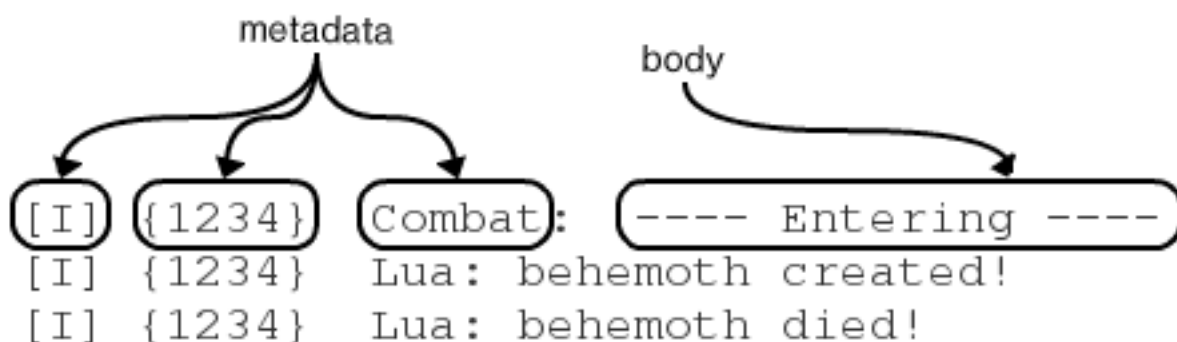
Hi again. In the [first part](#) of our tutorial we create a minimal view that returned what was captured by the klass format. In this part, we will be exploring the concept of vertical aggregation; formatting data doesn't have to be only horizontal, where it's done using a single entry, but can also be contextualized and span any number of entries!

What we will see as a result of this tutorial:

1. information about every monkey that was summoned during the battle; its name, ID, and owner.
2. all the events that took place during a battle and the things, or monkeys, it affected

## The format

If we take another look at the log file, we can see that all the entries start with a logging context, followed by a module and then a message.



Notice also that battle entries are all prefixed with a {1234} which is an id generated from the - hypothetical - game server. We will use these fields to build our format ([PCREck it](#)):

```
grind.define_format("pendulum", "battles", [==[(\[A-Z\])\n{(.*)} (\w+): (.*)]==])
```

And the extractor, as before:

```
grind.define_extractor("pendulum", "battles",
  function(context, battle_id, module, body, timestamp)
    return {
      timestamp = timestamp,
      context = context,
      battle_id = battle_id,
      module = module
    }, body
  end)
```

## The klass

The klass definition this time is a little different; we will be using an extra argument that was omitted before, the condition function. This function is invoked when a message has been matched by a format the klass is bound to, and it's expected to return true if it wants to handle the message, and false otherwise.

The condition function is passed the matching format as the first argument, and the so-far-aggregated entry which contains the metadata captured by the signature format and an `entry.meta.raw` field that contains the full raw message.

```
grind.define_klass("pendulum", { "battles" }, "battles",
  function(fmt, entry)
    return entry.meta.battle_id -- <= a condition
  end)
```

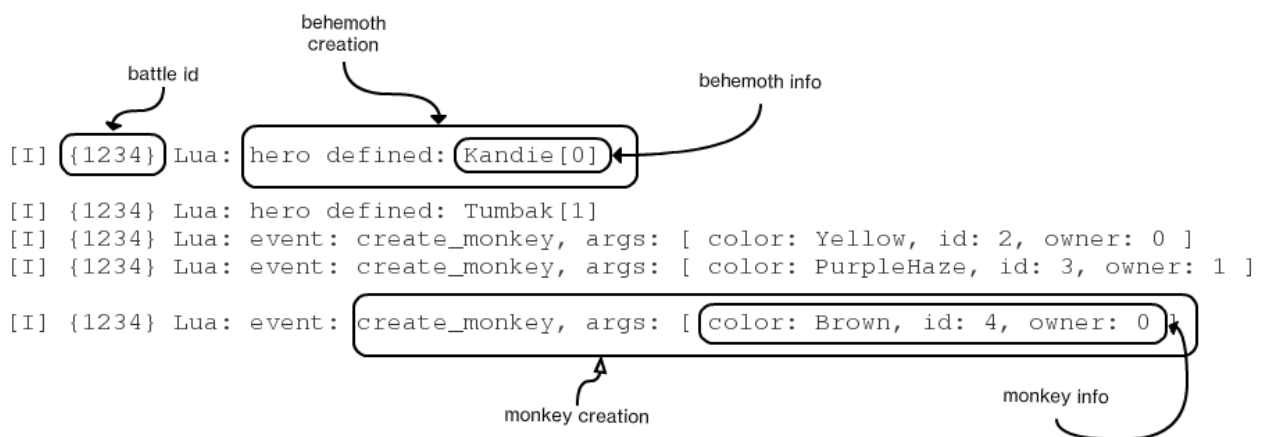
As you can see, we will only accept the entry if the `battle_id` flag is set, which is in truth not necessary because the format won't match if it wasn't there anyway. In later tutorials where we will be binding a klass to multiple formats, the condition function will be required.

# Modeling the data... Views revisited

The modeling stage this time is much more interesting. We will now get to the feature we mentioned in the beginning of the tutorial; vertical aggregation.

