Mopidy Documentation

Release 0.19.5

Stein Magnus Jodal and contributors

Apr 23, 2023

Contents

Mopidy is an extensible music server written in Python.

Mopidy plays music from local disk, Spotify, SoundCloud, Google Play Music, and more. You edit the playlist from any phone, tablet, or computer using a range of MPD and web clients.

Stream music from the cloud

Vanilla Mopidy only plays music from your *local disk* and *radio streams*. Through *extensions*, Mopidy can play music from cloud services like Spotify, SoundCloud, and Google Play Music. With Mopidy's extension support, backends for new music sources can be easily added.

Mopidy is just a server

Mopidy is a Python application that runs in a terminal or in the background on Linux computers or Macs that have network connectivity and audio output. Out of the box, Mopidy is an *MPD* and *HTTP* server. *Additional frontends* for controlling Mopidy can be installed from extensions.

Everybody use their favorite client

You and the people around you can all connect their favorite *MPD* or *web client* to the Mopidy server to search for music and manage the playlist together. With a browser or MPD client, which is available for all popular operating systems, you can control the music from any phone, tablet, or computer.

Mopidy on Raspberry Pi

The *Raspberry Pi* is a popular device to run Mopidy on, either using Raspbian or Arch Linux. It is quite slow, but it is very affordable. In fact, the Kickstarter funded Gramofon: Modern Cloud Jukebox project used Mopidy on a Raspberry Pi to prototype the Gramofon device. Mopidy is also a major building block in the Pi Musicbox integrated audio jukebox system for Raspberry Pi.

Mopidy is hackable

Mopidy's extension support and *Python*, *JSON-RPC*, and *JavaScript APIs* makes Mopidy perfect for building your own hacks. In one project, a Raspberry Pi was embedded in an old cassette player. The buttons and volume control are wired up with GPIO on the Raspberry Pi, and is used to control playback through a custom Mopidy extension. The cassettes have NFC tags used to select playlists from Spotify.

Getting started

To get started with Mopidy, start by reading *Installation*.

Getting help

If you get stuck, you can get help at the Mopidy discussion forum. We also hang around at IRC on the `#mopidy` channel at irc.freenode.net. The IRC channel has public searchable logs.

If you stumble into a bug or have a feature request, please create an issue in the issue tracker. If you're unsure if it's a bug or not, ask for help in the forum or at IRC first. The source code may also be of help.

If you want to stay up to date on Mopidy developments, you can follow @mopidy on Twitter. There's also a mailing list used for announcements related to Mopidy and Mopidy extensions.

CHAPTER 1

Usage

1.1 Installation

There are several ways to install Mopidy. What way is best depends upon your OS and/or distribution.

If you want to contribute to the development of Mopidy, you should first read the general installation instructions, then have a look at *Running Mopidy from Git*.

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

If you run a Debian based Linux distribution, like Ubuntu, the easiest way to install Mopidy is from the Mopidy APT archive. When installing from the APT archive, you will automatically get updates to Mopidy in the same way as you get updates to the rest of your system.

If you're on a Raspberry Pi running Debian or Raspbian, the following instructions should work for you as well. If you're setting up a Raspberry Pi from scratch, we have a guide for installing Debian/Raspbian and Mopidy. See *Raspberry Pi: Mopidy on a credit card*.

Note: The packages should work with:

- Debian stable and testing,
- Raspbian stable and testing,
- Ubuntu 14.04 LTS and later.

Some of the packages, including the core "mopidy" packages, does *not* work on Ubuntu 12.04 LTS.

This is just what we currently support, not a promise to continue to support the same in the future. We *will* drop support for older distributions when supporting those stops us from moving forward with the project.

---

1. Add the archive's GPG key:

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

2. Add the following to `/etc/apt/sources.list`, or if you have the directory `/etc/apt/sources.list.d/`, add it to a file called `mopidy.list` in that directory:

```
# Mopidy APT archive
deb http://apt.mopidy.com/ stable main contrib non-free
deb-src http://apt.mopidy.com/ stable main contrib non-free
```

   For the lazy, you can simply run the following command to create `/etc/apt/sources.list.d/mopidy.list`:

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

3. Install Mopidy and all dependencies:

```
sudo apt-get update
sudo apt-get install mopidy
```

4. Optional: If you want to use any Mopidy extensions, like Spotify support or Last.fm scrobbling, you need to install additional packages.

   To list all the extensions available from apt.mopidy.com, you can run:

```
apt-cache search mopidy
```

   To install one of the listed packages, e.g. `mopidy-spotify`, simply run:

```
sudo apt-get install mopidy-spotify
```

   For a full list of available Mopidy extensions, including those not installable from apt.mopidy.com, see *Extensions*.

5. Before continuing, make sure you've read the *Debian package* section to learn about the differences between running Mopidy as a system service and manually as your own system user.

6. Finally, you need to set a couple of *config values*, and then you're ready to *run Mopidy*.

When a new release of Mopidy is out, and you can't wait for you system to figure it out for itself, run the following to upgrade right away:

```
sudo apt-get update
sudo apt-get dist-upgrade
```

1.1.2 Arch Linux: Install from AUR

If you are running Arch Linux, you can install Mopidy using the mopidy package found in AUR.

1. To install Mopidy with all dependencies, you can use for example yaourt:

```
yaourt -S mopidy
```

   To upgrade Mopidy to future releases, just upgrade your system using:

```
yaourt -Syu
```

2. Optional: If you want to use any Mopidy extensions, like Spotify support or Last.fm scrobbling, AUR also has packages for several Mopidy extensions.

   For a full list of available Mopidy extensions, including those not installable from AUR, see *Extensions*.

3. Finally, you need to set a couple of *config values*, and then you're ready to *run Mopidy*.

1.1.3 OS X: Install from Homebrew

If you are running OS X, you can install everything needed with Homebrew.

1. Install Xcode command line developer tools. Do this even if you already have Xcode installed:

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

2. Install XQuartz. This is needed by GStreamer which Mopidy use heavily.

3. Install Homebrew.

4. If you are already using Homebrew, make sure your installation is up to date before you continue:

   ```
   brew update
   brew upgrade
   ```

   Notice that this will upgrade all software on your system that have been installed with Homebrew.

5. Mopidy works out of box if you have installed Python from Homebrew:

   ```
   brew install python
   ```

   Note: If you want to use the Python version bundled with OS X, you'll need to include Python packages installed by Homebrew in your `PYTHONPATH`. If you don't do this, the `mopidy` executable will not find its dependencies and will crash.

   You can either amend your `PYTHONPATH` permanently, by adding the following statement to your shell's init file, e.g. `~/.bashrc`:

   ```
   export PYTHONPATH=$(brew --prefix)/lib/python2.7/site-packages:$PYTHONPATH
   ```

   And then reload the shell's init file or restart your terminal:

   ```
   source ~/.bashrc
   ```

   Or, you can prefix the Mopidy command every time you run it:

   ```
   PYTHONPATH=$(brew --prefix)/lib/python2.7/site-packages mopidy
   ```

   ---

6. Mopidy has its own Homebrew formula repo, called a "tap". To enable our Homebrew tap, run:

   ```
   brew tap mopidy/mopidy
   ```

7. To install Mopidy, run:

   ```
   brew install mopidy
   ```

8. Optional: If you want to use any Mopidy extensions, like Spotify support or Last.fm scrobbling, the Homebrew tap has formulas for several Mopidy extensions as well.

   To list all the extensions available from our tap, you can run:

   ```
   brew search mopidy
   ```

   For a full list of available Mopidy extensions, including those not installable from Homebrew, see *Extensions*.

9. Finally, you need to set a couple of *config values*, and then you're ready to *run Mopidy*.

1.1.4 Install from source

If you are on Linux, but can't install *from the APT archive* or *from AUR*, you can install Mopidy from source by hand.

1. First of all, you need Python 2.7. Check if you have Python and what version by running:

   ```
   python --version
   ```

2. You need to make sure you have `pip`, the Python package installer. You'll also need a C compiler and the Python development headers to build pyspotify later.

   This is how you install it on Debian/Ubuntu:

   ```
   sudo apt-get install build-essential python-dev python-pip
   ```

   And on Arch Linux from the official repository:

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

   And on Fedora Linux from the official repositories:

   ```
   sudo yum install -y gcc python-devel python-pip
   ```

   Note: On Fedora Linux, you must replace `pip` with `pip-python` in the following steps.

3. Then you'll need to install GStreamer 0.10 (>= 0.10.31, < 0.11), with Python bindings. GStreamer is packaged for most popular Linux distributions. Search for GStreamer in your package manager, and make sure to install the Python bindings, and the "good" and "ugly" plugin sets.

   If you use Debian/Ubuntu you can install GStreamer like this:

   ```
   sudo apt-get install python-gst0.10 gstreamer0.10-plugins-good \
       gstreamer0.10-plugins-ugly gstreamer0.10-tools
   ```

   If you use Arch Linux, install the following packages from the official repository:

   ```
   sudo pacman -S gstreamer0.10-python gstreamer0.10-good-plugins \
       gstreamer0.10-ugly-plugins
   ```

   If you use Fedora you can install GStreamer like this:

   ```
   sudo yum install -y python-gst0.10 gstreamer0.10-plugins-good \
       gstreamer0.10-plugins-ugly gstreamer0.10-tools
   ```

   If you use Gentoo you need to be careful because GStreamer 0.10 is in a different lower slot than 1.0, the default. Your emerge commands will need to include the slot:

```
emerge -av gst-python gst-plugins-bad:0.10 gst-plugins-good:0.10 \
    gst-plugins-ugly:0.10 gst-plugins-meta:0.10
```

`gst-plugins-meta:0.10` is the one that actually pulls in the plugins you want, so pay attention to the use flags, e.g. `alsa`, `mp3`, etc.

4. Install the latest release of Mopidy:

```
sudo pip install -U mopidy
```

To upgrade Mopidy to future releases, just rerun this command.

Alternatively, if you want to track Mopidy development closer, you may install a snapshot of Mopidy's `develop` Git branch using pip:

```
sudo pip install --allow-unverified=mopidy mopidy==dev
```

5. Optional: If you want Spotify support in Mopidy, you'll need to install libspotify and the Mopidy-Spotify extension.

   1. Download and install the latest version of libspotify for your OS and CPU architecture from Spotify.

      For libspotify 12.1.51 for 64-bit Linux the process is as follows:

      ```
      wget https://developer.spotify.com/download/libspotify/libspotify-12.1.51-
      ↪Linux-x86_64-release.tar.gz
      tar zxfv libspotify-12.1.51-Linux-x86_64-release.tar.gz
      cd libspotify-12.1.51-Linux-x86_64-release/
      sudo make install prefix=/usr/local
      ```

      Remember to adjust the above example for the latest libspotify version supported by pyspotify, your OS, and your CPU architecture.

   2. If you're on Fedora, you must add a configuration file so libspotify.so can be found:

      ```
      echo /usr/local/lib | sudo tee /etc/ld.so.conf.d/libspotify.conf
      sudo ldconfig
      ```

   3. Then install the latest release of Mopidy-Spotify using pip:

      ```
      sudo pip install -U mopidy-spotify
      ```

6. Optional: If you want to scrobble your played tracks to Last.fm, you need to install Mopidy-Scrobbler:

```
sudo pip install -U mopidy-scrobbler
```

7. For a full list of available Mopidy extensions, see *Extensions*.
8. Finally, you need to set a couple of *config values*, and then you're ready to *run Mopidy*.

1.1.5 Raspberry Pi: Mopidy on a credit card

Mopidy runs nicely on a Raspberry Pi. As of January 2013, Mopidy will run with Spotify support on both the armel (soft-float) and armhf (hard-float) architectures, which includes the Raspbian distribution.

How to for Raspbian "wheezy" and Debian "wheezy"

This guide applies for both:

- Raspbian "wheezy" for armhf (hard-float), and
- Debian "wheezy" for armel (soft-float)

If you don't know which one to select, go for the armhf variant, as it'll give you a lot better performance.

1. Download the latest "wheezy" disk image from http://www.raspberrypi.org/downloads/. This was last tested with the images from 2013-05-25 for armhf and 2013-05-29 for armel.
2. Flash the OS image to your SD card. See http://elinux.org/RPi_Easy_SD_Card_Setup for help.
3. If you have an SD card that's >2 GB, you don't have to resize the file systems on another computer. Just boot up your Raspberry Pi with the unaltered partions, and it will boot right into the `raspi-config` tool, which will let you grow the root file system to fill the SD card. This tool will also allow you do other useful stuff, like turning on the SSH server.
4. You can login to the default user using username `pi` and password `raspberry`. To become root, just enter `sudo -i`.
5. To avoid a couple of potential problems with Mopidy, turn on IPv6 support:
   - Load the IPv6 kernel module now:

     ```
     sudo modprobe ipv6
     ```

   - Add `ipv6` to `/etc/modules` to ensure the IPv6 kernel module is loaded on boot:

```
echo ipv6 | sudo tee -a /etc/modules
```

6. Since I have a HDMI cable connected, but want the sound on the analog sound connector, I have to run:

   ```
   sudo amixer cset numid=3 1
   ```

   to force it to use analog output. `1` means analog, `0` means auto, and is the default, while `2` means HDMI. You can test sound output independent of Mopidy by running:

   ```
   aplay /usr/share/sounds/alsa/Front_Center.wav
   ```

   If you hear a voice saying "Front Center", then your sound is working.

   To make the change to analog output stick, you can add the `amixer` command to e.g. `/etc/rc.local`, which will be executed when the system is booting.

7. Install Mopidy and its dependencies as described in *Debian/Ubuntu: Install from apt.mopidy.com*.

8. Finally, you need to set a couple of *config values*, and then you're ready to *run Mopidy*. Alternatively you may want to have Mopidy run as a *system service*, automatically starting at boot.

Appendix A: Fixing audio quality issues

As of about April 2013 the following steps should resolve any audio issues for HDMI and analog without the use of an external USB sound card.

1. Ensure your system is up to date. On Debian based systems run:

   ```
   sudo apt-get update
   sudo apt-get dist-upgrade
   ```

2. Ensure you have a new enough firmware. On Debian based systems rpi-update can be used.

3. Update either `~/.asoundrc` or `/etc/asound.conf` to the following:

   ```
   pcm.!default {
           type hw
           card 0
   }
   ctl.!default {
           type hw
           card 0
   }
   ```

   Note that if you have an `~/.asoundrc` it will overide any global settings from `/etc/asound.conf`.

4. For Mopidy to output audio directly to ALSA, instead of Jack which GStreamer usually defaults to on Raspberry Pi, install the `gstreamer0.10-alsa` package:

   ```
   sudo apt-get install gstreamer0.10-alsa
   ```

   Then update your `~/.config/mopidy/mopidy.conf` to contain:

   ```
   [audio]
   output = alsasink
   ```

Following these steps you should be able to get crackle free sound on either HDMI or analog. Note that you might need to ensure that PulseAudio is no longer running to get this working nicely.

This recipe has been confirmed as working by a number of users on our issue tracker and IRC. As a reference, the following versions where used for testing this, however all newer and some older version are likely to work as we have not determined the exact revision that fixed this:

```
$ uname -a
Linux raspberrypi 3.6.11+ #408 PREEMPT Wed Apr 10 20:33:39 BST 2013 armv6l GNU/Linux

$ /opt/vc/bin/vcgencmd version
Apr 25 2013 01:07:36
Copyright (c) 2012 Broadcom
version 386589 (release)
```

The only remaining known issue is a slight gap in playback at track changes this is likely due to gapless playback not being implemented and is being worked on irrespective of Raspberry Pi related work.

Appendix B: Raspbmc not booting

Due to a dependency version problem where XBMC uses another version of libtag than what Debian originally ships with, you might have to make some minor changes for Raspbmc to start properly after installing Mopidy.

If you notice that XBMC is not starting but gets stuck in a loop, you need to make the following changes:

```
sudo ln -sf /home/pi/.xbmc-current/xbmc-bin/lib/xbmc/system/libtag.so.1 \
    /usr/lib/arm-linux-gnueabihf/libtag.so.1
```

However, this will not persist the changes. To persist the changes edit `/etc/ld.so.conf.d/arm-linux-gnueabihf.conf` and add the following at the top:

```
/home/pi/.xbmc-current/xbmc-bin/lib/xbmc/system
```

It's very important to add it at the top of the file as this indicates the priority of the folder in which to look for shared libraries.

XBMC doesn't play nicely with the system wide installed version of libtag that got installed together with Mopidy, but rather vendors in its own version.

More info about this issue can be found in this post.

Please note that if you're running Xbian or another XBMC distribution these instructions might vary for your system.

1.2 Configuration

Mopidy has a lot of config values you can tweak, but you only need to change a few to get up and running. A complete `~/.config/mopidy/mopidy.conf` may be as simple as this:

```
[mpd]
hostname = ::

[spotify]
username = alice
password = mysecret
```

Mopidy primarily reads config from the file `~/.config/mopidy/mopidy.conf`, where `~` means your *home directory*. If your username is `alice` and you are running Linux, the config file should probably be at `/home/alice/.config/mopidy/mopidy.conf`. You can either create the configuration file yourself, or run the `mopidy` com-

mand, and it will create an empty config file for you and print what config values must be set to successfully start Mopidy.

When you have created the configuration file, open it in a text editor, and add the config values you want to change. If you want to keep the default for a config value, you **should not** add it to the config file, but leave it out so that when we change the default value in a future version, you won't have to change your configuration accordingly.

To see what's the effective configuration for your Mopidy installation, you can run:

```
mopidy config
```

This will print your full effective config with passwords masked out so that you safely can share the output with others for debugging.

You can find a description of all config values belonging to Mopidy's core below, together with their default values. In addition, all *extensions* got additional config values. The extension's config values and config defaults are documented on the *extension pages*.

1.2.1 Default core configuration

```
[logging]
color = true
console_format = %(levelname)-8s %(message)s
debug_format = %(levelname)-8s %(asctime)s [%(process)d:%(threadName)s] %(name)s\n
↪%(message)s
debug_file = mopidy.log
config_file =

[audio]
mixer = software
mixer_volume =
output = autoaudiosink
visualizer =

[proxy]
scheme =
hostname =
port =
username =
password =
```

1.2.2 Core configuration values

Mopidy's core has the following configuration values that you can change.

Audio configuration

audio/mixer

Audio mixer to use.

The default is `software`, which does volume control inside Mopidy before the audio is sent to the audio output. This mixer does not affect the volume of any other audio playback on the system. It is the only mixer that will affect the audio volume if you're streaming the audio from Mopidy through Shoutcast.

If you want to use a hardware mixer, you need to install a Mopidy extension which integrates with your sound subsystem. E.g. for ALSA, install Mopidy-ALSAMixer.

audio/mixer_volume
Initial volume for the audio mixer.

Expects an integer between 0 and 100.

Setting the config value to blank leaves the audio mixer volume unchanged. For the software mixer blank means 100.

audio/output
Audio output to use.

Expects a GStreamer sink. Typical values are `autoaudiosink`, `alsasink`, `osssink`, `oss4sink`, `pulsesink`, and `shout2send`, and additional arguments specific to each sink. You can use the command `gst-inspect-0.10` to see what output properties can be set on the sink. For example: `gst-inspect-0.10 shout2send`

audio/visualizer
Visualizer to use.

Can be left blank if no visualizer is desired. Otherwise this expects a GStreamer visualizer. Typical values are `monoscope`, `goom`, `goom2k1` or one of the libvisual visualizers.

Logging configuration

logging/color
Whether or not to colorize the console log based on log level. Defaults to `true`.

logging/console_format
The log format used for informational logging.

See the Python logging docs for details on the format.

logging/debug_format
The log format used for debug logging.

See the Python logging docs for details on the format.

logging/debug_file
The file to dump debug log data to when Mopidy is run with the *`mopidy --save-debug-log`* option.

logging/config_file
Config file that overrides all logging config values, see the Python logging docs for details.

loglevels/*
The `loglevels` config section can be used to change the log level for specific parts of Mopidy during development or debugging. Each key in the config section should match the name of a logger. The value is the log level to use for that logger, one of `debug`, `info`, `warning`, `error`, or `critical`.

Proxy configuration

Not all parts of Mopidy or all Mopidy extensions respect the proxy server configuration when connecting to the Internt. Currently, this is at least used when Mopidy's audio subsystem reads media directly from the network, like when listening to Internet radio streams, and by the Mopidy-Spotify extension. With time, we hope that more of the Mopidy ecosystem will respect these configurations to help users on locked down networks.

proxy/scheme
URI scheme for the proxy server. Typically `http`, `https`, `socks4`, or `socks5`.

proxy/hostname
Hostname of the proxy server.

proxy/port
Port number of the proxy server.

proxy/username
Username for the proxy server, if needed.

proxy/password
Password for the proxy server, if needed.

1.2.3 Extension configuration

Mopidy's extensions have their own config values that you may want to tweak. For the available config values, please refer to the docs for each extension. Most, if not all, can be found at *Extensions*.

Mopidy extensions are enabled by default when they are installed. If you want to disable an extension without uninstalling it, all extensions support the `enabled` config value even if it isn't explicitly documented by all extensions. If the `enabled` config value is set to `false` the extension will not be started. For example, to disable the Spotify extension, add the following to your `mopidy.conf`:

```
[spotify]
enabled = false
```

1.2.4 Advanced configurations

Custom audio sink

If you have successfully installed GStreamer, and then run the `gst-inspect` or `gst-inspect-0.10` command, you should see a long listing of installed plugins, ending in a summary line:

```
$ gst-inspect-0.10
... long list of installed plugins ...
Total count: 254 plugins (1 blacklist entry not shown), 1156 features
```

Next, you should be able to produce a audible tone by running:

```
gst-launch-0.10 audiotestsrc ! audioresample ! autoaudiosink
```

If you cannot hear any sound when running this command, you won't hear any sound from Mopidy either, as Mopidy by default uses GStreamer's `autoaudiosink` to play audio. Thus, make this work before you file a bug against Mopidy.

If you for some reason want to use some other GStreamer audio sink than `autoaudiosink`, you can set the *audio/output* config value to a partial GStreamer pipeline description describing the GStreamer sink you want to use.

Example `mopidy.conf` for using OSS4:

```
[audio]
output = oss4sink
```

Again, this is the equivalent of the following `gst-inspect` command, so make this work first:

```
gst-launch-0.10 audiotestsrc ! audioresample ! oss4sink
```

Streaming through SHOUTcast/Icecast

> **Warning:** Known issue
>
> Currently, Mopidy does not handle end-of-track vs end-of-stream signalling in GStreamer correctly. This causes the SHOUTcast stream to be disconnected at the end of each track, rendering it quite useless. For further details, see #492. You can also try the *workaround* mentioned below.

If you want to play the audio on another computer than the one running Mopidy, you can stream the audio from Mopidy through an SHOUTcast or Icecast audio streaming server. Multiple media players can then be connected to the streaming server simultaneously. To use the SHOUTcast output, do the following:

1. Install, configure and start the Icecast server. It can be found in the `icecast2` package in Debian/Ubuntu.
2. Set the *`audio/output`* config value to `lame ! shout2send`. An Ogg Vorbis encoder could be used instead of the lame MP3 encoder.
3. You might also need to change the `shout2send` default settings, run `gst-inspect-0.10 shout2send` to see the available settings. Most likely you want to change `ip`, `username`, `password`, and `mount`.

   Example for MP3 streaming:

   ```
   [audio]
   output = lame ! shout2send mount=mopidy ip=127.0.0.1 port=8000 password=hackme
   ```

   Example for Ogg Vorbis streaming:

   ```
   [audio]
   output = audioresample ! audioconvert ! vorbisenc ! oggmux ! shout2send␣
   ↪mount=mopidy ip=127.0.0.1 port=8000 password=hackme
   ```

Other advanced setups are also possible for outputs. Basically, anything you can use with the `gst-launch-0.10` command can be plugged into *`audio/output`*.

Workaround for end-of-track issues - fallback streams

By using a *fallback stream* playing silence, you can somewhat mitigate the signalling issues.

Example Icecast configuration:

```
<mount>
  <mount-name>/mopidy</mount-name>
  <fallback-mount>/silence.mp3</fallback-mount>
  <fallback-override>1</fallback-override>
</mount>
```

The `silence.mp3` file needs to be placed in the directory defined by `<webroot>...</webroot>`.

New configuration values

Mopidy's config validator will stop you from defining any config values in your config file that Mopidy doesn't know about. This may sound obnoxious, but it helps us detect typos in your config, and deprecated config values that should be removed or updated.

If you're extending Mopidy, and want to use Mopidy's configuration system, you can add new sections to the config without triggering the config validator. We recommend that you choose a good and unique name for the config section so that multiple extensions to Mopidy can be used at the same time without any danger of naming collisions.

1.3 Running Mopidy

To start Mopidy, simply open a terminal and run:

```
mopidy
```

For a complete reference to the Mopidy commands and their command line options, see *mopidy command*.

When Mopidy says `MPD server running at [127.0.0.1]:6600` it's ready to accept connections by any MPD client. Check out our non-exhaustive *MPD clients* list to find recommended clients.

1.3.1 Stopping Mopidy

To stop Mopidy, press `CTRL+C` in the terminal where you started Mopidy.

Mopidy will also shut down properly if you send it the TERM signal, e.g. by using `pkill`:

```
pkill mopidy
```

1.3.2 Init scripts

- The `mopidy` package at apt.mopidy.com comes with an sysvinit init script. For more details, see the *Debian package* section of the docs.
- The `mopidy` package in Arch Linux AUR comes with a systemd init script.
- A blog post by Benjamin Guillet explains how to Daemonize Mopidy and Launch It at Login on OS X.
- Issue #266 contains a bunch of init scripts for Mopidy, including Upstart init scripts.

1.4 Troubleshooting

If you run into problems with Mopidy, we usually hang around at `#mopidy` at irc.freenode.net and also have a mailing list at Google Groups. If you stumble into a bug or have a feature request, please create an issue in the issue tracker.

When you're debugging yourself or asking for help, there are some tools built into Mopidy that you should know about.

1.4.1 Sharing config and log output

If you're getting help at IRC, we recommend that you use a pastebin, like pastebin.com or GitHub Gist, to share your configuration and log output. Pasting more than a couple of lines on IRC is generally frowned upon. On the mailing list or when reporting an issue, somewhat longer text dumps are accepted, but large logs should still be shared through a pastebin.

1.4.2 Show effective configuration

The command `mopidy config` will print your full effective configuration the way Mopidy sees it after all defaults and all config files have been merged into a single config document. Any secret values like passwords are masked out, so the output of the command should be safe to share with others for debugging.

1.4.3 Show installed dependencies

The command `mopidy deps` will list the paths to and versions of any dependency Mopidy or the extensions might need to work. This is very useful data for checking that you're using the right versions, and that you're using the right installation if you have multiple installations of a dependency on your system.

1.4.4 Debug logging

If you run *`mopidy -v`* or `mopidy -vv` or `mopidy -vvv` Mopidy will print more and more debug log to stdout. All three options will give you debug level output from Mopidy and extensions, while `-vv` and `-vvv` will give you more log output from their dependencies as well.

If you run *`mopidy --save-debug-log`*, it will save the log equivalent with `-vvv` to the file `mopidy.log` in the directory you ran the command from.

If you want to reduce the logging for some component, see the docs for the *`loglevels/*`* config section.

1.4.5 Debugging deadlocks

If Mopidy hangs without an obvious explanation, you can send the `SIGUSR1` signal to the Mopidy process. If Mopidy's main thread is still responsive, it will log a traceback for each running thread, showing what the threads are currently doing. This is a very useful tool for understanding exactly how the system is deadlocking. If you have the `pkill` command installed, you can use this by simply running:

```
pkill -SIGUSR1 mopidy
```

1.4.6 Debugging GStreamer

If you really want to dig in and debug GStreamer behaviour, then check out the Debugging section of GStreamer's documentation for your options. Note that Mopidy does not support the GStreamer command line options, like `--gst-debug-level=3`, but setting GStreamer environment variables, like `GST_DEBUG`, works with Mopidy. For example, to run Mopidy with debug logging and GStreamer logging at level 3, you can run:

```
GST_DEBUG=3 mopidy -v
```

This will produce a lot of output, but given some GStreamer knowledge this is very useful for debugging GStreamer pipeline issues.

