Yats - Paolo Lulli
Generated on: 12/12/2024

YATS Manual

Yats Server

Installing YATS Server

Building the sources

For the Community Edition the sources are available from:

```
$ git clone https://code.kevwe.com/git/yats.git
```

In order to build the server

```
cd server
go build
```

The server executable is named **yats-server**

Service Configuration

You can start different named instances of the program. An instance named ‘default’ is expected to read a configuration file from:

```
 ~/.yats/default.json
```

where the content is something like :

```
{
  "databaseUsername": "cassandra",
  "databasePassword": "cassandra-password",
  "databasePort": "9042",
  "databaseHost": "127.0.0.1",
  "databaseSchema": "yats",
  "restAddress": "127.0.0.1:18081",
  "grpcAddress": ":50051",
  "tcpAddress": ":1025",
  "archiveFrequency": "weekly",
  "archiveDirectory": "/home/goodstuff",
  "archiveNode": "true"
}
```

Starting the server

```
cd yats
./script/service.sh start
```

Stop the Server

```
cd /opt/ows/service
./script/service.sh stop
```

YATS as a system Service

In order to keep the service active after a reboot, Systemd can be used in Linux with a configuration like the following:

```
# cat /etc/systemd/system/yats-service.service
[Unit]
Description=YATS Server process
After=network-online.target
[Service]
Type=exec
ExecStart=/path/to/yats/script/service.sh default start
ExecStop=/path/to/yats/script/service.sh default stop
Restart=on-failure
[Install]
WantedBy=multi-user.target
```

Yats Client

Installing YATS Client

Building the sources

For the Community Edition the sources are available from:

```
$ git clone https://code.kevwe.com/git/yats.git
```

In order to build the client

```
cd client
go build
```

The client executable is named **yats**

Access to the help

You can easily access the manual page by typing in the terminal:

```
$ ./yats --help
```

This will print-out the full list of commandline options

```
Usage of ./yats:
  -b, --base64-decode           decode base64 output
      --date                    midnight of day <YYYYMMDD> at GMT
      --day                     days ago
  -f, --format                  format output
      --from int                from tstamp
      --hour                    hours ago
  -a, --latitude float          latitude
  -e, --list-events             list events
  -m, --list-metrics string     list metrics with name
  -p, --list-positions string   list positions with name
  -o, --longitude float         longitude
  -v, --metric-value string     metric value
      --min                     minutes ago
      --pki-init                create CSR
      --sec                     seconds ago
  -S, --show-permissions        show user permissions
  -s, --source string           source application
      --timestamp int           timeStamp
      --to int                  to tstamp
  -u, --unpack-json             unpack base64/json output
  -l, --value-only              log value only output
  -E, --write-event string      write event with name
  -M, --write-metric string     write metric with name
  -P, --write-position string   write position name
      --year                    years ago
```

A guided walk-through

Yats client allows to manage 3 kind of entities from the commandline:

- Events
- Metrics
- Positions

Writing Entities

All options have a long version, that is rather self-explanatory, and a shorter version for better ergonomics. For you to remember easily, short commandline switches to **write** entities are uppercase in order to stand out.

Writing an Event from commandline

Event is the simplest entity in YATS. An event in YATS is any conventional string message that has a meaning in itself. It can be, for example a **systemSTART** in the case of some sort of service that you want to track. You could have in this case **systemSTART** and **systemSTOP** message.

With the client, you can register those events as follows

```
 $ yats -E systemSTART
payload: { "name":"systemSTART" }{"ret":"OK"}
```

```
.
.
$ yats -E systemSTOP
payload: { "name":"systemSTOP" }{"ret":"OK"}paolo@phoenix ~/dev/simple/yats/client {master} $
```

Instead of the **-E** switch, it can be called like this:

```
yats --write-event <event-name>
```

The recorded story for those events will look something like this in the Cassandra table:

```
cas@cqlsh:yats> SELECT * from event where id_client='myclientname';

 id_client             | etime                           | name
-----------------------+---------------------------------+-------------------
 myclientname          | 2024-11-03 18:00:48.159000+0000 |       systemSTART
 myclientname          | 2024-11-03 18:00:52.681000+0000 |        systemSTOP
```

Writing Metric from commandline

Metric in Yats is any kind of measure that you want to record. It can be a single sensor read, or it can contain a data structure with more reads.

