pyspotify

Release 2.1.4

unknown

Aug 09, 2022

CONTENTS

CHAPTER

ONE

WARNING: THIS LIBRARY NO LONGER WORKS

pyspotify is a Python wrapper around the libspotify C library, and thus depends on libspotify for everything it does.

In May 2015, libspotify was deprecated by Spotify and active maintenance stopped. At this point, libspotify had been the main way to integrate with Spotify for six years, and was part of numerous open source projects and commercial applications, including many receivers and even cars. It remained the only API for playback outside Android and iOS.

In February 2016, server side changes to the Spotify API caused the search functionality to stop working, without Spotify ever acknowledging it. Users of pyspotify could work around this by using the Spotify web API for searches and pyspotify for playback.

In April 2022, Spotify announced that they would sunset the libspotify API one month later.

In May 2022, new libspotify connections to Spotify started failing. With libspotify dead, pyspotify was dead too.

After two years in development from May 2013 to May 2015, and seven years of loyal service this project has reached its end.

There will be no further updates to pyspotify.

Hopefully, the pyspotify source code can still serve as a complete example of how to successfully wrap a large C library in Python using CFFI.

CHAPTER

TWO

INTRODUCTION

pyspotify provides a Python interface to Spotify's online music streaming service.

With pyspotify you can access music metadata, search in Spotify's library of 20+ million tracks, manage your Spotify playlists, and play music from Spotify. All from your own Python applications.

pyspotify uses CFFI to make a pure Python wrapper around the official libspotify library. It works on CPython 2.7 and 3.5+, as well as PyPy 2.7 and 3.5+. It is known to work on Linux and macOS. Windows support should be possible, but is awaiting a contributor with the interest and knowledge to maintain it.

CHAPTER

THREE

PROJECT RESOURCES

- Documentation
- Source code
- Issue tracker

CHAPTER

FOUR

USER'S GUIDE

4.1 Installation

pyspotify is packaged for various operating systems and in multiple Linux distributions. What way to install pyspotify is best for you depends upon your OS and/or distribution.

4.1.1 Debian/Ubuntu: Install from apt.mopidy.com

The Mopidy project runs its own APT archive which includes pyspotify built for:

- Debian 9 (Stretch), which also works for Ubuntu 18.04 LTS.
- Debian 10 (Buster), which also works for Ubuntu 19.10 and newer.

The packages are available for multiple CPU architectures: i386, amd64, armel, and armhf (compatible with Raspbian and all Raspberry Pi models).

To install and receive future updates:

1. Add the archive's GPG key:

```
wget -q -O - https://apt.mopidy.com/mopidy.gpg | sudo apt-key add -
```

2. If you run Debian stretch or Ubuntu 16.04 LTS:

```
sudo wget -q -O /etc/apt/sources.list.d/mopidy.list https://apt.mopidy.com/stretch.
↪list
```

   Or, if you run any newer Debian/Ubuntu distro:

```
sudo wget -q -O /etc/apt/sources.list.d/mopidy.list https://apt.mopidy.com/buster.
↪list
```

3. Install pyspotify and all dependencies:

```
sudo apt-get update
sudo apt-get install python-spotify
```

4.1.2 Arch Linux: Install from AUR

If you are running Arch Linux on x86 or x86_64, you can install pyspotify using the python2-pyspotify package found in AUR.

1. To install pyspotify with all dependencies, run:

```
yay -S python2-pyspotify
```

Note: AUR does not provide libspotify for all CPU architectures e.g. arm. See *installing from source* in these cases.

4.1.3 macOS: Install wheel package from PyPI with pip

From PyPI, you can install precompiled wheel packages of pyspotify that bundle libspotify. The packages should work on all combinations of:

- macOS 10.6 and newer
- 32-bit and 64-bit
- Apple-Python, Python.org-Python, Homebrew-Python

1. Make sure you have a recent version of pip, which will default to installing a wheel package if available:

```
pip install --upgrade pip
```

2. Install pyspotify:

```
pip install pyspotify
```

4.1.4 macOS: Install from Homebrew

The Mopidy project maintains its own Homebrew tap which includes pyspotify and its dependencies.

1. Install Homebrew.
2. Make sure your installation is up to date:

```
brew update
brew upgrade --all
```

3. Install pyspotify from the mopidy/mopidy tap:

```
brew install mopidy/mopidy/pyspotify
```

4.1.5 Install from source

If you are on Linux, but your distro don't package pyspotify, you can install pyspotify from PyPI using the pip installer. However, since pyspotify is a Python wrapper around the libspotify library, pyspotify necessarily depends on libspotify and it must be installed first.

libspotify

libspotify is provided as a binary download for a selection of operating systems and CPU architectures from our unofficial libspotify archive. If libspotify isn't available for your OS or architecture, then you're out of luck and can't use pyspotify either.

To install libspotify, use one of the options below, or follow the instructions in the README file of the libspotify tarball.

Debian/Ubuntu

If you're running a Debian-based Linux distribution, like Ubuntu, you can get Debian packages of libspotify from apt.mopidy.com. Follow the instructions *above* to make the apt.mopidy.com archive available on your system, then install libspotify:

```
sudo apt-get install libspotify-dev
```

Arch Linux

libspotify for x86 and x86_64 is packaged in AUR. To install libspotify, run:

```
yay -S libspotify
```

Note: AUR only provides libspotify binaries for x86 and x86_64 CPUs. If you require libspotify for a different CPU architecture you'll need to download it from our unofficial libspotify archive instead.

macOS

If you're using Homebrew, it has a formula for libspotify in the mopidy/mopidy tap:

```
brew install mopidy/mopidy/libspotify
```

Build tools

To build pyspotify, you need a C compiler, Python development headers, and libffi development headers. All of this is easily installed using your system's package manager.

Debian/Ubuntu

If you're on a Debian-based system, you can install the pyspotify build dependencies by running:

```
sudo apt install build-essential python-dev python3-dev libffi-dev
```

Arch Linux

If you're on Arch Linux, you can install the pyspotify build dependencies by running:

```
sudo pacman -S base-devel python2 python
```

macOS

If you're on macOS, you'll need to install the Xcode command line developer tools. Even if you've already installed Xcode from the App Store, e.g. to get Homebrew working, you should run this command:

```
xcode-select --install
```

Note: If you get an error about `ffi.h` not being found when installing the cffi Python package, try running the above command.

---

pyspotify

With libspotify and the build tools in place, you can finally build pyspotify.

To download and build pyspotify from PyPI, run:

```
pip install pyspotify
```

Or, if you have a checkout of the pyspotify git repo, run:

```
pip install -e path/to/my/pyspotify/git/clone
```

Once you have pyspotify installed, you should head over to *Quickstart* for a short introduction to pyspotify.

4.2 Quickstart

This guide will quickly introduce you to some of the core features of pyspotify. It assumes that you've already installed pyspotify. If you do not, check out *Installation*. For a complete reference of what pyspotify provides, refer to the *API reference*.

4.2.1 Application keys

Every app that use libspotify needs its own libspotify application key. Application keys can be obtained automatically and free of charge from Spotify.

1. Go to the Spotify developer pages and login using your Spotify account.
2. Find the libspotify application keys management page and request an application key for your application.
3. Once the key is issued, download the "binary" version. The "C code" version of the key will not work with pyspotify.
4. If you place the application key in the same directory as your application's main Python file, pyspotify will automatically find it and use it. If you want to keep the application key in another location, you'll need to set `application_key` in your session config or call `load_application_key_file()` to load the session key file correctly.

4.2.2 Creating a session

Once pyspotify is installed and the application key is in place, we can start writing some Python code. Almost everything in pyspotify requires a *Session*, so we'll start with creating a session with the default config:

```
>>> import spotify
>>> session = spotify.Session()
```

All config must be done before the session is created. Thus, if you need to change any config to something else than the default, you must create a *Config* object first, and then pass it to the session:

```
>>> import spotify
>>> config = spotify.Config()
>>> config.user_agent = 'My awesome Spotify client'
>>> config.tracefile = b'/tmp/libspotify-trace.log'
>>> session = spotify.Session(config)
```

4.2.3 Text encoding

libspotify encodes all text as UTF-8. pyspotify converts the UTF-8 bytestrings to Unicode strings before returning them to you, so you don't have to be worried about text encoding.

Similarly, pyspotify will convert any string you give it from Unicode to UTF-8 encoded bytestrings before passing them on to libspotify. The only exception is file system paths, like `tracefile` above, which is passed directly to libspotify. This is in case you have a file system which doesn't use UTF-8 encoding for file names.