1.5 Debian package

The Mopidy Debian package is available from apt.mopidy.com as well as from Debian, Ubuntu and other Debian-based Linux distributions.

1.5.1 Installation

See *Debian/Ubuntu: Install from apt.mopidy.com*.

1.5.2 Running as a system service

The Debian package comes with an init script. It starts Mopidy as a system service running as the `mopidy` user, which is created by the package.

The Debian package version 0.18.3-1 and older starts Mopidy as a system service by default. Version 0.18.3-2 and newer asks if you want to run Mopidy as a system service, defaulting to not doing so.

If you're running 0.18.3-2 or newer, and you've changed your mind about whether or not to run Mopidy as a system service, just run the following command to reconfigure the package:

```
sudo dpkg-reconfigure mopidy
```

If you're running 0.18.3-1 or older, and don't want to use the init script to run Mopidy as a system service, but instead just run Mopidy manually using your own user, you need to disable the init script and stop Mopidy by running:

```
sudo update-rc.d mopidy disable
sudo service mopidy stop
```

This way of disabling the system service is compatible with the improved 0.18.3-2 or newer version of the Debian package, so if you later upgrade to a newer version, you can change your mind using the `dpkg-reconfigure` command above.

1.5.3 Differences when running as a system service

If you want to run Mopidy using the init script, there's a few differences from a regular Mopidy setup you'll want to know about.

- All configuration is in `/etc/mopidy`, not in your user's home directory. The main configuration file is `/etc/mopidy/mopidy.conf`. You can do all your changes in this file.
- Mopidy extensions installed from Debian packages will sometimes install additional configuration files in `/usr/share/mopidy/conf.d/`. These files just provide different defaults for the extension when run as a system service. You can override anything from `/usr/share/mopidy/conf.d/` in the `/etc/mopidy/mopidy.conf` configuration file.

  Previously, the extension's default config was installed in `/etc/mopidy/extensions.d/`. This was removed with the Debian package mopidy 0.19.4-3. If you have modified any files in `/etc/mopidy/extensions.d/`, you should redo your modifications in `/etc/mopidy/mopidy.conf` and delete the `/etc/mopidy/extensions.d/` directory.
- The init script runs Mopidy as the `mopidy` user. The `mopidy` user will need read access to any local music you want Mopidy to play.
- To run Mopidy subcommands with the same user and config files as the init script uses, you can use `sudo mopidyctl <subcommand>`. In other words, where you'll usually run:

  ```
  mopidy config
  ```

  You should instead run the following to inspect the system service's configuration:

  ```
  sudo mopidyctl config
  ```

  The same applies to scanning your local music collection. Where you'll normally run:

  ```
  mopidy local scan
  ```

  You should instead run:

```
sudo mopidyctl local scan
```

Previously, you used `sudo service mopidy run <subcommand>` instead of `mopidyctl`. This was deprecated in Debian package version 0.19.4-3 in favor of `mopidyctl`, which also work for systems using systemd instead of sysvinit and traditional init scripts.

- Mopidy is started, stopped, and restarted just like any other system service:

```
sudo service mopidy start
sudo service mopidy stop
sudo service mopidy restart
```

- You can check if Mopidy is currently running as a system service by running:

```
sudo service mopidy status
```

- Mopidy installed from a Debian package can use both Mopidy extensions installed both from Debian packages and extensions installed with pip.

  The other way around does not work: Mopidy installed with pip can use extensions installed with pip, but not extensions installed from a Debian package. This is because the Debian packages install extensions into `/usr/share/mopidy` which is normally not on your `PYTHONPATH`. Thus, your pip-installed Mopidy will not find the Debian package-installed extensions.

CHAPTER 2

Extensions

2.1 Mopidy-Local

Mopidy-Local is an extension for playing music from your local music archive. It is bundled with Mopidy and enabled by default. Though, you'll have to scan your music collection to build a cache of metadata before the Mopidy-Local will be able to play your music.

This backend handles URIs starting with `local:`.

2.1.1 Generating a local library

The command **`mopidy local scan`** will scan the path set in the *`local/media_dir`* config value for any audio files and build a library of metadata.

To make a local library for your music available for Mopidy:

1. Ensure that the *`local/media_dir`* config value points to where your music is located. Check the current setting by running:

   ```
   mopidy config
   ```

2. Scan your media library.:

   ```
   mopidy local scan
   ```

3. Start Mopidy, find the music library in a client, and play some local music!

2.1.2 Pluggable library support

Local libraries are fully pluggable. What this means is that users may opt to disable the current default library `json`, replacing it with a third party one. When running **`mopidy local scan`** Mopidy will populate whatever the current active library is with data. Only one library may be active at a time.

To create a new library provider you must create class that implements the *mopidy.local.Library* interface and install it in the extension registry under `local:library`. Any data that the library needs to store on disc should be stored in *local/data_dir* using the library name as part of the filename or directory to avoid any conflicts.

2.1.3 Configuration

See *Configuration* for general help on configuring Mopidy.

```
[local]
enabled = true
library = json
media_dir = $XDG_MUSIC_DIR
data_dir = $XDG_DATA_DIR/mopidy/local
playlists_dir = $XDG_DATA_DIR/mopidy/local/playlists
scan_timeout = 1000
scan_flush_threshold = 1000
excluded_file_extensions =
  .directory
  .html
  .jpeg
  .jpg
  .log
  .nfo
  .png
  .txt
```

local/enabled
: If the local extension should be enabled or not.

local/library
: Local library provider to use, change this if you want to use a third party library for local files.

local/media_dir
: Path to directory with local media files.

local/data_dir
: Path to directory to store local metadata such as libraries and playlists in.

local/playlists_dir
: Path to playlists directory with m3u files for local media.

local/scan_timeout
: Number of milliseconds before giving up scanning a file and moving on to the next file.

local/scan_flush_threshold
: Number of tracks to wait before telling library it should try and store its progress so far. Some libraries might not respect this setting. Set this to zero to disable flushing.

local/excluded_file_extensions
: File extensions to exclude when scanning the media directory. Values should be separated by either comma or newline.

2.2 Mopidy-Stream

Mopidy-Stream is an extension for playing streaming music. It is bundled with Mopidy and enabled by default.

This backend does not provide a library or playlist storage. It simply accepts any URI added to Mopidy's tracklist that matches any of the protocols in the *stream/protocols* config value. It then tries to retrieve metadata and play back the URI using GStreamer. For example, if you're using an MPD client, you'll just have to find your clients "add URI" interface, and provide it with the URI of a stream.

In addition to playing streams, the extension also understands how to extract streams from a lot of playlist formats. This is convenient as most Internet radio stations links to playlists instead of directly to the radio streams.

If you're having trouble playing back a stream, run the `mopidy deps` command to check if you have all relevant GStreamer plugins installed.

2.2.1 Configuration

See *Configuration* for general help on configuring Mopidy.

```
[stream]
enabled = true
protocols =
    file
    http
    https
    mms
    rtmp
    rtmps
    rtsp
timeout = 5000
metadata_blacklist =
```

stream/enabled
If the stream extension should be enabled or not.

stream/protocols
Whitelist of URI schemas to allow streaming from. Values should be separated by either comma or newline.

stream/timeout
Number of milliseconds before giving up looking up stream metadata.

stream/metadata_blacklist
List of URI globs to not fetch metadata from before playing. This feature is typically needed for play once URIs provided by certain streaming providers. Regular POSIX glob semantics apply, so `http://*.example.com/*` would match all example.com sub-domains.

2.3 Mopidy-HTTP

Mopidy-HTTP is an extension that lets you control Mopidy through HTTP and WebSockets, for example from a web client. It is bundled with Mopidy and enabled by default.

When it is enabled it starts a web server at the port specified by the *http/port* config value.

Warning: As a simple security measure, the web server is by default only available from localhost. To make it available from other computers, change the *http/hostname* config value. Before you do so, note that the HTTP extension does not feature any form of user authentication or authorization. Anyone able to access the web server can use the full core API of Mopidy. Thus, you probably only want to make the web server available from your local network or place it behind a web proxy which takes care or user authentication. You have been warned.

2.3.1 Hosting web clients

Mopidy-HTTP's web server can also host Tornado apps or any static files, for example the HTML, CSS, JavaScript, and images needed for a web based Mopidy client. See *HTTP server side API* for how to make static files or server-side functionality from a Mopidy extension available through Mopidy's web server.

If you're making a web based client and want to do server side development using some other technology than Tornado, you are of course free to run your own web server and just use Mopidy's web server to host the API endpoints. But, for clients implemented purely in JavaScript, letting Mopidy host the files is a simpler solution.

See *HTTP JSON-RPC API* for details on how to integrate with Mopidy over HTTP. If you're looking for a web based client for Mopidy, go check out *HTTP clients*.

2.3.2 Configuration

See *Configuration* for general help on configuring Mopidy.

```
[http]
enabled = true
hostname = 127.0.0.1
port = 6680
static_dir =
zeroconf = Mopidy HTTP server on $hostname
```

http/enabled
If the HTTP extension should be enabled or not.

http/hostname
Which address the HTTP server should bind to.

`127.0.0.1` Listens only on the IPv4 loopback interface

`::1` Listens only on the IPv6 loopback interface

`0.0.0.0` Listens on all IPv4 interfaces

`::` Listens on all interfaces, both IPv4 and IPv6

http/port
Which TCP port the HTTP server should listen to.

http/static_dir
Which directory the HTTP server should serve at "/"

Change this to have Mopidy serve e.g. files for your JavaScript client. "/mopidy" will continue to work as usual even if you change this setting.

This config value isn't deprecated yet, but you're strongly encouraged to make Mopidy extensions which use the the *HTTP server side API* to host static files on Mopidy's web server instead of using *http/static_dir*. That way, installation of your web client will be a lot easier for your end users, and multiple web clients can easily share the same web server.

http/zeroconf
Name of the HTTP service when published through Zeroconf. The variables `$hostname` and `$port` can be used in the name.

If set, the Zeroconf services `_http._tcp` and `_mopidy-http._tcp` will be published.

Set to an empty string to disable Zeroconf for HTTP.

2.4 Mopidy-MPD

Mopidy-MPD is an extension that provides a full MPD server implementation to make Mopidy available to *MPD clients*. It is bundled with Mopidy and enabled by default.

Warning: As a simple security measure, the MPD server is by default only available from localhost. To make it available from other computers, change the `mpd/hostname` config value. Before you do so, note that the MPD server does not support any form of encryption and only a single clear text password (see `mpd/password`) for weak authentication. Anyone able to access the MPD server can control music playback on your computer. Thus, you probably only want to make the MPD server available from your local network. You have been warned.

MPD stands for Music Player Daemon, which is also the name of the original MPD server project. Mopidy does not depend on the original MPD server, but implements the MPD protocol itself, and is thus compatible with clients for the original MPD server.

For more details on our MPD server implementation, see `mopidy.mpd`.

2.4.1 Limitations

This is a non exhaustive list of MPD features that Mopidy doesn't support. Items on this list will probably not be supported in the near future.

- Only a single password is supported. It gives all-or-nothing access.
- Toggling of audio outputs is not supported
- Channels for client-to-client communication are not supported
- Stickers are not supported
- Crossfade is not supported
- Replay gain is not supported
- `stats` does not provide any statistics
- `decoders` does not provide information about available decoders

The following items are currently not supported, but should be added in the near future:

- Modifying stored playlists is not supported
- `tagtypes` is not supported
- Live update of the music database is not supported

2.4.2 Configuration

See *Configuration* for general help on configuring Mopidy.

```
[mpd]
enabled = true
hostname = 127.0.0.1
port = 6600
password =
max_connections = 20
```

(continues on next page)

(continued from previous page)

```
connection_timeout = 60
zeroconf = Mopidy MPD server on $hostname
```

mpd/enabled
If the MPD extension should be enabled or not.

mpd/hostname
Which address the MPD server should bind to.

`127.0.0.1` Listens only on the IPv4 loopback interface

`::1` Listens only on the IPv6 loopback interface

`0.0.0.0` Listens on all IPv4 interfaces

`::` Listens on all interfaces, both IPv4 and IPv6

mpd/port
Which TCP port the MPD server should listen to.

mpd/password
The password required for connecting to the MPD server. If blank, no password is required.

mpd/max_connections
The maximum number of concurrent connections the MPD server will accept.

mpd/connection_timeout
Number of seconds an MPD client can stay inactive before the connection is closed by the server.

mpd/zeroconf
Name of the MPD service when published through Zeroconf. The variables `$hostname` and `$port` can be used in the name.

Set to an empty string to disable Zeroconf for MPD.

2.5 Mopidy-SoftwareMixer

Mopidy-SoftwareMixer is an extension for controlling audio volume in software through GStreamer. It is the only mixer bundled with Mopidy and is enabled by default.

If you use PulseAudio, the software mixer will control the per-application volume for Mopidy in PulseAudio, and any changes to the per-application volume done from outside Mopidy will be reflected by the software mixer.

If you don't use PulseAudio, the mixer will adjust the volume internally in Mopidy's GStreamer pipeline.

2.5.1 Configuration

Multiple mixers can be installed and enabled at the same time, but only the mixer pointed to by the *audio/mixer* config value will actually be used.

See *Configuration* for general help on configuring Mopidy.

```
[softwaremixer]
enabled = true
```

softwaremixer/enabled
If the software mixer should be enabled or not. Usually you don't want to change this, but instead change the *audio/mixer* config value to decide which mixer is actually used.

2.6 Mixer extensions

Here you can find a list of external packages that extend Mopidy with additional audio mixers by implementing the *Audio mixer API* which was added in Mopidy 0.19.

This list is moderated and updated on a regular basis. If you want your package to show up here, follow the *guide on creating extensions*.

2.6.1 Mopidy-ALSAMixer

https://github.com/mopidy/mopidy-alsamixer

Extension for controlling volume one a Linux system using ALSA.

2.6.2 Mopidy-Arcam

https://github.com/TooDizzy/mopidy-arcam

Extension for controlling volume using an external Arcam amplifier. Developed and tested with an Arcam AVR-300.

2.6.3 Mopidy-NAD

https://github.com/mopidy/mopidy-nad

Extension for controlling volume using an external NAD amplifier. Developed and tested with a NAD C355BEE.

2.6.4 Mopidy-SoftwareMixer

Bundled with Mopidy. See *Mopidy-SoftwareMixer*.

2.6.5 Mopidy-Yamaha

https://github.com/knutz3n/mopidy-yamaha

Extension for controlling volume using an external Yamaha network connected amplifier.

2.7 Backend extensions

Here you can find a list of external packages that extend Mopidy with additional music sources by implementing the *Backend API*.

This list is moderated and updated on a regular basis. If you want your package to show up here, follow the *guide on creating extensions*.

2.7.1 Mopidy-AudioAddict

https://github.com/nilicule/mopidy-audioaddict

Provides a backend for playing music from the AudioAddict network of sites, including Digitally Imported, RadioTunes, RockRadio, JazzRadio, and FrescaRadio.

2.7.2 Mopidy-Banshee

https://github.com/tamland/mopidy-banshee

Provides a backend for playing music from the Banshee music player's music library.

2.7.3 Mopidy-Bassdrive

https://github.com/felixb/mopidy-Bassdrive

Provides a backend for playing radio streams from BassDrive.

2.7.4 Mopidy-Beets

https://github.com/mopidy/mopidy-beets

Provides a backend for playing music from your Beets music library through Beets' web extension.

2.7.5 Mopidy-Dirble

https://github.com/mopidy/mopidy-dirble

Provides a backend for browsing the Internet radio channels from the Dirble directory.

2.7.6 Mopidy-GMusic

https://github.com/hechtus/mopidy-gmusic

Provides a backend for playing music from Google Play Music.

2.7.7 Mopidy-InternetArchive

https://github.com/tkem/mopidy-internetarchive

Extension for playing music and audio from the Internet Archive.

2.7.8 Mopidy-LeftAsRain

https://github.com/naglis/mopidy-leftasrain

Extension for playing music from the leftasrain.com music blog.

2.7.9 Mopidy-Local

Bundled with Mopidy. See *Mopidy-Local*.

2.7.10 Mopidy-Local-SQLite

https://github.com/tkem/mopidy-local-sqlite

Extension which plugs into Mopidy-Local to use an SQLite database to keep track of your local media. This extension lets you browse your music collection by album, artist, composer and performer, and provides full-text search capabilities based on SQLite's FTS modules. It also notices updates via `mopidy local scan` while Mopidy is running, so you can scan your media library periodically from a cron job, for example.

2.7.11 Mopidy-OE1

https://github.com/tischlda/mopidy-oe1

Extension for playing the live stream and browsing the 7-day archive of the Austrian radio station OE1.

2.7.12 Mopidy-Podcast

https://github.com/tkem/mopidy-podcast

Extension for browsing RSS feeds of podcasts and stream the episodes.

2.7.13 Mopidy-Podcast-gpodder.net

https://github.com/tkem/mopidy-podcast-gpodder

Extension for Mopidy-Podcast that lets you search and browse podcasts from the gpodder.net web site.

2.7.14 Mopidy-Podcast-iTunes

https://github.com/tkem/mopidy-podcast-itunes

Extension for Mopidy-Podcast that lets you search and browse podcasts from the Apple iTunes Store.

2.7.15 Mopidy-radio-de

https://github.com/hechtus/mopidy-radio-de

Extension for listening to Internet radio stations and podcasts listed at radio.de, rad.io, radio.fr, and radio.at.

2.7.16 Mopidy-SomaFM

https://github.com/AlexandrePTJ/mopidy-somafm

Provides a backend for playing music from the SomaFM service.

2.7.17 Mopidy-SoundCloud

https://github.com/mopidy/mopidy-soundcloud

Provides a backend for playing music from the SoundCloud service.

2.7.18 Mopidy-Spotify

https://github.com/mopidy/mopidy-spotify

Extension for playing music from the Spotify music streaming service.

2.7.19 Mopidy-Spotify-Tunigo

https://github.com/trygveaa/mopidy-spotify-tunigo

Extension for providing the browse feature of Spotify. This lets you browse playlists, genres and new releases.

2.7.20 Mopidy-Stream

Bundled with Mopidy. See *Mopidy-Stream*.

2.7.21 Mopidy-Subsonic

https://github.com/rattboi/mopidy-subsonic

Provides a backend for playing music from a Subsonic Music Streamer library.

2.7.22 Mopidy-TuneIn

https://github.com/kingosticks/mopidy-tunein

Provides a backend for playing music from the TuneIn online radio service.

2.7.23 Mopidy-VKontakte

https://github.com/sibuser/mopidy-vkontakte

Provides a backend for playing music from the VKontakte social network.

2.7.24 Mopidy-YouTube

https://github.com/dz0ny/mopidy-youtube

Provides a backend for playing music from the YouTube service.

2.8 Frontend extensions

Here you can find a list of external packages that extend Mopidy with additional frontends, which includes just about anything that use the *Core API*.

This list is moderated and updated on a regular basis. If you want your package to show up here, follow the *guide on creating extensions*.

2.8.1 Mopidy-EvtDev

https://github.com/liamw9534/mopidy-evtdev

Extension for controll Mopidy from virtual input devices.

2.8.2 Mopidy-HTTP

Bundled with Mopidy. See *Mopidy-HTTP*.

2.8.3 Mopidy-MPD

Bundled with Mopidy. See *Mopidy-MPD*.

2.8.4 Mopidy-MPRIS

https://github.com/mopidy/mopidy-mpris

Extension for controlling Mopidy through the MPRIS D-Bus interface, for example using the Ubuntu Sound Menu.

2.8.5 Mopidy-Notifier

https://github.com/sauberfred/mopidy-notifier

Extension for displaying track info as User Notifications in Mac OS X.

2.8.6 Mopidy-Scrobbler

https://github.com/mopidy/mopidy-scrobbler

Extension for scrobbling played tracks to Last.fm.

2.8.7 Mopidy-Touchscreen

https://github.com/9and3r/mopidy-touchscreen

Extension for displaying track info and controlling Mopidy from a touch screen using PyGame/SDL.

2.8.8 Mopidy-TtsGpio

https://github.com/9and3r/mopidy-ttsgpio

Extension for controlling Mopidy without a display by using e.g. buttons connected to GPIO and text-to-speech for track information.

2.8.9 Mopidy-Webhooks

https://github.com/paddycarey/mopidy-webhooks

Extension for sending HTTP POST requests with JSON payloads to a remote server on when Mopidy core triggers an event and on regular intervals.

2.9 Web extensions

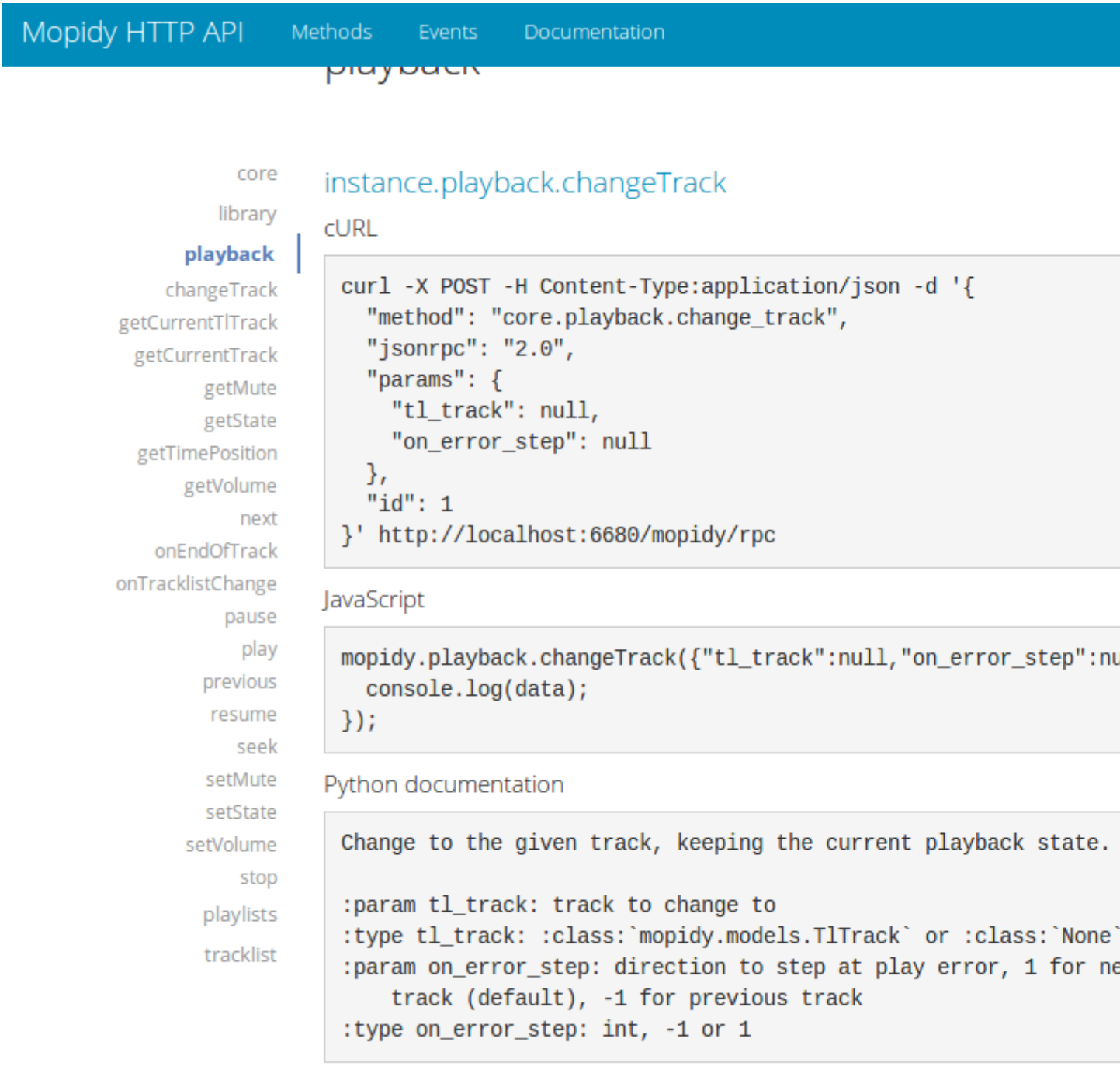
Here you can find a list of external packages that extend Mopidy with additional web interfaces by implementing the *HTTP server side API*, which was added in Mopidy 0.19, and optionally using the *HTTP JSON-RPC API*.

This list is moderated and updated on a regular basis. If you want your package to show up here, follow the *guide on creating extensions*.

2.9.1 Mopidy-API-Explorer

https://github.com/dz0ny/mopidy-api-explorer

Web extension for browsing the Mopidy HTTP API.



To install, run:

```
pip install Mopidy-API-Explorer
```

2.9.2 Mopidy-HTTP-Kuechenradio

https://github.com/tkem/mopidy-http-kuechenradio

A deliberately simple Mopidy Web client for mobile devices. Made with jQuery Mobile by Thomas Kemmer.

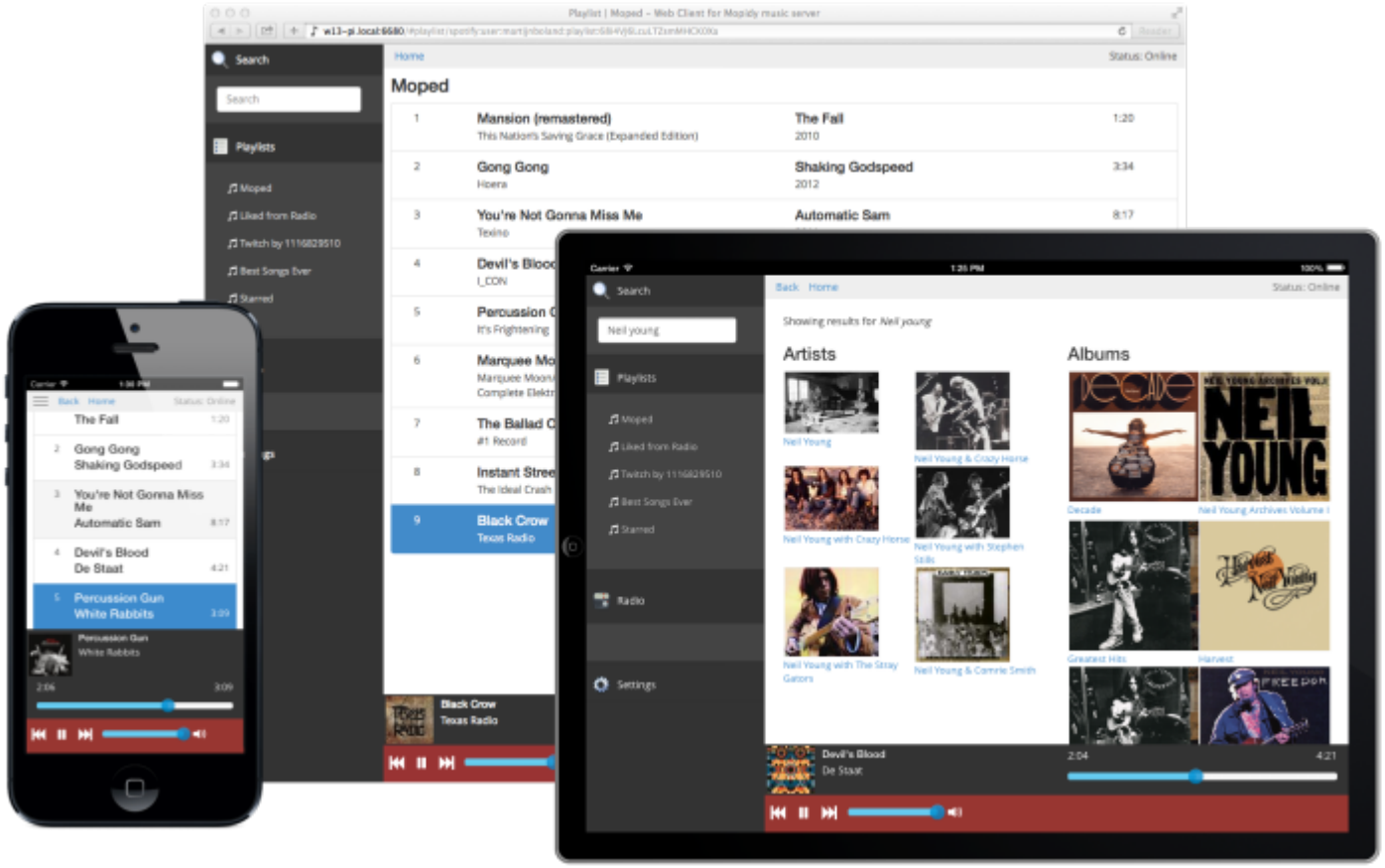
To install, run:

```
pip install Mopidy-HTTP-Kuechenradio
```

2.9.3 Mopidy-Moped

https://github.com/martijnboland/moped

A Mopidy web client made with AngularJS by Martijn Boland.



