Maestral API Documentation

Release 1.5.1

Sam Schott

Oct 21, 2021

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

This module handles starting the maestral daemon on user login and supports multiple platform specific backends such as launchd or systemd.

Note that launchd agents will not show as "login items" in macOS system preferences. As a result, the user does not have a convenient UI to remove Maestral autostart entries manually outside of Maestral itself. Login items however only support app bundles and provide no option to pass command line arguments to the app. They would therefore neither support pip installed packages or multiple configurations.

class maestral.autostart.**SupportedImplementations**(*value*)

Bases: enum.Enum

Enumeration of supported implementations

systemd = 'systemd'

launchd = 'launchd'

xdg_desktop = 'xdg_desktop'

class maestral.autostart.**AutoStartBase**

Bases: object

Base class for autostart backends

enable()

Enable autostart. Must be implemented in subclass.

Return type None

disable()

Disable autostart. Must be implemented in subclass.

Return type None

property enabled: bool

Returns the enabled status as bool. Must be implemented in subclass.

class maestral.autostart.**AutoStartSystemd**(*service_name*, *start_cmd*, *unit_dict=None*, *service_dict=None*, *install_dict=None*)

Bases: *maestral.autostart.AutoStartBase*

Autostart backend for systemd

Parameters

- **service_name** (*str*) – Name of systemd service.
- **start_cmd** (*str*) – Absolute path to executable and optional program arguments.

- **unit_dict** (*Optional[Dict[str, str]]*) – Dictionary of additional keys and values for the Unit section.
- **service_dict** (*Optional[Dict[str, str]]*) – Dictionary of additional keys and values for the Service section.
- **install_dict** (*Optional[Dict[str, str]]*) – Dictionary of additional keys and values for the Install section.

Return type None

enable()

Return type None

disable()

Return type None

property enabled: bool
Checks if the systemd service is enabled.

class maestral.autostart.**AutoStartLaunchd**(*bundle_id*, *start_cmd*, ***kwargs*)
Bases: *maestral.autostart.AutoStartBase*

Autostart backend for launchd

Parameters

- **bundle_id** (*str*) – Bundle ID for the, e.g., "com.google.calendar".
- **start_cmd** (*str*) – Absolute path to executable and optional program arguments.
- **kwargs** – Additional key, value pairs to add to plist. Values may be strings, booleans, lists or dictionaries.

Return type None

enable()

Return type None

disable()

Return type None

property enabled: bool
Checks if the launchd plist exists in ~/Library/LaunchAgents.

class maestral.autostart.**AutoStartXDGDesktop**(*app_name*, *start_cmd*, *filename*, ***kwargs*)
Bases: *maestral.autostart.AutoStartBase*

Autostart backend for XDG desktop entries

Used to start a GUI on user login for most Linux desktops. For a full specifications, please see:

https://specifications.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html

Parameters

- **app_name** (*str*) – Name of application.

- **start_cmd** (*str*) – Executable on $PATH or absolute path to executable and optional program arguments.
- **filename** (*Optional[str]*) – Name of desktop entry file. If not given, the application name will be used.
- **kwargs** (*str*) – Additional key, value pairs to be used in the desktop entries. Values must be strings and may not contain "=", otherwise no additional validation will be performed.

Return type None

enable()

Return type None

disable()

Return type None

property enabled: bool
Checks if the XDG desktop entry exists in ~/.config/autostart.

maestral.autostart.**get_available_implementation()**
Returns the supported implementation depending on the platform.

Return type Optional[*maestral.autostart.SupportedImplementations*]

maestral.autostart.**get_maestral_command_path()**
Returns the path to the maestral executable. May be an empty string if the executable cannot be found.

Return type str

class maestral.autostart.**AutoStart**(*config_name*)
Bases: object

Starts Maestral on user log-in

Creates auto-start files in the appropriate system location to automatically start Maestral when the user logs in. Different backends are used depending on the platform.

Parameters **config_name** (*str*) – Name of Maestral config.

Return type None

property enabled: bool
True if autostart is enabled, False otherwise.

toggle()
Toggles autostart on or off.

Return type None

enable()
Enable autostart.

Return type None

disable()
Disable autostart.

Return type None

CHAPTER

SEVEN

MAESTRAL.CLIENT

This modules contains the Dropbox API client. It wraps calls to the Dropbox Python SDK and handles exceptions, chunked uploads or downloads, etc.

class maestral.client.**DropboxClient**(*config_name*, *timeout=100*, *session=None*)

Bases: object

Client for the Dropbox SDK

This client defines basic methods to wrap Dropbox Python SDK calls, such as creating, moving, modifying and deleting files and folders on Dropbox and downloading files from Dropbox.

All Dropbox SDK exceptions, OSErrors from the local file system API and connection errors will be caught and reraised as a subclass of *maestral.errors.MaestralApiError*.

This class can be used as a context manager to clean up any network resources from the API requests.

Example

```
>>> from maestral.client import DropboxClient
>>> with DropboxClient("maestral") as client:
...     res = client.list_folder("/")
>>> print(res.entries)
```

Parameters

- **config_name** (*str*) – Name of config file and state file to use.
- **timeout** (*float*) – Timeout for individual requests. Defaults to 100 sec if not given.
- **session** (*Optional[requests.sessions.Session]*) – Optional requests session to use. If not given, a new session will be created with dropbox.dropbox_client.create_session().

Return type None

SDK_VERSION: str = '2.0'

property dbx_base: dropbox.dropbox_client.Dropbox

The underlying Dropbox SDK instance without namespace headers.

property dbx: dropbox.dropbox_client.Dropbox

The underlying Dropbox SDK instance with namespace headers.

property linked: bool

Indicates if the client is linked to a Dropbox account (read only). This will block until the user's keyring is unlocked to load the saved auth token.

Raises *KeyringAccessError* – if keyring access fails.

get_auth_url()
Returns a URL to authorize access to a Dropbox account. To link a Dropbox account, retrieve an auth token from the URL and link Maestral by calling *link()* with the provided token.

Returns URL to retrieve an OAuth token.

Return type str

link(*token*)
Links Maestral with a Dropbox account using the given access token. The token will be stored for future usage as documented in the `oauth` module.

Parameters **token** (*str*) – OAuth token for Dropbox access.

Returns 0 on success, 1 for an invalid token and 2 for connection errors.

Return type int

unlink()
Unlinks the Dropbox account.

Raises

- ***KeyringAccessError*** – if keyring access fails.
- ***DropboxAuthError*** – if we cannot authenticate with Dropbox.

Return type None

property account_info: **dropbox.users.FullAccount**
Returns cached account info. Use *get_account_info()* to get the latest account info from Dropbox servers.

property namespace_id: **str**
The namespace ID of the path root currently used by the DropboxClient. All file paths will be interpreted as relative to the root namespace. Use *update_path_root()* to update the root namespace after the user joins or leaves a team with a Team Space.

property is_team_space: **bool**
Whether the user's Dropbox uses a Team Space. Use *update_path_root()* to update the root namespace after the user joins or eaves a team with a Team Space.

close()
Cleans up all resources like the request session/network connection.

Return type None

clone(*config_name=None*, *timeout=None*, *session=None*)
Creates a new copy of the Dropbox client with the same defaults unless modified by arguments to *clone()*.

Parameters

- **config_name** (*Optional[str]*) – Name of config file and state file to use.
- **timeout** (*Optional[float]*) – Timeout for individual requests.
- **session** (*Optional[requests.sessions.Session]*) – Requests session to use.

Returns A new instance of DropboxClient.

Return type *maestral.client.DropboxClient*

clone_with_new_session()
Creates a new copy of the Dropbox client with the same defaults but a new requests session.

Returns A new instance of DropboxClient.

Return type *maestral.client.DropboxClient*

update_path_root(*root_info=None*)

Updates the root path for the Dropbox client. All files paths given as arguments to API calls such as *list_folder()* or *get_metadata()* will be interpreted as relative to the root path. All file paths returned by API calls, for instance in file metadata, will be relative to this root path.

The root namespace will change when the user joins or leaves a Dropbox Team with Team Spaces. If this happens, API calls using the old root namespace will raise a `PathRootError`. Use this method to update to the new root namespace.

See https://developers.dropbox.com/dbx-team-files-guide and https://www.dropbox.com/developers/reference/path-root-header-modes for more information on Dropbox Team namespaces and path root headers in API calls.

Note: We don't automatically switch root namespaces because API users may want to take action when the path root has changed before making further API calls. Be prepared to handle :class:`PathRootError`s and act accordingly.

Parameters **root_info** (*Optional[dropbox.common.RootInfo]*) – Optional dropbox.common.RootInfo describing the path root. If not given, the latest root info will be fetched from Dropbox servers.

Return type None

get_account_info(*dbid=None*)

Gets current account information.

Parameters **dbid** (*Optional[str]*) – Dropbox ID of account. If not given, will get the info of the currently linked account.

Returns Account info.

Return type dropbox.users.FullAccount

get_space_usage()

Returns The space usage of the currently linked account.

Return type dropbox.users.SpaceUsage

get_metadata(*dbx_path*, ***kwargs*)

Gets metadata for an item on Dropbox or returns `False` if no metadata is available. Keyword arguments are passed on to Dropbox SDK files_get_metadata call.

Parameters

- **dbx_path** (*str*) – Path of folder on Dropbox.
- **kwargs** – Keyword arguments for Dropbox SDK files_get_metadata.

Returns Metadata of item at the given path or `None` if item cannot be found.

Return type Optional[dropbox.files.Metadata]

list_revisions(*dbx_path*, *mode='path'*, *limit=10*)

Lists all file revisions for the given file.

Parameters

- **dbx_path** (*str*) – Path to file on Dropbox.

- **mode** (*str*) – Must be 'path' or 'id'. If 'id', specify the Dropbox file ID instead of the file path to get revisions across move and rename events.
- **limit** (*int*) – Maximum number of revisions to list.

Returns File revision history.

Return type dropbox.files.ListRevisionsResult

restore(*dbx_path*, *rev*)
Restore an old revision of a file.

Parameters

- **dbx_path** (*str*) – The path to save the restored file.
- **rev** (*str*) – The revision to restore. Old revisions can be listed with *list_revisions()*.

Returns Metadata of restored file.

Return type dropbox.files.FileMetadata

download(*dbx_path*, *local_path*, *sync_event=None*, ***kwargs*)
Downloads a file from Dropbox to given local path.

Parameters

- **dbx_path** (*str*) – Path to file on Dropbox or rev number.
- **local_path** (*str*) – Path to local download destination.
- **sync_event** (*Optional[SyncEvent]*) – If given, the sync event will be updated with the number of downloaded bytes.
- **kwargs** – Keyword arguments for the Dropbox API files_download endpoint.

Returns Metadata of downloaded item.

Return type dropbox.files.FileMetadata

upload(*local_path*, *dbx_path*, *chunk_size=5000000*, *sync_event=None*, ***kwargs*)
Uploads local file to Dropbox.

Parameters

- **local_path** (*str*) – Path of local file to upload.
- **dbx_path** (*str*) – Path to save file on Dropbox.
- **kwargs** – Keyword arguments for Dropbox SDK files_upload.
- **chunk_size** (*int*) – Maximum size for individual uploads. If larger than 150 MB, it will be set to 150 MB.
- **sync_event** (*Optional[SyncEvent]*) – If given, the sync event will be updated with the number of downloaded bytes.

Returns Metadata of uploaded file.

Return type dropbox.files.FileMetadata

remove(*dbx_path*, ***kwargs*)
Removes a file / folder from Dropbox.

Parameters

- **dbx_path** (*str*) – Path to file on Dropbox.
- **kwargs** – Keyword arguments for the Dropbox API files_delete_v2 endpoint.

Returns Metadata of deleted item.

Return type dropbox.files.Metadata

remove_batch(*entries*, *batch_size=900*)
Deletes multiple items on Dropbox in a batch job.

Parameters

- **entries** (*List[Tuple[str, str]]*) – List of Dropbox paths and "rev"s to delete. If a "rev" is not None, the file will only be deleted if it matches the rev on Dropbox. This is not supported when deleting a folder.
- **batch_size** (*int*) – Number of items to delete in each batch. Dropbox allows batches of up to 1,000 items. Larger values will be capped automatically.

Returns List of Metadata for deleted items or SyncErrors for failures. Results will be in the same order as the original input.

Return type List[Union[dropbox.files.Metadata, *maestral.errors.MaestralApiError*]]

move(*dbx_path*, *new_path*, ***kwargs*)
Moves / renames files or folders on Dropbox.

Parameters

- **dbx_path** (*str*) – Path to file/folder on Dropbox.
- **new_path** (*str*) – New path on Dropbox to move to.
- **kwargs** – Keyword arguments for the Dropbox API files_move_v2 endpoint.

Returns Metadata of moved item.

Return type dropbox.files.Metadata

make_dir(*dbx_path*, ***kwargs*)
Creates a folder on Dropbox.

Parameters

- **dbx_path** (*str*) – Path of Dropbox folder.
- **kwargs** – Keyword arguments for the Dropbox API files_create_folder_v2 endpoint.

Returns Metadata of created folder.

Return type dropbox.files.FolderMetadata

make_dir_batch(*dbx_paths*, *batch_size=900*, ***kwargs*)
Creates multiple folders on Dropbox in a batch job.

Parameters

- **dbx_paths** (*List[str]*) – List of dropbox folder paths.
- **batch_size** (*int*) – Number of folders to create in each batch. Dropbox allows batches of up to 1,000 folders. Larger values will be capped automatically.
- **kwargs** – Keyword arguments for the Dropbox API files/create_folder_batch endpoint.

Returns List of Metadata for created folders or SyncError for failures. Entries will be in the same order as given paths.

Return type List[Union[dropbox.files.Metadata, *maestral.errors.MaestralApiError*]]

share_dir(*dbx_path*, ***kwargs*)
Converts a Dropbox folder to a shared folder. Creates the folder if it does not exist.

Parameters

- **dbx_path** (*str*) – Path of Dropbox folder.
- **kwargs** – Keyword arguments for the Dropbox API files_create_folder_v2 endpoint.

Returns Metadata of shared folder.

Return type dropbox.sharing.SharedFolderMetadata

get_latest_cursor(*dbx_path*, *include_non_downloadable_files=False*, ***kwargs*)

Gets the latest cursor for the given folder and subfolders.

Parameters

- **dbx_path** (*str*) – Path of folder on Dropbox.
- **include_non_downloadable_files** (*bool*) – If `True`, files that cannot be downloaded (at the moment only G-suite files on Dropbox) will be included.
- **kwargs** – Additional keyword arguments for Dropbox API files/list_folder/get_latest_cursor endpoint.

Returns The latest cursor representing a state of a folder and its subfolders.

Return type str

list_folder(*dbx_path*, *max_retries_on_timeout=4*, *include_non_downloadable_files=False*, ***kwargs*)

Lists the contents of a folder on Dropbox. Similar to *list_folder_iterator()* but returns all entries in a single `dropbox.files.ListFolderResult` instance.

Parameters

- **dbx_path** (*str*) – Path of folder on Dropbox.
- **max_retries_on_timeout** (*int*) – Number of times to try again if Dropbox servers do not respond within the timeout. Occasional timeouts may occur for very large Dropbox folders.
- **include_non_downloadable_files** (*bool*) – If `True`, files that cannot be downloaded (at the moment only G-suite files on Dropbox) will be included.
- **kwargs** – Additional keyword arguments for Dropbox API files/list_folder endpoint.

Returns Content of given folder.

Return type dropbox.files.ListFolderResult

list_folder_iterator(*dbx_path*, *max_retries_on_timeout=4*, *include_non_downloadable_files=False*, ***kwargs*)

Lists the contents of a folder on Dropbox. Returns an iterator yielding `dropbox.files.ListFolderResult` instances. The number of entries returned in each iteration corresponds to the number of entries returned by a single Dropbox API call and will be typically around 500.

Parameters

- **dbx_path** (*str*) – Path of folder on Dropbox.
- **max_retries_on_timeout** (*int*) – Number of times to try again if Dropbox servers do not respond within the timeout. Occasional timeouts may occur for very large Dropbox folders.
- **include_non_downloadable_files** (*bool*) – If `True`, files that cannot be downloaded (at the moment only G-suite files on Dropbox) will be included.
- **kwargs** – Additional keyword arguments for the Dropbox API files/list_folder endpoint.

Returns Iterator over content of given folder.

Return type Iterator[dropbox.files.ListFolderResult]

`wait_for_remote_changes`(*last_cursor*, *timeout=40*)

Waits for remote changes since `last_cursor`. Call this method after starting the Dropbox client and periodically to get the latest updates.

Parameters

- **last_cursor** (*str*) – Last to cursor to compare for changes.
- **timeout** (*int*) – Seconds to wait until timeout. Must be between 30 and 480. The Dropbox API will add a random jitter of up to 60 sec to this value.

Returns `True` if changes are available, `False` otherwise.

