Hive v0.8.3 - User Guide

Kajetan Rzepecki

August 20, 2014

Contents

1 Dependencies

The following section lists Hive's dependencies.

1.1 Erlang/OTP

Hive is known to build and run under **Erlang R16B** in the following setup:

```
$ erl
Erlang R16B (erts-5.10.1) [source-05f1189] [64-bit] [smp:2:2] [async-threads:10]
  [hipe] [kernel-poll:true]
  Eshell V5.10.1
```

Furthermore, Hive uses **escript** for scripting, so an instance has to be provided (it is usually boundled together with Erlang/OTP).

1.2 Build tools

- **rebar** - Hive uses Rebar build tool for managing library dependencies and the building process in general. A version of **2.0.0** is required.
- **make** - for convenience a **Makefile** build script is provided that wraps Rebar. Any **make** will do.
- **git** - rebar uses **git** to fetch library dependencies from GitHub, so **git** in a recent-ish version is required.

1.3 3rd party libraries

The libraries, listed belowe, are downloaded automatically by Rebar via **git**, so no special set-up other than **Rebar and git** is required.

- **Lager** - an easy to use logging library found here. Hive uses version **2.0.0**.
- **Cowboy** - a web server optimized for speed. Hive requires Cowboy **0.8.6**.
- **Ranch** - a TCP pool pulled in by Cowboy from here, Hive requires version **0.8.4**.
- **JSONx** - JSON parser optimized for speed, found here. There is **no stable version** as of $August 20, 2014$; Hive is known to build and run on commit **a8381a34d126a93eded62c248989dd0529cd257d**, so in case of any problems some fiddling might be required.
- **Jesse** - JSON schema validator, found here. Hive is know to build and run using commit **6bd-fce3e988239a8878d7bd19877d7e3d46ae7d7**.
- **Ibrowse** - an HTTP client; Hive uses version **v4.0.2**.

- **Folsom** - a monitoring tool; Hive requires version **0.7.4**.
- **Poolboy** - worker pool management library found here. Hive requires version **1.0.0**.
- **ERedis** - a Redis connector, Hive requires version **v1.0.5**.

2 Installation & Running

The following section describes how to build & run the Hive server.

2.1 Obtaining Hive

Use **git** to clone the repository:

```
$ git clone https://host:port/path/to/hive.git hive
```

After the checkout you will be greeted with the following repository layout:

```
$ tree hive
  hive
  |-- docs                 - Documentation files.
  |   '-- ...
  |-- include              - Erlang header files.
  |   '-- ...
  |-- etc                  - Default configuration files.
  |   |-- schema           - Config validation schema files.
  |   '-- ...
  |-- Makefile             - Build script for make.
  |-- plugins              - Hive plugins directory.
  |   '-- ...
  |-- priv                 - Additional scripts directory.
  |   '-- ...
  |-- rebar                - Rebar build tool.
  |-- rebar.config         - Rebar config (lists dependencies).
  |-- src                  - Erlang source files.
  |   '-- ...
  '-- test                 - Erlang unit-tests.
      '-- ...
```

2.2 Building Hive

Simply type **make** in the root of the repository and the supplied **Makefile** script will handle all of the build process for you:

```
$ make
```

Building Hive for the first time takes a considerable amount of time as Rebar downloads and compiles all of the dependencies, go grab a coffee! Other build script options:

- Building a Debian `.deb` package for easy distribution:

```
$ make deb-package                                 - Builds using default options.
$ make deb-package TARGET="target --and --options" - Used to pass additional build options.
```

- Cleaning up the repository - this will clean up all temporary and compiled files:

```
$ make clean
```

- Unit-testing - this will run **EUnit** testing framework and dump results into the **.eunit** directory:

```
$ make unit-test
```

- Testing Hive configuration - this will test the Hive configuration file:

```
$ make test-config                            - Tests the default configuration file.
$ make test-config CONFIG=path/to/config.json - Tests CONFIG file.
```

- Running Hive - this will run Hive as described in the next subsection:

```
$ make run                                - Runs in production mode.
$ make run CONFIG=path/to/config.json     - Runs in production mode using CONFIG.
$ make run-dev                            - Runs in development mode.
$ make run-dev CONFIG=path/to/config.json - Runs in development mode using CONFIG.
```

2.3 Running Hive

Before running Hive make sure that the supplied configuration file is valid and loads properly by invoking:

```
$ make test-config CONFIG=path/to/config.json
```

If the configuration file loads properly you can attempt to run Hive. For convenience there's a **make** rule for that defined in the Makefile:

```
$ make run CONFIG=path/to/config.json
```

After a quick boot-up you will be greeted by a log similar to this one (note that the order of the log lines might be different at each execution as Hive consists of multiple processes running in parallel):

```
18:21:02.724 [notice] Starting Hive...
18:21:02.734 [notice] Starting Hive Top-level Supervisor...
18:21:02.740 [notice] Starting Hive Environment Supervisor...
18:21:02.769 [notice] Starting Hive Monitor...
18:21:02.780 [notice] Hive Monitor started!
18:21:02.785 [notice] Starting Hive Plugins Manager...
18:21:02.791 [notice] Loading Hive Plugin: plugin_1
...
18:21:02.852 [notice] Hive Plugins Manager started!
18:21:02.860 [notice] Starting Hive Config Manager...
18:21:02.886 [notice] Hive Config Manager started!
18:21:02.887 [notice] Starting Hive Cluster Manager...
18:21:02.887 [notice] Hive Cluster Manager started!
18:21:02.889 [notice] Hive Environment Supervisor started!
18:21:02.894 [notice] Starting Hive Module Supervisor...
18:21:02.918 [notice] Starting Hive Connectors Supervisor...
18:21:02.925 [notice] Starting Hive Connectors Manager...
18:21:02.925 [notice] Hive Connectors Manager started!
18:21:02.926 [notice] Hive Connectors Supervisor started!
18:21:02.932 [notice] Starting Hive Connectors Pool Supervisor...
18:21:02.932 [notice] Hive Connectors Pool Supervisor started!
18:21:02.941 [notice] Starting Hive Router Supervisor...
18:21:02.945 [notice] Starting Hive Connector pool_1...
...
18:21:02.959 [notice] Starting Hive Router...
18:21:02.959 [notice] Hive Router started!
18:21:02.959 [notice] Hive Router Supervisor started!
18:21:02.969 [notice] Starting Hive Clients Supervisor...
18:21:02.969 [notice] Hive Clients Supervisor started!
18:21:02.974 [notice] Starting Hive Pub-Sub Supervisor...
18:21:02.996 [notice] Starting Hive Pub-Sub...
18:21:02.997 [notice] Hive Pub-Sub started!
18:21:02.998 [notice] Hive Pub-Sub Supervisor started!
18:21:03.008 [notice] Starting Hive Pub-Sub Channel Supervisor...
18:21:03.008 [notice] Hive Pub-Sub Channel Supervisor started!
18:21:03.014 [notice] Starting Hive Server Supervisor...
18:21:03.020 [notice] Starting Hive Monitor Server...
18:21:03.041 [notice] Hive Monitor Server started!
18:21:03.046 [notice] Starting Hive API Server...
18:21:03.058 [notice] Hive API Server started!
18:21:03.070 [notice] Starting Hive Server...
18:21:03.085 [notice] Hive Server started!
18:21:03.086 [notice] Hive Server Supervisor started!
18:21:03.086 [notice] Hive Module Supervisor started!
18:21:03.089 [notice] Hive Top-level Supervisor started!
18:21:03.089 [notice] Hive started!
```

. . . and the supervision tree will look something like this:

```
hive_root_sup                           - Top-level Hive supervisor.
|-- hive_env_sup                        - Hive environment supervisor.
|   |-- hive_config                     - Hive Config Manager.
|   |-- hive_monitor                    - Hive Monitor Manager.
|   |-- hive_plugins                    - Hive Plugins Manager.
|   '-- hive_cluster                    - Hive Cluster Manager.
'-- hive_sup                            - Hive Modules supervisor.
    |-- hive_connectors_sup             - Hive Connectors supervisor.
    |   |-- hive_connectors             - Hive Connectors Manager.
    |   '-- hive_connectors_pool_sup - Hive Connectors pool supervisor.
    |        |-- connector1             - Various Connectors.
    |        |    '-- ...               - Connector pool workers.
    |        '-- ...
    |-- hive_pubsub_sup                 - Pub-Sub supervisor.
    |   |-- hive_pubsub_channel_sup     - Pub-Sub channel supervisor.
    |   |   '-- ...                     - Various Pub-Sub channels.
    |   '-- hive_pubsub                 - Pub-Sub manager.
    |-- hive_router_sup                 - Router supervisor.
    |   |-- hive_client_sup             - Clients supervisor.
    |   |   '-- ...                     - Client-related processes.
    |   '-- hive_router                 - Hive Router.
    '-- hive_web_sup                    - Hive Server & Monitor supervisor.
        |-- hive_api                    - The main Hive API Server entry point.
        |-- hive_monitor_server         - The Hive Monitor Server entry point.
        '-- hive_web                    - The main Hive Server entry point.
```

2.4 Stopping Hive

The Hive server employs a **graceful termination** strategy - after requesting a server termination Clients are asked *nicely* to close their connections and after a certain timeout Hive forces their termination by killing them. No new connections are accepted during the graceful termination period.
The configuration parameters responsible for the graceful termination behaviour are described here.
The graceful termination RESTful API is described here.

2.5 Repository layout reference

```
$ tree hive
  hive
  |-- deps                    - Dependencies directory.
  |   '-- ...
  |-- docs                    - Documentation files.
  |   |-- docs.pdf            - YOU ARE HERE.
  |   '-- ...
  |-- ebin                    - Compiled Hive source files.
  |   '-- ...
  |-- etc                     - Hive configuration files.
  |   |-- hive.json           - The default configuration file.
  |   |-- schema              - Hive config validation files.
  |   '-- ...
  |-- include                 - Hive Erlang include files.
  |   '-- ...
  |-- log
  |   '-- hive
  |       |-- console.log     - Hive console log (similar to the one in console).
  |       |-- crash.log       - Hive crash log (only crash reports).
  |       '-- error.log       - Hive error log (only error messages).
  |-- Makefile                - Make rules file.
  |-- plugins                 - Hive plugins directory (source code).
  |   '-- ...
  |-- priv                    - Directory containing additional stuff.
  |   |-- start-dev.sh        - Hive running script (dev version).
```

```
|   |-- start.sh           - Hive running script.
|   |-- prep_hive.erl      - A script that prepares Hives execution environment.
|   |-- test_config.erl    - Hive configuration testing script.
|   |-- tsung_hive.xml     - A config file for Tsung.
|   '-- ...
|-- rebar                  - Rebar build tool.
|-- rebar.config           - Rebar config (lists dependencies).
|-- src                    - Hive Erlang source files.
|   |-- hive_client*       - Hive generic client code.
|   |-- hive_websocket*    - Hive WebSocket client related code.
|   |-- hive_xhr_polling*  - Hive XHR-polling client related code.
|   |---hive_socketio*     - Hive Socket.IO related code.
|   |-- hive_connectors*   - Hive Connectors manager.
|   |-- hive_*_connector*  - Various connectors.
|   |-- hive_*_client*     - Hive Client handler related code.
|   |-- hive_hooks*        - Hive Hooks related code.
|   |-- hive_events*       - Hive Internal Events related code.
|   |-- hive_monitor*      - Hive Monitor related code.
|   |-- hive_pubsub*       - Hive Pub-Sub related code.
|   |-- hive_router*       - Hive Router related code.
|   |-- hive_config*       - Hive Config related code.
|   |-- hive_cluster*      - Hive Cluster Manager related code.
|   |-- hive_api*          - Hive RESTful API related code.
|   |-- hive_*_sup*        - Various Hive supervisors.
|   '-- ...
'-- test                   - Hive unit-tests.
    '-- ...
```

3 Configuration & Tweaking

The following section lists and describes various configuration parameters and their purpose. The configuration file uses **JSON** format that is later transformed into Erlang terms. It shouldn't matter, but keep this in mind in case something weird starts to happen **wink, wink**. The default configuration file is **etc/hive.json**.
The configuration file is divided into several sections, each of which controls a different part of the Hive server. Section ordering in the configuration file does not matter, neither does parameter ordering mid-section. The general layout of the file is as follows:

```
{
    "sectionA" : {
        "parameterA" : "valueA",
        "parameterB" : "valueB",
        ...
    },
    "sectionB" : {
        "parameterB" : "valueB",
        "parameterC" : "valueC",
        ...
    },
    ...
}
```

At the moment, there are several sections recognized by Hive and all of them are required. The sections are:

```
{
    "hive" : {
        // Contains general Hive related parameters.
    },
    "socketio" : {
        // Contains Socket.IO protocol related parameters.
    },
    "clients" : {
        // Contains Client FSM & Hooks related parameters.
    },
    "connectors" : {
        // Contains parameters controlling various Hive Connectors.
    },
    "pubsub" : {
        // Contains parameters controlling Hive Pub-Sub channels.
    },
    "api" : {
        // Contains parameters controlling the Hive API Server.
    },
    "monitor" : {
        // Contains parameters controlling the Hive Monitor.
    },
    "log" : {
        // Contains logging related parameters.
    }
}
```

3.1 Configuration parameters

Each of the configuration file sections, their parameters (some of which are optional) and accepted values are described in the following subsections. By convention each description will use the full qualified parameter name, for example:

```
sectionA.parameterB
sectionB.parameterB
```

3.1.1 hive

This subsection describes the **hive** part of the configuration file and corresponds to the general configuration of the Hive server - it contains stuff that didn't really fit elsewhere.
Required parameters:

- `hive.acceptors` - the number of HTTP acceptors that receive and prepare HTTP connections. It is in no way related to the maximum number of connections. This has to be **a positive integer**.
- `hive.port` - the port on which Hive will listen for incoming connections. It has to be **a non-negative integer lower than 65536**.
- `hive.allowed_origins` - a JavaScript array of the origins that are allowed to access Hive services. Has to be **an array of strings**, each of which names a single origin - **a URL** (possibly with wildcards) or the special value `null` (equivalent to an undefined origin). Some examples:

```
null, "null", "*.*", "http://zadane.pl"
```

- `hive.graceful_termination_timeout` - the time (**in milliseconds**) after which Client Workers will be forced to terminate on server termination.
- `hive.max_client_slots` - the maximal number of clients that can be connected to Hive. It has to be **a positive integer**.

Additional, optional parameters:

- `hive.name` - a string representing the name of this Hive instance, defaults to unqualified "hive". It is described in detail in a later section.
- `hive.cluster_name` - a string representing the name of the Hive cluster formed by this Hive instance, defaults to "hive$_{\text{cluster}}$".
- `hive.cluster_nodes` - a JavaScript array of node names to which this instance will attept to connect. Every element of this array should be a fully qualified `hive.name` of another Hive instance.
- `hive.cluster_port_min` - an integer in range (0, 65535) representing the lower bound of the range of ports used to maintain Hive cluster connections.
- `hive.cluster_port_max` - an integer in range (0, 65535) representing the upper bound of the range of ports used to maintain Hive cluster connections.
- `hive.direct_websocket` - **a boolean** flag determining whether WebSocket-based clients should skip the Hive Router in order to speed up the client-server communication. Skipping the router involves a trade off in that the router won't be able to manage clients communication (so no Socket.IO event related router logs, possible message counts inconsistencies, etc).
- `hive.websocket_ping_timeout` - the timeout (**in milliseconds**) used by the WebSocket connection handlers to determine whether they are still active; if there is no pong message received from the client for this amount of time, its Client worker will be terminated. It has to be **a positive integer**.
- `hive.max_processes` - the maximal number of processes that can exist in the Erlang VM. It has to be **a positive integer**. Keep in mind that the actual maximal number of processes might be higher (courtesy of the Erlang VM).

Example values:

```
"hive" : {
    "name" : "hive@zadane.pl",
    "cluster_name" : "hive_server_cluster",
    "cluster_port_min" : 1234,
    "cluster_port_max" : 2345,

    "acceptors" : 1000,
    "port" : 8080,
    "allowed_origins" : ["http://zadane.pl"],
```

```
    "max_client_slots" : 1000,
    "max_processes" : 15000,
    "graceful_termination_timeout" : 10000,

    "direct_websocket" : true,
    "websocket_ping_timeout" : 500
}
```

3.1.2 socketio

This subsection describes the **socketio** part of the configuration file. It is used to tweak the underlying **Socket.IO protocol**.
Required parameters:

- socketio.heartbeat_timeout - the heartbeat timeout **in milliseconds** used by the server (the client receives around 110% of this value). It has to be **a positive integer greater than or equal to 1000**.
- socketio.reconnect_timeout - the reconnection timeout **in milliseconds**, currently not used. It has to be **a positive integer greater than or equal to 1000**.
- socketio.poll_timeout - the polling timeout **in milliseconds**, used by the server to bound message polling times that happens before sending a reply to the client. Has to be **a positive integer**.
- socketio.init_timeout - the initialization timeout **in milliseconds**, it is started after the Socket.IO handshake in order to make sure that uninitialized clients don't clog the memory for all of eternity and beyond. Again, has to be **a positive integer**.
- socketio.session_timeout - the session timeout **in milliseconds**; if there are no messages received from the client for this amount of time, its connection will be terminated. It has to be **a positive integer**.
- socketio.transports - lists the available transport protocols for use by the Socket.IO protocol. Has to be **a JavaScript array of strings**, each of which names a single transport. Currently supported transports:

```
"xhr-polling", "websocket", "flashsocket"
```

Example values:

```
"socketio" : {
    "heartbeat_timeout" : 30000,
    "reconnect_timeout" : 120000,
    "poll_timeout" : 500,
    "init_timeout" : 5000,
    "session_timeout" : 120000,
    "transports" : ["websocket", "xhr-polling"]
}
```

3.1.3 clients

This subsection describes the **clients** part of the configuration file and corresponds to the general configuration of the Hive Client FSMs.
Required parameters:

- clients.state - the descriptor of a State Manager to be used by the Client processes. It has to be **a JSON object** containing exactly three fields: state_manager, initial_value and args. State Management and available State Managers are described in greater detail in a later section.