A Metric is made by a name and a value. The name is used to distinguish the meaning of the data, while the value is the actual payload.

For example, you can store the registered temperature in a greenhouse with:

```
$ yats --write-metric greenhouseDegrees --metric-value 32.3
```

or, using the shorthands:

```
 $ yats -M greenhouseDegrees -v 32.3
```

Different writes of this measure would produce a time series that looks like this:

```
cas@cqlsh:yats> SELECT * from metric where id_client='myclientname' and name='greenhouseDegrees';

 id_client             | name              | mtime                           | value
-----------------------+-------------------+---------------------------------+-------
 myclientname          | greenhouseDegrees | 2024-11-03 18:19:35.084000+0000 |  32.3
 myclientname          | greenhouseDegrees | 2024-11-03 18:21:27.632000+0000 |  32.3
 myclientname          | greenhouseDegrees | 2024-11-03 18:23:01.831000+0000 |    32
 myclientname          | greenhouseDegrees | 2024-11-03 18:23:06.232000+0000 |    31
 myclientname          | greenhouseDegrees | 2024-11-03 18:23:09.039000+0000 |    30
 myclientname          | greenhouseDegrees | 2024-11-03 18:23:13.863000+0000 |    27
```

Writing Position from commandline

Position in YATS works as a special type of measure, as such it has its own switches:

```
  -P, --write-position string   write position name
```

and:

```
  -a, --latitude float          latitude
  -o, --longitude float         longitude
```

As such an example write could be the following:

```
$ yats -P myposition --latitude 41.890251 --longitude 12.492373
```

Similarly to metrics, the data stored will look something like this:

```
cas@cqlsh:yats> SELECT * from position where id_client='myclientname';

 id_client             | ptime                           | lat      | lon      | name
-----------------------+---------------------------------+----------+----------+------------
 myclientname          | 2024-11-03 18:30:26.630000+0000 | 41.89025 | 12.49237 | myposition
 .
 .
```

Reading Data Series

All read operations on dataseries will require you to specify somehow a time interval, by means of a **FROM** and a **TO** clause.

A **FROM** clause is always recommended, as otherwise queries will start returning results from time 0 UTC @ 0GMT. In some cases, omitting the **TO** clause can makes sense in order to retrieve all results up to the latest recorded point in time.

Event Data Series

Where not differently specified, the input timestamp are interpreted as seconds in UTC time recorded in the timezone @ 0GMT

For example, to list all the events for the current client from **1728901943** corresponding to: **Mon 14 Oct 12:32:23 CEST 2024** you can issue the following command:

```
$ yats -e --from 1728901943
```

The result is not really pretty-printed, it can be if you have **jq** installed, like this:

```
$ yats -e --from 1728901943 | jq
```

which will give you a formatted output:

```
{
  "data": [
    {
      "id_client": "myclientname",
      "etime": 1728901943952,
      "name": "systemSTART"
    },
    {
      "id_client": "myclientname",
      "etime": 1730656600787,
      "name": "systemSTOP"
    },
    {
      "id_client": "myclientname",
      "etime": 1730656848159,
      "name": "systemSTART"
    },
    {
      "id_client": "myclientname",
      "etime": 1730656852681,
      "name": "systemSTOP"
    }
  ],
  "maxpage": 1730656852681
}
```

Differently from the input, the timestamps in the response have the definition of milliseconds.