Return type bool

`list_remote_changes`(*last_cursor*)

Lists changes to remote Dropbox since `last_cursor`. Same as *list_remote_changes_iterator()* but fetches all changes first and returns a single `dropbox.files.ListFolderResult`. This may be useful if you want to fetch all changes in advance before starting to process them.

Parameters **last_cursor** (*str*) – Last to cursor to compare for changes.

Returns Remote changes since given cursor.

Return type dropbox.files.ListFolderResult

`list_remote_changes_iterator`(*last_cursor*)

Lists changes to the remote Dropbox since `last_cursor`. Returns an iterator yielding `dropbox.files.ListFolderResult` instances. The number of entries returned in each iteration corresponds to the number of entries returned by a single Dropbox API call and will be typically around 500.

Call this after *wait_for_remote_changes()* returns `True`.

Parameters **last_cursor** (*str*) – Last to cursor to compare for changes.

Returns Iterator over remote changes since given cursor.

Return type Iterator[dropbox.files.ListFolderResult]

`create_shared_link`(*dbx_path*, *visibility=RequestedVisibility('public', None)*, *password=None*, *expires=None*, ***kwargs*)

Creates a shared link for the given path. Some options are only available for Professional and Business accounts. Note that the requested visibility as access level for the link may not be granted, depending on the Dropbox folder or team settings. Check the returned link metadata to verify the visibility and access level.

Parameters

- **dbx_path** (*str*) – Dropbox path to file or folder to share.
- **visibility** (*dropbox.sharing.RequestedVisibility*) – The visibility of the shared link. Can be public, team-only, or password protected. In case of the latter, the password argument must be given. Only available for Professional and Business accounts.
- **password** (*Optional[str]*) – Password to protect shared link. Is required if visibility is set to password protected and will be ignored otherwise
- **expires** (*Optional[datetime.datetime]*) – Expiry time for shared link. Only available for Professional and Business accounts.

- **kwargs** – Additional keyword arguments to create the dropbox.sharing.SharedLinkSettings instance.

Returns Metadata for shared link.

Return type dropbox.sharing.SharedLinkMetadata

revoke_shared_link(*url*)
Revokes a shared link.

Parameters **url** (*str*) – URL to revoke.

Return type None

list_shared_links(*dbx_path=None*)
Lists all shared links for a given Dropbox path (file or folder). If no path is given, list all shared links for the account, up to a maximum of 1,000 links.

Parameters **dbx_path** (*Optional[str]*) – Dropbox path to file or folder.

Returns Shared links for a path, including any shared links for parents through which this path is accessible.

Return type dropbox.sharing.ListSharedLinksResult

static flatten_results(*results*, *attribute_name*)
Flattens a list of Dropbox API results from a pagination to a single result with the cursor of the last result in the list.

Parameters

- **results** (*List[Union[dropbox.sharing.ListSharedLinksResult, dropbox.files.ListFolderResult]]*) – List of :results to flatten.
- **attribute_name** (*str*) – Name of attribute to flatten.

Returns Flattened result.

Return type Union[dropbox.sharing.ListSharedLinksResult, dropbox.files.ListFolderResult]

maestral.client.**dropbox_to_maestral_error**(*exc*, *dbx_path=None*, *local_path=None*)
Converts a Dropbox SDK exception to a *maestral.errors.MaestralApiError* and tries to add a reasonably informative error title and message.

Parameters

- **exc** (*Union[dropbox.exceptions.DropboxException, stone.backends.python_rsrc.stone_validators.ValidationError, requests.exceptions.HTTPError]*) – Dropbox SDK exception..
- **dbx_path** (*Optional[str]*) – Dropbox path associated with the error.
- **local_path** (*Optional[str]*) – Local path associated with the error.

Returns Converted exception.

Return type *maestral.errors.MaestralApiError*

maestral.client.**os_to_maestral_error**(*exc*, *dbx_path=None*, *local_path=None*)
Converts a OSError to a *maestral.errors.MaestralApiError* and tries to add a reasonably informative error title and message.

Parameters

- **exc** (*OSError*) – Original OSError.
- **dbx_path** (*Optional[str]*) – Dropbox path associated with the error.

- **local_path** (*Optional[str]*) – Local path associated with the error.

Returns Converted exception.

Return type Union[*maestral.errors.MaestralApiError*, OSError]

maestral.client.**convert_api_errors**(*dbx_path=None*, *local_path=None*)

A context manager that catches and re-raises instances of `OSError` and `dropbox.exceptions.DropboxException` as *`maestral.errors.MaestralApiError`* or `ConnectionError`.

Parameters

- **dbx_path** (*Optional[str]*) – Dropbox path associated with the error.
- **local_path** (*Optional[str]*) – Local path associated with the error.

Return type Iterator[None]

CHAPTER

EIGHT

MAESTRAL.CONFIG

8.1 Submodules

8.1.1 maestral.config.base

8.1.2 maestral.config.main

This module contains the default configuration and state values and functions to return existing config or state instances for a specified config_name.

maestral.config.main.**MaestralConfig**(*config_name*)
Returns an existing config instance or creates a new one.

Parameters **config_name** (*str*) – Name of maestral configuration to run. A new config file will be created if none exists for the given config_name.

Returns Maestral config instance which saves any changes to the drive.

Return type *maestral.config.user.UserConfig*

maestral.config.main.**MaestralState**(*config_name*)
Returns an existing state instance or creates a new one.

Parameters **config_name** (*str*) – Name of maestral configuration to run. A new state file will be created if none exists for the given config_name.

Returns Maestral state instance which saves any changes to the drive.

Return type *maestral.config.user.UserConfig*

8.1.3 maestral.config.user

This module provides user configuration file management and is mostly copied from the config module of the Spyder IDE.

class maestral.config.user.**NoDefault**
Bases: object

class maestral.config.user.**DefaultsConfig**(*path*)
Bases: configparser.ConfigParser

Class used to save defaults to a file and as base class for UserConfig.

Parameters **path** (*str*) –

Return type None

save()
Save config into the associated file.

Return type None

property config_path: str
The ini file where this configuration is stored.

class maestral.config.user.**UserConfig**(*path*, *defaults=None*, *load=True*, *version=<Version('0.0.0')>*, *backup=False*, *remove_obsolete=False*)
Bases: *maestral.config.user.DefaultsConfig*

UserConfig class, based on ConfigParser. This class is save to use from different threads but must not be used from different processes!

Parameters

- **path** (*str*) – Configuration file will be saved to this path.
- **defaults** (*Optional[Dict[str, Dict[str, Any]]]*) – Dictionary containing options.
- **version** (*packaging.version.Version*) – Version of the configuration file.
- **backup** (*bool*) – Whether to create a backup on version changes and on initial setup.
- **remove_obsolete** (*bool*) – If *True*, values that were removed from the configuration on version change, are removed from the saved configuration file.
- **load** (*bool*) –

Return type None

Note: The `get` and `set` arguments number and type differ from the reimplemented methods.

DEFAULT_SECTION_NAME = 'main'

remove_deprecated_options()
Remove options which are present in the file but not in defaults.

Return type None

backup_path_for_version(*version*)
Get backup location based on version.

Parameters **version** (*Optional[packaging.version.Version]*) – The version of the backup, if any.

Returns The back for the backup file.

Return type str

apply_configuration_patches(*old_version*)
Apply any patch to configuration values on version changes.

To be reimplemented if patches to configuration values are needed.

Parameters **old_version** (*packaging.version.Version*) – Old config version to patch.

Return type None

get_version()
Get the current config version.

Returns Configuration (not application!) version.

Return type packaging.version.Version

`set_version`(*version*, *save=True*)
Set configuration (not application!) version.

Parameters

- **version** (*packaging.version.Version*) – New version to set.
- **save** (*bool*) – Whether to save changes to drive.

Return type None

`reset_to_defaults`(*section=None*, *save=True*)
Reset config to default values.

Parameters

- **section** (*Optional[str]*) – The section to reset. If not given, reset all sections.
- **save** (*bool*) – Whether to save the changes to the drive.

Return type None

`get_default`(*section*, *option*)
Get default value for a given `section` and `option`.

This is useful for type checking in *get* method.

Parameters

- **section** (*str*) – Section to search for option.
- **option** (*str*) – Config option.

Returns Config value or None if section / option do not exist.

Return type Any

`get`(*section*, *option*, *default=<class 'maestral.config.user.NoDefault'>*)
Get an option.

Parameters

- **section** (*str*) – Config section to search in.
- **option** (*str*) – Config option to get.
- **default** (*Any*) – Default value to fall back to if not present.

Returns Config value.

Raises

- **cp.NoSectionError** – if the section does not exist.
- **cp.NoOptionError** – if the option does not exist and no default is given.

Return type Any

`set_default`(*section*, *option*, *default_value*)
Set Default value for a given *section*, *option*.

If the section or option does not exist, it will be created.

Parameters

- **section** (*str*) –
- **option** (*str*) –

- **default_value** (*Any*) –

Return type None

set(*section*, *option*, *value*, *save=True*)
Set an option` on a given ``section.

If section is None, the option is added to the default section.

Parameters

- **section** (*str*) – Config section to search in.
- **option** (*str*) – Config option to set.
- **value** (*Any*) – Config value.
- **save** (*bool*) – Whether to save the changes to the drive.

Return type None

remove_section(*section*, *save=True*)
Remove section and all options within it.

Parameters

- **section** (*str*) – Section to remove from the config file.
- **save** (*bool*) – Whether to save the changes to the drive.

Returns Whether the section was removed successfully.

Return type bool

remove_option(*section*, *option*, *save=True*)
Remove option from section.

Parameters

- **section** (*str*) – Section to look for the option.
- **option** (*str*) – Option to remove from the config file.
- **save** (*bool*) – Whether to save the changes to the drive.

Returns Whether the section was removed successfully.

Return type bool

cleanup()
Remove files associated with config and reset to defaults.

Return type None

8.2 Module contents

maestral.config.**MaestralConfig**(*config_name*)
Returns an existing config instance or creates a new one.

Parameters **config_name** (*str*) – Name of maestral configuration to run. A new config file will be created if none exists for the given config_name.

Returns Maestral config instance which saves any changes to the drive.

Return type *maestral.config.user.UserConfig*

maestral.config.**MaestralState**(*config_name*)
Returns an existing state instance or creates a new one.

Parameters **config_name** (*str*) – Name of maestral configuration to run. A new state file will be created if none exists for the given config_name.

Returns Maestral state instance which saves any changes to the drive.

Return type *maestral.config.user.UserConfig*

maestral.config.**list_configs**()
Lists all maestral configs.

Returns A list of all currently existing config files.

Return type List[str]

maestral.config.**remove_configuration**(*config_name*)
Removes all config and state files associated with the given configuration.

Parameters **config_name** (*str*) – The configuration to remove.

Return type None

maestral.config.**validate_config_name**(*string*)
Validates that the config name does not contain any whitespace.

Parameters **string** (*maestral.config._C*) – String to validate.

Returns The input value.

Raises **ValueError** – if the config name contains whitespace.

Return type maestral.config._C

CHAPTER

NINE

MAESTRAL.CONSTANTS

This module provides constants used throughout the maestral, the GUI and CLI. It should be kept free of memory heavy imports.

class maestral.constants.**FileStatus**(*value*)

Bases: enum.Enum

Enumeration of sync status

Unwatched = 'unwatched'

Uploading = 'uploading'

Downloading = 'downloading'

Error = 'error'

Synced = 'up to date'

CHAPTER

TEN

MAESTRAL.DAEMON

This module defines functions to start and stop the sync daemon and retrieve proxy objects for a running daemon.

class maestral.daemon.**Stop**(*value*)
Bases: enum.Enum

Enumeration of daemon exit results

Ok = 0

Killed = 1

NotRunning = 2

Failed = 3

class maestral.daemon.**Start**(*value*)
Bases: enum.Enum

Enumeration of daemon start results

Ok = 0

AlreadyRunning = 1

Failed = 2

class maestral.daemon.**Lock**(*path*)
Bases: object

A inter-process and inter-thread lock

This internally uses `fasteners.InterProcessLock` but provides non-blocking acquire. It also guarantees thread-safety when using the *singleton()* class method to create / retrieve a lock instance.

Parameters **path** (*str*) – Path of the lock file to use / create.

Return type None

classmethod singleton(*path*)
Retrieve an existing lock object with a given 'name' or create a new one. Use this method for thread-safe locks.

Parameters **path** (*str*) – Path of the lock file to use / create.

Return type *maestral.daemon.Lock*

acquire()
Attempts to acquire the given lock.

Returns Whether or not the acquisition succeeded.

Return type bool

release()
Release the previously acquired lock.

Return type None

locked()
Checks if the lock is currently held by any thread or process.

Returns Whether the lock is acquired.

Return type bool

locking_pid()
Returns the PID of the process which currently holds the lock or None. This should work on macOS, OpenBSD and Linux but may fail on some platforms. Always use *locked()* to check if the lock is held by any process.

Returns The PID of the process which currently holds the lock or None.

Return type Optional[int]

maestral.daemon.**maestral_lock**(*config_name*)
Returns an inter-process and inter-thread lock for Maestral. This is a wrapper around *Lock* which fills out the appropriate lockfile path for he given config name.

Parameters **config_name** (*str*) – The name of the Maestral configuration.

Returns Lock instance for the config name

Return type *maestral.daemon.Lock*

maestral.daemon.**get_maestral_pid**(*config_name*)
Returns the PID of the daemon if it is running, None otherwise.

Parameters **config_name** (*str*) – The name of the Maestral configuration.

Returns The daemon's PID.

Return type Optional[int]

maestral.daemon.**sockpath_for_config**(*config_name*)
Returns the unix socket location to be used for the config. This should default to the apps runtime directory + 'CONFIG_NAME.sock'.

Parameters **config_name** (*str*) – The name of the Maestral configuration.

Returns Socket path.

Return type str

maestral.daemon.**lockpath_for_config**(*config_name*)
Returns the lock file location to be used for the config. This will be the apps runtime directory + 'CONFIG_NAME.lock'.

Parameters **config_name** (*str*) – The name of the Maestral configuration.

Returns Path of lock file to use.

Return type str

maestral.daemon.**wait_for_startup**(*config_name*, *timeout=20*)
Waits until we can communicate with the maestral daemon for config_name.

Parameters

- **config_name** (*str*) – Configuration to connect to.

- **timeout** (*float*) – Timeout it seconds until we raise an error.

Raises *CommunicationError* – if we cannot communicate with the daemon within the given timeout.

Return type None

maestral.daemon.**is_running**(*config_name*)
Checks if a daemon is currently running.

Parameters **config_name** (*str*) – The name of the Maestral configuration.

Returns Whether the daemon is running.

Return type bool

maestral.daemon.**freeze_support**()
Call this as early as possible in the main entry point of a frozen executable. This call will start the sync daemon if a matching command line arguments are detected and do nothing otherwise.

Return type None

maestral.daemon.**start_maestral_daemon**(*config_name='maestral'*, *log_to_stderr=False*)
Starts the Maestral daemon with event loop in the current thread.

Startup is race free: there will never be more than one daemon running with the same config name. The daemon is a *maestral.main.Maestral* instance which is exposed as Pyro daemon object and listens for requests on a unix domain socket. This call starts an asyncio event loop to process client requests and blocks until the event loop shuts down. On macOS, the event loop is integrated with Cocoa's CFRunLoop. This allows processing Cocoa events and callbacks, for instance for desktop notifications.

Parameters

- **config_name** (*str*) – The name of the Maestral configuration to use.
- **log_to_stderr** (*bool*) – If `True`, write logs to stderr.

Raises `RuntimeError` – if a daemon for the given `config_name` is already running.

Return type None

maestral.daemon.**start_maestral_daemon_process**(*config_name='maestral'*, *timeout=20*)
Starts the Maestral daemon in a new process by calling *start_maestral_daemon()*.

Startup is race free: there will never be more than one daemon running for the same config name. This function will use `sys.executable` as a Python executable to start the daemon.

Environment variables from the current process will be preserved and updated with the environment variables defined in `constants.ENV`.

Parameters

- **config_name** (*str*) – The name of the Maestral configuration to use.
- **timeout** (*float*) – Time in sec to wait for daemon to start.

Returns *Start.Ok* if successful, *Start.AlreadyRunning* if the daemon was already running or *Start.Failed* if startup failed. It is possible that *Start.Ok* may be returned instead of *Start.AlreadyRunning* in case of a race but the daemon is nevertheless started only once.

Return type *maestral.daemon.Start*

maestral.daemon.**stop_maestral_daemon_process**(*config_name='maestral'*, *timeout=20*)
Stops a maestral daemon process by finding its PID and shutting it down.

This function first tries to shut down Maestral gracefully. If this fails and we know its PID, it will send SIGTERM. If that fails as well, it will send SIGKILL to the process.

Parameters

- **config_name** (*str*) – The name of the Maestral configuration to use.
- **timeout** (*float*) – Number of sec to wait for daemon to shut down before killing it.

