ActivityWatch Documentation

Release 0.7.0

Erik Bjäreholt and contributors

Mar 27, 2018

User documentation

Note: ActivityWatch is currently under development and should not be considered stable software, yet.

CHAPTER 1

Introduction

ActivityWatch is a bundle of software that tracks your computer activity. You are, by default, the sole owner of your data.

It also offers an ecosystem of software to work around it, including ways to collect more data and do different kinds of analysis,

1.1 What ActivityWatch is

- A set of watchers that record relevant information about what you do and what happens on your computer (such as if you are AFK or not, or which window is currently active).
- A way of storing data collected by the watchers.
- A dataformat accomodating most logging needs due to its flexibility.
- An ecosystem of tools to help users extend the software to fit their needs.

1.2 Reason for existence

There are plenty of companies offering services which do collection of Quantified Self data with goals ranging from increasing personal producivity to understanding the people that managers manage (organizational productivity). However, all known services suffer from a significant disadvantage, the users data is in the hands of the service providers which leads to the problem of trust. Every customer of these companies have their data in hands they are forced to trust if they want to use their service.

This is a significant problem, but the true reason that we decided to do something about it was that existing solutions were inadequate. They focused on short-term insight, a goal worthy in itself, but we also want long-term understanding. We made it completely free and open source so anyone can use, improve and extend it.

1.3 Data philosophy

Data in it's raw form is always the most valuable.

Quantified self data doesn't take much space by todays standards, but for services such as RescueTime which have over than thousand of customers, every megabyte per user counts.

For the users however, every megabyte of data is worth it. It is therefore of importance that we collect and store data in the highest reasonable resolution such that we later don't have to "fill the gaps" in lower resolution data with lossy heuristics.

Many services doing collection and analysis of QS data today don't actually store the raw data but instead store only summaries (such as only storing how long you used an applicatin during a given hour, instead of storing the individual uses). This is a problem with existing services: they store summarized data instead of the raw data.

This is indicative of that they actually lack a long-term plan. They want to provide a certain type of analysis *today* but we expect to want to do some unknown analysis in the future, and for that we might need the raw data.

Simply put: It is of importance that we start collecting raw data now, because if we don't it will be forever lost.

CHAPTER 2

Getting started

Note: We're currently working on improving the installation experience by creating proper installers and packages, but for now we offer standalone archives containing everything you need.

2.1 Installation

Note: The prebuilt packages are known to sometimes have issues on Linux. If they don't work for you, please create an issue and consider *Installing from source*.

1. First, grab the latest release from GitHub for your operating system.
2. Unzip the archive into an appropriate directory and you're done!

2.2 Usage

The aw-qt application is the easiest way to use ActivityWatch. It creates a trayicon and automatically starts the server and the default watchers.

Simply run the `./aw-qt` binary in the installation directory (either from your terminal or on Windows by double-clicking). You now should see an icon appear in your system tray (unless you're running Gnome 3, where there is no system tray).

You should now also have the web interface running at localhost:5600 and will in a few minutes be able to view your data under the Activity section!

Note: If you want more advanced ways to run ActivityWatch (including running it without aw-qt), check out the "Running" section of *Installing from source*.

2.3 Autostart

You might want to make `aw-qt` start automatically on login. We hope to automate this for you in the future but for now you'll have to do it yourself. Searching the web for "autostart application <your operating system>" should get you some good results that don't take long.

2.4 Config

Configuration files for ActivityWatch can be found at the following default locations:

- Unix: `~/.config/activitywatch` or the path defined by the `$XDG_CONFIG_HOME` environment variable.
- Mac OS X: `~/Library/Preferences/activitywatch`
- Windows 7 & 10: `C:\Users\<username>\AppData\Local\activitywatch\activitywatch`
- Windows XP: `C:\Documents and Settings\<username>\Application Data\activitywatch\activitywatch`

Config options for the server, client, and default watchers are listed below:

- aw-server
- `host` Hostname to start the server on. Currently only `localhost` or `127.0.0.1` are supported.
- `port` Port number to start the server on.
- `storage` Type of storage for holding buckets and events. Supported types are `memory`, `mongodb`, or `peewee`.
- aw-client
- `hostname` Hostname of the server to connect to.
- `port` Port number of the server to connect to.
- aw-watcher-afk
- `timeout` Time in seconds with no activity required to become afk.
- `poll_time` Time in seconds between checks for activity.
- `update_time` Not yet implemented.
- aw-watcher-window:
- `poll_time` Time in seconds between window checks.
- `update_time` Not yet implemented.

CHAPTER 3

FAQ

Note: Some of these questions are technically not frequently asked.

3.1 How does ActivityWatch know when I am AFK?

On Windows and macOS, we use functionality offered by those platforms that gives us the time since last input.

On Linux, we monitor all mouse and keyboard activity so that we can calculate the time since last input. We do not store what that activity was, just that it happened.

With this data (seconds since last input) we then check if there is less than 3 minutes between input activity. If there is, we consider you not-AFK. If more than 3 minutes passes without any input, we consider that as if you were AFK from the last input until the next input occurs.

3.2 Why is the active window logged as "unknown" when using Wayland?

The Wayland protocol does not have a notion of an active window, and it is unlikely to ever have. Wayland is also developed in security in mind, so access should be handed out on an app-by-app basis. This is a good idea, any application shouldn't just give that privacy-sensitive information away freely.