- clients.actions - **a JSON object** listing various **internal event** dispatchers recognized by the Client Workers on a **per-action** basis, that is, each field names an action will trigger Hive Internal Event dispatchers listed in this field's value. The value has to be **a JavaScript array** of Hive Internal Event dispatcher descriptors, each of which has to be **a JSON object** that defines several fields:
  - action - the name of the Hive Internal Event dispatcher to run when triggered. Available Hive Internal Event dispatchers are described in greated detail in a later section.

– `args` - the arguments passed to the Hive Internal Event dispatcher when it is run. Accepted argument descriptions are found in a later section.

- `clients.hooks` - **a JSON object** containing various Hive Hook definitions on a **per-event** basis, that is, each field of the JSON object names an event that will trigger Hive Hooks listed in this field's value. The value has to be **a JavaScript array** of Hive Hook descriptors, each of which has to be **a JSON object** that defines several fields:
  – `hook` - the name of the Hive Hook to run when triggered. Available Hive Hooks are described in greated detail in a later section.
  – `args` - the arguments passed to the Hive Hook when it is run. Accepted arguments descriptions are found in a later section.

Example values:

```
"clients" : {
    "state" : {
        "state_manager" : "sm.redis",
        "initial_value" : [1, 2, 3],
        "args" : {
            "connector" : "database",
            "expiration_timeout" : 60000
        }
    },
    "actions" : {
        "action_1" : [
            {
                "action" : "action.store",
                "args" : null
            },
            ...
        ],
        ...
    },
    "hooks" : {
        "event_1" : [
            {
                "hook" : "utils.echo",
                "args" : null
            },
            ...
        ],
        ...
    }
}
```

3.1.4 `connectors`

This subsection describes the **connectors** part of the configuration file. It is used to control various Hive Connectors.
Required parameters:

- `connectors.rent_timeout` - the timeout **in milliseconds** used when waiting for an available worker in a given connectors pool. If there are no available workers in a pool, the renting process will wait for at most this much time. It has to be **a positive integer**.

- `connectors.pools` - **a JSON object** of **name/pool descriptor** pairs - pairs of **strings** representing pool names and **JSON objects** representing pools themselves. Each pool descriptor has to define several required parameters:
  – `size` - the base size of the pool; **a non-negative integer**.

  - `overflow` - the maximum number of additional workers that are created under heavy server load (the total number of available workers is therefore `size` + `overflow`). It has to be **a non-negative integer**.
  - `args` - the arguments that will be passed to the connector workers at initialization. The accepted values depend heavily on the type of the connector, and will be described later in an appropriate section.
  - `connector` - the name of the connector plugin to use for this pool. It has to be **a string**. The available Connector pools will be described in a later section.

Optional parameters:

- `connectors.backoff_num` - the number of times a client will try various Hive Connectors requests in case of an error. Has to be **a non-negative integer** greater than zero. Defaults to 1.
- `connectors.backoff_time` - the maximal backoff timeout in **milliseconds**. Has to be **a non-negative integer**. The concrete backoff timeout will be uniformly selected in the range [0, `connectors.backoff_time`]. Defaults to 0.

Example values:

```
"connectors" : {
    "rent_timeout" : 5000,
    "backoff_num" : 3,
    "backoff_time" : 1000,

    "pools" : {
        "database" : {
            "connector" : "connector.redis",
            "size" : 100,
            "overflow" : 50,
            "args": {
                "host" : "127.0.0.1",
                "port" : 6379,
                "database" : 0,
                "password" : "",
                "reconnect_timeout" : 100
            }
        }
    }
}
```

3.1.5 pubsub

This subsection describes the **pubsub** portion of the configuration file. It is used to control the Hive Pub-Sub facitily. More about the Hive Pub-Sub can be found in a later section.
Required parameters:

- `pubsub.channels` - **a JSON object** containing various Hive Pub-Sub channel definitions on **a per-prefix** basis, that is, each field of the JSON object names a Pub-Sub channel prefix which might later generate concrete Pub-Sub channels. Each **channel descriptor** has to be **a JSON object** that defines several fields:
  - `timeout` - the time (**in milliseconds**) after which the channel will cease to exist if there are no more Client workers subscribed to it. It has to be **a non-negative integer**. A timeout of **0 means infinity (obviously)** - a permanent channel that never ceases to exist.
  - `privilege` - the access type of a channel, specifies the privilege level required to operate (subscribe or unsubscribe) on a channel. It has to be **a string**. Accepted values:

```
"private", "public"
```

Example values:

```
"pubsub" : {
    "channels" : {
        "foo" : {
            "privilege" : "public",
            "timeout" : 1000
        },

        "bar" : {
            "privilege" : "private",
            "timeout" : 0
        }
    }
}
```

3.1.6 `api`

This subsection describes the **api** part of the configuration file. Parameters described in this section control the behaviour of the Hive API Server.
Required parameters:

- `api.acceptors` - similar to `server.acceptors`, names the number of HTTP listeners that accept new HTTP connections. Has to be **a positive integer**.
- `api.port` - the port used by the API Server to listen for HTTP connections. Has to be **a non-negative integer that is lower than 65536**.
- `api.hash` - an API key used to secure the Hive API Server accesses. It has to be **a string of length 8 or more**.

Example values:

```
"api" : {
    "acceptors" : 100,
    "port" : 1235,
    "hash" : "abcde12345"
}
```

3.1.7 `monitor`

This subsection describes the **monitor** part of the configuration file. It controls the behaviour of the Hive Monitor.
Required parameters:

- `monitor.acceptors` - similarily to `server.acceptors` names the number of HTTP listeners that accept new HTTP connections. Has to be **a positive integer**.
- `monitor.port` - the port used by the Monitor to listen for HTTP connections. Has to be **a non-negative integer that is lower than 65536**.
- `monitor.hash` - an API key used to secure the Hive Monitor accesses. It has to be **a string of length 8 or more**.

Example values:

```
"monitor" : {
    "acceptors" : 100,
    "port" : 1234,
    "hash" : "12345abcde"
}
```

3.1.8 `log`

This subsection describes the **log** part of the configuration file. It controls the behaviour of the logger. Required parameters:

- `log.dir` - the directory name for log files and crash dumps to reside on the hard drive. Has to be **a string naming a valid file system location** (the directory does not have to exist).

Additional, optional parameters:

- `log.file_level` - the minimal log-level of messages dumped to the `log.file` ("none" turns off any file logging). Has to be any of the following:

```
"debug", "info", "notice", "warning", "error", "critical", "alert", "emergency", "none"

// Additional filter specifies:
"info"     // Info and higher (>= is implicit).
"=debug"   // Only the debug level.
"!=info"   // Everything but the info level.
"<=notice" // Notice and below.
"<warning" // Anything less than warning.
```

- `log.console_level` - the minimal log-level of messages dumped to the console ("none" turns off any console logging). Has to be any of the following:

```
"debug", "info", "notice", "warning", "error", "critical", "alert", "emergency" "none"

// Additional filter specifies:
"info"     // Info and higher (>= is implicit).
"=debug"   // Only the debug level.
"!=info"   // Everything but the info level.
"<=notice" // Notice and below.
"<warning" // Anything less than warning.
```

Example values:

```
"log" : {
    "dir" : "log/",
    "console_level" : "debug",
    "file_level" : "info"
}
```

3.2 Configuration validation

The configuration file can be validated using a supplied configuration testing script by invoking the following command in the root directory of the repository:

```
$ make testconfig CONFIG=path/to/config.json
```

Hive uses **JSON Schema** to validate its configuration files. The schema files used by the Hive Config validator can be found in **etc/schema** directory in the root repository. Each configuration sections' schema resides in a separate file named **section.jsonschema**. When defining new configuration parameters it is essential to include them in the validation schema.

3.3 Organizing the configuration

For convenience each configuration section can be stored in its own, separate file. If that is the case, the main configuration file has to list the file-name where a section configuration can be found relative to the main config file. For example:

```
{
    "hive" : {
        // Some setup...
    },
```

```
    "socketio" : "path/to/socketio.json",

    "clients" : "path/to/clients.json",

    "connectors" : {
        // Some setup...
    },
    ...
}
```

4 Monitoring & Statistics

This section describes the monitoring facilities of the Hive server. Statistics gathering is performed by changing values of various in-memory counters during Hive run-time, using Folsom external library. Most of the counters are represented by **non-negative integers**, and each of them corresponds to a specific metric of the Hive server.

4.1 Statistics structure

There are many different metrics which are grouped into several sections, and subsections. The structure of the statistics sections is shown below:

```
{
    "hive" : {
        // Various general metrics.
        "memory" : {
            // Memory usage related metrics.
        },
        "router" : {
            // Hive Router related metrics.
        },
        "cluster" : {
            // Hive Cluster related metrics.
        }
    },
    "clients" : {
        // Various general, client related metrics.
        "state_mgr" : {
            // Client State Manager related metrics.
        },
        "states" : {
            // Client FSM state related metrics.
        },
        "events" : {
            // Dispatched Socket.IO/internal events related metrics.
        },
        "transports" : {
            // Socket.IO transports related metrics.
        },
        "hooks" : {
            // Hive Hooks related metrics.
            "event1" : {
                // Hive Hooks related metrics (per event name).
            },
            ...
        }
    },
    "connectors" : {
        // Various Hive Connectors related metrics.
        "http" : {
            "connector1" : {
                // HTTP Connectors related metrics (per pool name).
            },
            ...
        },
        "redis" : {
            // Redis Connectors related metrics (per pool name).
            ...
        },
        "tcp" : {
            // TCP Connectors related metrics (per pool name).
```

```
            ...
        },
        ...
    },
    "pubsub" : {
        // Hive Pub-Sub related metrics.
        "channel_prefix1" : {
            // Various channel prefix related metrics.
        },
        ...
    },
    "api" : {
        // Hive API Server related metrics.
    },
    ...
}
```

4.2 Statistics metrics

Each section and its metrics are described in the following subsections. By convention each description will use the full qualified metric name, for example:

```
hive.memory.total
connectors.redis.pool_name_2.errors
```

4.2.1 `hive`

This subsection describes the **hive** portion of the Hive statistics. Metrics found in this subsection measure various general quantities that didn't fit elsewhere:

- `hive.uptime` - the uptime (**in milliseconds**) of the Hive.
- `hive.errors` - the number of **critical** (mostly supervision tree related) Hive errors encountered; it **does not** include errors from other sections.
- `hive.total_processes` - the number of Erlang processes currently executing in the VM.
- `hive.plugins` - the number of currently loaded Hive Plugins.
- `hive.plugin_errors` the number of Hive Plugin related errors.
- `hive.config_errors` the number of Hive Manager errors.

4.2.2 `hive.memory`

This subsection describes the **hive.memory** portion of the Hive statistics. Metrics found in this subsection measure the memory usage of the Hive Server:

- `hive.memory.total` - the total amount of memory used by the Erlang VM.
- `hive.memory.processes` - the amount of memory used by the Erlang processes.
- `hive.memory.system` - the amount of memory not directly related to any Erlang process. It includes `atom`, `binary`, `code` and `ets` values.
- `hive.memory.atom` - the amount of memory used by the Erlang Atom table. This metric **does not decrease** as Atoms are not garbage collected by the Erlang VM.
- `hive.memory.binary` - the amount of memory used by the Erlang VM to share binary data between the processes.
- `hive.memory.code` - the amount of memory used by the loaded Erlang code.
- `hive.memory.ets` - the amount of memory used by the Erlang ETS tables.

4.2.3 `hive.router`

This subsection describes the **hive.router** portion of the Hive statistics. Metrics found in this subsection measure the state of the Hive Router:

- `hive.router.uptime` - the router uptime (**in milliseconds**).
- `hive.router.spawned_clients` - the total number of client processes ever spawned by the Router.
- `hive.router.current_clients` - the current number of running client processes managed by the Router.
- `hive.router.requests` - the total number of requests processed by the Router.
- `hive.router.msg_queue_length` - the length of the Erlang message queue of the Router process. Corresponds directly to the number of requests queued on the router.
- `hive.router.routed_events` - the total number of internal events routed to the client processes by the Router.
- `hive.router.routed_msgs` - the total number of Socket.IO messages (external events) routed to the client processes by the Router.
- `hive.router.errors` - the total number of failed requests and other errors encountered by the Router.

4.2.4 `hive.cluster`

This subsection describes the **hive.cluster** portion of the Hive statistics. Metrics found in this subsection measure the state of the Hive Cluster Manager of a node:

- `hive.cluster.nodes` - a JavaScript array of strings listing other connected Hive nodes,
- `hive.cluster.size` - the number of nodes currently connected to the cluster,
- `hive.cluster.calls` - the total number of cluster wide, synchronous requests performed by this nodes Cluster Manager,
- `hive.cluster.casts` - the total number of cluster wide, asynchronous requests performed by this nodes Cluster Manager,
- `hive.cluster.errors` - the total number of failed requests and other errors encountered by the Cluster Manager.

4.2.5 `clients`

This subsection describes the **clients** part of the Hive statistics. Each metric found in this section measures the general quantities related to the Client FSMs:

- `clients.total` - the total number of alive clients (for Router debugging purposes).
- `clients.websocket` - the total number of alive WebSocket-based clients.
- `clients.xhr_polling` - the total number of alive XHR-polling-based clients.
- `clients.errors` - the total number of errors encountered by the client processes.

4.2.6 `clients.states`

This subsection describes the **clients.states** portion of the Hive statistics. Each metric found here describes the operation of the Client FSMs - state transitions and such:

- `clients.states.generic` - the total number of generic (uninitialized via Socket.IO handshake) clients.
- `clients.states.transient` - the total number of clients in **transient** state - clients that are unable to send Socket.IO messages for various reasons (for example, waiting for XHR-polling GET request).
- `clients.states.waiting` - the total number of clients in **waiting** state - clients ready to communicate, but awaiting a response/action from the Hive server.
- `clients.states.polling` - the total number of clients in **polling** state - clients buffering and processing replies.
- `clients.states.transitions` - the total number of state transitions of the client FSMs.

4.2.7 `clients.events`

This subsection describes the **clients.events** portion of the Hive statistics. Each metric found here describes the the operation of the Client logic - dispatched Socket.IO events, events received from the rest of the Hive server, etc:

- `clients.events.total` - the total number of events processed by the Client processes, includes `control`, `external` and `internal`.
- `clients.events.errors` - the total number of event related errors encountered by the Client processes, includes `control_errors`, `external_errors` and `internal_errors`.
- `clients.events.external` - the total number of external events (Socket.IO messages received) processed by the Client processes.
- `clients.events.internal` - the total number of internal events (internal messages and Socket.IO replies) processed by the Client processes.
- `clients.events.control` - the total number of control events (messages used internally by various Hive submodules) processed by the Client processes.
- `clients.events.internal_errors` - the total number of errors encountered when processing internal events.
- `clients.events.external_errors` - the total number of errors encountered when processing external events (Socket.IO messages).
- `clients.events.control_errors` - the total number of errors encountered when processing control events.

4.2.8 `clients.state_mgr`

This subsection describes the **clients.state_mgr** portion of the Hive statistics. Metrics in this subsection correspond to various Client State Manager quantities.

- `clients.state_mgr.requests` - the total number of State Manager requests.
- `clients.state_mgr.errors` - the total number of errors encountered by the State Manager.
- `clients.state_mgr.init` - the total number of State Manager `init` requests.
- `clients.state_mgr.get` - the total number of State Manager `get` requests.
- `clients.state_mgr.set` - the total number of State Manager `set` requests.
- `clients.state_mgr.cleanup` - the total number of State Manager `cleanup` requests.

4.2.9 `clients.transports`

This subsection describes the **clients.transports** portion of the Hive statistics. Metrics in this subsection measure the underlying transport protocols state and operation. Currently there are two subsections defined, for **WebSocket** and **XHR-polling** transports respectively:

- `clients.transports.http.requests` - the total number of HTTP requests received (includes the Socket.IO handshakes but not HTTP Connectors, etc).
- `clients.transports.http.errors` - the total number of bad requests and errors encountered by the transports.
- `clients.transports.http.2XX` - the total number of HTTP code 2XX replies.
- `clients.transports.http.4XX` - the total number of HTTP code 4XX replies.
- `clients.transports.http.5XX` - the total number of HTTP code 5XX replies.
- `clients.transports.http.???` - the total number of HTTP replies with other codes.
- `clients.transports.http.hang_up` - the total number of prematurely closed connections (for example, browser closing).