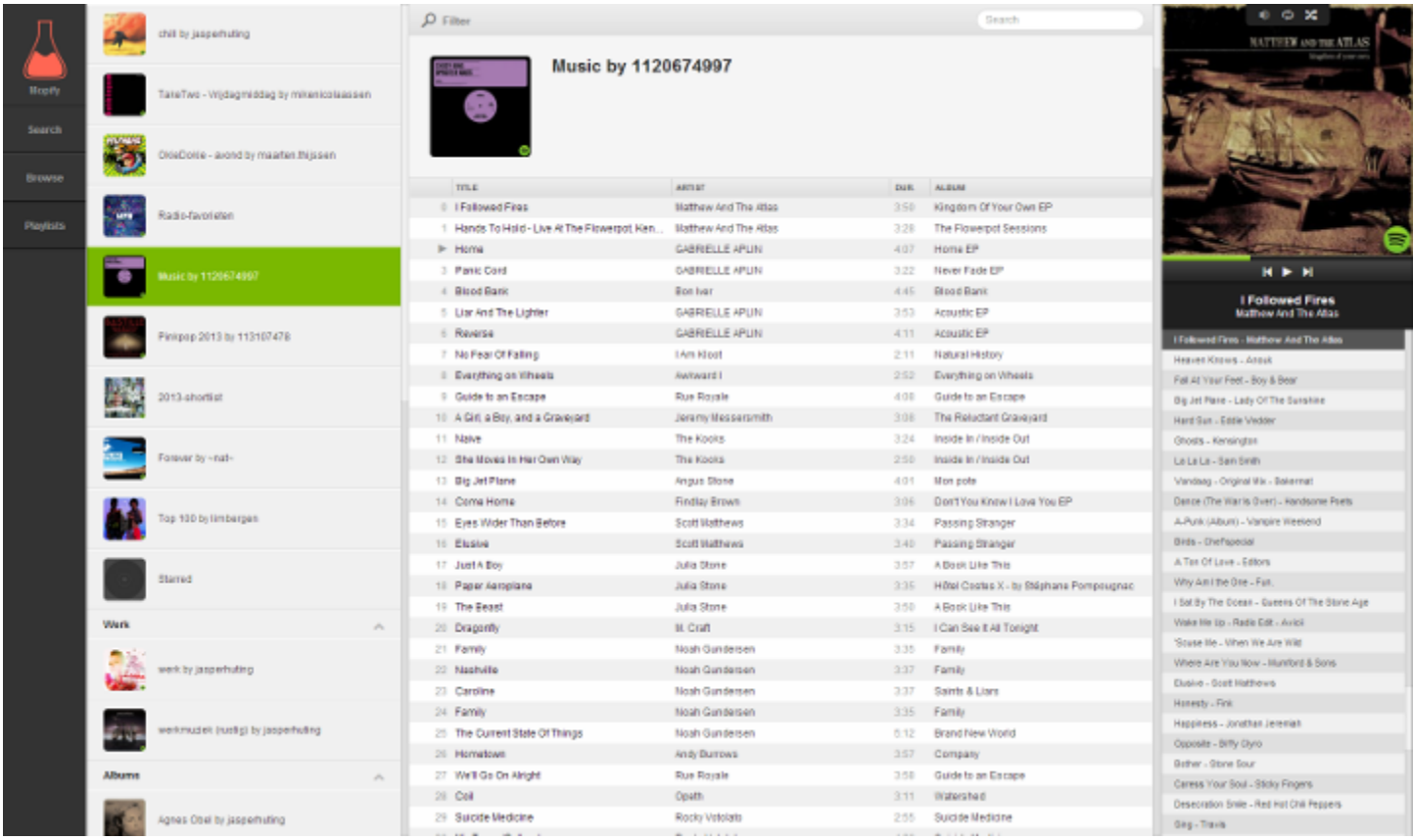
To install, run:

```
pip install Mopidy-Moped
```

2.9.4 Mopidy-Mopify

https://github.com/dirkgroenen/mopidy-mopify

An web client that mainly targets using Spotify through Mopidy. Made by Dirk Groenen.



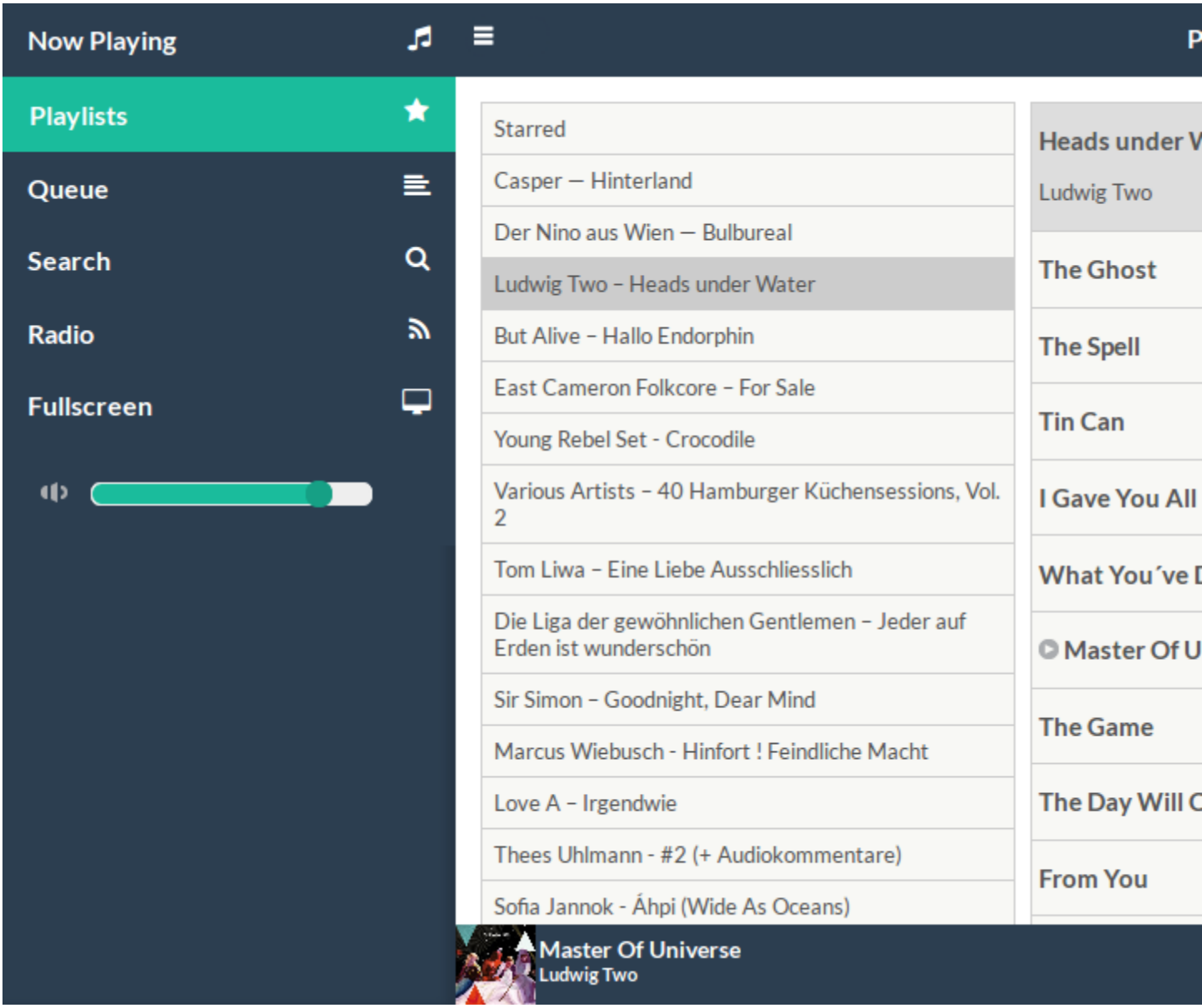
To install, run:

```
pip install Mopidy-Mopify
```

2.9.5 Mopidy-MusicBox-Webclient

https://github.com/woutervanwijk/Mopidy-MusicBox-Webclient

The first web client for Mopidy, made with jQuery Mobile by Wouter van Wijk. Also the web client used for Wouter's popular Pi Musicbox image for Raspberry Pi.



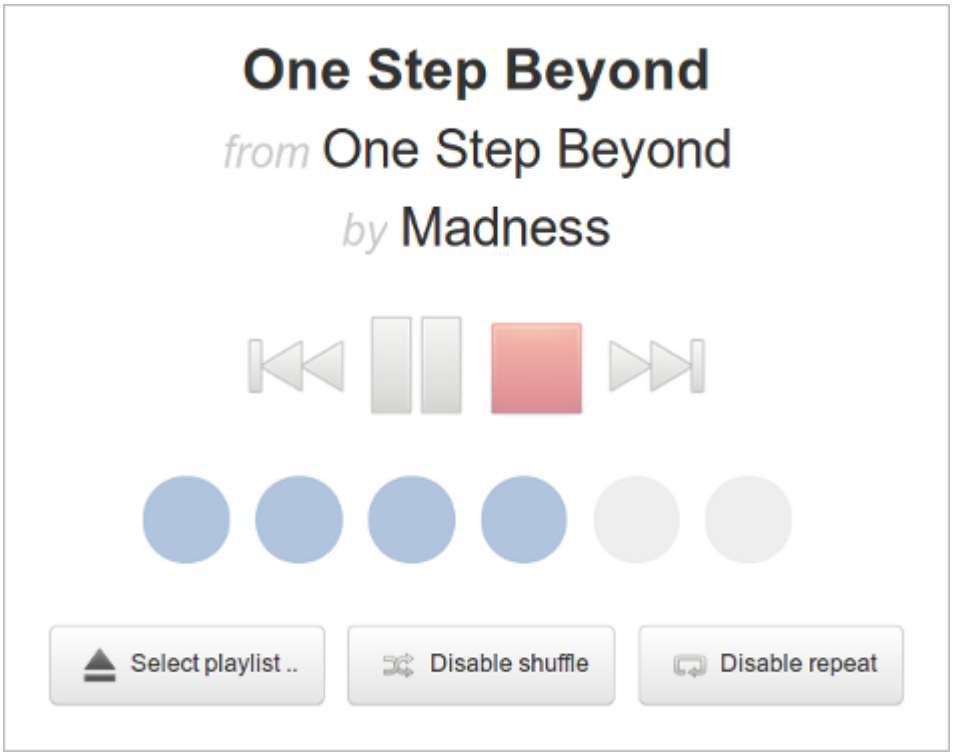
To install, run:

```
pip install Mopidy-MusicBox-Webclient
```

2.9.6 Mopidy-Simple-Webclient

https://github.com/xolox/mopidy-simple-webclient

A minimalistic web client targeted for mobile devices. Made with jQuery and Bootstrap by Peter Odding.



To install, run:

```
pip install Mopidy-Simple-Webclient
```

2.9.7 Mopidy-WebSettings

https://github.com/woutervanwijk/mopidy-websettings

A web extension for changing settings. Used by the Pi MusicBox distribution for Raspberry Pi, but also usable for other projects.

2.9.8 Other web clients

There's also some other web clients for Mopidy that use the *HTTP JSON-RPC API*, but isn't installable using `pip`:

- Apollo Player
- JukePi

In addition, there's several web based MPD clients, which doesn't use the *Mopidy-HTTP* frontend at all, but connect to Mopidy through our *Mopidy-MPD* frontend. For a list of those, see *Web clients*.

CHAPTER 3

Clients

3.1 HTTP clients

See *Web extensions*.

3.2 MPD clients

This is a list of MPD clients we either know works well with Mopidy, or that we know won't work well. For a more exhaustive list of MPD clients, see http://mpd.wikia.com/wiki/Clients.

Contents

3.2.1 Test procedure

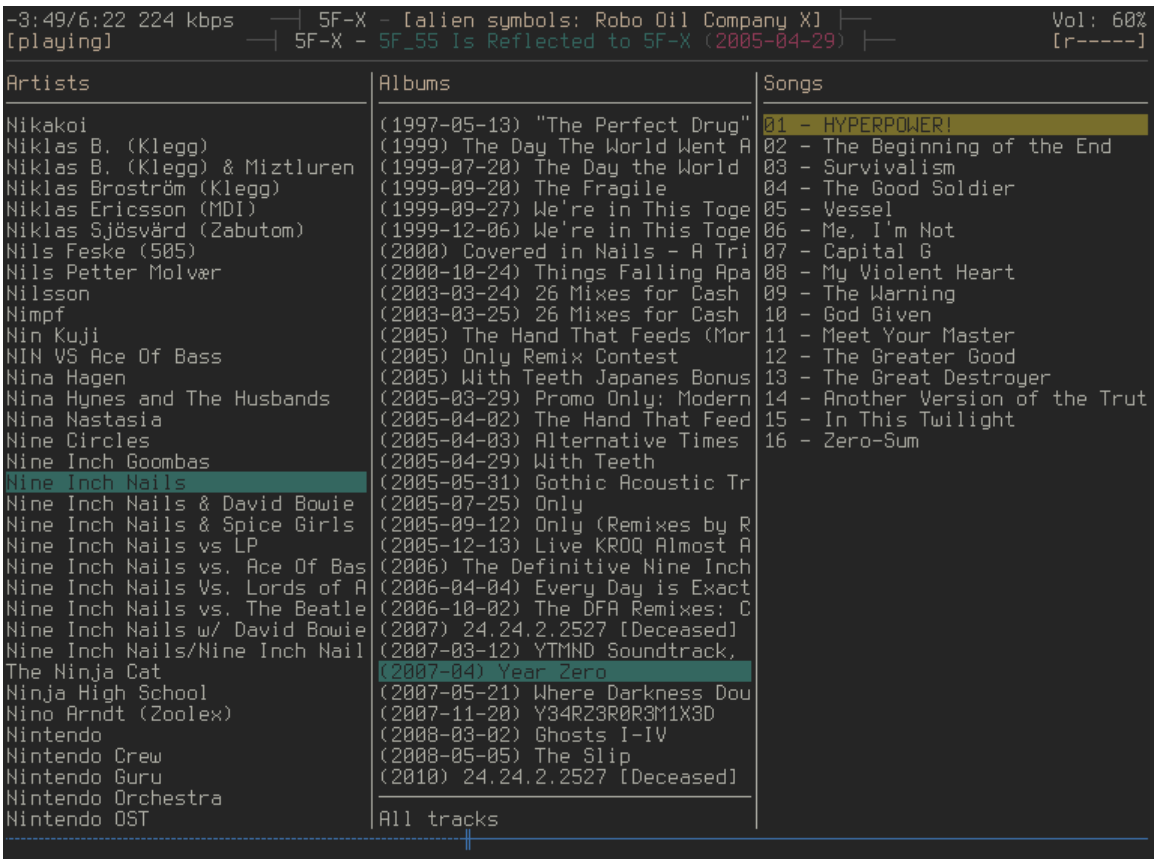
In some cases, we've used the following test procedure to compare the feature completeness of clients:

1. Connect to Mopidy
2. Search for "foo", with search type "any" if it can be selected
3. Add "The Pretender" from the search results to the current playlist
4. Start playback
5. Pause and resume playback
6. Adjust volume
7. Find a playlist and append it to the current playlist
8. Skip to next track
9. Skip to previous track
10. Select the last track from the current playlist
11. Turn on repeat mode
12. Seek to 10 seconds or so before the end of the track
13. Wait for the end of the track and confirm that playback continues at the start of the playlist
14. Turn off repeat mode
15. Turn on random mode
16. Skip to next track and confirm that it random mode works
17. Turn off random mode
18. Stop playback
19. Check if the app got support for single mode and consume mode
20. Kill Mopidy and confirm that the app handles it without crashing

3.2.2 Console clients

ncmpcpp

A console client that works well with Mopidy, and is regularly used by Mopidy developers.



Search does not work in the "Match if tag contains search phrase (regexes supported)" mode because the client tries to fetch all known metadata and do the search on the client side. The two other search modes works nicely, so this is not a problem.

The library view is very slow when used together with Mopidy-Spotify. A workaround is to edit the ncmpcpp configuration file (`~/.ncmpcpp/config`) and set:

```
media_library_display_date = "no"
```

With this change ncmpcpp's library view will still be a bit slow, but usable.

ncmpc

A console client. Works with Mopidy 0.6 and upwards. Uses the `idle` MPD command, but in a resource inefficient way.

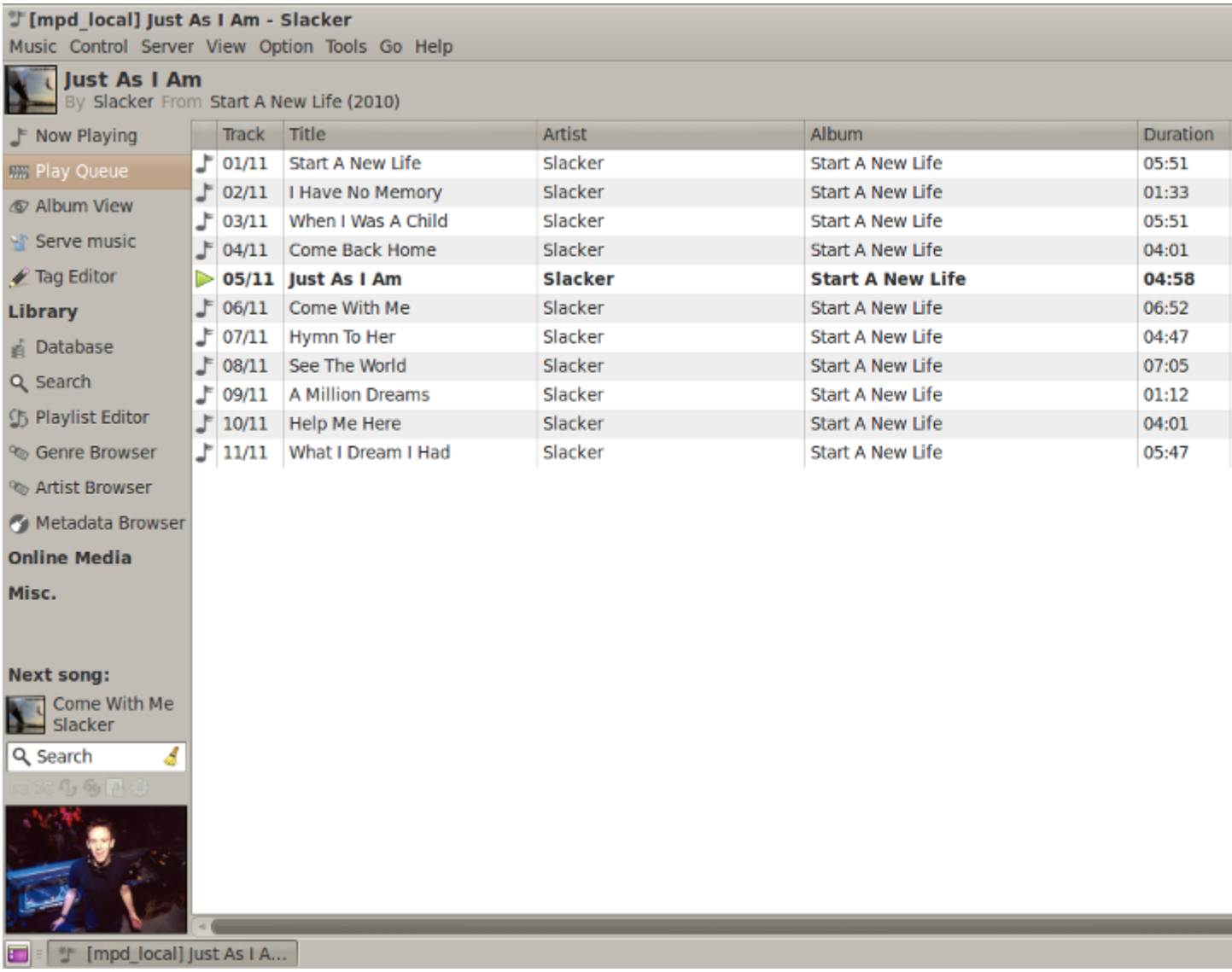
mpc

A command line client. Version 0.16 and upwards seems to work nicely with Mopidy.

3.2.3 Graphical clients

GMPC

GMPC is a graphical MPD client (GTK+) which works well with Mopidy.



GMPC may sometimes requests a lot of meta data of related albums, artists, etc. This takes more time with Mopidy, which needs to query Spotify for the data, than with a normal MPD server, which has a local cache of meta data. Thus, GMPC may sometimes feel frozen, but usually you just need to give it a bit of slack before it will catch up.

Sonata

Sonata is a graphical MPD client (GTK+). It generally works well with Mopidy, except for search.

When you search in Sonata, it only sends the first to letters of the search query to Mopidy, and then does the rest of the filtering itself on the client side. Since Spotify has a collection of millions of tracks and they only return the first 100 hits for any search query, searching for two-letter combinations seldom returns any useful results. See #1 and the closed Sonata bug for details.

Theremin

Theremin is a graphical MPD client for OS X. It is unmaintained, but generally works well with Mopidy.

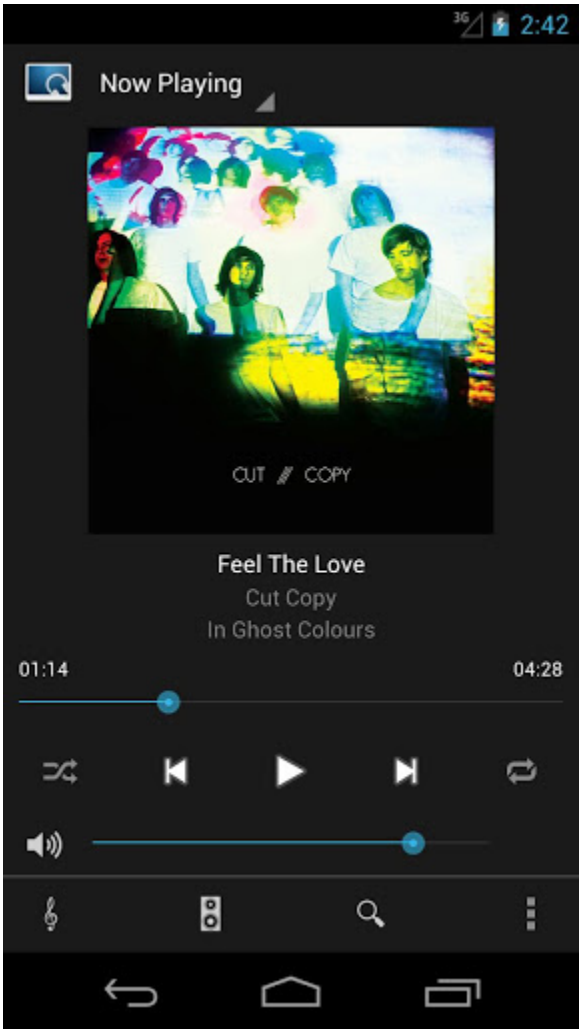
3.2.4 Android clients

We've tested all five MPD clients we could find for Android with Mopidy 0.8.1 on a Samsung Galaxy Nexus with Android 4.1.2, using our standard test procedure.

MPDroid

Test date: 2012-11-06

Tested version: 1.03.1 (released 2012-10-16)



You can get MPDroid from Google Play.

- MPDroid started out as a fork of PMix, and is now much better.
- MPDroid's user interface looks nice.
- Everything in the test procedure works.
- In contrast to all other Android clients, MPDroid does support single mode or consume mode.
- When Mopidy is killed, MPDroid handles it gracefully and asks if you want to try to reconnect.

MPDroid is a good MPD client, and really the only one we can recommend.

BitMPC

Test date: 2012-11-06

Tested version: 1.0.0 (released 2010-04-12)

You can get BitMPC from Google Play.

- The user interface lacks some finishing touches. E.g. you can't enter a hostname for the server. Only IPv4 addresses are allowed.

- When we last tested the same version of BitMPC using Android 2.1:
  - All features exercised in the test procedure worked.
  - BitMPC lacked support for single mode and consume mode.
  - BitMPC crashed if Mopidy was killed or crashed.
- When we tried to test using Android 4.1.1, BitMPC started and connected to Mopidy without problems, but the app crashed as soon as we fired off our search, and continued to crash on startup after that.

In conclusion, BitMPC is usable if you got an older Android phone and don't care about looks. For newer Android versions, BitMPC will probably not work as it hasn't been maintained for 2.5 years.

Droid MPD Client

Test date: 2012-11-06

Tested version: 1.4.0 (released 2011-12-20)

You can get Droid MPD Client from Google Play.

- No intutive way to ask the app to connect to the server after adding the server hostname to the settings.
- To find the search functionality, you have to select the menu, then "Playlist manager", then the search tab. I do not understand why search is hidden inside "Playlist manager".
- The tabs "Artists" and "Albums" did not contain anything, and did not cause any requests.
- The tab "Folders" showed a spinner and said "Updating data. . . " but did not send any requests.
- Searching for "foo" did nothing. No request was sent to the server.
- Droid MPD client does not support single mode or consume mode.
- Not able to complete the test procedure, due to the above problems.

In conclusion, not a client we can recommend.

PMix

Test date: 2012-11-06

Tested version: 0.4.0 (released 2010-03-06)

You can get PMix from Google Play.

PMix haven't been updated for 2.5 years, and has less working features than it's fork MPDroid. Ignore PMix and use MPDroid instead.

MPD Remote

Test date: 2012-11-06

Tested version: 1.0 (released 2012-05-01)

You can get MPD Remote from Google Play.

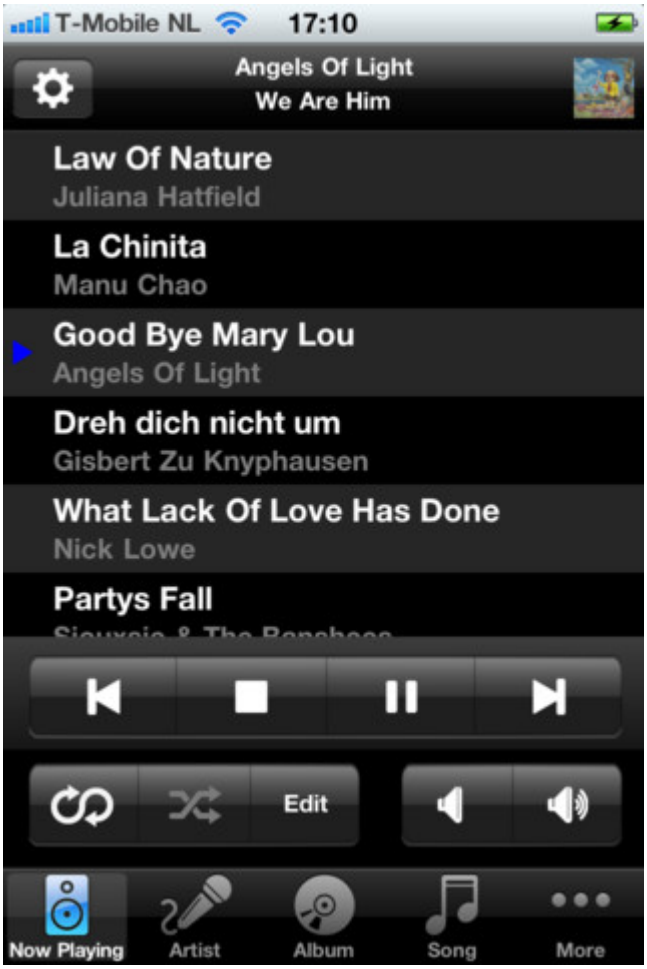
This app looks terrible in the screen shots, got just 100+ downloads, and got a terrible rating. I honestly didn't take the time to test it.