Unfortunately, in Wayland compositors like Gnome's Mutter there is no way at all to get the current window, this leaves the window watcher completely disabled in Wayland.

Solution: Switch to using X11 (the best option), and if you can't: bother the developer of your Wayland compositor.

You can see the general status of the ability of getting the active window in Wayland on StackOverflow or follow the issue for ActivityWatch tracking the problem.

3.3 How do I programmatically use ActivityWatch?

See the documentation for *Extending ActivityWatch* or checkout the aw-client repository.

3.4 How do I understand the data that is stored?

All ActivityWatch data is represented using *Buckets and Events*.

All events from have the fields `timestamp` (ISO 8601 formatted), `duration` (in seconds), and `data` (a JSON object).

You can programmatically get some events yourself to inspect with the following code:

```
ac = aw_client.ActivityWatchClient("")

# Returns a dict with information about every bucket
buckets = ac.get_buckets()

# Get the first bucket
bucket_id = next(buckets.keys())
events = ac.get_events(bucket_id)
```

As an example for AFK events: The data object contains has one attribute `status` which can be `afk` or `not-afk`.

No two events in a bucket should cover the same moment, if that happens there is an issue with the watcher that should be resolved.

3.5 What happens if it is down or crashes?

Since ActivityWatch consists of several modules running independently, one thing crashing will have limited impact on the rest of the system.

If the server crashes, all watchers which use the heartbeat queue should simply queue heartbeats until the server becomes available again. Since heartbeats are currently sent immediately to the server for storage, all data before the crash should be untouched.

If a watcher crashes, its bucket will simply remain untouched until it is restarted.

3.6 What happens when my computer is off or asleep?

If your computer is off or asleep, watchers will usually record nothing. i.e. one events ending (`timestamp + duration`) will not match up with the following event's beginning (`timestamp`).

3.7 Some events have 0 duration. What does this mean?

Watchers most commonly use a polling method called heartbeats in order to store information on the server. Heartbeats are received regularly with some data, and when two consecutive heartbeats have identical data they get merged and the duration of the new one becomes the time difference between the previous two. Sometimes, a single heartbeat doesn't get a following event with identical data. It is then impossible to know the duration of that event.

The assumption could be made to consider all zero-duration events actually have a duration equal to the time of the next event, but all such assumptions are left to the analysis stage.

CHAPTER 4

Features

Here we will document a few features.

4.1 User Interface

4.1.1 Web Interface

ActivityWatch comes with a web interface which currently has the following features:

- **Activity overview**
  - Most used applications by day
  - Timeline
  - Most time spent on a website *(requires the ActivityWatch browser extension)*
- **Bucket overview**
  - When a bucket was last updated
  - Listing of the latest events

More advanced and configurable visualization (such as the ones found in Zenobase and RescueTime) is not a priority and is unlikely to get implemented as a part of the core ActivityWatch project anytime soon.

4.1.2 Tray icon

The tray icon (aw-qt) manages the core ActivityWatch services (server + watchers) and offers:

- Manage which ActivityWatch services to run
- Popup when a service crashes

4.2 Exporting data

If you go to the "Raw Data" page in the ActivityWatch webui you can download any of the buckets which contain every collected datapoint in ActivityWatch as a single file.

You can also export data programatically using the REST API, but we do not have a guide for that yet **todo: explain how**

4.3 Pausing logging

The possibility to pause logging is a low-tech solution to filter sensitive data. You can do it easily by simply unchecking the watcher module you want to pause in the aw-qt trayicon menu, this will stop the watcher until you check it again.

So, if you for example want to pause the logging of window titles:

- Click the ActivityWatch trayicon
- Uncheck 'Modules -> aw-watcher-window'

4.4 Filtering data

Note: This is a planned feature.

ActivityWatch was born out of a frustration with the privacy issues of existing life logging solutions. We feel that it's important that some things that are exceptionally sensitive shouldn't be logged at all. This way the cost of data breach is bounded, and the barrier to sharing your own data will hopefully become smaller.

This is expected to be almost impossible to perfect since what someone considers exceptionally sensitive might not be for someone else (due to e.g. culture and law). But the basics are easy to get right (such as not logging private browser tabs).

For the ones who believe they can adequately protect their data, they should be offered the option to disable the filter.

Currently, the only way to do this is by manually pausing logging.

CHAPTER 5

Watchers

Watchers are the parts of ActivityWatch that do all the data collecting.

For help on how to write your own watcher, see *Writing your first watcher*.

5.1 Default

ActivityWatch comes bundled with two watchers by default:

- aw-watcher-afk - Watches for mouse & keyboard activity to detect if the user is active.
- aw-watcher-window - Watches the active window and its title.

5.2 Others

The default watchers are collecting some of the most important data. But there is more to collect, so here are some other watchers that let you do so.

- aw-watcher-web - (Beta) A browser extension that watches the active tab and its title along with its URL.
- aw-watcher-spotify - (Beta) Uses the Spotify Web API to get the active track.
- aw-watcher-chromecast - (WIP) Watches whatever your Chromecast is up to.

Have you written one yourself? Send us an email or PR to have it included!

CHAPTER 6

Extending ActivityWatch

So, you want to do something more with ActivityWatch? Great!

We've tried to make things easy for you (and ourselves) so here's some advice on how to get started.