Returns *Stop.Ok* if successful, *Stop.Killed* if killed, *Stop.NotRunning* if the daemon was not running and *Stop.Failed* if killing the process failed because we could not retrieve its PID.

Return type *maestral.daemon.Stop*

class maestral.daemon.**MaestralProxy**(*config_name='maestral'*, *fallback=False*)

Bases: object

A Proxy to the Maestral daemon

All methods and properties of Maestral's public API are accessible and calls / access will be forwarded to the corresponding Maestral instance. This class can be used as a context manager to close the connection to the daemon on exit.

Example Use MaestralProxy as a context manager:

```
>>> with MaestralProxy() as m:
...     print(m.status)
```

Use MaestralProxy directly:

```
>>> m = MaestralProxy()
>>> print(m.status)
>>> m._disconnect()
```

Variables **_is_fallback** – Whether we are using an actual Maestral instance as fallback instead of a Proxy.

Parameters

- **config_name** (*str*) – The name of the Maestral configuration to use.
- **fallback** (*bool*) – If `True`, a new instance of Maestral will created in the current process when the daemon is not running.

Raises *CommunicationError* – if the daemon is running but cannot be reached or if the daemon is not running and `fallback` is `False`.

Return type None

exception maestral.daemon.**CommunicationError**

Bases: Pyro5.errors.PyroError

Base class for the errors related to network communication problems.

CHAPTER

ELEVEN

MAESTRAL.DATABASE

This module contains the definitions of our data base tables which store the index, sync history and cache of content hashes. Each table is defined by a subclass of *maestral.utils.orm.Model* with properties representing database columns. Class instances then represent table rows.

class maestral.database.**SyncDirection**(*value*)

Bases: enum.Enum

Enumeration of sync directions

Up = 'up'

Down = 'down'

class maestral.database.**SyncStatus**(*value*)

Bases: enum.Enum

Enumeration of sync status

Queued = 'queued'

Syncing = 'syncing'

Done = 'done'

Failed = 'failed'

Skipped = 'skipped'

Aborted = 'aborted'

class maestral.database.**ItemType**(*value*)

Bases: enum.Enum

Enumeration of SyncEvent types

File = 'file'

Folder = 'folder'

Unknown = 'unknown'

class maestral.database.**ChangeType**(*value*)

Bases: enum.Enum

Enumeration of SyncEvent change types

Added = 'added'

Removed = 'removed'

Moved = 'moved'

Modified = 'modified'

class maestral.database.**SyncEvent**(***kwargs*)

Bases: *maestral.utils.orm.Model*

Represents a file or folder change in the sync queue

This class is used to represent both local and remote file system changes and track their sync progress. Some instance attributes will depend on the state of the sync session, e.g., *local_path* will depend on the current path of the local Dropbox folder. They may therefore become invalid between sync sessions.

The class methods *from_dbx_metadata()* and *from_file_system_event()* should be used to properly construct a *SyncEvent* from a `dropbox.files.Metadata` instance or a `watchdog.events.FileSystemEvent` instance, respectively.

Initialise with keyword arguments corresponding to column names and values.

Parameters **kwargs** – Keyword arguments assigning values to table columns.

Return type None

property id: Any
A unique identifier of the SyncEvent.

property direction: Any
The *SyncDirection*.

property item_type: Any
The *ItemType*. May be undetermined for remote deletions.

property sync_time: Any
The time the SyncEvent was registered.

property dbx_id: Any
A unique dropbox ID for the file or folder. Will only be set for download events which are not deletions.

property dbx_path: Any
Upper case Dropbox path of the item to sync. If the sync represents a move operation, this will be the destination path. Follows the casing from the path_display attribute of Dropbox metadata.

property dbx_path_lower: Any
Dropbox path of the item to sync. If the sync represents a move operation, this will be the destination path. This is normalised as the path_lower attribute of Dropbox metadata.

property local_path: Any
Local path of the item to sync. If the sync represents a move operation, this will be the destination path. This will be correctly cased.

property dbx_path_from: Any
Dropbox path that this item was moved from. Will only be set if *change_type* is *ChangeType.Moved*. Follows the casing from the path_display attribute of Dropbox metadata.

property dbx_path_from_lower: Any
Dropbox path that this item was moved from. Will only be set if *change_type* is *ChangeType.Moved*. This is normalised as the path_lower attribute of Dropbox metadata.

property local_path_from: Any
Local path that this item was moved from. Will only be set if *change_type* is *ChangeType.Moved*. This will be correctly cased.

property rev: Any
The file revision. Will only be set for remote changes. Will be `'folder'` for folders and `None` for deletions.

property content_hash: Any
A hash representing the file content. Will be `'folder'` for folders and `None` for deletions. Set for both local and remote changes.

property change_type: Any
The *ChangeType*. Remote SyncEvents currently do not generate moved events but are reported as deleted and added at the new location.

property change_time: Any
Local ctime or remote `client_modified` time for files. `None` for folders or for remote deletions. Note that `client_modified` may not be reliable as it is set by other clients and not verified.

property change_dbid: Any
The Dropbox ID of the account which performed the changes. This may not be set for added folders or deletions on the server.

property change_user_name: Any
The user name corresponding to *change_dbid*, if the account still exists. This field may not be set for performance reasons.

property status: Any
The *SyncStatus*.

property size: Any
Size of the item in bytes. Always zero for folders.

property completed: Any
File size in bytes which has already been uploaded or downloaded. Always zero for folders.

property change_time_or_sync_time: float
Change time when available, otherwise sync time. This can be used for sorting or user information purposes.

property is_file: bool
Returns True for file changes

property is_directory: bool
Returns True for folder changes

property is_added: bool
Returns True for added items

property is_moved: bool
Returns True for moved items

property is_changed: bool
Returns True for changed file contents

property is_deleted: bool
Returns True for deleted items

property is_upload: bool
Returns True for changes to upload

property is_download: bool
Returns True for changes to download

classmethod from_dbx_metadata(*md*, *sync_engine*)
Initializes a SyncEvent from the given Dropbox metadata.

Parameters

- **md** (*dropbox.files.Metadata*) – Dropbox Metadata.

- **sync_engine** (SyncEngine) – SyncEngine instance.

Returns An instance of this class with attributes populated from the given Dropbox Metadata.

Return type *SyncEvent*

classmethod from_file_system_event(*event*, *sync_engine*)
Initializes a SyncEvent from the given local file system event.

Parameters

- **event** (*watchdog.events.FileSystemEvent*) – Local file system event.
- **sync_engine** (SyncEngine) – SyncEngine instance.

Returns An instance of this class with attributes populated from the given SyncEvent.

Return type *SyncEvent*

class maestral.database.**IndexEntry**(***kwargs*)
Bases: *maestral.utils.orm.Model*

Represents an entry in our local sync index

Initialise with keyword arguments corresponding to column names and values.

Parameters **kwargs** – Keyword arguments assigning values to table columns.

Return type None

property dbx_path_lower: Any
Dropbox path of the item in lower case. This acts as a primary key for the SQLites database since there can only be one entry per case-insensitive Dropbox path. Corresponds to the path_lower field of Dropbox metadata.

property dbx_path_cased: Any
Dropbox path of the item, correctly cased. Corresponds to the path_display field of Dropbox metadata.

property dbx_id: Any
The unique dropbox ID for the item.

property item_type: Any
The *ItemType*.

property last_sync: Any
The last time a local change was uploaded. Should be the ctime of the local item.

property rev: Any
The file revision. Will be `'folder'` for folders.

property content_hash: Any
A hash representing the file content. Will be `'folder'` for folders. May be `None` if not yet calculated.

property is_file: bool
Returns True for file changes

property is_directory: bool
Returns True for folder changes

class maestral.database.**HashCacheEntry**(***kwargs*)
Bases: *maestral.utils.orm.Model*

Represents an entry in our cache of content hashes

Initialise with keyword arguments corresponding to column names and values.

Parameters **kwargs** – Keyword arguments assigning values to table columns.

Return type None

property local_path: Any
The local path of the item.

property hash_str: Any
The content hash of the item.

property mtime: Any
The mtime of the item just before the hash was computed. When the current ctime is newer, the hash will need to be recalculated.

CHAPTER

TWELVE

MAESTRAL.ERRORS

This module defines Maestral's error classes. It should be kept free of memory heavy imports.

All errors inherit from *MaestralApiError* which has title and message attributes to display the error to the user. Errors which are related to syncing a specific file or folder inherit from *SyncError*, a subclass of *MaestralApiError*.

exception maestral.errors.**MaestralApiError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: Exception

Base class for Maestral errors

MaestralApiError provides attributes that can be used to generate human-readable error messages and metadata regarding affected file paths (if any).

Errors originating from the Dropbox API or the 'local API' both inherit from MaestralApiError.

Parameters

- **title** (*str*) – A short description of the error type. This can be used in a CLI or GUI to give a short error summary.
- **message** (*str*) – A more verbose description which can include instructions on how to proceed to fix the error.
- **dbx_path** (*Optional[str]*) – Dropbox path of the file that caused the error.
- **dbx_path_dst** (*Optional[str]*) – Dropbox destination path of the file that caused the error. This should be set for instance when error occurs when moving an item.
- **local_path** (*Optional[str]*) – Local path of the file that caused the error.
- **local_path_dst** (*Optional[str]*) – Local destination path of the file that caused the error. This should be set for instance when error occurs when moving an item.

Return type None

exception maestral.errors.**SyncError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Base class for recoverable sync issues.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –

- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**InsufficientPermissionsError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.SyncError*

Raised when accessing a file or folder fails due to insufficient permissions, both locally and on Dropbox servers.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**InsufficientSpaceError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.SyncError*

Raised when the Dropbox account or local drive has insufficient storage space.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**PathError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.SyncError*

Raised when there is an issue with the provided file or folder path such as invalid characters, a too long file name, etc.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –

- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**NotFoundError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.SyncError*

Raised when a file or folder is requested but does not exist.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**ConflictError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.SyncError*

Raised when trying to create a file or folder which already exists.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**FileConflictError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.ConflictError*

Raised when trying to create a file which already exists.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**FolderConflictError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.SyncError*

Raised when trying to create or folder which already exists.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**IsAFolderError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.SyncError*

Raised when a file is required but a folder is provided.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**NotAFolderError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.SyncError*

Raised when a folder is required but a file is provided.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**DropboxServerError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.SyncError*

Raised in case of internal Dropbox errors.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**RestrictedContentError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.SyncError*

Raised when trying to sync restricted content, for instance when adding a file with a DMCA takedown notice to a public folder.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**UnsupportedFileError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.SyncError*

Raised when this file type cannot be downloaded but only exported. This is the case for G-suite files.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**FileSizeError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.SyncError*

Raised when attempting to upload a file larger than 350 GB in an upload session or larger than 150 MB in a single upload. Also raised when attempting to download a file with a size that exceeds file system's limit.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**FileReadError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.SyncError*

Raised when reading a local file failed.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**DropboxConnectionError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when the connection to Dropbox fails

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**CancelledError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when syncing is cancelled by the user.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**NotLinkedError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when no Dropbox account is linked.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**InvalidDbidError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when the given Dropbox ID does not correspond to an existing account.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**KeyringAccessError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when retrieving a saved auth token from the user keyring fails.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**NoDropboxDirError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when the local Dropbox folder cannot be found.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**CacheDirError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when creating the cache directory fails.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**InotifyError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when the local Dropbox folder is too large to monitor with inotify.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**OutOfMemoryError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when there is insufficient memory to complete an operation.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**DatabaseError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when reading or writing to the database fails.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**DropboxAuthError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when authentication fails.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**TokenExpiredError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.DropboxAuthError*

Raised when authentication fails because the user's token has expired.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**TokenRevokedError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.DropboxAuthError*

Raised when authentication fails because the user's token has been revoked.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**CursorResetError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when the cursor used for a longpoll or list-folder request has been invalidated. Dropbox will very rarely invalidate a cursor. If this happens, a new cursor for the respective folder has to be obtained through files_list_folder. This may require re-syncing the entire Dropbox.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**BadInputError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when an API request is made with bad input. This should not happen during syncing but only in case of manual API calls.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**BusyError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when trying to perform an action which is only possible in the idle state and we cannot block or queue the job.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**UnsupportedFileTypeForDiff**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when a diff for an unsupported file type was issued.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**SharedLinkError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when creating a shared link fails.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

exception maestral.errors.**PathRootError**(*title*, *message=''*, *dbx_path=None*, *dbx_path_dst=None*, *local_path=None*, *local_path_dst=None*)

Bases: *maestral.errors.MaestralApiError*

Raised when making an API call with an invalid path root header.

Parameters

- **title** (*str*) –
- **message** (*str*) –
- **dbx_path** (*Optional[str]*) –
- **dbx_path_dst** (*Optional[str]*) –
- **local_path** (*Optional[str]*) –
- **local_path_dst** (*Optional[str]*) –

Return type None

CHAPTER

THIRTEEN

MAESTRAL.FSEVENTS

13.1 Submodules

13.1.1 maestral.fsevents.polling

Looking at the source code for `watchdog.utils.dirsnapshot.DirectorySnapshotDiff`, the event types are categorised as follows:

- Created event: The inode is unique to the new snapshot. The path may be unique to the new snapshot or exist in both. In the second case, there will be a preceding Deleted event or a Moved event with the path as starting point (the old item was deleted or moved away).
- Deleted event: The inode is unique to the old snapshot. The path may be unique to the old snapshot or exist in both. In the second case, there will be a subsequent Created event or a Moved event with the path as end point (something else was created at or moved to the location).
- Moved event: The inode exists in both snapshots but with different paths.
- Modified event: The inode exists in both snapshots and the mtime or file size are different. DirectorySnapshotDiff will always use the inode's path from the old snapshot.

From the above classification, there can be at most two created/deleted/moved events that share the same path in one snapshot diff:

- Deleted(path1) + Created(path1)
- Moved(path1, path2) + Created(path1)
- Deleted(path1) + Moved(path0, path1)

Any Modified event will come before a Moved event or stand alone. Modified events will never be combined by themselves with created or deleted events because they require the inode to be present in both snapshots.

From the above, we can achieve correct ordering for unique path by always adding Deleted events to the queue first, Modified events second, Moved events third and Created events last:

> Deleted -> Modified -> Moved -> Created

The ordering won't be correct between unrelated paths and between files and folder. The first does not matter for syncing. We solve the second by assuming that when a directory is deleted, so are its children. And before a child is created, its parent directory must exist.

MovedEvents which are not unique (their paths appear in other events) will be split into Deleted and Created events by Maestral.

class maestral.fsevents.polling.**OrderedPollingEmitter**(*event_queue*, *watch*, *timeout=1*, *stat=<built-in function stat>*, *listdir=<built-in function scandir>*)

Bases: watchdog.observers.polling.PollingEmitter

Ordered polling file system event emitter

Platform-independent emitter that polls a directory to detect file system changes. Events are emitted in an order which can be used to produce the new file system state from the old one.

queue_events(*timeout*)

class maestral.fsevents.polling.**OrderedPollingObserver**(*timeout=1*)

Bases: watchdog.observers.polling.PollingObserver

13.2 Module contents

This module provides a custom polling file system event emitter for the watchdog package that sorts file system events in an order which can be applied to reproduce the new state from the old state. This is only required for the polling emitter which uses period directory snapshots and compares them with a watchdog.utils.dirsnapshot.DirectorySnapshotDiff to generate file system events.

maestral.fsevents.**Observer**

alias of watchdog.observers.inotify.InotifyObserver

CHAPTER

FOURTEEN

MAESTRAL.LOGGING

This module defines custom logging records and handlers.

class maestral.logging.**CachedHandler**(*level=0*, *maxlen=None*)

Bases: logging.Handler

Handler which stores past records

This is used to populate Maestral's status and error interfaces. The method *wait_for_emit()* can be used from another thread to block until a new record is emitted, for instance to react to state changes.

Parameters

- **level** (*int*) – Initial log level. Defaults to NOTSET.
- **maxlen** (*Optional[int]*) – Maximum number of records to store. If `None`, all records will be stored. Defaults to `None`.

Return type None

cached_records: Deque[logging.LogRecord]

emit(*record*)

Logs the specified log record and saves it to the cache.

Parameters **record** (*logging.LogRecord*) – Log record.

Return type None

wait_for_emit(*timeout*)

Blocks until a new record is emitted.

Parameters **timeout** (*Optional[float]*) – Maximum time to block before returning.

Returns `True` if there was a status change, `False` in case of a timeout.

Return type bool

getLastMessage()

Returns The log message of the last record or an empty string.

Return type str

getAllMessages()

Returns A list of all record messages.

Return type List[str]

clear()
Clears all cached records.

Return type None

class maestral.logging.SdNotificationHandler(*level=0*)
Bases: logging.Handler

Handler which emits messages as systemd notifications

This is useful when used from a systemd service and will do nothing when no NOTIFY_SOCKET is provided.

