Taxi Documentation

Release 6.2.0

Sylvain Fankhauser

May 15, 2023

Contents

CHAPTER 1

What is Taxi ?

Taxi is a timesheeting tool that focuses on simplicity to help you write your timesheets without wasting time. All you'll do is edit a text file and write down what you've worked on and how long, like so:

```
23/01/2014

pingpong 09:00-10:00 Play ping-pong
infra         -11:00 Repair coffee machine
```

You can then get a summary of your timesheet:

```
Staging changes :

# Thursday 23 january #
pingpong (123/456)            1.00  Play ping-pong
infra (123/42)                1.00  Repair coffee machine
                              2.00

Total                         2.00

Use `taxi ci` to commit staging changes to the server
```

Through the use of backends, Taxi allows you to push your timesheets to different systems.

CHAPTER 2

Getting started

Refer to the "Installation" section in the docs.

CHAPTER 3

Supported backends

- zebra : Liip's zebra backend
- tempo : Atlassian JIRA's Tempo Timesheets backend
- tipee : Gammadia's tipee backend
- bexio : Bexio Timesheets backend
- multi : a special backend to push entries over multiple other backends
- clockify : backend for the free timesheeting tool clockify.me

CHAPTER 4

Contrib packages

These resources, not part of Taxi core, provide an enhanced experience for certain use cases.

- Cabdriver (generate taxi entries based on Google Calendar, Slack, etc)
- Syntax highlighting for VSCode
- Vim plugin (features syntax highlighting, auto completion)

CHAPTER 5

Documentation index

5.1 User guide

5.1.1 Installation

To install Taxi, follow the steps below specific to your system.

OS X, Windows, generic Linux

Make sure you have Python at least 3.7 installed (by running `python3 --version`), then use `python3 -m pip` to install taxi in your user directory (you should **not** use sudo or run this command as root):

```
$ python3 -m pip install --user taxi
```

You'll probably want to install a backend too, that will allow you to push your timesheets. To install the Zebra backend for example (again, **no** sudo or root user needed):

```
$ python3 -m pip install --user taxi-zebra
```

To upgrade Taxi and the Zebra plugin, run `python3 -m pip install --user --upgrade taxi taxi-zebra`

Debian & Ubuntu

Run the following commands to add the Taxi repository and install it along with the Zebra backend:

```
sudo apt install apt-transport-https
wget 'https://taxi-packages.liip.ch/taxi-packages.liip.ch.key' -O - | sudo apt-key add
echo "deb [arch=amd64] https://taxi-packages.liip.ch/ unstable-ci main" | sudo tee /
↪etc/apt/sources.list.d/taxi.list
sudo apt update
sudo apt install taxi taxi-backend-zebra
```

NixOS

If you're running NixOS, you can then install it declaratively by adding it to your `/etc/nixos/configuration.nix` file and then running `nixos-rebuild switch`:

```
let
  taxi = import <taxi>;
in
environment.systemPackages = [
  # ...
  taxi.taxi
]
```

Nix

Create a `flake.nix` file in a directory with the following contents:

```
{
  inputs.taxi.url = "github:sephii/taxi";
  inputs.flake-utils.url = "github:numtide/flake-utils";

  outputs = { self, nixpkgs, taxi, flake-utils }:
  flake-utils.lib.eachDefaultSystem (system: {
    packages.taxi = taxi.defaultPackage.${system}.withPlugins
      (plugins: [ plugins.clockify plugins.zebra ]);
    });
}
```

In this example, the clockify and zebra plugins are enabled. Feel free to adapt this file with the plugins you want to enable. You'll find a list of available plugins in *the ‘‘availablePlugins‘* attribute of the `pkgs.nix` file <https://github.com/sephii/taxi/blob/main/pkgs.nix#L121>‘_.

Make sure your `flake.nix` and `flake.lock` files exist and are version controlled:

```
git init
git add flake.nix
nix flake lock
git add flake.lock
git commit -m "Init taxi"
```

Add the flake to your registry:

```
nix registry add taxi /path/to/your/flake/dir
```

Now you can finally install the package:

```
nix profile install taxi#taxi
```

Running the `taxi` command should now work!