3.2.5 iOS clients

MPoD

Test date: 2012-11-06

Tested version: 1.7.1



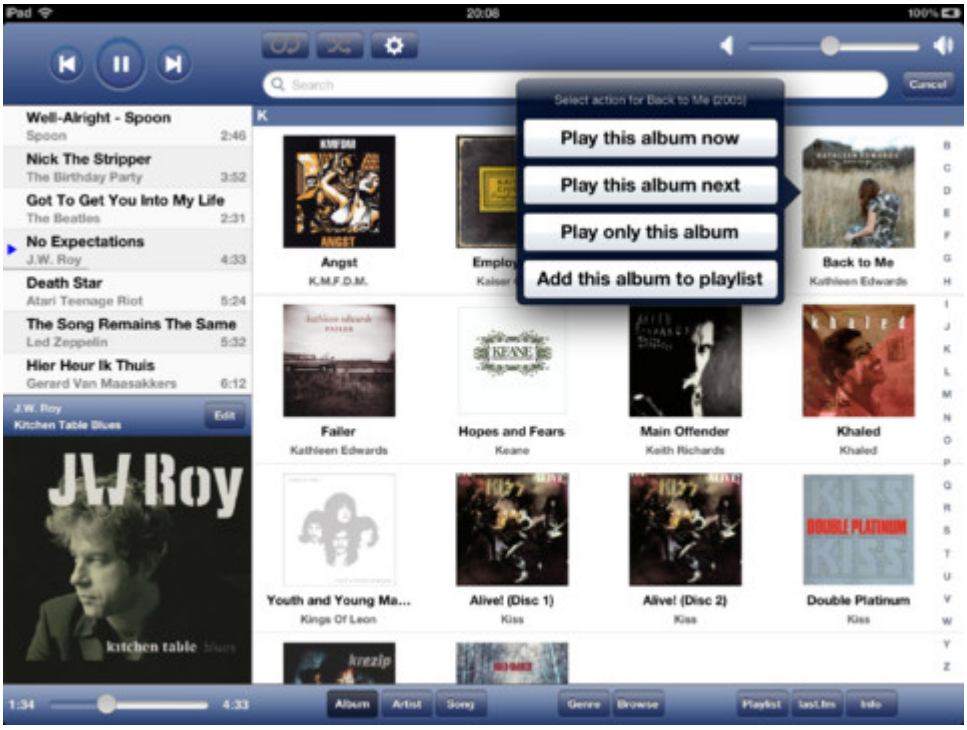
The MPoD iPhone/iPod Touch app can be installed from MPoD at iTunes Store.

- The user interface looks nice.
- All features exercised in the test procedure worked with MPaD, except seek, which I didn't figure out to do.
- Search only works in the "Browse" tab, and not under in the "Artist", "Album", or "Song" tabs. For the tabs where search doesn't work, no queries are sent to Mopidy when searching.
- Single mode and consume mode is supported.

MPaD

Test date: 2012-11-06

Tested version: 1.7.1



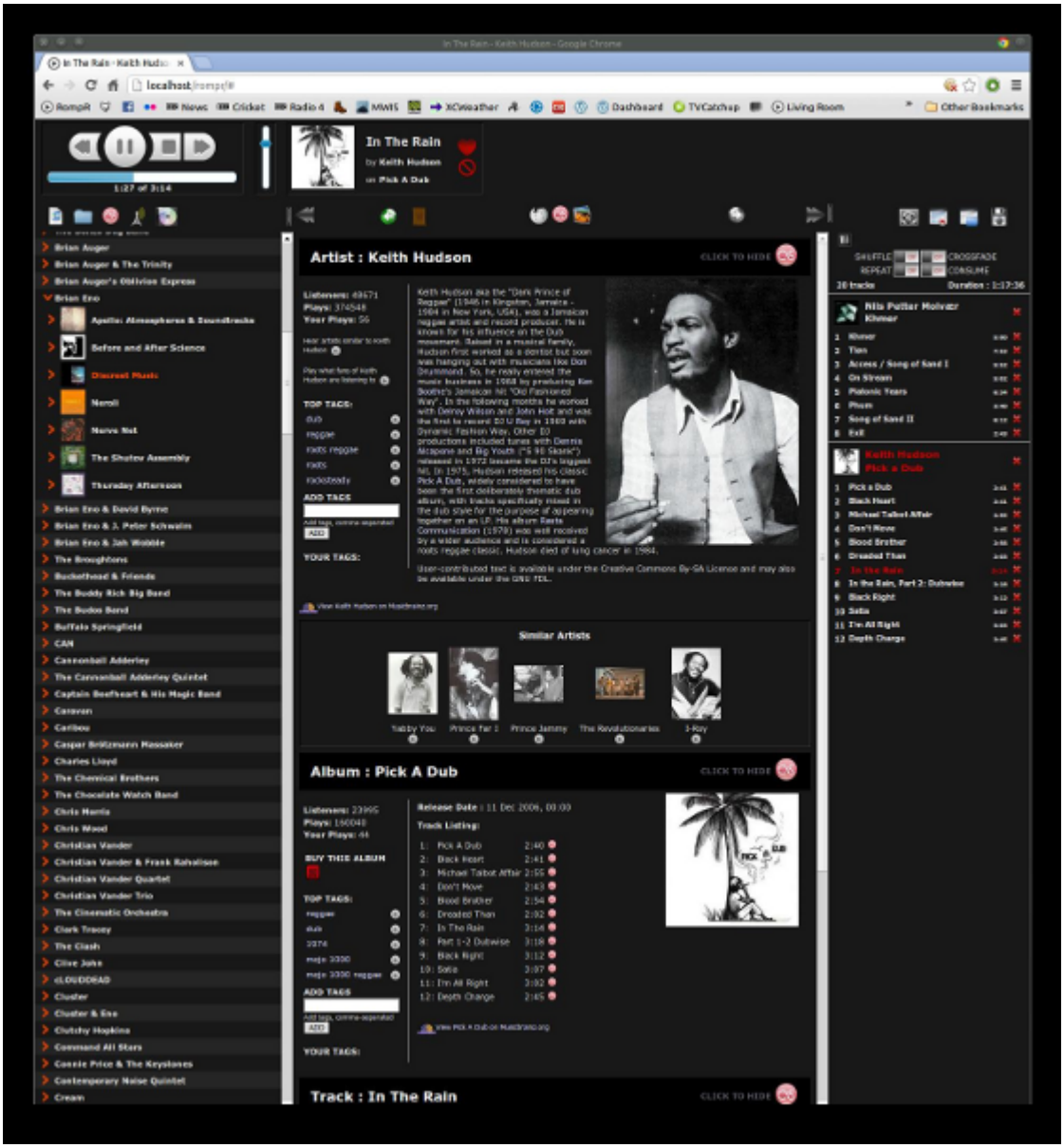
The MPaD iPad app can be purchased from MPaD at iTunes Store

- The user interface looks nice, though I would like to be able to view the current playlist in the large part of the split view.
- All features exercised in the test procedure worked with MPaD.
- Search only works in the "Browse" tab, and not under in the "Artist", "Album", or "Song" tabs. For the tabs where search doesn't work, no queries are sent to Mopidy when searching.
- Single mode and consume mode is supported.
- The server menu can be very slow top open, and there is no visible feedback when waiting for the connection to a server to succeed.

3.2.6 Web clients

The following web clients use the MPD protocol to communicate with Mopidy. For other web clients, see *HTTP clients*.

Rompr



Rompr is a web based MPD client. mrvanes, a Mopidy and Rompr user, said: "These projects are a real match made in heaven."

Partify

Partify is a web based MPD client focusing on making music playing collaborative and social.

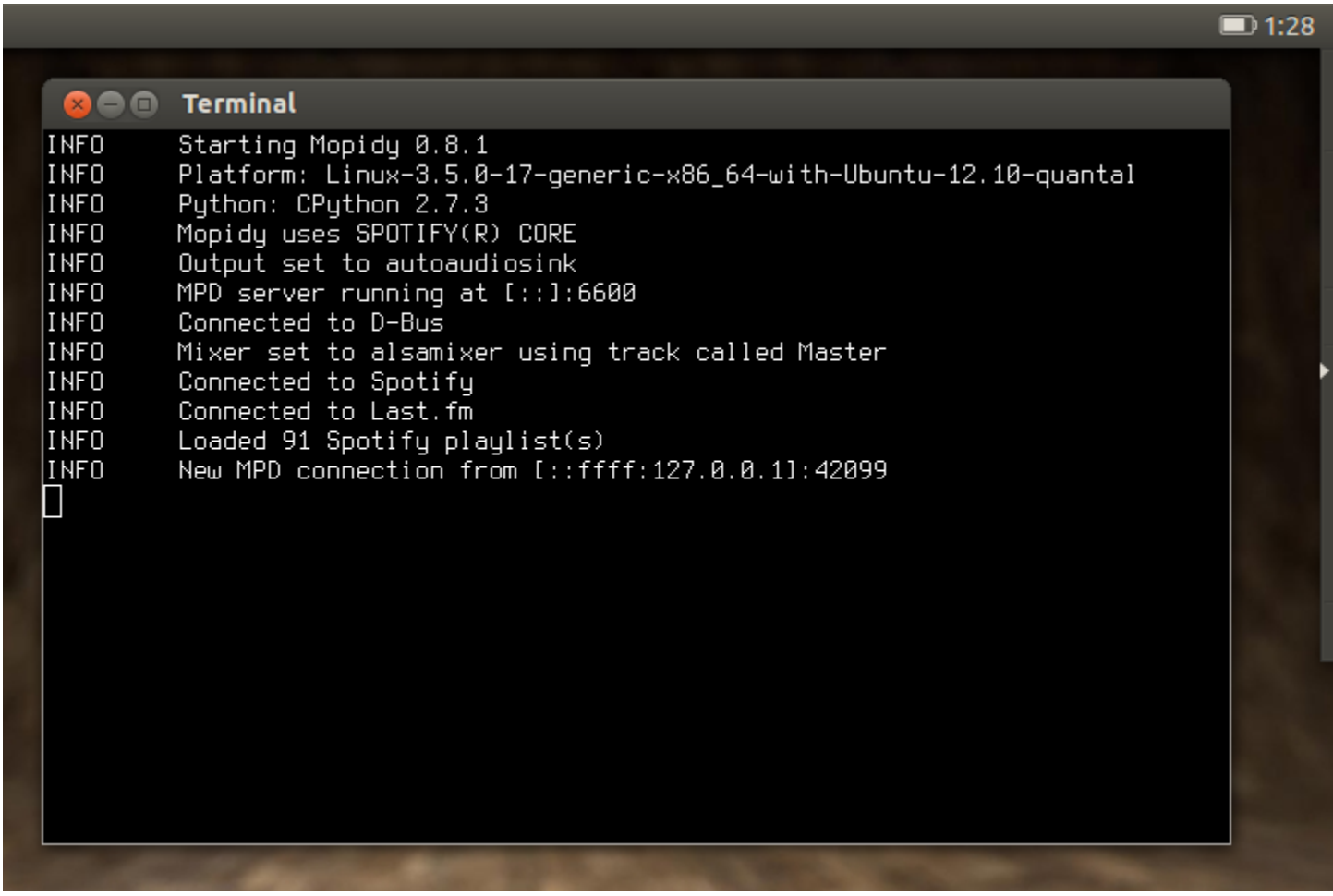
3.3 MPRIS clients

MPRIS is short for Media Player Remote Interfacing Specification. It's a spec that describes a standard D-Bus interface for making media players available to other applications on the same system.

The MPRIS frontend provided by the Mopidy-MPRIS extension currently implements all required parts of the MPRIS spec, plus the optional playlist interface. It does not implement the optional tracklist interface.

3.3.1 Ubuntu Sound Menu

The Ubuntu Sound Menu is the default sound menu in Ubuntu since 10.10 or 11.04. By default, it only includes the Rhytmbox music player, but many other players can integrate with the sound menu, including the official Spotify player and Mopidy.



If you install Mopidy from apt.mopidy.com, the sound menu should work out of the box. If you install Mopidy in any other way, you need to make sure that the file located at `data/mopidy.desktop` in the Mopidy git repo is installed as `/usr/share/applications/mopidy.desktop`, and that the properties `TryExec` and `Exec` in the file points to an existing executable file, preferably your Mopidy executable. If this isn't in place, the sound menu will not detect that Mopidy is running.

Next, Mopidy's MPRIS frontend must be running for the sound menu to be able to control Mopidy. The frontend is enabled by default, so as long as you have all its dependencies available, you should be good to go. Keep an eye out for warnings or errors from the MPRIS frontend when you start Mopidy, since it may fail because of missing dependencies or because Mopidy is started outside of X; the frontend won't work if `$DISPLAY` isn't set when Mopidy is started.

Under normal use, if Mopidy isn't running and you open the menu and click on "Mopidy Music Server", a terminal window will open and automatically start Mopidy. If Mopidy is already running, you'll see that Mopidy is marked with an arrow to the left of its name, like in the screen shot above, and the player controls will be visible. Mopidy doesn't support the MPRIS spec's optional playlist interface yet, so you'll not be able to select what track to play from

the sound menu. If you use an MPD client to queue a playlist, you can use the sound menu to check what you're currently playing, pause, resume, and skip to the next and previous track.

In summary, Mopidy's sound menu integration is currently not a full featured client, but it's a convenient addition to an MPD client since it's always easily available on Unity's menu bar.

3.3.2 Rygel

Rygel is an application that will translate between Mopidy's MPRIS interface and UPnP, and thus make Mopidy controllable from devices compatible with UPnP and/or DLNA. To read more about this, see *UPnP clients*.

3.4 UPnP clients

UPnP is a set of specifications for media sharing, playing, remote control, etc, across a home network. The specs are supported by a lot of consumer devices (like smartphones, TVs, Xbox, and PlayStation) that are often labeled as being DLNA compatible or certified.

The DLNA guidelines and UPnP specifications defines several device roles, of which Mopidy may play two:

DLNA Digital Media Server (DMS) / UPnP AV MediaServer:

> A MediaServer provides a library of media and is capable of streaming that media to a MediaRenderer. If Mopidy was a MediaServer, you could browse and play Mopidy's music on a TV, smartphone, or tablet supporting UPnP. Mopidy does not currently support this, but we may in the future. #52 is the relevant wishlist issue.

DLNA Digital Media Renderer (DMR) / UPnP AV MediaRenderer:

> A MediaRenderer is asked by some remote controller to play some given media, typically served by a MediaServer. If Mopidy was a MediaRenderer, you could use e.g. your smartphone or tablet to make Mopidy play media. Mopidy *does already* have experimental support for being a MediaRenderer, as you can read more about below.

3.4.1 Mopidy as an UPnP MediaRenderer

There are two ways Mopidy can be made available as an UPnP MediaRenderer: Using Mopidy-MPRIS and Rygel, or using Mopidy-MPD and upmpdcli.

upmpdcli

upmpdcli is recommended, since it is easier to setup, and offers **'OpenHome <http://www.openhome.org> ohMedia'_** compatibility. upmpdcli exposes a UPnP MediaRenderer to the network, while using the MPD protocol to control Mopidy.

1. Install upmpdcli. On Debian/Ubuntu:

   ```
   apt-get install upmpdcli
   ```

   Alternatively, follow the instructions from the upmpdcli website.

2. The default settings of upmpdcli will work with the default settings of *Mopidy-MPD*. Edit `/etc/upmpdcli.conf` if you want to use different ports, hosts, or other settings.

3. Start upmpdcli using the command:

```
upmpdcli
```

Or, run it in the background as a service:

```
sudo service upmpdcli start
```

4. A UPnP renderer should be available now.

Rygel

With the help of the Rygel project Mopidy can be made available as an UPnP MediaRenderer. Rygel will interface with the MPRIS interface provided by the Mopidy-MPRIS extension, and make Mopidy available as a MediaRenderer on the local network. Since this depends on the MPRIS frontend, which again depends on D-Bus being available, this will only work on Linux, and not OS X. MPRIS/D-Bus is only available to other applications on the same host, so Rygel must be running on the same machine as Mopidy.

1. Start Mopidy and make sure the MPRIS frontend is working. It is activated by default when the Mopidy-MPRIS extension is installed, but you may miss dependencies or be using OS X, in which case it will not work. Check the console output when Mopidy is started for any errors related to the MPRIS frontend. If you're unsure it is working, there are instructions for how to test it in the Mopidy-MPRIS readme.
2. Install Rygel. On Debian/Ubuntu:

```
sudo apt-get install rygel
```

3. Enable Rygel's MPRIS plugin. On Debian/Ubuntu, edit `/etc/rygel.conf`, find the `[MPRIS]` section, and change `enabled=false` to `enabled=true`.
4. Start Rygel by running:

```
rygel
```

Example output:

```
$ rygel
Rygel-Message: New plugin 'MediaExport' available
Rygel-Message: New plugin 'org.mpris.MediaPlayer2.mopidy' available
```

In the above example, you can see that Rygel found Mopidy, and it is now making Mopidy available through Rygel.

3.4.2 The UPnP-Inspector client

UPnP-Inspector is a graphical analyzer and debugging tool for UPnP services. It will detect any UPnP devices on your network, and show these in a tree structure. This is not a tool for your everyday music listening while relaxing on the couch, but it may be of use for testing that your setup works correctly.

1. Install UPnP-Inspector. On Debian/Ubuntu:

```
sudo apt-get install upnp-inspector
```

2. Run it:

```
upnp-inspector
```

3. Assuming that Mopidy is running with a working MPRIS frontend, and that Rygel is running on the same machine, Mopidy should now appear in UPnP-Inspector's device list.
4. If you expand the tree item saying `Mopidy (MediaRenderer:2)` or similiar, and then the sub element named `AVTransport:2` or similar, you'll find a list of commands you can invoke. E.g. if you double-click the `Pause` command, you'll get a new window where you can press an `Invoke` button, and then Mopidy should be paused.

Note that if you have a firewall on the host running Mopidy and Rygel, and you want this to be exposed to the rest of your local network, you need to open up your firewall for UPnP traffic. UPnP use UDP port 1900 as well as some dynamically assigned ports. I've only verified that this procedure works across the network by temporarily disabling the firewall on the the two hosts involved, so I'll leave any firewall configuration as an exercise to the reader.

3.4.3 Other clients

For a long list of UPnP clients for all possible platforms, see Wikipedia's List of UPnP AV media servers and clients.

CHAPTER 4

About

4.1 Authors

The following persons have contributed to Mopidy. 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>
- Johannes Knutsen <johannes@knutseninfo.no>
- Thomas Adamcik <thomas@adamcik.no>
- Kristian Klette <klette@samfundet.no>
- Martins Grunskis <martins@grunskis.com>
- Henrik Olsson <henrik@fixme.se>
- Antoine Pierlot-Garcin <antoine@bokbox.com>
- John Bäckstrand <sopues@gmail.com>
- Fred Hatfull <fred.hatfull@gmail.com>
- Erling Børresen <erling@fenicore.net>
- David C <dav@dav.com>
- Christian Johansen <christian@cjohansen.no>
- Matt Bray <mattjbray@gmail.com>
- Trygve Aaberge <trygveaa@gmail.com>
- Wouter van Wijk <woutervanwijk@gmail.com>
- Jeremy B. Merrill <jeremybmerrill@gmail.com>

- 0xadam <radx@live.com.au>
- herrernst <herr.ernst@gmail.com>
- Nick Steel <kingosticks@gmail.com>
- Zan Dobersek <zandobersek@gmail.com>
- Thomas Refis <refis.thomas@gmail.com>
- Janez Troha <janez.troha@gmail.com>
- Tobias Sauerwein <cgtobi@gmail.com>
- Alli Witheford <alzeih@gmail.com>
- Alexandre Petitjean <alpetitjean@gmail.com>
- Terje Larsen <terlar@gmail.com>
- Javier Domingo Cansino <javierdo1@gmail.com>
- Pavol Babincak <scroolik@gmail.com>
- Javier Domingo <javierdo1@gmail.com>
- Lasse Bigum <lasse@bigum.org>
- David Eisner <david.eisner@oriel.oxon.org>
- Pål Ruud <ruudud@gmail.com>
- Thomas Kemmer <tkemmer@computer.org>
- Paul Connolley <paul.connolley@gmail.com>
- Luke Giuliani <luke@giuliani.com.au>
- Colin Montgomerie <kiteflyingmonkey@gmail.com>
- Simon de Bakker <simon@simbits.nl>
- Arnaud Barisain-Monrose <abarisain@gmail.com>
- nathanharper <nathan.sam.harper@gmail.com>
- Pierpaolo Frasa <pfrasa@smail.uni-koeln.de>
- Thomas Scholtes <thomas-scholtes@gmx.de>
- Sam Willcocks <sam@wlcx.cc>
- Ignasi Fosch <natx@y10k.ws>
- Arjun Naik <arjun@arjunnaik.in>
- Christopher Schirner <christopher@hackerspace-bamberg.de>
- Dmitry Sandalov <dmitry@sandalov.org>
- Deni Bertovic <deni@kset.org>
- Thomas Amland <thomas.amland@gmail.com>

If you already enjoy Mopidy, or don't enjoy it and want to help us making Mopidy better, the best way to do so is to contribute back to the community. You can contribute code, documentation, tests, bug reports, or help other users, spreading the word, etc. See *Contributing* for a head start.

4.2 Sponsors

The Mopidy project would like to thank the following sponsors for supporting the project.

4.2.1 Rackspace

Rackspace lets Mopidy use their hosting services for free. We use their services for the following sites:

- Hosting of the APT package repository at https://apt.mopidy.com.
- Hosting of the Discourse forum at https://discuss.mopidy.com.
- Mailgun for sending emails from the Discourse forum.
- Hosting of the Jenkins CI server at https://ci.mopidy.com.
- Hosting of a Linux worker for https://ci.mopidy.com.
- Hosting of a Windows worker for https://ci.mopidy.com.
- CDN hosting at http://dl.mopidy.com, which is used to distribute Pi Musicbox images.

4.2.2 GlobalSign

GlobalSign provides Mopidy with a free wildcard SSL certificate for mopidy.com, which we use to secure access to all our web sites.

4.3 Changelog

This changelog is used to track all major changes to Mopidy.

4.3.1 v0.19.5 (2014-12-23)

Today is Mopidy's five year anniversary. We're celebrating with a bugfix release and are looking forward to the next five years!

- Config: Support UTF-8 in extension's default config. If an extension with non-ASCII characters in its default config was installed, and Mopidy didn't already have a config file, Mopidy would crashed when trying to create the initial config file based on the default config of all available extensions. (Fixes: discuss.mopidy.com/t/428)
- Extensions: Fix crash when unpacking data from `pkg_resources.VersionConflict` created with a single argument. (Fixes: #911)
- Models: Hide empty collections from `repr()` representations.
- Models: Field values are no longer stored on the model instance when the value matches the default value for the field. This makes two models equal when they have a field which in one case is implicitly set to the default value and in the other case explicitly set to the default value, but with otherwise equal fields. (Fixes: #837)
- Models: Changed the default value of *`mopidy.models.Album.num_tracks`*, *`mopidy.models.Track.track_no`*, and *`mopidy.models.Track.last_modified`* from `0` to `None`.
- Core: When skipping to the next track in consume mode, remove the skipped track from the tracklist. This is consistent with the original MPD server's behavior. (Fixes: #902)
- Local: Fix scanning of modified files. (PR: #904)

- MPD: Re-enable browsing of empty directories. (PR: #906)
- MPD: Remove track comments from responses. They are not included by the original MPD server, and this works around #881. (PR: #882)
- HTTP: Errors while starting HTTP apps are logged instead of crashing the HTTP server. (Fixes: #875)

4.3.2 v0.19.4 (2014-09-01)

Bug fix release.

- Configuration: *mopidy --config* now supports directories.
- Logging: Fix that some loggers would be disabled if *logging/config_file* was set. (Fixes: #740)
- Quit process with exit code 1 when stopping because of a backend, frontend, or mixer initialization error.
- Backend API: Update *mopidy.backend.LibraryProvider.browse()* signature and docs to match how the core use the backend's browse method. (Fixes: #833)
- Local library API: Add *mopidy.local.Library.ROOT_DIRECTORY_URI* constant for use by implementors of *mopidy.local.Library.browse()*. (Related to: #833)
- HTTP frontend: Guard against double close of WebSocket, which causes an `AttributeError` on Tornado < 3.2.
- MPD frontend: Make the `list` command return albums when sending 3 arguments. This was incorrectly returning artists after the MPD command changes in 0.19.0. (Fixes: #817)
- MPD frontend: Fix a race condition where two threads could try to free the same data simultaneously. (Fixes: #781)

4.3.3 v0.19.3 (2014-08-03)

Bug fix release.

- Audio: Fix negative track length for radio streams. (Fixes: #662, PR: #796)
- Audio: Tell GStreamer to not pick Jack sink. (Fixes: #604)
- Zeroconf: Fix discovery by adding `.local` to the announced hostname. (PR: #795)
- Zeroconf: Fix intermittent DBus/Avahi exception.
- Extensions: Fail early if trying to setup an extension which doesn't implement the *mopidy.ext.Extension.setup()* method. (Fixes: #813)

4.3.4 v0.19.2 (2014-07-26)

Bug fix release, directly from the Mopidy development sprint at EuroPython 2014 in Berlin.

- Audio: Make *audio/mixer_volume* work on the software mixer again. This was broken with the mixer changes in 0.19.0. (Fixes: #791)
- HTTP frontend: When using Tornado 4.0, allow WebSocket requests from other hosts. (Fixes: #788)
- MPD frontend: Fix crash when MPD commands are called with the wrong number of arguments. This was broken with the MPD command changes in 0.19.0. (Fixes: #789)

4.3.5 v0.19.1 (2014-07-23)

Bug fix release.

- Dependencies: Mopidy now requires Tornado >= 2.3, instead of >= 3.1. This should make Mopidy continue to work on Debian/Raspbian stable, where Tornado 2.3 is the newest version available.
- HTTP frontend: Add missing string interpolation placeholder.
- Development: `mopidy --version` and `mopidy.core.Core.get_version()` now returns the correct version when Mopidy is run from a Git repo other than Mopidy's own. (Related to #706)

4.3.6 v0.19.0 (2014-07-21)

The focus of 0.19 have been on improving the MPD implementation, replacing GStreamer mixers with our own mixer API, and on making web clients installable with `pip`, like any other Mopidy extension.

Since the release of 0.18, we've closed or merged 53 issues and pull requests through about 445 commits by *12 people*, including five new guys. Thanks to everyone that has contributed!

Dependencies

- Mopidy now requires Tornado >= 3.1.
- Mopidy no longer requires CherryPy or ws4py. Previously, these were optional dependencies required for the HTTP frontend to work.

Backend API

- *Breaking change:* Imports of the backend API from `mopidy.backends` no longer works. The new API introced in v0.18 is now required. Most extensions already use the new API location.

Commands

- The `mopidy-convert-config` tool for migrating the `setings.py` configuration file used by Mopidy up until 0.14 to the new config file format has been removed after over a year of trusty service. If you still need to convert your old `settings.py` configuration file, do so using and older release, like Mopidy 0.18, or migrate the configuration to the new format by hand.

Configuration

- Add `optional=True` support to `mopidy.config.Boolean`.

Logging

- Fix proper decoding of exception messages that depends on the user's locale.
- Colorize logs depending on log level. This can be turned off with the new *`logging/color`* configuration. (Fixes: #772)

Extension support

- *Breaking change:* Removed the *`Extension`* methods that were deprecated in 0.18: `get_backend_classes()`, `get_frontend_classes()`, and `register_gstreamer_elements()`. Use *`mopidy.ext.Extension.setup()`* instead, as most extensions already do.

Audio

- *Breaking change:* Removed support for GStreamer mixers. GStreamer 1.x does not support volume control, so we changed to use software mixing by default in v0.17.0. Now, we're removing support for all other GStreamer mixers and are reintroducing mixers as something extensions can provide independently of GStreamer. (Fixes: #665, PR: #760)

- *Breaking change:* Changed the *audio/mixer* config value to refer to Mopidy mixer extensions instead of GStreamer mixers. The default value, `software`, still has the same behavior. All other values will either no longer work or will at the very least require you to install an additional extension.
- Changed the *audio/mixer_volume* config value behavior from affecting GStreamer mixers to affecting Mopidy mixer extensions instead. The end result should be the same without any changes to this config value.
- Deprecated the `audio/mixer_track` config value. This config value is no longer in use. Mixer extensions that need additional configuration handle this themselves.
- Use *Proxy configuration* when streaming media from the Internet. (Partly fixing #390)
- Fix proper decoding of exception messages that depends on the user's locale.
- Fix recognition of ASX and XSPF playlists with tags in all caps or with carriage return line endings. (Fixes: #687)
- Support simpler ASX playlist variant with `<ENTRY>` elements without children.
- Added `target_state` attribute to the audio layer's *state_changed()* event. Currently, it is `None` except when we're paused because of buffering. Then the new field exposes our target state after buffering has completed.

