Maestral API Documentation

Release 1.5.2

Sam Schott

Feb 10, 2022

BACKGROUND

This documentation provides an API reference for the maestral daemon. It is built from the current dev branch and is intended for developers. For a user manual and an overview of Maestral's functionality, please refer to maestral.app.

CHAPTER

ONE

SYNC LOGIC

The `maestral.sync.SyncEngine` class provides access to the current sync state through its properties and provides the methods which are necessary to complete an upload or a download sync cycle. This includes methods to wait for and return local and remote changes, to sort those changes and discard any excluded items and to apply local changes to the Dropbox server and vice versa.

The `maestral.sync.SyncMonitor` class coordinates the sync process with its threads for startup, download-sync, upload-sync and periodic maintenance.

1.1 Processing of sync events

Remote events come in three types: `dropbox.files.DeletedMetadata`, `dropbox.files.FolderMetadata` and `dropbox.files.FileMetadata`. The Dropbox API does not differentiate between created, moved or modified events. Maestral processes remote events as follows:

1) `SyncEngine.wait_for_remote_changes()` blocks until remote changes are available.
2) `SyncEngine.list_remote_changes()` lists all remote changes since the last sync. Those events are processed at follows:
    - Events for entries which are excluded by selective sync and hard-coded file names which are always excluded (e.g., '.DS_Store') are filtered out at this stage.
    - Multiple events per file path are combined to one. This is rarely necessary, Dropbox typically already provides only a single event per path but this is not guaranteed and may change. One exception is sharing a folder: Dropbox does this by removing the folder from the user's root and re-mounting it as a shared folder. This produces at least one DeletedMetadata and one FolderMetadata event. If querying for changes *during* this process, multiple `dropbox.files.DeletedMetadata` events may be returned.
    - If a file / folder event implies a type changes, e.g., replacing a folder with a file, we explicitly generate the necessary `dropbox.files.DeletedMetadata` here to simplify conflict resolution.
3) `SyncEngine.apply_remote_changes()`: Sorts all events hierarchically, with top-level events coming first. Deleted and folder events are processed in order, file events in parallel with up to 6 worker threads.
4) `SyncEngine.notify_user()`: Shows a desktop notification for the remote changes.

Local file events come in eight types: For both files and folders we collect created, moved, modified and deleted events. They are processed as follows:

1) `SyncEngine.wait_for_local_changes()`: Blocks until local changes are registered by `FSEventHandler`.
2) `SyncEngine.list_local_changes()`: Lists all local file events. Those are processed as follows:
    - Events ignored by a "mignore" pattern as well as hard-coded file names and changes in our cache path are filtered out at this stage.

- Events are further cleaned up to return the minimum number of events necessary to reproduce the actual changes: Multiple events per path are combined into a single event which reproduces the file change. The only exception is when the entry type changes from file to folder or vice versa: in this case, both deleted and created events are kept. Further, when a whole folder is moved or deleted, we discard the moved or deleted events for its children.

2) `SyncEngine.apply_local_changes()`: Sorts local changes hierarchically and applies events in the order of deleted, folders and files. Deleted, created and modified events will be applies to the remote Dropbox in parallel with up to 6 threads. Moves will be carried out synchronously.

Before processing, we convert all Dropbox metadata and local file events to a unified format of `maestral.database.SyncEvent` instances which are also used to store the sync history data in our SQLite database.

1.2 Detection and resolution of sync conflicts

Sync conflicts during a download are detected by comparing the file's "rev" with the locally saved revision identifier in Maestral's index. We assign folders a rev of `'folder'` and deleted / non-existent items a rev of `None`.

1. If both revs are equal, the local item is either the same as on Dropbox or newer and the local changes haven't been uploaded and committed to our index yet. No download sync occurs (including deletion of the local file).
2. If revs are different, we compare content hashes. If those hashes are equal, no download occurs.
3. If content hashes are different, we check if the local item has been modified since the last download sync. In case of a folder, we take the most recent change of any of its children. If the local entry has not been modified since the last sync, it will be replaced. Otherwise, we create a conflicting copy.