Initializes the instance - basically setting the formatter to None and the filter list to empty.

notifier = <sdnotify.SystemdNotifier object>

emit(*record*)
Sends the record massage to systemd as service status.

Parameters **record** (*logging.LogRecord*) – Log record.

Return type None

maestral.logging.setup_logging(*config_name*, *log_to_stderr=True*)
Sets up logging handlers for the given config name. The following handlers are installed for the root logger:

- RotatingFileHandler: Writes logs to the appropriate log file for the config. Log level is determined by the config value.
- StreamHandler: Writes logs to stderr. Log level is determined by the config value. This will be replaced by a null handler if `log_to_stderr` is `False`.
- SdNotificationHandler: Sends all log messages of level INFO and higher to the NOTIFY_SOCKET if provided as an environment variable. The log level is fixed.
- JournalHandler: Writes logs to the systemd journal. Log level is determined by the config value. Will be replaced by a null handler if not started as a systemd service or if python-systemd is not installed.

Any previous loggers are cleared.

Parameters

- **config_name** (*str*) – The config name.
- **log_to_stderr** (*bool*) – Whether to log to stderr.

Returns (log_handler_file, log_handler_stream, log_handler_sd, log_handler_journal)

Return type Tuple[logging.handlers.RotatingFileHandler, Union[logging.StreamHandler, logging.NullHandler], *maestral.logging.SdNotificationHandler*, Union[journal.JournalHandler, logging.NullHandler]]

maestral.logging.scoped_logger(*module_name*, *config_name='maestral'*)
Returns a logger for the module `module_name`, scoped to the given config.

Parameters

- **module_name** (*str*) – Module name.
- **config_name** (*str*) – Config name.

Returns Logger instances scoped to the config.

Return type logging.Logger

CHAPTER

FIFTEEN

MAESTRAL.MAIN

This module defines the main API which is exposed to the CLI or GUI.

class maestral.main.**Maestral**(*config_name='maestral'*, *log_to_stderr=False*)

Bases: object

The public API

All methods and properties return objects or raise exceptions which can safely be serialized, i.e., pure Python types. The only exception are instances of *maestral.errors.MaestralApiError*: they need to be registered explicitly with the serpent serializer which is used for communication to frontends.

Sync errors and fatal errors which occur in the sync threads can be read with the properties *sync_errors* and *fatal_errors*, respectively.

Example First create an instance with a new config_name. In this example, we choose "private" to sync a private Dropbox account. Then link the created config to an existing Dropbox account and set up the local Dropbox folder. If successful, invoke *start_sync()* to start syncing.

```
>>> from maestral.main import Maestral
>>> m = Maestral(config_name='private')
>>> 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)
>>> if res == 0:
...     m.create_dropbox_directory('~/Dropbox (Private)')
...     m.start_sync()
```

Parameters

- **config_name** (*str*) – Name of maestral configuration to run. Must not contain any whitespace. If the given config file does exist, it will be created.
- **log_to_stderr** (*bool*) – If `True`, Maestral will print log messages to stderr. When started as a systemd services, this can result in duplicate log messages in the systemd journal. Defaults to `False`.

Return type None

property version: str

Returns the current Maestral version.

get_auth_url()

Returns a URL to authorize access to a Dropbox account. To link a Dropbox account, retrieve an auth token from the URL and link Maestral by calling *link()* with the provided token.

Returns URL to retrieve an OAuth token.

Return type str

link(*token*)
Links Maestral with a Dropbox account using the given access token. The token will be stored for future usage as documented in the `oauth` module. Supported keyring backends are, in order of preference:

- MacOS Keychain
- Any keyring implementing the SecretService Dbus specification
- KWallet
- Gnome Keyring
- Plain text storage

Parameters **token** (*str*) – OAuth token for Dropbox access.

Returns 0 on success, 1 for an invalid token and 2 for connection errors.

Return type int

unlink()
Unlinks the configured Dropbox account but leaves all downloaded files in place. All syncing metadata will be removed as well. Connection and API errors will be handled silently but the Dropbox access key will always be removed from the user's PC.

Raises ***NotLinkedError*** – if no Dropbox account is linked.

Return type None

property config_name: str
The selected configuration.

set_conf(*section*, *name*, *value*)
Sets a configuration option.

Parameters

- **section** (*str*) – Name of section in config file.
- **name** (*str*) – Name of config option.
- **value** (*Any*) – Config value. May be any type accepted by `ast.literal_eval`.

Return type None

get_conf(*section*, *name*)
Gets a configuration option.

Parameters

- **section** (*str*) – Name of section in config file.
- **name** (*str*) – Name of config option.

Returns Config value. May be any type accepted by `ast.literal_eval`.

Return type Any

set_state(*section*, *name*, *value*)
Sets a state value.

Parameters

- **section** (*str*) – Name of section in state file.
- **name** (*str*) – Name of state variable.
- **value** (*Any*) – State value. May be any type accepted by `ast.literal_eval`.

Return type None

`get_state`(*section*, *name*)
Gets a state value.

Parameters

- **section** (*str*) – Name of section in state file.
- **name** (*str*) – Name of state variable.

Returns State value. May be any type accepted by `ast.literal_eval`.

Return type Any

`property dropbox_path: str`
Returns the path to the local Dropbox folder (read only). This will be an empty string if not Dropbox folder has been set up yet. Use *create_dropbox_directory()* or *move_dropbox_directory()* to set or change the Dropbox directory location instead.

Raises ***NotLinkedError*** – if no Dropbox account is linked.

`property excluded_items: List[str]`
The list of files and folders excluded by selective sync. Any changes to this list will be applied immediately if we have already performed the initial sync. I.e., paths which have been added to the list will be deleted from the local drive and paths which have been removed will be downloaded.

Use *exclude_item()* and *include_item()* to add or remove individual items from selective sync.

`property log_level: int`
Log level for log files, stderr and the systemd journal.

`property notification_snooze: float`
Snooze time for desktop notifications in minutes. Defaults to 0 if notifications are not snoozed.

`property notification_level: int`
Level for desktop notifications. See `utils.notify` for level definitions.

`status_change_longpoll`(*timeout=60*)
Blocks until there is a change in status or until a timeout occurs. This method can be used by frontends to wait for status changes without constant polling.

Parameters **timeout** (*Optional[float]*) – Maximum time to block before returning, even if there is no status change.

Returns `True``if there was a status change, ``False` in case of a timeout.

Return type bool

New in version 1.3.0.

`property pending_link: bool`
Indicates if Maestral is linked to a Dropbox account (read only). This will block until the user's keyring is unlocked to load the saved auth token.

`property pending_dropbox_folder: bool`
Indicates if a local Dropbox directory has been configured (read only). This will not check if the configured directory actually exists, starting the sync may still raise a `NoDropboxDirError`.

property pending_first_download: bool
Indicates if the initial download has already occurred (read only).

property paused: bool
Indicates if syncing is paused by the user (read only). This is set by calling `pause()`.

property running: bool
Indicates if sync threads are running (read only). They will be stopped before *start_sync()* is called, when shutting down or because of an exception.

property connected: bool
Indicates if Dropbox servers can be reached (read only).

property status: str
The last status message (read only). This can be displayed as information to the user but should not be relied on otherwise.

property sync_errors: List[Dict[str, Optional[Union[str, Sequence[str]]]]]
A list of current sync errors as dicts (read only). This list is populated by the sync threads. The following keys will always be present but may contain empty values: "type", "inherits", "title", "traceback", "title", "message", "local_path", "dbx_path".

Raises *NotLinkedError* – if no Dropbox account is linked.

property fatal_errors: List[Dict[str, Optional[Union[str, Sequence[str]]]]]
Returns a list of fatal errors as dicts (read only). This does not include lost internet connections or file sync errors which only emit warnings and are tracked and cleared separately. Errors listed here must be acted upon for Maestral to continue syncing.

The following keys will always be present but may contain empty values: "type", "inherits", "title", "traceback", "title", and "message".s

This list is populated from all log messages with level ERROR or higher that have `exc_info` attached.

clear_fatal_errors()
Manually clears all fatal errors. This should be used after they have been resolved by the user through the GUI or CLI.

Return type None

property account_profile_pic_path: str
The path of the current account's profile picture (read only). There may not be an actual file at that path if the user did not set a profile picture or the picture has not yet been downloaded.

get_file_status(*local_path*)
Returns the sync status of a file or folder. The returned status is recursive for folders, e.g., the file status will be "uploading" for a a folder if any file inside that folder is being uploaded.

New in version 1.4.4: Recursive behavior. Previous versions would return "up to date" even if in case of syncing children.

Parameters **local_path** (*str*) – Path to file on the local drive. May be relative to the current working directory.

Returns String indicating the sync status. Can be 'uploading', 'downloading', 'up to date', 'error', or 'unwatched' (for files outside of the Dropbox directory). This will always be 'unwatched' if syncing is paused.

Return type str

get_activity(*limit=100*)
Returns the current upload / download activity.

Parameters **limit** (*Optional[int]*) – Maximum number of items to return. If None, all entries will be returned.

Returns A lists of all sync events currently queued for or being uploaded or downloaded with the events furthest up in the queue coming first.

Raises ***NotLinkedError*** – if no Dropbox account is linked.

Return type List[Dict[str, Optional[Union[str, float, bool]]]]

`get_history`(*limit=100*)

Returns the historic upload / download activity. Up to 1,000 sync events are kept in the database. Any events which occurred before the interval specified by the `keep_history` config value are discarded.

Parameters **limit** (*Optional[int]*) – Maximum number of items to return. If None, all entries will be returned.

Returns A lists of all sync events since `keep_history` sorted by time with the oldest event first.

Raises ***NotLinkedError*** – if no Dropbox account is linked.

Return type List[Dict[str, Optional[Union[str, float, bool]]]]

`get_account_info`()

Returns the account information from Dropbox and returns it as a dictionary.

Returns Dropbox account information.

Raises

- ***DropboxAuthError*** – in case of an invalid access token.
- ***DropboxServerError*** – for internal Dropbox errors.
- `ConnectionError` – if the connection to Dropbox fails.
- ***NotLinkedError*** – if no Dropbox account is linked.

Return type Dict[str, Optional[Union[str, float, bool]]]

`get_space_usage`()

Gets the space usage from Dropbox and returns it as a dictionary.

Returns Dropbox space usage information.

Raises

- ***DropboxAuthError*** – in case of an invalid access token.
- ***DropboxServerError*** – for internal Dropbox errors.
- `ConnectionError` – if the connection to Dropbox fails.
- ***NotLinkedError*** – if no Dropbox account is linked.

Return type Dict[str, Optional[Union[str, float, bool]]]

`get_profile_pic`()

Attempts to download the user's profile picture from Dropbox. The picture is saved in Maestral's cache directory for retrieval when there is no internet connection.

Returns Path to saved profile picture or `None` if no profile picture was downloaded.

Raises

- ***DropboxAuthError*** – in case of an invalid access token.
- ***DropboxServerError*** – for internal Dropbox errors.

- **ConnectionError** – if the connection to Dropbox fails.
- ***NotLinkedError*** – if no Dropbox account is linked.

Return type Optional[str]

get_metadata(*dbx_path*)

Returns metadata for a file or folder on Dropbox.

Parameters **dbx_path** (*str*) – Path to file or folder on Dropbox.

Returns Dropbox item metadata as dict. See `dropbox.files.Metadata` for keys and values.

Raises

- ***NotFoundError*** – if there is nothing at the given path.
- ***DropboxAuthError*** – in case of an invalid access token.
- ***DropboxServerError*** – for internal Dropbox errors.
- **ConnectionError** – if the connection to Dropbox fails.
- ***NotLinkedError*** – if no Dropbox account is linked.

Return type Optional[Dict[str, Optional[Union[str, float, bool]]]]

list_folder(*dbx_path*, ***kwargs*)

List all items inside the folder given by `dbx_path`. Keyword arguments are passed on the Dropbox API call `client.DropboxClient.list_folder()`.

Parameters **dbx_path** (*str*) – Path to folder on Dropbox.

Returns List of Dropbox item metadata as dicts. See `dropbox.files.Metadata` for keys and values.

Raises

- ***NotFoundError*** – if there is nothing at the given path.
- ***NotAFolderError*** – if the given path refers to a file.
- ***DropboxAuthError*** – in case of an invalid access token.
- ***DropboxServerError*** – for internal Dropbox errors.
- **ConnectionError** – if the connection to Dropbox fails.
- ***NotLinkedError*** – if no Dropbox account is linked.

Return type List[Dict[str, Optional[Union[str, float, bool]]]]

list_folder_iterator(*dbx_path*, ***kwargs*)

Returns an iterator over items inside the folder given by `dbx_path`. Keyword arguments are passed on the client call `client.DropboxClient.list_folder_iterator()`. Each iteration will yield a list of approximately 500 entries, depending on the number of entries returned by an individual API call.

Parameters **dbx_path** (*str*) – Path to folder on Dropbox.

Returns Iterator over list of Dropbox item metadata as dicts. See `dropbox.files.Metadata` for keys and values.

Raises

- ***NotFoundError*** – if there is nothing at the given path.
- ***NotAFolderError*** – if the given path refers to a file.
- ***DropboxAuthError*** – in case of an invalid access token.

- *DropboxServerError* – for internal Dropbox errors.
- ConnectionError – if the connection to Dropbox fails.
- *NotLinkedError* – if no Dropbox account is linked.

Return type Iterator[List[Dict[str, Optional[Union[str, float, bool]]]]]

list_revisions(*dbx_path*, *limit=10*)

List revisions of old files at the given path dbx_path. This will also return revisions if the file has already been deleted.

Parameters

- **dbx_path** (*str*) – Path to file on Dropbox.
- **limit** (*int*) – Maximum number of revisions to list.

Returns List of Dropbox file metadata as dicts. See dropbox.files.Metadata for keys and values.

Return type List[Dict[str, Optional[Union[str, float, bool]]]]

:raises NotFoundError:if there never was a file at the given path. :raises IsAFolderError: if the given path refers to a folder :raises DropboxAuthError: in case of an invalid access token. :raises DropboxServerError: for internal Dropbox errors. :raises ConnectionError: if the connection to Dropbox fails.

get_file_diff(*old_rev*, *new_rev=None*)

Compare to revisions of a text file using Python's difflib. The versions will be downloaded to temporary files. If new_rev is None, the old revision will be compared to the corresponding local file, if any.

Parameters

- **old_rev** (*str*) – Identifier of old revision.
- **new_rev** (*Optional[str]*) – Identifier of new revision.

Returns Diff as a list of strings (lines).

Raises

- *UnsupportedFileTypeForDiff* – if file type is not supported.
- *UnsupportedFileTypeForDiff* – if file content could not be decoded.
- *MaestralApiError* – if file could not be read for any other reason.

Return type List[str]

restore(*dbx_path*, *rev*)

Restore an old revision of a file.

Parameters

- **dbx_path** (*str*) – The path to save the restored file.
- **rev** (*str*) – The revision to restore. Old revisions can be listed with *list_revisions()*.

Returns Metadata of the returned file. See dropbox.files.FileMetadata for keys and values.

Raises

- *DropboxAuthError* – in case of an invalid access token.
- *DropboxServerError* – for internal Dropbox errors.
- ConnectionError – if the connection to Dropbox fails.

Return type Dict[str, Optional[Union[str, float, bool]]]

rebuild_index()
Rebuilds the rev file by comparing remote with local files and updating rev numbers from the Dropbox server. Files are compared by their content hashes and conflicting copies are created if the contents differ. File changes during the rebuild process will be queued and uploaded once rebuilding has completed.

Rebuilding will be performed asynchronously and errors can be accessed through *sync_errors* or maestral_errors.

Raises

- ***NotLinkedError*** – if no Dropbox account is linked.
- ***NoDropboxDirError*** – if local Dropbox folder is not set up.

Return type None

start_sync()
Creates syncing threads and starts syncing.

Raises

- ***NotLinkedError*** – if no Dropbox account is linked.
- ***NoDropboxDirError*** – if local Dropbox folder is not set up.

Return type None

stop_sync()
Stops all syncing threads if running. Call *start_sync()* to restart syncing.

Return type None

reset_sync_state()
Resets the sync index and state. Only call this to clean up leftover state information if a Dropbox was improperly unlinked (e.g., auth token has been manually deleted). Otherwise leave state management to Maestral.

Raises ***NotLinkedError*** – if no Dropbox account is linked.

Return type None

set_excluded_items(*items*)

Parameters **items** (*List[str]*) –

Return type None

exclude_item(*dbx_path*)
Excludes file or folder from sync and deletes it locally. It is safe to call this method with items which have already been excluded.

Parameters **dbx_path** (*str*) – Dropbox path of item to exclude.

Raises

- ***NotFoundError*** – if there is nothing at the given path.
- **ConnectionError** – if the connection to Dropbox fails.
- ***DropboxAuthError*** – in case of an invalid access token.
- ***DropboxServerError*** – for internal Dropbox errors.
- **ConnectionError** – if the connection to Dropbox fails.