Mixers

- Added new *mopidy.mixer.Mixer* API which can be implemented by extensions.
- Created a bundled extension, *Mopidy-SoftwareMixer*, for controlling volume in software in GStreamer's pipeline. This is Mopidy's default mixer. To use this mixer, set the *audio/mixer* config value to `software`.
- Created an external extension, Mopidy-ALSAMixer, for controlling volume with hardware through ALSA. To use this mixer, install the extension, and set the *audio/mixer* config value to `alsamixer`.

HTTP frontend

- CherryPy and ws4py have been replaced with Tornado. This will hopefully reduce CPU usage on OS X (#445) and improve error handling in corner cases, like when returning from suspend (#718).
- Added support for packaging web clients as Mopidy extensions and installing them using pip. See the *HTTP server side API* for details. (Fixes: #440)
- Added web page at `/mopidy/` which lists all web clients installed as Mopidy extensions. (Fixes: #440)
- Added support for extending the HTTP frontend with additional server side functionality. See *HTTP server side API* for details.
- Exposed the core API using HTTP POST requests with JSON-RPC payloads at `/mopidy/rpc`. This is the same JSON-RPC interface as is exposed over the WebSocket at `/mopidy/ws`, so you can run any core API command.

  The HTTP POST interfaces does not give you access to events from Mopidy, like the WebSocket does. The WebSocket interface is still recommended for web clients. The HTTP POST interface may be easier to use for simpler programs, that just needs to query the currently playing track or similar. See *HTTP POST API* for details.
- If Zeroconf is enabled, we now announce the `_mopidy-http._tcp` service in addition to `_http._tcp`. This is to make it easier to automatically find Mopidy's HTTP server among other Zeroconf-published HTTP servers on the local network.

Mopidy.js client library

This version has been released to npm as Mopidy.js v0.4.0.

- Update Mopidy.js to use when.js 3. If you maintain a Mopidy client, you should review the differences between when.js 2 and 3 and the when.js debugging guide.

- All of Mopidy.js' promise rejection values are now of the Error type. This ensures that all JavaScript VMs will show a useful stack trace if a rejected promise's value is used to throw an exception. To allow catch clauses to handle different errors differently, server side errors are of the type `Mopidy.ServerError`, and connection related errors are of the type `Mopidy.ConnectionError`.
- Add support for method calls with by-name arguments. The old calling convention, `by-position-only`, is still the default, but this will change in the future. A warning is logged to the console if you don't explicitly select a calling convention. See the *Mopidy.js JavaScript library* docs for details.

MPD frontend

- Proper command tokenization for MPD requests. This replaces the old regex based system with an MPD protocol specific tokenizer responsible for breaking requests into pieces before the handlers have at them. (Fixes: #591 and #592)
- Updated command handler system. As part of the tokenizer cleanup we've updated how commands are registered and making it simpler to create new handlers.
- Simplified a bunch of handlers. All the "browse" type commands now use a common browse helper under the hood for less repetition. Likewise the query handling of "search" commands has been somewhat simplified.
- Adds placeholders for missing MPD commands, preparing the way for bumping the protocol version once they have been added.
- Respond to all pending requests before closing connection. (PR: #722)
- Stop incorrectly catching *LookupError* in command handling. (Fixes: #741)
- Browse support for playlists and albums has been added. (PR: #749, #754)
- The `lsinfo` command now returns browse results before local playlists. This is helpful as not all clients sort the returned items. (PR: #755)
- Browse now supports different entries with identical names. (PR: #762)
- Search terms that are empty or consists of only whitespace are no longer included in the search query sent to backends. (PR: #758)

Local backend

- The JSON local library backend now logs a friendly message telling you about `mopidy local scan` if you don't have a local library cache. (Fixes: #711)
- The `local scan` command now use multiple threads to walk the file system and check files' modification time. This speeds up scanning, escpecially when scanning remote file systems over e.g. NFS.
- the `local scan` command now creates necessary folders if they don't already exist. Previously, this was only done by the Mopidy server, so doing a `local scan` before running the server the first time resulted in a crash. (Fixes: #703)
- Fix proper decoding of exception messages that depends on the user's locale.

Stream backend

- Add config value *`stream/metadata_blacklist`* to blacklist certain URIs we should not open to read metadata from before they are opened for playback. This is typically needed for services that invalidate URIs after a single use. (Fixes: #660)

4.3.7 v0.18.3 (2014-02-16)

Bug fix release.

- Fix documentation build.

4.3.8 v0.18.2 (2014-02-16)

Bug fix release.

- We now log warnings for wrongly configured extensions, and clearly label them in `mopidy config`, but does no longer stop Mopidy from starting because of misconfigured extensions. (Fixes: #682)
- Fix a crash in the server side WebSocket handler caused by connection problems with clients. (Fixes: #428, #571)
- Fix the `time_position` field of the `track_playback_ended` event, which has been always 0 since v0.18.0. This made scrobbles by Mopidy-Scrobbler not be persisted by Last.fm, because Mopidy reported that you listened to 0 seconds of each track. (Fixes: #674)
- Fix the log setup so that it is possible to increase the amount of logging from a specific logger using the `loglevels` config section. (Fixes: #684)
- Serialization of *`Playlist`* models with the `last_modified` field set to a `datetime.datetime` instance did not work. The type of *`mopidy.models.Playlist.last_modified`* has been redefined from a `datetime.datetime` instance to the number of milliseconds since Unix epoch as an integer. This makes serialization of the time stamp simpler.
- Minor refactor of the MPD server context so that Mopidy's MPD protocol implementation can easier be reused. (Fixes: #646)
- Network and signal handling has been updated to play nice on Windows systems.

4.3.9 v0.18.1 (2014-01-23)

Bug fix release.

- Disable extension instead of crashing if a dependency has the wrong version. (Fixes: #657)
- Make logging work to both console, debug log file, and any custom logging setup from *`logging/config_file`* at the same time. (Fixes: #661)

4.3.10 v0.18.0 (2014-01-19)

The focus of 0.18 have been on two fronts: the local library and browsing.

First, the local library's old tag cache file used for storing the track metadata scanned from your music collection has been replaced with a far simpler implementation using JSON as the storage format. At the same time, the local library have been made replaceable by extensions, so you can now create extensions that use your favorite database to store the metadata.

Second, we've finally implemented the long awaited "file system" browsing feature that you know from MPD. It is supported by both the MPD frontend and the local and Spotify backends. It is also used by the new Mopidy-Dirble extension to provide you with a directory of Internet radio stations from all over the world.

Since the release of 0.17, we've closed or merged 49 issues and pull requests through about 285 commits by *11 people*, including six new guys. Thanks to everyone that has contributed!

Core API

- Add *`mopidy.core.Core.version()`* for HTTP clients to manage compatibility between API versions. (Fixes: #597)
- Add *`mopidy.models.Ref`* class for use as a lightweight reference to other model types, containing just an URI, a name, and an object type. It is barely used for now, but its use will be extended over time.

- Add *mopidy.core.LibraryController.browse()* method for browsing a virtual file system of tracks. Backends can implement support for this by implementing *mopidy.backend.LibraryProvider.browse()*.
- Events emitted on play/stop, pause/resume, next/previous and on end of track has been cleaned up to work consistently. See the message of commit 1d108752f6 for the full details. (Fixes: #629)

Backend API

- Move the backend API classes from `mopidy.backends.base` to *mopidy.backend* and remove the `Base` prefix from the class names:
  - From `mopidy.backends.base.Backend` to *mopidy.backend.Backend*
  - From `mopidy.backends.base.BaseLibraryProvider` to *mopidy.backend.LibraryProvider*
  - From `mopidy.backends.base.BasePlaybackProvider` to *mopidy.backend.PlaybackProvider*
  - From `mopidy.backends.base.BasePlaylistsProvider` to *mopidy.backend.PlaylistsProvider*
  - From `mopidy.backends.listener.BackendListener` to *mopidy.backend.BackendListener*

  Imports from the old locations still works, but are deprecated.
- Add *mopidy.backend.LibraryProvider.browse()*, which can be implemented by backends that wants to expose directories of tracks in Mopidy's virtual file system.

Frontend API

- The dummy backend used for testing many frontends have moved from `mopidy.backends.dummy` to `mopidy.backend.dummy`.

Commands

- Reduce amount of logging from dependencies when using *mopidy -v*. (Fixes: #593)
- Add support for additional logging verbosity levels with `mopidy -vv` and `mopidy -vvv` which increases the amount of logging from dependencies. (Fixes: #593)

Configuration

- The default for the *mopidy --config* option has been updated to include `$XDG_CONFIG_DIRS` in addition to `$XDG_CONFIG_DIR`. (Fixes #431)
- Added support for deprecating config values in order to allow for graceful removal of the no longer used config value `local/tag_cache_file`.

Extension support

- Switched to using a registry model for classes provided by extension. This allows extensions to be extended by other extensions, as needed by for example pluggable libraries for the local backend. See *mopidy.ext.Registry* for details. (Fixes #601)
- Added the new method *mopidy.ext.Extension.setup()*. This method replaces the now deprecated `get_backend_classes()`, `get_frontend_classes()`, and `register_gstreamer_elements()`.

Audio

- Added *audio/mixer_volume* to set the initial volume of mixers. This is especially useful for setting the software mixer volume to something else than the default 100%. (Fixes: #633)

Local backend

Note: After upgrading to Mopidy 0.18 you must run `mopidy local scan` to reindex your local music collection. This is due to the change of storage format.

- Added support for browsing local directories in Mopidy's virtual file system.
- Finished the work on creating pluggable libraries. Users can now reconfigure Mopidy to use alternate library providers of their choosing for local files. (Fixes issue #44, partially resolves #397, and causes a temporary regression of #527.)
- Switched default local library provider from a "tag cache" file that closely resembled the one used by the original MPD server to a compressed JSON file. This greatly simplifies our library code and reuses our existing model serialization code, as used by the HTTP API and web clients.
- Removed our outdated and bug-ridden "tag cache" local library implementation.
- Added the config value *`local/library`* to select which library to use. It defaults to `json`, which is the only local library bundled with Mopidy.
- Added the config value *`local/data_dir`* to have a common config for where to store local library data. This is intended to avoid every single local library provider having to have it's own config value for this.
- Added the config value *`local/scan_flush_threshold`* to control how often to tell local libraries to store changes when scanning local music.

Streaming backend

- Add live lookup of URI metadata. (Fixes #540)
- Add support for extended M3U playlist, meaning that basic track metadata stored in playlists will be used by Mopidy.

HTTP frontend

- Upgrade Mopidy.js dependencies and add support for using Mopidy.js with Browserify. This version has been released to npm as Mopidy.js v0.2.0. (Fixes: #609)

MPD frontend

- Make the `lsinfo`, `listall`, and `listallinfo` commands support browsing of Mopidy's virtual file system. (Fixes: #145)
- Empty commands now return a `ACK [5@0] {} No command given` error instead of `OK`. This is consistent with the original MPD server implementation.

Internal changes

- Events from the audio actor, backends, and core actor are now emitted asyncronously through the GObject event loop. This should resolve the issue that has blocked the merge of the EOT-vs-EOS fix for a long time.

4.3.11 v0.17.0 (2013-11-23)

The focus of 0.17 has been on introducing subcommands to the `mopidy` command, making it possible for extensions to add subcommands of their own, and to improve the default config file when starting Mopidy the first time. In addition, we've grown support for Zeroconf publishing of the MPD and HTTP servers, and gotten a much faster scanner. The scanner now also scans some additional tags like composers and performers.

Since the release of 0.16, we've closed or merged 22 issues and pull requests through about 200 commits by *five people*, including one new contributor.

Commands

- Switched to subcommands for the `mopidy` command , this implies the following changes: (Fixes: #437)

Old command	New command
mopidy –show-deps	mopidy deps
mopidy –show-config	mopidy config
mopidy-scan	mopidy local scan

- Added hooks for extensions to create their own custom subcommands and converted `mopidy-scan` as a first user of the new API. (Fixes: #436)

Configuration

- When `mopidy` is started for the first time we create an empty *`$XDG_CONFIG_DIR`*`/mopidy/mopidy.conf` file. We now populate this file with the default config for all installed extensions so it'll be easier to set up Mopidy without looking through all the documentation for relevant config values. (Fixes: #467)

Core API

- The *`Track`* model has grown fields for `composers`, `performers`, `genre`, and `comment`.
- The search field `track` has been renamed to `track_name` to avoid confusion with `track_no`. (Fixes: #535)
- The signature of the tracklist's *`filter()`* and *`remove()`* methods have changed. Previously, they expected e.g. `tracklist.filter(tlid=17)`. Now, the value must always be a list, e.g. `tracklist.filter(tlid=[17])`. This change allows you to get or remove multiple tracks with a single call, e.g. `tracklist.remove(tlid=[1, 2, 7])`. This is especially useful for web clients, as requests can be batched. This also brings the interface closer to the library's *`find_exact()`* and *`search()`* methods.

Audio

- Change default volume mixer from `autoaudiomixer` to `software`. GStreamer 1.x does not support volume control, so we're changing to use software mixing by default, as that may be the only thing we'll support in the future when we upgrade to GStreamer 1.x.

Local backend

- Library scanning has been switched back from GStreamer's discoverer to our custom implementation due to various issues with GStreamer 0.10's built in scanner. This also fixes the scanner slowdown. (Fixes: #565)
- When scanning, we no longer default the album artist to be the same as the track artist. Album artist is now only populated if the scanned file got an explicit album artist set.
- The scanner will now extract multiple artists from files with multiple artist tags.
- The scanner will now extract composers and performers, as well as genre, bitrate, and comments. (Fixes: #577)
- Fix scanner so that time of last modification is respected when deciding which files can be skipped when scanning the music collection for changes.
- The scanner now ignores the capitalization of file extensions in *`local/excluded_file_extensions`*, so you no longer need to list both `.jpg` and `.JPG` to ignore JPEG files when scanning. (Fixes: #525)
- The scanner now by default ignores `*.nfo` and `*.html` files too.

MPD frontend

- The MPD service is now published as a Zeroconf service if avahi-daemon is running on the system. Some MPD clients will use this to present Mopidy as an available server on the local network without needing any configuration. See the *`mpd/zeroconf`* config value to change the service name or disable the service. (Fixes: #39)

- Add support for `composer`, `performer`, `comment`, `genre`, and `performer`. These tags can be used with `list ...`, `search ...`, and `find ...` and their variants, and are supported in the `any` tag also
- The `bitrate` field in the `status` response is now always an integer. This follows the behavior of the original MPD server. (Fixes: #577)

HTTP frontend

- The HTTP service is now published as a Zeroconf service if avahi-daemon is running on the system. Some browsers will present HTTP Zeroconf services on the local network as "local sites" bookmarks. See the *`http/zeroconf`* config value to change the service name or disable the service. (Fixes: #39)

DBUS/MPRIS

- The `mopidy` process now registers it's GObject event loop as the default eventloop for dbus-python. (Fixes: mopidy-mpris#2)

4.3.12 v0.16.1 (2013-11-02)

This is very small release to get Mopidy's Debian package ready for inclusion in Debian.

Commands

- Fix removal of last dir level in paths to dependencies in `mopidy --show-deps` output.
- Add manpages for all commands.

Local backend

- Fix search filtering by track number that was added in 0.16.0.

MPD frontend

- Add support for `list "albumartist" ...` which was missed when `find` and `search` learned to handle `albumartist` in 0.16.0. (Fixes: #553)

4.3.13 v0.16.0 (2013-10-27)

The goals for 0.16 were to add support for queuing playlists of e.g. radio streams directly to Mopidy, without manually extracting the stream URLs from the playlist first, and to move the Spotify, Last.fm, and MPRIS support out to independent Mopidy extensions, living outside the main Mopidy repo. In addition, we've seen some cleanup to the playback vs tracklist part of the core API, which will require some changes for users of the HTTP/JavaScript APIs, as well as the addition of audio muting to the core API. To speed up the *development of new extensions*, we've added a cookiecutter project to get the skeleton of a Mopidy extension up and running in a matter of minutes. Read below for all the details and for links to issues with even more details.

Since the release of 0.15, we've closed or merged 31 issues and pull requests through about 200 commits by *five people*, including three new contributors.

Dependencies

Parts of Mopidy have been moved to their own external extensions. If you want Mopidy to continue to work like it used to, you may have to install one or more of the following extensions as well:

- The Spotify backend has been moved to Mopidy-Spotify.
- The Last.fm scrobbler has been moved to Mopidy-Scrobbler.
- The MPRIS frontend has been moved to Mopidy-MPRIS.

Core

- Parts of the functionality in *mopidy.core.PlaybackController* have been moved to *mopidy.core.TracklistController*:

Old location	New location
playback.get_consume()	tracklist.get_consume()
playback.set_consume(v)	tracklist.set_consume(v)
playback.consume	tracklist.consume
playback.get_random()	tracklist.get_random()
playback.set_random(v)	tracklist.set_random(v)
playback.random	tracklist.random
playback.get_repeat()	tracklist.get_repeat()
playback.set_repeat(v)	tracklist.set_repeat(v)
playback.repeat	tracklist.repeat
playback.get_single()	tracklist.get_single()
playback.set_single(v)	tracklist.set_single(v)
playback.single	tracklist.single
playback.get_tracklist_position()	tracklist.index(tl_track)
playback.tracklist_position	tracklist.index(tl_track)
playback.get_tl_track_at_eot()	tracklist.eot_track(tl_track)
playback.tl_track_at_eot	tracklist.eot_track(tl_track)
playback.get_tl_track_at_next()	tracklist.next_track(tl_track)
playback.tl_track_at_next	tracklist.next_track(tl_track)
playback.get_tl_track_at_previous()	tracklist.previous_track(tl_track)
playback.tl_track_at_previous	tracklist.previous_track(tl_track)

  The `tl_track` argument to the last four new functions are used as the reference `tl_track` in the tracklist to find e.g. the next track. Usually, this will be *current_tl_track*.

- Added *mopidy.core.PlaybackController.mute* for muting and unmuting audio. (Fixes: #186)
- Added *mopidy.core.CoreListener.mute_changed()* event that is triggered when the mute state changes.
- In "random" mode, after a full playthrough of the tracklist, playback continued from the last track played to the end of the playlist in non-random order. It now stops when all tracks have been played once, unless "repeat" mode is enabled. (Fixes: #453)
- In "single" mode, after a track ended, playback continued with the next track in the tracklist. It now stops after playing a single track, unless "repeat" mode is enabled. (Fixes: #496)

Audio

- Added support for parsing and playback of playlists in GStreamer. For end users this basically means that you can now add a radio playlist to Mopidy and we will automatically download it and play the stream inside it. Currently we support M3U, PLS, XSPF and ASX files. Also note that we can currently only play the first stream in the playlist.
- We now handle the rare case where an audio track has max volume equal to min. This was causing divide by zero errors when scaling volumes to a zero to hundred scale. (Fixes: #525)
- Added support for muting audio without setting the volume to 0. This works both for the software and hardware mixers. (Fixes: #186)

Local backend

- Replaced our custom media library scanner with GStreamer's builtin scanner. This should make scanning less error prone and faster as timeouts should be infrequent. (Fixes: #198)

- Media files with less than 100ms duration are now excluded from the library.
- Media files with the file extensions `.jpeg`, `.jpg`, `.png`, `.txt`, and `.log` are now skipped by the media library scanner. You can change the list of excluded file extensions by setting the *`local/excluded_file_extensions`* config value. (Fixes: #516)
- Unknown URIs found in playlists are now made into track objects with the URI set instead of being ignored. This makes it possible to have playlists with e.g. HTTP radio streams and not just `local:track:...` URIs. This used to work, but was broken in Mopidy 0.15.0. (Fixes: #527)
- Fixed crash when playing `local:track:...` URIs which contained non-ASCII chars after uridecode.
- Removed media files are now also removed from the in-memory media library when the media library is reloaded from disk. (Fixes: #500)

MPD frontend

- Made the formerly unused commands `outputs`, `enableoutput`, and `disableoutput` mute/unmute audio. (Related to: #186)
- The MPD command `list` now works with `"albumartist"` as its second argument, e.g. `list "album" "albumartist" "anartist"`. (Fixes: #468)
- The MPD commands `find` and `search` now accepts `albumartist` and `track` (this is the track number, not the track name) as field types to limit the search result with.
- The MPD command `count` is now implemented. It accepts the same type of arguments as `find` and `search`, but returns the number of tracks and their total playtime instead.

Extension support

- A cookiecutter project for quickly creating new Mopidy extensions have been created. You can find it at cookiecutter-mopidy-ext. (Fixes: #522)

4.3.14 v0.15.0 (2013-09-19)

A release with a number of small and medium fixes, with no specific focus.

Dependencies

- Mopidy no longer supports Python 2.6. Currently, the only Python version supported by Mopidy is Python 2.7. We're continuously working towards running Mopidy on Python 3. (Fixes: #344)

Command line options

- Converted from the optparse to the argparse library for handling command line options.
- `mopidy --show-config` will now take into consideration any *`mopidy --option`* arguments appearing later on the command line. This helps you see the effective configuration for runs with the same `mopidy --options` arguments.

Audio

- Added support for audio visualization. *`audio/visualizer`* can now be set to GStreamer visualizers.
- Properly encode localized mixer names before logging.

Local backend

- An album's number of discs and a track's disc number are now extracted when scanning your music collection.
- The scanner now gives up scanning a file after a second, and continues with the next file. This fixes some hangs on non-media files, like logs. (Fixes: #476, #483)

- Added support for pluggable library updaters. This allows extension writers to start providing their own custom libraries instead of being stuck with just our tag cache as the only option.
- Converted local backend to use new `local:playlist:path` and `local:track:path` URI scheme. Also moves support of `file://` to streaming backend.

Spotify backend

- Prepend playlist folder names to the playlist name, so that the playlist hierarchy from your Spotify account is available in Mopidy. (Fixes: #62)
- Fix proxy config values that was broken with the config system change in 0.14. (Fixes: #472)

MPD frontend

- Replace newline, carriage return and forward slash in playlist names. (Fixes: #474, #480)
- Accept `listall` and `listallinfo` commands without the URI parameter. The methods are still not implemented, but now the commands are accepted as valid.

HTTP frontend

- Fix too broad truth test that caused *`mopidy.models.TlTrack`* objects with `tlid` set to `0` to be sent to the HTTP client without the `tlid` field. (Fixes: #501)
- Upgrade Mopidy.js dependencies. This version has been released to npm as Mopidy.js v0.1.1.

Extension support

- `mopidy.config.Secret` is now deserialized to unicode instead of bytes. This may require modifications to extensions.

4.3.15 v0.14.2 (2013-07-01)

This is a maintenance release to make Mopidy 0.14 work with pyspotify 1.11.

Dependencies

- pyspotify >= 1.9, < 2 is now required for Spotify support. In other words, you're free to upgrade to pyspotify 1.11, but it isn't a requirement.

4.3.16 v0.14.1 (2013-04-28)

This release addresses an issue in v0.14.0 where the new `mopidy-convert-config` tool and the new *`mopidy --option`* command line option was broken because some string operations inadvertently converted some byte strings to unicode.

4.3.17 v0.14.0 (2013-04-28)

The 0.14 release has a clear focus on two things: the new configuration system and extension support. Mopidy's documentation has also been greatly extended and improved.

Since the last release a month ago, we've closed or merged 53 issues and pull requests. A total of seven *authors* have contributed, including one new.

Dependencies

- setuptools or distribute is now required. We've introduced this dependency to use setuptools' entry points functionality to find installed Mopidy extensions.

New configuration system

- Mopidy has a new configuration system based on ini-style files instead of a Python file. This makes configuration easier for users, and also makes it possible for Mopidy extensions to have their own config sections.

  As part of this change we have cleaned up the naming of our config values.

  To ease migration we've made a tool named `mopidy-convert-config` for automatically converting the old `settings.py` to a new `mopidy.conf` file. This tool takes care of all the renamed config values as well. See `mopidy-convert-config` for details on how to use it.

- A long wanted feature: You can now enable or disable specific frontends or backends without having to redefine `FRONTENDS` or `BACKENDS` in your config. Those config values are gone completely.

Extension support

- Mopidy now supports extensions. This means that any developer now easily can create a Mopidy extension to add new control interfaces or music backends. This helps spread the maintenance burden across more developers, and also makes it possible to extend Mopidy with new backends the core developers are unable to create and/or maintain because of geo restrictions, etc. If you're interested in creating an extension for Mopidy, read up on *Extension development*.
- All of Mopidy's existing frontends and backends are now plugged into Mopidy as extensions, but they are still distributed together with Mopidy and are enabled by default.
- The NAD mixer have been moved out of Mopidy core to its own project, Mopidy-NAD. See *Extensions* for more information.
- Janez Troha has made the first two external extensions for Mopidy: a backend for playing music from Soundcloud, and a backend for playing music from a Beets music library. See *Extensions* for more information.

Command line options

- The command option `mopidy --list-settings` is now named `mopidy --show-config`.
- The command option `mopidy --list-deps` is now named `mopidy --show-deps`.
- What configuration files to use can now be specified through the command option *`mopidy --config`*, multiple files can be specified using colon as a separator.
- Configuration values can now be overridden through the command option *`mopidy --option`*. For example: `mopidy --option spotify/enabled=false`.
- The GStreamer command line options, `mopidy --gst-*` and `mopidy --help-gst` are no longer supported. To set GStreamer debug flags, you can use environment variables such as `GST_DEBUG`. Refer to GStreamer's documentation for details.

Spotify backend

- Add support for starred playlists, both your own and those owned by other users. (Fixes: #326)
- Fix crash when a new playlist is added by another Spotify client. (Fixes: #387, #425)

MPD frontend

- Playlists with identical names are now handled properly by the MPD frontend by suffixing the duplicate names with e.g. `[2]`. This is needed because MPD identify playlists by name only, while Mopidy and Spotify supports multiple playlists with the same name, and identify them using an URI. (Fixes: #114)

MPRIS frontend

- The frontend is now disabled if the `DISPLAY` environment variable is unset. This avoids some harmless error messages, that have been known to confuse new users debugging other problems.

Development

- Developers running Mopidy from a Git clone now need to run `python setup.py develop` to register the bundled extensions. If you don't do this, Mopidy will not find any frontends or backends. Note that we highly recomend you do this in a virtualenv, not system wide. As a bonus, the command also gives you a `mopidy` executable in your search path.

4.3.18 v0.13.0 (2013-03-31)

The 0.13 release brings small improvements and bugfixes throughout Mopidy. There are no major new features, just incremental improvement of what we already have.

Dependencies

- Pykka >= 1.1 is now required.

Core

- Removed the `mopidy.settings.DEBUG_THREAD` setting and the `--debug-thread` command line option. Sending SIGUSR1 to the Mopidy process will now always make it log tracebacks for all alive threads.
- Log a warning if a track isn't playable to make it more obvious that backend X needs backend Y to be present for playback to work.
- *mopidy.core.TracklistController.add()* now accepts an `uri` which it will lookup in the library and then add to the tracklist. This is helpful for e.g. web clients that doesn't want to transfer all track meta data back to the server just to add it to the tracklist when the server already got all the needed information easily available. (Fixes: #325)
- Change the following methods to accept an `uris` keyword argument:
  - *mopidy.core.LibraryController.find_exact()*
  - *mopidy.core.LibraryController.search()*

  Search queries will only be forwarded to backends handling the given URI roots, and the backends may use the URI roots to further limit what results are returned. For example, a search with `uris=['file:']` will only be processed by the local backend. A search with `uris=['file:///media/music']` will only be processed by the local backend, and, if such filtering is supported by the backend, will only return results with URIs within the given URI root.