Conflict resolution for uploads is handled as follows:

1. For created and moved events, we check if the new path has been excluded by the user with selective sync but still exists on Dropbox. If yes, it will be renamed by appending "(selective sync conflict)".
2. On case-sensitive file systems, we check if the new path differs only in casing from an existing path. If yes, it will be renamed by appending "(case conflict)".
3. If a file has been replaced with a folder or vice versa, we check if any un-synced changes will be lost by replacing the remote item and create a conflicting copy if necessary.
4. For created or modified files, check if the local content hash equals the remote content hash. If yes, we don't upload but update our rev number. If no, we upload the changes and specify the rev which we want to replace or delete. If the remote item is newer (different rev), Dropbox will handle conflict resolution for us.
5. We finally confirm the successful upload and check if Dropbox has renamed the item to a conflicting copy. In the latter case, we apply those changes locally.

CHAPTER

TWO

LOGGING

Maestral makes extensive use of Python's logging module to collect debug, status and error information from different parts of the program and distribute it to the appropriate channels.

Broadly speaking, the builtin log levels are used as follows:

Level	Typical messages
DEBUG	Individual sync progress, conflict resolution, daemon startup and environment info
INFO	Cumulative sync progress, sync errors that require user resolution (e.g., insufficient space)
WARNING	Failure to initialise non-critical functionality (e.g., desktop notifications, sytem keyring)
ERROR	Errors that prevent all syncing (revoked access token, deleted folder, etc)

Maestral defines a number of log handlers to process those messages, some of them for internal usage, others for external communication. For instance, cached logging handlers are used to populate the public APIs `Maestral.status` and `Maestral.fatal_errors` and therefore use fixed log levels. Logging to stderr, the systemd journal (if applicable) and to our log files uses the user defined log level from `Maestral.log_level` which defaults to INFO.

Target	Log level	Enabled
Log file	User defined (default: INFO)	Alyways
Stderr	User defined (default: INFO)	If verbose flag is passed
Systemd journal	User defined (default: INFO)	If started as systemd service
Systemd notify status	INFO	If started as systemd notify service
`Maestral.status` API	INFO	Alyways
Desktop notifications	WARNING	Alyways
`Maestral.fatal_errors` API	ERROR	Alyways

All custom handlers are defined in the `maestral.logging` module. Maestral also subclasses the default `logging.LogRecord` to guarantee that any surrogate escape characters in file paths are replaced before emitting a log and flushing to any streams. Otherwise, incorrectly encoded file paths could prevent logging from working properly in exactly those cases where it would be particularly useful.

CHAPTER

THREE

CONFIG FILES

The config files are located at `$XDG_CONFIG_HOME/maestral` on Linux (typically `~/.config/maestral`) and `~/Library/Application Support/maestral` on macOS. Each configuration will get its own INI file with the settings documented below.

Config values for `path` and `excluded_items` should not be edited manually but rather through the corresponding CLI commands or GUI options. This is because changes of these settings require Maestral to perform accompanying actions, e.g., download items which have been removed from the excluded list or move the local Dropbox directory. Those will not be performed if the user edits the options manually.

This also holds for the `account_id` which will be written to the config file after successfully completing the OAuth flow with Dropbox servers.

Any changes will only take effect once Maestral is restarted. Any changes made to the config file may be overwritten without warning if made while the sync daemon is running.

```
[main]

# Config file version (not the Maestral version!)
version = 15.0.0

[auth]

# Unique Dropbox account ID. The account's email
# address may change and is therefore not stored here.
account_id = dbid:AABP7CC5bpYd8ghjIColDFrMoc9SdhACA4

# The keychain to use to store user credentials. If "automatic",
# will be set automatically from available backends when
# completing the OAuth flow. Mus be a fully qualified class name.
keyring = keyring.backends.macOS.Keyring

[app]

# Level for desktop notifications:
# 15 = FILECHANGE
# 30 = SYNCISSUE
# 40 = ERROR
# 100 = NONE
notification_level = 15

# Level for log messages:
# 10 = DEBUG
```

(continues on next page)

(continued from previous page)

```
# 20 = INFO
# 30 = WARNING
# 40 = ERR0R
log_level = 20

# Interval in sec to check for updates
update_notification_interval = 604800

[sync]

# The current Dropbox directory
path = /Users/UserName/Dropbox (Maestral)

# List of excluded files and folders
excluded_items = ['/test_folder', '/sub/folder']

# Interval in sec to perform a full reindexing
reindex_interval = 604800

# Maximum CPU usage per core
max_cpu_percent = 20.0

# Sync history to keep in seconds
keep_history = 604800

# Enable upload syncing
upload = True

# Enable download syncing
download = True
```