- *NotLinkedError* – if no Dropbox account is linked.
- *NoDropboxDirError* – if local Dropbox folder is not set up.

Return type None

include_item(*dbx_path*)

Includes a file or folder in sync and downloads it in the background. It is safe to call this method with items which have already been included, they will not be downloaded again.

If the path lies inside an excluded folder, all its immediate parents will be included. Other children of the excluded folder will remain excluded.

If any children of dbx_path were excluded, they will now be included.

Any downloads will be carried out by the sync threads. Errors during the download can be accessed through *sync_errors* or `maestral_errors`.

Parameters **dbx_path** (*str*) – Dropbox path of item to include.

Raises

- *NotFoundError* – if there is nothing at the given path.
- *DropboxAuthError* – in case of an invalid access token.
- *DropboxServerError* – for internal Dropbox errors.
- `ConnectionError` – if the connection to Dropbox fails.
- *NotLinkedError* – if no Dropbox account is linked.
- *NoDropboxDirError* – if local Dropbox folder is not set up.

Return type None

excluded_status(*dbx_path*)

Returns 'excluded', 'partially excluded' or 'included'. This function will not check if the item actually exists on Dropbox.

Parameters **dbx_path** (*str*) – Path to item on Dropbox.

Returns Excluded status.

Raises *NotLinkedError* – if no Dropbox account is linked.

Return type str

move_dropbox_directory(*new_path*)

Sets the local Dropbox directory. This moves all local files to the new location and resumes syncing afterwards.

Parameters **new_path** (*str*) – Full path to local Dropbox folder. "~" will be expanded to the user's home directory.

Raises

- `OSError` – if moving the directory fails.
- *NotLinkedError* – if no Dropbox account is linked.
- *NoDropboxDirError* – if local Dropbox folder is not set up.

Return type None

create_dropbox_directory(*path*)

Creates a new Dropbox directory. Only call this during setup.

Parameters **path** (*str*) – Full path to local Dropbox folder. "~" will be expanded to the user's home directory.

Raises

- **OSError** – if creation fails.
- ***NotLinkedError*** – if no Dropbox account is linked.

Return type None

create_shared_link(*dbx_path*, *visibility='public'*, *password=None*, *expires=None*)

Creates a shared link for the given **dbx_path**. Returns a dictionary with information regarding the link, including the URL, access permissions, expiry time, etc. The shared link will grant read / download access only. Note that basic accounts do not support password protection or expiry times.

Parameters

- **dbx_path** (*str*) – Path to item on Dropbox.
- **visibility** (*str*) – Requested visibility of the shared link. Must be "public", "team_only" or "password". The actual visibility may be different, depending on the team and folder settings. Inspect the "link_permissions" entry of the returned dictionary.
- **password** (*Optional[str]*) – An optional password required to access the link. Will be ignored if the visibility is not "password".
- **expires** (*Optional[float]*) – An optional expiry time for the link as POSIX timestamp.

Returns Shared link information as dict. See dropbox.sharing.SharedLinkMetadata for keys and values.

Raises

- **ValueError** – if visibility is 'password' but no password is provided.
- ***DropboxAuthError*** – in case of an invalid access token.
- ***DropboxServerError*** – for internal Dropbox errors.
- **ConnectionError** – if the connection to Dropbox fails.
- ***NotLinkedError*** – if no Dropbox account is linked.

Return type Dict[str, Optional[Union[str, float, bool]]]

revoke_shared_link(*url*)

Revokes the given shared link. Note that any other links to the same file or folder will remain valid.

Parameters **url** (*str*) – URL of shared link to revoke.

Raises

- ***DropboxAuthError*** – in case of an invalid access token.
- ***DropboxServerError*** – for internal Dropbox errors.
- **ConnectionError** – if the connection to Dropbox fails.
- ***NotLinkedError*** – if no Dropbox account is linked.

Return type None

list_shared_links(*dbx_path=None*)

Returns a list of all shared links for the given Dropbox path. If no path is given, return all shared links for the account, up to a maximum of 1,000 links.

Parameters **dbx_path** (*Optional[str]*) – Path to item on Dropbox.

Returns List of shared link information as dictionaries. See `dropbox.sharing.SharedLinkMetadata` for keys and values.

Raises

- ***DropboxAuthError*** – in case of an invalid access token.
- ***DropboxServerError*** – for internal Dropbox errors.
- **ConnectionError** – if the connection to Dropbox fails.
- ***NotLinkedError*** – if no Dropbox account is linked.

Return type List[Dict[str, Optional[Union[str, float, bool]]]]

to_local_path(*dbx_path*)

Converts a path relative to the Dropbox folder to a correctly cased local file system path.

Parameters **dbx_path** (*str*) – Path relative to Dropbox root.

Returns Corresponding path on local hard drive.

Raises

- ***NotLinkedError*** – if no Dropbox account is linked.
- ***NoDropboxDirError*** – if local Dropbox folder is not set up.

Return type str

check_for_updates()

Checks if an update is available.

Returns A dictionary with information about the latest release with the fields 'update_available' (bool), 'latest_release' (str), 'release_notes' (str) and 'error' (str or None).

Return type Dict[str, Optional[Union[str, bool]]]

shutdown_daemon()

Stop syncing and clean up our asyncio tasks. Set a result for the `shutdown_complete` future.

Return type None

CHAPTER

SIXTEEN

MAESTRAL.MANAGER

This module contains the classes to coordinate sync threads.

class maestral.manager.**SyncManager**(*client*)

Bases: object

Class to manage sync threads

Parameters **client** (maestral.client.DropboxClient) – The Dropbox API client, a wrapper around the Dropbox Python SDK.

added_item_queue: Queue[str]

Queue of dropbox paths which have been newly included in syncing.

property reindex_interval: float

Interval in sec for period reindexing. Changes will be saved to state file.

property activity: Dict[str, *maestral.database.SyncEvent*]

Returns a list all items queued for or currently syncing.

property history: List[*maestral.database.SyncEvent*]

A list of the last SyncEvents in our history. History will be kept for the interval specified by the config value `keep_history` (defaults to two weeks) but at most 1,000 events will kept.

property idle_time: float

Returns the idle time in seconds since the last file change or since startup if there haven't been any changes in our current session.

start()

Creates observer threads and starts syncing.

Return type None

stop()

Stops syncing and destroys worker threads.

Return type None

reset_sync_state()

Resets all saved sync state. Settings are not affected.

Return type None

rebuild_index()

Rebuilds the rev file by comparing remote with local files and updating rev numbers from the Dropbox server. Files are compared by their content hashes and conflicting copies are created if the contents differ. File changes during the rebuild process will be queued and uploaded once rebuilding has completed.

Rebuilding will be performed asynchronously.

Return type None

check_and_update_path_root()
Checks if the user's root namespace corresponds to the currently configured path root. Updates the root namespace if required and migrates the local folder structure. Syncing will be paused during the migration.

Returns Whether the path root was updated.

Return type bool

connection_monitor()
Monitors the connection to Dropbox servers. Pauses syncing when the connection is lost and resumes syncing when reconnected and syncing has not been paused by the user.

Return type None

download_worker(*running*, *startup_completed*, *autostart*)
Worker to sync changes of remote Dropbox with local folder.

Parameters

- **running** (*threading.Event*) – Event to shutdown local file event handler and worker threads.
- **startup_completed** (*threading.Event*) – Set when startup sync is completed.
- **autostart** (*threading.Event*) – Set when syncing should automatically resume on connection.

Return type None

download_worker_added_item(*running*, *startup_completed*, *autostart*)
Worker to download items which have been newly included in sync.

Parameters

- **running** (*threading.Event*) – Event to shutdown local file event handler and worker threads.
- **startup_completed** (*threading.Event*) – Set when startup sync is completed.
- **autostart** (*threading.Event*) – Set when syncing should automatically resume on connection.

Return type None

upload_worker(*running*, *startup_completed*, *autostart*)
Worker to sync local changes to remote Dropbox.

Parameters

- **running** (*threading.Event*) – Event to shutdown local file event handler and worker threads.
- **startup_completed** (*threading.Event*) – Set when startup sync is completed.
- **autostart** (*threading.Event*) – Set when syncing should automatically resume on connection.

Return type None

startup_worker(*running*, *startup_completed*, *autostart*)
Worker to sync local changes to remote Dropbox.

Parameters

- **running** (*threading.Event*) – Event to shutdown local file event handler and worker threads.
- **startup_completed** (*threading.Event*) – Set when startup sync is completed.
- **autostart** (*threading.Event*) – Set when syncing should automatically resume on connection.

Return type None

CHAPTER

SEVENTEEN

MAESTRAL.NOTIFY

This module handles desktop notifications It uses the desktop-notifier package as a backend for cross-platform desktop notifications.

maestral.notify.**NONE = 100**
No desktop notifications

maestral.notify.**ERROR = 40**
Notify only on fatal errors

maestral.notify.**SYNCISSUE = 30**
Notify for sync issues and higher

maestral.notify.**FILECHANGE = 15**
Notify for all remote file changes

maestral.notify.**level_name_to_number**(*name*)
Converts a Maestral notification level name to number.

Parameters **name** (*str*) – Level name.

Returns Level number.

Return type int

maestral.notify.**level_number_to_name**(*number*)
Converts a Maestral notification level number to name.

Parameters **number** (*int*) – Level number.

Returns Level name.

Return type str

class maestral.notify.**MaestralDesktopNotifier**(*config_name*)
Bases: object

Desktop notification emitter for Maestral

Desktop notifier with snooze functionality and variable notification levels. Must be instantiated in the main thread.

Parameters **config_name** (*str*) – Config name. This is used to access notification settings for the daemon.

Return type None

property notify_level: int
Custom notification level. Notifications with a lower level will be discarded.

property snoozed: float
Time in minutes to snooze notifications. Applied to FILECHANGE level only.

notify(*title*, *message*, *level=15*, *on_click=None*, *actions=None*)
Sends a desktop notification. This will schedule a notification task in the asyncio loop of the thread where `DesktopNotifier` was instantiated.

Parameters

- **title** (*str*) – Notification title.
- **message** (*str*) – Notification message.
- **level** (*int*) – Notification level of the message.
- **on_click** (*Optional[Callable]*) – A callback to execute when the notification is clicked. The provided callable must not take any arguments.
- **actions** (*Optional[Dict[str, Callable]]*) – A dictionary with button names and callbacks for the notification.

Return type None

CHAPTER

EIGHTEEN

MAESTRAL.OAUTH

This module is responsible for authorization and token store in the system keyring.

class maestral.oauth.**OAuth2Session**(*config_name*, *app_key='2jmbq42w7vof78h'*)

Bases: object

Provides Dropbox OAuth flow and key store interface

OAuth2Session provides OAuth 2 login and token store in the preferred system keyring. To authenticate with Dropbox, run *get_auth_url()* first and direct the user to visit that URL and retrieve an auth token. Verify the provided auth token with *verify_auth_token()* and save it in the system keyring together with the corresponding Dropbox ID by calling *save_creds()*. Supported keyring backends are, in order of preference:

- MacOS Keychain
- Any keyring implementing the SecretService Dbus specification
- KWallet
- Plain text storage

When the auth flow is completed, a short-lived access token and a long-lived refresh token are generated. They can be accessed through the properties *access_token* and *refresh_token*. Only the long-lived refresh token will be saved in the system keychain for future sessions, the Dropbox SDK will use it to generate short-lived access tokens as needed.

If the auth flow was previously completed before Dropbox migrated to short-lived tokens, the *token_access_type* will be 'legacy' and only a long-lived access token will be available.

Note: Once the token has been stored with a keyring backend, that backend will be saved in the config file and remembered until the user unlinks the account. This module will therefore never switch keyring backends while linked.

Warning: Unlike MacOS Keychain, Gnome Keyring and KWallet do not support app-specific access to passwords. If the user unlocks those keyrings, we and any other application in the same user session get access to *all* saved passwords.

Parameters

- **config_name** (*str*) – Name of maestral config.
- **app_key** (*str*) – Public key of the app, as registered with Dropbox. Used for the PKCE OAuth 2.0 flow.

Return type None

Success = 0
Exit code for successful auth.

InvalidToken = 1
Exit code for invalid token.

ConnectionFailed = 2
Exit code for connection errors.

default_token_access_type = 'offline'

property keyring: keyring.backend.KeyringBackend
The keyring backend currently being used to store auth tokens.

property linked: bool
Whether we have full auth credentials (read only).

property account_id: Optional[str]
The account ID (read only). This call may block until the keyring is unlocked.

property token_access_type: Optional[str]
The type of access token (read only). If 'legacy', we have a long-lived access token. If 'offline', we have a short-lived access token with an expiry time and a long-lived refresh token to generate new access tokens. This call may block until the keyring is unlocked.

property access_token: Optional[str]
The access token (read only). This will always be set for a 'legacy' token. For an 'offline' token, this will only be set if we completed the auth flow in the current session. In case of an 'offline' token, use the refresh token to retrieve a short-lived access token through the Dropbox API instead. This call may block until the keyring is unlocked.

property refresh_token: Optional[str]
The refresh token (read only). This will only be set for an 'offline' token. This call may block until the keyring is unlocked.

property access_token_expiration: Optional[datetime.datetime]
The expiry time for the short-lived access token (read only). This will only be set for an 'offline' token and if we completed the flow during the current session.

load_token()
Loads auth token from system keyring. This will be called automatically when accessing any of the properties *linked*, *access_token*, *refresh_token* or *token_access_type*. This call may block until the keyring is unlocked.

Raises ***KeyringAccessError*** – if the system keyring is locked or otherwise cannot be accessed (for example if the app bundle signature has been invalidated).

Return type None

get_auth_url()
Retrieves an auth URL to start the OAuth2 implicit grant flow.

Returns Dropbox auth URL.

Return type str

verify_auth_token(*code*)
If the user approves the app, they will be presented with a single usage "authorization code". Have the user copy/paste that authorization code into the app and then call this method to exchange it for a long-lived auth token.

Parameters **code** (*str*) – Ephemeral auth code.

Returns *Success*, *InvalidToken*, or *ConnectionFailed*.

Return type int

`save_creds()`
Saves the auth token to system keyring. Falls back to plain text storage if the user denies access to keyring. This should be called after *verify_auth_token()* returned successfully.

Return type None

`delete_creds()`
Deletes auth token from system keyring.

Raises ***KeyringAccessError*** – if the system keyring is locked or otherwise cannot be accessed (for example if the app bundle signature has been invalidated).

Return type None

CHAPTER

NINETEEN

MAESTRAL.SYNC

This module contains the main syncing functionality.

class maestral.sync.**Conflict**(*value*)

Bases: enum.Enum

Enumeration of sync conflict types

RemoteNewer = 'remote newer'

Conflict = 'conflict'

Identical = 'identical'

LocalNewerOrIdentical = 'local newer or identical'

class maestral.sync.**FSEventHandler**(*file_event_types=('created', 'deleted', 'modified', 'moved')*, *dir_event_types=('created', 'deleted', 'moved')*)

Bases: watchdog.events.FileSystemEventHandler

A local file event handler

Handles captured file events and adds them to *SyncEngine*'s file event queue to be uploaded by `upload_worker()`. This acts as a translation layer between `watchdog.Observer` and *SyncEngine*.

White lists of event types to handle are supplied as `file_event_types` and `dir_event_types`. This is for forward compatibility as additional event types may be added to watchdog in the future.

Parameters

- **file_event_types** (*Tuple[str, ...]*) – Types of file events to handle. This acts as a whitelist.
- **dir_event_types** (*Tuple[str, ...]*) – Types of folder events to handle. This acts as a whitelist.

Variables **ignore_timeout** (*float*) – Timeout in seconds after which filters for ignored events will expire.

Return type None

local_file_event_queue: Queue[FileSystemEvent]

property enabled: bool

Whether queuing of events is enabled.

enable()

Turn on queueing of events.

Return type None

disable()
Turn off queueing of new events and remove all events from queue.

Return type None

ignore(**events*, *recursive=True*)
A context manager to ignore local file events

Once a matching event has been registered, further matching events will no longer be ignored unless `recursive` is `True`. If no matching event has occurred before leaving the context, the event will be ignored for `ignore_timeout` sec after leaving then context and then discarded. This accounts for possible delays in the emission of local file system events.

This context manager is used to filter out file system events caused by maestral itself, for instance during a download or when moving a conflict.

Example Prevent triggereing a sync event when creating a local file:

```
>>> from watchdog.events import FileCreatedEvent
>>> from maestral.main import Maestral
>>> m = Maestral()
>>> with m.sync.fs_events.ignore(FileCreatedEvent('path')):
...     open('path').close()
```

Parameters

- **events** (*watchdog.events.FileSystemEvent*) – Local events to ignore.
- **recursive** (*bool*) – If `True`, all child events of a directory event will be ignored as well. This parameter will be ignored for file events.