6.1 Collecting more data

ActivityWatch is written to be flexible to be able to gather most types of data. Except for the included aw-watcher-window and aw-watcher-afk which tracks your application usage, there are additional so-called *Watchers* for activitywatch. Watchers are small programs that collect data and send it off to the server. The only requirement for what kind of data is sent to aw-server as an event is that it has to contain a starttime (and preferably a duration aswell) so it can fit on a timeline.

If you want to write a watcher of your own, see *Writing your first watcher*.

6.2 Fetching Data

If you want to fetch data from aw-server for visualization, exporting, backup or something we have not yet thought of, there are a few ways you can do this:

- `Exporting a Bucket` If you want a complete dump of all events of bucket
- Bucket REST API If you want to export raw events in a specific time interval from a bucket
- Writing a Query If you want to summarize/aggregate one or more buckets into more easily readable data

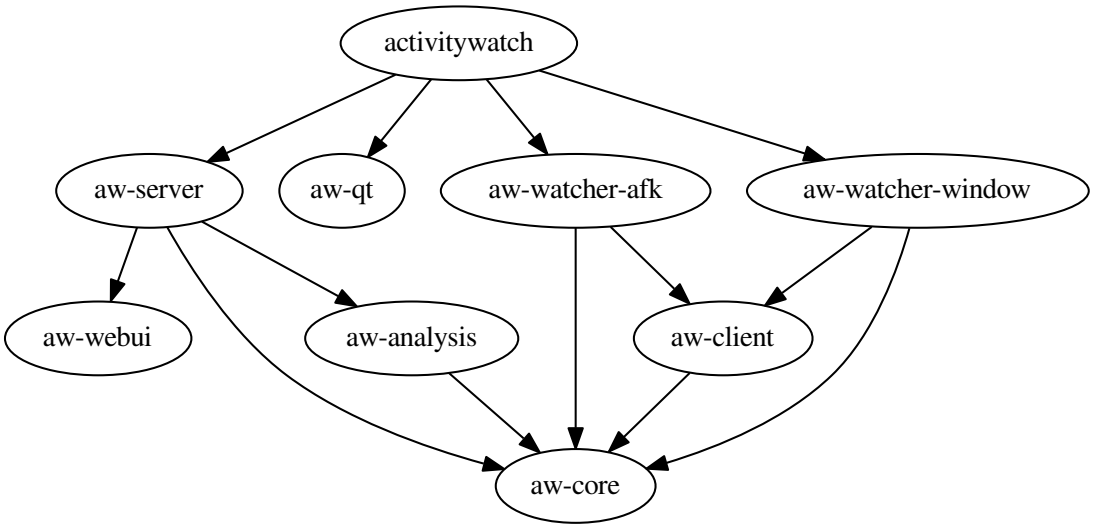
CHAPTER 7

Development

Note: This part is a work in progress, reach out to the maintainers if you have any questions!

We recommend you follow Kenneth Reitz folder structure guide when writing Python programs which will be under the control of the ActivityWatch organisation: http://docs.python-guide.org/en/latest/writing/structure/

7.1 Dependency graph



7.2 Working with submodules

Working with submodules comes with some complexity, here are a few neat tricks to make things easier:

- We recommend configuring git to include submodule changes in `git status`, you can do so with the following: `git config --global status.submoduleSummary true`
- If you want the latest committed version of all submodules, use: `git submodule update --recursive`
- If you want the latest master branch on all submodules, use: `git submodule update --recursive --remote`
- If you want to ensured you've pushed all commits in the submodules, use: `git submodule foreach 'git push'`

A longer guide to git submodules can be found here.

7.3 Making a release

1. Close milestone on GitHub if one exists.
2. Ensure that all the tests pass: `make test && make test-integration`
3. Test the latest build and check that it works correctly
4. Write a changelog entry in `docs/changelog.rst`
5. Sign the commit: `git commit -a -S -m "bumped version"`
6. Create a signed tag: `git tag -s v0.7.1`
7. Push the commit and tag: `git push origin refs/tags/v0.7.1`
8. Create a release on GitHub
9. Wait for the builds to finish
10. Post about it online: Twitter, the forum, mailinglist (if major)

CHAPTER 8

Architecture

Here we hope to clarify the architecture of ActivityWatch for you. Please file an issue or pull request if you think something is missing.

8.1 Server

Known as aw-server, it handles storage and retrieval of all activities/entries in buckets. Usually there exists one bucket per watcher.

The server also hosts the Web UI (aw-webui) which does all communication with the server using the REST API.

8.2 Watchers

Since aw-server doesn't do any data collection on it's own, we need watchers that observe the world and sent the data off to aw-server for storage.

These utilize the aw-client library for making requests to the aw-server.

For a list of watchers, see *Watchers*.

8.3 User interfaces

ActivityWatch currently has two user interfaces, aw-qt and aw-webui.

- aw-qt - Manages the server and watchers to make ActivityWatch easy to use for end-users.
- aw-webui - Offers visualization and an overview of the database. Hosted by aw-server in the bundle.

8.4 Libraries

Some of the logic of ActivityWatch is shared across the server and clients, for these cases we moved some logic into seperate libraries.

8.4.1 aw-core

The aw-core library contains many of the essential parts of ActivityWatch, notably:

- The *Buckets and Events*
- The datastore layer
- Utilities (configuration, logging, decorators)

8.4.2 aw-client

Writing these clients is something we've tried to make as easy as possible by creating client libraries with a clear API. A client could both be a watcher which sends data as well as a visualizer which fetches and presents data from the aw-server.