Audio sub-system

- Make audio error logging handle log messages with non-ASCII chars. (Fixes: #347)

Local backend

- Make `mopidy-scan` work with Ogg Vorbis files. (Fixes: #275)
- Fix playback of files with non-ASCII chars in their file path. (Fixes: #353)

Spotify backend

- Let GStreamer handle time position tracking and seeks. (Fixes: #191)
- For all playlists owned by other Spotify users, we now append the owner's username to the playlist name. (Partly fixes: #114)

HTTP frontend

- Mopidy.js now works both from browsers and from Node.js environments. This means that you now can make Mopidy clients in Node.js. Mopidy.js has been published to the npm registry for easy installation in Node.js projects.
- Upgrade Mopidy.js' build system Grunt from 0.3 to 0.4.

- Upgrade Mopidy.js' dependencies when.js from 1.6.1 to 2.0.0.
- Expose `mopidy.core.Core.get_uri_schemes()` to HTTP clients. It is available through Mopidy.js as `mopidy.getUriSchemes()`.

MPRIS frontend

- Publish album art URIs if available.
- Publish disc number of track if available.

4.3.19 v0.12.0 (2013-03-12)

The 0.12 release has been delayed for a while because of some issues related some ongoing GStreamer cleanup we didn't invest enough time to finish. Finally, we've come to our senses and have now cherry-picked the good parts to bring you a new release, while postponing the GStreamer changes to 0.13. The release adds a new backend for playing audio streams, as well as various minor improvements throughout Mopidy.

- Make Mopidy work on early Python 2.6 versions. (Fixes: #302)
  - `optparse` fails if the first argument to `add_option` is a unicode string on Python < 2.6.2rc1.
  - `foo(**data)` fails if the keys in `data` is unicode strings on Python < 2.6.5rc1.

Audio sub-system

- Improve selection of mixer tracks for volume control. (Fixes: #307)

Local backend

- Make `mopidy-scan` support symlinks.

Stream backend

We've added a new backend for playing audio streams, the `stream backend`. It is activated by default. The stream backend supports the intersection of what your GStreamer installation supports and what protocols are included in the `mopidy.settings.STREAM_PROTOCOLS` setting.

Current limitations:

- No metadata about the current track in the stream is available.
- Playlists are not parsed, so you can't play e.g. a M3U or PLS file which contains stream URIs. You need to extract the stream URL from the playlist yourself. See #303 for progress on this.

Core API

- `mopidy.core.PlaylistsController.get_playlists()` now accepts an argument `include_tracks`. This defaults to `True`, which has the same old behavior. If set to `False`, the tracks are stripped from the playlists before they are returned. This can be used to limit the amount of data returned if the response is to be passed out of the application, e.g. to a web client. (Fixes: #297)

Models

- Add *`mopidy.models.Album.images`* field for including album art URIs. (Partly fixes #263)
- Add *`mopidy.models.Track.disc_no`* field. (Partly fixes: #286)
- Add *`mopidy.models.Album.num_discs`* field. (Partly fixes: #286)

4.3.20 v0.11.1 (2012-12-24)

Spotify search was broken in 0.11.0 for users of Python 2.6. This release fixes it. If you're using Python 2.7, v0.11.0 and v0.11.1 should be equivalent.

4.3.21 v0.11.0 (2012-12-24)

In celebration of Mopidy's three year anniversary December 23, we're releasing Mopidy 0.11. This release brings several improvements, most notably better search which now includes matching artists and albums from Spotify in the search results.

Settings

- The settings validator now complains if a setting which expects a tuple of values (e.g. `mopidy.settings.BACKENDS`, `mopidy.settings.FRONTENDS`) has a non-iterable value. This typically happens because the setting value contains a single value and one has forgotten to add a comma after the string, making the value a tuple. (Fixes: #278)

Spotify backend

- Add `mopidy.settings.SPOTIFY_TIMEOUT` setting which allows you to control how long we should wait before giving up on Spotify searches, etc.
- Add support for looking up albums, artists, and playlists by URI in addition to tracks. (Fixes: #67)

  As an example of how this can be used, you can try the the following MPD commands which now all adds one or more tracks to your tracklist:

```
add "spotify:track:1mwt9hzaH7idmC5UCoOUkz"
add "spotify:album:3gpHG5MGwnipnap32lFYvI"
add "spotify:artist:5TgQ66WuWkoQ2xYxaSTnVP"
add "spotify:user:p3.no:playlist:0XX6tamRiqEgh3t6FPFEkw"
```

- Increase max number of tracks returned by searches from 100 to 200, which seems to be Spotify's current max limit.

Local backend

- Load track dates from tag cache.
- Add support for searching by track date.

MPD frontend

- Add `mopidy.settings.MPD_SERVER_CONNECTION_TIMEOUT` setting which controls how long an MPD client can stay inactive before the connection is closed by the server.
- Add support for the `findadd` command.
- Updated to match the MPD 0.17 protocol (Fixes: #228):
  - Add support for `seekcur` command.
  - Add support for `config` command.
  - Add support for loading a range of tracks from a playlist to the `load` command.
  - Add support for `searchadd` command.
  - Add support for `searchaddpl` command.
  - Add empty stubs for channel commands for client to client communication.
- Add support for search by date.

- Make `seek` and `seekid` not restart the current track before seeking in it.
- Include fake tracks representing albums and artists in the search results. When these are added to the tracklist, they expand to either all tracks in the album or all tracks by the artist. This makes it easy to play full albums in proper order, which is a feature that have been frequently requested. (Fixes: #67, #148)

Internal changes

Models:

- Specified that *mopidy.models.Playlist.last_modified* should be in UTC.
- Added *mopidy.models.SearchResult* model to encapsulate search results consisting of more than just tracks.

Core API:

- Change the following methods to return *mopidy.models.SearchResult* objects which can include both track results and other results:
  - *mopidy.core.LibraryController.find_exact()*
  - *mopidy.core.LibraryController.search()*
- Change the following methods to accept either a dict with filters or kwargs. Previously they only accepted kwargs, which made them impossible to use from the Mopidy.js through JSON-RPC, which doesn't support kwargs.
  - *mopidy.core.LibraryController.find_exact()*
  - *mopidy.core.LibraryController.search()*
  - *mopidy.core.PlaylistsController.filter()*
  - *mopidy.core.TracklistController.filter()*
  - *mopidy.core.TracklistController.remove()*
- Actually trigger the *mopidy.core.CoreListener.volume_changed()* event.
- Include the new volume level in the *mopidy.core.CoreListener.volume_changed()* event.
- The `track_playback_{paused,resumed,started,ended}` events now include a *mopidy.models.TlTrack* instead of a *mopidy.models.Track*.

Audio:

- Mixers with fewer than 100 volume levels could report another volume level than what you just set due to the conversion between Mopidy's 0-100 range and the mixer's range. Now Mopidy returns the recently set volume if the mixer reports a volume level that matches the recently set volume, otherwise the mixer's volume level is rescaled to the 1-100 range and returned.

4.3.22 v0.10.0 (2012-12-12)

We've added an HTTP frontend for those wanting to build web clients for Mopidy!

Dependencies

- pyspotify >= 1.9, < 1.11 is now required for Spotify support. In other words, you're free to upgrade to pyspotify 1.10, but it isn't a requirement.

Documentation

- Added installation instructions for Fedora.

Spotify backend

- Save a lot of memory by reusing artist, album, and track models.
- Make sure the playlist loading hack only runs once.

Local backend

- Change log level from error to warning on messages emitted when the tag cache isn't found and a couple of similar cases.
- Make `mopidy-scan` ignore invalid dates, e.g. dates in years outside the range 1-9999.
- Make `mopidy-scan` accept `-q`/`--quiet` and `-v`/`--verbose` options to control the amount of logging output when scanning.
- The scanner can now handle files with other encodings than UTF-8. Rebuild your tag cache with `mopidy-scan` to include tracks that may have been ignored previously.

HTTP frontend

- Added new optional HTTP frontend which exposes Mopidy's core API through JSON-RPC 2.0 messages over a WebSocket. See *HTTP JSON-RPC API* for further details.
- Added a JavaScript library, Mopidy.js, to make it easier to develop web based Mopidy clients using the new HTTP frontend.

Bug fixes

- #256: Fix crash caused by non-ASCII characters in paths returned from `glib`. The bug can be worked around by overriding the settings that includes offending `$XDG_` variables.

4.3.23 v0.9.0 (2012-11-21)

Support for using the local and Spotify backends simultaneously have for a very long time been our most requested feature. Finally, it's here!

Dependencies

- pyspotify >= 1.9, < 1.10 is now required for Spotify support.

Documentation

- New *Installation* guides, organized by OS and distribution so that you can follow one concise list of instructions instead of jumping around the docs to look for instructions for each dependency.
- Moved *Raspberry Pi: Mopidy on a credit card* howto from the wiki to the docs.
- Updated *MPD clients* overview.
- Added *MPRIS clients* and *UPnP clients* overview.

Multiple backends support

- Both the local backend and the Spotify backend are now turned on by default. The local backend is listed first in the `mopidy.settings.BACKENDS` setting, and are thus given the highest priority in e.g. search results, meaning that we're listing search hits from the local backend first. If you want to prioritize the backends in another way, simply set `BACKENDS` in your own settings file and reorder the backends.

  There are no other setting changes related to the local and Spotify backends. As always, see `mopidy.settings` for the full list of available settings.

Spotify backend

- The Spotify backend now includes release year and artist on albums.
- #233: The Spotify backend now returns the track if you search for the Spotify track URI.

- Added support for connecting to the Spotify service through an HTTP or SOCKS proxy, which is supported by pyspotify >= 1.9.
- Subscriptions to other Spotify user's "starred" playlists are ignored, as they currently isn't fully supported by pyspotify.

Local backend

- #236: The `mopidy-scan` command failed to include tags from ALAC files (Apple lossless) because it didn't support multiple tag messages from GStreamer per track it scanned.
- Added support for search by filename to local backend.

MPD frontend

- #218: The MPD commands `listplaylist` and `listplaylistinfo` now accepts unquoted playlist names if they don't contain spaces.
- #246: The MPD command `list album artist ""` and similar `search`, `find`, and `list` commands with empty filter values caused a `LookupError`, but should have been ignored by the MPD server.
- The MPD frontend no longer lowercases search queries. This broke e.g. search by URI, where casing may be essential.
- The MPD command `plchanges` always returned the entire playlist. It now returns an empty response when the client has seen the latest version.
- The MPD commands `search` and `find` now allows the key `file`, which is used by ncmpcpp instead of `filename`.
- The MPD commands `search` and `find` now allow search query values to be empty strings.
- The MPD command `listplaylists` will no longer return playlists without a name. This could crash ncmpcpp.
- The MPD command `list` will no longer return artist names, album names, or dates that are blank.
- The MPD command `decoders` will now return an empty response instead of a "not implemented" error to make the ncmpcpp browse view work the first time it is opened.

MPRIS frontend

- The MPRIS playlists interface is now supported by our MPRIS frontend. This means that you now can select playlists to queue and play from the Ubuntu Sound Menu.

Audio mixers

- Made the `NAD mixer` responsive to interrupts during amplifier calibration. It will now quit immediately, while previously it completed the calibration first, and then quit, which could take more than 15 seconds.

Developer support

- Added optional background thread for debugging deadlocks. When the feature is enabled via the `--debug-thread` option or `mopidy.settings.DEBUG_THREAD` setting a `SIGUSR1` signal will dump the traceback for all running threads.
- The settings validator will now allow any setting prefixed with `CUSTOM_` to exist in the settings file.

Internal changes

Internally, Mopidy have seen a lot of changes to pave the way for multiple backends and the future HTTP frontend.

- A new layer and actor, "core", has been added to our stack, inbetween the frontends and the backends. The responsibility of the core layer and actor is to take requests from the frontends, pass them on to one or more backends, and combining the response from the backends into a single response to the requesting frontend.

Frontends no longer know anything about the backends. They just use the *Core API*.

- The dependency graph between the core controllers and the backend providers have been straightened out, so that we don't have any circular dependencies. The frontend, core, backend, and audio layers are now strictly separate. The frontend layer calls on the core layer, and the core layer calls on the backend layer. Both the core layer and the backends are allowed to call on the audio layer. Any data flow in the opposite direction is done by broadcasting of events to listeners, through e.g. *mopidy.core.CoreListener* and *mopidy.audio.AudioListener*.

  See *Architecture and concepts* for more details and illustrations of all the relations.

- All dependencies are now explicitly passed to the constructors of the frontends, core, and the backends. This makes testing each layer with dummy/mocked lower layers easier than with the old variant, where dependencies where looked up in Pykka's actor registry.

- All properties in the core API now got getters, and setters if setting them is allowed. They are not explictly listed in the docs as they have the same behavior as the documented properties, but they are available and may be used. This is useful for the future HTTP frontend.

Models:

- Added *mopidy.models.Album.date* attribute. It has the same format as the existing *mopidy.models.Track.date*.

- Added *mopidy.models.ModelJSONEncoder* and *mopidy.models.model_json_decoder()* for automatic JSON serialization and deserialization of data structures which contains Mopidy models. This is useful for the future HTTP frontend.

Library:

- *mopidy.core.LibraryController.find_exact()* and *mopidy.core.LibraryController.search()* now returns plain lists of tracks instead of playlist objects.

- *mopidy.core.LibraryController.lookup()* now returns a list of tracks instead of a single track. This makes it possible to support lookup of artist or album URIs which then can expand to a list of tracks.

Playback:

- The base playback provider has been updated with sane default behavior instead of empty functions. By default, the playback provider now lets GStreamer keep track of the current track's time position. The local backend simply uses the base playback provider without any changes. Any future backend that just feeds URIs to GStreamer to play can also use the base playback provider without any changes.

- Removed `mopidy.core.PlaybackController.track_at_previous`. Use `mopidy.core.PlaybackController.tl_track_at_previous` instead.

- Removed `mopidy.core.PlaybackController.track_at_next`. Use `mopidy.core.PlaybackController.tl_track_at_next` instead.

- Removed `mopidy.core.PlaybackController.track_at_eot`. Use `mopidy.core.PlaybackController.tl_track_at_eot` instead.

- Removed `mopidy.core.PlaybackController.current_tlid`. Use *mopidy.core.PlaybackController.current_tl_track* instead.

Playlists:

The playlists part of the core API has been revised to be more focused around the playlist URI, and some redundant functionality has been removed:

- Renamed "stored playlists" to "playlists" everywhere, including the core API used by frontends.

- *mopidy.core.PlaylistsController.playlists* no longer supports assignment to it. The *playlists* property on the backend layer still does, and all functionality is maintained by assigning to the playlists collections at the backend level.
- *mopidy.core.PlaylistsController.delete()* now accepts an URI, and not a playlist object.
- *mopidy.core.PlaylistsController.save()* now returns the saved playlist. The returned playlist may differ from the saved playlist, and should thus be used instead of the playlist passed to *mopidy.core.PlaylistsController.save()*.
- `mopidy.core.PlaylistsController.rename()` has been removed, since renaming can be done with *mopidy.core.PlaylistsController.save()*.
- `mopidy.core.PlaylistsController.get()` has been replaced by *mopidy.core.PlaylistsController.filter()*.
- The event *mopidy.core.CoreListener.playlist_changed()* has been changed to include the playlist that was changed.

Tracklist:

- Renamed "current playlist" to "tracklist" everywhere, including the core API used by frontends.
- Removed `mopidy.core.TracklistController.append()`. Use *mopidy.core.TracklistController.add()* instead, which is now capable of adding multiple tracks.
- `mopidy.core.TracklistController.get()` has been replaced by *mopidy.core.TracklistController.filter()*.
- *mopidy.core.TracklistController.remove()* can now remove multiple tracks, and returns the tracks it removed.
- When the tracklist is changed, we now trigger the new *mopidy.core.CoreListener.tracklist_changed()* event. Previously we triggered *mopidy.core.CoreListener.playlist_changed()*, which is intended for stored playlists, not the tracklist.

Towards Python 3 support:

- Make the entire code base use unicode strings by default, and only fall back to bytestrings where it is required. Another step closer to Python 3.

4.3.24 v0.8.1 (2012-10-30)

A small maintenance release to fix a bug introduced in 0.8.0 and update Mopidy to work with Pykka 1.0.

Dependencies

- Pykka >= 1.0 is now required.

Bug fixes

- #213: Fix "streaming task paused, reason not-negotiated" errors observed by some users on some Spotify tracks due to a change introduced in 0.8.0. See the issue for a patch that applies to 0.8.0.
- #216: Volume returned by the MPD command *status* contained a floating point `.0` suffix. This bug was introduced with the large audio output and mixer changes in v0.8.0 and broke the MPDroid Android client. It now returns an integer again.

4.3.25 v0.8.0 (2012-09-20)

This release does not include any major new features. We've done a major cleanup of how audio outputs and audio mixers work, and on the way we've resolved a bunch of related issues.

Audio output and mixer changes

- Removed multiple outputs support. Having this feature currently seems to be more trouble than what it is worth. The `mopidy.settings.OUTPUTS` setting is no longer supported, and has been replaced with `mopidy.settings.OUTPUT` which is a GStreamer bin description string in the same format as `gst-launch` expects. Default value is `autoaudiosink`. (Fixes: #81, #115, #121, #159)
- Switch to pure GStreamer based mixing. This implies that users setup a GStreamer bin with a mixer in it in `mopidy.settings.MIXER`. The default value is `autoaudiomixer`, a custom mixer that attempts to find a mixer that will work on your system. If this picks the wrong mixer you can of course override it. Setting the mixer to `None` is also supported. MPD protocol support for volume has also been updated to return -1 when we have no mixer set. `software` can be used to force software mixing.
- Removed the Denon hardware mixer, as it is not maintained.
- Updated the NAD hardware mixer to work in the new GStreamer based mixing regime. Settings are now passed as GStreamer element properties. In practice that means that the following old-style config:

```
MIXER = u'mopidy.mixers.nad.NadMixer'
MIXER_EXT_PORT = u'/dev/ttyUSB0'
MIXER_EXT_SOURCE = u'Aux'
MIXER_EXT_SPEAKERS_A = u'On'
MIXER_EXT_SPEAKERS_B = u'Off'
```

  Now is reduced to simply:

```
MIXER = u'nadmixer port=/dev/ttyUSB0 source=aux speakers-a=on speakers-b=off'
```

  The `port` property defaults to `/dev/ttyUSB0`, and the rest of the properties may be left out if you don't want the mixer to adjust the settings on your NAD amplifier when Mopidy is started.

Changes

- When unknown settings are encountered, we now check if it's similar to a known setting, and suggests to the user what we think the setting should have been.
- Added `--list-deps` option to the `mopidy` command that lists required and optional dependencies, their current versions, and some other information useful for debugging. (Fixes: #74)
- Added `tools/debug-proxy.py` to tee client requests to two backends and diff responses. Intended as a developer tool for checking for MPD protocol changes and various client support. Requires gevent, which currently is not a dependency of Mopidy.
- Support tracks with only release year, and not a full release date, like e.g. Spotify tracks.
- Default value of `LOCAL_MUSIC_PATH` has been updated to be `$XDG_MUSIC_DIR`, which on most systems this is set to `$HOME`. Users of local backend that relied on the old default `~/music` need to update their settings. Note that the code responsible for finding this music now also ignores UNIX hidden files and folders.
- File and path settings now support `$XDG_CACHE_DIR`, `$XDG_DATA_DIR` and `$XDG_MUSIC_DIR` substitution. Defaults for such settings have been updated to use this instead of hidden away defaults.
- Playback is now done using `playbin2` from GStreamer instead of rolling our own. This is the first step towards resolving #171.

Bug fixes

- #72: Created a Spotify track proxy that will switch to using loaded data as soon as it becomes available.
- #150: Fix bug which caused some clients to block Mopidy completely. The bug was caused by some clients sending `close` and then shutting down the connection right away. This trigged a situation in which the connection cleanup code would wait for an response that would never come inside the event loop, blocking everything else.
- #162: Fixed bug when the MPD command `playlistinfo` is used with a track position. Track position and CPID was intermixed, so it would cause a crash if a CPID matching the track position didn't exist.
- Fixed crash on lookup of unknown path when using local backend.
- #189: `LOCAL_MUSIC_PATH` and path handling in rest of settings has been updated so all of the code now uses the correct value.
- Fixed incorrect track URIs generated by M3U playlist parsing code. Generated tracks are now relative to `LOCAL_MUSIC_PATH`.
- #203: Re-add support for software mixing.

4.3.26 v0.7.3 (2012-08-11)

A small maintenance release to fix a crash affecting a few users, and a couple of small adjustments to the Spotify backend.

Changes

- Fixed crash when logging `IOError` exceptions on systems using languages with non-ASCII characters, like French.
- Move the default location of the Spotify cache from *~/.cache/mopidy* to *~/.cache/mopidy/spotify*. You can change this by setting `mopidy.settings.SPOTIFY_CACHE_PATH`.
- Reduce time required to update the Spotify cache on startup. One one system/Spotify account, the time from clean cache to ready for use was reduced from 35s to 12s.

4.3.27 v0.7.2 (2012-05-07)

This is a maintenance release to make Mopidy 0.7 build on systems without all of Mopidy's runtime dependencies, like Launchpad PPAs.

Changes

- Change from version tuple at `mopidy.VERSION` to **PEP 386** compliant version string at `mopidy.__version__` to conform to **PEP 396**.

4.3.28 v0.7.1 (2012-04-22)

This is a maintenance release to make Mopidy 0.7 work with pyspotify >= 1.7.

Changes

- Don't override pyspotify's `notify_main_thread` callback. The default implementation is sensible, while our override did nothing.

4.3.29 v0.7.0 (2012-02-25)

Not a big release with regard to features, but this release got some performance improvements over v0.6, especially for slower Atom systems. It also fixes a couple of other bugs, including one which made Mopidy crash when using GStreamer from the prereleases of Ubuntu 12.04.

Changes

- The MPD command `playlistinfo` is now faster, thanks to John Bäckstrand.
- Added the method `mopidy.backends.base.CurrentPlaylistController.length()`, `mopidy.backends.base.CurrentPlaylistController.index()`, and `mopidy.backends.base.CurrentPlaylistController.slice()` to reduce the need for copying the entire current playlist from one thread to another. Thanks to John Bäckstrand for pinpointing the issue.
- Fix crash on creation of config and cache directories if intermediate directories does not exist. This was especially the case on OS X, where `~/.config` doesn't exist for most users.
- Fix `gst.LinkError` which appeared when using newer versions of GStreamer, e.g. on Ubuntu 12.04 Alpha. (Fixes: #144)
- Fix crash on mismatching quotation in `list` MPD queries. (Fixes: #137)
- Volume is now reported to be the same as the volume was set to, also when internal rounding have been done due to `mopidy.settings.MIXER_MAX_VOLUME` has been set to cap the volume. This should make it possible to manage capped volume from clients that only increase volume with one step at a time, like ncmpcpp does.

4.3.30 v0.6.1 (2011-12-28)

This is a maintenance release to make Mopidy 0.6 work with pyspotify >= 1.5, which Mopidy's develop branch have supported for a long time. This should also make the Debian packages work out of the box again.

Important changes

- pyspotify 1.5 or greater is required.

Changes

- Spotify playlist folder boundaries are now properly detected. In other words, if you use playlist folders, you will no longer get lots of log messages about bad playlists.

4.3.31 v0.6.0 (2011-10-09)

The development of Mopidy have been quite slow for the last couple of months, but we do have some goodies to release which have been idling in the develop branch since the warmer days of the summer. This release brings support for the MPD `idle` command, which makes it possible for a client wait for updates from the server instead of polling every second. Also, we've added support for the MPRIS standard, so that Mopidy can be controlled over D-Bus from e.g. the Ubuntu Sound Menu.

Please note that 0.6.0 requires some updated dependencies, as listed under *Important changes* below.

Important changes

- Pykka 0.12.3 or greater is required.
- pyspotify 1.4 or greater is required.
- All config, data, and cache locations are now based on the XDG spec.
  - This means that your settings file will need to be moved from `~/.mopidy/settings.py` to `~/.config/mopidy/settings.py`.

  - Your Spotify cache will now be stored in `~/.cache/mopidy` instead of `~/.mopidy/spotify_cache`.
  - The local backend's `tag_cache` should now be in `~/.local/share/mopidy/tag_cache`, likewise your playlists will be in `~/.local/share/mopidy/playlists`.
  - The local client now tries to lookup where your music is via XDG, it will fall-back to `~/music` or use whatever setting you set manually.

- The MPD command `idle` is now supported by Mopidy for the following subsystems: player, playlist, options, and mixer. (Fixes: #32)
- A new frontend `mopidy.frontends.mpris` have been added. It exposes Mopidy through the MPRIS interface over D-Bus. In practice, this makes it possible to control Mopidy through the Ubuntu Sound Menu.

Changes

- Replace `mopidy.backends.base.Backend.uri_handlers` with `mopidy.backends.base.Backend.uri_schemes`, which just takes the part up to the colon of an URI, and not any prefix.
- Add Listener API, `mopidy.listeners`, to be implemented by actors wanting to receive events from the backend. This is a formalization of the ad hoc events the Last.fm scrobbler has already been using for some time.
- Replaced all of the MPD network code that was provided by asyncore with custom stack. This change was made to facilitate support for the `idle` command, and to reduce the number of event loops being used.
- Fix metadata update in Shoutcast streaming. (Fixes: #122)
- Unescape all incoming MPD requests. (Fixes: #113)
- Increase the maximum number of results returned by Spotify searches from 32 to 100.
- Send Spotify search queries to pyspotify as unicode objects, as required by pyspotify 1.4. (Fixes: #129)
- Add setting `mopidy.settings.MPD_SERVER_MAX_CONNECTIONS`. (Fixes: #134)
- Remove *destroy()* methods from backend controller and provider APIs, as it was not in use and actually not called by any code. Will reintroduce when needed.

4.3.32 v0.5.0 (2011-06-15)

Since last time we've added support for audio streaming to SHOUTcast servers and fixed the longstanding playlist loading issue in the Spotify backend. As always the release has a bunch of bug fixes and minor improvements.

Please note that 0.5.0 requires some updated dependencies, as listed under *Important changes* below.

Important changes

- If you use the Spotify backend, you *must* upgrade to libspotify 0.0.8 and pyspotify 1.3. If you install from APT, libspotify and pyspotify will automatically be upgraded. If you are not installing from APT, follow the instructions at *Installation*.
- If you have explicitly set the `mopidy.settings.SPOTIFY_HIGH_BITRATE` setting, you must update your settings file. The new setting is named `mopidy.settings.SPOTIFY_BITRATE` and accepts the integer values 96, 160, and 320.
- Mopidy now supports running with 1 to N outputs at the same time. This feature was mainly added to facilitate SHOUTcast support, which Mopidy has also gained. In its current state outputs can not be toggled during runtime.

Changes

- Local backend:

  - Fix local backend time query errors that where coming from stopped pipeline. (Fixes: #87)

- Spotify backend:
  - Thanks to Antoine Pierlot-Garcin's recent work on updating and improving pyspotify, stored playlists will again load when Mopidy starts. The workaround of searching and reconnecting to make the playlists appear are no longer necessary. (Fixes: #59)
  - Track's that are no longer available in Spotify's archives are now "autolinked" to corresponding tracks in other albums, just like the official Spotify clients do. (Fixes: #34)
- MPD frontend:
  - Refactoring and cleanup. Most notably, all request handlers now get an instance of `mopidy.frontends.mpd.dispatcher.MpdContext` as the first argument. The new class contains reference to any object in Mopidy the MPD protocol implementation should need access to.
  - Close the client connection when the command `close` is received.
  - Do not allow access to the command `kill`.
  - `commands` and `notcommands` now have correct output if password authentication is turned on, but the connected user has not been authenticated yet.
- Command line usage:
  - Support passing options to GStreamer. See `--help-gst` for a list of available options. (Fixes: #95)
  - Improve `--list-settings` output. (Fixes: #91)
  - Added `--interactive` for reading missing local settings from `stdin`. (Fixes: #96)
  - Improve shutdown procedure at CTRL+C. Add signal handler for `SIGTERM`, which initiates the same shutdown procedure as CTRL+C does.
- Tag cache generator:
  - Made it possible to abort **`mopidy-scan`** with CTRL+C.
  - Fixed bug regarding handling of bad dates.
  - Use `logging` instead of `print` statements.
  - Found and worked around strange WMA metadata behaviour.
- Backend API:
  - Calling on `mopidy.backends.base.playback.PlaybackController.next()` and `mopidy.backends.base.playback.PlaybackController.previous()` no longer implies that playback should be started. The playback state–whether playing, paused or stopped–will now be kept.
  - The method `mopidy.backends.base.playback.PlaybackController.change_track()` has been added. Like `next()`, and `prev()`, it changes the current track without changing the playback state.