4.2.4 Login and event processing

With a session we can do a few things, like creating objects from Spotify URIs:

```
>>> import spotify
>>> session = spotify.Session()
>>> album = session.get_album('spotify:album:0XHpO9qTpqJJQwa2zFxAAE')
>>> album
Album(u'spotify:album:0XHpO9qTpqJJQwa2zFxAAE')
>>> album.link
```

(continues on next page)

(continued from previous page)

```
Link(u'spotify:album:0XHpO9qTpqJJQwa2zFxAAE')
>>> album.link.uri
u'spotify:album:0XHpO9qTpqJJQwa2zFxAAE'
```

But that's mostly how far you get with a fresh session. To do more, you need to login to the Spotify service using a Spotify account with the Premium subscription.

Warning: pyspotify and all other libspotify applications required a Spotify Premium subscription.

The Free Spotify subscription, or the old Unlimited subscription, will not work with pyspotify or any other applications using libspotify.

```
>>> import spotify
>>> session = spotify.Session()
>>> session.login('alice', 's3cretpassword')
```

For alternative ways to login, refer to the `login()` documentation.

The `login()` method is asynchronous, so we must ask the session to `process_events()` until the login has succeeded or failed:

```
>>> session.connection.state
<ConnectionState.LOGGED_OUT: 0>
>>> session.process_events()
>>> session.connection.state
<ConnectionState.OFFLINE: 1>
>>> session.process_events()
>>> session.connection.state
<ConnectionState.LOGGED_IN: 1>
```

Note: The connection state is a representation of both your authentication state and your offline mode. If libspotify has cached your user object from a previous session, it may authenticate you without a connection to Spotify's servers. Thus, you may very well be logged in, but still offline.

The connection state in the above example goes from the `LOGGED_OUT` state, to `OFFLINE`, to `LOGGED_IN`. If libspotify hasn't cached any information about your Spotify user account, the connection state will probably go directly from `LOGGED_OUT` to `LOGGED_IN`. Your application should be prepared for this.

For more details, see the `session.connection.state` documentation.

We only called `process_events()` twice, which may not be enough to get to the `LOGGED_IN` connection state. A more robust solution is to call it repeatedly until the `CONNECTION_STATE_UPDATED` event is emitted on the *Session* object and `session.connection.state` is `LOGGED_IN`:

```
>>> import threading
>>> logged_in_event = threading.Event()
>>> def connection_state_listener(session):
...     if session.connection.state is spotify.ConnectionState.LOGGED_IN:
...         logged_in_event.set()
...
>>> session = spotify.Session()
```

(continues on next page)

(continued from previous page)

```
>>> session.on(
...     spotify.SessionEvent.CONNECTION_STATE_UPDATED,
...     connection_state_listener)
...
>>> session.login('alice', 's3cretpassword')
>>> session.connection.state
<ConnectionState.LOGGED_OUT: 0>
>>> while not logged_in_event.wait(0.1):
...     session.process_events()
...
>>> session.connection.state
<ConnectionState.LOGGED_IN: 1>
>>> session.user
User(u'spotify:user:alice')
```

This solution works properly, but is a bit tedious. pyspotify provides an *EventLoop* helper thread that can make the `process_events()` calls in the background. With it running, we can simplify the login process:

```
>>> import threading
>>> logged_in_event = threading.Event()
>>> def connection_state_listener(session):
...     if session.connection.state is spotify.ConnectionState.LOGGED_IN:
...         logged_in_event.set()
...
>>> session = spotify.Session()
>>> loop = spotify.EventLoop(session)
>>> loop.start()
>>> session.on(
...     spotify.SessionEvent.CONNECTION_STATE_UPDATED,
...     connection_state_listener)
...
>>> session.connection.state
<ConnectionState.LOGGED_OUT: 0>
>>> session.login('alice', 's3cretpassword')
>>> session.connection.state
<ConnectionState.OFFLINE: 4>
>>> logged_in_event.wait()
>>> session.connection.state
<ConnectionState.LOGGED_IN: 1>
>>> session.user
User(u'spotify:user:alice')
```

Note that when using *EventLoop*, your event listener functions are called from the *EventLoop* thread, and not from your main thread. You may need to add synchronization primitives to protect your application code from threading issues.

4.2.5 Logging

pyspotify uses Python's standard `logging` module for logging. All log records emitted by pyspotify are issued to the logger named `spotify`, or a sublogger of it.

Out of the box, pyspotify is set up with `logging.NullHandler` as the only log record handler. This is the recommended approach for logging in libraries, so that the application developer using the library will have full control over how the log records from the library will be exposed to the application's users. In other words, if you want to see the log records from pyspotify anywhere, you need to add a useful handler to the root logger or the logger named `spotify` to get any log output from pyspotify. The defaults provided by `logging.basicConfig()` is enough to get debug log statements out of pyspotify:

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

If your application is already using `logging`, and you want debug log output from your own application, but not from pyspotify, you can ignore debug log messages from pyspotify by increasing the threshold on the "spotify" logger to "info" level or higher:

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.getLogger('spotify').setLevel(logging.INFO)
```

For more details on how to use `logging`, please refer to the Python standard library documentation.

If we turn on logging, the login process is a bit more informative:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> import spotify
>>> session = spotify.Session()
>>> session.login('alice', 's3cretpassword')
DEBUG:spotify.session:Notify main thread
DEBUG:spotify.session:Log message from Spotify: 19:15:54.829 I [ap:1752] Connecting to␣
↪AP ap.spotify.com:4070
DEBUG:spotify.session:Log message from Spotify: 19:15:54.862 I [ap:1226] Connected to␣
↪AP: 78.31.12.11:4070
>>> session.process_events()
DEBUG:spotify.session:Notify main thread
DEBUG:spotify.session:Log message from Spotify: 19:17:27.972 E [session:926] Not all␣
↪tracks cached
INFO:spotify.session:Logged in
DEBUG:spotify.session:Credentials blob updated: 'NfFEO...'
DEBUG:spotify.session:Connection state updated
43
>>> session.user
User(u'spotify:user:alice')
```

4.2.6 Browsing metadata

When we're logged in, the objects we created from Spotify URIs becomes a lot more interesting:

```
>>> album = session.get_album('spotify:album:0XHpO9qTpqJJQwa2zFxAAE')
```

If the object isn't loaded, you can call `load()` to block until the object is loaded with data:

```
>>> album.is_loaded
False
>>> album.name is None
True
>>> album.load()
Album('spotify:album:0XHpO9qTpqJJQwa2zFxAAE')
>>> album.name
u'Reach For Glory'
>>> album.artist
Artist(u'spotify:artist:4kjWnaLfIRcLJ1Dy4Wr6tY')
>>> album.artist.load().name
u'Blackmill'
```

The *Album* object give you the most basic information about an album. For more metadata, you can call `browse()` to get an *AlbumBrowser*:

```
>>> browser = album.browse()
```

The browser also needs to load data, but once its loaded, most related objects are in place with data as well:

```
>>> browser.load()
AlbumBrowser(u'spotify:album:0XHpO9qTpqJJQwa2zFxAAE')
>>> browser.copyrights
[u'2011 Blackmill']
>>> browser.tracks
[Track(u'spotify:track:4FXj4ZKMO2dSkqiAhV7L8t'),
 Track(u'spotify:track:1sYClIlZZsL6dVMVTxCYRm'),
 Track(u'spotify:track:1uY4O332HuqLIcSSJlg4NX'),
 Track(u'spotify:track:58qbTrCRGyjF9tnjvHDqAD'),
 Track(u'spotify:track:3RZzg8yZs5HaRjQiDiBIsV'),
 Track(u'spotify:track:4jIzCryeLdBgE671gdQ6QD'),
 Track(u'spotify:track:4JNpKcFjVFYIzt1D95dmi0'),
 Track(u'spotify:track:7wAtUSgh6wN5ZmuPRRXHyL'),
 Track(u'spotify:track:7HYOVVLd5XnfY4yyV5Neke'),
 Track(u'spotify:track:2YfVXi6dTux0x8KkWeZdd3'),
 Track(u'spotify:track:6HPKugiH3p0pUJBNgUQoou')]
>>> [(t.index, t.name, t.duration // 1000) for t in browser.tracks]
[(1, u'Evil Beauty', 228),
 (2, u'City Lights', 299),
 (3, u'A Reach For Glory', 254),
 (4, u'Relentless', 194),
 (5, u'In The Night Of Wilderness', 327),
 (6, u"Journey's End", 296),
 (7, u'Oh Miah', 333),
 (8, u'Flesh and Bones', 276),
 (9, u'Sacred River', 266),
```

(continues on next page)

(continued from previous page)

```
 (10, u'Rain', 359),
 (11, u'As Time Goes By', 97)]
```

4.2.7 Downloading cover art

While we're at it, let's do something a bit more impressive; getting cover art:

```
>>> cover = album.cover(spotify.ImageSize.LARGE)
>>> cover.load()
Image(u'spotify:image:16eaba4959d5d97e8c0ca04289e0b1baaefae55f')
```

Currently, all covers are in JPEG format:

```
>>> cover.format
<ImageFormat.JPEG: 0>
```

The *Image* object gives access to the raw JPEG data:

```
>>> len(cover.data)
37204
>>> cover.data[:20]
'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x01\x00H\x00H\x00\x00'
```

For convenience, it also provides the same data encoded as a `data:` URI for easy embedding into HTML documents:

```
>>> len(cover.data_uri)
49631
>>> cover.data_uri[:60]
u''
```

If you're following along, you can try writing the image data out to files and inspect the result yourself:

```
>>> open('/tmp/cover.jpg', 'w+').write(cover.data)
>>> open('/tmp/cover.html', 'w+').write('<img src="%s">' % cover.data_uri)
```

4.2.8 Searching

Warning: The search API was broken at 2016-02-03 by a server-side change made by Spotify. The functionality was never restored.