To upgrade taxi, `cd` to the directory where you created the flake and run:

```
nix flake lock --update-input taxi
git add flake.nix flake.lock
git commit -m "Update taxi"
nix registry pin taxi
nix profile install taxi#taxi
```

NixOS

Use the overlay in `taxi.overlay`:

```
{
  inputs.taxi.url = "github:sephii/taxi";

  outputs = attrs@{ nixpkgs, taxi, ... }: let
    pkgs = import nixpkgs {
      overlays = [ taxi.overlay ];
    };
  in {
    nixosConfigurations.myConfig = nixpkgs.lib.nixosSystem {
      modules = [
        ({ pkgs, ... }: { environment.systemPackages = [ pkgs.taxi-cli.withPlugins␣
↪(plugins: [ plugins.clockify plugins.zebra ]) ]; })
      ];
    }
  }
}
```

Adapt the configuration depending on the plugins you need. You'll find a list of available plugins in *the ''available-Plugins'* attribute of the `pkgs.nix` file <https://github.com/sephii/taxi/blob/main/pkgs.nix#L121>'_.

5.1.2 Common installation issues

taxi: command not found

This usually means the Python user binary path (where the `taxi` binary is installed) is not in your `PATH` environment variable.

Run the following command to identify the Python user binary path:

```
$ python3 -c "import os, site; print(os.path.join(site.getuserbase(), 'bin'))"
/home/sephi/.local/bin
```

Add this directory to your `PATH` environment variable, for example by following this guide.

python3: command not found

Run the following command:

```
$ python --version
Python 3.8.5
```

Check that the version is at least 3.7. If that's the case, replace `python3` by `python` when running commands. If that's not the case, install Python 3.

5.1.3 First steps with Taxi

Once Taxi is installed, you'll probably want to fetch the projects list from your backend:

```
taxi update
```

Since this is the first time you run Taxi, you'll get asked a few questions:

```
Welcome to Taxi!
================

It looks like this is the first time you run Taxi. You will need a
configuration file (~/.config/taxi/taxirc) in order to proceed.
Please answer a few questions to create your configuration file.

Backend you want to use (choices are dummy, zebra): zebra
Username or token: b4b8123f4addb27ad0eb0b2b0a0ae81730af96b8
Password (leave empty if you're using a token) []:
Editor command to edit your timesheets [vim]:
Hostname of the backend (eg. timesheets.example.com): zebra.example.com
```

Taxi is now ready to use! Let's start by recording the time we spent installing Taxi:

```
taxi edit
```

Note: If you didn't choose the correct editor when running Taxi for the first time you might get into an editor called *vim* at this point. To exit it, type *:q!*. Then to manually set the editor Taxi should use, open your Taxi configuration file (by using the command *taxi config*), and change the value of the *editor* setting to the editor you want. If you're using Linux, you might put *gedit*. If you're using OS X, you might put *open -a TextEdit*.

Your editor will pop up and you'll see the current date has been automatically added for you. Let's add an entry so your file looks something like that:

```
09/05/2016

intro 10:15-10:30 Install Taxi
```

An entry consists of 3 parts:

- An alias (*intro*)
- A duration (*10:15-10:30*)
- A description (*Install Taxi*)

Aliases allow you to map meaningful names to activity ids. At that point you'll probably don't really know what alias to use, so let's just try that for now and we'll see what Taxi has to say about it.

Save the file and close your editor. You should see Taxi displaying a summary of what you did:

```
Staging changes :

Monday 09 may

intro (inexistent alias)        0.25  Install Taxi
    Did you mean one of the following: _internal, _infra, _interview?
                                0.25

Total                           0.25

Use `taxi ci` to commit staging changes to the server
```

Note: Depending on the editor you're using you might not see anything happening when you close the file and you might need to run *taxi status* to get this output.