- `clients.transports.websocket.requests` - the total number of WebSocket requests (corresponds directly to the number of WebSocket protocol upgrades).
- `clients.transports.websocket.errors` - the total number of errors encountered when processing WebSocket errors.
- `clients.transports.websocket.frames` - the total number of WebSocket frames received.
- `clients.transports.websocket.ok` - the total number of “good” WebSocket replies (analogous to `http.2XX`).
- `clients.transports.websocket.bad` - the total number of “bad” WebSocket replies (analogous to `http.4XX` and `http.5XX`).
- `clients.transports.websocket.hang_up` - the total number of prematurely closed connections (for example, browser closing).

4.2.10 `clients.hooks`

This subsection describes the **clients.hooks** portion of the Hive statistics. Metrics in this subsection measure various quantities related to the Hive Hooks facility:

- `clients.hooks.calls` - the total number of Hive Hooks invocations (either due to Client Messages or external dispatch requests).
- `clients.hooks.errors` - the total number of Hive Hooks errors encountered by the Client processes.

Additionally, the same set of counters is defined on a per-hook-event basis. For an event named **$(name)**, the folowing counters will be added to the Monitor. Due to the dynamic nature of the hooks and their per-client character, **counters described below are added on use** and may not be included in the Monitor output at all times:

- `clients.hooks.$(name).calls` - the total number of Hive Hooks invocations.
- `clients.hooks.$(name).errors` - the total number of Hive Hooks errors encountered by the Client processes.
- `clients.hooks.hp`
  The Hive Protocol hook related metrics.
  - `clients.hooks.hp.get` - the total number of `hp.get` Hook invocations.
  - `clients.hooks.hp.put` - the total number of `hp.put` Hook invocations.
  - `clients.hooks.hp.post` - the total number of `hp.post` Hook invocations.

  Additionally, the same set of counters is defined on a per-hook-event basis. For an event named **$(name)**, the folowing counters will be added to the Monitor. Due to the dynamic nature of the hooks and their per-client character, **counters described below are added on use** and may not be included in the Monitor output at all times:
  - `clients.hooks.$(name).hp.get` - the total number of `hp.get` Hook invocations.
  - `clients.hooks.$(name).hp.put` - the total number of `hp.put` Hook invocations.
  - `clients.hooks.$(name).hp.post` - the total number of `hp.post` Hook invocations.
- `clients.hooks.pubsub`
  The Hive Pub-Sub hook related metrics.
  - `clients.hooks.pubsub.publish` - the total number of Pub-Sub channel publish requests.
  - `clients.hooks.pubsub.subscribe` - the total number of Pub-Sub channel subscriptions.
  - `clients.hooks.pubsub.unsubscribe` - the total number of Pub-Sub channel unsubscriptions.

  Additionally, the same set of counters is defined on a per-hook-event basis. For an event named **$(name)**, the folowing counters will be added to the Monitor. Due to the dynamic nature of the hooks and their per-client character, **counters described below are added on use** and may not be included in the Monitor output at all times:

  - `clients.hooks.$(name).pubsub.publish` - the total number of **name** channel publish requests.
  - `clients.hooks.$(name).pubsub.subscribe` - the total number of **name** channel subscriptions.
  - `clients.hooks.$(name).pubsub.unsubscribe` - the total number of **name** channel unsubscriptions.

4.2.11 `connectors`

This subsection describes the **connectors** part of the Hive statistics. Each metric found here measures the general quantities related to the Hive Connectors:

- `connectors.requests` - the total number of requests to the Connectors Manager.
- `connectors.errors` - the total number of errors encountered by the Connectors Manager.
- `connectors.pools` - the total number of connector pools running in the Hive server.
- `connectors.starts` - the total number of pool starts.
- `connectors.stops` - the total number of pool stops.
- `connectors.unsafe_transactions` - the total number of unsafe transactions performed on the Connector pools.
- `connectors.safe_transactions` - the total number of safe transactions performed on the Connector pools.
- `connectors.rents` - the total number of Connector rents (acquisition from a pool for later use).
- `connectors.returns` - the total number of Connector returns (returns to a pool).

The connector pools are grouped by their type and the name they were given, so it is possible to measure multiple instances of each connector pool type.

4.2.12 `connectors.http`

This subsection describes the **connectors.http.$(name)** portion of the Hive statistics. Each metric found here is related to an HTTP connector pool named **name**:

- `connectors.http.$(name).workers` - the total number of active workers in the pool.
- `connectors.http.$(name).requests` - the total number of requested actions performed by the workers.
- `connectors.http.$(name).errors` - the total number of errors encountered by the workers.
- `connectors.http.$(name).sync_gets` - the total number of synchronous GET requests requested.
- `connectors.http.$(name).sync_posts` - the total number of synchronous POST requests.
- `connectors.http.$(name).async_posts` - the total number of asynchronous POST requests.

4.2.13 `connectors.redis`

This subsection describes the **connectors.redis.$(name)** portion of the Hive statistics. Each metric in this subsection is related to the Redis connector pool named **name**:

- `connectors.redis.$(name).workers` - the total number of active workers in the pool.
- `connectors.redis.$(name).requests` - the total number of requested actions performed by the workers.
- `connectors.redis.$(name).errors` - the total number of errors encountered by the workers.
- `connectors.redis.$(name).queries` - the total number of Redis queries sent by the workers.

4.2.14 `connectors.tcp`

This subsection describes the **connectors.tcp.$(name)** part of the Hive statistics. Each metric in here is related to the TCP connector pool named **name**:

- `connectors.tcp.$(name).workers` - the total number of active workers in the pool.
- `connectors.tcp.$(name).requests` - the total number of requested actions performed by the workers.
- `connectors.tcp.$(name).errors` - the total number of errors encountered by the workers.
- `connectors.tcp.$(name).send` - the total number of **send** requests processed by the workers.
- `connectors.tcp.$(name).recv` - the total number of **recv** requests processed by the workers.

4.2.15 `pubsub`

This subsection describes the **pubsub** portion of the Hive statistics. Metrics in this subsection measure various quantities related to the Hive Pub-Sub facility:

- `pubsub.uptime` - the uptime (in **milliseconds**) of the Hive Pub-Sub Manager.
- `pubsub.requests` - the total number of Hive Pub-Sub requests.
- `pubsub.errors` - the total number of errors encountered by Hive Pub-Sub facility.
- `pubsub.total_channels` - the total number of active Hive Pub-Sub channels.
- `pubsub.status` - the total number of `status` requests issued to the Pub-Sub channels.
- `pubsub.subscribe` - the total number of `subscribe` requests issued to the Pub-Sub channels.
- `pubsub.unsubscribe` - the total number of `unsubscribe` requests issued to the Pub-Sub channels.
- `pubsub.join` - the total number of `join` requests issued to the Pub-Sub channels.
- `pubsub.leave` - the total number of `leave` requests issued to the Pub-Sub channels.
- `pubsub.publish` - the total number of `publish` requests issued to the Pub-Sub channels.
- `pubsub.published_events` - the total number of events published on the Pub-Sub channels.

Additionally, a similar set of counters is defined on a per-channel-prefix basis. For a channel prefixed **$(name)**, the folowing counters will be added to the Monitor.

- `pubsub.channels.$(name).requests` - the total number of Hive Pub-Sub requests issued to channels with prefix **name**.
- `pubsub.channels.$(name).errors` - the total number of errors encountered by Hive Pub-Sub channels with prefix **name**.
- `pubsub.channels.$(name).total_channels` - the total number of active Hive Pub-Sub channels with prefix **name**.
- `pubsub.channels.$(name).status` - the total number of `status` requests issued to channels with prefix **name**.
- `pubsub.channels.$(name).subscribe` - the total number of `subscribe` requests issued to channels with prefix **name**.
- `pubsub.channels.$(name).unsubscribe` - the total number of `unsubscribe` requests issued to channels with prefix **name**.
- `pubsub.channels.$(name).publish` - the total number of `publish` requests issued to channels with prefix **name**.
- `pubsub.channels.$(name).subscribed_clients` - the total number of Clients subscribed to channels with prefix **name** (note that a Client subscribed to several channels with the same prefix **will appear multiple times in this metric**).
- `pubsub.channels.$(name).published_events` - the total number of events published on channels with prefix **name**.

4.2.16 `api`

This subsection describes the **api** portion of the Hive statistics. Metrics in this subsection measure various quantities related to the Hive API Server:

- `api.requests` - the total number of Hive API Server requests.
- `api.errors` - the total number of errors encountered by the Hive API Server while processing requests.

4.2.17 `api.hive`

This subsection describes the **api.hive** portion of the Hive statistics. Metrics in this subsection measure various quantities related to the Pub-Sub part of the Hive API:

- `api.hive.requests` - the total number of Hive API Server related requests.
- `api.hive.errors` - the total number of errors encountered by the Hive API Server while processing general requests.

4.2.18 `api.router`

This subsection describes the **api.router** portion of the Hive statistics. Metrics in this subsection measure various quantities related to the Pub-Sub part of the Hive API:

- `api.pubsub.requests` - the total number of Hive API Server Router related requests.
- `api.pubsub.errors` - the total number of errors encountered by the Hive API Server while processing Router related requests.

4.2.19 `api.pubsub`

This subsection describes the **api.pubsub** portion of the Hive statistics. Metrics in this subsection measure various quantities related to the Pub-Sub part of the Hive API:

- `api.pubsub.requests` - the total number of Hive API Server Pub-Sub related requests.
- `api.pubsub.errors` - the total number of errors encountered by the Hive API Server while processing Pub-Sub related requests.

4.2.20 `api.clients`

This subsection describes the **api.clients** portion of the Hive statistics. Metrics in this subsection measure various quantities related to the Clients related parte of the Hive API:

- `api.clients.requests` - the total number of Hive API Server Clients related requests.
- `api.clients.errors` - the total number of errors encountered by the Hive API Server while processing Clients related requests.

4.3 Monitor API

This section describes the RESTful API exposed by the Hive Monitor.
The API is available on the same host as the rest of the Hive server, on a configured port. The structure of the Hive Monitor URL is as follows:

```
Host [ ':' Port ] '/monitor/' APIKey [ '/' StatisticsSection ]
```

The `APIKey` can be configured via the Hive configuration file. The `StatisticsSection` is the full qualified name of a statistics section, or a metric, for example:

```
host:port/monitor/apikey/hive.memory
host:port/monitor/apikey/connectors.http.pool_name
host:port/monitor/apikey/clients.transports.websocket.frames
```

Hive Monitor supports two HTTP methods:

- `DELETE` - Resets the value of a given metric to 0. The `StatisticsSection` portion of the URL has to be a full qualified name of a metric.
- `GET` - Returns the value of a given section of the Hive statistics in the format described in the next subsection. The `StatisticsSection` portion of the URL has to be a valid statistics section, or a metric.

In case of a bad request an appropriate error message is returned.

4.4 Monitor response format

This section describes the format of data returned by the Hive Monitor via its RESTful API. The Hive Monitor uses JSON format to represent its output. If a Monitor request results in an error, a convention described in the next section is used, otherwise the returned JSON objects are structured as described in statistics structure section. The resulting JSON object **always wraps** the output in all requested sections. Example (pretty formatted) Hive Monitor output:

```
// GET host:port/monitor/apikey/hive.memory
{
    "hive" : {
        "memory" : {
            "total" : 22668016,
            "system" : 13182056,
            "processes" : 9485960,
            "ets" : 672616,
            "code" : 8332965,
            "binary" : 249856,
            "atom" : 339441
        }
    }
}

// GET host:port/monitor/apikey/clients.hooks.on_connect.calls
{
    "clients" : {
        "hooks" : {
            "on_connect" : {
                "calls" : 1
            }
        }
    }
}
```

5 Logging & Errors

This section describes how error handling and logging is performed in the Hive server.

5.1 Log

Hive uses Lager for logging purposes and therefore all of its quirks apply. The only difference is the configuration, which is intercepted by Hive and greatly simplified for convenience (described earlier).
In general, the logging back-ends use a fairly simple log format shown below:

```
Timestamp '[' LogLevel ']' Pid '@' ModuleName ':' FunctionName ':' Line LogMessage
```

Each log line consists of a timestamp, a log level, a Pid of the process that the log line originated from (useful for live-debugging) and the exact location of the log line in the source code. The timestamp is in the following format:

```
YYYY-MM-DD HH:MM:SS.mmm
```

5.1.1 Log file

The log file back-end uses the full logging format and provides all the necessary information needed to identify the origin of the log line and (usually) the reason for its existence. The log file is named **hive.log** and its associated log-level can be configured in the configuration file.

5.1.2 Console log

The console logging back-end uses a simplified logging format for obvious reasons of clarity:

```
Timestamp '[' LogLevel ']' LogMessage
```

It provides sufficient information to identify problems, and the timestamp can be easily searched for in the log file if need-be. Additionally, a copy of the console log is stored in logging directory under **console.log** file.

5.1.3 Error log

The error log (**error.log**) is a filtered version of the Log file that contains only highest priority messages, namely, the error ones. It is created by default and it always exists. You can thank me later.

5.1.4 Crash log

The crash log (**crash.log**) does not strictly follow the logging convention, because it consists of crash reports, which might occur even before the logging is set up. It is created by default and always exists. Additionally, unaltered Erlang VM crash dumps are stored in the log directory as well.

5.2 Error messages

This section gives various details concerting error messages that are generated throught the Hive server. Internally, errors are represented as Erlang tuples consisting of an `error_code` (an Erlang symbol) and a short description (*usually* an Erlang binary string):

```
{ error_code, <<"Short error description.">> }
```

Errors are logged at the place of their origin and additionally at each *layer* they pass through providing an error-trace useful for debugging. For example:

```
14:45:32.213 [warning] Tried accessing an unknown Hive Pub-Sub channel: foo.bar.baz
14:45:32.213 [debug] Hive Pub-Sub Hook encountered an error:
             {unknown_channel_id,<<"Tried accessing an unknown Hive Pub-Sub channel: foo.bar.baz">>}
14:45:32.213 [debug] Hive Hooks Client encountered an error:
             {unknown_channel_id,<<"Tried accessing an unknown Hive Pub-Sub channel: foo.bar.baz">>}
14:45:32.213 [debug] Hive Client encountered an error:
             {unknown_channel_id,<<"Tried accessing an unknown Hive Pub-Sub channel: foo.bar.baz">>}
```

This behaviour can be adjusted using the log-levels of the **log** section of the configuration file. The following diagram presents **a simplified** Hive layer model (errors generated in top layers “sink” to the lower layers and eventually reach the bottom, where they are reported to the Client):

```
+---------------------+---------------------+
|      Connectors     |    State Manager    |
+---------------------+---------------------+
|        Hooks        |   Internal Events   |
+---------------------+---------------------+
|                Hooks Client               |
+---------------------+---------------------+
|     Client (XHR)    |  Client (WebSocket) |
+---------------------+---------------------+---------------------+
|     XHR Handler     |  WebSocket Handler  |  Pub-Sub Channels   |
+---------------------+---------+-----------+---------------------+---------------------+
|            Router             |              Pub-Sub            |  Config Validator   |
+-------------------------------+---------------------------------+---------------------+
|             API               |            HTTP  Servers        |       Monitor       |
+-------------------------------+---------------------------------+---------------------+
```

5.3 Error codes

This section lists and gives a short destription of various error codes used throughout the Hive server. Longer problem descriptions (hopefully sufficient to determine the solution) are attached to every error instance.

- `bad_api_request` - invalid Hive API Server request.
- `bad_monitor_request` - invalid Hive Monitor request.
- `bad_request` - invalid Hive HTTP Server request (mostly invalid initial Socket.IO requests).
- `bad_origin` - origin specified in the request header is not accepted by the Hive server.
- `connectors_error` - generic Hive Connectors error (most likely internal event resulting in unhandled requests).
- `bad_connector_id` - tried accessing an invalid Hive Connector id (most likely a Hive Connector has been shut down and not restarted).
- `http_error` - Hive HTTP Connector related error, usually signalizes a bad HTTP response received by the connector.
- `tcp_error` - Hive TCP Connector related error, usually signalizes a TCP socket error encountered by the connector.
- `hp_error` - Hive Protocol Hook related error, usually signalizes a problem with the backend that Hive talks to.
- `redis_error` - Hive Redis Connector related error, usually signalizes problems with the Redis database encountered by the connector.
- `cluster_error` - Hive Cluster related error (most likely a cluster collapse indication).
- `router_error` - Hive Router related error (most likely an unhandled Hive Router request).
- `bad_session_id` - tried accessing (via Hive Router) a nonexistent Session ID.
- `pubsub_error` - Hive Pub-Sub related error (most likely an unhandled Hive Pub-Sub request).
- `bad_channel_id` - tried accessing a non-existent Hive Pub-Sub channel.
- `unknown_channel_id` - tried accessing (via Hive Pub-Sub) a nonexistent Channel ID.
- `access_denied` - privilege level specified for a Hive Pub-Sub requests is insufficient to perform it.
- `pubsub_channel_error` - Hive Pub-Sub Channel related error (most likely an unhandled Hive Pub-Sub Channel request or lack of privileges).