Please use the Spotify Web API to perform searches.

If you don't have the URI to a Spotify object, another way to get started is to `search()`:

```
>>> search = session.search('massive attack')
>>> search.load()
Search(u'spotify:search:massive+attack')
```

A search returns lists of matching artists, albums, tracks, and playlists:

```
>>> (search.artist_total, search.album_total, search.track_total, track.playlist_total)
(5, 50, 564, 125)
>>> search.artists[0].load().name
u'Massive Attack'
>>> [a.load().name for a in search.artists[:3]]
[u'Massive Attack',
 u'Kwanzaa Posse feat. Massive Attack',
 u'Massive Attack Vs. Mad Professor']
```

Only the first 20 items in each list are returned by default:

```
>>> len(search.artists)
5
>>> len(search.tracks)
20
```

The *Search* object can help you with getting `more()` results from the same query:

```
>>> search2 = search.more().load()
>>> len(search2.artists)
0
>>> len(search2.tracks)
20
>>> search.track_offset
0
>>> search.tracks[0]
Track(u'spotify:track:67Hna13dNDkZvBpTXRIaOJ')
>>> search2.track_offset
20
>>> search2.tracks[0]
Track(u'spotify:track:3kKVqFF4pv4EXeQe428zl2')
```

You can also do searches where Spotify tries to figure out what you mean based on popularity, etc. instead of exact token matches:

```
>>> search = session.search('mas').load()
Search(u'spotify:search:mas')
>>> search.artists[0].load().name
u'X-Mas Allstars'

>>> search = session.search('mas', search_type=spotify.SearchType.SUGGEST).load()
Search(u'spotify:search:mas')
>>> search.artists[0].load().name
u'Massive Attack'
```

4.2.9 Playlist management

Warning: The playlists API was broken at 2018-05-24 by a server-side change made by Spotify. The functionality was never restored.

Please use the Spotify Web API to work with playlists.

Another way to find some music is to use your Spotify *Playlist*, which can be found in `playlist_container`:

```
>>> len(session.playlist_container)
53
>>> playlist = session.playlist_container[0]
>>> playlist.load()
Playlist(u'spotify:user:jodal:playlist:5hBcGwxKlnzNnSrREQ4aUe')
>>> playlist.name
u'The Glitch Mob - Love Death Immortality'
```

The *PlaylistContainer* object lets you add, remove, move and rename playlists as well as playlist folders. See the API docs for *PlaylistContainer* for more examples.

```
>>> del session.playlist_container[0]
>>> len(session.playlist_container)
52
>>> session.playlist_container.insert(0, playlist)
>>> len(session.playlist_container)
53
```

The *Playlist* objects let you add, remove and move tracks in a playlist, as well as turning on things like syncing of the playlist for offline playback:

```
>>> playlist.offline_status
<PlaylistOfflineStatus.NO: 0>
>>> playlist.set_offline_mode(True)
>>> playlist.offline_status
<PlaylistOfflineStatus.WAITING: 3>
>>> session.process_events()
# Probably needed multiple times, before syncing begins
>>> playlist.offline_status
<PlaylistOfflineStatus.DOWNLOADING: 2>
>>> playlist.offline_download_completed
20
# More process_events()
>>> playlist.offline_status
<PlaylistOfflineStatus.YES: 1>
```

For more details, see the API docs for *Playlist*.

4.2.10 Playing music

Music data is delivered to the `MUSIC_DELIVERY` event listener as PCM frames. If you want to have full control of audio playback, you can deliver these audio frames to your operating systems' audio subsystem yourself. If you want some help on the road, pyspotify comes with audio sinks for some select audio subsystems.

For details, have a look at the *spotify.AlsaSink* and *spotify.PortAudioSink* documentation, and the `examples/play_track.py` and `examples/shell.py` examples.

4.2.11 Thread safety

If you've read the libspotify documentation, you may have noticed that libspotify itself isn't thread safe. This means that you must take care to never call libspotify functions from two threads at the same time, and to finish your work with any pointers, e.g. strings, returned by libspotify functions before calling the next libspotify function. In summary, you'll need to use a single thread for all your use of libspotify, or protect all libspotify function calls with a single lock.

pyspotify, on the other hand, improves on this so that you can use pyspotify from multiple threads. pyspotify has a single global lock. This lock is acquired during all calls to libspotify, for as long as we're working with pointers returned from libspotify functions, and during all access to pyspotify's own internal state, like for example the collections of event listeners. In other words, pyspotify should be safe to use from multiple threads simultaneously.

Even though pyspotify itself is thread safe, you cannot disregard threading issues entirely when using pyspotify. There's two things to watch out for. First, event listeners for a number of the events listed in *SessionEvent* will be called from internal threads in libspotify itself. This is clearly marked in the documentation for the relevant events. Second, if you use the *EventLoop* helper thread, listeners for all other events—that is, events *not* emitted from internal threads in libspotify—will be called from the *EventLoop* thread. This shouldn't be an issue if you just use pyspotify itself from within the event listeners, but the moment you start working with your application's state from inside event listeners, you'll need to apply the proper thread synchronization primitives to avoid getting into trouble.

CHAPTER

FIVE

API REFERENCE

5.1 API reference

The pyspotify API follows the libspotify API closely. Thus, you can refer to the similarly named functions in the libspotify docs for further details.

spotify.**__version__**

pyspotify's version number in the **PEP 386** format.

```
>>> import spotify
>>> spotify.__version__
u'2.0.0'
```

spotify.**get_libspotify_api_version**()

Get the API compatibility level of the wrapped libspotify library.

```
>>> import spotify
>>> spotify.get_libspotify_api_version()
12
```

spotify.**get_libspotify_build_id**()

Get the build ID of the wrapped libspotify library.

```
>>> import spotify
>>> spotify.get_libspotify_build_id()
u'12.1.51.g86c92b43 Release Linux-x86_64 '
```

Sections

5.1.1 Error handling

exception spotify.**Error**

A Spotify error.

This is the superclass of all custom exceptions raised by pyspotify.

class spotify.**ErrorType**(*value*)

exception spotify.**LibError**(*error_type*)

A libspotify error.

Where many libspotify functions return error codes that must be checked after each and every function call, pyspotify raises the *LibError* exception instead. This helps you to not accidentally swallow and hide errors when using pyspotify.

exception spotify.**Timeout**(*timeout*)

Exception raised by an operation not completing within the given timeout.

5.1.2 Configuration

class spotify.**Config**

The session config.

Create an instance and assign to its attributes to configure. Then use the config object to create a session:

```
>>> config = spotify.Config()
>>> config.user_agent = 'My awesome Spotify client'
>>> # Etc ...
>>> session = spotify.Session(config=config)
```

5.1.3 Sessions

class spotify.**Session**(*config=None*)

The Spotify session.

If no `config` is provided, the default config is used.

The session object will emit a number of events. See *SessionEvent* for a list of all available events and how to connect your own listener functions up to get called when the events happens.

Warning: You can only have one *Session* instance per process. This is a libspotify limitation. If you create a second *Session* instance in the same process pyspotify will raise a `RuntimeError` with the message "Session has already been initialized".

Parameters **config** (*Config* or `None`) – the session config

class spotify.**SessionEvent**

Session events.

Using the *Session* object, you can register listener functions to be called when various session related events occurs. This class enumerates the available events and the arguments your listener functions will be called with.

Example usage:

```
import spotify

def logged_in(session, error_type):
    if error_type is spotify.ErrorType.OK:
        print('Logged in as %s' % session.user)
    else:
```

(continues on next page)

(continued from previous page)

```
        print('Login failed: %s' % error_type)

session = spotify.Session()
session.on(spotify.SessionEvent.LOGGED_IN, logged_in)
session.login('alice', 's3cret')
```

All events will cause debug log statements to be emitted, even if no listeners are registered. Thus, there is no need to register listener functions just to log that they're called.

5.1.4 Event loop

class spotify.**EventLoop**(*session*)

Event loop for automatically processing events from libspotify.