CHAPTER

FOUR

STATE FILES

Maestral saves its persistent state in two files: `{config_name}.db` for the file index and `{config_name}.state` for anything else. Both files are located at `$XDG_DATA_DIR/maestral` on Linux (typically `~/.local/share/maestral`) and `~/Library/Application Support/maestral` on macOS. Each configuration will get its own state file.

4.1 Database

The index is stored in a SQLite database with contains the sync index, the sync event history of the last week and a cache of locally calculated content hashes. We use our own light-weight ORM layer, defined in `maestral.utils.orm`, to manage the mapping between Python objects and database rows. The actual table declarations are given by the definitions of `maestral.database.IndexEntry`, `maestral.database.SyncEvent` and `maestral.database.HashCacheEntry`.

4.2 State file

The state file has the following sections:

```
[main]

# State file version (not the Maestral version!)
version = 12.0.0

[account]

email = foo@bar.com
display_name = Foo Bar
abbreviated_name = FB
type = business
usage = 39.2% of 1312.8TB used
usage_type = team

# Information about the user's root namespace. This
# may be a personal namespace or a Team Space for certain
# Dropbox Business accounts.
path_root_type = team
path_root_nsid = 1287234
home_path = /User Name
```

(continues on next page)

(continued from previous page)

```
[auth]

# The type of OAuth access token used:
# legacy: long-lived token
# offline: short-lived token with long-lived refresh token
token_access_type = offline

[app]

# Version for which update / migration scripts have
# run. This is bumped to the currently installed
# version after an update.
updated_scripts_completed = 1.2.0

# Time stamp of last update notification
update_notification_last = 0.0

# Latest avilable release
latest_release = 1.2.0

[sync]

# Cursor reflecting last-synced remote state
cursor = ...

# Time stamp reflecting last-synced local state
lastsync = 1589979736.623609

# Time stamp of last full reindexing
last_reindex = 1589577566.8533309

# Lower case Dropbox paths with upload sync errors
upload_errors = []

# Lower case Dropbox paths with download sync errors
download_errors = []

# Lower case Dropbox paths of interrupted uploads
pending_uploads = []

# Lower case Dropbox paths of interrupted downloads
pending_downloads = []
```

Notably, account info which can be changed by the user such as the email address is saved in the state file while only the fixed Dropbox ID is saved in the config file.

CHAPTER

FIVE

CONTRIBUTING

Thank you for your interest in contributing!

5.1 Code

To start, install maestral with the `dev` extra to get all dependencies required for development:

```
pip3 install maestral[dev]
```

This will install packages to check and enforce the code style, use pre-commit hooks and bump the current version.

Code is formatted with black. Coding style is checked with flake8. Type hints, PEP484, are checked with mypy.

You can check the format, coding style, and type hints at the same time by running the provided pre-commit hook from the git directory:

```
pre-commit run -a
```

You can also install the provided pre-commit hook to run checks on every commit. This will however significantly slow down commits. An introduction to pre-commit commit hooks is given at https://pre-commit.com.

5.2 Documentation

The documentation is built using sphinx and a few of its extensions. It is built from the develop and master branches and hosted on Read The Docs. If you want to build the documentation locally, install maestral with the `docs` extra to get all required dependencies:

```
pip3 install maestral[docs]
```

The API documentation is mostly based on doc strings. Inline comments should be used whenever code may be difficult to understand for others.

5.3 Tests

The test suite uses a mixture of unittest and pytest, depending on what is most convenient for the actual test and the preference of the author. Pytest should be used as the test runner.

Test are grouped into those which require a linked Dropbox account ("linked") and those who can run by themselves ("offline"). The former tend to be integration test while the latter are mostly unit tests. The current focus currently lies on integration tests, especially for the sync engine, as they are easier to maintain when the implementation and internal APIs change. Exceptions are made for performance tests, for instance for indexing and cleaning up sync events, and for particularly complex functions that are prone to regressions.