Return type Iterator[None]

expire_ignored_events()
Removes all expired ignore entries.

Return type None

on_any_event(*event*)
Checks if the system file event should be ignored. If not, adds it to the queue for events to upload. If syncing is paused or stopped, all events will be ignored.

Parameters **event** (*watchdog.events.FileSystemEvent*) – Watchdog file event.

Return type None

queue_event(*event*)
Queues an individual file system event. Notifies / wakes up all threads that are waiting with *wait_for_event()*.

Parameters **event** (*watchdog.events.FileSystemEvent*) – File system event to queue.

Return type None

wait_for_event(*timeout=40*)
Blocks until an event is available in the queue or a timeout occurs, whichever comes first. You can use with method to wait for file system events in another thread.

Note: If there are multiple threads waiting for events, all of them will be notified. If one of those threads starts getting events from *local_file_event_queue*, other threads may find that the queue is empty

despite being woken. You should therefore be prepared to handle an empty queue even if this method returns `True`.

Parameters **timeout** (*float*) – Maximum time to block in seconds.

Returns `True` if an event is available, `False` if the call returns due to a timeout.

Return type bool

class maestral.sync.**SyncEngine**(*client*)

Bases: `object`

Class that handles syncing with Dropbox

Provides methods to wait for local or remote changes and sync them, including conflict resolution and updates to our index.

Parameters **client** (`maestral.client.DropboxClient`) – Dropbox API client instance.

sync_errors: Set[*maestral.errors.SyncError*]

syncing: Dict[str, *maestral.database.SyncEvent*]

reload_cached_config()

Reloads all config and state values that are otherwise cached by this class for faster access. Call this method if config or state values where modified directly instead of using *SyncEngine* APIs.

Return type None

property dropbox_path: str

Path to local Dropbox folder, as loaded from the config file. Before changing *dropbox_path*, make sure that syncing is paused. Move the dropbox folder to the new location before resuming the sync. Changes are saved to the config file.

property database_path: str

Path SQLite database.

property file_cache_path: str

Path to cache folder for temporary files (read only). The cache folder '.maestral.cache' is located inside the local Dropbox folder to prevent file transfer between different partitions or drives during sync.

property excluded_items: List[str]

List of all files and folders excluded from sync. Changes are saved to the config file. If a parent folder is excluded, its children will automatically be removed from the list. If only children are given but not the parent folder, any new items added to the parent will be synced. Change this property *before* downloading newly included items or deleting excluded items.

static clean_excluded_items_list(*folder_list*)

Removes all duplicates and children of excluded items from the excluded items list.

Parameters **folder_list** (*List[str]*) – Dropbox paths to exclude.

Returns Cleaned up items.

Return type List[str]

property max_cpu_percent: float

Maximum CPU usage for parallel downloads or uploads in percent of the total available CPU time per core. Individual workers in a thread pool will pause until the usage drops below this value. Tasks in the main thread such as indexing file changes may still use more CPU time. Setting this to 200% means that two full logical CPU core can be used.

property remote_cursor: str
Cursor from last sync with remote Dropbox. The value is updated and saved to the config file on every successful download of remote changes.

property local_cursor: float
Time stamp from last sync with remote Dropbox. The value is updated and saved to the config file on every successful upload of local changes.

property last_change: float
The time stamp of the last file change or 0.0 if there are no file changes in our history.

property last_reindex: float
Time stamp of last full indexing. This is used to determine when the next full indexing should take place.

property history: List[*maestral.database.SyncEvent*]
A list of the last SyncEvents in our history. History will be kept for the interval specified by the config value `keep_history` (defaults to two weeks) but at most 1,000 events will be kept.

clear_sync_history()
Clears the sync history.

Return type None

reset_sync_state()
Resets all saved sync state. Settings are not affected.

Return type None

get_index()
Returns a copy of the local index of synced files and folders.

Returns List of index entries.

Return type List[*maestral.database.IndexEntry*]

iter_index()
Returns an iterator over the local index of synced files and folders.

Returns Iterator over index entries.

Return type Iterator[*maestral.database.IndexEntry*]

index_count()
Returns the number if items in our index without loading any items.

Returns Number of index entries.

Return type int

get_local_rev(*dbx_path_lower*)
Gets revision number of local file.

Parameters **dbx_path_lower** (*str*) – Normalized lower case Dropbox path.

Returns Revision number as str or `None` if no local revision number has been saved.

Return type Optional[str]

get_last_sync(*dbx_path_lower*)
Returns the timestamp of last sync for an individual path.

Parameters **dbx_path_lower** (*str*) – Normalized lower case Dropbox path.

Returns Time of last sync.

Return type float

get_index_entry(*dbx_path_lower*)
Gets the index entry for the given Dropbox path.

Parameters **dbx_path_lower** (*str*) – Normalized lower case Dropbox path.

Returns Index entry or `None` if no entry exists for the given path.

Return type Optional[*maestral.database.IndexEntry*]

get_local_hash(*local_path*)
Computes content hash of a local file.

Parameters **local_path** (*str*) – Absolute path on local drive.

Returns Content hash to compare with Dropbox's content hash, or 'folder' if the path points to a directory. `None` if there is nothing at the path.

Return type Optional[str]

clear_hash_cache()
Clears the sync history.

Return type None

update_index_from_sync_event(*event*)
Updates the local index from a SyncEvent.

Parameters **event** (*maestral.database.SyncEvent*) – SyncEvent from download.

Return type None

update_index_from_dbx_metadata(*md*, *client=None*)
Updates the local index from Dropbox metadata.

Parameters

- **md** (*dropbox.files.Metadata*) – Dropbox metadata.
- **client** (*Optional[maestral.client.DropboxClient]*) – DropboxClient instance to use. If not given, use the global instance.

Return type None

remove_node_from_index(*dbx_path_lower*)
Removes any local index entries for the given path and all its children.

Parameters **dbx_path_lower** (*str*) – Normalized lower case Dropbox path.

Return type None

clear_index()
Clears the revision index.

Return type None

property mignore_path: str
Path to mignore file on local drive (read only).

property mignore_rules: pathspec.pathspec.PathSpec
List of mignore rules following git wildmatch syntax (read only).

load_mignore_file()
Loads rules from mignore file. No rules are loaded if the file does not exist or cannot be read.

Returns PathSpec instance with ignore patterns.

Return type None

ensure_dropbox_folder_present()
Checks if the Dropbox folder still exists where we expect it to be.

Raises *NoDropboxDirError* – When local Dropbox directory does not exist.

Return type None

ensure_cache_dir_present()
Checks for or creates a directory at *file_cache_path*.

Raises *CacheDirError* – When local cache directory cannot be created.

Return type None

clean_cache_dir(*raise_error=True*)
Removes all items in the cache directory.

Parameters **raise_error** (*bool*) – Whether errors should raised or only logged.

Return type None

correct_case(*dbx_path*, *client=None*)
Converts a Dropbox path with correctly cased basename to a fully cased path. This is useful because the Dropbox API guarantees the correct casing for the basename only. In practice, casing of parent directories is often incorrect. This method retrieves the correct casing of of all ancestors in the path, either from our cache, our database, or from Dropbox servers.

Performance may vary significantly with the number of parent folders and the method used to resolve the casing of all parent directory names:

1) If the parent directory is already in our cache, performance is O(1).
2) If the parent directory is already in our sync index, performance is slower because it requires a sqlite query but still O(1).
3) If the parent directory is unknown to us, its metadata (including the correct casing of directory's basename) is queried from Dropbox. This is used to construct a correctly cased path by calling *correct_case()* again. At best, performance will be of O(2) if the parent directory is known to us, at worst if will be of order O(n) involving queries to Dropbox servers for each parent directory.

When calling *correct_case()* repeatedly for paths from the same tree, it is therefore best to do so in hierarchical order.

Parameters

- **dbx_path** (*str*) – Dropbox path with correctly cased basename, as provided by dropbox.files.Metadata.path_display or dropbox.files.Metadata.name.
- **client** (*Optional[maestral.client.DropboxClient]*) – Client instance to use. If not given, use the instance provided in the constructor.

Returns Correctly cased Dropbox path.

Return type str

to_dbx_path(*local_path*)
Converts a local path to a path relative to the Dropbox folder. Casing of the given `local_path` will be preserved.

Parameters **local_path** (*str*) – Absolute path on local drive.

Returns Relative path with respect to Dropbox folder.

Raises ValueError – When the path lies outside of the local Dropbox folder.

Return type str

to_dbx_path_lower(*local_path*)
Converts a local path to a path relative to the Dropbox folder. The path will be normalized as on Dropbox servers (lower case and some additional normalisations).

Parameters **local_path** (*str*) – Absolute path on local drive.

Returns Relative path with respect to Dropbox folder.

Raises **ValueError** – When the path lies outside of the local Dropbox folder.

Return type str

to_local_path_from_cased(*dbx_path_cased*)
Converts a correctly cased Dropbox path to the corresponding local path. This is more efficient than *to_local_path()* which accepts uncased paths.

Parameters **dbx_path_cased** (*str*) – Path relative to Dropbox folder, correctly cased.

Returns Corresponding local path on drive.

Return type str

to_local_path(*dbx_path*, *client=None*)
Converts a Dropbox path to the corresponding local path. Only the basename must be correctly cased, as guaranteed by the Dropbox API for the `display_path` attribute of file or folder metadata.

This method slower than *to_local_path_from_cased()*.

Parameters

- **dbx_path** (*str*) – Path relative to Dropbox folder, must be correctly cased in its basename.
- **client** (*Optional[maestral.client.DropboxClient]*) – Client instance to use. If not given, use the instance provided in the constructor.

Returns Corresponding local path on drive.

Return type str

has_sync_errors()
Returns `True` in case of sync errors, `False` otherwise.

Return type bool

clear_sync_error(*local_path=None*, *dbx_path=None*)
Clears all sync errors for `local_path` or `dbx_path`.

Parameters

- **local_path** (*Optional[str]*) – Absolute path on local drive.
- **dbx_path** (*Optional[str]*) – Path relative to Dropbox folder.

Return type None

clear_sync_errors()
Clears all sync errors.

Return type None

static is_excluded(*path*)
Checks if a file is excluded from sync. Certain file names are always excluded from syncing, following the Dropbox support article:

https://help.dropbox.com/installs-integrations/sync-uploads/files-not-syncing

This includes file system files such as 'desktop.ini' and '.DS_Store' and some temporary files as well as caches used by Dropbox or Maestral. *is_excluded* accepts both local and Dropbox paths.

Parameters **path** (*str*) – Can be an absolute path, a path relative to the Dropbox folder or just a file name. Does not need to be normalized.

Returns Whether the path is excluded from syncing.

Return type bool

is_excluded_by_user(*dbx_path_lower*)
Check if file has been excluded through "selective sync" by the user.

Parameters **dbx_path_lower** (*str*) – Normalised lower case Dropbox path.

Returns Whether the path is excluded from download syncing by the user.

Return type bool

is_mignore(*event*)
Check if local file change has been excluded by an mignore pattern.

Parameters **event** (maestral.database.SyncEvent) – SyncEvent for local file event.

Returns Whether the path is excluded from upload syncing by the user.

Return type bool

cancel_sync()
Raises a *maestral.errors.CancelledError* in all sync threads and waits for them to shut down.

Return type None

busy()
Checks if we are currently syncing.

Returns `True` if `sync_lock` cannot be acquired, `False` otherwise.

Return type bool

upload_local_changes_while_inactive()
Collects changes while sync has not been running and uploads them to Dropbox. Call this method when resuming sync.

Return type None

wait_for_local_changes(*timeout=40*)
Blocks until local changes are available.

Parameters **timeout** (*float*) – Maximum time in seconds to wait.

Returns `True` if changes are available, `False` otherwise.

Return type bool

upload_sync_cycle()
Performs a full upload sync cycle by calling in order:

1) *list_local_changes()*
2) *apply_local_changes()*

Handles updating the local cursor for you. If monitoring for local file events was interrupted, call *upload_local_changes_while_inactive()* instead.

list_local_changes(*delay=1*)
Waits for local file changes. Returns a list of local changes with at most one entry per path.

Parameters **delay** (*float*) – Delay in sec to wait for subsequent changes before returning.

Returns (list of sync times events, time_stamp)

Return type Tuple[List[*maestral.database.SyncEvent*], float]

apply_local_changes(*sync_events*)

Applies locally detected changes to the remote Dropbox. Changes which should be ignored (mignore or always ignored files) are skipped.

Parameters **sync_events** (*List[maestral.database.SyncEvent]*) – List of local file system events.

Return type List[*maestral.database.SyncEvent*]

get_remote_item(*dbx_path*, *client=None*)

Downloads a remote file or folder and updates its local rev. If the remote item does not exist, any corresponding local items will be deleted. If `dbx_path` refers to a folder, the download will be handled by `_get_remote_folder()`. If it refers to a single file, the download will be performed by `_create_local_entry()`.

This method can be used to fetch individual items outside of the regular sync cycle, for instance when including a previously excluded file or folder.

Parameters

- **dbx_path** (*str*) – Path relative to Dropbox folder.
- **client** (*Optional[maestral.client.DropboxClient]*) – Client instance to use. If not given, use the instance provided in the constructor.

Returns Whether download was successful.

Return type bool

wait_for_remote_changes(*last_cursor*, *timeout=40*, *client=None*)

Blocks until changes to the remote Dropbox are available.

Parameters

- **last_cursor** (*str*) – Cursor form last sync.
- **timeout** (*int*) – Timeout in seconds before returning even if there are no changes. Dropbox adds random jitter of up to 90 sec to this value.
- **client** (*Optional[maestral.client.DropboxClient]*) – Client instance to use. If not given, use the instance provided in the constructor.

Returns `True` if changes are available, `False` otherwise.

Return type bool

download_sync_cycle(*client=None*)

Performs a full download sync cycle by calling in order:

1) *list_remote_changes_iterator()*

2) *apply_remote_changes()*

Handles updating the remote cursor and resuming interrupted syncs for you. Calling this method will perform a full indexing if this is the first download.

Parameters **client** (*Optional[maestral.client.DropboxClient]*) – Client instance to use. If not given, use the instance provided in the constructor.

Return type None

`list_remote_changes_iterator`(*last_cursor*, *client=None*)

Get remote changes since the last download sync, as specified by `last_cursor`. If the `last_cursor` is from paginating through a previous set of changes, continue where we left off. If `last_cursor` is an emtpy string, tart a full indexing of the Dropbox folder.

Parameters

- **last_cursor** (*str*) – Cursor from last download sync.
- **client** (*Optional[maestral.client.DropboxClient]*) – Client instance to use. If not given, use the instance provided in the constructor.

Returns Iterator yielding tuples with remote changes and corresponding cursor.

Return type Iterator[Tuple[List[*maestral.database.SyncEvent*], str]]

`apply_remote_changes`(*sync_events*)

Applies remote changes to local folder. Call this on the result of `list_remote_changes()`. The saved cursor is updated after a set of changes has been successfully applied. Entries in the local index are created after successful completion.

Parameters **sync_events** (*List[maestral.database.SyncEvent]*) – List of remote changes.

Returns List of changes that were made to local files and bool indicating if all download syncs were successful.

Return type List[*maestral.database.SyncEvent*]

`notify_user`(*sync_events*, *client=None*)

Shows a desktop notification for the given file changes.

Parameters

- **sync_events** (*List[maestral.database.SyncEvent]*) – List of SyncEvents from download sync.
- **client** (*Optional[maestral.client.DropboxClient]*) – Client instance to use. If not given, use the instance provided in the constructor.

Return type None

`rescan`(*local_path*)

Forces a rescan of a local path: schedules created events for every folder, modified events for every file and deleted events for every deleted item (compared to our index).

Parameters **local_path** (*str*) – Path to rescan.

Return type None

CHAPTER

TWENTY

MAESTRAL.UTILS

20.1 Submodules

20.1.1 maestral.utils.appdirs

This module contains functions to retrieve platform dependent locations to store app data. It supports macOS and Linux.

maestral.utils.appdirs.**get_log_path**(*subfolder=None*, *filename=None*, *create=True*)
Returns the default log path for the platform. This will be:

- macOS: "~/Library/Logs/SUBFOLDER/FILENAME"
- Linux: "$XDG_CACHE_HOME/SUBFOLDER/FILENAME"
- fallback: "$HOME/.cache/SUBFOLDER/FILENAME"

Parameters

- **subfolder** (*Optional[str]*) – The subfolder for the app.
- **filename** (*Optional[str]*) – The filename to append for the app.
- **create** (*bool*) – If `True`, the folder `subfolder` will be created on-demand.