Metric Data Series

Differently from Event, retrieving a Metric serie for a specific client will require you to specify the name of the metric as a parameter, like this:

```
$ yats -m greenhouseDegrees  --from 1728901943
```

which produces:

```
1730657975084 greenhouseDegrees=32.3
1730658087632 greenhouseDegrees=32.3
1730658181831 greenhouseDegrees=32
1730658186232 greenhouseDegrees=31
1730658189039 greenhouseDegrees=30
1730658193863 greenhouseDegrees=27
```

Clearly defining time intervals in terms of UTC seconds is not always the most confortable choice. Therefore it’s possible to specify the time in a different format. For example, you can specify the relative time from the current moment in minutes like this:

```
$ yats -m greenhouseDegrees  --from 60 --min
```

Or, using hours as unit of measure:

```
yats -m greenhouseDegrees  --from 1 --hour
```

both commands will produce the same output:

```
1730657975084 greenhouseDegrees=32.3
1730658087632 greenhouseDegrees=32.3
1730658181831 greenhouseDegrees=32
1730658186232 greenhouseDegrees=31
1730658189039 greenhouseDegrees=30
1730658193863 greenhouseDegrees=27
```

At the same way, seconds **–sec** , days **–day** , years **–year** can be used.

Last, but not the least, a more practical **YYYYMMDD** date format can be specified with the **–date** parameter.

See for example:

```
$ yats -m greenhouseDegrees  --date 20241103
1730657975084 greenhouseDegrees=32.3
1730658087632 greenhouseDegrees=32.3
1730658181831 greenhouseDegrees=32
1730658186232 greenhouseDegrees=31
1730658189039 greenhouseDegrees=30
1730658193863 greenhouseDegrees=27
```

Structured Data

Data in Metric can be either a single measure, or a structured format, typically encoded with Base64 for safe storage.

One use-case for simple Base64-encoded text is application logging.

For example, you could have logged something like this:

```
cas@cqlsh:yats> SELECT * from metric where id_client='myclientname' and name='some-log';

 id_client             | name        | mtime                           | value
-----------------------+-------------+---------------------------------+--------------------------
--------------------
 myclientname          | some-log    | 2024-11-03 19:54:04.000000+0000 |
VGhpcyBzZXJ2aWNlIGdhdmUgZXJyb3IgY29kZSBYWVoK
 myclientname          | some-log    | 2024-11-03 19:54:13.000000+0000 |
RXZlcnl0aGluZyBpcyBmaW5lCg==
```

The relevant information here is base64-encoded in the value column. From the yats client, the following query:

```
 $ yats -m some-log --from 1730663612
```

would produce the following output:

```
1730663644000 some-log=VGhpcyBzZXJ2aWNlIGdhdmUgZXJyb3IgY29kZSBYWVoK
1730663653000 some-log=RXZlcnl0aGluZyBpcyBmaW5lCg==
```

which, depending on how you are consuming the information, might or might not be something useful.

If you are only interested in the Base64 portion, you can print that with the **-l** switch, like this:

```
$ yats -m some-log --from 1730663612 -l
```

which gets you:

```
VGhpcyBzZXJ2aWNlIGdhdmUgZXJyb3IgY29kZSBYWVoK
RXZlcnl0aGluZyBpcyBmaW5lCg==
```

Or, you can get the decoded Base64 directly with the combination of switches: **-lb** like this:

```
$ yats -m some-log --from 1730663612 -lb
```

which prints out in this case the text logs:

```
- This service gave error code XYZ
- Everything is fine
```

The encoded base64 text could be a Json message, in which case another helper switch is provided: **-u** The use is described in an example below.