Currently the primary client library is written in Python (known simply as aw-client) but a client library written in JavaScript is on the way and is expected to have the same level of support in the future.

- aw-client (Python)
- aw-client-js (JavaScript, work in progress)

8.4.3 aw-analysis

There are also plans to create a library called aw-analysis to aid in different types of analysis and transformation one might want to make using ActivityWatch data.

CHAPTER 9

Buckets and Events

9.1 Buckets

```
bucket = {
  "id": "aw-watcher-mywatcher_myhostname",
  "created": "2017-05-16T13:37:00.000000",
  "name": "A short but descriptive human readable bucketname",
  "type": "mybuckettype",
  "client": "aw-watcher-mywatcher",
  "hostname": "myhostname",
}
```

In ActivityWatch we try and separate each kind of datapoint. Normally what is most convenient is to have one bucket per client and host. For example if we have a watcher which tracks info about the focused window on your computer we would upon startup create a bucket named 'aw-watcher-window_my-host-name' to fill all of our events with datapoints to.

Each bucket also has a buckettype. The buckettype specifies the format of the data in the event, so for example for the bucket of our focused window watcher we could have the buckettype 'currentwindow'. The buckettype is just a ordinary string, but it is good for clients who want to visualize the data in ActivityWatch to know what the buckets contain.

9.2 Events

The ActivityWatch event model is pretty simple, here's its representation in JSON:

```
event = {
  "timestamp": "2016-04-27T15:23:55Z",  // ISO8601 formatted timestamp
  "duration": 3.14,                     // Duration in seconds
  "data": {"key": "value"},  // A JSON object, the schema of this depends on the␣
↪bucket type
}
```

It should be noted that all timestamps are stored as UTC. Timezone information (UTC offset) is currently discarded.

The content in the "data" field could be any JSON object, but it is recommended that every event in a bucket should follow some format depending on the buckettype so the data is easy to analyze.

CHAPTER 10

Writing your first watcher

Writing watchers for ActivityWatch is pretty easy, all you need is the `aw-client` library.

Note: These examples runs the client in *testing* mode, which means that it will try to connect to a aw-server in testing mode on the port 5666 instead of the normal 5600.

10.1 Minimal client

Below is a minimal template client to quickly get started. This example will:

- create a bucket
- insert an event
- fetch an event from an aw-server bucket
- delete the bucket again

```
#!/usr/bin/env python3

from datetime import datetime, timezone

from aw_core.models import Event
from aw_client import ActivityWatchClient

# We'll run with testing=True so we don't mess up any production instance.
# Make sure you've started aw-server with the `--testing` flag as well.
client = ActivityWatchClient("test-client", testing=True)

bucket_id = "{}_{}".format("test-client-bucket", client.hostname)
client.create_bucket(bucket_id, event_type="dummydata")

shutdown_data = {"label": "some interesting data"}
```

(continues on next page)

(continued from previous page)

```
now = datetime.now(timezone.utc)
shutdown_event = Event(timestamp=now, data=shutdown_data)
inserted_event = client.insert_event(bucket_id, shutdown_event)

events = client.get_events(bucket_id=bucket_id, limit=1)
print(events) # Should print a single event in a list

client.delete_bucket(bucket_id)
```

10.2 Reference client

Below is a example of a watcher with more in-depth comments. This example will describe how to:

- how to create buckets
- how to send events by heartbeats
- how to insertion events without heartbeats
- how to do syncronous as well as asyncronous requests
- fetch events from a aw-server bucket
- delete buckets

```
#!/usr/bin/env python3

from time import sleep
from datetime import datetime, timedelta, timezone

from aw_core.models import Event
from aw_client import ActivityWatchClient

# We'll run with testing=True so we don't mess up any production instance.
# Make sure you've started aw-server with the `--testing` flag as well.
client = ActivityWatchClient("test-client", testing=True)

# Make the bucket_id unique for both the client and host
# The convention is to use client-name_hostname as bucket name,
# but if you have multiple buckets in one client you can add a
# suffix such as client-name-event-type or similar
bucket_id = "{}_{}".format("test-client-bucket", client.hostname)
# A short and descriptive event type name
# Will be used by visualizers (such as aw-webui) to detect what type and format the␣
↪events are in
# Can for example be "currentwindow", "afkstatus", "ping" or "currentsong"
event_type = "dummydata"

# First we need a bucket to send events/heartbeats to.
# If the bucket already exists aw-server will simply return 304 NOT MODIFIED,
# so run this every time the clients starts up to verify that the bucket exists.
# If the client was unable to connect to aw-server or something failed
# during the creation of the bucket, an exception will be raised.
client.create_bucket(bucket_id, event_type="test")

# Asynchronous loop example
```

(continues on next page)