The event loop is a `Thread` that listens to `NOTIFY_MAIN_THREAD` events and calls `process_events()` when needed.

To use it, pass it your *Session* instance and call `start()`:

```
>>> session = spotify.Session()
>>> event_loop = spotify.EventLoop(session)
>>> event_loop.start()
```

The event loop thread is a daemon thread, so it will not stop your application from exiting. If you wish to stop the event loop without stopping your entire application, call `stop()`. You may call `join()` to block until the event loop thread has finished, just like for any other thread.

Warning: If you use *EventLoop* to process the libspotify events, any event listeners you've registered will be called from the event loop thread. pyspotify itself is thread safe, but you'll need to ensure that you have proper synchronization in your own application code, as always when working with threads.

5.1.5 Connection

class spotify.connection.**Connection**(*session*)

Connection controller.

You'll never need to create an instance of this class yourself. You'll find it ready to use as the `connection` attribute on the `Session` instance.

class spotify.**ConnectionRule**(*value*)

class spotify.**ConnectionState**(*value*)

class spotify.**ConnectionType**(*value*)

5.1.6 Offline sync

class spotify.offline.**Offline**(*session*)

Offline sync controller.

You'll never need to create an instance of this class yourself. You'll find it ready to use as the `offline` attribute on the `Session` instance.

class spotify.**OfflineSyncStatus**(*sp_offline_sync_status*)

A Spotify offline sync status object.

You'll never need to create an instance of this class yourself. You'll find it ready for use as the `sync_status` attribute on the `offline` attribute on the *Session* instance.

5.1.7 Links

class spotify.**Link**(*session*, *uri=None*, *sp_link=None*, *add_ref=True*)

A Spotify object link.

Call the `get_link()` method on your *Session* instance to get a *Link* object from a Spotify URI. You can also get links from the `link` attribute on most objects, e.g. `Track.link`.

To get the URI from the link object you can use the `uri` attribute, or simply use the link as a string:

```
>>> session = spotify.Session()
# ...
>>> link = session.get_link(
...     'spotify:track:2Foc5Q5nqNiosCNqttzHof')
>>> link
Link('spotify:track:2Foc5Q5nqNiosCNqttzHof')
>>> link.uri
'spotify:track:2Foc5Q5nqNiosCNqttzHof'
>>> str(link)
'spotify:track:2Foc5Q5nqNiosCNqttzHof'
>>> link.type
<LinkType.TRACK: 1>
>>> track = link.as_track()
>>> track.link
Link('spotify:track:2Foc5Q5nqNiosCNqttzHof')
>>> track.load().name
u'Get Lucky'
```

You can also get *Link* objects from open.spotify.com and play.spotify.com URLs:

```
>>> session.get_link(
...     'http://open.spotify.com/track/4wl1dK5dHGp3Ig51stvxb0')
Link('spotify:track:4wl1dK5dHGp3Ig51stvxb0')
>>> session.get_link(
...     'https://play.spotify.com/track/4wl1dK5dHGp3Ig51stvxb0'
...     '?play=true&utm_source=open.spotify.com&utm_medium=open')
Link('spotify:track:4wl1dK5dHGp3Ig51stvxb0')
```

class spotify.**LinkType**(*value*)

5.1.8 Users

class spotify.**User**(*session*, *uri=None*, *sp_user=None*, *add_ref=True*)

A Spotify user.

You can get users from the session, or you can create a *User* yourself from a Spotify URI:

```
>>> session = spotify.Session()
# ...
>>> user = session.get_user('spotify:user:jodal')
>>> user.load().display_name
u'jodal'
```

5.1.9 Tracks

class spotify.**Track**(*session*, *uri=None*, *sp_track=None*, *add_ref=True*)

A Spotify track.

You can get tracks from playlists or albums, or you can create a *Track* yourself from a Spotify URI:

```
>>> session = spotify.Session()
# ...
>>> track = session.get_track('spotify:track:2Foc5Q5nqNiosCNqttzHof')
>>> track.load().name
u'Get Lucky'
```

class spotify.**TrackAvailability**(*value*)

class spotify.**TrackOfflineStatus**(*value*)

5.1.10 Albums

class spotify.**Album**(*session*, *uri=None*, *sp_album=None*, *add_ref=True*)

A Spotify album.

You can get an album from a track or an artist, or you can create an *Album* yourself from a Spotify URI:

```
>>> session = spotify.Session()
# ...
>>> album = session.get_album('spotify:album:6wXDbHLesy6zWqQawAa91d')
>>> album.load().name
u'Forward / Return'
```

class spotify.**AlbumBrowser**(*session*, *album=None*, *callback=None*, *sp_albumbrowse=None*, *add_ref=True*)

An album browser for a Spotify album.

You can get an album browser from any *Album* instance by calling `Album.browse()`:

```
>>> session = spotify.Session()
# ...
>>> album = session.get_album('spotify:album:6wXDbHLesy6zWqQawAa91d')
>>> browser = album.browse()
```

(continues on next page)

(continued from previous page)

```
>>> browser.load()
>>> len(browser.tracks)
7
```

class spotify.**AlbumType**(*value*)

5.1.11 Artists

class spotify.**Artist**(*session*, *uri=None*, *sp_artist=None*, *add_ref=True*)

A Spotify artist.

You can get artists from tracks and albums, or you can create an *Artist* yourself from a Spotify URI:

```
>>> session = spotify.Session()
# ...
>>> artist = session.get_artist(
...     'spotify:artist:22xRIphSN7IkPVbErICu7s')
>>> artist.load().name
u'Rob Dougan'
```

class spotify.**ArtistBrowser**(*session*, *artist=None*, *type=None*, *callback=None*, *sp_artistbrowse=None*, *add_ref=True*)

An artist browser for a Spotify artist.

You can get an artist browser from any *Artist* instance by calling `Artist.browse()`:

```
>>> session = spotify.Session()
# ...
>>> artist = session.get_artist(
...     'spotify:artist:421vyBBkhgRAOz4cYPvrZJ')
>>> browser = artist.browse()
>>> browser.load()
>>> len(browser.albums)
7
```

class spotify.**ArtistBrowserType**(*value*)

5.1.12 Images

class spotify.**Image**(*session*, *uri=None*, *sp_image=None*, *add_ref=True*, *callback=None*)

A Spotify image.

You can get images from `Album.cover()`, `Artist.portrait()`, `Playlist.image()`, or you can create an *Image* yourself from a Spotify URI:

```
>>> session = spotify.Session()
# ...
>>> image = session.get_image(
...     'spotify:image:a0bdcbe11b5cd126968e519b5ed1050b0e8183d0')
>>> image.load().data_uri[:50]
u''
```

If `callback` isn't `None`, it is expected to be a callable that accepts a single argument, an *Image* instance, when the image is done loading.

class spotify.**ImageFormat**(*value*)

class spotify.**ImageSize**(*value*)

5.1.13 Search

Warning: The search API was broken at 2016-02-03 by a server-side change made by Spotify. The functionality was never restored.

Please use the Spotify Web API to perform searches.

class spotify.**Search**(*session*, *query=''*, *callback=None*, *track_offset=0*, *track_count=20*, *album_offset=0*, *album_count=20*, *artist_offset=0*, *artist_count=20*, *playlist_offset=0*, *playlist_count=20*, *search_type=None*, *sp_search=None*, *add_ref=True*)

A Spotify search result.

Call the `search()` method on your *Session* instance to do a search and get a *Search* back.

class spotify.**SearchPlaylist**(*session*, *name*, *uri*, *image_uri*)

A playlist matching a search query.

class spotify.**SearchType**(*value*)

5.1.14 Playlists

Warning: The playlists API was broken at 2018-05-24 by a server-side change made by Spotify. The functionality was never restored.

Please use the Spotify Web API to work with playlists.

class spotify.**Playlist**(*session*, *uri=None*, *sp_playlist=None*, *add_ref=True*)

A Spotify playlist.

You can get playlists from the `playlist_container`, `inbox`, `get_starred()`, `search()`, etc., or you can create a playlist yourself from a Spotify URI:

```
>>> session = spotify.Session()
# ...
>>> playlist = session.get_playlist(
...     'spotify:user:fiat500c:playlist:54k50VZdvtnIPt4d8RBCmZ')
>>> playlist.load().name
u'500C feelgood playlist'
```

class spotify.**PlaylistEvent**

Playlist events.

Using *Playlist* objects, you can register listener functions to be called when various events occurs in the playlist. This class enumerates the available events and the arguments your listener functions will be called with.

Example usage:

```
import spotify

def tracks_added(playlist, tracks, index):
    print('Tracks added to playlist')

session = spotify.Session()
# Login, etc...

playlist = session.playlist_container[0]
playlist.on(spotify.PlaylistEvent.TRACKS_ADDED, tracks_added)
```

All events will cause debug log statements to be emitted, even if no listeners are registered. Thus, there is no need to register listener functions just to log that they're called.

class spotify.**PlaylistContainer**(*session*, *sp_playlistcontainer*, *add_ref=True*)

A Spotify playlist container.