Whoops! It looks like the alias we used doesn't exist. Taxi tried to help us by suggesting similar matches among available aliases, and actually *_internal* looks like the correct alias to use. We could have searched for aliases that look like *internal* with the following command: `taxi alias list internal`.

Note: This alias *_internal* exists because we ran *taxi update* before, which synchronized the aliases database from the remote backend. You can also use custom aliases that will not be shared with the remote backend. Refer to the *alias* command help by running `taxi alias --help`.

Let's edit our file once again and fix that:

```
taxi edit
```

Replace the *intro* alias with *_internal*:

```
09/05/2016

_internal 10:15-10:30 Install Taxi
```

Close your editor and run *taxi status* if needed and check the output:

```
Staging changes :

Monday 09 may

_internal (7/16, liip)          0.25  Install Taxi
                                0.25

Total                           0.25

Use `taxi ci` to commit staging changes to the server
```

You can now see the *_internal* alias has been recognized as mapped to project id 7, activity id 16 on the *liip* backend. If you're satisfied with that, you can now push this to the remote server (*ci* is a shorthand for *commit*, which is equivalent):

```
taxi ci
```

Searching for aliases

The whole point of Taxi is to record your time spend on activities, but how do you know which activities you can use? As explained in the introduction, activities are fetched with the *update* command. To see the available aliases, use the *alias list* command:

```
$> taxi alias list

[dummy] my_alias -> 2000/11 (My project, my activity)
```

The part that appears in brackets is the backend that will be used to push the entries when using the *commit* command. The information on the right of the arrow is the "mapping", that is a project id and an activity id, whose names are in parentheses.

You can search for a specific alias by adding a search string to the *alias list* command:

```
$> taxi alias list my_awesome_alias
```

You can also limit the results to aliases you have already used in your timesheets with the *–used* option:

```
$> taxi alias list --used
```

Filtering entries

The *status* and *commit* options support the *–since*, *–until* and *–today/–not-today* options that allow you to specify which entries should be included in the command. For example let's say you entered entries for yesterday and today (Wednesday 21 june):

```
$> taxi status
Staging changes :

Tuesday 20 june

_internal                       0.25  Install Taxi
                                0.25
Wednesday 21 june

_internal                       1.00  First steps with Taxi
                                1.00

Total                           1.25

Use `taxi ci` to commit staging changes to the server
```

And you only want to commit yesterday's entry. You can use the *–not-today* option that will ignore today's entries. Since you can use this option both with the *status* and *commit* command, you can review what you're about to commit with the *status* command:

```
$> taxi status --not-today
Staging changes :

Tuesday 20 june

_internal                       0.25  Install Taxi
                                0.25

Total                           0.25

Use `taxi ci` to commit staging changes to the server
```

If you wanted to only include today's entries, you could use the *–since* option. Both *–since* and *–until* support the following notations:

- Relative: 5 days ago, 2 weeks ago, 1 month ago, 1 year ago, today, yesterday
- Absolute: 21.05.2017

Back to our entries, let's filter yesterday's entry:

```
$> taxi status --since=today
Staging changes :

```

(continues on next page)

(continued from previous page)

```
Wednesday 21 june

_internal                       1.00  First steps with Taxi
                                1.00

Total                           1.00

Use `taxi ci` to commit staging changes to the server
```

In fact, the *–today* option is just a shortcut for *–since=today –until=today*.

Ignored entries

You'll sometimes have entries for which you're not sure which alias you should use and that shouldn't be pushed until you have a confirmation from someone else. Simply prefix the entry line with *?* and the entry will be ignored. If we run the `edit` command and add a question mark to our `pingpong` alias like so:

```
23/02/2015

? pingpong 09:00-10:30 Play ping-pong
```

The output becomes:

```
Staging changes :

Monday 23 february
pingpong (ignored)             1.50  Play ping-pong
                               1.50

Total                          1.50

Use `taxi ci` to commit staging changes to the server
```