(continued from previous page)

```
with client:
    # This context manager starts the queue dispatcher thread and stops it when done,␣
↪always use it when setting queued=True.
    # Alternatively you can use client.connect() and client.disconnect() instead if␣
↪you prefer that

    # Create a sample event to send as heartbeat
    heartbeat_data = {"label": "heartbeat"}
    now = datetime.now(timezone.utc)
    heartbeat_event = Event(timestamp=now, data=heartbeat_data)

    # Now we can send some events via heartbeats
    # This will send one heartbeat every second 5 times
    sleeptime = 1
    for i in range(5):
        # The duration between the heartbeats will be less than pulsetime, so they␣
↪will get merged.
        # TODO: Make a section with an illustration on how heartbeats work and insert␣
↪a link here
        print("Sending heartbeat {}".format(i))
        client.heartbeat(bucket_id, heartbeat_event, pulsetime=sleeptime+1,␣
↪queued=True)

        # Sleep a second until next heartbeat
        sleep(sleeptime)

        # Update timestamp for next heartbeat
        heartbeat_event.timestamp = datetime.now(timezone.utc)

    # Give the dispatcher thread some time to complete sending the last events.
    # If we don't do this the events might possibly queue up and be sent the
    # next time the client starts instead.
    sleep(1)

# Synchronous example, insert an event
event_data = {"label": "non-heartbeat event"}
now = datetime.now(timezone.utc)
event = Event(timestamp=now, data=event_data)
inserted_event = client.insert_event(bucket_id, event)

# The event returned from insert_event has been assigned an id by aw-server
assert inserted_event.id is not None

# Fetch last 10 events from bucket
# Should be two events in order of newest to oldest
# - "shutdown" event with a duration of 0
# - "heartbeat" event with a duration of 5*sleeptime
events = client.get_events(bucket_id=bucket_id, limit=10)
print(events)

# Now lets clean up after us.
# You probably don't want this in your watchers though!
client.delete_bucket(bucket_id)

# If something doesn't work, run aw-server with --verbose to see why some request␣
↪doesn't go through
# Good luck with writing your own watchers :-)
```

CHAPTER 11

Querying Data

There are a couple of ways to query data in activitywatch.

aw-server supplies an "/query" endpoint (also accesible via aw-client's query method) which supplies a basic scripting language which you can utilize to do transformations on the server-side. This option is good for basic analysis and for lightweight clients (such as aw-webui).

Another option is to fetch events from the "/buckets/bucketname/events" endpoint (also accesible via aw-client's get_events method) and either program your own transformations or use transformation methods available in the aw-analysis python library (which includes all transformations available in the query endpoint). This require a lot of more work since you will likely have to reprogram transformations already available in the query API, but on the other hand it is much more flexible.

11.1 Writing a Query

Note: This section is still WIP. There is still no documentation of all the transform functions, but for most simple queries these examples should be enough.

Queries are the easiest yet advanced way to get events from aw-server buckets in a format which fits most needs. Queries can be done by doing a POST request to aw-server either manually or with the aw-client library.

In a query you start by getting events from a bucket and assign that collection of events to a variable, then there are multiple transform functions which you can use to for example filter, limit, sort, and merge events from a bucket. After that you assign what you want to receive from the request to the RETURN variable.

Minimal query which only gets events from a bucket and returns it

```
events = query_bucket("my_bucket");
RETURN = events;
```

A query which merges events from a bucket in a key1->key2 hierarchy

```
events = query_bucket("my_bucket");
events = merge_events_by_keys(events, "merged_key1", "merged_key2");
RETURN = events;
```

A simplified query example of how to summarize what programs used while not afk. The query intersects the not-afk events from the afk bucket with the events from the window bucket, merges keys from the result and sorts by duration.

```
window_events = query_bucket("window_bucket");
not_afk_events = query_bucket("afk_bucket");
not_afk_events = filter_keyvals(not_afk_events, "status", "not-afk");
window_events = filter_period_intersect(window_events, afk_events);
events = merge_events_by_keys(window_events, "appname");
events = sort_by_duration(events);
RETURN = events;
```

This is an example of how you can do analysis and aggregation with the query method in python with aw-client

```
#!/usr/bin/env python3

from time import sleep
from datetime import datetime, timedelta, timezone

from aw_core.models import Event
from aw_client import ActivityWatchClient

client = ActivityWatchClient("test-client", testing=True)

now = datetime.now(timezone.utc)
start = now

query = "RETURN=0;"
res = client.query(query, "1970-01-01", "2100-01-01")
print(res) # Should print 0

bucket_id = "{}_{}".format("test-client-bucket", client.hostname)
event_type = "dummydata"
client.create_bucket(bucket_id, event_type="test")

def insert_events(label: str, count: int):
    global now
    events = []
    for i in range(count):
        e = Event(timestamp=now,
                  duration=timedelta(seconds=1),
                  data={"label": label})
        events.append(e)
        now = now + timedelta(seconds=1)
    client.insert_events(bucket_id, events)

insert_events("a", 5)

query = "RETURN = query_bucket('{}');".format(bucket_id)

res = client.query(query, "1970", "2100")
print(res) # Should print the last 5 events

res = client.query(query, start + timedelta(seconds=1), now - timedelta(seconds=2))
```

(continues on next page)

(continued from previous page)

```
print(res) # Should print three events

insert_events("b", 10)

query = """
events = query_bucket('{}');
merged_events = merge_events_by_keys(events, 'label');
RETURN=merged_events;
""".format(bucket_id)
res = client.query(query, "1970", "2100")
# Should print two merged events
# Event "a" with a duration of 5s and event "b" with a duration of 10s
print(res)

client.delete_bucket(bucket_id)
```

11.2 Fetching Raw Events

TODO: Write this section

Bucket REST API

CHAPTER 12

Installing from source

Here's the guide to installing ActivityWatch from source. If you are just looking to try it out, see the getting started guide instead.