- `client_error` - Hive Client related error (most likely problems with Client initialization).
- `hive_error` - **critical** Hive server error, indicates some serious problems concerning the supervision tree, or various Hive modules.
- `bad_internal_event` - JSON related, indicates malformed Internal Event.
- `bad_external_event` - JSON related, indicase malformed External Event.
- `big_num` - JSON related, indicates invalid numeric value.
- `invalid_json` - JSON related, indicates malformed JSON data.
- `invalid_string` - JSON related, indicates invalid JSON string literal.
- `trailing_data` - JSON related, indicates trailing JSON data in an otherwise valid JSON object.
- `internal_error` - generic error code, returned by the `error` internal event action, or in **production mode**.
- `value_undefined` - Hive Config validator related error, indicates that a required value wasn't defined in the configuration file.
- `invalid_config` - Hive Config validator related error, indicates that a value wasn't conforming to its constraints (for example, by being outside of accepted range).
- `bad_hook_descriptor` - Hive Hooks related error, indicates that a supplied Hive Hook descriptor was malformed.
- `bad_connector_descriptor` - Hive Connectors related error, indicates that a supplied Hive Connectors descriptor was malformed.
- `file_missing` - Hive Config validator related error, indicates that a configuration file (or a JSON schema file) could not be found.
- `file_error` - Hive Config validator related error, indicates that a configuration file could not be accessed (most likely wrong permission).
- `bad_subschema_id` - Hive Config validator related error, indicates that a requested JSON Schema ID could not be resolved.
- Other - Hive Plugins may return their own error codes and descriptions.

5.4 Error response format

This section describes the format of error messages that are sent to the outside world (via API/Monitor replies or otherwise directly to the Client). Errors are represented as **JSON objects** that define exactly two fields: `error` and `description`:

```
{
    "error" : "error_code",
    "description" : "A short description."
}
```

Example (pretty formatted) error responses:

```
// Hive error response:
{
    "error" : "unknown_channel_id",
    "description" : "Tried accessing an unknown Hive Pub-Sub channel: foo.bar.baz"
}
// Monitor error message:
{
    "error" : "bad_monitor_request",
    "description" : "Requested metric \"foo\" does not exist."
}
```

Error responses that are sent to the Client are additionally wrapped in **a Socket.IO event** of the following form:

```
{
    "name" : "hive_error",
    "args" : "Error response."
}
```

For example:

```
{
    "name" : "hive_error",
    "args" : [
        {
            "error" : "bad_channel_id",
            "description" : "Tried unsubscribing an unknown channel: foo.bar.baz"
        }
    ]
}
```

6 API Servers

This section describes the RESTful API exposed by the Hive Server.
The API is available on the same host as the rest of the Hive server, on a configured port. The structure of the Hive Monitor URL is as follows:

```
Host [ ':' Port ] '/api/' APIKey '/' APISection '/' Action [ '/' Arguments ... ]
```

The `APIKey` can be configured via the Hive configuration file. The `APISection` is the name of an API section while `Action` and `Arguments` are the name of an action to take and its arguments respectively, for example:

```
host:port/api/apikey/pubsub/publish/...
host:port/api/apikey/clients/dispatch/...
```

6.1 RESTful API

The following subsections describe various parts of the Hive API. Each section names an endpoint, consisting of `APISection`, `Action` and `Arguments`, used to perform the action, lists HTTP request methods it accepts and describes the structure of data required by the action.

6.1.1 `/hive/version/`

- Accepted HTTP methods:
  - `GET` - returns a string representation of the server version.
- Data format - none.

6.1.2 `/hive/stop/`

- Accepted HTTP methods:
  - `POST` - initiates graceful termination of the **entire Hive Server**.
- Data format - none.

6.1.3 `/router/enable/`

- Accepted HTTP methods:
  - `POST` - enables the Hive Router to accept new connections.
- Data format - none.

6.1.4 `/router/disable/`

- Accepted HTTP methods:
  - `POST` - disables the Hive Router. While disabled, Hive Router won't accept any new connections.
- Data format - none.

6.1.5 `/router/terminate/`

- Accepted HTTP methods:
  - `POST` - assumes the body of the request is in **a string** representing a termination reason and initiates graceful termination of the **currently connected Client Workers**. The Hive Router will be disabled during the termination and will stay disabled until explicitly enabled with a separate API call.
- Data format - a string.

6.1.6 `/clients/action/sid/`

- Accepted HTTP methods:
  - `POST` - assumes the body of the request is an **internal event** (or a JavaScript array of internal events) and routes it to the `sid` Client worker.
- Data format - an internal event, as described in a later section.

6.1.7 `/pubsub/action/cid/`

- Accepted HTTP request methods:
  - `POST` - assumes the body of the request is an **internal event** (or a JavaScript array of internal events) and publishes it to the Hive Pub-Sub channel `cid`.
- Data format - an internal event, as described in a later section.

6.1.8 `/pubsub/publish/cid/`

- Accepted HTTP request methods:
  - `POST` - assumes the body of the request is a **single external event** and publishes it directly to the Hive Pub-Sub channel `cid`.
- Data format - an external event.

6.1.9 `/pubsub/subscribe/id/`

- Accepted HTTP request methods:
  - `GET` - returns the total number of Clients subscribed to the Hive Pub-Sub Channel `id`.
  - `POST` - assumes the body of the request to be an array of Hive Pub-Sub Channel IDs and subscribes the Client worker corresponding to the session id `id` to them.
  - `DELETE` - assumes the body of the request to be an array of Hive Pub-Sub Channel IDs and unsubscribes the Client worker corresponding to the session id `id` to them.
- Data format - **a JSON array of strings** representing Hive Pub-Sub Channel IDs.

6.2 API response format

The Hive API Server responses (if any) are encoded as **simple JSON objects**. In case of encountering any errors, an error response conforming to a previously defined format will be genarated.
Example Hive API Server output:

```
{
    "subscribed_clients" : 1
}
```

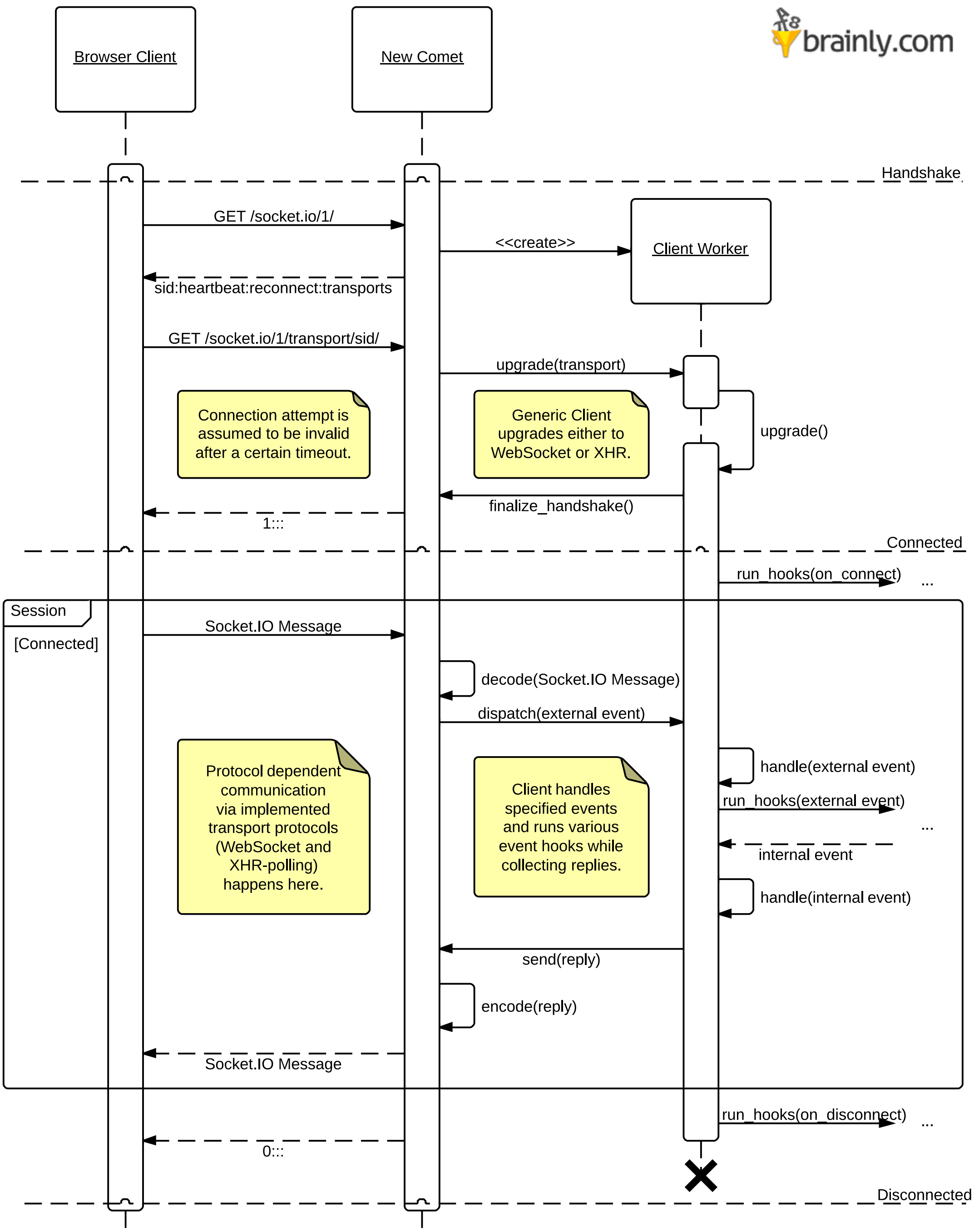
7 Hive Modules

The following section describes various Hive modules and gives a general idea of how they work and interact with each-other.

7.1 Client Workers

7.1.1 Client/Server communication

The following diagram shows the Client-Server communication routine.



Client initializes the connection to the Hive server and receives a Socket.IO handshake response. Next, the client is obliged to attempt to finalize the handshake by sending an initial request to the assigned **session ID** under a **transport of his choice**. During the handshake period a **generic client worker** process is created, which is later upgraded to a **specific client worker** once a transport has been selected. Connection attempt is assumed invalid after an **initialization timeout** specified in the configuration file of the Hive server and the client worker is removed.

Once the Socket.IO handshake has been finalized, Client-Server communication may take place. The Client sends Socket.IO messages to the Hive server over the selected transport; the Hive server decodes them into an internal format, routes them to the correct Client worker process (specified by the session ID) and **handles** them using a **worker module**. Every such **external event** (external as in "outside of the Brainly.com infrastructure") may trigger several actions of the following types:

- **running event hooks** - described in greater detail here and here; this may generate **internal events** (internal as in "originating from the Brainly.com infrastructure") which are handled similarly to the external ones, producing replies, state updates or running other event hooks, in turn generating even more internal events,
- **updating the client worker state** - described here,
- **sending a reply** - the trivial case, where a reply is sent to the client immediately (this is generally used for Socket.IO control messages, error handling and such).

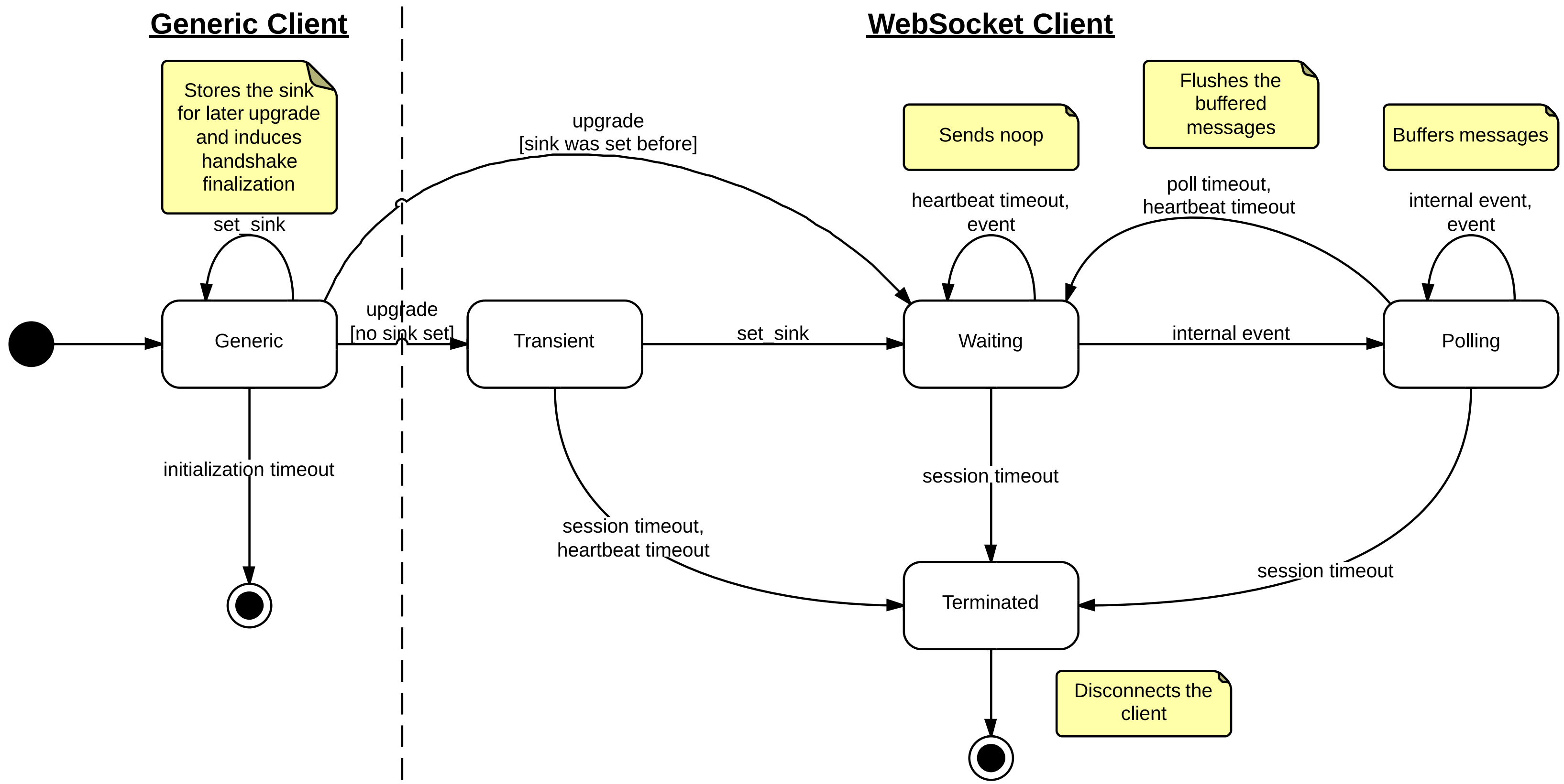
If a reply has been generated the Hive server starts polling for a specified amount of time (**poll timeout**) collecting more replies. Each reply is encoded and fed to the underlying transport handling code, which sends it back to the Client. If no replies are generated for a specified amount of time (**heartbeat timeout**) the Hive server will send an empty response to make sure the connection is kept alive. When a session is terminated (either by the client closing the connection, a session timeout or for any other internal reason, such as receiving a specific internal event) the connection is closed and the Client worker process is terminated.