## View #1 - Monkeys

Take a look at the following:



We want to grab every monkey that has been summoned. The following entry body covers that:

```
event: create_monkey, args: [ name: Fetish, id: 2, owner: 0 ]
```

However, we also want to replace the owner field with the name of the behemoth, which was logged several entries earlier:

```
behemoth defined: Kandie[0]
```

The number between the `[\d]` is the id that maps to the owner field, obviously. Okay, let's first write the expressions to extract this data:

```
local extractors = {
  behemoth = create_regex([==[behemoth defined: (\w+)\[(\d+)\]]==]),
  monkey = create_regex([==[event: create_monkey, args: \[ name: (.*), id: (\d+), owner: (\d+) \]]==])
}
```

We will need to store the behemoth information somewhere, let's put them in this container:

```
local behemoths = {}
```

And finally, our view definition (inline comments will explain as required):

```
grind.define_view("pendulum", "battles", "monkeys",
  { "Battle", "Owner", "Monkey ID", "Color" },
  function(fmt, ctx, entry)

    -- for easy access
    local battle = entry.meta.battle_id

    -- is it a behemoth?
    local _,_,behemoth,id = extractors.behemoth:find(entry.body)
    if behemoth then
      -- track it so we can use its name when it creates a monkey:

      -- since behemoths are unique only in one battle,
      -- we must store them inside another collection identified
      -- by the battle id
      if not behemoths[battle] then
        behemoths[battle] = {}
      end

      behemoths[battle][id] = behemoth .. "[" .. id .. "]"

      return false -- nothing else to do with this entry
    end

    -- is it a monkey?
    local _,_,name,id,owner = extractors.monkey:find(entry.body)
    if name then
      return true, {
        Battle = entry.meta.battle_id,
        -- look up the behemoth we tracked earlier
        Owner = behemoths[battle][owner],
        -- we must surround the key by [""] since it has a space
      }
    end
  end
```

```

    ["Monkey ID"] = id,
    Color = name
  }
end

-- ok, it's neither a behemoth nor a monkey, return nothing!
return false
end)

```

Here's the result:

grind

pendulum >> battles >> monkeys

highlighted: 0 total entries: 3

Feed options + show

id	Battle	Owner	Monkey ID	Color
1	1234	Kandie[0]	2	Yellow
2	1234	Tumbak[1]	3	PurpleHaze
3	1234	Kandie[0]	4	Brown

[1] see the [regular expression matches](#) notes



# Appendix A: Some Lua to get you running with grind

This is by no means a complete tutorial to Lua, it's simply a listing of the common things you really should know about before attempting to configure grind.

Here's a list of good reference material to keep at hand:

1. [The Lua online book](#) - is the best reference you should use to get comfortable with the language
2. [Lua community tutorials](#) - a comprehensive set of tutorials to get you up to speed
3. [Unofficial Lua FAQ](#) - could prove an interesting read

## Lua variables and data-types

Lua supports integers, real numbers, strings, functions, and tables.

### Scoping of variables

When a variable declaration is prefixed with the keyword `local`, it can be used only within the scope it was declared in.

### Strings

Strings in Lua can be created by surrounding any word using `' '` or `" "`, they're equivalent. The length of a string can be retrieved using the `#` operator:

```
print(#"Hello") -- 5
```

Strings can be concatenated using the `..` operator:

```
print("foo" .. "bar") -- foobar
```

Be sure to check out this [Lua string tutorial](#) for a complete reference.

### Functions

[github.com/amireh/grind](https://github.com/amireh/grind)

Functions in Lua are first-class citizens:

From Wikipedia: a first-class citizen is an entity that can be constructed at run-time, passed as a parameter, returned from a subroutine, or assigned into a variable.

Functions can return multiple values:

```
function the_truth()  
    return "hello", 42, "world"  
end
```

## Tables

Tables in Lua can be used as lists and arrays:

```
local my_list = { 1, 2, "a", "b" } -- indexes start at 1  
print(my_list[1]) -- 1  
print(my_list[3]) -- a
```

or as Python-style key-value dictionaries:

```
local my_dict = { a = 1, b = { 150.5 }, c = function()  
print( "weeew" ) end }  
print(my_dict.a) -- 1  
print(my_dict['a']) -- 1  
print(my_dict.b[1]) -- 150.5  
my_dict.c() -- weeew
```

Tables are very flexible and can contain all the types mentioned earlier in either key or value!

To iterate through the values of a table:

```
for k,v in pairs( { 10,20,30 } ) do  
    print(k .. " => " .. v) -- 1 => 10, 2 => 20, 3 => 30  
end
```

```
for k,v in pairs( { a = "100", b = 300 } ) do  
    print(k .. " => " .. v) -- a => 100, b => 300
```

**end**

If you want to store a key that is a string containing a space, you must use the `[ " " ]` notation:

```
local t = { [ "My Key" ] = 10, a = 5 }
```

# Appendix B: grind internals

This article explains some technical and design decisions taken when grind was written, as well as some limitations that should be known about by people who plan on extending or modifying the core.

## Composite signature formats

While it's not apparent from the grind configuration interface, when defining more than one feed signature for a certain group, they are internally converted to a single, composite PCRE regular expression using the branch resetting feature `((?|))`. Why this might be important to know is that by using that feature, a regular expression is no longer allowed to have different named subpatterns at similar locations. For example, if you were specifying a named capture for a timestamp as follows:

```
(<timestamp>\d{2}:\d{2}:\d{2})
```

you can not define another signature to capture a host then a timestamp, like this:

```
(<host>[\w|\.|.]+) (<timestamp>\d{2}:\d{2}:\d{2})
```

because of the name conflict at subpattern location 1.

## Transport format

grind's output is serialized using JSON (with UTF-8 character encoding) internally between the core and Watchers or Keepers.

While using JSON allows for flexibility in how the data is handled, it has a heavy performance hit. In the future, grind might migrate to some form of binary encoding.

# Labels are unique

Entities in grind, such as application groups, classes, and views, are all unique by their label within their context. So, only one group can be called "converter", and it can have only one class called "errors", but it's okay for another group, "eradicators" to also have a class called "errors".