Let’s say that you have the following data saved in the backend:

```
cas@cqlsh:yats> SELECT * from metric where id_client='myclientname' and name='some-log';

 id_client    | name     | mtime                           | value
--------------+----------+---------------------------------+--------------------------------------
----------------------------------------
 myclientname | some-log | 2024-11-03 20:13:22.000000+0000 |
eyJrMSI6InYxIiwibWVzc2FnZSI6ImZpcnN0IG1lc3NhZ2UiLCJ0cyI6IjE3MzA2NjQ4MDIifQo=
 myclientname | some-log | 2024-11-03 20:13:26.000000+0000 |
eyJrMSI6InYxIiwibWVzc2FnZSI6InNlY29uZCBtZXNzYWdlIiwidHMiOiIxNzMwNjY0ODA2In0K
 myclientname | some-log | 2024-11-03 20:13:31.000000+0000 |
eyJrMSI6InYxIiwibWVzc2FnZSI6InRoaXJkIG1lc3NhZ2UiLCJ0cyI6IjE3MzA2NjQ4MTEifQo=
```

the literal value of the logs can be printed out by doing:

```
$ yats -m some-log --from 1730664802
1730664806000 some-
log=eyJrMSI6InYxIiwibWVzc2FnZSI6InNlY29uZCBtZXNzYWdlIiwidHMiOiIxNzMwNjY0ODA2In0K
1730664811000 some-
log=eyJrMSI6InYxIiwibWVzc2FnZSI6InRoaXJkIG1lc3NhZ2UiLCJ0cyI6IjE3MzA2NjQ4MTEifQo=
```

which is, of course, not very readable. Base64-decoding looks like this, instead:

```
$ yats -m some-log --from 1730664802  -lb
{"k1":"v1","message":"second message","ts":"1730664806"}
{"k1":"v1","message":"third message","ts":"1730664811"}
```

Here the payload is printed in it json form. In fact you can verify that:

```
eyJrMSI6InYxIiwibWVzc2FnZSI6InNlY29uZCBtZXNzYWdlIiwidHMiOiIxNzMwNjY0ODA2In0K
```

is the same as:

```
{"k1":"v1","message":"second message","ts":"1730664806"}
```

You can use the following command to check that:

```
$ echo "eyJrMSI6InYxIiwibWVzc2FnZSI6InNlY29uZCBtZXNzYWdlIiwidHMiOiIxNzMwNjY0ODA2In0K" | base64 -d
{"k1":"v1","message":"second message","ts":"1730664806"}
```

Now, when you are querying for such json logs in the form of Base64 text, rather than having a stack of strings like the following:

```
$ yats -m some-log --from 1730664802  -lb
{"k1":"v1","message":"second message","ts":"1730664806"}
{"k1":"v1","message":"third message","ts":"1730664811"}
{"k1":"v1","message":"fourth message","ts":"1730738703"}
{"k1":"v1","message":"fifth message","ts":"1730738809"}
```

it can be convenient if the json payload is formatted as rows and column in a CSV representation. You con obtain this visualization like this:

```
 $ yats -m some-log --from 1730664802  -u
"timestamp","k1","message","ts";
2024-11-03T21:13:26+01:00,"v1","second message","1730664806";
2024-11-03T21:13:31+01:00,"v1","third message","1730664811";
2024-11-04T17:45:03+01:00,"v1","fourth message","1730738703";
2024-11-04T17:46:49+01:00,"v1","fifth message","1730738809";
```

The output is a CSV that you can save and open as a spreadsheet.

Client Auto-Paging

In analogy with what we have seen for Event, the webservice from YATS server will return with every query an attribute like the following:

```
 "maxpage": 1730656852681
```

This is the maximum value for all the timestamps in a single server response. The server responds to queries with at most maxResults. At the time of this writing, **maxResults=100** but it can possibly change in future, in order to provide maximum efficiency.

The Yats client, when querying for a time-series between **FROM** and **TO** will check if the maxpage in the results is equal to the **TO** clause, and in case it’s not, it will query further, until either the point in time is reached, or there are no more results to retrieve.

Position Data Series

Position data series can be queried specifying time windows exactly as described for Event and Metric.

```
$ yats -p myposition --from 1728901943
```

Building Yats CE from Sources

Requirements

- Golang 1.22
- Git
- make

Downloading the latest sources

```
git clone https://code.kevwe.com/git/yats.git
```

Building Yats Server

```
cd yats/server
go build
```

Building Yats Client

```
cd yats/client
go build
```

Building All there is to it

```
make
```

CrossCompile for different architectures

Yats is implemented in Go and tested in a Linux environment. It can be run in any combinations of OS and architecture that Go supports, therefore you should be able to just use the *GOOS* and *GOARCH* environment variables for the purpose.