It is easier to think (and it is the case of the implementation!) that the Client worker acts as a **finite state machine** with a given set of states and a state transition function. The states are:

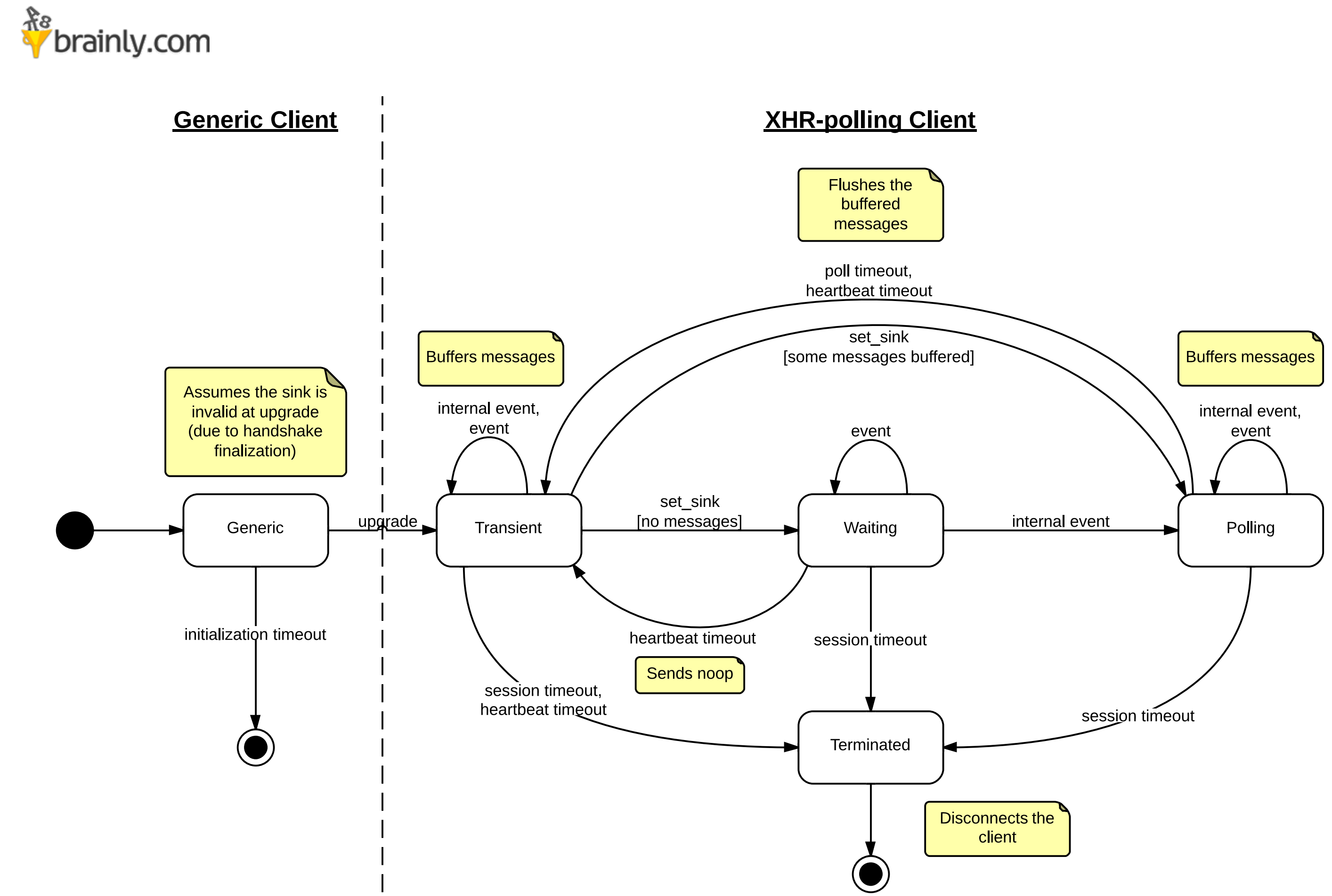
- **Generic** - a state in which the Client worker is initialized, but not yet finalized; the only valid action for this state is to transition to either of the other states after the client upgrade (upgrade that happens once the Client connects via a specified transport); if the Client fails to finalize the connection in a specified time (**initialization timeout**), the FSM transitions to the **Terminated** state,
- **Transient** - a "synchronization" state where all the Client worker initialization happens; received events are queued in this state and will be handled as soon as the FSM transitions to another state,
- **Waiting** - a state in which the Client worker is waiting for external and internal events to handle; events are handled and in case of a **reply** the FSM transitions to the **Polling** state,
- **Polling** - a state in which the Client worker is collecting more replies to send them as a batch; events are handled and replies are queued; after a specified time (**poll timeout**) the messages are sent to the Client and the FSM transitions to the **Waiting** state,
- **Terminated** - a state where the Client worker cleans-up after itself and is terminated; no state transitions are valid for this state.

The state transition scheme outlined above is very general and *not quite true* as it turns out, because different transport protocols require different approaches and special behaviours to be taken care of. For example, **the sink** (an abstraction representing the underlying transport handling code, where the Client worker **flushes** the replies) once initialized is always valid for WebSocket, but only until a reply is sent for XHR-polling meaning that XHR-polling clients will occasionally enter the Transient state for short periods of time but the WebSocket ones won't. For this reason the following subsections contain correct and complete state transition diagrams for the concrete Client worker types (currently **WebSocket** and **XHR-polling**).

7.1.2 WebSocket/FlashSocket Client



7.1.3 XHR-polling Client



7.1.4 Client Hooks

This subsection describes the Hive Hooks facility. Hooks are used to dispatch received **external** and **internal events** and perform certain actions based on the event's type and payload.

- Hook basics
  The Hive hooks are defined on a per-event basis, that is, each event type may (or may not) have several hooks associated with it. These hooks will be triggered, and actions they encapsulate will be performed each time an event of that type is received by a Client worker. This holds true for all **external events** (those originating from the Client) but not for all **internal events** (those originating from the rest of the system), the later must request further event dispatch (described here). There are several **special event types** that trigger Hive hooks, that are not received from the Client nor the rest of the system:

  - `on_connect` - event generated once the Client worker is upgraded to the final, specialized type; hooks associated with this event **must not return any replies**, as it is not yet certain that the connection is valid, and sending replies might fail,
  - `on_terminate` - event generated on graceful Hive termination; this hook is called shortly before the server termination; any return values are dispatched as normal, but keep in mind that the Client Process **might be forced to terminate** before any meaningful actions are taken,
  - `on_disconnect` - event generated on Client worker termination; any return values of the hooks associated with this event are **ignored and discarded**.

  Each Hive hook encapsulates a simple action that ts performes upon its invocation. The concrete actions depend on the hook type, and are described in a later section, but a general convention is kept that each action is described in terms of **metadata**, **arguments**, **modification** and **return values**:

  - Actions receive some Client worker **metadata** and have it available during execution (more on this later).
  - Actions take additional **arguments**, that can be specified in the configuration file.
  - Actions may **modify** the state of the Client worker during their execution.
  - Actions **return** one of three result types:
    * **no reply** - an empty response,
    * **reply** - returns a reply that will be sent to the Client,
    * **error** - signalizes an error that will be sent to the Client,
    * **stop** - stops the hooks execution and terminates the Client worker.

  Hooks are run until either of **stop** or **error** result is encountered or until there are no more hooks to run. All **replies** are accumulated and sent to the Client as a batch. Hook order **does** matter, since each hook might modify the Client workers state, which will be then passed to the next hook in the list.

  Hook actions may produce **internal events** that will be dispatched the same way as mentioned previously.

  Example hook definitions:

```
"hooks" : {
    "on_connect" : [
        { "hook" : "utils.console_dump", "args" : "Connected!" }
    ],
    "on_disconnect" : [
        { "hook" : "utils.console_dump", "args" : "Disconnected!" }
    ],
    "ping" : [
        { "hook" : "utils.console_dump", "args" : "Pinged!" },
        { "hook" : "utils.echo", "args" : null }
    ],
}
```

  Hooks defined above will cause each Hive Client worker to print "Connected!" and "Disconnected!" to the console on their initialization and termination respectively. Additionally, a ping event will be logged and echoed back to the Client each time it is received.

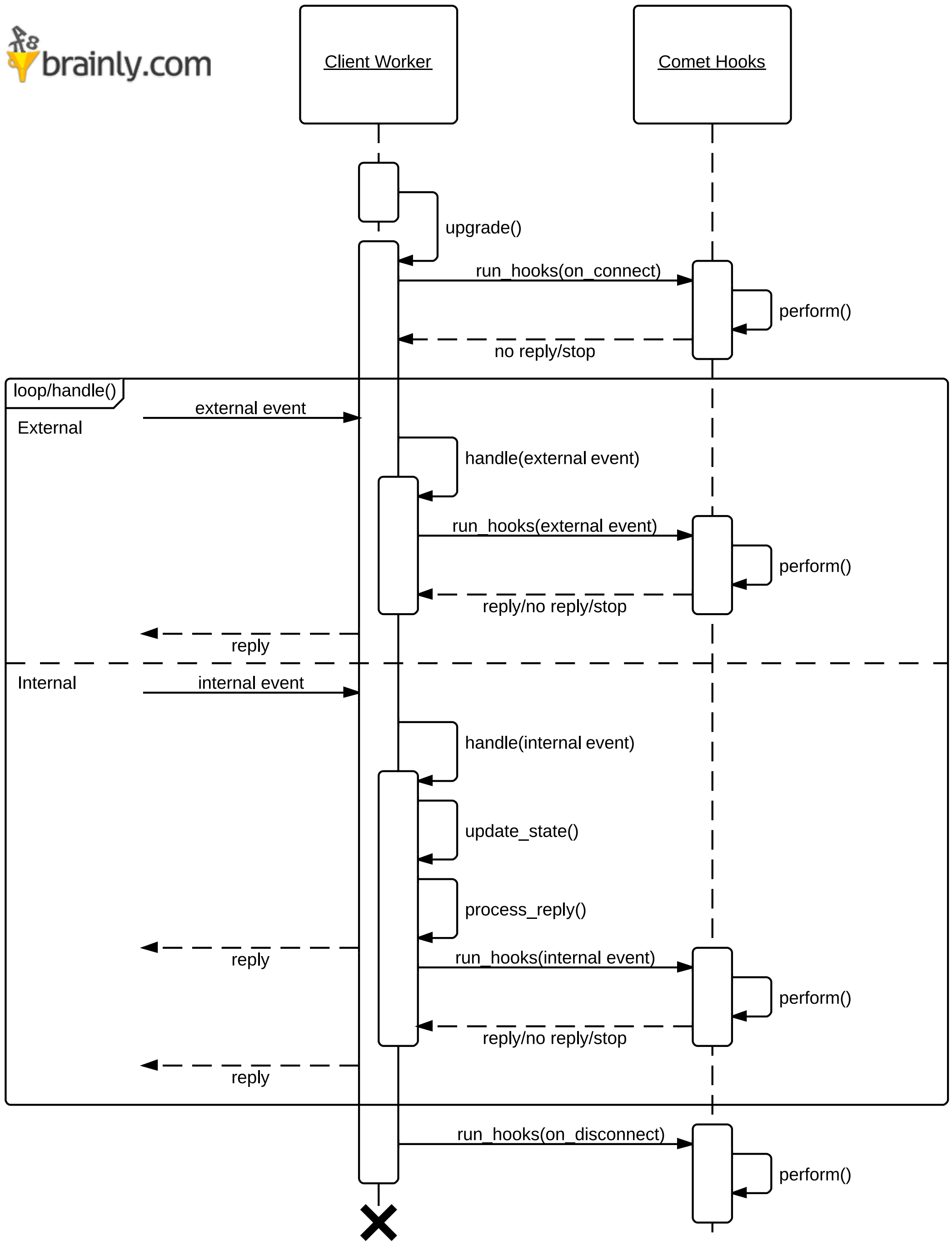
- Client metadata
  This subsection describes the metadata used by various Hive Hooks. The metadata consists of the internal state of a Client worker,its Session ID and the event that triggered the hook. It is passed to the hook together with additional arguments defined in the configuration file and it is represented as **a JSON object** conforming to the following format:

```
{
    "sid" : "the Session ID of a Client",
    "state" : "the state of the Client worker",
    "trigger" : "the hook-triggering event or null if not present"
    "trigger_id" : "the ACK id of the triggering event"
}
```

  For example:

```
{
    "sid" : "1238db436e20dbffff182466c8efaa5d757231",
    "trigger" : {
        "name" : "ping",
        "args" : ["pong"]
    },
    "trigger_id" : "1+",
    "state" : {
        "initial_value" : null
    }
}
```

- Hooks summary
  The Client worker/hook interaction is summarized on the following diagram:



7.1.5 Client state management

This subsection describes the Client State Manager - a facility used to manage the Client worker state as a key-value store. Most of the technical details have been omitted for various reasons.
The configuration file has to specify **a state manager descriptor**, which defines several fields:

```
{
    "state_manager" : "The name of the State Manager to use, e.g. sm.redis.",
    "initial_value" : "Initial state of the Client.",
```

```
    "args" : "Additional arguments required by the State Manager."
}
```

The `initial_value` may be **any JSON value**, however since the State Manager provides a key-value store interface, JSON objects are treated as lists of key-value pairs with each value stored at each key, and values other than JSON objects are stored using `initial_value` key. For example:

```
"initial_value" : {"foo" : "bar", "bar" : "baz"} // Stored as:  foo:bar, bar:baz
"initial_value" : [1, 2, 3]                      // Stored as: initial_value:[1, 2, 3]
"initial_value" : "foo"                          // Stored as: initial_value:foo
```

A list all available State Managers and their configuration parameters can be found here.

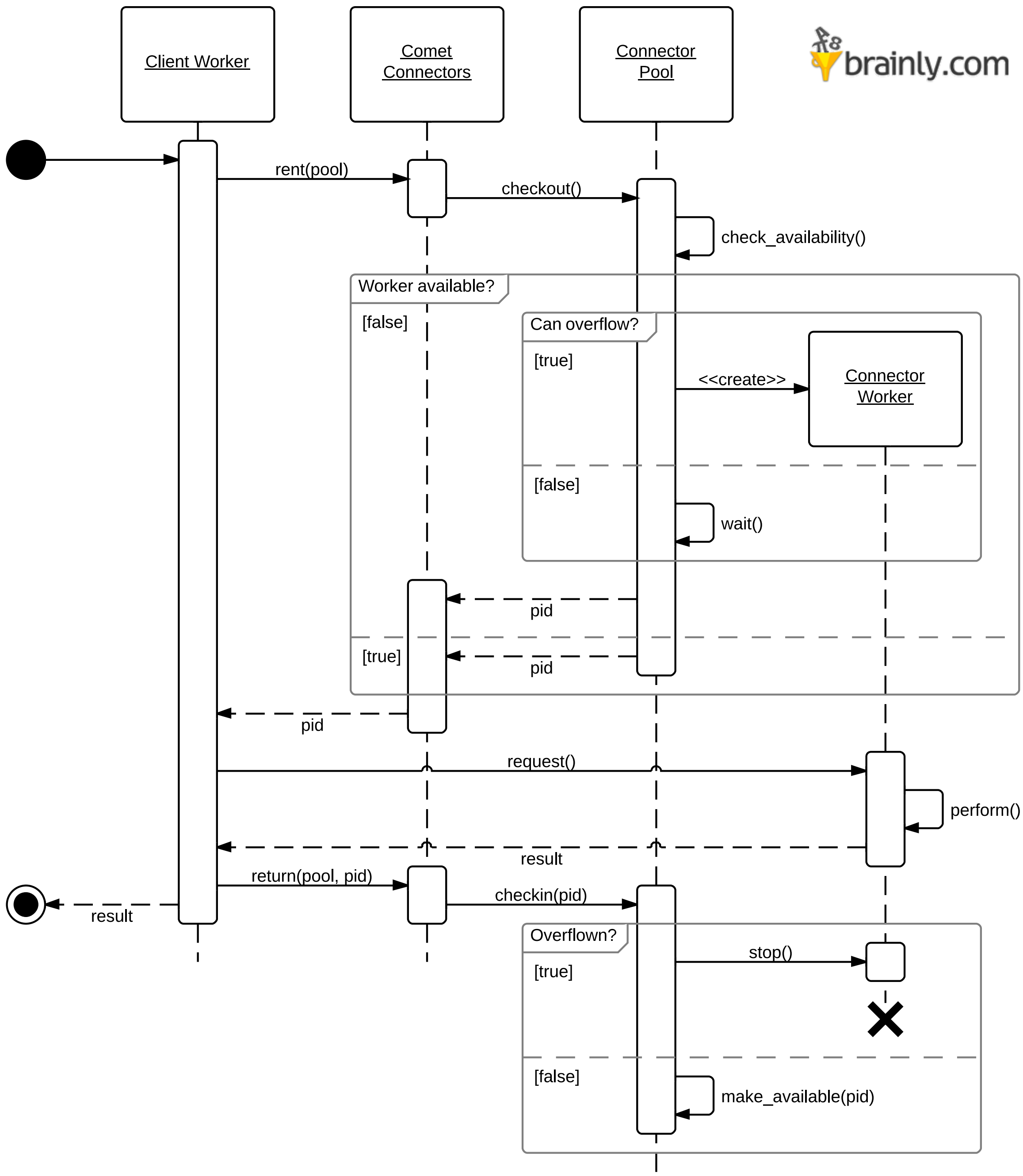
7.2 Service Connectors

This section describes other details of the Hive Connectors facility. The Hive Connectors facility that provides access to various services, such as Redis databases or HTTP servers, uses several schemes of worker pool management. A pool consists of a supervisor and a number of **worker processes**, which are **rented** by various other Hive modules (for example Hive Protocol hooks). Technical details of pool management and worker renting were omitted from this user guide.

Each pool can be **named** and has a configurable size that may increase temporarily during run-time, therefore each pool can be described in terms of its **name**, a **connector plugin**, **size** and **overflow** (the maximum number of additional worker processes to spawn under heavy server load) and **arguments** it uses for set-up, which, incidentally, are the configuration parameters required by each Hive Connectors pool.
A list all available Connector pools can be found here.

7.2.1 Connectors summary

The Client worker/Connector pool interaction is summarized on the following diagram:



Client Worker
Comet Connectors
Connector Pool
brainly.com
rent(pool)
checkout()
check_availability()
Worker available?
[false]
Can overflow?
[true]
<<create>>
Connector Worker
[false]
wait()
pid
[true]
pid
pid
request()
perform()
result
return(pool, pid)
result
checkin(pid)
Overflown?
[true]
stop()
[false]
make_available(pid)

7.3 Pub-Sub Channels

This subsection and its subsections describe the Hive Pub-Sub facility.