The playlist container can be accessed as a regular Python collection to work with the playlists:

```
>>> import spotify
>>> session = spotify.Session()
# Login, etc.
>>> container = session.playlist_container
>>> container.is_loaded
False
>>> container.load()
[Playlist(u'spotify:user:jodal:playlist:6xkJysqhkj9uwufFbUb8sP'),
 Playlist(u'spotify:user:jodal:playlist:0agJjPcOhHnstLIQunJHxo'),
 PlaylistFolder(id=8027491506140518932L, name=u'Shared playlists',
    type=<PlaylistType.START_FOLDER: 1>),
 Playlist(u'spotify:user:p3.no:playlist:7DkMndS2KNVQuf2fOpMt10'),
 PlaylistFolder(id=8027491506140518932L, name=u'',
    type=<PlaylistType.END_FOLDER: 2>)]
>>> container[0]
Playlist(u'spotify:user:jodal:playlist:6xkJysqhkj9uwufFbUb8sP')
```

As you can see, a playlist container can contain a mix of *Playlist* and *PlaylistFolder* objects.

The container supports operations that changes the container as well.

To add a playlist you can use `append()` or `insert()` with either the name of a new playlist or an existing playlist object. For example:

```
>>> playlist = session.get_playlist(
...     'spotify:user:fiat500c:playlist:54k50VZdvtnIPt4d8RBCmZ')
>>> container.insert(3, playlist)
>>> container.append('New empty playlist')
```

To remove a playlist or folder you can use `remove_playlist()`, or:

```
>>> del container[0]
```

To replace an existing playlist or folder with a new empty playlist with the given name you can use `remove_playlist()` and `add_new_playlist()`, or:

```
>>> container[0] = 'My other new empty playlist'
```

To replace an existing playlist or folder with an existing playlist you can :use `remove_playlist()` and `add_playlist()`, or:

```
>>> container[0] = playlist
```

class spotify.**PlaylistContainerEvent**

Playlist container events.

Using *PlaylistContainer* objects, you can register listener functions to be called when various events occurs in the playlist container. This class enumerates the available events and the arguments your listener functions will be called with.

Example usage:

```
import spotify

def container_loaded(playlist_container):
    print('Playlist container loaded')

session = spotify.Session()
# Login, etc...

session.playlist_container.on(
    spotify.PlaylistContainerEvent.CONTAINER_LOADED, container_loaded)
```

All events will cause debug log statements to be emitted, even if no listeners are registered. Thus, there is no need to register listener functions just to log that they're called.

class spotify.**PlaylistFolder**(*id*, *name*, *type*)

An object marking the start or end of a playlist folder.

id

An opaque ID that matches the ID of the *PlaylistFolder* object at the other end of the folder.

name

Name of the playlist folder. This is an empty string for the `END_FOLDER`.

type

The *PlaylistType* of the folder. Either `START_FOLDER` or `END_FOLDER`.

class spotify.**PlaylistOfflineStatus**(*value*)

class spotify.**PlaylistPlaceholder**

An object marking an unknown entry in the playlist container.

class spotify.**PlaylistTrack**(*session*, *sp_playlist*, *index*)

A playlist track with metadata specific to the playlist.

Use `tracks_with_metadata` to get a list of *PlaylistTrack*.

class spotify.**PlaylistType**(*value*)

class spotify.**PlaylistUnseenTracks**(*session*, *sp_playlistcontainer*, *sp_playlist*)

A list of unseen tracks in a playlist.

The list may contain items that are `None`.

Returned by `PlaylistContainer.get_unseen_tracks()`.

5.1.15 Toplists

class spotify.**Toplist**(*session*, *type=None*, *region=None*, *canonical_username=None*, *callback=None*, *sp_toplistbrowse=None*, *add_ref=True*)

A Spotify toplist of artists, albums or tracks that are currently most popular worldwide or in a specific region.

Call the `get_toplist()` method on your *Session* instance to get a *Toplist* back.

class spotify.**ToplistRegion**(*value*)

class spotify.**ToplistType**(*value*)

5.1.16 Inbox

class spotify.**InboxPostResult**(*session*, *canonical_username=None*, *tracks=None*, *message=''*, *callback=None*, *sp_inbox=None*, *add_ref=True*)

The result object returned by `Session.inbox_post_tracks()`.

5.1.17 Social

class spotify.social.**Social**(*session*)

Social sharing controller.

You'll never need to create an instance of this class yourself. You'll find it ready to use as the `social` attribute on the `Session` instance.

class spotify.**ScrobblingState**(*value*)

class spotify.**SocialProvider**(*value*)

5.1.18 Player

class spotify.player.**Player**(*session*)

Playback controller.

You'll never need to create an instance of this class yourself. You'll find it ready to use as the `player` attribute on the `Session` instance.

class spotify.player.**PlayerState**

5.1.19 Audio

class spotify.**AudioBufferStats**(*samples*, *stutter*)

Stats about the application's audio buffers.

samples

Number of samples currently in the buffer.

stutter

Number of stutters (audio dropouts) since the last query.

class spotify.**AudioFormat**(*sp_audioformat*)

A Spotify audio format object.

You'll never need to create an instance of this class yourself, but you'll get *AudioFormat* objects as the `audio_format` argument to the `music_delivery` callback.

class spotify.**Bitrate**(*value*)

class spotify.**SampleType**(*value*)

5.1.20 Audio sinks

class spotify.**AlsaSink**(*session*, *device='default'*)

Audio sink for systems using ALSA, e.g. most Linux systems.

This audio sink requires pyalsaaudio. pyalsaaudio is probably packaged in your Linux distribution.

For example, on Debian/Ubuntu you can install it from APT:

```
sudo apt-get install python-alsaaudio
```

Or, if you want to install pyalsaaudio inside a virtualenv, install the ALSA development headers from APT, then pyalsaaudio:

```
sudo apt-get install libasound2-dev
pip install pyalsaaudio
```

The `device` keyword argument is passed on to `alsaaudio.PCM`. Please refer to the pyalsaaudio documentation for details.

Example:

```
>>> import spotify
>>> session = spotify.Session()
>>> audio = spotify.AlsaSink(session)
>>> loop = spotify.EventLoop(session)
>>> loop.start()
# Login, etc...
>>> track = session.get_track('spotify:track:3N2UhXZI4Gf64Ku3cCjz2g')
>>> track.load()
>>> session.player.load(track)
>>> session.player.play()
# Listen to music...
```

class spotify.**PortAudioSink**(*session*)

Audio sink for PortAudio.

PortAudio is available for many platforms, including Linux, macOS, and Windows. This audio sink requires PyAudio. PyAudio is probably packaged in your Linux distribution.

On Debian/Ubuntu you can install PyAudio from APT:

```
sudo apt-get install python-pyaudio
```

Or, if you want to install PyAudio inside a virtualenv, install the PortAudio development headers from APT, then PyAudio:

```
sudo apt-get install portaudio19-dev
pip install --allow-unverified=pyaudio pyaudio
```

On macOS you can install PortAudio using Homebrew:

```
brew install portaudio
pip install --allow-unverified=pyaudio pyaudio
```

For an example of how to use this class, see the *AlsaSink* example. Just replace `AlsaSink` with `PortAudioSink`.

5.1.21 Internal API

Warning: This page documents pyspotify's internal APIs. Its intended audience is developers working on pyspotify itself. You should not use anything you find on this page in your own applications.

libspotify CFFI interface

The CFFI wrapper for the full libspotify API is available as *spotify.ffi* and *spotify.lib*.

spotify.**ffi**

`cffi.FFI` instance which knows about libspotify types.

```
>>> import spotify
>>> spotify.ffi.new('sp_audioformat *')
<cdata 'struct sp_audioformat *' owning 12 bytes>
```

spotify.**lib**

Dynamic wrapper around the full libspotify C API.

```
>>> import spotify
>>> msg = spotify.lib.sp_error_message(spotify.lib.SP_ERROR_OK)
>>> msg
<cdata 'char *' 0x7f29fd922cb5>
>>> spotify.ffi.string(msg)
'No error'
```

spotify.lib will always reflect the contents of the `spotify/api.processed.h` file in the pyspotify distribution. To update the API:

1. Update the file `spotify/api.h` with the latest header file from libspotify.

2. Run the Invoke task `preprocess_header` defined in `tasks.py` by running:

```
invoke preprocess_header
```

   The task will update the `spotify/api.processed.h` file.

3. Commit both header files so that they are distributed with pyspotify.

Thread safety utils

spotify.**serialized**(*f*)

Decorator that serializes access to all decorated functions.