Note: This is written for Linux and macOS. For Windows the build process is more complicated and we therefore suggest using the pre-built packages instead on that operating system.

12.1 Cloning the submodules

Since the ActivityWatch bundlerepo uses submodules, you first need to clone the submodules.

This can either be done at the cloning stage with:

```
git clone --recursive https://github.com/ActivityWatch/activitywatch.git
```

Or afterwards (if you've already cloned normally) using:

```
git submodule update --init --recursive
```

12.2 Checking dependencies

You need to ensure you have:

- Python 3.5 or later, check with `python3 -V` (required to build the core components)
- Node 5 or higher, check with `node -v` and `npm -v` (required to build the web UI)

12.3 Using a virtualenv

It is recommended to use a virtualenv in order to avoid polluting your system with ActivityWatch-specific packages Python packages. It also makes it easier to uninstall since all you have to do is remove the virtualenv folder.

Note: If you don't want to use a virtualenv you could instead set the environment variable `PIP_USER=true` when building. This will install ActivityWatch for your user only instead of trying to install it system-wide (which would fail since it requires root). If you do this, make sure that the folder `~/.local/bin` (on Linux) or `~/Library/Python/<version>/bin` (on macOS) is in your PATH so that you can run the programs once installed.

```
python3 -m venv venv
```

Now activate the virtualenv in your current shell session:

```
# For bash/zsh users:
source ./venv/bin/activate
# For fish users:
source ./venv/bin/activate.fish
```

12.4 Building and installing

Build and install everything into the virtualenv:

```
make build
```

Note: If you are going to develop we suggest building/installing using `make build DEV=true` which installs all Python packages with pip's handy `--editable` flag. By doing this you wont have to reinstall everything whenever you want to try out a code change.

12.5 Running

Now you should be able to start ActivityWatch **from the terminal where you've activated the virtualenv**. Or, if you were using the `PIP_USER` trick, from any terminal with a correctly configured PATH. You have two options:

1. Use the trayicon manager (Recommended for normal use)
    - Run from your terminal with: `aw-qt`
2. Start each module separately (Recommended for developing)
    - Run from your terminal with: `aw-server`, `aw-watcher-afk`, and `aw-watcher-window`

Both methods take the `--testing` flag as a command line parameter to run in testing mode. This runs the server on a different port (5666) and uses a separate database file to avoid mixing your important data with your testing data.

Now everything should be running! Check out the web UI at http://localhost:5600/

If anything doesn't work, let us know!

Note: On Linux, if you want to run from source using a `.desktop` file launcher, see this issue.

12.6 Updating from source

First pull the latest version of the repo with `git pull` then get the updated submodules with `git submodule update --init --recursive`. All that's needed then is a `make build`.

If it doesn't work, you can first try to run `make uninstall` and then do a fresh `make build`. If that fails as well, remove the virtualenv and start over.

Please report all issues you might have so we can make things easier for future users.

CHAPTER 13

API Reference

Here's an API reference for some of the most central components in `aw_core`, `aw_client` and `aw_server`. These are the most important packages in ActivityWatch. A lot of it currently lacks proper docstrings, but it's a start.

Contents

13.1 aw_core

13.1.1 aw_core.models

class aw_core.models.**Event**(*id: Union[int, str, NoneType] = None, timestamp: Union[datetime.datetime, str] = None, duration: Union[datetime.timedelta, int, float] = 0, data: Dict[str, Any] = {}*) → None

Used to represents an event.

data

duration

id

timestamp

to_json_dict() → dict
Useful when sending data over the wire. Any mongodb interop should not use do this as it accepts datetimes.

to_json_str() → str

13.1.2 aw_core.log

aw_core.log.**get_latest_log_file**(*name*, *testing=False*) → Union[str, NoneType]
Returns the filename of the last logfile with name. Useful when you want to read the logfile of another ActivityWatch service.

aw_core.log.**get_log_file_path**() → Union[str, NoneType]
DEPRECATED: Use get_latest_log_file instead.

aw_core.log.**setup_logging**(*name: str*, *testing=False*, *verbose=False*, *log_stderr=True*, *log_file=False*, *log_file_json=False*)

13.1.3 aw_core.dirs

aw_core.dirs.**ensure_path_exists**(*path: str*) → None

aw_core.dirs.**get_config_dir**(*module_name: Union[str, NoneType]*) → str

aw_core.dirs.**get_data_dir**(*module_name: Union[str, NoneType]*) → str

aw_core.dirs.**get_log_dir**(*module_name: Union[str, NoneType]*) → str

13.2 aw_client

The aw_client package contains a programmer-friendly wrapper around the servers REST API.

class aw_client.**ActivityWatchClient**(*client_name: str = 'unknown'*, *testing=False*) → None
A handy wrapper around the aw-server REST API. The recommended way of interacting with the server.

Can be used with a with-statement as an alternative to manually calling connect and disconnect in a try-finally clause.