7.3.1 Channel templates

A set of Pub-Sub channel templates can be defined in the Hive configuration file. Each template consists of a **channel prefix** - the first part of the channel ID, and a **channel descriptor** - as described here. At runtime, channels are created using one of the templates determined by their channel ID. Any attempt at creating a channel that doesn't follow any channel template defined in the configuration file will fail and an error response will be produced.

7.3.2 Channel types

Each template defines a channel type. Hive Pub-Sub currently supports two types of channels:

- **private** channels - accesible only via the Hive API Server or via Client Pub-Sub Hook with a sufficient privilege level.
- **public** channels - accessible by all Clients.

7.3.3 Channel management

Each Pub-Sub channel is created with the first Client attempting a subscription. As long as there are Clients subscribed to a channel it'll stay active and publish messages to its subscribents. When the last Client unsubscribes from the channel, depending on the timeout that was configured for the corresponding channel template, the channel might expire and cease to exist, or continue waiting for more subscribents.
Channel (un)subscriptions can be managed in either of the following ways:

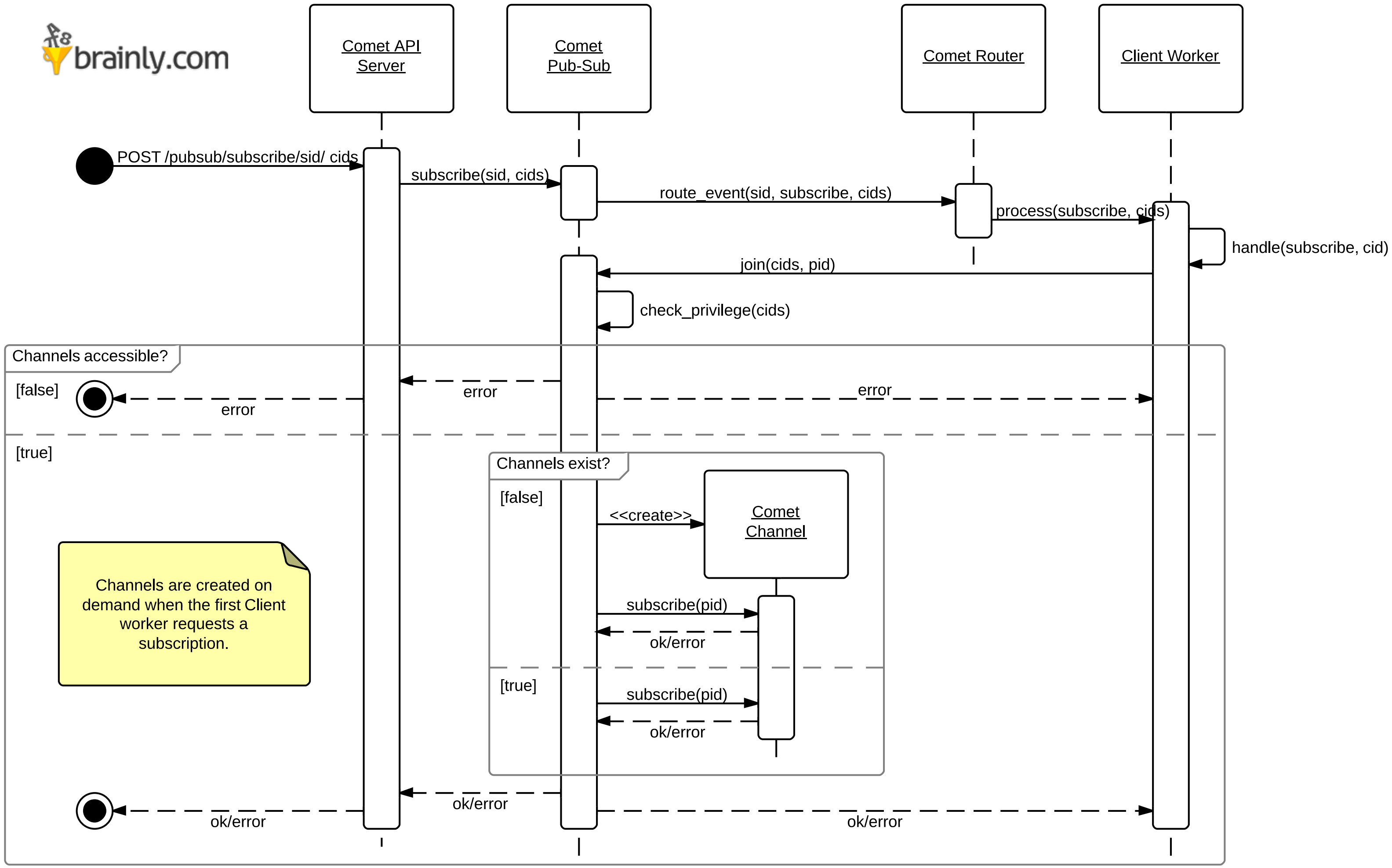
- via the Hive API Server - as described here,
- via the Hive Pub-Sub Hook - as described here.

Events can be published to the Hive Pub-Sub channel only via the Hive API Server (as described here).

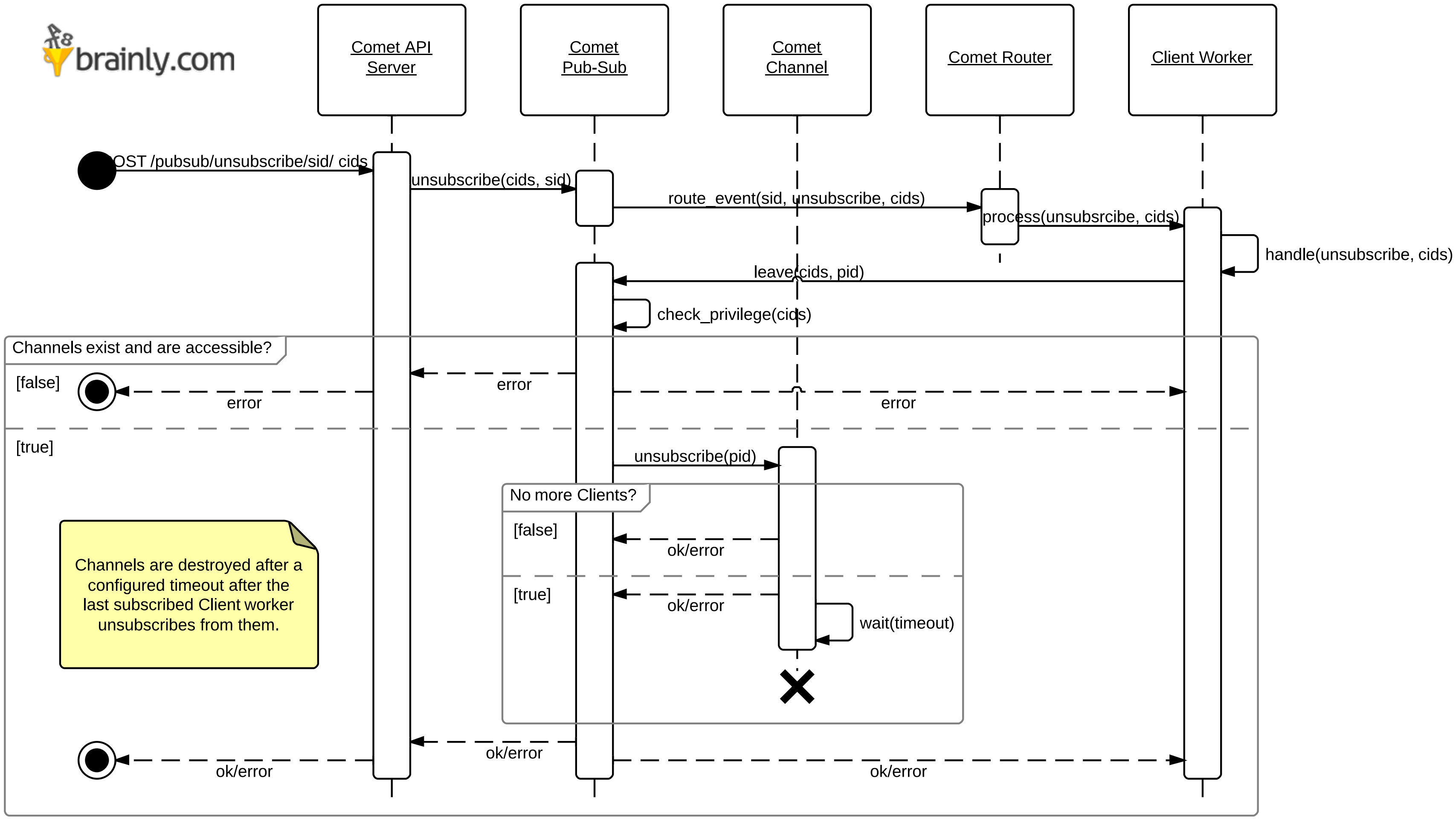
7.3.4 Pub-Sub summary

The following diagrams summarize the Hive Pub-Sub channel operation.

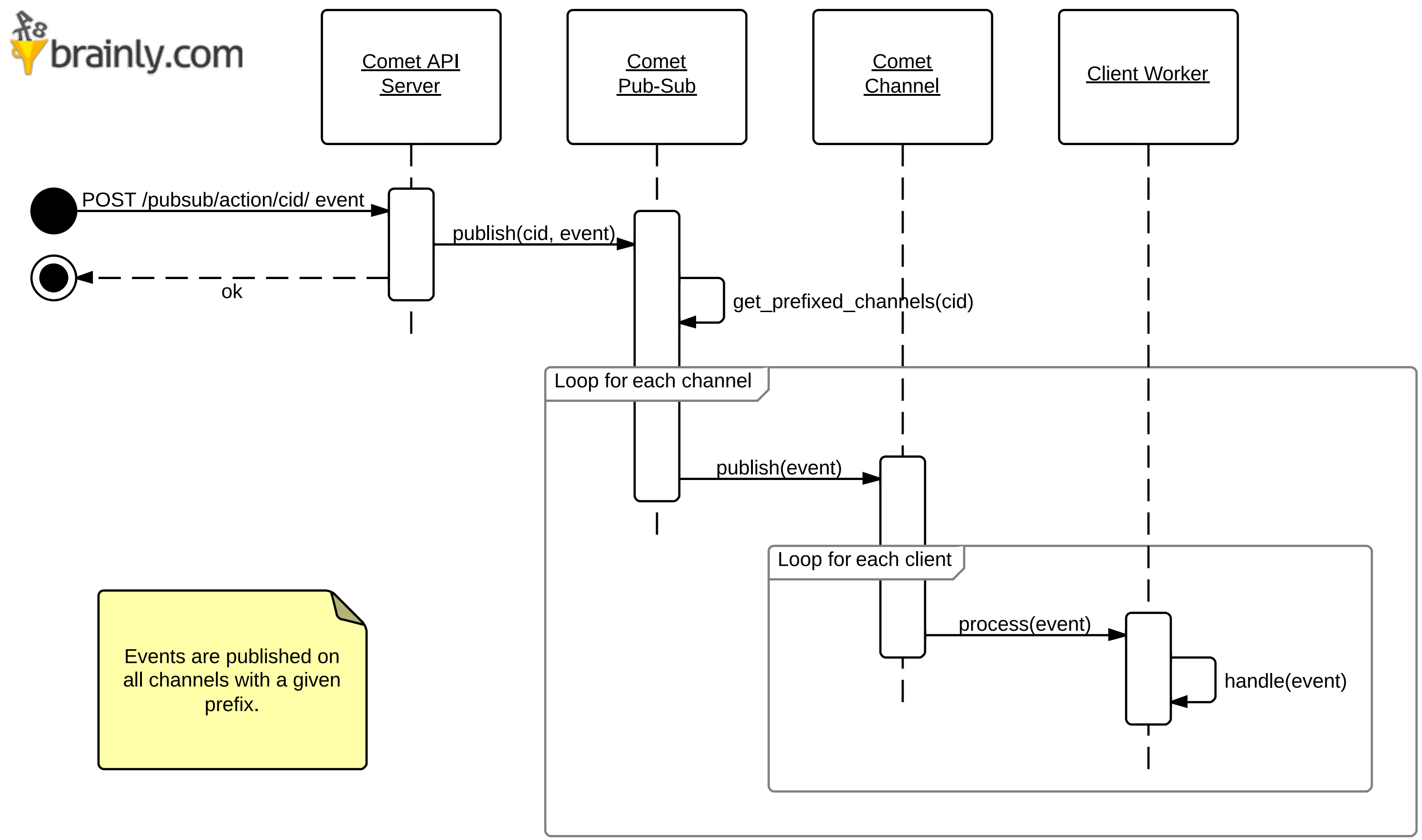
- Hive API Server subscription:



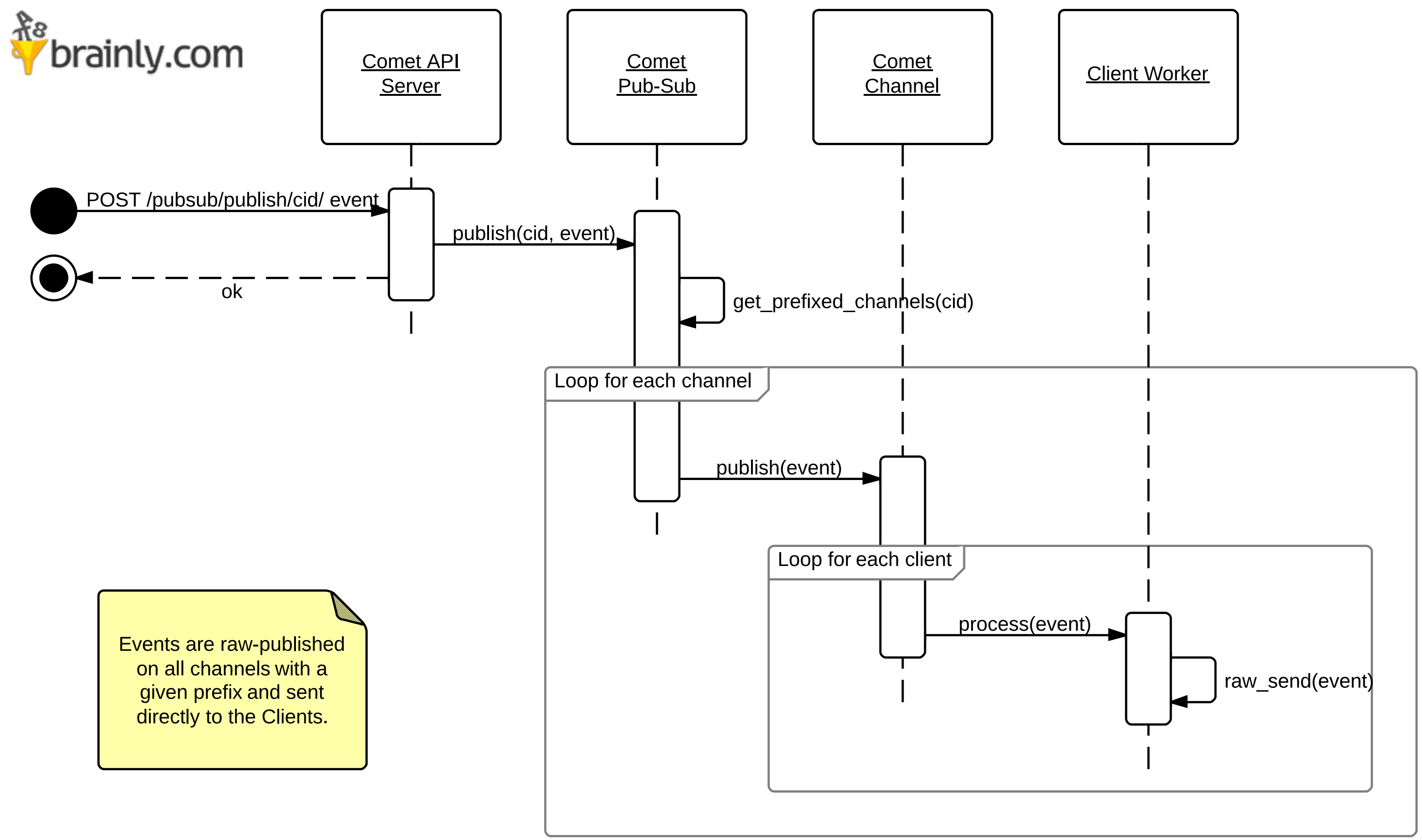
- Hive API Server unsubscription:



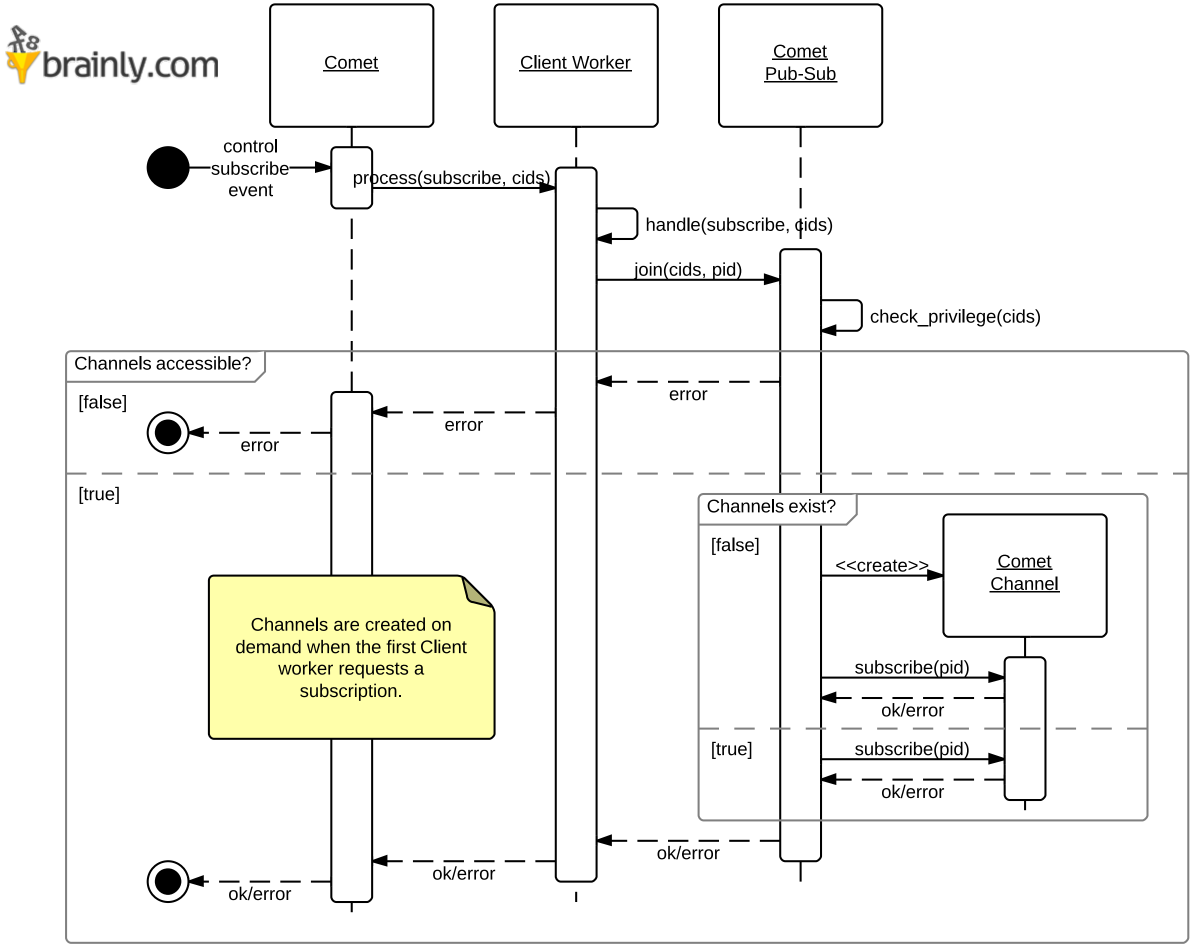
- Hive API Server publishing:



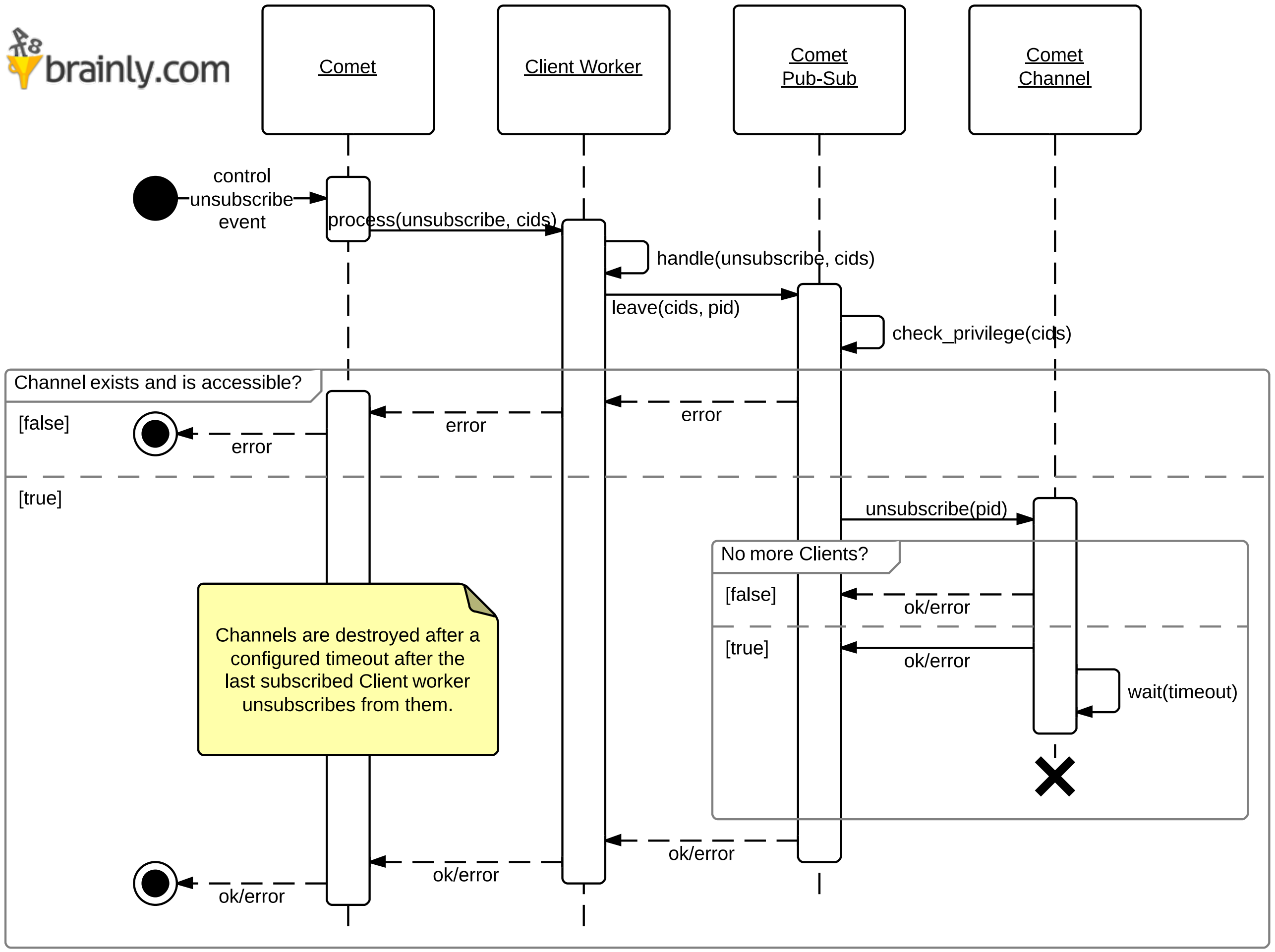
- Hive API Server raw-publishing:



- Hive Client Worker subscription:



- Hive Client worker unsubscription:



7.4 Non-Erlang modules

The Hive server may be combined with a number of modules written in non-Erlang programming languages for convenience and easy integration with existing backends. All that is needed is a little Hive Protocol conformance on the backend side.

7.4.1 Hive Protocol basics

The **Hive Protocol** is a set of communication rules and data formats that are recognized by the Hive server. It involves three components:

- **Internal Events** & **Client Metadata JSON** - described in detail here and here.
- **Hive Protocol hook** - described in detail here.
- **Hive API** - described in detail previously.

Hive Protocol communication is either **simplex** or **duplex**:

- Simplex communication on the Hive-Backend path results in Client Metadata being sent to the Backend.
- Simplex communication on the Backend-Hive path results in Internal Events being dispatched on vaia Hive API calls.
- Duplex communication on the Hive-Backend path results in Client Metadata being sent to the Backend and Internal Events being received from the Backend as a response. This is the only duplex communication in the Hive Protocol.

An example of a Hive Protocol based communication is presented below:

```
// Browser Client sends an event:
{
    "name" : "ping",
    "args" : []
}

// Hive Worker dispatches on the event
// using ping hook configured as follows:
"ping" : [
    {
        "hook" : "hp.post",
        "args" : {
            "endpoint" : "ping.php",
```

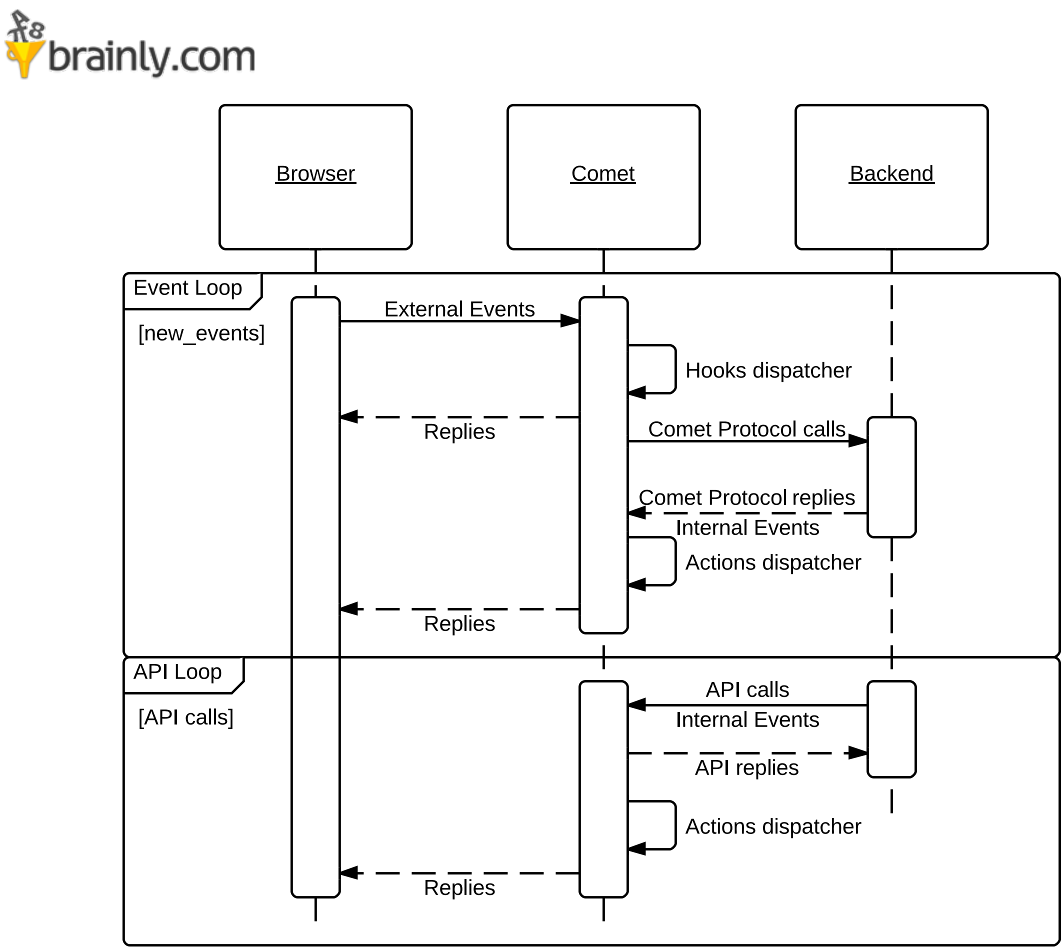
```
            "connector" : "php_backend"
        }
    }
]

// A Client Metadata JSON is passed to the backend via
// php_backend connector in a POST request:
{
    "sid" : "client_id",
    "trigger" : {
        "name" : "ping",
        "args" : []
    },
    "trigger_id" : "",
    "state" : {
        "initial_value" : null
    }
}

// The backend processes the request and generates a reply:
{
    "action" : "reply",
    "args" : {
        "name" : "pong",
        "args" : ["one"]
    }
}

// The backend proceeds to send more events via
// a request to Hive API /client/action/client_id endpoint:
{
    "action" : "reply",
    "args" : {
        "name" : "pong",
        "args" : ["two"]
    }
}

// The Hive API passes the event to the Client Worker.

// The Client worker dispatches on the replies
// using reply dispatcher configured as follows:
"reply" : [
    {
        "action" : "action.send_event",
        "args" : null
    }
]

// Both replies are sent to the Browser client:
{
    "name" : "pong",
    "args" : ["one"]
},
{
    "name" : "pong",
    "args" : ["two"]
}
```

7.4.2 Non-Erlang modules summary

The Hive-Backend interaction is somewhat simplistic. It uses all of the major built-it Hive facilities such as the Hive Hooks, Hive Internal Event dispatchers and Hive API servers and can be divided into two itertwinded flows that run simultaneously

- **the Event loop -** this is a reactive event loop where **external events** originating from the Clients are dispatched resulting in **Hive Protocol Hooks** invocations and Hive-Backend communication.
- **the API loop -** this is a proactive loop where **API calls** originating from the Backend are handled what may result in a number of **Internal Event dispatchers** invocations and Hive-Browser communication.

The Hive-Backend interaction has been summarized on the following diagram:



8 Clusterization

Hive can be configured to run in a cluster, where various messages are propagated between several nodes to achieve some transparency of operation - for example, Pub-Sub channels, even though distributed between several nodes, will act as a single facility, transparently forwarding messages to any connected clients. The following subsections sketch Hive clusterization and give some hints on Cluster naming convention and internal communication.

8.1 Clusterization basics

In order to clusterize Hive you have to configure relevant parameters and run several instances of the server. Everything else will be handled for you automatically. Instances need to know about at least one of currently live cluster nodes in order to successfully join a cluster.

Every node in a Hive cluster is of the same rank, there is no distinction into master or slave nodes. This means that the cluster can be easily resized at run-time simply by starting new Hive instances which are configured to connect to any of the already running nodes.

Each newly connected node will automatically start receiving all necessary messages comming from the rest of the Cluster and it'll act accordingly.

8.2 Cluster naming convention

Each Hive node has to be given a name in order to successfully form or join a cluster. The node name syntax is as follows:

```
IDPart [ '@' ( HostPart '.' DomainPart | IPAddress ) ]
```

The address part names the machine Hive is running on, usually this will be an `IPAddress`, but a **fully qualified domain name** can be used. The address part is optional in the configuration file - when left out, Hive will attempt to infere it from your hosts configuration. If inference is successful, the `HostPart` will contain the host name of your machine, and the `DomainPart` will contain the domain associated with your machines IP address.

The `IDPart` is a short identificator of the particular Hive instance, it doesn't have to be unique as long as the entire name is unique.

Similarily, the Hive Cluster has to be given a name in order for each Hive instance to identify other cluster members. The cluster name is largely arbitrary as long as it's a non-empty string.

8.3 Cluster-wide communication

Currently, Hive nodes propagate several types of messages throughout the Hive Cluster:

- any Internal Events routed using the Hive Router,
- any Internal Events routed using the Hive Pub-Sub facility - but only on the Pub-Sub Manager level, each Pub-Sub Channel propagates messages to the Clients locally.

Additionally, Hive nodes will broadcast some messages in order to maintain the Cluster structure. These include:

- heartbeats,
- status updates,
- connection/disconnection notifications.

9 Hive Plugins

Hive Plugins framework is an elaborate way to define extentions for the Hive server. Detailed description of the Hive Plugins framework is outside of the scope of this document. This section describes the Hive Plugins framework and gives a brief description of the available built-in plugins.

9.1 Plugins basics

Hive Plugins reside in the **plugins** directory and are written in the Erlang programming language. Plugins are *loosely* structured and do not have to reside in a single module, however plugins must follow several conventions:
The **type convention** - Hive server supports currently for types of plugins that differ in their API:

- **State Managers** - manage the state of the Hive Workers,
- **Hooks** - allow the Hive Workers to perform some actions,
- **Internal Events** - allow the backend to control the Hive Workers,
- **Service Connectors** - provide the means to connect to various services.

The **API convention** - Hive server defines a common Hive Plugin API in addition to the specific plugin-type API that allows for an automated plugins loading during Hive startup:

- **load** - loads plugins from a plugin-module and performs all necessary initial checks; there may be several plugins defined in a single plugin module,
- **validate** - validates plugin configuration and performs all additional checks,
- **unload** - unloads a plugin and performs all necessary cleanup.

And least importantly, the **naming convention** - plugins once loaded reside in a common namespace. Some care must be taken to avoid name clashes. Generally, the following naming convention is used:

- **sm.plugin** - names a State Manager,
- **purpose.plugin** - names a Hook that serves some **purpose**,
- **action.plugin** - names an Internal Event dispatcher,
- **connector.plugin** - names a Service Connector.

Every plugin expects to receive a configuration descriptor on initialization. The descriptor should be of the following format ("type" is either `connector`, `state_manager`, `hook` or `event` depending on the plugin type):

```
{
    "type" : "plugin",
    "args" : "Additional arguments",
    "additional" : "fields",
    ...
}
```

The built-in plugins are described shortly in the following subsections.