The decorator acquires pyspotify's single global lock while calling any wrapped function. It is used to serialize access to:

- All calls to functions on *spotify.lib*.
- All code blocks working on pointers returned from functions on *spotify.lib*.
- All code blocks working on other internal data structures in pyspotify.

Together this is what makes pyspotify safe to use from multiple threads and enables convenient features like the *EventLoop*.

Internal function.

Event emitter utils

class spotify.utils.**EventEmitter**

Mixin for adding event emitter functionality to a class.

Enumeration utils

class spotify.utils.**IntEnum**(*value*)

An enum type for values mapping to integers.

Tries to stay as close as possible to the enum type specified in **PEP 435** and introduced in Python 3.4.

spotify.utils.**make_enum**(*lib_prefix*, *enum_prefix=''*)

Class decorator for automatically adding enum values.

The values are read directly from the *spotify.lib* CFFI wrapper around libspotify. All values starting with `lib_prefix` are added. The `lib_prefix` is stripped from the name. Optionally, `enum_prefix` can be specified to add a prefix to all the names.

Object loading utils

spotify.utils.**load**(*session*, *obj*, *timeout=None*)

Block until the object's data is loaded.

If the session isn't logged in, a *spotify.Error* is raised as Spotify objects cannot be loaded without being online and logged in.

The `obj` must at least have the `is_loaded` attribute. If it also has an `error()` method, it will be checked for errors to raise.

After `timeout` seconds with no results *Timeout* is raised.

If unspecified, the `timeout` defaults to 10s. Any timeout is better than no timeout, since no timeout would cause programs to potentially hang forever without any information to help debug the issue.

The method returns `self` to allow for chaining of calls.

Sequence utils

class spotify.utils.**Sequence**(*sp_obj*, *add_ref_func*, *release_func*, *len_func*, *getitem_func*)

Helper class for making sequences from a length and getitem function.

The `sp_obj` is assumed to already have gotten an extra reference through `sp_*_add_ref` and to be automatically released through `sp_*_release` when the `sp_obj` object is GC-ed.

String conversion utils

spotify.utils.**get_with_fixed_buffer**(*buffer_length*, *func*, **args*)

Get a unicode string from a C function that takes a fixed-size buffer.

The C function `func` is called with any arguments given in `args`, a buffer of the given `buffer_length`, and `buffer_length`.

Returns the buffer's value decoded from UTF-8 to a unicode string.

spotify.utils.**get_with_growing_buffer**(*func*, **args*)

Get a unicode string from a C function that returns the buffer size needed to return the full string.

The C function `func` is called with any arguments given in `args`, a buffer of fixed size, and the buffer size. If the C function returns a size that is larger than the buffer already filled, the C function is called again with a buffer large enough to get the full string from the C function.

Returns the buffer's value decoded from UTF-8 to a unicode string.

spotify.utils.**to_bytes**(*value*)

Converts bytes, unicode, and C char arrays to bytes.

Unicode strings are encoded to UTF-8.

spotify.utils.**to_bytes_or_none**(*value*)

Converts C char arrays to bytes and C NULL values to None.

spotify.utils.**to_unicode**(*value*)

Converts bytes, unicode, and C char arrays to unicode strings.

Bytes and C char arrays are decoded from UTF-8.

spotify.utils.**to_unicode_or_none**(*value*)

Converts C char arrays to unicode and C NULL values to None.

C char arrays are decoded from UTF-8.

spotify.utils.**to_char**(*value*)

Converts bytes, unicode, and C char arrays to C char arrays.

spotify.utils.**to_char_or_null**(*value*)

Converts bytes, unicode, and C char arrays to C char arrays, and `None` to C NULL values.

Country code utils

spotify.utils.**to_country**(*code*)

Converts a numeric libspotify country code to an ISO 3166-1 two-letter country code in a unicode string.

spotify.utils.**to_country_code**(*country*)

Converts an ISO 3166-1 two-letter country code in a unicode string to a numeric libspotify country code.

CHAPTER

SIX

ABOUT

6.1 Authors

pyspotify 2.x is copyright 2013-2022 Stein Magnus Jodal and contributors. pyspotify is licensed under the Apache License, Version 2.0.

Thanks to Thomas Adamcik who continuously reviewed code and provided feedback during the development of pyspotify 2.x.

The following persons have contributed to pyspotify 2.x. The list is in the order of first contribution. For details on who have contributed what, please refer to our Git repository.

- Stein Magnus Jodal <stein.magnus@jodal.no>
- Richard Ive <richard@xanox.net>
- Thomas Vander Stichele <thomas@apestaart.org>
- Nick Steel <kingosticks@gmail.com>
- Trygve Aaberge <trygveaa@gmail.com>
- Tarik Dadi <daditarik@gmail.com>
- vrs01 <vrs01@users.noreply.github.com>
- Thomas Adamcik <thomas@adamcik.no>
- Edward Betts <edward@4angle.com>

If you want to contribute to pyspotify, see *Contributing*.

6.2 Changelog

6.2.1 v2.1.4 (2022-06-15)

Maintenance release.

- Declare the pyspotify project as dead.
- Add support for Python 3.9 and 3.10. No changes was required, but the test suite now runs on these versions too. (PR: #203)
- Switch from CircleCI to GitHub Actions.

6.2.2 v2.1.3 (2019-12-29)

Maintenance release.

- Document that the playlists API is broken. If it is used, emit a warning to notify the user of the playlist functionality.
- Update project links.

6.2.3 v2.1.2 (2019-12-10)

Maintenance release.

- Silently abort libspotify `sp_*_release()` function calls that happen during process shutdown, after pyspotify's global lock is freed. (Fixes: #202)

6.2.4 v2.1.1 (2019-11-17)

Maintenance release.

- Add support for Python 3.8. No changes was required, but the test suite now runs on this version too.
- Switch from Travis CI to CircleCI.

6.2.5 v2.1.0 (2019-07-08)

Maintenance release.

- Drop support for Python 3.3 and 3.4, as both has reached end of life.
- Add support for Python 3.6 and 3.7. No changes was required, but the test suite now runs on these versions too.
- On Python 3, import `Iterable`, `MutableSequence`, and `Sequence` from `collections.abc` instead of `collections`. This fixes a deprecation warning on Python 3.7 and prepares for Python 3.8.
- Document that the search API is broken. If it is used, raise an exception instead of sending the search to Spotify, as that seems to disconnect your session. (Fixes: #183)
- Format source code with Black.

6.2.6 v2.0.5 (2015-09-22)

Bug fix release.

- To follow up on the previous release, the getters for the proxy configs now convert empty strings in the `sp_session_config` struct back to `None`. Thus, the need to set these configs to empty strings in the struct to make sure the cached settings are cleared from disk are now an internal detail, hidden from the user of pyspotify.
- Make `tracefile` default to `None` and set to `NULL` in the libspotify config struct. If it is set to an empty string by default, libspotify will try to use a file with an empty filename for cache and fail with "LibError: Unable to open trace file". Now empty strings are set as `NULL` in the `sp_session_config` struct. (Fixes: mopidy-spotify#70)
- libspotify segfaults if the `device_id` config is set to an empty string. We now avoid this segfault if `device_id` is set to an empty string by setting the `device_id` field in libspotify's `sp_session_config` struct to `NULL` instead.

- As some test tools (like coverage.py 4.0) no longer support Python 3.2, we no longer test pyspotify on Python 3.2. Though, we have not done anything to intentionally break support for Python 3.2 ourselves.

6.2.7 v2.0.4 (2015-09-15)

Bug fix release.

- It has been observed that libspotify will reuse cached proxy settings from previous sessions if the proxy fields on the `sp_session_config` struct are set to `NULL`. When the `sp_session_config` fields are set to an empty string, the cached settings are updated. When attributes on *spotify.Config* are set to `None`, we now set the fields on `sp_session_config` to empty strings instead of `NULL`.

6.2.8 v2.0.3 (2015-09-05)

Bug fix release.

- Make moving a playlist to its own location a no-op instead of causing an error like libspotify does. (Fixes: #175)
- New better installation instructions. (Fixes: #174)

6.2.9 v2.0.2 (2015-08-06)

Bug fix release.

- Use `sp_session_starred_for_user_create(session, username)` instead of `sp_playlist_create(session, link)` to get starred playlists by URI. The previous approach caused segfaults under some circumstances. (Fixes: mopidy-spotify#60)

6.2.10 v2.0.1 (2015-07-20)

Bug fix release.

- Make `spotify.Session.get_playlist()` acquire the global lock before modifying the global playlist cache.
- Make *Playlist* and *PlaylistContainer* register callbacks with libspotify if and only if a Python event handler is added to the object. Previously, we always registered the callbacks with libspotify. Hopefully, this will remove the preconditions for the crashes in #122, #153, and #165.