4.3.33 v0.4.1 (2011-05-06)

This is a bug fix release fixing audio problems on older GStreamer and some minor bugs.

Bug fixes

- Fix broken audio on at least GStreamer 0.10.30, which affects Ubuntu 10.10. The GStreamer *appsrc* bin wasn't being linked due to lack of default caps. (Fixes: #85)

- Fix crash in `mopidy.mixers.nad` that occures at startup when the `io` module is available. We used an *eol* keyword argument which is supported by `serial.FileLike.readline()`, but not by `io.RawBaseIO.readline()`. When the `io` module is available, it is used by PySerial instead of the *FileLike* implementation.
- Fix UnicodeDecodeError in MPD frontend on non-english locale. Thanks to Antoine Pierlot-Garcin for the patch. (Fixes: #88)
- Do not create Pykka proxies that are not going to be used in *`mopidy.core`*. The underlying actor may already intentionally be dead, and thus the program may crash on creating a proxy it doesn't need. Combined with the Pykka 0.12.2 release this fixes a crash in the Last.fm frontend which may occur when all dependencies are installed, but the frontend isn't configured. (Fixes: #84)

4.3.34 v0.4.0 (2011-04-27)

Mopidy 0.4.0 is another release without major feature additions. In 0.4.0 we've fixed a bunch of issues and bugs, with the help of several new contributors who are credited in the changelog below. The major change of 0.4.0 is an internal refactoring which clears way for future features, and which also make Mopidy work on Python 2.7. In other words, Mopidy 0.4.0 works on Ubuntu 11.04 and Arch Linux.

Please note that 0.4.0 requires some updated dependencies, as listed under *Important changes* below. Also, the known bug in the Spotify playlist loading from Mopidy 0.3.0 is still present.

> **Warning:** Known bug in Spotify playlist loading
>
> There is a known bug in the loading of Spotify playlists. To avoid the bug, follow the simple workaround described at #59.

Important changes

- Mopidy now depends on Pykka >=0.12. If you install from APT, Pykka will automatically be installed. If you are not installing from APT, you may install Pykka from PyPI:

```
sudo pip install -U Pykka
```

- If you use the Spotify backend, you *should* upgrade to libspotify 0.0.7 and the latest pyspotify from the Mopidy developers. If you install from APT, libspotify and pyspotify will automatically be upgraded. If you are not installing from APT, follow the instructions at *Installation*.

Changes

- Mopidy now use Pykka actors for thread management and inter-thread communication. The immediate advantage of this is that Mopidy now works on Python 2.7, which is the default on e.g. Ubuntu 11.04. (Fixes: #66)
- Spotify backend:
  - Fixed multiple segmentation faults due to bugs in Pyspotify. Thanks to Antoine Pierlot-Garcin and Jamie Kirkpatrick for patches to Pyspotify.
  - Better error messages on wrong login or network problems. Thanks to Antoine Pierlot-Garcin for patches to Mopidy and Pyspotify. (Fixes: #77)
  - Reduce log level for trivial log messages from warning to info. (Fixes: #71)
  - Pause playback on network connection errors. (Fixes: #65)
- Local backend:

  - Fix crash in **mopidy-scan** if a track has no artist name. Thanks to Martins Grunskis for test and patch and "octe" for patch.
  - Fix crash in *tag_cache* parsing if a track has no total number of tracks in the album. Thanks to Martins Grunskis for the patch.
- MPD frontend:
  - Add support for "date" queries to both the `find` and `search` commands. This makes media library browsing in ncmpcpp work, though very slow due to all the meta data requests to Spotify.
  - Add support for `play "-1"` when in playing or paused state, which fixes resume and addition of tracks to the current playlist while playing for the MPoD client.
  - Fix bug where `status` returned `song: None`, which caused MPDroid to crash. (Fixes: #69)
  - Gracefully fallback to IPv4 sockets on systems that supports IPv6, but has turned it off. (Fixes: #75)
- GStreamer output:
  - Use `uridecodebin` for playing audio from both Spotify and the local backend. This contributes to support for multiple backends simultaneously.
- Settings:
  - Fix crash on `--list-settings` on clean installation. Thanks to Martins Grunskis for the bug report and patch. (Fixes: #63)
- Packaging:
  - Replace test data symlinks with real files to avoid symlink issues when installing with pip. (Fixes: #68)
- Debugging:
  - Include platform, architecture, Linux distribution, and Python version in the debug log, to ease debugging of issues with attached debug logs.

4.3.35 v0.3.1 (2011-01-22)

A couple of fixes to the 0.3.0 release is needed to get a smooth installation.

Bug fixes

- The Spotify application key was missing from the Python package.
- Installation of the Python package as a normal user failed because it did not have permissions to install `mopidy.desktop`. The file is now only installed if the installation is executed as the root user.

4.3.36 v0.3.0 (2011-01-22)

Mopidy 0.3.0 brings a bunch of small changes all over the place, but no large changes. The main features are support for high bitrate audio from Spotify, and MPD password authentication.

Regarding the docs, we've improved the *installation instructions* and done a bit of testing of the available *Android* and *iOS clients* for MPD.

Please note that 0.3.0 requires some updated dependencies, as listed under *Important changes* below. Also, there is a known bug in the Spotify playlist loading, as described below. As the bug will take some time to fix and has a known workaround, we did not want to delay the release while waiting for a fix to this problem.

Warning: Known bug in Spotify playlist loading

There is a known bug in the loading of Spotify playlists. This bug affects both Mopidy 0.2.1 and 0.3.0, given that you use libspotify 0.0.6. To avoid the bug, either use Mopidy 0.2.1 with libspotify 0.0.4, or use either Mopidy version with libspotify 0.0.6 and follow the simple workaround described at #59.

Important changes

- If you use the Spotify backend, you need to upgrade to libspotify 0.0.6 and the latest pyspotify from the Mopidy developers. Follow the instructions at *Installation*.
- If you use the Last.fm frontend, you need to upgrade to pylast 0.5.7. Run `sudo pip install --upgrade pylast` or install Mopidy from APT.

Changes

- Spotify backend:
  - Support high bitrate (320k) audio. Set the new setting `mopidy.settings.SPOTIFY_HIGH_BITRATE` to `True` to switch to high bitrate audio.
  - Rename `mopidy.backends.libspotify` to `mopidy.backends.spotify`. If you have set `mopidy.settings.BACKENDS` explicitly, you may need to update the setting's value.
  - Catch and log error caused by playlist folder boundaries being threated as normal playlists. More permanent fix requires support for checking playlist types in pyspotify (see #62).
  - Fix crash on failed lookup of track by URI. (Fixes: #60)
- Local backend:
  - Add **`mopidy-scan`** command to generate `tag_cache` files without any help from the original MPD server. See *Generating a local library* for instructions on how to use it.
  - Fix support for UTF-8 encoding in tag caches.
- MPD frontend:
  - Add support for password authentication. See `mopidy.settings.MPD_SERVER_PASSWORD` for details on how to use it. (Fixes: #41)
  - Support `setvol 50` without quotes around the argument. Fixes volume control in Droid MPD.
  - Support `seek 1 120` without quotes around the arguments. Fixes seek in Droid MPD.
- Last.fm frontend:
  - Update to use Last.fm's new Scrobbling 2.0 API, as the old Submissions Protocol 1.2.1 is deprecated. (Fixes: #33)
  - Fix crash when track object does not contain all the expected meta data.
  - Fix crash when response from Last.fm cannot be decoded as UTF-8. (Fixes: #37)
  - Fix crash when response from Last.fm contains invalid XML.
  - Fix crash when response from Last.fm has an invalid HTTP status line.
- Mixers:
  - Support use of unicode strings for settings specific to `mopidy.mixers.nad`.
- Settings:
  - Automatically expand the "~" characted to the user's home directory and make the path absolute for settings with names ending in `_PATH` or `_FILE`.

  - Rename the following settings. The settings validator will warn you if you need to change your local settings.
    * `LOCAL_MUSIC_FOLDER` to `mopidy.settings.LOCAL_MUSIC_PATH`
    * `LOCAL_PLAYLIST_FOLDER` to `mopidy.settings.LOCAL_PLAYLIST_PATH`
    * `LOCAL_TAG_CACHE` to `mopidy.settings.LOCAL_TAG_CACHE_FILE`
    * `SPOTIFY_LIB_CACHE` to `mopidy.settings.SPOTIFY_CACHE_PATH`
  - Fix bug which made settings set to `None` or 0 cause a `mopidy.SettingsError` to be raised.
- Packaging and distribution:
  - Setup APT repository and create Debian packages of Mopidy. See *Installation* for instructions for how to install Mopidy, including all dependencies, from APT.
  - Install `mopidy.desktop` file that makes Mopidy available from e.g. Gnome application menus.
- API:
  - Rename and generalize `Playlist._with(**kwargs)` to *mopidy.models.ImmutableObject.copy()*.
  - Add `musicbrainz_id` field to *mopidy.models.Artist*, *mopidy.models.Album*, and *mopidy.models.Track*.
  - Prepare for multi-backend support (see #40) by introducing the *provider concept*. Split the backend API into a *backend controller API* (for frontend use) and a *backend provider API* (for backend implementation use), which includes the following changes:
    * Rename `BaseBackend` to `mopidy.backends.base.Backend`.
    * Rename `BaseCurrentPlaylistController` to `mopidy.backends.base.CurrentPlaylistController`.
    * Split `BaseLibraryController` to `mopidy.backends.base.LibraryController` and `mopidy.backends.base.BaseLibraryProvider`.
    * Split `BasePlaybackController` to `mopidy.backends.base.PlaybackController` and `mopidy.backends.base.BasePlaybackProvider`.
    * Split `BaseStoredPlaylistsController` to `mopidy.backends.base.StoredPlaylistsController` and `mopidy.backends.base.BaseStoredPlaylistsProvider`.
  - Move `BaseMixer` to `mopidy.mixers.base.BaseMixer`.
  - Add docs for the current non-stable output API, `mopidy.outputs.base.BaseOutput`.

4.3.37 v0.2.1 (2011-01-07)

This is a maintenance release without any new features.

Bug fixes

- Fix crash in `mopidy.frontends.lastfm` which occurred at playback if either `pylast` was not installed or the Last.fm scrobbling was not correctly configured. The scrobbling thread now shuts properly down at failure.

4.3.38 v0.2.0 (2010-10-24)

In Mopidy 0.2.0 we've added a Last.fm scrobbling support, which means that Mopidy now can submit meta data about the tracks you play to your Last.fm profile. See `mopidy.frontends.lastfm` for details on new dependencies and settings. If you use Mopidy's Last.fm support, please join the Mopidy group at Last.fm.

With the exception of the work on the Last.fm scrobbler, there has been a couple of quiet months in the Mopidy camp. About the only thing going on, has been stabilization work and bug fixing. All bugs reported on GitHub, plus some, have been fixed in 0.2.0. Thus, we hope this will be a great release!

We've worked a bit on OS X support, but not all issues are completely solved yet. #25 is the one that is currently blocking OS X support. Any help solving it will be greatly appreciated!

Finally, please *update your pyspotify installation* when upgrading to Mopidy 0.2.0. The latest pyspotify got a fix for the segmentation fault that occurred when playing music and searching at the same time, thanks to Valentin David.

Important changes

- Added a Last.fm scrobbler. See `mopidy.frontends.lastfm` for details.

Changes

- Logging and command line options:
  - Simplify the default log format, `mopidy.settings.CONSOLE_LOG_FORMAT`. From a user's point of view: Less noise, more information.
  - Rename the `--dump` command line option to `--save-debug-log`.
  - Rename setting `mopidy.settings.DUMP_LOG_FORMAT` to `mopidy.settings.DEBUG_LOG_FORMAT` and use it for `--verbose` too.
  - Rename setting `mopidy.settings.DUMP_LOG_FILENAME` to `mopidy.settings.DEBUG_LOG_FILENAME`.
- MPD frontend:
  - MPD command `list` now supports queries by artist, album name, and date, as used by e.g. the Ario client. (Fixes: #20)
  - MPD command `add ""` and `addid ""` now behaves as expected. (Fixes #16)
  - MPD command `playid "-1"` now correctly resumes playback if paused.
- Random mode:
  - Fix wrong behavior on end of track and next after random mode has been used. (Fixes: #18)
  - Fix infinite recursion loop crash on playback of non-playable tracks when in random mode. (Fixes #17)
  - Fix assertion error that happened if one removed tracks from the current playlist, while in random mode. (Fixes #22)
- Switched from using subprocesses to threads. (Fixes: #14)
- `mopidy.outputs.gstreamer`: Set `caps` on the `appsrc` bin before use. This makes sound output work with GStreamer >= 0.10.29, which includes the versions used in Ubuntu 10.10 and on OS X if using Homebrew. (Fixes: #21, #24, contributes to #14)
- Improved handling of uncaught exceptions in threads. The entire process should now exit immediately.

4.3.39 v0.1.0 (2010-08-23)

After three weeks of long nights and sprints we're finally pleased enough with the state of Mopidy to remove the alpha label, and do a regular release.

Mopidy 0.1.0 got important improvements in search functionality, working track position seeking, no known stability issues, and greatly improved MPD client support. There are lots of changes since 0.1.0a3, and we urge you to at least read the *important changes* below.

This release does not support OS X. We're sorry about that, and are working on fixing the OS X issues for a future release. You can track the progress at #14.

Important changes

- License changed from GPLv2 to Apache License, version 2.0.
- GStreamer is now a required dependency. See our *GStreamer installation docs*.
- `mopidy.backends.libspotify` is now the default backend. `mopidy.backends.despotify` is no longer available. This means that you need to install the *dependencies for libspotify*.
- If you used `mopidy.backends.libspotify` previously, pyspotify must be updated when updating to this release, to get working seek functionality.
- `mopidy.settings.SERVER_HOSTNAME` and `mopidy.settings.SERVER_PORT` has been renamed to `mopidy.settings.MPD_SERVER_HOSTNAME` and `mopidy.settings.MPD_SERVER_PORT` to allow for multiple frontends in the future.

Changes

- Exit early if not Python >= 2.6, < 3.
- Validate settings at startup and print useful error messages if the settings has not been updated or anything is misspelled.
- Add command line option `--list-settings` to print the currently active settings.
- Include Sphinx scripts for building docs, pylintrc, tests and test data in the packages created by `setup.py` for i.e. PyPI.
- MPD frontend:
  - Search improvements, including support for multi-word search.
  - Fixed `play "-1"` and `playid "-1"` behaviour when playlist is empty or when a current track is set.
  - Support `plchanges "-1"` to work better with MPDroid.
  - Support `pause` without arguments to work better with MPDroid.
  - Support `plchanges`, `play`, `consume`, `random`, `repeat`, and `single` without quotes to work better with BitMPC.
  - Fixed deletion of the currently playing track from the current playlist, which crashed several clients.
  - Implement `seek` and `seekid`.
  - Fix `playlistfind` output so the correct song is played when playing songs directly from search results in GMPC.
  - Fix `load` so that one can append a playlist to the current playlist, and make it return the correct error message if the playlist is not found.
  - Support for single track repeat added. (Fixes: #4)
  - Relocate from *`mopidy.mpd`* to `mopidy.frontends.mpd`.

  - Split gigantic protocol implementation into eleven modules.
  - Rename `mopidy.frontends.mpd.{serializer => translator}` to match naming in backends.
  - Remove setting `mopidy.settings.SERVER` and `mopidy.settings.FRONTEND` in favour of the new `mopidy.settings.FRONTENDS`.
  - Run MPD server in its own process.
- Backends:
  - Rename `mopidy.backends.gstreamer` to `mopidy.backends.local`.
  - Remove `mopidy.backends.despotify`, as Despotify is little maintained and the Libspotify backend is working much better. (Fixes: #9, #10, #13)
  - A Spotify application key is now bundled with the source. `mopidy.settings.SPOTIFY_LIB_APPKEY` is thus removed.
  - If failing to play a track, playback will skip to the next track.
  - Both `mopidy.backends.libspotify` and `mopidy.backends.local` have been rewritten to use the new common GStreamer audio output module, `mopidy.outputs.gstreamer`.
- Mixers:
  - Added new `mopidy.mixers.gstreamer_software.GStreamerSoftwareMixer` which now is the default mixer on all platforms.
  - New setting `mopidy.settings.MIXER_MAX_VOLUME` for capping the maximum output volume.
- Backend API:
  - Relocate from `mopidy.backends` to `mopidy.backends.base`.
  - The `id` field of *`mopidy.models.Track`* has been removed, as it is no longer needed after the CPID refactoring.
  - `mopidy.backends.base.BaseBackend()` now accepts an `output_queue` which it can use to send messages (i.e. audio data) to the output process.
  - `mopidy.backends.base.BaseLibraryController.find_exact()` now accepts keyword arguments of the form `find_exact(artist=['foo'], album=['bar'])`.
  - `mopidy.backends.base.BaseLibraryController.search()` now accepts keyword arguments of the form `search(artist=['foo', 'fighters'], album=['bar', 'grooves'])`.
  - `mopidy.backends.base.BaseCurrentPlaylistController.append()` replaces `mopidy.backends.base.BaseCurrentPlaylistController.load()`. Use `mopidy.backends.base.BaseCurrentPlaylistController.clear()` if you want to clear the current playlist.
  - The following fields in `mopidy.backends.base.BasePlaybackController` has been renamed to reflect their relation to methods called on the controller:
    * `next_track` to `track_at_next`
    * `next_cp_track` to `cp_track_at_next`
    * `previous_track` to `track_at_previous`
    * `previous_cp_track` to `cp_track_at_previous`

  - `mopidy.backends.base.BasePlaybackController.track_at_eot` and `mopidy.backends.base.BasePlaybackController.cp_track_at_eot` has been added to better handle the difference between the user pressing next and the current track ending.
  - Rename `mopidy.backends.base.BasePlaybackController.new_playlist_loaded_callback()` to `mopidy.backends.base.BasePlaybackController.on_current_playlist_change()`.
  - Rename `mopidy.backends.base.BasePlaybackController.end_of_track_callback()` to `mopidy.backends.base.BasePlaybackController.on_end_of_track()`.
  - Remove `mopidy.backends.base.BaseStoredPlaylistsController.search()` since it was barely used, untested, and we got no use case for non-exact search in stored playlists yet. Use `mopidy.backends.base.BaseStoredPlaylistsController.get()` instead.

4.3.40 v0.1.0a3 (2010-08-03)

In the last two months, Mopidy's MPD frontend has gotten lots of stability fixes and error handling improvements, proper support for having the same track multiple times in a playlist, and support for IPv6. We have also fixed the choppy playback on the libspotify backend. For the road ahead of us, we got an updated release roadmap with our goals for the 0.1 to 0.3 releases.

Enjoy the best alpha relase of Mopidy ever :-)

Changes

- MPD frontend:
  - Support IPv6.
  - `addid` responds properly on errors instead of crashing.
  - `commands` support, which makes RelaXXPlayer work with Mopidy. (Fixes: #6)
  - Does no longer crash on invalid data, i.e. non-UTF-8 data.
  - `ACK` error messages are now MPD-compliant, which should make clients handle errors from Mopidy better.
  - Requests to existing commands with wrong arguments are no longer reported as unknown commands.
  - `command_list_end` before `command_list_start` now returns unknown command error instead of crashing.
  - `list` accepts field argument without quotes and capitalized, to work with GMPC and ncmpc.
  - `noidle` command now returns `OK` instead of an error. Should make some clients work a bit better.
  - Having multiple identical tracks in a playlist is now working properly. (CPID refactoring)
- Despotify backend:
  - Catch and log `spytify.SpytifyError`. (Fixes: #11)
- Libspotify backend:
  - Fix choppy playback using the Libspotify backend by using blocking ALSA mode. (Fixes: #7)
- Backend API:
  - A new data structure called `cp_track` is now used in the current playlist controller and the playback controller. A `cp_track` is a two-tuple of (CPID integer, *`mopidy.models.Track`*), identifying an instance of a track uniquely within the current playlist.

- `mopidy.backends.BaseCurrentPlaylistController.load()` now accepts lists of *`mopidy.models.Track`* instead of *`mopidy.models.Playlist`*, as none of the other fields on the `Playlist` model was in use.
- `mopidy.backends.BaseCurrentPlaylistController.add()` now returns the `cp_track` added to the current playlist.
- `mopidy.backends.BaseCurrentPlaylistController.remove()` now takes criterias, just like `mopidy.backends.BaseCurrentPlaylistController.get()`.
- `mopidy.backends.BaseCurrentPlaylistController.get()` now returns a `cp_track`.
- `mopidy.backends.BaseCurrentPlaylistController.tracks` is now read-only. Use the methods to change its contents.
- `mopidy.backends.BaseCurrentPlaylistController.cp_tracks` is a read-only list of `cp_track`. Use the methods to change its contents.
- `mopidy.backends.BasePlaybackController.current_track` is now just for convenience and read-only. To set the current track, assign a `cp_track` to `mopidy.backends.BasePlaybackController.current_cp_track`.
- `mopidy.backends.BasePlaybackController.current_cpid` is the read-only CPID of the current track.
- `mopidy.backends.BasePlaybackController.next_cp_track` is the next `cp_track` in the playlist.
- `mopidy.backends.BasePlaybackController.previous_cp_track` is the previous `cp_track` in the playlist.
- `mopidy.backends.BasePlaybackController.play()` now takes a `cp_track`.

4.3.41 v0.1.0a2 (2010-06-02)

It has been a rather slow month for Mopidy, but we would like to keep up with the established pace of at least a release per month.

Changes

- Improvements to MPD protocol handling, making Mopidy work much better with a group of clients, including ncmpc, MPoD, and Theremin.
- New command line flag `--dump` for dumping debug log to `dump.log` in the current directory.
- New setting `mopidy.settings.MIXER_ALSA_CONTROL` for forcing what ALSA control `mopidy.mixers.alsa.AlsaMixer` should use.

4.3.42 v0.1.0a1 (2010-05-04)

Since the previous release Mopidy has seen about 300 commits, more than 200 new tests, a libspotify release, and major feature additions to Spotify. The new releases from Spotify have lead to updates to our dependencies, and also to new bugs in Mopidy. Thus, this is primarily a bugfix release, even though the not yet finished work on a GStreamer backend have been merged.

All users are recommended to upgrade to 0.1.0a1, and should at the same time ensure that they have the latest versions of our dependencies: Despotify r508 if you are using DespotifyBackend, and pyspotify 1.1 with libspotify 0.0.4 if you are using LibspotifyBackend.

As always, report problems at our IRC channel or our issue tracker. Thanks!

Changes

- Backend API changes:
  - Removed `backend.playback.volume` wrapper. Use `backend.mixer.volume` directly.
  - Renamed `backend.playback.playlist_position` to `current_playlist_position` to match naming of `current_track`.
  - Replaced `get_by_id()` with a more flexible `get(**criteria)`.
- Merged the `gstreamer` branch from Thomas Adamcik:
  - More than 200 new tests, and thus several bug fixes to existing code.
  - Several new generic features, like shuffle, consume, and playlist repeat. (Fixes: #3)
  - **[Work in Progress]** A new backend for playing music from a local music archive using the GStreamer library.
- Made `mopidy.mixers.alsa.AlsaMixer` work on machines without a mixer named "Master".
- Make `mopidy.backends.DespotifyBackend` ignore local files in playlists (feature added in Spotify 0.4.3). Reported by Richard Haugen Olsen.
- And much more.

4.3.43 v0.1.0a0 (2010-03-27)

"*Release early. Release often. Listen to your customers.*" wrote Eric S. Raymond in *The Cathedral and the Bazaar*.

Three months of development should be more than enough. We have more to do, but Mopidy is working and usable. 0.1.0a0 is an alpha release, which basicly means we will still change APIs, add features, etc. before the final 0.1.0 release. But the software is usable as is, so we release it. Please give it a try and give us feedback, either at our IRC channel or through the issue tracker. Thanks!

Changes

- Initial version. No changelog available.

4.4 Versioning

Mopidy uses Semantic Versioning, but since we're still pre-1.0 that doesn't mean much yet.

4.4.1 Release schedule

We intend to have about one feature release every month in periods of active development. The feature releases are numbered 0.x.0. The features added is a mix of what we feel is most important/requested of the missing features, and features we develop just because we find them fun to make, even though they may be useful for very few users or for a limited use case.

Bugfix releases, numbered 0.x.y, will be released whenever we discover bugs that are too serious to wait for the next feature release. We will only release bugfix releases for the last feature release. E.g. when 0.14.0 is released, we will no longer provide bugfix releases for the 0.13 series. In other words, there will be just a single supported release at any point in time. This is to not spread our limited resources too thin.

CHAPTER 5

Development

5.1 Contributing

If you are thinking about making Mopidy better, or you just want to hack on it, that's great. Here are some tips to get you started.

5.1.1 Getting started

1. Make sure you have a GitHub account.
2. Submit a ticket for your issue, assuming one does not already exist. Clearly describe the issue including steps to reproduce when it is a bug.
3. Fork the repository on GitHub.

5.1.2 Making changes

1. Clone your fork on GitHub to your computer.
2. Consider making a Python virtualenv for Mopidy development to wall of Mopidy and it's dependencies from the rest of your system. If you do so, create the virtualenv with the `--system-site-packages` flag so that Mopidy can use globally installed dependencies like GStreamer. If you don't use a virtualenv, you may need to run the following `pip` and `python setup.py` commands with `sudo` to install stuff globally on your computer.
3. Install dependencies as described in the *Installation* section.
4. Install additional development dependencies:

```
pip install -r dev-requirements.txt
```

5. Checkout a new branch (usually based on `develop`) and name it accordingly to what you intend to do.
   - Features get the prefix `feature/`

- Bug fixes get the prefix `fix/`
- Improvements to the documentation get the prefix `docs/`

5.1.3 Running Mopidy from Git

If you want to hack on Mopidy, you should run Mopidy directly from the Git repo.

1. Go to the Git repo root:

   ```
   cd mopidy/
   ```

2. To get a `mopidy` executable and register all bundled extensions with setuptools, run:

   ```
   python setup.py develop
   ```

   It still works to run `python mopidy` directly on the `mopidy` Python package directory, but if you have never run `python setup.py develop` the extensions bundled with Mopidy isn't registered with setuptools, so Mopidy will start without any frontends or backends, making it quite useless.

3. Now you can run the Mopidy command, and it will run using the code in the Git repo:

   ```
   mopidy
   ```

   If you do any changes to the code, you'll just need to restart `mopidy` to see the changes take effect.

5.1.4 Testing

Mopidy has quite good test coverage, and we would like all new code going into Mopidy to come with tests.

1. To run all tests, go to the project directory and run:

   ```
   nosetests
   ```

   To run tests with test coverage statistics:

   ```
   nosetests --with-coverage
   ```

   Test coverage statistics can also be viewed online at coveralls.io.

2. Always check the code for errors and style issues using flake8:

   ```
   flake8
   ```

   If successful, the command will not print anything at all.

3. Finally, there is the ultimate but a bit slower command. To run both tests, docs build, and flake8 linting, run:

   ```
   tox
   ```

   This will run exactly the same tests as Travis CI runs for all our branches and pull requests. If this command turns green, you can be quite confident that your pull request will get the green flag from Travis as well, which is a requirement for it to be merged.

5.1.5 Submitting changes

- One branch per feature or fix. Keep branches small and on topic.
- Follow the *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
- Send a pull request to the `develop` branch. See the GitHub pull request docs for help.

5.1.6 Additional resources

- IRC channel: `#mopidy` at irc.freenode.net
- Issue tracker
- Mailing List
- GitHub documentation

5.2 Development tools

Here you'll find description of the development tools we use.

5.2.1 Continuous integration

Mopidy uses the free service Travis CI for automatically running the test suite when code is pushed to GitHub. This works both for the main Mopidy repo, but also for any forks. This way, any contributions to Mopidy through GitHub will automatically be tested by Travis CI, and the build status will be visible in the GitHub pull request interface, making it easier to evaluate the quality of pull requests.