Return type str

maestral.utils.appdirs.**get_cache_path**(*subfolder=None*, *filename=None*, *create=True*)
Returns the default cache path for the platform. This will be:

- macOS: "~/Library/Caches/SUBFOLDER/FILENAME"
- Linux: "$XDG_CACHE_HOME/SUBFOLDER/FILENAME"
- fallback: "$HOME/.cache/SUBFOLDER/FILENAME"

Parameters

- **subfolder** (*Optional[str]*) – The subfolder for the app.
- **filename** (*Optional[str]*) – The filename to append for the app.
- **create** (*bool*) – If `True`, the folder `subfolder` will be created on-demand.

Return type str

maestral.utils.appdirs.**get_autostart_path**(*filename=None*, *create=True*)
Returns the default path for login items for the platform. This will be:

- macOS: "~/Library/LaunchAgents/FILENAME"
- Linux: "$XDG_CONFIG_HOME/autostart/FILENAME"
- fallback: "$HOME/.config/autostart/FILENAME"

Parameters

- **filename** (*Optional[str]*) – The filename to append for the app.
- **create** (*bool*) – If `True`, the folder `subfolder` will be created on-demand.

Return type str

maestral.utils.appdirs.**get_runtime_path**(*subfolder=None*, *filename=None*, *create=True*)
Returns the default runtime path for the platform. This will be:

- macOS: "~/Library/Application Support/SUBFOLDER/FILENAME"
- Linux: "$XDG_RUNTIME_DIR/SUBFOLDER/FILENAME"
- fallback: "$HOME/.cache/SUBFOLDER/FILENAME"

Parameters

- **subfolder** (*Optional[str]*) – The subfolder for the app.
- **filename** (*Optional[str]*) – The filename to append for the app.
- **create** (*bool*) – If `True`, the folder `subfolder` will be created on-demand.

Return type str

maestral.utils.appdirs.**get_conf_path**(*subfolder=None*, *filename=None*, *create=True*)
Returns the default config path for the platform. This will be:

- macOS: "~/Library/Application Support/<subfolder>/<filename>."
- Linux: ``XDG_CONFIG_HOME/<subfolder>/<filename>"
- other: "~/.config/<subfolder>/<filename>"

Parameters

- **subfolder** (*Optional[str]*) – The subfolder for the app.
- **filename** (*Optional[str]*) – The filename to append for the app.
- **create** (*bool*) – If `True`, the folder `subfolder` will be created on-demand.

Return type str

maestral.utils.appdirs.**get_home_dir**()
Returns user home directory. This will be determined from the first valid result out of (osp.expanduser("~"), $HOME, $USERPROFILE, $TMP).

Return type str

maestral.utils.appdirs.**get_data_path**(*subfolder=None*, *filename=None*, *create=True*)
Returns the default path to save application data for the platform. This will be:

- macOS: "~/Library/Application Support/SUBFOLDER/FILENAME"
- Linux: "$XDG_DATA_DIR/SUBFOLDER/FILENAME"
- fallback: "$HOME/.local/share/SUBFOLDER/FILENAME"

Note: We do not use "~/Library/Saved Application State" on macOS since this folder is reserved for user interface state and can be cleared by the user / system.

Parameters

- **subfolder** (*Optional[str]*) – The subfolder for the app.
- **filename** (*Optional[str]*) – The filename to append for the app.
- **create** (*bool*) – If `True`, the folder `subfolder` will be created on-demand.

Return type str

20.1.2 maestral.utils.caches

Module containing cache implementations.

class maestral.utils.caches.**LRUCache**(*capacity*)

Bases: object

A simple LRU cache implementation

Parameters **capacity** (*int*) – Maximum number of of entries to keep.

Return type None

get(*key*)

Get the cached value for a key. Mark as most recently used.

Parameters **key** (*Any*) – Key to query.

Returns Cached value or None.

Return type Any

put(*key*, *value*)

Set the cached value for a key. Mark as most recently used.

Parameters

- **key** (*Any*) – Key to use. Must be hashable.
- **value** (*Any*) – Value to cache.

Return type None

clear()

Clears the cache.

Return type None

20.1.3 maestral.utils.cli

Module to print neatly formatted tables and grids to the terminal.

class maestral.utils.cli.**Align**(*value*)

Bases: enum.Enum

Text alignment in column

Left = 0

Right = 1

class maestral.utils.cli.**Elide**(*value*)
Bases: enum.Enum

Elide directives

Leading = 0

Center = 1

Trailing = 2

class maestral.utils.cli.**Prefix**(*value*)
Bases: enum.Enum

Prefix for command line output

Info = 0

Ok = 1

Warn = 2

NONE = 3

maestral.utils.cli.**elide**(*text*, *width*, *placeholder='...'*, *elide=Elide.Trailing*)
Elides a string to fit into the given width.

Parameters

- **text** (*str*) – Text to truncate.
- **width** (*int*) – Target width.
- **placeholder** (*str*) – Placeholder string to indicate truncated text.
- **elide** (maestral.utils.cli.Elide) – Which part to truncate.

Returns Truncated text.

Return type str

maestral.utils.cli.**adjust**(*text*, *width*, *align=Align.Left*)
Pads a string with spaces up the desired width. Preserves ANSI color codes without counting them towards the width.

This function is similar to `str.ljust` and `str.rjust`.

Parameters

- **text** (*str*) – Initial string.
- **width** (*int*) – Target width. If smaller than the given text, nothing is done.
- **align** (maestral.utils.cli.Align) – Side to align the padded string: to the left or to the right.

Return type str

class maestral.utils.cli.**Field**
Bases: object

Base class to represent a field in a table.

property display_width: int
The requested total width of the content in characters when not wrapped or shortened in any way.

format(*width*)
Returns the field content formatted to fit the requested width.

Parameters **width** (*int*) – Width to fit.

Returns Shortened or wrapped string.

Return type List[str]

class maestral.utils.cli.**TextField**(*text*, *align=Align.Left*, *wraps=False*, *elide=Elide.Trailing*, ***style*)

Bases: *maestral.utils.cli.Field*

A text field for a table.

Parameters

- **text** (*str*) – Text to represent.
- **align** (maestral.utils.cli.Align) – Text alignment: right or left.
- **wraps** (*bool*) – Whether to wrap the text instead of truncating it to fit into a requested width.
- **style** – Styling passed on to `click.style()` when styling the text.
- **elide** (maestral.utils.cli.Elide) –

Elide Truncation strategy: trailing, center or leading.

Return type None

property display_width: **int**

format(*width*)

Parameters **width** (*int*) –

Return type List[str]

class maestral.utils.cli.**DateField**(*dt*, ***style*)

Bases: *maestral.utils.cli.Field*

A datetime field for a table. The formatting of the datetime will be adjusted depending on the available width. Does not currently support localisation.

Parameters

- **dt** (*datetime*) – Datetime to represent.
- **style** – Styling passed on to `click.style()` when styling the text.

Return type None

property display_width: **int**

format(*width*)

Parameters **width** (*int*) –

Return type List[str]

class maestral.utils.cli.**Column**(*title*, *fields=()*, *align=Align.Left*, *wraps=False*, *elide=Elide.Trailing*)

Bases: object

A table column.

Parameters

- **title** (*Optional[str]*) – Column title.

- **fields** (*List[maestral.utils.cli.Field]*) – Fields in the table. Any sequence of objects can be given and will be converted to *Field* instances as appropriate.
- **align** (*maestral.utils.cli.Align*) – How to align text inside the column. Will only be used for :class:`TextField`s.
- **wraps** (*bool*) – Whether to wrap fields to fit into the column width instead of truncating them. Will only be used for :class:`TextField`s.
- **elide** (*maestral.utils.cli.Elide*) – How to elide text which is too wide for a column. Will only be used for :class:`TextField`s.

Return type None

fields: List[*maestral.utils.cli.Field*]

property display_width: int

property has_title

append(*field*)

Parameters **field** (*Any*) –

Return type None

insert(*index*, *field*)

Parameters

- **index** (*int*) –
- **field** (*Any*) –

Return type None

class maestral.utils.cli.**Table**(*columns*, *padding=2*)

Bases: object

A table which can be printed to stdout.

Parameters

- **columns** (*List[maestral.utils.cli.Column]*) – Table columns. Can be a list of *Column* instances or table titles.
- **padding** (*int*) – Padding between columns.

Return type None

columns: List[*maestral.utils.cli.Column*]

property ncols: int

The number of columns

property nrows: int

The number of rows

append(*row*)

Appends a new row to the table.

Parameters **row** (*Sequence*) – List of fields to append to each column. Length must match the number of columns.

Return type None

rows()
Returns a list of rows in the table. Each row is a list of fields.

Return type List[List[*maestral.utils.cli.Field*]]

format_lines(*width=None*)
Iterator over formatted lines of the table. Fields may span multiple lines if they are set to wrap instead of truncate.

Parameters **width** (*Optional[int]*) – Width to fit the table.

Returns Iterator over lines which can be printed to the terminal.

Return type Iterator[str]

format(*width=None*)
Returns a fully formatted table as a string with linebreaks.

Parameters **width** (*Optional[int]*) – Width to fit the table.

Returns Formatted table.

Return type str

echo()
Prints the table to the terminal.

class maestral.utils.cli.**Grid**(*fields=()*, *padding=2*, *align=Align.Left*)
Bases: object

A grid of fields which can be printed to stdout.

Parameters

- **fields** (*List[maestral.utils.cli.Field]*) – A sequence of fields (strings, datetimes, any objects with a string representation).
- **padding** (*int*) – Padding between fields.
- **align** (maestral.utils.cli.Align) – Alignment of strings in the grid.

fields: List[*maestral.utils.cli.Field*]

append(*field*)
Appends a field to the grid.

Parameters **field** (*Any*) –

Return type None

format_lines(*width=None*)
Iterator over formatted lines of the grid.

Parameters **width** (*Optional[int]*) – Width to fit the grid.

Returns Iterator over lines which can be printed to the terminal.

Return type Iterator[str]

format(*width=None*)
Returns a fully formatted grid as a string with linebreaks.

Parameters **width** (*Optional[int]*) – Width to fit the table.

Returns Formatted grid.

Return type str

echo()
Prints the grid to the terminal.

maestral.utils.cli.**echo**(*message*, *nl=True*, *prefix=Prefix.NONE*)

Parameters

- **message** (*str*) –
- **nl** (*bool*) –
- **prefix** (maestral.utils.cli.Prefix) –

Return type None

maestral.utils.cli.**info**(*message*, *nl=True*)

Parameters

- **message** (*str*) –
- **nl** (*bool*) –

Return type None

maestral.utils.cli.**warn**(*message*, *nl=True*)

Parameters

- **message** (*str*) –
- **nl** (*bool*) –

Return type None

maestral.utils.cli.**ok**(*message*, *nl=True*)

Parameters

- **message** (*str*) –
- **nl** (*bool*) –

Return type None

maestral.utils.cli.**prompt**(*message*, *default=None*, *validate=None*)

Parameters

- **message** (*str*) –
- **default** (*Optional[str]*) –
- **validate** (*Optional[Callable]*) –

Return type str

maestral.utils.cli.**confirm**(*message*, *default=True*)

Parameters

- **message** (*str*) –

- **default** (*Optional[bool]*) –

Return type bool

maestral.utils.cli.**select**(*message*, *options*, *hint=''*)

Parameters

- **message** (*str*) –
- **options** (*Sequence[str]*) –

Return type int

maestral.utils.cli.**select_multiple**(*message*, *options*, *hint=''*)

Parameters

- **message** (*str*) –
- **options** (*Sequence[str]*) –

Return type List[int]

maestral.utils.cli.**select_path**(*message*, *default=None*, *validate=<function <lambda>>*, *exists=False*, *files_allowed=True*, *dirs_allowed=True*)

Parameters

- **message** (*str*) –
- **default** (*Optional[str]*) –
- **validate** (*Callable*) –
- **exists** (*bool*) –
- **files_allowed** (*bool*) –
- **dirs_allowed** (*bool*) –

Return type str

exception maestral.utils.cli.**CliException**(*message*)

Bases: click.exceptions.ClickException

Parameters **message** (*str*) –

Return type None

show(*file=None*)

Return type None

20.1.4 maestral.utils.content_hasher

Module for content hashing.

class maestral.utils.content_hasher.**DropboxContentHasher**

Bases: object

Computes a hash using the same algorithm that the Dropbox API uses for the the "content_hash" metadata field.

The *digest()* method returns a raw binary representation of the hash. The *hexdigest()* convenience method returns a hexadecimal-encoded version, which is what the "content_hash" metadata field uses.

This class has the same interface as the hashers in the standard 'hashlib' package.

Example Read a file in chunks of 1024 bytes and compute its content hash:

```
>>> hasher = DropboxContentHasher()
>>> with open('some-file', 'rb') as f:
...     while True:
...         chunk = f.read(1024)
...         if len(chunk) == 0:
...             break
...         hasher.update(chunk)
... print(hasher.hexdigest())
```

BLOCK_SIZE = 4194304

update(*new_data*)

digest()

hexdigest()

copy()

class maestral.utils.content_hasher.**StreamHasher**(*f*, *hasher*)

Bases: object

A wrapper around a file-like object (either for reading or writing) that hashes everything that passes through it. Can be used with *DropboxContentHasher* or any 'hashlib' hasher.

Example

```
>>> hasher = DropboxContentHasher()
>>> with open('some-file', 'rb') as f:
...     wrapped_f = StreamHasher(f, hasher)
...     response = some_api_client.upload(wrapped_f)
>>> locally_computed = hasher.hexdigest()
>>> assert response.content_hash == locally_computed
```

close()

flush()

fileno()

tell()

read(**args*)

write(*b*)

next()

readline(**args*)

readlines(**args*)

20.1.5 maestral.utils.integration

This module provides functions for platform integration. Most of the functionality here could also be achieved with psutils but we want to avoid the large dependency.

maestral.utils.integration.**get_ac_state**()
Checks if the current device has AC power or is running on battery.

Returns `True` if the device has AC power, `False` otherwise.

Return type *maestral.utils.integration.ACState*

class maestral.utils.integration.**ACState**(*value*)
Bases: `enum.Enum`

Enumeration of AC power states

Connected = 'Connected'

Disconnected = 'Disconnected'

Undetermined = 'Undetermined'

maestral.utils.integration.**get_inotify_limits**()
Returns the current inotify limit settings as tuple.

Returns `(max_user_watches, max_user_instances, max_queued_events)`

Raises `OSError` – if the settings cannot be read from /proc/sys/fs/inotify. This may happen if /proc/sys is left out of the kernel image or simply not mounted.

Return type Tuple[int, int, int]

maestral.utils.integration.**cpu_usage_percent**(*interval=0.1*)
Returns a float representing the CPU utilization of the current process as a percentage. This duplicates the similar method from psutil to avoid the psutil dependency.

Compares process times to system CPU times elapsed before and after the interval (blocking). It is recommended for accuracy that this function be called with an interval of at least 0.1 sec.

A value > 100.0 can be returned in case of processes running multiple threads on different CPU cores. The returned value is explicitly NOT split evenly between all available logical CPUs. This means that a busy loop process running on a system with 2 logical CPUs will be reported as having 100% CPU utilization instead of 50%.

Parameters **interval** (*float*) – Interval in sec between comparisons of CPU times.

Returns CPU usage during interval in percent.

Return type float

maestral.utils.integration.**check_connection**(*hostname*, *timeout=2*)
A low latency check for an internet connection.

Parameters

- **hostname** (*str*) – Hostname to use for connection check.
- **timeout** (*int*) – Timeout in seconds for connection check.

Returns Connection availability.

Return type bool

20.1.6 maestral.utils.orm

A basic object relational mapper for SQLite.

This is a very simple ORM implementation which contains only functionality needed by Maestral. Many operations will still require explicit SQL statements. This module is no alternative to fully featured ORMs such as sqlalchemy but may be useful when system memory is constrained.

class maestral.utils.orm.**SqlType**
Bases: object

Base class to represent Python types in SQLite table

sql_type = 'TEXT'

py_type
alias of str

static sql_to_py(*value*)
Converts the return value from sqlite3 to the target Python type.

static py_to_sql(*value*)
Converts a Python value to a type accepted by sqlite3.

class maestral.utils.orm.**SqlString**
Bases: *maestral.utils.orm.SqlType*

Class to represent Python strings in SQLite table

sql_type = 'TEXT'

py_type
alias of str

class maestral.utils.orm.**SqlInt**
Bases: *maestral.utils.orm.SqlType*

Class to represent Python integers in SQLite table

sql_type = 'INTEGER'

py_type
alias of int

class maestral.utils.orm.**SqlFloat**
Bases: *maestral.utils.orm.SqlType*

Class to represent Python floats in SQLite table

sql_type = 'REAL'

py_type
alias of float

class maestral.utils.orm.**SqlPath**
Bases: *maestral.utils.orm.SqlType*

Class to represent Python paths in SQLite table

This class contains special handling for strings with surrogate escape characters which can appear in badly encoded file names.

sql_type = 'TEXT'

py_type
alias of str

class maestral.utils.orm.**SqlEnum**(*enum*)
Bases: *maestral.utils.orm.SqlType*

Class to represent Python enums in SQLite table

Parameters **enum** (*Iterable[enum.Enum]*) –

Return type None

sql_type = 'TEXT'

py_type
alias of enum.Enum

sql_to_py(*value*)

Parameters **value** (*Optional[str]*) –

Return type Optional[enum.Enum]

static py_to_sql(*value*)

Parameters **value** (*Optional[enum.Enum]*) –

Return type Optional[str]

class maestral.utils.orm.**Column**(*type*, *nullable=True*, *unique=False*, *primary_key=False*, *default=None*)
Bases: property

Represents a column in a database table.