The current test suite uses a Dropbox access token provided by the environment variable `DROPBOX_ACCESS_TOKEN` or a refresh token provided by `DROPBOX_REFRESH_TOKEN` to connect to a real account. The GitHub action which is running the tests will set the `DROPBOX_ACCESS_TOKEN` environment variable for you with a temporary access token that expires after 4 hours. Tests are run on `ubuntu-latest` and `macos-latest` in parallel on different accounts.

When using the GitHub test runner, you should acquire a "lock" on the account before running tests to prevent them from interfering which each other by creating a folder `test.lock` in the root of the Dropbox folder. This folder should have a `client_modified` time set in the future, to the expiry time of the lock. Fixtures to create and clean up a test config and to acquire a lock are provided in the `tests/linked/conftest.py`.

If you run the tests locally, you will need to provide a refresh or access token for your own Dropbox account. If your account is already linked with Maestral, it will have saved a long-lived "refresh token" in your system keyring. You can access it manually or through the Python API:

```
from maestral.main import Maestral

m = Maestral()
print(m.client.auth.refresh_token)
```

You can then store the retrieved refresh token in the environment variable `DROPBOX_REFRESH_TOKEN` to be automatically picked up by the tests.

CHAPTER

SIX

MAESTRAL.AUTOSTART

CHAPTER

SEVEN

MAESTRAL.CLIENT

CHAPTER

EIGHT

MAESTRAL.CONFIG

8.1 Submodules

8.1.1 maestral.config.base

8.1.2 maestral.config.main

8.1.3 maestral.config.user

8.2 Module contents

CHAPTER

NINE

MAESTRAL.CONSTANTS

CHAPTER

TEN

MAESTRAL.DAEMON

CHAPTER

ELEVEN

MAESTRAL.DATABASE

CHAPTER

TWELVE

MAESTRAL.ERRORS

CHAPTER

THIRTEEN

MAESTRAL.FSEVENTS

13.1 Submodules

13.1.1 maestral.fsevents.polling

13.2 Module contents

CHAPTER

FOURTEEN

MAESTRAL.LOGGING

CHAPTER

FIFTEEN

MAESTRAL.MAIN

CHAPTER

SIXTEEN

MAESTRAL.MANAGER

CHAPTER

SEVENTEEN

MAESTRAL.NOTIFY

CHAPTER

EIGHTEEN

MAESTRAL.OAUTH

CHAPTER

NINETEEN

MAESTRAL.SYNC

CHAPTER

TWENTY

MAESTRAL.UTILS

20.1 Submodules

20.1.1 maestral.utils.appdirs

20.1.2 maestral.utils.caches

20.1.3 maestral.utils.cli

20.1.4 maestral.utils.content_hasher

20.1.5 maestral.utils.integration

20.1.6 maestral.utils.orm

20.1.7 maestral.utils.path

20.1.8 maestral.utils.serializer

20.2 Module contents

CHAPTER

TWENTYONE

GETTING STARTED

To use the Maestral API in a Python interpreter, import the main module first and initialize a Maestral instance with a configuration name. For this example, we use a new configuration "private" which is not yet linked to a Dropbox account:

```
>>> from maestral.main import Maestral
>>> m = Maestral(config_name="private")
```

Config files will be created on-demand for the new configuration, as described in *Config files* and *State files*.

We now link the instance to an existing Dropbox account. This is done by generating a Dropbox URL for the user to visit and authorize Maestral. Using the `link()` method, the resulting auth code is exchanged for an access token to make Dropbox API calls. See Dropbox's oauth-guide for details on the OAuth2 PKCE flow which we use. When the auth flow is successfully completed, the credentials will be saved in the system keyring (e.g., macOS Keychain or Gnome Keyring).

```
>>> url = m.get_auth_url()  # get token from Dropbox website
>>> print(f"Please go to {url} to retrieve a Dropbox authorization token.")
>>> token = input("Enter auth token: ")
>>> res = m.link(token)
```

The call to `link()` will return 0 on success, 1 for an invalid code and 2 for connection errors. We verify that linking succeeded and proceed to create a local Dropbox folder and start syncing:

```
>>> if res == 0:
...     m.create_dropbox_directory("~/Dropbox (Private)")
...     m.start_sync()
```