9.2 Client Hooks

9.2.1 `utils.echo`

Echoes any received Socket.IO messages back to the Client.

- metadata used - none,
- arguments taken - either none or **an external event** to reply with,
- modifications to the state - none,
- results in - **always replies** with a Socket.IO message.

Example declarations:

```
{ "hook" : "utils.echo", "args" : null }
{ "hook" : "utils.echo", "args" : { "name" : "event", "args" : "arguments" } }
```

9.2.2 `utils.console_dump`

Dumps the state of a Client worker to the console together with a distinct message passed as an argument. Meant for debugging purposes.

- metadata used - the Client workers internal state (Session ID, stored values, event that triggered the hook),
- arguments taken - a short message to be printed in the console; has to be **a string**,
- modifications to the state - none (but accesses the stored values),
- results in - **never replies nor stops**.

Example declaration:

```
{ "hook" : "utils.console_dump", "args" : "Hello Hive!" }
```

9.2.3 `utils.dispatch`

Dispatches on the triggering external event as if it were an interternal one.

- metadata used - the Client workers internal state (Session ID, stored values, event that triggered the hook),
- arguments taken - none,
- modifications to the state - may modify anything as it treats the triggering event as an internal event,
- results in - might return either **reply**, **no reply** or even stop the Client worker.

Example declaration:

```
{ "hook" : "utils.dispatch", "args" : null }
```

9.2.4 `pubsub.subscribe`

Subscribes a Client to the Hive Pub-Sub channels specified in arguments of the triggering event.

- metadata used - the Client workers internal state (Session ID, stored values, event that triggered the hook),
- arguments taken - **a string** representing the privilege level of this hook,
- modifications to the state - none,
- results in - does **not reply**, might stop the Client worker in case of a failure.

Example declaration:

```
{ "hook" : "pubsub.subscribe", "args" : "public" }
```

Example triggering event (causes subscription to channels `foo.bar.baz`, `faz.baz` and `fez`):

```
{
    "name" : "subscribe",
    "args" : [
        {
            "foo" : {
                "bar" : ["baz"]
            },
            "faz" : "baz"
        },
        "fez"
    ]
}
```

9.2.5 `pubsub.resubscribe`

First attempts to leave all subscribed channels matching prefixes given in the triggering event and then resubscribes the Client to all specified, concrete channel IDs.

- metadata used - the Client workers internal state (Session ID, stored values, event that triggered the hook),
- arguments taken - **a string** representing the privilege level of this hook,
- modifications to the state - none,
- results in - does **not reply**, might stop the Client worker in case of a failure.

Example declaration:

```
{ "hook" : "pubsub.resubscribe", "args" : "public" }
```

Example triggering event (causes subscription to channels `foo.bar.baz`, `faz.baz` and `fez`):

```
{
    "name" : "resubscribe",
    "args" : [
        {
            "prefxi_1" : {
                "bar" : ["baz"]
            },
            "prefix_2" : "baz"
        }
    ]
}
```

9.2.6 `pubsub.unsubscribe`

Unsubscribes a Client from the Hive Pub-Sub channels specified in arguments of the triggering event.

- metadata used - the Client workers internal state (Session ID, stored values, event that triggered the hook),
- arguments taken - **a string** representing the privilege level of this hook,
- modifications to the state - none,
- results in - does **not reply**, might stop the Client worker in case of a failure.

Example declaration:

```
{ "hook" : "pubsub.unsubscribe", "args" : "public" }
```

Example triggering event (causes unsubscription from channels `foo.bar.baz`, `faz.baz` and `fez`):

```
{
    "name" : "subscribe",
    "args" : [
        {
            "foo" : {
                "bar" : ["baz"]
            },
            "faz" : "baz"
        },
        "fez"
    ]
}
```

9.2.7 `pubsub.publish`

Publishes internal events passed as the arguments of the triggering event on a Hive Pub-Sub channels specified in arguments.

- metadata used - the Client workers internal state (Session ID, stored values, event that triggered the hook),
- arguments taken - **a JSON object** containing the Hive Pub-Sub channel IDs and the privilege level of this hook,
- modifications to the state - none,
- results in - does **not reply**, might stop the Client worker in case of a failure.

Example declaration:

```
{
    "hook" : "pubsub.publish",
    "args" : {
        "cids" : ["foo", "bar"],
        "privilege" : "public"
    }
}
```

Example triggering event (causes the Client to publish events on Hive Pub-Sub channels `foo` and `bar`):

```
{
    "name" : "publish",
    "args" : [
        { "action" : "event1", "args" : "arguments" },
        ...
    ]
}
```

9.2.8 `hp.get`

Performs a synchronous request, similar in semantics to an HTTP GET, to a given endpoint using a given Hive Protocol compatible Hive Connector pool (listed here).

- metadata used - none,
- arguments taken - **a JSON object** that contains: **endpoint** - a string representing the endpoint, **connector** - name of the Hive Protocol compatible Hive Connectors pool to use.
- modifications to the state - none,
- results in - dispatches the reply immediately and may **not reply**, **reply**, **error** or even **stop** the Client worker.

Example declaration:

```
{
    "hook" : "hp.get",
    "args" : {
        "endpoint" : "/",
        "connector" : "connector1"
    }
}
```

9.2.9 `hp.put`

Performs an asynchronous request, similar in semantics to an HTTP PUT, to a given endpoint using a given Hive Protocol compatible Hive Connector pool (listed here).

- metadata used - the Client workers internal state (Session ID, stored values, event that triggered the hook),
- arguments taken - **a JSON object** that contains: **endpoint** - a string representing the endpoint, **connector** - name of the Hive Connectors pool to use,
- modifications to the state - none,
- results in - does not wait for a reply, but might **error** or **stop** the Client in case of connection errors in the underlying Hive Connector.

Example declaration:

```
{
    "hook" : "hp.put",
    "args" : {
        "endpoint" : "/",
        "connector" : "connector1"
    }
}
```

9.2.10 `hp.post`

Performs a synchronous request, similar in semantics to an HTTP POST, to a given endpoint using a given Hive Protocol compatible Hive Connector pool (listed here).

- metadata used - the Client workers internal state (Session ID, stored values, event that triggered the hook),
- arguments taken - **a JSON object** that contains: **endpoint** - a string representing the endpoint, **connector** - name of the Hive Connectors pool to use,
- modifications to the state - none,
- results in - transfers the Client metadata and awaits a reply which it then treats as an **internal event**; may result in either **reply**, **no reply**, **error** or even **stop**.

Example declaration:

```
{
    "hook" : "hp.post",
    "args" : {
        "endpoint" : "/",
        "connector" : "connector1"
    }
}
```

9.3 Internal Events

Internal events are used to control the Hive server. These events are generated via the Hive API Server requests or received as a response to a hook action invoked earlier. Internal events must conform to the following format:

```
{
    "action" : "action_id",
    "trigger_id" : "optional ACK ID"
    "args" : "action arguments"
}
```

The `trigger_id` field is optional, and defaults to the empty string. It is interpreted differently by different actions.
Currently supported actions are listed below.

9.3.1 `action.stop`

Stops the Client worker. `args` is **a string** with a short termination reason. Example event:

```
{
    "action" : "stop_client",
    "args" : "Rage-quit"
}
```

Example declaration:

```
"stop_client" : [
    {
        "action" : "action.stop",
        "args" : null
    }
]
```

9.3.2 `action.error`

Signalizes an error to the Client worker. `args` is **a string** with a short error description. The error is sent to the Client as oon as possible. Example event:

```
{
    "action" : "signalize_error",
    "args" : "Core melting!"
}
```

Example declaration:

```
"signalize_error" : [
    {
        "action" : "action.error",
        "args" : null
    }
]
```

9.3.3 `action.send_event`, `action.send_message`, `action.send_json`, `action.send_ack`

Replies to the Client using event/message/json/ack Socket.IO message types. `args` is **a JSON encoded reply**. The `trigger_id` field is used as follows: `action.send_ack` will append it to the payload and send it to the client as an actual acknowledgement; other actions will send it via the ACK ID field of the Socket.IO message protocol.
Example event:

```
// message action:
{
    "action" : "message",
    "args" : "About to rage-quit..."
}

// event action:
{
    "action" : "event",
    "trigger_id" : "23+",
    "args" : {
        "name" : "ping",
        "args" : ["pong"]
    }
}
```

Example declarations:

```
// message declaration:
"message" : [
    {
        "action" : "action.send_message",
        "args" : null
    }
],

// Event declaration:
"event" : [
    {
        "action" : "action.send_event",
        "args" : null
    }
]
```

9.3.4 action.update_state

Updates the internal Client worker state via its State Manager. Assumes `args` to be **a JSON object**, and sets each field-name key to its corresponding value in the internal state. Example event:

```
{
    "action" : "store",
    "args" : {
        "field_1" : "value_1",
        "field_2" : [1, 2, 3, 4, 5],
        "field_3" : null
    }
}
```

Example declaration:

```
"store" : [
    {
        "action" : "action.update_state",
        "args" : null
    }
]
```

9.3.5 action.dispatch

Dispatches the event passed in `args` and treats it as an external event causing associated hooks to trigger. Assumes `args` to be a valid Socket.IO event. Example event:

```
{
    "action" : "dispatch",
    "args" : {
        "name" : "ping",
        "args" : ["pong"]
    }
}
```

Example declaration:

```
"dispatch" : [
    {
        "action" : "action.dispatch",
        "args" : null
    }
]
```

9.3.6 `action.add_hooks`

Dynamically validates, initializes and adds hooks to the Client worker hooks. Assumes `args` to be **a JavaScript array** of hook descriptors (same as described before). Example event:

```
{
    "action" : "add_hooks",
    "args" : {
        "event_1" : [
            { "hook" : "hook_1", "args" : null },
            { "hook" : "hook_2", "args" : [1, 2, 3] },
            ...
        ],
        ...
    }
}
```

Example declaration:

```
"add_hooks" : [
    {
        "action" : "action.add_hooks",
        "args" : null
    }
]
```

9.3.7 `action.remove_hooks`

Removes hooks from the Client worker. `args` is assumed to be **a JavaScript array** of event names which hooks should be removed. Example event:

```
{
    "action" : "remove_hooks",
    "args" : ["event_1", "event_2", ...]
}
```

Example declaration:

```
"remove_hooks" : [
    {
        "action" : "action.remove_hooks",
        "args" : null
    }
]
```

9.3.8 `action.start_connectors`

Starts Hive Connector pools. Assumes `args` to be **a JSON object** representing name/Connector descriptor pairs (the same as described before). Example event:

```
{
    "action" : "start_connector",
    "args" : {
        "bar_fo" : {
            "worker_module" : "module1",
            "size" : 10,
            "overflow" 23,
            "args" : ...
        },
        "foo_bar" : {
            "worker_module" : "module2",
            "size" : 10,
            "overflow" 23,
```

```
            "args" : ...
        }
    }
}
```

Example declaration:

```
"start_connectors" : [
    {
        "action" : "action.start_connectors",
        "args" : null
    }
]
```

9.3.9 action.stop_connectors

Stops Hive Connector pools. Assumes `args` to be **a JavaScript array** of Connector pool names. Example event:

```
{
    "action" : "stop_connectors",
    "args" : ["foo_bar", "baz_foo"]
}
```

Example declaration:

```
"stop_connectors" : [
    {
        "action" : "action.stop_connectors",
        "args" : null
    }
]
```

9.3.10 pubsub.publish

Publishes an External Event to a list of Hive PubSub channel IDs. Example event:

```
{
    "action" : "publish",
    "args" : {
        "event" : "an External Event to publish",
        "cids" : {
            "prefix_1" : [1, 2, ...],
            ...
        }
    }
}
```

Example declaration:

```
"publish" : [
    {
        "action" : "pubsub.publish",
        "args" : "public"
    }
]
```

9.3.11 pubsub.publish_raw

Publishes a raw External Event to a list of Hive PubSub channels. Example event:

```
{
    "action" : "publish_raw",
    "args" : {
        "event" : "an External Event to publish",
        "cids" : {
            "prefix_1" : [1, 2, ...],
            ...
        }
    }
}
```

Example declaration:

```
"publish_raw" : [
    {
        "action" : "pubsub.publish_raw",
        "args" : "public"
    }
]
```

9.3.12 `pubsub.subscribe`

Subscribes the Client to a list of Hive PubSub channels. Example event:

```
{
    "action" : "subscribe",
    "args" : {
        "prefix_1" : [1, 2, ...],
        ...
    }
}
```

Example declaration:

```
"subscribe" : [
    {
        "action" : "pubsub.subscribe",
        "args" : "public"
    }
]
```

9.3.13 `pubsub.unsubscribe`

Unsubscribes the Client from a list of Hive PubSub channels. Example event:

```
{
    "action" : "unsubscribe",
    "args" : {
        "prefix_1" : [1, 2, ...],
        ...
    }
}
```

Example declaration:

```
"unsubscribe" : [
    {
        "action" : "pubsub.unsubscribe",
        "args" : "public"
    }
]
```

9.3.14 `pubsub.resubscribe`

Resubscribes the Client to a list of Hive PubSub channels. Example event:

```
{
    "action" : "resubscribe",
    "args" : {
        "prefix_1" : [1, 2, ...],
        ...
    }
}
```

Example declaration:

```
"resubscribe" : [
    {
        "action" : "pubsub.resubscribe",
        "args" : "public"
    }
]
```

9.4 Service Connectors

Currently available Hive Connector pool types and their arguments are described below:

9.4.1 `connector.redis`

The Redis database connector.

- `connector` - `connector.redis`
- `args` - **a JSON object** containing the following fields:
  - `host` - **a string** representing the host name to connect to,
  - `port` - **a non-negative integer lower than 65536**, the port on which to connect,
  - `database` - **a non-negative integer**, the Redis database index,
  - `password` - **a string**, the Redis database password,
  - `restart_timeout` - **a non-negative integer**, the timeout **in milliseconds** between periodical Connector worker restarts,
  - `reconnect_timeout` - **a non-negative integer**, the timeout **in milliseconds** between reconnect attempts (in case of a connection error).
  - `max_reconnect_timeout` - **a non-negative integer**, the timeout **in milliseconds** after which a connection is assumed to be dead and no further reconnection attempts should be performed.

9.4.2 `connector.http`

A Hive Protocol compatible HTTP connector. Provides Hive Protocol interface over HTTP. All data is transfered in the bodies of the HTTP requests with no additional encapsulation or serialization.

- `connector` - `connector.http`
- `args` - **a JSON object** containing the following fields:
  - `base_url` - **a string** representing the base URL each worker will connect to; the URL should specify the port unless the default port 80 is meant to be used. The specific endpoint will be supplied later, per request.
  - `max_connections` - **a non-negative integer**, the maximal number of simultaneous HTTP keep-alive connections.
  - `max_connection_timeout` - **a non-negative integer**, the timeout **in milliseconds** after which a connection is assumed to be dead.

9.4.3 `connector.tcp`

A Hive Protocol compatible TCP connector. Provides Hive Protocol interface over TCP using a pool of connection listeners and a simple data serialization subprotocol - all data sent to/received from a given endpoint are encoded as Socket.IO messages with the **message** opcode and are preceded by a short header consisting of a frame marker, a string representation of the message length and another frame marker:

```
"\UFFFD", "string representation of the message length", "\UFFFD", "3::", Endpoint, ":", Payload
```

- `connector` - `connector.tcp`
- `args` - **a JSON object** containing the following fields:
  - `port` - **a non-negative integer lower than 65536**, the port on which to listen (`listening` set to `true`) or connect (`listening` set to `false`),
  - `restart_timeout` - **a non-negative integer**, the timeout **in milliseconds** between periodical Connector worker restarts,
  - `max_connection_timeout` - **a non-negative integer**, the timeout **in milliseconds** after which a connection is assumed to be dead.

9.5 State Managers

This section lists and describes all available Hive Client State Managers.

9.5.1 `sm.local`

The Local State Manager **does not require any additional arguments**. It stores the Client workers state internally within the Client worker process, **decreasing access time** and **increasing memory use**.
Example Local State Manager descriptor:

```
{
    "state_manager" : "sm.local",
    "initial_value" : "value",
    "args" : null
}
```

9.5.2 `sm.redis`

The Redis State Manager stores the Client workers state in a remote Redis database. Client workers' state is stored as a Redis hash map (accessed via `HGET` and `HSET`) with non-trivial objects serialized to JSON format (integers, strings and other simple values are stored as-is while JavaScript arrays and objects are serialized and stored as strings). The hash map is accessible under a key equal to the **Session ID** of the Client session. Additionally a Client Worker-side cacheing is employed to limit unnecessary Redis database requests. Arguments received in `args`:

- `connector` - the Redis Connector pool to use to access the database; must be **a string** naming a valid Hive Connector pool,
- `expiration_timeout` - an expiration timeout (**in milliseconds**) of the data stored in the database; **optional**, but must be **a non-negative integer** if defined; the default value is 48 hours.

Example Redis State Manager descriptor:

```
{
    "state_manager" : "sm.redis",
    "initial_value" : "value",
    "args" : {
        "connector" : "database",
        "expiration_timeout" : 3600
    }
}
```