Example

```
from aw_client import ActivityWatchClient

# We'll run with testing=True so we don't mess up any production instance.
# Make sure you've started aw-server with the `--testing` flag as well.
client = ActivityWatchClient("test-client", testing=True)

# Make the bucket_id unique for both the client and host
# The convention is to use client-name_hostname as bucket name,
# but if you have multiple buckets in one client you can add a
# suffix such as client-name-event-type or similar
bucket_id = "{}_{}".format("test-client-bucket", client.hostname)
```

(continues on next page)

(continued from previous page)

```
# A short and descriptive event type name
# Will be used by visualizers (such as aw-webui) to detect what type and format␣
↪the events are in
# Can for example be "currentwindow", "afkstatus", "ping" or "currentsong"
event_type = "dummydata"

# First we need a bucket to send events/heartbeats to.
# If the bucket already exists aw-server will simply return 304 NOT MODIFIED,
# so run this every time the clients starts up to verify that the bucket exists.
# If the client was unable to connect to aw-server or something failed
# during the creation of the bucket, an exception will be raised.
client.create_bucket(bucket_id, event_type="test")

# Asynchronous loop example
with client:
    # This context manager starts the queue dispatcher thread and stops it when␣
↪done, always use it when setting queued=True.
    # Alternatively you can use client.connect() and client.disconnect() instead␣
↪if you prefer that

    # Create a sample event to send as heartbeat
    heartbeat_data = {"label": "heartbeat"}
    now = datetime.now(timezone.utc)
    heartbeat_event = Event(timestamp=now, data=heartbeat_data)

    # Now we can send some events via heartbeats
    # This will send one heartbeat every second 5 times
    sleeptime = 1
    for i in range(5):
        # The duration between the heartbeats will be less than pulsetime, so␣
↪they will get merged.
        # TODO: Make a section with an illustration on how heartbeats work and␣
↪insert a link here
        print("Sending heartbeat {}".format(i))
        client.heartbeat(bucket_id, heartbeat_event, pulsetime=sleeptime+1,␣
↪queued=True)

        # Sleep a second until next heartbeat
        sleep(sleeptime)

        # Update timestamp for next heartbeat
        heartbeat_event.timestamp = datetime.now(timezone.utc)

    # Give the dispatcher thread some time to complete sending the last events.
    # If we don't do this the events might possibly queue up and be sent the
    # next time the client starts instead.
    sleep(1)

# Synchronous example, insert an event
event_data = {"label": "non-heartbeat event"}
now = datetime.now(timezone.utc)
event = Event(timestamp=now, data=event_data)
inserted_event = client.insert_event(bucket_id, event)

# The event returned from insert_event has been assigned an id by aw-server
assert inserted_event.id is not None
```

(continues on next page)

(continued from previous page)

```
# Fetch last 10 events from bucket
# Should be two events in order of newest to oldest
# - "shutdown" event with a duration of 0
# - "heartbeat" event with a duration of 5*sleeptime
events = client.get_events(bucket_id=bucket_id, limit=10)
print(events)

# Now lets clean up after us.
# You probably don't want this in your watchers though!
client.delete_bucket(bucket_id)

# If something doesn't work, run aw-server with --verbose to see why some request␣
↪doesn't go through
# Good luck with writing your own watchers :-)
```

connect()

create_bucket(*bucket_id: str*, *event_type: str*, *queued=False*)

delete_bucket(*bucket_id: str*)

disconnect()

get_buckets()

get_eventcount(*bucket_id: str*, *limit: int = 100*, *start: datetime.datetime = None*, *end: datetime.datetime = None*) → int

get_events(*bucket_id: str*, *limit: int = 100*, *start: datetime.datetime = None*, *end: datetime.datetime = None*) → List[aw_core.models.Event]

get_info()
Returns a dict currently containing the keys 'hostname' and 'testing'.

heartbeat(*bucket_id: str*, *event: aw_core.models.Event*, *pulsetime: float*, *queued: bool = False*) → Union[aw_core.models.Event, NoneType]
This endpoint can use the failed requests retry queue. This makes the request itself non-blocking and therefore the function will in that case always returns None.

insert_event(*bucket_id: str*, *event: aw_core.models.Event*) → aw_core.models.Event

insert_events(*bucket_id: str, events: List[aw_core.models.Event]*) → None

query(*query: str*, *start: datetime.datetime*, *end: datetime.datetime*, *name=''*, *cache: bool = False*) → Union[int, dict]

send_event(*bucket_id: str*, *event: aw_core.models.Event*)

send_events(*bucket_id: str, events: List[aw_core.models.Event]*)

setup_bucket(*bucket_id: str*, *event_type: str*)

13.3 aw_server

13.3.1 aw_server.api

The `ServerAPI` class contains the basic API methods, these methods are primarily called from RPC layers such as the one found in `aw_server.rest`.

class aw_server.api.**ServerAPI**(*db*, *testing*) → None

create_bucket(*bucket_id: str*, *event_type: str*, *client: str*, *hostname: str*) → bool
Create bucket. Returns True if successful, otherwise false if a bucket with the given ID already existed.

create_events(*bucket_id: str, events: List[aw_core.models.Event]*) → Union[aw_core.models.Event, NoneType]
Create events for a bucket. Can handle both single events and multiple ones.