For example, you can build the project for Arm while running on *x86_64*. The following is the correct command for compiling on the popular board RaspberryPi:

```
GOOS=linux GOARCH=arm go build
```

Or, if you are on Linux, you might want Windows executables for some reason:

```
GOOS=windows GOARCH=amd64 go build
```

License

The code is freely available under the Affero GPL License see: COPYING

Additional commercial support and licensing is available on request. Just issue a support request and mention you are interested in yats

Logging to Yats from Java

Creating a Java Client

```
import jakarta.servlet.http.HttpServletRequest;

import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;
import java.util.Base64;
import java.util.Objects;
import java.util.concurrent.CompletableFuture;

public class YatsLog {
    private final String logName;
    private final String clientId;
    private final String endpoint;
    private static final HttpClient httpClient = HttpClient.newBuilder().build();

    public static YatsLog with(String endpoint, String clientId, String logName) {
        return new YatsLog(endpoint, clientId, logName);
    }

    private YatsLog(String endpoint, String clientId, String logName) {
        Objects.requireNonNull(endpoint);
        Objects.requireNonNull(clientId);
        Objects.requireNonNull(logName);

        this.endpoint = endpoint;
        this.logName = logName;
        this.clientId = clientId;
    }

    public CompletableFuture logMessage(Long timestamp, String message) {
        var json = String.format("""
                 {"mtime": %d, "name": "%s", "value": "%s"}
                """, timestamp, logName, message);

        System.out.printf("endpoint = %s, clientId = %s, logName = %s\n", endpoint, clientId,
logName);
        System.out.printf("json = [%s]", json);

        return CompletableFuture.supplyAsync(
                () -> post(clientId, endpoint + "/metric", json)
        );
    }

    public CompletableFuture logRequest(Long timestamp, HttpServletRequest request) {
        var payloadJson = Json.with("created", timestamp)
                .and("urlname", request.getRequestURL().toString())
                .and("referer", request.getHeader("referer"))
                .and("host", request.getHeader("host"))
                .and("userAgent", request.getHeader("user-agent"))
                .and("x-ssl-client-cn", request.getHeader("x-ssl-client-cn"))
                .and("x-forwarded-for", request.getHeader("x-forwarded-for"));

        var base64 = Base64.getEncoder()
                .encodeToString(payloadJson.toString().getBytes(StandardCharsets.UTF_8));

        var jsonMessage = String.format("""
                 {"mtime": %d, "name": "%s", "value": "%s"}
                """, timestamp, logName, base64);

        return CompletableFuture.supplyAsync(
                () -> post(clientId, endpoint + "/metric", jsonMessage)
        );
    }

    private static String post(String clientId, String url, String jsonString) {
        try {
            var request =
HttpRequest.newBuilder().POST(HttpRequest.BodyPublishers.ofString(jsonString))
                    .uri(URI.create(url))
                    .header("accept", "application/json")
                    .header("X-SSL-Client-CN", clientId)
                    .build();
            var response = httpClient.send(request, HttpResponse.BodyHandlers.ofInputStream());
            return new String((response.body()).readAllBytes());
        } catch (Exception e) {
            return null;
        }
    }
}
```

Logging Requests with a Filter

```
import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

public class YatsLogFilter implements Filter {
    private final YatsLog yatsLog = Main.yatsLog;

    public void doFilter(
            ServletRequest req,
            ServletResponse res,
            FilterChain chain
    ) {
        try {
            var resp = (HttpServletResponse) res;
            yatsLog.logRequest(System.currentTimeMillis() / 1000, (HttpServletRequest) req);
            chain.doFilter(req, resp);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

© 2024 - Kevwe Technology AB