Entry continuation

Having entries that follow each other, eg. 10:00-11:00, then 11:00-13:00, etc is a common pattern. That's why you can skip the start time of an entry if the previous entry has an end time. The previous example would become (note that spaces don't matter, you don't need to align them):

```
23/02/2015

pingpong 09:00-10:30 Play ping-pong
taxi          -12:00 Write documentation
```

You can also chain them:

```
23/02/2015

pingpong 09:00-10:30 Play ping-pong
taxi          -12:00 Write documentation
internal      -13:00 Debug coffee machine
```

Internal aliases

Some people like to timesheet everything they do: lunch, ping-pong games, going to the restroom… anyway, if you're that kind of people you probably don't want these entries to be pushed. To achieve that, start by adding a dummy backend to your configuration file (to open it, run *taxi config*):

```
[backends]
internal = dummy://
```

Then to add an internal alias, either add it in the corresponding section in your configuration file:

```
[internal_aliases]
_pingpong
_lunch
_shit
```

Or use the `alias` command:

```
taxi alias add -b internal _pingpong ""
```

Getting help

Use `taxi <command> --help` to get help on any Taxi command.

5.1.4 Upgrading Taxi

To upgrade Taxi, run `python3 -m pip install --upgrade taxi`. If you have any plugins, you'll also need to manually upgrade them, by running for example `python3 -m pip install --upgrade taxi-zebra`.

5.1.5 Timesheet syntax

Taxi uses a simple syntax for timesheets, which are composed of dates and entries. If you used the `edit` command, you already saw the dates. A date is a string that can have one of the following formats:

- dd/mm/yyyy
- dd/mm/yy
- yyyy/mm/dd

Actually the separator can be any special character. You can control the format Taxi uses when automatically inserting dates in your entries file with the *date_format* configuration option.

Timesheets also contain comments, which are denoted by the `#` character. Any line starting with `#` will be ignored.

Entries are the entity that allow you to record the time spent an various activities. The basic syntax is:

```
alias duration description
```

`alias` can be any string matching a mapping defined either by your configuration, or a shared alias. If an alias is not found in the configured aliases, a list of suggestions will be given and the alias will be ignored when pushing entries.

`duration` can either be a time range or a duration in hours. If it's a time range, it should be in the format `start-end`, where `start` can be left blank if the previous entry also used a time range and had a time defined, and `end` can be `?` if the end time is not known yet, leading to the entry being ignored. Each part of the range should have the format `HH:mm`, or `HHmm`. If `duration` is a duration, it should just be a number, eg. 2 for 2 hours, or 1.75 for 1 hour and 45 minutes.

`description` can be any text but cannot be left blank.

5.1.6 Backends

Note: The *plugin* command is available starting from Taxi 4.2.

Backends are provided through Taxi plugins. To install (or upgrade) a plugin, use the *plugin install* command:

```
taxi plugin install zebra
```

This will fetch and install the backend plugin. Once installed, you'll still need to tell Taxi to use it. This is explained in the next section.

You can also see which plugins are installed with *plugin list*:

```
$> taxi plugin list
zebra (1.2.0)
```

Note: This is only valid if you installed Taxi with the install script, that transparently deals with installing Taxi in an isolated environment. If you installed it differently (eg. by using a Debian package or by using pip), either install the corresponding Debian package for the backend you want to use or use pip (eg. `pip install taxi-zebra`).

Configuration

You can open your configuration file using the command *taxi config*.

The configuration file uses the XDG user directories specification. This means the location is the following:

- Linux: `~/.config/taxi/taxirc`
- OS X: `~/Library/Application Support/taxi/taxirc`
- Windows: `%LOCALAPPDATA%\sephii\taxi\taxirc` or `C:\Users\<User>\AppData\Local\sephii\taxi\ta`

You can see the location of the configuration file used by running taxi in verbose mode, for example:

```
$ taxi -vvv status
DEBUG:root:Using configuration file in /home/sephi/.config/taxi/taxirc
...
```

The configuration file has a section named `backends` that allows you to define the active backends and the credentials you want to use. The syntax of the backends part is:

```
[backends]
default = <backend_name>://<user>:<password>@<host>:<port><path><options>
```

Here a backend named *default* is defined. The `backend_name` is the adapter this backend will use. You'll find this name in the specific backend package documentation. The `backend_name` is the only mandatory part, as some backends won't care about the `user`, `password`, or other configuration options.