Parameters

- **type** (maestral.utils.orm.SqlType) – Column type in database table. Python types which don't have SQLite equivalents, such as enum.Enum, will be converted appropriately.
- **nullable** (*bool*) – When set to False, will cause the "NOT NULL" phrase to be added when generating the column.
- **unique** (*bool*) – If True, sets a unique constraint on the column.
- **primary_key** (*bool*) – If True, marks this column as a primary key column. Currently, only a single primary key column is supported.
- **default** (*Union[str, int, float, enum.Enum, None, Type[NoDefault]]*) – Default value for the column. Set to *NoDefault* if no default value should be used. Note than None / NULL is a valid default for an SQLite column.

render_constraints()
Returns a string with constraints for the SQLite column definition.

Return type str

render_properties()
Returns a string with properties for the SQLite column definition.

Return type str

render_column()
Returns a string with the full SQLite column definition.

Return type str

py_to_sql(*value*)
Converts a Python value to a value which can be stored in the database column.

Parameters **value** (*Optional[Union[str, int, float, enum.Enum]]*) – Native Python value.

Returns Converted Python value to store in database. Will only return str, int, float or None.

Return type Optional[Union[str, int, float]]

sql_to_py(*value*)
Converts a database column value to the original Python type.

Parameters **value** (*Optional[Union[str, int, float]]*) – Value from database column. Only accepts str, int, float or None.

Returns Converted Python value.

Return type Optional[Union[str, int, float, enum.Enum]]

class maestral.utils.orm.**NoDefault**
Bases: object

Class to denote the absence of a default value.

This is distinct from None which may be a valid default.

class maestral.utils.orm.**Database**(**args*, ***kwargs*)
Bases: object

Proxy class to access sqlite3.connect method.

Return type None

property connection: sqlite3.Connection
Returns an existing SQL connection or creates a new one.

close()
Closes the SQL connection.

Return type None

commit()
Commits SQL changes.

Return type None

execute(*sql*, **args*)
Creates a cursor and executes the given SQL statement.

Parameters

- **sql** (*str*) – SQL statement to execute.
- **args** – Parameters to substitute for placeholders in SQL statement.

Returns The created cursor.

Return type sqlite3.Cursor

executescript(*script*)
Creates a cursor and executes the given SQL script.

Parameters **script** (*str*) – SQL script to execute.

Returns The created cursor.

Return type None

class maestral.utils.orm.**Manager**(*db*, *model*)

Bases: object

A data mapper interface for a table model.

Creates the table as defined in the model if it doesn't already exist. Keeps a cache of weak references to all retrieved and created rows to speed up queries. The cache should be cleared manually changes where made to the table from outside this manager.

Parameters

- **db** (maestral.utils.orm.Database) – Database to use.
- **model** (*Type[Model]*) – Model for database table.

Return type None

create_table()

Creates the table as defined by the model.

Return type None

clear_cache()

Clears our cache.

Return type None

get_primary_key(*obj*)

Returns the primary key for a model object / row in the table.

Parameters **obj** (maestral.utils.orm.Model) – Model instance which represents the row.

Returns Primary key for row.

Return type Optional[Union[str, int, float]]

all()

Get all model objects / rows from database in a single query.

Returns List of model objects.

Return type List[*maestral.utils.orm.Model*]

iter_all(*size=1000*)

Get all model objects / rows from database in multiple queries.

Parameters **size** (*int*) – Number of rows to fetch in each query.

Returns Iterator over lists of model objects.

Return type Generator[List[*maestral.utils.orm.Model*], Any, None]

create(***kwargs*)

Create a model object from SQL column values

Parameters **kwargs** – Column values.

Returns Model object.

Return type *maestral.utils.orm.Model*

delete(*obj*)

Delete a model object / row from database

Parameters **obj** (maestral.utils.orm.Model) – Object / row to delete.

Return type None

get(*primary_key*)
Gets a model object from database by its primary key. This will return a cached value if available and None if no row with the primary key exists.

Parameters **primary_key** (*Optional[Union[str, int, float, enum.Enum]]*) – Primary key for row.

Returns Model object representing the row.

Return type Optional[*maestral.utils.orm.Model*]

has(*primary_key*)
Checks if a model object exists in database by its primary key

Parameters **primary_key** (*Optional[Union[str, int, float, enum.Enum]]*) – The primary key.

Returns Whether the corresponding row exists in the table.

Return type bool

save(*obj*)
Saves a model object to the database table. If the primary key is None, a new primary key will be generated by SQLite on inserting the row. This key will be retrieved and stored in the primary key property of the object.

Parameters **obj** (maestral.utils.orm.Model) – Model object to save.

Returns Saved model object.

Return type *maestral.utils.orm.Model*

update(*obj*)
Updates the database table from a model object.

Parameters **obj** (maestral.utils.orm.Model) – The object to update.

Return type None

query_to_objects(*sql*, **args*)
Performs the given SQL query and converts any returned rows to model objects.

Parameters

- **sql** (*str*) – SQL statement to execute.
- **args** – Parameters to substitute for placeholders in SQL statement.

Returns List of model objects from the query.

Return type List[*maestral.utils.orm.Model*]

count()
Returns the number of rows in the table.

Return type int

class maestral.utils.orm.**Model**(***kwargs*)
Bases: object

Abstract object model to represent an SQL table.

Instances of this class are model objects which correspond to rows in the database table.

To define a table, subclass this Model and define Column`s as class properties. Override the ``__tablename__` attribute with the actual table name.

Initialise with keyword arguments corresponding to column names and values.

Parameters **kwargs** – Keyword arguments assigning values to table columns.

Return type None

20.1.7 maestral.utils.path

This module contains functions for common path operations.

maestral.utils.path.**normalize_case**(*string*)
Converts a string to lower case. Todo: Follow Python 2.5 / Dropbox conventions.

Parameters **string** (*str*) – Original string.

Returns Lowercase string.

Return type str

maestral.utils.path.**normalize_unicode**(*string*)
Normalizes a string to replace all decomposed unicode characters with their single character equivalents.

Parameters **string** (*str*) – Original string.

Returns Normalized string.

Return type str

maestral.utils.path.**normalize**(*string*)
Replicates the path normalization performed by Dropbox servers. This typically only involves converting the path to lower case, with a few (undocumented) exceptions:

- Unicode normalization: decomposed characters are converted to composed characters.
- Lower casing of non-ascii characters: Dropbox uses the Python 2.5 behavior for conversion to lower case. This means that some cyrillic characters are incorrectly lower-cased. For example: “”.lower() -> “” instead of “” “”.lower() -> “” instead of “”
- Trailing spaces are stripped from folder names. We do not perform this normalization here because the Dropbox API will raise sync errors for such names anyways.

Note that calling *normalize()* on an already normalized path will return the unmodified input.

Todo: Follow Python 2.5 / Dropbox conventions instead of Python 3 conventions.

Parameters **string** (*str*) – Original path.

Returns Normalized path.

Return type str

maestral.utils.path.**is_fs_case_sensitive**(*path*)
Checks if `path` lies on a partition with a case-sensitive file system.

Parameters **path** (*str*) – Path to check.

Returns Whether `path` lies on a partition with a case-sensitive file system.

Return type bool

maestral.utils.path.**is_child**(*path*, *parent*)
Checks if `path` semantically is inside `parent`. Neither path needs to refer to an actual item on the drive. This function is case sensitive.

Parameters

- **path** (*str*) – Item path.
- **parent** (*str*) – Parent path.

Returns Whether `path` semantically lies inside `parent`.

Return type bool

maestral.utils.path.**is_equal_or_child**(*path*, *parent*)

Checks if `path` semantically is inside `parent` or equals `parent`. Neither path needs to refer to an actual item on the drive. This function is case sensitive.

Parameters

- **path** (*str*) – Item path.
- **parent** (*str*) – Parent path.

Returns `True` if `path` semantically lies inside `parent` or `path == parent`.

Return type bool

maestral.utils.path.**equivalent_path_candidates**(*path*, *root='/'*, *norm_func=<function normalize>*)

Given a "normalized" path using an injective (one-directional) normalization function, this method returns a list of matching un-normalized local paths.

If no such local path exists, the normalized path itself is returned. If a local path can be followed up to a certain parent in the hierarchy, it will be taked and the remaining normalized path will be appended.

Example Assume the normalization function is `str.lower()`. If a root directory contains two folders "/parent/subfolder/child" and "/parent/Subfolder/child", two matches will be returned for "path = /parent/subfolder/child/file.txt".

Parameters

- **path** (*str*) – Normalized path relative to `root`.
- **root** (*str*) – Parent directory to search in. There are significant performance improvements if a root directory with a small tree is given.
- **norm_func** (*Callable*) – Normalization function to use. Defaults to *normalize()*.

Returns Candidates for correctly cased local paths.

Return type List[str]

maestral.utils.path.**denormalize_path**(*path*, *root='/'*)

Returns a denormalized version of the given path as far as corresponding nodes with the same normalization exist in the given root directory. If multiple matches are found, only one is returned. If `path` does not exist in root `root` or `root` does not exist, the return value will be `os.path.join(root, path)`.

Parameters

- **path** (*str*) – Original path relative to `root`.
- **root** (*str*) – Parent directory to search in. There are significant performance improvements if a root directory with a small tree is given.

Returns Absolute and cased version of given path.

Return type str

maestral.utils.path.**to_existing_unnormalized_path**(*path*, *root='/'*)

Returns a cased version of the given path if corresponding nodes (with arbitrary casing) exist in the given root directory. If multiple matches are found, only one is returned.

Parameters

- **path** (*str*) – Original path relative to `root`.
- **root** (*str*) – Parent directory to search in. There are significant performance improvements if a root directory with a small tree is given.

Returns Absolute and cased version of given path.

Raises `FileNotFoundError` – if `path` does not exist in root `root` or `root` itself does not exist.

Return type str

`maestral.utils.path.`**`normalized_path_exists`**(*path*, *root='/'*)

Checks if a `path` exists in given `root` directory, similar to `os.path.exists` but case-insensitive. Normalisation is performed as by Dropbox servers (lower case and unicode normalisation).

Parameters

- **path** (*str*) – Path relative to `root`.
- **root** (*str*) – Directory where we will look for `path`. There are significant performance improvements if a root directory with a small tree is given.

Returns Whether an arbitrarily cased version of `path` exists.

Return type bool

`maestral.utils.path.`**`generate_cc_name`**(*path*, *suffix='conflicting copy'*)

Generates a path for a conflicting copy of `path`. The file name is created by inserting the given `suffix` between the the filename and extension. For instance:

"my_file.txt" -> "my_file (conflicting copy).txt"

If a file with the resulting path already exists (case-insensitive!), we additionally append an integer number, for instance:

"my_file.txt" -> "my_file (conflicting copy 1).txt"

Parameters

- **path** (*str*) – Original path name.
- **suffix** (*str*) – Suffix to use. Defaults to "conflicting copy".

Returns New path.

Return type str

`maestral.utils.path.`**`delete`**(*path*, *raise_error=False*)

Deletes a file or folder at `path`.

Parameters

- **path** (*str*) – Path of item to delete.
- **raise_error** (*bool*) – Whether to raise errors or return them.

Returns Any caught exception during the deletion.

Return type Optional[OSError]

`maestral.utils.path.`**`move`**(*src_path*, *dest_path*, *raise_error=False*, *preserve_dest_permissions=False*)

Moves a file or folder from `src_path` to `dest_path`. If either the source or the destination path no longer exist, this function does nothing. Any other exceptions are either raised or returned if `raise_error` is False.

Parameters

- **src_path** (*str*) – Path of item to move.

- **dest_path** (*str*) – Destination path. Any existing file at this path will be replaced by the move. Any existing **empty** folder will be replaced if the source is also a folder.
- **raise_error** (*bool*) – Whether to raise errors or return them.
- **preserve_dest_permissions** – Whether to apply the permissions of the source path to the destination path. Permissions will not be set recursively.

Returns Any caught exception during the move.

Return type Optional[OSError]

maestral.utils.path.**walk**(*root*, *listdir=<built-in function scandir>*)

Iterates recursively over the content of a folder.

Parameters

- **root** (*Union[str, os.PathLike]*) – Root folder to walk.
- **listdir** (*Callable[[Union[str, os.PathLike]], Iterable[posix.DirEntry]]*) – Function to call to get the folder content.

Returns Iterator over (path, stat) results.

Return type Iterator[Tuple[str, os.stat_result]]

maestral.utils.path.**content_hash**(*local_path*, *chunk_size=1024*)

Computes content hash of a local file.

Parameters

- **local_path** (*str*) – Absolute path on local drive.
- **chunk_size** (*int*) – Size of chunks to hash in bites.

Returns Content hash to compare with Dropbox's content hash and mtime just before the hash was computed.

Return type Tuple[Optional[str], Optional[float]]

20.1.8 maestral.utils.serializer

This module contains functions to serialize class instances for communication between the daemon and frontends.

maestral.utils.serializer.**dropbox_stone_to_dict**(*obj*)

Converts the result of a Dropbox SDK call to a dictionary.

Parameters **obj** (*StoneStruct*) –

Return type Dict[str, Optional[Union[str, float, bool]]]

maestral.utils.serializer.**error_to_dict**(*err*)

Converts an exception to a dict. Keys will be strings and entries are native Python types.

Parameters **err** (*Exception*) – Exception to convert.

Returns Dictionary where all keys and values are strings. The following keys will always be present but may contain emtpy strings: 'type', 'inherits', 'title', 'traceback', 'title', and 'message'.

Return type Dict[str, Optional[Union[str, Sequence[str]]]]

maestral.utils.serializer.**sync_event_to_dict**(*event*)

Converts a SyncEvent to a dict. Keys will be strings and entries are native Python types.

Parameters **event** (maestral.database.SyncEvent) – SyncEvent to convert.

Returns Serialized SyncEvent.

Return type Dict[str, Optional[Union[str, float, bool]]]

20.2 Module contents

Utility modules and functions

maestral.utils.**natural_size**(*num*, *unit='B'*, *sep=True*)
Convert number to a human readable string with decimal prefix.

Parameters

- **num** (*float*) – Value in given unit.
- **unit** (*str*) – Unit suffix.
- **sep** (*bool*) – Whether to separate unit and value with a space.

Returns Human readable string with decimal prefixes.

Return type str

maestral.utils.**chunks**(*lst*, *n*, *consume=False*)
Partitions an iterable into chunks of length `n`.

Parameters

- **lst** (*list*) – Iterable to partition.
- **n** (*int*) – Chunk size.
- **consume** (*bool*) – If True, the list will be consumed (emptied) during the iteration. This can be used to free memory in case of large lists.

Returns Iterator over chunks.

Return type Iterator[list]

maestral.utils.**clamp**(*n*, *minn*, *maxn*)
Clamps a number between a minimum and maximum value.

Parameters

- **n** (*maestral.utils._N*) – Original value.
- **minn** (*maestral.utils._N*) – Minimum allowed value.
- **maxn** (*maestral.utils._N*) – Maximum allowed value.

Returns Clamped value.

Return type maestral.utils._N

maestral.utils.**get_newer_version**(*version*, *releases*)
Checks a given release version against a version list of releases to see if an update is available. Only offers newer versions if they are not a prerelease.

Parameters

- **version** (*str*) – The current version.
- **releases** (*Iterable[str]*) – A list of valid cleaned releases.

Returns The version string of the latest release if a newer release is available.

Return type Optional[str]

maestral.utils.**removeprefix**(*string*, *prefix*)

Removes the given prefix from a string. Only the first instance of the prefix is removed. The original string is returned if it does not start with the given prefix.

This follows the Python 3.9 implementation of `str.removeprefix`.

Parameters

- **string** (*str*) – Original string.
- **prefix** (*str*) – Prefix to remove.

Returns String without prefix.

Return type str

maestral.utils.**sanitize_string**(*string*)

Converts a string provided by file system APIs, which may contain surrogate escapes for bytes with unknown encoding, to a string which can always be displayed or printed. This is done by replacing invalid characters with "".

Parameters **string** (*str*) – Original string.

Returns Sanitised path where all surrogate escapes have been replaced with "".

Return type str

maestral.utils.**exc_info_tuple**(*exc*)

Creates an exc-info tuple from an exception.

Parameters **exc** (*BaseException*) –

Return type Tuple[Type[BaseException], BaseException, Optional[types.TracebackType]]

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()
```

PYTHON MODULE INDEX

m

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X