6.2.11 v2.0.0 (2015-06-01)

pyspotify 2.x is a full rewrite of pyspotify. While pyspotify 1.x is a CPython C extension, pyspotify 2.x uses CFFI to wrap the libspotify C library. It works on CPython 2.7 and 3.2+, as well as PyPy 2.6+. pyspotify 2.0 makes 100% of the libspotify 12.1.51 API available from Python, going far beyond the API coverage of pyspotify 1.x.

The following are the changes since pyspotify 2.0.0b5.

Dependency changes

- Require cffi >= 1.0. (Fixes: #133, #160)
- If you're using pyspotify with PyPy you need version 2.6 or newer as older versions of PyPy come with a too old cffi version. For PyPy3, you'll probably need the yet to be released PyPy3 2.5.

ALSA sink

- Changed the *spotify.AlsaSink* keyword argument `card` to `device` to align with pyalsaaudio 0.8.
- Updated to work with pyalsaaudio 0.8 which changed the signature of `alsaaudio.PCM`. *spotify.AlsaSink* still works with pyalsaaudio 0.7, but 0.8 is recommended at least for Python 3 users, as it fixes a memory leak present on Python 3 (see #127). (Fixes: #162)

6.2.12 v2.0.0b5 (2015-05-09)

A fifth beta with a couple of bug fixes.

Minor changes

- Changed `spotify.Link.as_playlist()` to also support creating playlists from links with type `spotify.LinkType.STARRED`.
- Changed all `load()` methods to raise *spotify.Error* instead of `RuntimeError` if the session isn't logged in.
- Changed from nose to py.test as test runner.

Bug fixes

- Work around segfault in libspotify when `spotify.Config.cache_location` is set to `None` and then used to create a session. (Fixes: #151)
- Return a *spotify.PlaylistPlaceholder* object instead of raising an exception if the playlist container contains an element of type `PLACEHOLDER`. (Fixes: #159)

6.2.13 v2.0.0b4 (2015-01-13)

The fourth beta includes a single API change, a couple of API additions, and otherwise minor tweaks to logging.

pyspotify 2.x has been verified to work on PyPy3, and PyPy3 is now part of the test matrix.

Minor changes

- Added `spotify.Link.url` which returns an `https://open.spotify.com/...` URL for the link object.
- Adjusted `info`, `warning`, and `error` level log messages to include the word "Spotify" or "pyspotify" for context in applications not including the logger name in the log. `debug` level messages have not been changed, as it is assumed that more details, including the logger name, is included in debug logs.
- Added `spotify.player.Player.state` which is maintained by calls to the various *Player* methods.

Bug fixes

- Fix `spotify.Playlist.reorder_tracks()`. It now accepts a list of track indexes instead of a list of tracks. This makes it possible to reorder any of multiple identical tracks in a playlist and is consistent with `spotify.Playlist.remove_tracks()`. (Fixes: #134)
- Fix pause/resume/stop in the `examples/shell.py` example. (PR: #140)
- Errors passed to session callbacks are now logged with the full error type representation, instead of just the integer value. E.g. where previously only "8" was logged, we now log "<ErrorType.UNABLE_TO_CONTACT_SERVER: 8>".

6.2.14 v2.0.0b3 (2014-05-04)

The third beta includes a couple of changes to the API in the name of consistency, as well as three minor improvements.

Also worth noticing is that with this release, pyspotify 2.x has been in development for a year and a day. Happy birthday, pyspotify 2!

Refactoring: Connection cleanup

Parts of *spotify.Session* and `spotify.Session.offline` has been moved to `spotify.Session.connection`:

- `set_connection_type()` has been replaced by `session.connection.type`, which now also allows reading the current connection type.
- `set_connection_rules()` has been replaced by:
  - `allow_network`
  - `allow_network_if_roaming`
  - `allow_sync_over_wifi`
  - `allow_sync_over_mobile`

  The new attributes allow reading the current connection rules, so your application don't have to keep track of what rules it has set.
- `session.connection_state` has been replaced by `session.connection.state`

Refactoring: position vs index

Originally, pyspotify named everything identically with libspotify and have thus ended up with a mix of the terms "position" and "index" for the same concept. Now, we use "index" all over the place, as that's also the name used in the Python world at large. This changes the signature of three methods, which may affect you if you use keyword arguments to call the methods. There's also a number of affected events, but these changes shouldn't stop your code from working.

Affected functions include:

- `spotify.Playlist.add_tracks()` now takes `index` instead of `position`.
- `spotify.Playlist.remove_tracks()` now takes `indexes` instead of `positions`.
- `spotify.Playlist.reorder_tracks()` now takes `new_index` instead of `new_position`.

Affected events include:

- `spotify.PlaylistContainerEvent.PLAYLIST_ADDED`

- spotify.PlaylistContainerEvent.PLAYLIST_REMOVED
- spotify.PlaylistContainerEvent.PLAYLIST_MOVED
- spotify.PlaylistEvent.TRACKS_ADDED
- spotify.PlaylistEvent.TRACKS_REMOVED
- spotify.PlaylistEvent.TRACKS_MOVED
- spotify.PlaylistEvent.TRACK_CREATED_CHANGED
- spotify.PlaylistEvent.TRACK_SEEN_CHANGED
- spotify.PlaylistEvent.TRACK_MESSAGE_CHANGED

Minor changes

- load() methods now return the object if it is already loaded, even if state isn't LOGGED_IN. Previously, a RuntimeError was raised requiring the session to be logged in and online before loading already loaded objects.
- spotify.Playlist.tracks now implements the collections.MutableSequence contract, supporting deleting items with del playlist.tracks[i], adding items with playlist.tracks[i] = track, etc.
- spotify.Session.get_link() and all other methods accepting Spotify URIs now also understand open.spotify.com and play.spotify.com URLs.

6.2.15 v2.0.0b2 (2014-04-29)

The second beta is a minor bug fix release.

Bug fixes

- Fix spotify.Playlist.remove_tracks. It now accepts a list of track positions instead of a list of tracks. This makes it possible to remove any of multiple identical tracks in a playlist. (Fixes: #128)

Minor changes

- Make all objects compare as equal and have the same hash if they wrap the same libspotify object. This makes it possible to find the index of a track in a playlist by doing playlist.tracks.index(track), where playlist.tracks is a custom collection always returning new *Track* instances. (Related to: #128)
- spotify.Config.ca_certs_filename now works on systems where libspotify has this field. On systems where this field isn't present in libspotify, assigning to it will have no effect. Previously, assignment to this field was a noop on all platforms because the field is missing from libspotify on OS X.

6.2.16 v2.0.0b1 (2014-04-24)

pyspotify 2.x is a full rewrite of pyspotify. While pyspotify 1.x is a CPython C extension, pyspotify 2.x uses CFFI to make 100% of the libspotify C library available from Python. It works on CPython 2.7 and 3.2+, as well as PyPy 2.1+.

Since the previous release, pyspotify has become thread safe. That is, pyspotify can safely be used from multiple threads. The added thread safety made an integrated event loop possible, which greatly simplifies the usage of pyspotify, as can be seen from the updated example in `examples/shell.py`. Audio sink helpers for ALSA and PortAudio have been added, together with updated examples that can play music. A number of bugs have been fixed, and at the time of the release, there are no known issues.

The pyspotify 2.0.0b1 release marks the completion of all planned features for pyspotify 2.x. The plans for the next releases are focused on fixing bugs as they surface, incrementally improving the documentation, and integrating feedback from increased usage of the library in the wild.

Feature: Thread safety

- Hold the global lock while we are working with pointers returned by libspotify. This ensures that we never call libspotify from another thread while we are still working on the data returned by the previous libspotify call, which could make the data garbage.
- Ensure we never edit shared data structures without holding the global lock.

Feature: Event loop

- Add *spotify.EventLoop* helper thread that reacts to `NOTIFY_MAIN_THREAD` events and calls `process_events()` for you when appropriate.
- Update `examples/shell.py` to be a lot simpler with the help of the new event loop.

Feature: Audio playback

- Add *spotify.AlsaSink*, an audio sink for playback through ALSA on Linux systems.
- Add *spotify.PortAudioSink*, an audio sink for playback through PortAudio on most platforms, including Linux, OS X, and Windows.
- Update `examples/shell.py` to use the ALSA sink to play music.
- Add `examples/play_track.py` as a simpler example of audio playback.

Refactoring: Remove global state

To prepare for removing all global state, the use of the module attribute `spotify.session_instance` has been replaced with explicit passing of the session object to all objects that needs it. To allow for this, the following new methods have been added, and should be used instead of their old equivalents:

- `spotify.Session.get_link()` replaces *spotify.Link*.
- `spotify.Session.get_track()` replaces *spotify.Track*.
- `spotify.Session.get_local_track()` replaces `spotify.LocalTrack`.
- `spotify.Session.get_album()` replaces *spotify.Album*.
- `spotify.Session.get_artist()` replaces *spotify.Artist*.
- `spotify.Session.get_playlist()` replaces *spotify.Playlist*.