The name of each backend should be unique, and it will be used when defining aliases. Each backend will have a section named `[backend_name_aliases]` and `[backend_name_shared_aliases]`, where *backend_name* is the name of the backend, each containing the user-defined aliases, and the automatic aliases fetched with the `update` command.

Note: If you have any special character in your password, make sure it is URL-encoded, as Taxi won't be able to correctly parse the URI otherwise. You can use the following snippet to encode your password:

```
>>> import urllib
>>> urllib.quote('my_password', safe='')
```

On Python 3:

```
>>> from urllib import parse
>>> parse.quote('my_password', safe='')
```

5.1.7 Configuration options

auto_add

Default: auto

This specifies where the new entries will be inserted when you use *start* and *edit* commands. Possible values are *auto* (automatic detection based on your current entries), *bottom* (values are added to the end of the file), or *top* (values are added to the top of the file) or *no* (no auto add for the edit command).

auto_fill_days

Default: 0,1,2,3,4

When running the *edit* command, Taxi will add all the dates that are not present in your entries file until the current date if they match any day present in `auto_fill_days` (0 is Monday, 6 is Sunday). You must have *auto_add* set to something else than *no* for this option to take effect.

date_format

Default: %d/%m/%Y

This is the format of the dates that'll be automatically inserted in your entries file(s), for example when using the *start* and *edit* commands. You can use the same date placeholders as for the *file* option.

editor

When running the *edit* command, your editor command will be deducted from your environment but if you want to use a custom command you can set it here.

file

Default: ~/zebra/%Y/%m.tks

The path of your entries file. You're free to use a single file to store all your entries but you're strongly encouraged to use date placeholders here. The following will expand to `~/zebra/2011/11.tks` if you're in November 2011.

You can use any datetime format code defined in the strftime documentation down to a resolution of a day (hours, minutes and seconds format codes are not supported because they make little sense).

regroup_entries

Default: true

If set to false, similar entries (ie. entries on the same date that are on the same alias and have the same description) won't be regrouped.

Note: This setting is available starting from Taxi 4.1

nb_previous_files

Default: 1

Defines the number of previous timesheet files Taxi should try to parse. This allows you to make sure you don't forget hours in files from previous months when starting a new month.

This option only makes sense if you're using date placeholders in *file*.

round_entries

Default: 15

Number of minutes to round entries duration to when using the *stop* command. For example, if you start working on a task at 10:02 and you run *taxi stop* at 10:10 with the default *round_entries* setting you'll get *10:02-10:17*. Note that entries are always rounded up, never down.

5.1.8 Flags characters customization

By default Taxi uses the = character for pushed entries and *?* for ignored entries. You can customize them in the *[flags]* section of the configuration file. Note that using # as a flag character will make any flagged entry interpreted as a comment and won't be parsed by Taxi. Example of using custom characters for the *ignored* and *pushed* flags:

```
[flags]
ignored = !
pushed = @
```

5.2 Developer guide

5.2.1 Timesheets and entries

The `Entry` class is the base of Taxi. An entry is the record of an activity for a certain period of time. It consists of an activity, a time span, and a description:

```
>>> from taxi.timesheet import Entry
>>> my_entry = Entry('_internal', 1, 'Play ping-pong')
```

Entries duration can be expressed either as a fixed duration, in hours (eg. 0.5 for half an hour), or as time spans. Time span notation works with 2-items tuples, like so: *(start_time, end_time)*. In the following example, the entry starts at 9:30 and ends at 10, thus having a duration of half an hour:

```
>>> from datetime import time
>>> my_entry = Entry('_internal', (time(9, 30), time(10)), 'Play ping-ping')
>>> my_entry.hours
0.5
```

end_time can be left blank, in that case the entry will be considered as being "in progress". This is useful in certain situations, for example the *start* command uses this feature to start an entry which can then be "stopped" with the *stop* commands (which detects the last in-progress activity and sets its end time).

```
>>> my_entry = Entry('_internal', (time(9), None), 'Play ping-pong')
>>> my_entry.hours
0
>>> my_entry.in_progress
True
```