Returns the inserted event when a single event was inserted, otherwise None.

delete_bucket(*bucket_id: str*) → None
Delete a bucket

delete_event(*bucket_id: str*, *event_id*) → bool
Delete a single event from a bucket

export_all() → Dict[str, dict]
Exports all buckets and their events to a format consistent across versions

export_bucket(*bucket_id: str*) → Dict[str, Any]
Export a bucket to a dataformat consistent across versions, including all events in it.

get_bucket_metadata(*bucket_id: str*) → Dict[str, Any]
Get metadata about bucket.

get_buckets() → Dict[str, Dict]
Get dict {bucket_name: Bucket} of all buckets

get_eventcount(*bucket_id: str*, *start: datetime.datetime = None*, *end: datetime.datetime = None*) → int
Get eventcount from a bucket

get_events(*bucket_id: str*, *limit: int = -1*, *start: datetime.datetime = None*, *end: datetime.datetime = None*) → List[aw_core.models.Event]
Get events from a bucket

get_info() → Dict[str, Dict]
Get server info

get_log()
Get the server log in json format

heartbeat(*bucket_id: str*, *heartbeat: aw_core.models.Event*, *pulsetime: float*) → aw_core.models.Event
Heartbeats are useful when implementing watchers that simply keep track of a state, how long it's in that state and when it changes. A single heartbeat always has a duration of zero.

If the heartbeat was identical to the last (apart from timestamp), then the last event has its duration updated. If the heartbeat differed, then a new event is created.

Such as:

- Active application and window title - Example: aw-watcher-window
- Currently open document/browser tab/playing song - Example: wakatime - Example: aw-watcher-web - Example: aw-watcher-spotify
- Is the user active/inactive? Send an event on some interval indicating if the user is active or not. - Example: aw-watcher-afk

Inspired by: https://wakatime.com/developers#heartbeats

query2(*name*, *query*, *timeperiods*, *cache*)

CHAPTER 14

REST API

ActivityWatch uses a REST API for all communication between aw-server and clients. Most applications should never use HTTP directly but should instead use the client libraries available. If no such library yet exists for a given language, this document is meant to provide enough specification to create one.

Warning: The API is currently under development, and is subject to change. It will be documented in better detail when first version has been frozen.

Note: Part of the documentation might be outdated, you can get up-to-date API documentation in the API browser available from the web UI of your aw-server instance.

14.1 REST Security

Note: Our current security consists only of not allowing non-localhost connections, this is likely to be the case for quite a while.

Clients might in the future be able to have read-only or append-only access to buckets, providing additional security and preventing compromised clients from being able to cause a severe security breach. All clients will probably also encrypt data in transit.

14.2 REST Reference

Note: This reference is highly incomplete. For an interactive view of the API, try out the API playground running on your local server at: http://localhost:5600/api/

14.2.1 Buckets API

The most common API used by ActivityWatch clients is the API providing read and append access buckets. Buckets are data containers used to group data together which shares some metadata (such as client type, hostname or location).

Get Bucket Metadata

Will return 404 if bucket does not exist

```
GET /api/0/buckets/<bucket_id>
```

List

```
GET /api/0/buckets/
```

Create

Will return 304 if bucket already exists

```
POST /api/0/buckets/<bucket_id>
```

14.2.2 Events API

The most common API used by ActivityWatch clients is the API providing read and append *Events* to buckets. Buckets are data containers used to group data together which shares some metadata (such as client type, hostname or location).

Get events

```
GET /api/0/buckets/<bucket_id>/events
```

Create event

```
POST /api/0/buckets/<bucket_id>/events
```

14.2.3 Heartbeat API

The heartbeat API is one of the most useful endpoints for writing watchers.

```
POST /api/0/buckets/<bucket_id>/heartbeat
```

14.2.4 Query API

TODO: Add link to writing queries once that page is done

CHAPTER 15

Changelog

Warning: We haven't gotten to the point where we keep a flawless changelog yet. Please refer to the git history for more detail and certainty.

15.1 Released

15.1.1 v0.7.1

- Actually fixed the timezone issue in the web UI (issue #117).
- All issues assigned to the v0.7 milestone can be found on GitHub.

15.1.2 v0.7.0b4

- The ActivityWatch WebExtension is now supported from this version forward, see the announcement on the forum.
- (Not really, see v0.7.0b5) Fixed pesky timezone issue in web UI (issue #117).
- Fixed bug on macOS where keyboard activity would not be used to detect AFK state.
- Fixed packaging bugs (macOS, PyInstaller).
- The web extension now has a better look and notifies if connection to server failed.

15.1.3 v0.7.0b3

- Even more improvements to the web UI.
- Major improvements to the documentation, notably instructions on how to install from builds and sources.

15.1.4 v0.7.0b2

- Improvements to the web UI: a new visualization method (the "today" view) and information for users about the state of the project on the first page.

15.1.5 v0.7.0b1

There have been several major changes since v0.6. Much of it wont end up here but hopefully the major things will.

Note: If you are upgrading from a previous version, you might want to stop all loggers for the duration of your UTC offset to prevent issues which we've had difficulty debugging (or you can just start right away and expect your first hours to end up a bit weird).

- Now works on Windows.
- Working standalone packages. (edit: not reliable on all systems, but a lot easier to get running in many cases)
- All timestamps are now in UTC.
- Updated outdated parts of the documentation.
- Makefiles are now used throughout the projects to manage building, testing, and CI.
- A lot of bug fixes (and hopefully not too many new bugs).
- Vastly improved code quality.

15.1.6 v0.6.0 and older

We haven't been keeping track of changes very well for older versions. Please refer to the git history.

15.2 Upcoming

Warning: **Unreleased**: These are planned changelogs and will therefore change when plans change.

15.2.1 v0.7.2 (planned)

- New query2 API for querying and transforming data
- Web UI now has a view for the most-visited domains
- Web UI now has a button for removing buckets

CHAPTER 16

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index