- spotify.Session.get_user() replaces *spotify.User*.
- spotify.Session.get_image() replaces *spotify.Image*.
- spotify.Session.get_toplist() replaces *spotify.Toplist*.

Refactoring: Consistent naming of Session members

With all the above getters added to the *spotify.Session* object, it made sense to rename some existing methods of *Session* for consistency:

- spotify.Session.starred_for_user() is replaced by get_starred().
- spotify.Session.starred to get the currently logged in user's starred playlist is replaced by get_starred() without any arguments.
- spotify.Session.get_published_playlists() replaces published_playlists_for_user(). As previously, it returns the published playlists for the currently logged in user if no username is provided.

Refactoring: Consistent naming of threading.Event objects

All threading.Event objects have been renamed to be consistently named across classes.

- spotify.AlbumBrowser.loaded_event replaces spotify.AlbumBrowser.complete_event.
- spotify.ArtistBrowser.loaded_event replaces spotify.ArtistBrowser.complete_event.
- spotify.Image.loaded_event replaces spotify.Image.load_event.
- spotify.InboxPostResult.loaded_event replaces spotify.InboxPostResult.complete_event.
- spotify.Search.loaded_event replaces spotify.Search.complete_event.
- spotify.Toplist.loaded_event replaces spotify.Toplist.complete_event.

Refactoring: Change how to register image load listeners

pyspotify has two main schemes for registering listener functions:

- Objects that only emit an event when it is done loading, like *AlbumBrowser*, *ArtistBrowser*, *InboxPostResult*, *Search*, and *Toplist*, accept a single callback as a callback argument to its constructor or constructor methods.
- Objects that have multiple callback events, like *Session*, *PlaylistContainer*, and *Playlist*, accept the registration and unregistration of one or more listener functions for each event it emits. This can happen any time during the object's life cycle.

Due to pyspotify's close mapping to libspotify's organization, Image objects used to use a third variant with two methods, add_load_callback() and remove_load_callback(), for adding and removing load callbacks. These methods have now been removed, and *Image* accepts a callback argument to its constructor and constructor methods:

- spotify.Album.cover() accepts a callback argument.
- spotify.Artist.portrait() accepts a callback argument.
- spotify.ArtistBrowser.portraits() is now a method and accepts a callback argument.
- spotify.Link.as_image() accepts a callback argument.
- spotify.Playlist.image() is now a method and accepts a callback argument.
- spotify.Session.get_image() accepts a callback argument.

Bug fixes

- Remove multiple extra `sp_link_add_ref()` calls, potentially causing memory leaks in libspotify.
- Add missing error check to `spotify.Playlist.add_tracks()`.
- Keep album, artist, image, inbox, search, and toplist objects alive until their complete/load callbacks have been called, even if the library user doesn't keep any references to the objects. (Fixes: #121)
- Fix flipped logic causing crash in `spotify.Album.cover_link()`. (Fixes: #126)
- Work around segfault in libspotify if `private_session` is set before the session is logged in and the first events are processed. This is a bug in libspotify which has been reported to Spotify through their IRC channel.
- Multiple attributes on *Track* raised an exception if accessed before the track was loaded. They now return `None` or similar as documented.
- Fix segfault when creating local tracks without all arguments specified. `NULL` was used as the placeholder instead of the empty string.
- Support negative indexes on all custom sequence types. For example, `collection[-1]` returns the last element in the collection.
- We now cache playlists when created from URIs. Previously, only playlists created from `sp_playlist` objects were cached. This avoids a potentially large number of wrapper object recreations due to a flood of updates to the playlist when it is initially loaded. Combined with having registered a callback for the libspotify `playlist_update_in_progress` callback, this could cause deep call stacks reaching the maximum recursion depth. (Fixes: #122)

Minor changes

- Add *spotify.get_libspotify_api_version()* and *spotify.get_libspotify_build_id()*.
- Running `python setup.py test` now runs the test suite.
- The tests are now compatible with CPython 3.4. No changes to the implementation was required.
- The test suite now runs on Mac OS X, using CPython 2.7, 3.2, 3.3, 3.4, and PyPy 2.2, on every push to GitHub.

6.2.17 v2.0.0a1 (2014-02-14)

pyspotify 2.x is a full rewrite of pyspotify. While pyspotify 1.x is a CPython C extension, pyspotify 2.x uses CFFI to wrap the libspotify C library. It works on CPython 2.7 and 3.2+, as well as PyPy 2.1+.

This first alpha release of pyspotify 2.0.0 makes 100% of the libspotify 12.1.51 API available from Python, going far beyond the API coverage of pyspotify 1.x.

pyspotify 2.0.0a1 has an extensive test suite with 98% line coverage. All tests pass on all combinations of CPython 2.7, 3.2, 3.3, PyPy 2.2 running on Linux on i386, amd64, armel, and armhf. Mac OS X should work, but has not been tested recently.

This release *does not* provide:

- thread safety,
- an event loop for regularly processing libspotify events, or
- audio playback drivers.

These features are planned for the upcoming prereleases.

Development milestones

- 2014-02-13: Playlist callbacks complete. pyspotify 2.x now covers 100% of the libspotify 12 API. Docs reviewed, quickstart guide extended. Redundant getters/setters removed.
- 2014-02-08: Playlist container callbacks complete.
- 2014-01-31: Redesign session event listening to a model supporting multiple listeners per event, with a nicer API for registering listeners.
- 2013-12-16: Ensure we never call libspotify from two different threads at the same time. We can't assume that the CPython GIL will ensure this for us, as we target non-CPython interpreters like PyPy.
- 2013-12-13: Artist browsing complete.
- 2013-12-13: Album browsing complete.
- 2013-11-29: Toplist subsystem complete.
- 2013-11-27: Inbox subsystem complete.
- 2013-10-14: Playlist subsystem *almost* complete.
- 2013-06-21: Search subsystem complete.
- 2013-06-10: Album subsystem complete.
- 2013-06-09: Track and artist subsystem complete.
- 2013-06-02: Session subsystem complete, with all methods.
- 2013-06-01: Session callbacks complete.
- 2013-05-25: Session config complete.
- 2013-05-16: Link subsystem complete.
- 2013-05-09: User subsystem complete.
- 2013-05-08: Session configuration and creation, with login and logout works.
- 2013-05-03: The Python object `spotify.lib` is a working CFFI wrapper around the entire libspotify 12 API. This will be the foundation for more pythonic APIs. The library currently works on CPython 2.7, 3.3 and PyPy 2.

6.2.18 v1.x series

See the pyspotify 1.x changelog.

6.3 Contributing

Contributions to pyspotify are welcome! Here are some tips to get you started hacking on pyspotify and contributing back your patches.

6.3.1 Development setup

1. Make sure you have the following Python versions installed:
   - CPython 2.7
   - CPython 3.5
   - CPython 3.6
   - CPython 3.7
   - PyPy2.7 6.0+
   - PyPy3.5 6.0+

   If you're on Ubuntu, the Dead Snakes PPA has packages of both old and new Python versions.

2. Install the following with development headers: Python, libffi, and libspotify.

   On Debian/Ubuntu, make sure you have apt.mopidy.com in your APT sources to get the libspotify package, then run:

   ```
   sudo apt-get install python-all-dev python3-all-dev libffi-dev libspotify-dev
   ```

3. Create and activate a virtualenv:

   ```
   virtualenv ve
   source ve/bin/activate
   ```

4. Install development dependencies:

   ```
   pip install -e ".[dev]
   ```

5. Run tests.

   For a quick test suite run, using the virtualenv's Python version:

   ```
   py.test
   ```

   For a complete test suite run, using all the Python implementations:

   ```
   tox
   ```

6. For some more development task helpers, install `invoke`:

   ```
   pip install invoke
   ```

   To list available tasks, run:

   ```
   invoke --list
   ```

   For example, to run tests on any file change, run:

   ```
   invoke test --watch
   ```

   Or, to build docs when any file changes, run:

   ```
   invoke docs --watch
   ```

   See the file `tasks.py` for the task definitions.

6.3.2 Submitting changes

- Code should be accompanied by tests and documentation. Maintain our excellent test coverage.
- Follow the existing code style, especially make sure `flake8` does not complain about anything.
- Write good commit messages. Here's three blog posts on how to do it right:
  - Writing Git commit messages
  - A Note About Git Commit Messages
  - On commit messages
- One branch per feature or fix. Keep branches small and on topic.
- Send a pull request to the `v2.x/master` branch. See the GitHub pull request docs for help.

6.3.3 Additional resources

- Issue tracker
- GitHub documentation
- libspotify downloads and documentation archive

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

E

F

G

I

L

M

N

O

P

S

T

U