Now we know how to create entries, we can put them together in timesheets, which is a collection of entries and dates. You might have noticed entries don't have an associated date: that's because the link between dates and entries is in the timesheet itself. Let's create a timesheet:

```
>>> from datetime import date
>>> from taxi.timesheet import Timesheet
>>> timesheet = Timesheet()
>>> timesheet.entries
{}
```

Now we have a timesheet, we can start adding entries to it:

```
>>> timesheet.entries.add(date(2017, 6, 7), my_entry)
>>> timesheet.entries
{datetime.date(2017, 6, 7): [<Entry: "_internal 0 Play ping-pong">]}
```

You can dump the timesheet contents by casting it to a string:

```
>>> str(timesheet)
'07.06.2017\n\n_internal 09:00-? Play ping-pong'
```

Entries also have flags: *pushed* and *ignored*. Ignored and pushed entries will be excluded from the commit process:

```
>>> my_entry = Entry('_internal', 1, 'Play ping-pong')
>>> my_entry.ignored = True
>>> timesheet = Timesheet()
>>> timesheet.entries.add(date(2017, 6, 7), my_entry)
>>> str(timesheet)
'07.06.2017\n\n? _internal 1 Play ping-pong'
```

5.2.2 Loading and saving timesheets

Use the *load* method to create a timesheet from a file:

```
>>> timesheet = Timesheet.load('/tmp/timesheet.tks')
>>> timesheet.entries.add(date(2017, 6, 7), Entry('_internal', 1, 'Play ping-pong'))
>>> timesheet.save()
```

You can also save the timesheet to a different file from the file it was loaded from:

```
>>> timesheet.save('/tmp/new_timesheet.tks')
```

5.2.3 Timesheet collections

Dealing with multiple timesheets is achieved through the `taxi.timesheet.TimesheetCollection` class. This is useful if you want to run operations on multiple timesheets in a single command. The *TimesheetCollection* class proxies all calls to the associated timesheets and aggregates the results. The following example illustrates how the *entries* attribute from a timesheet collection can be used to transparently access entries from all associated timesheets:

```
>>> from taxi.timesheet import TimesheetCollection
>>> timesheets = [Timesheet(), Timesheet()]
>>> timesheets[0].entries.add(date(2017, 6, 8), Entry('_internal', 1, 'Play ping-pong
↪'))
>>> timesheets[1].entries.add(date(2017, 7, 8), Entry('_internal', 1, 'Play ping-pong
↪'))
>>> timesheet_collection = TimesheetCollection(timesheets)
>>> timesheet_collection.entries
{datetime.date(2017, 6, 8): [<Entry: ...>], datetime.date(2017, 7, 8): [<Entry: ...>]}
>>> timesheet_collection.get_hours()
2
```

5.2.4 Creating a backend

A backend is a Python package that can be installed independently of Taxi and that persists the entries transmitted by the `commit` command. To create a backend, you'll need to create a new Python package, which is hopefully quite easy to do.

As an example, we'll build a simple backend that sends the timesheets it receives by mail. We'll call it `taxi_mail`.

Registering the backend

A backend provides functionality but should not contain harcoded configuration such as usernames or passwords. Think about other people who will want to use your backend, they'll probably don't have the same credentials as you.

A backend is defined and configured by a URI that allows you to configure it. The full syntax is:

```
[backends]
default = <backend_name>://<user>:<password>@<host>:<port><path><options>
```

Your backend obviously doesn't have to use all the parts of the URI. For example an unauthenticated backend won't need any user or password, and the user is allowed to leave them blank in the configuration file.

Let's start to write our backend. The first thing you'll want to do is define a `setup.py` file. Here's an example:

```
#!/usr/bin/env python
from setuptools import find_packages, setup

```

(continues on next page)

(continued from previous page)

```
setup(
    name='taxi_mail',
    version='1.0',
    packages=find_packages(),
    description='Mail backend for Taxi',
    author='Me',
    author_email='me@example.com',
    url='https://github.com/me/taxi-mail',
    license='wtfpl',
    entry_points={
        'taxi.backends': 'smtp = taxi_mail.backend:MailBackend'
    }
)
```

The important part is the `entry_points`. This is what will tell Taxi the class to use for the backend. The key `smtp` is the name of the backend. This is what the user will put in `<backend_name>` in the configuration file. The part `taxi_mail.backend:MailBackend` is the path to our backend class. This basically means `from taxi_mail.backend import MailBackend`.

Let's create the backend class:

```
# file: taxi_mail/backend.py

from taxi.backends import BaseBackend

class MailBackend(BaseBackend):
    pass
```

The first thing our backend will need to do is store the information we want from the URI so that we can use it later. The `BaseBackend` already defines an `__init__` method that stores all the parts of the backend URI so there isn't much to do. Let's think about how the user will configure our backend. The following syntax would probably make sense:

```
[backends]
mail = smtp://user:password@smtp.gmail.com/me@example.com
```

We decided to use the `<path>` part for the e-mail address of the recipient. There's one detail though: the path here is `/me@example.com`, so we need to get rid of that initial slash. Let's do it:

```
class MailBackend(BaseBackend):
    def __init__(self, **kwargs):
        super(MailBackend, self).__init__(**kwargs)
        self.path = self.path.lstrip('/')
```

Pushing entries

We now have all the information we need to send mails. For the actual sending, we could implement the `push_entry` method. However this will fire for every entry, which means we would get one mail per entry. Obviously this is not what we want, but hopefully you can implement the `post_push_entries` method, which is called once after all entries have been committed. This method also gives you a chance to raise an exception for failing entries.

So let's buffer the entries to put in the mail in the `push_entry` method and send them all in the `post_push_entries` method. The code could look like that:

```
from collections import defaultdict
import smtplib

from taxi.backends import BaseBackend

class MailBackend(BaseBackend):
    def __init__(self, **kwargs):
        super(MailBackend, self).__init__(**kwargs)
        self.path = self.path.lstrip('/')
        self.entries = defaultdict(list)

    def push_entry(self, date, entry):
        self.entries[date].append(entry)

    def post_push_entries(self):
        timesheet = []

        for date, entries in self.entries.items():
            timesheet.append(date.strftime('%d %m %Y'))

            for entry in entries:
                timesheet.append(str(entry))

        smtp = smtplib.SMTP_SSL(self.hostname)
        smtp.login(self.username, self.password)
        smtp.sendmail('taxi@example.com', self.path, '\n'.join(timesheet))
        smtp.quit()
```

Note that for the sake of brevity, we didn't catch any exception at all in this example. It's of course a good idea to do it, so that the user knows why the entries couldn't be pushed. If your backends raises an exception, all entries will be considered to have failed and will be reported as such. If you want to report only certain entries as failed in `post_push_entries`, raise a `PushEntriesFailed` exception, with a parameter `entries` that will be a *entry: error* dictionary.

We now have a fully working backend that can be used to push entries!

5.2.5 Creating custom commands

Taxi will load any module defined in the `taxi.commands` entry point. Let's create a `current` command that displays the path to the current timesheet. First, let's create the command (in `taxi_current/commands.py`):

```
import click

from taxi.commands.base import cli

@cli.command()
@click.pass_context
def current(ctx):
    timesheet_path = ctx.obj['settings'].get_entries_file_path(expand_date=True)
    click.echo("Current timesheet path is " + timesheet_path)
```

The `cli.command` part allows us to create a Taxi subcommand. For more information on how to use Click, refer to the official Click documentation. Also feel free to check the source code of the existing commands that can give a good base to start from.

As with custom backend creation, your package should also have a `setup.py` file. The commands module should be registered in the `taxi.commands` entry point (in the `setup.py` file):

```
#!/usr/bin/env python
from setuptools import find_packages, setup

setup(
    name='taxi_current',
    version='1.0',
    packages=find_packages(),
    description='Show current timesheet',
    author='Me',
    author_email='me@example.com',
    url='https://github.com/me/taxi-current',
    license='wtfpl',
    entry_points={
        'taxi.commands': 'current = taxi_current.commands'
    }
)
```

That's it! If you install your custom plugin (eg. with `./setup.py install` or by using `./setup.py develop` as explained in the *Getting a development environment* section, you will now be able to type `taxi current`!

5.2.6 Getting a development environment

Start by cloning Taxi (you'll probably want to use your fork URL instead of the public URL):

```
git clone https://github.com/sephii/taxi
```

Then create a virtual environment with mkvirtualenv:

```
mkvirtualenv taxi
```

Now run the setup script to create the development environment:

```
./setup.py develop
```

Now every time you'll want to work on taxi, start by running `workon taxi` first, so that you're using the version you checked out instead of the system-wide one.

5.2.7 Running tests

Setup a virtual environment as explained in the previous section, then install the test requirements in it:

```
pip install -r requirements_test.txt
```

To run the tests, run the following command:

```
pytest
```

When developing it's useful to only run certain tests, for this, use the following command:

```
pytest tests/commands/test_alias.py::AliasCommandTestCase::test_alias_list
```

You can also leave out `::test_alias_list` to run all tests in the `AliasCommandTestCase`, or leave out `::AliasCommandTestCase` as well if you have multiple test classes and you want to run them all.

5.3 API documentation

5.3.1 Backends

The role of a backend is to handle the communication with a backend tool that will store the timesheets and provide projects and activities.

class `taxi.backends.`**`BaseBackend`**(*username*, *password*, *hostname*, *port*, *path*, *options*, *context*)

All Taxi backends should inherit from the *`BaseBackend`* class. Backends are usually constructed from a URL in the form *<backend_name>://<username>:<password>@<hostname>:<port><path>?<options>*. The `PluginsRegistry` takes care of the parsing and the instanciation of the backend objects. The *options* parameter is a dictionary constructed from the backend URL querystring.

Construct the backend.

`get_projects`()

Return a list of projects and activities. These will be then stored for further use. The list should contain `Project` objects.

`post_push_entries`()

Called after the entries have been pushed. Useful if you need to do post-processing like closing connection, or sending entries buffered in *`push_entry()`*.

If an exception is raised in this method, the status of all the entries of the backend will be considered failed. You can also raise *`PushEntriesFailed`* with a custom user message to mark their status as failed. If you want to mark individual entries as failed, raise *`PushEntriesFailed`* with `entries` being a dictionary containing entries as keys, and error messages as values.

`push_entry`(*date*, *entry*)

Called when an entry should be pushed to the backend. *date* is a `datetime.date` object. *entry* is a `TimesheetEntry` object.

If the push fails, this method should raise a *`PushEntryFailed`* exception.

exception `taxi.backends.`**`PushEntriesFailed`**(*message=None*, *entries=None*)

Exception indicating that a set of entries couldn't be pushed. Typically raised by *`BaseBackend.post_push_entries()`*.

If `entries` is set, it should be a dictionary mapping `taxi.timesheet.entry.TimesheetEntry` with errors as strings.

exception `taxi.backends.`**`PushEntryFailed`**

Exception indicating that an entry couldn't be pushed.

Plugins

Exceptions

5.3.2 Timesheets

Timesheet lines

class `taxi.timesheet.lines.`**`DateLine`**(*date*, *text=None*)

Represents a date in a timesheet.

`is_date_line = True`

class taxi.timesheet.lines.**TextLine**(*text*)

The TextLine is either a blank line or a comment line.

is_text_line = True

Timesheet parsing

Exceptions

Python Module Index

Index