In addition, we run a Jenkins CI server at http://ci.mopidy.com/ that runs all test on multiple platforms (Ubuntu, OS X, x86, arm) for every commit we push to the `develop` branch in the main Mopidy repo on GitHub. Thus, new code isn't tested by Jenkins before it is merged into the `develop` branch, which is a bit late, but good enough to get broad testing before new code is released.

In addition to running tests, the Jenkins CI server also gathers coverage statistics and uses flake8 to check for errors and possible improvements in our code. So, if you're out of work, the code coverage and flake8 data at the CI server should give you a place to start.

5.2.2 Documentation writing

To write documentation, we use Sphinx. See their site for lots of documentation on how to use Sphinx. To generate HTML from the documentation files, you need some additional dependencies.

You can install them through Debian/Ubuntu package management:

```
sudo apt-get install python-sphinx python-pygraphviz graphviz
```

Then, to generate docs:

```
cd docs/
make          # For help on available targets
make html     # To generate HTML docs
```

The documentation at http://docs.mopidy.com/ is automatically updated when a documentation update is pushed to `mopidy/mopidy` at GitHub.

5.2.3 Creating releases

1. Update changelog and commit it.
2. Bump the version number in `mopidy/__init__.py`. Remember to update the test case in `tests/test_version.py`.
3. Merge the release branch (`develop` in the example) into master:

   ```
   git checkout master
   git merge --no-ff -m "Release v0.16.0" develop
   ```

4. Install/upgrade tools used for packaging:

   ```
   pip install -U twine wheel
   ```

5. Build package and test it manually in a new virtualenv. The following assumes the use of virtualenvwrapper:

   ```
   python setup.py sdist bdist_wheel

   mktmpenv
   pip install path/to/dist/Mopidy-0.16.0.tar.gz
   toggleglobalsitepackages
   # do manual test
   deactivate

   mktmpenv
   pip install path/to/dist/Mopidy-0.16.0-py27-none-any.whl
   toggleglobalsitepackages
   # do manual test
   deactivate
   ```

6. Tag the release:

   ```
   git tag -a -m "Release v0.16.0" v0.16.0
   ```

7. Push to GitHub:

   ```
   git push
   git push --tags
   ```

8. Upload the previously built and tested sdist and bdist_wheel packages to PyPI:

   ```
   twine upload dist/Mopidy-0.16.0*
   ```

9. Merge `master` back into `develop` and push the branch to GitHub.
10. Make sure the new tag is built by Read the Docs, and that the `latest` version shows the newly released version.
11. Spread the word through the topic on #mopidy on IRC, @mopidy on Twitter, and on the mailing list.

12. Update the Debian package.

5.2.4 Updating Debian packages

This howto is not intended to learn you all the details, just to give someone already familiar with Debian packaging an overview of how Mopidy's Debian packages is maintained.

1. Install the basic packaging tools:

```
sudo apt-get install build-essential git-buildpackage
```

2. Create a Wheezy pbuilder env if running on Ubuntu and this the first time. See #561 for details about why this is needed:

```
DIST=wheezy sudo git-pbuilder update --mirror=http://mirror.rackspace.com/debian/␣
↪--debootstrapopts --keyring=/usr/share/keyrings/debian-archive-keyring.gpg
```

3. Check out the `debian` branch of the repo:

```
git checkout -t origin/debian
git pull
```

4. Merge the latest release tag into the `debian` branch:

```
git merge v0.16.0
```

5. Update the `debian/changelog` with a "New upstream release" entry:

```
dch -v 0.16.0-0mopidy1
git add debian/changelog
git commit -m "debian: New upstream release"
```

6. Check if any dependencies in `debian/control` or similar needs updating.
7. Install any Build-Deps listed in `debian/control`.
8. Build the package and fix any issues repeatedly until the build succeeds and the Lintian check at the end of the build is satisfactory:

```
git buildpackage -uc -us
```

   If you are using the pbuilder make sure this command is:

```
sudo git buildpackage -uc -us --git-ignore-new --git-pbuilder --git-dist=wheezy --
↪git-no-pbuilder-autoconf
```

9. Install and test newly built package:

```
sudo debi
```

   Again for pbuilder use:

```
sudo debi --debs-dir /var/cache/pbuilder/result/
```

10. If everything is OK, build the package a final time to tag the package version:

```
git buildpackage -uc -us --git-tag
```

Pbuilder:

```
sudo git buildpackage -uc -us --git-ignore-new --git-pbuilder --git-dist=wheezy --
↪git-no-pbuilder-autoconf --git-tag
```

11. Push the changes you've done to the `debian` branch and the new tag:

```
git push
git push --tags
```

12. If you're building for multiple architectures, checkout the `debian` branch on the other builders and run:

```
git buildpackage -uc -us
```

Modify as above to use the pbuilder as needed.

13. Copy files to the APT server. Make sure to select the correct part of the repo, e.g. main, contrib, or non-free:

```
scp ../mopidy*_0.16* bonobo.mopidy.com:/srv/apt.mopidy.com/app/incoming/stable/
↪main
```

14. Update the APT repo:

```
ssh bonobo.mopidy.com
/srv/apt.mopidy.com/app/update.sh
```

15. Test installation from apt.mopidy.com:

```
sudo apt-get update
sudo apt-get dist-upgrade
```

5.3 Code style

- Always import `unicode_literals` and use unicode literals for everything except where you're explicitly working with bytes, which are marked with the `b` prefix.

  Do this:

```
from __future__ import unicode_literals

foo = 'I am a unicode string, which is a sane default'
bar = b'I am a bytestring'
```

  Not this:

```
foo = u'I am a unicode string'
bar = 'I am a bytestring, but was it intentional?'
```

- Follow **PEP 8** unless otherwise noted. flake8 should be used to check your code against the guidelines.
- Use four spaces for indentation, *never* tabs.
- Use CamelCase with initial caps for class names:

```
ClassNameWithCamelCase
```

- Use underscore to split variable, function and method names for readability. Don't use CamelCase.

```
lower_case_with_underscores
```

- Use the fact that empty strings, lists and tuples are `False` and don't compare boolean values using == and `!=`.
- Follow whitespace rules as described in **PEP 8**. Good examples:

```
spam(ham[1], {eggs: 2})
spam(1)
dict['key'] = list[index]
```

- Limit lines to 80 characters and avoid trailing whitespace. However note that wrapped lines should be *one* indentation level in from level above, except for `if`, `for`, `with`, and `while` lines which should have two levels of indentation:

```
if (foo and bar ...
        baz and foobar):
    a = 1

from foobar import (foo, bar, ...
    baz)
```

- For consistency, prefer `'` over `"` for strings, unless the string contains `'`.
- Take a look at **PEP 20** for a nice peek into a general mindset useful for Python coding.

5.4 Extension development

Mopidy started as simply an MPD server that could play music from Spotify. Early on, Mopidy got multiple "frontends" to expose Mopidy to more than just MPD clients: for example the scrobbler frontend that scrobbles your listening history to your Last.fm account, the MPRIS frontend that integrates Mopidy into the Ubuntu Sound Menu, and the HTTP server and JavaScript player API making web based Mopidy clients possible. In Mopidy 0.9 we added support for multiple music sources without stopping and reconfiguring Mopidy: for example the local backend for playing music from your disk, the stream backend for playing Internet radio streams, and the Spotify and SoundCloud backends, for playing music directly from those services.

All of these are examples of what you can accomplish by creating a Mopidy extension. If you want to create your own Mopidy extension for something that does not exist yet, this guide to extension development will help you get your extension running in no time, and make it feel the way users would expect your extension to behave.

5.4.1 Anatomy of an extension

Extensions are located in a Python package called `mopidy_something` where "something" is the name of the application, library or web service you want to integrate with Mopidy. So, for example, if you plan to add support for a service named Soundspot to Mopidy, you would name your extension's Python package `mopidy_soundspot`.

The extension must be shipped with a `setup.py` file and be registered on PyPI. The name of the distribution on PyPI would be something like "Mopidy-Soundspot". Make sure to include the name "Mopidy" somewhere in that name and that you check the capitalization. This is the name users will use when they install your extension from PyPI.

Mopidy extensions must be licensed under an Apache 2.0 (like Mopidy itself), BSD, MIT or more liberal license to be able to be enlisted in the Mopidy documentation. The license text should be included in the `LICENSE` file in the root of the extension's Git repo.

Combining this together, we get the following folder structure for our extension, Mopidy-Soundspot:

```
mopidy-soundspot/            # The Git repo root
    LICENSE                  # The license text
    MANIFEST.in              # List of data files to include in PyPI package
    README.rst               # Document what it is and how to use it
    mopidy_soundspot/        # Your code
        __init__.py
        ext.conf             # Default config for the extension
        ...
    setup.py                 # Installation script
```

Example content for the most important files follows below.

5.4.2 cookiecutter project template

We've also made a cookiecutter project template for creating new Mopidy extensions. If you install cookiecutter and run a single command, you're asked a few questions about the name of your extension, etc. This is used to create a folder structure similar to the above, with all the needed files and most of the details filled in for you. This saves you a lot of tedious work and copy-pasting from this howto. See the readme of cookiecutter-mopidy-ext for further details.

5.4.3 Example README.rst

The README file should quickly explain what the extension does, how to install it, and how to configure it. It should also contain a link to a tarball of the latest development version of the extension. It's important that this link ends with `#egg=Mopidy-Something-dev` for installation using `pip install Mopidy-Something==dev` to work.

```
****************
Mopidy-Soundspot
****************

`Mopidy <http://www.mopidy.com/>`_ extension for playing music from
`Soundspot <http://soundspot.example.com/>`_.

Requires a Soundspot Platina subscription and the pysoundspot library.


Installation
============

Install by running::

    sudo pip install Mopidy-Soundspot

Or, if available, install the Debian/Ubuntu package from `apt.mopidy.com
<http://apt.mopidy.com/>`_.


Configuration
=============

Before starting Mopidy, you must add your Soundspot username and password
to the Mopidy configuration file::

    [soundspot]
    username = alice
```

(continues on next page)

(continued from previous page)

```
    password = secret


Project resources
=================

- `Source code <https://github.com/mopidy/mopidy-soundspot>`_
- `Issue tracker <https://github.com/mopidy/mopidy-soundspot/issues>`_
- `Development branch tarball <https://github.com/mopidy/mopidy-soundspot/tarball/
↪master#egg=Mopidy-Soundspot-dev>`_


Changelog
=========

v0.1.0 (2013-09-17)
-------------------

- Initial release.
```

5.4.4 Example setup.py

The `setup.py` file must use setuptools, and not distutils. This is because Mopidy extensions use setuptools' entry point functionality to register themselves as available Mopidy extensions when they are installed on your system.

The example below also includes a couple of convenient tricks for reading the package version from the source code so that it is defined in a single place, and to reuse the README file as the long description of the package for the PyPI registration.

The package must have `install_requires` on `setuptools` and `Mopidy >= 0.14` (or a newer version, if your extension requires it), in addition to any other dependencies required by your extension. If you implement a Mopidy frontend or backend, you'll need to include `Pykka >= 1.1` in the requirements. The `entry_points` part must be included. The `mopidy.ext` part cannot be changed, but the innermost string should be changed. It's format is `ext_name = package_name:Extension`. `ext_name` should be a short name for your extension, typically the part after "Mopidy-" in lowercase. This name is used e.g. to name the config section for your extension. The `package_name:Extension` part is simply the Python path to the extension class that will connect the rest of the dots.

```
from __future__ import unicode_literals

import re
from setuptools import setup, find_packages


def get_version(filename):
    content = open(filename).read()
    metadata = dict(re.findall("__([a-z]+)__ = '([^']+)'", content))
    return metadata['version']


setup(
    name='Mopidy-Soundspot',
    version=get_version('mopidy_soundspot/__init__.py'),
    url='https://github.com/your-account/mopidy-soundspot',
    license='Apache License, Version 2.0',
```

(continues on next page)

(continued from previous page)

```
    author='Your Name',
    author_email='your-email@example.com',
    description='Very short description',
    long_description=open('README.rst').read(),
    packages=find_packages(exclude=['tests', 'tests.*']),
    zip_safe=False,
    include_package_data=True,
    install_requires=[
        'setuptools',
        'Mopidy >= 0.14',
        'Pykka >= 1.1',
        'pysoundspot',
    ],
    test_suite='nose.collector',
    tests_require=[
        'nose',
        'mock >= 1.0',
    ],
    entry_points={
        'mopidy.ext': [
            'soundspot = mopidy_soundspot:Extension',
        ],
    },
    classifiers=[
        'Environment :: No Input/Output (Daemon)',
        'Intended Audience :: End Users/Desktop',
        'License :: OSI Approved :: Apache Software License',
        'Operating System :: OS Independent',
        'Programming Language :: Python :: 2',
        'Topic :: Multimedia :: Sound/Audio :: Players',
    ],
)
```

To make sure your README, license file and default config file is included in the package that is uploaded to PyPI, we'll also need to add a `MANIFEST.in` file:

```
include LICENSE
include MANIFEST.in
include README.rst
include mopidy_soundspot/ext.conf
```

For details on the `MANIFEST.in` file format, check out the distutils docs. check-manifest is a very useful tool to check your `MANIFEST.in` file for completeness.

5.4.5 Example __init__.py

The `__init__.py` file should be placed inside the `mopidy_soundspot` Python package.

The root of your Python package should have an `__version__` attribute with a **PEP 386** compliant version number, for example "0.1". Next, it should have a class named `Extension` which inherits from Mopidy's extension base class, *`mopidy.ext.Extension`*. This is the class referred to in the `entry_points` part of `setup.py`. Any imports of other files in your extension, outside of Mopidy and it's core requirements, should be kept inside methods. This ensures that this file can be imported without raising `ImportError` exceptions for missing dependencies, etc.

The default configuration for the extension is defined by the `get_default_config()` method in the `Extension` class which returns a `ConfigParser` compatible config section. The config section's name must be the same as

the extension's short name, as defined in the `entry_points` part of `setup.py`, for example `soundspot`. All extensions must include an `enabled` config which normally should default to `true`. Provide good defaults for all config values so that as few users as possible will need to change them. The exception is if the config value has security implications; in that case you should default to the most secure configuration. Leave any configurations that don't have meaningful defaults blank, like `username` and `password`. In the example below, we've chosen to maintain the default config as a separate file named `ext.conf`. This makes it easy to include the default config in documentation without duplicating it.

This is `mopidy_soundspot/__init__.py`:

```
from __future__ import unicode_literals

import logging
import os

import pygst
pygst.require('0.10')
import gst
import gobject

from mopidy import config, exceptions, ext


__version__ = '0.1'

# If you need to log, use loggers named after the current Python module
logger = logging.getLogger(__name__)


class Extension(ext.Extension):

    dist_name = 'Mopidy-Soundspot'
    ext_name = 'soundspot'
    version = __version__

    def get_default_config(self):
        conf_file = os.path.join(os.path.dirname(__file__), 'ext.conf')
        return config.read(conf_file)

    def get_config_schema(self):
        schema = super(Extension, self).get_config_schema()
        schema['username'] = config.String()
        schema['password'] = config.Secret()
        return schema

    def get_command(self):
        from .commands import SoundspotCommand
        return SoundspotCommand()

    def validate_environment(self):
        # Any manual checks of the environment to fail early.
        # Dependencies described by setup.py are checked by Mopidy, so you
        # should not check their presence here.
        pass

    def setup(self, registry):
        # You will typically only do one of the following things in a
        # single extension.
```

(continues on next page)

(continued from previous page)

```
        # Register a frontend
        from .frontend import SoundspotFrontend
        registry.add('frontend', SoundspotFrontend)

        # Register a backend
        from .backend import SoundspotBackend
        registry.add('backend', SoundspotBackend)

        # Register a custom GStreamer element
        from .mixer import SoundspotMixer
        gobject.type_register(SoundspotMixer)
        gst.element_register(
            SoundspotMixer, 'soundspotmixer', gst.RANK_MARGINAL)

        # Or nothing to register e.g. command extension
        pass
```

And this is `mopidy_soundspot/ext.conf`:

```
[soundspot]
enabled = true
username =
password =
```

For more detailed documentation on the extension class, see the *Extension API*.

5.4.6 Example frontend

If you want to *use* Mopidy's core API from your extension, then you want to implement a frontend.

The skeleton of a frontend would look like this. Notice that the frontend gets passed a reference to the core API when it's created. See the *Frontend API* for more details.

```
import pykka

from mopidy import core


class SoundspotFrontend(pykka.ThreadingActor, core.CoreListener):
    def __init__(self, config, core):
        super(SoundspotFrontend, self).__init__()
        self.core = core

    # Your frontend implementation
```

5.4.7 Example backend

If you want to extend Mopidy to support new music and playlist sources, you want to implement a backend. A backend does not have access to Mopidy's core API at all, but it does have a bunch of interfaces it can implement to extend Mopidy.

The skeleton of a backend would look like this. See *Backend API* for more details.

```
import pykka

from mopidy import backend


class SoundspotBackend(pykka.ThreadingActor, backend.Backend):
    def __init__(self, config, audio):
        super(SoundspotBackend, self).__init__()
        self.audio = audio

    # Your backend implementation
```

5.4.8 Example command

If you want to extend the Mopidy with a new helper not run from the server, such as scanning for media, adding a command is the way to go. Your top level command name will always match your extension name, but you are free to add sub-commands with names of your choosing.

The skeleton of a command would look like this. See *Commands API* for more details.

```
from mopidy import commands


class SoundspotCommand(commands.Command):
    help = 'Some text that will show up in --help'

    def __init__(self):
        super(SoundspotCommand, self).__init__()
        self.add_argument('--foo')

    def run(self, args, config, extensions):
       # Your command implementation
       return 0
```

5.4.9 Example web application

As of Mopidy 0.19, extensions can use Mopidy's built-in web server to host static web clients as well as Tornado and WSGI web applications. For several examples, see the *HTTP server side API* docs or explore with *Mopidy-API-Explorer* extension.

5.4.10 Example GStreamer element

If you want to extend Mopidy's GStreamer pipeline with new custom GStreamer elements, you'll need to register them in GStreamer before they can be used.

Basically, you just implement your GStreamer element in Python and then make your `setup()` method register all your custom GStreamer elements.

5.4.11 Running an extension

Once your extension is ready to go, to see it in action you'll need to register it with Mopidy. Typically this is done by running `python setup.py install` from your extension's Git repo root directory. While developing your ex-

tension and to avoid doing this every time you make a change, you can instead run `python setup.py develop` to effectively link Mopidy directly with your development files.

5.4.12 Python conventions

In general, it would be nice if Mopidy extensions followed the same *Code style* as Mopidy itself, as they're part of the same ecosystem. Among other things, the code style guide explains why all the above examples start with `from __future__ import unicode_literals`.

5.4.13 Use of Mopidy APIs

When writing an extension, you should only use APIs documented at *API reference*. Other parts of Mopidy, like `mopidy.utils`, may change at any time and are not something extensions should use.

5.4.14 Logging in extensions

For servers like Mopidy, logging is essential for understanding what's going on. We use the `logging` module from Python's standard library. When creating a logger, always namespace the logger using your Python package name as this will be visible in Mopidy's debug log:

```
import logging

logger = logging.getLogger('mopidy_soundspot')

# Or even better, use the Python module name as the logger name:
logger = logging.getLogger(__name__)
```

When logging at logging level `info` or higher (i.e. `warning`, `error`, and `critical`, but not `debug`) the log message will be displayed to all Mopidy users. Thus, the log messages at those levels should be well written and easy to understand.

As the logger name is not included in Mopidy's default logging format, you should make it obvious from the log message who is the source of the log message. For example:

```
Loaded 17 Soundspot playlists
```

Is much better than:

```
Loaded 17 playlists
```

If you want to turn on debug logging for your own extension, but not for everything else due to the amount of noise, see the docs for the *loglevels/** config section.

CHAPTER 6

Reference

6.1 Glossary

backend A part of Mopidy providing music library, playlist storage and/or playback capability to the *core*. Mopidy have a backend for each music store or music service it supports. See *Backend API* for details.

core The part of Mopidy that makes multiple frontends capable of using multiple backends. The core module is also the owner of the *tracklist*. To use the core module, see *Core API*.

extension A Python package that can extend Mopidy with one or more *backends*, *frontends*, or GStreamer elements like *mixers*. See *Extensions* for a list of existing extensions and *Extension development* for how to make a new extension.

frontend A part of Mopidy *using* the *core* API. Existing frontends include the *MPD server*, the MPRIS/D-Bus integration, the Last.fm scrobbler, and the *HTTP server* with JavaScript API. See *Frontend API* for details.

mixer A GStreamer element that controls audio volume.

tracklist Mopidy's name for the play queue or current playlist. The name is inspired by the MPRIS specification.

6.2 mopidy command

6.2.1 Synopsis

mopidy [-h] [–version] [-q] [-v] [–save-debug-log] [–config CONFIG_FILES] [-o CONFIG_OVERRIDES] [COMMAND] ...

6.2.2 Description

Mopidy is a music server which can play music both from multiple sources, like your local hard drive, radio streams, and from Spotify and SoundCloud. Searches combines results from all music sources, and you can mix tracks from all sources in your play queue. Your playlists from Spotify or SoundCloud are also available for use.

The `mopidy` command is used to start the server.

6.2.3 Options

`--help, -h`
Show help message and exit.

`--version`
Show Mopidy's version number and exit.

`--quiet, -q`
Show less output: warning level and higher.

`--verbose, -v`
Show more output. Repeat up to 3 times for even more.

`--save-debug-log`
Save debug log to the file specified in the *`logging/debug_file`* config value, typically `./mopidy.log`.

`--config` `<file|directory>`
Specify config files and directories to use. To use multiple config files or directories, separate them with a colon. The later files override the earlier ones if there's a conflict. When specifying a directory, all files ending in .conf in the directory are used.

`--option` `<option>`**`, -o`** `<option>`
Specify additional config values in the `section/key=value` format. Can be provided multiple times.

6.2.4 Built in commands

`config`
Show the current effective config. All configuration sources are merged together to show the effective document. Secret values like passwords are masked out. Config for disabled extensions are not included.

`deps`
Show dependencies, their versions and installation location.

6.2.5 Extension commands

Additionally, extensions can provide extra commands. Run *mopidy –help* for a list of what is available on your system and command-specific help. Commands for disabled extensions will be listed, but can not be run.

`local clear`
Clear local media files from the local library.

`local scan`
Scan local media files present in your library.

6.2.6 Files

`/etc/mopidy/mopidy.conf` System wide Mopidy configuration file.

`~/.config/mopidy/mopidy.conf` Your personal Mopidy configuration file. Overrides any configs from the system wide configuration file.

6.2.7 Examples

To start the music server, run:

```
mopidy
```

To start the server with an additional config file than can override configs set in the default config files, run:

```
mopidy --config ./my-config.conf
```

To start the server and change a config value directly on the command line, run:

```
mopidy --option mpd/enabled=false
```

The *--option* flag may be repeated multiple times to change multiple configs:

```
mopidy -o mpd/enabled=false -o spotify/bitrate=320
```

The `mopidy config` output shows the effect of the *--option* flags:

```
mopidy -o mpd/enabled=false -o spotify/bitrate=320 config
```

6.2.8 Reporting bugs

Report bugs to Mopidy's issue tracker at <https://github.com/mopidy/mopidy/issues>

6.3 API reference

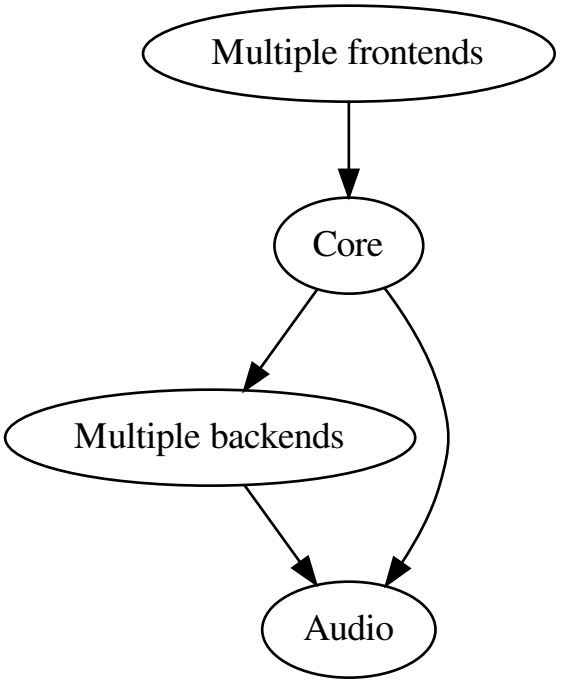
Warning: API stability

Only APIs documented here are public and open for use by Mopidy extensions. We will change these APIs, but will keep the changelog up to date with all breaking changes.

From Mopidy 1.0 and onwards, we intend to keep these APIs far more stable.

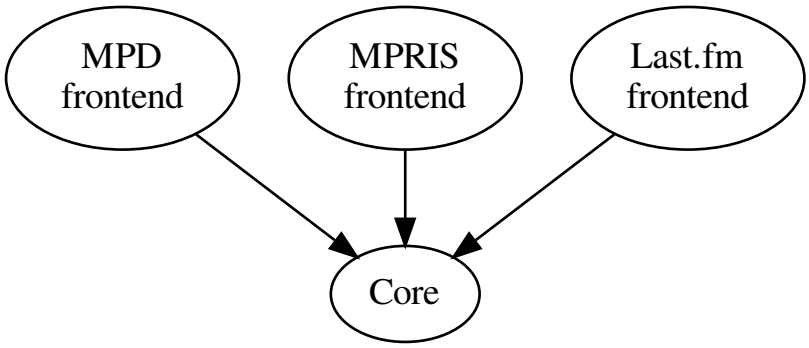
6.3.1 Architecture and concepts

The overall architecture of Mopidy is organized around multiple frontends and backends. The frontends use the core API. The core actor makes multiple backends work as one. The backends connect to various music sources. Both the core actor and the backends use the audio actor to play audio and control audio volume.



Frontends

Frontends expose Mopidy to the external world. They can implement servers for protocols like MPD and MPRIS, and they can be used to update other services when something happens in Mopidy, like the Last.fm scrobbler frontend does. See *Frontend API* for more details.



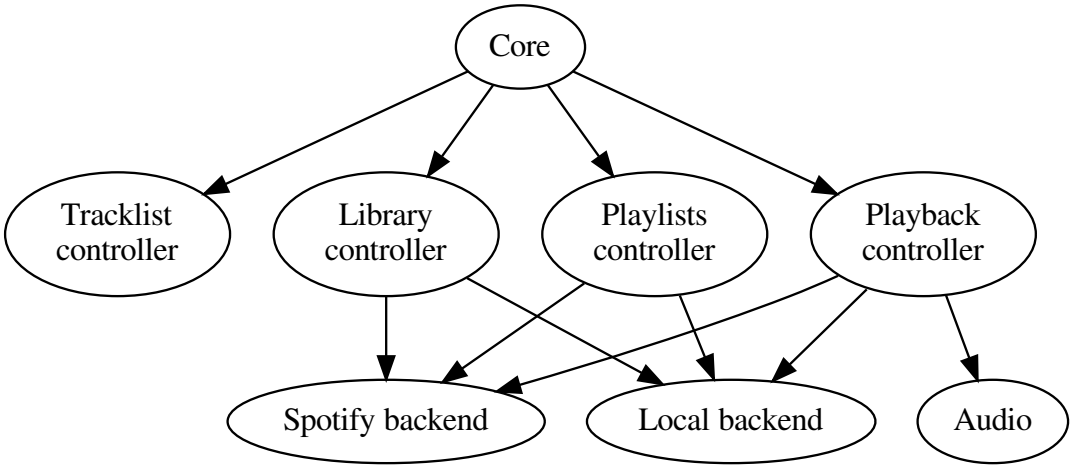
Core

The core is organized as a set of controllers with responsiblity for separate sets of functionality.

The core is the single actor that the frontends send their requests to. For every request from a frontend it calls out to one or more backends which does the real work, and when the backends respond, the core actor is responsible for combining the responses into a single response to the requesting frontend.

The core actor also keeps track of the tracklist, since it doesn't belong to a specific backend.

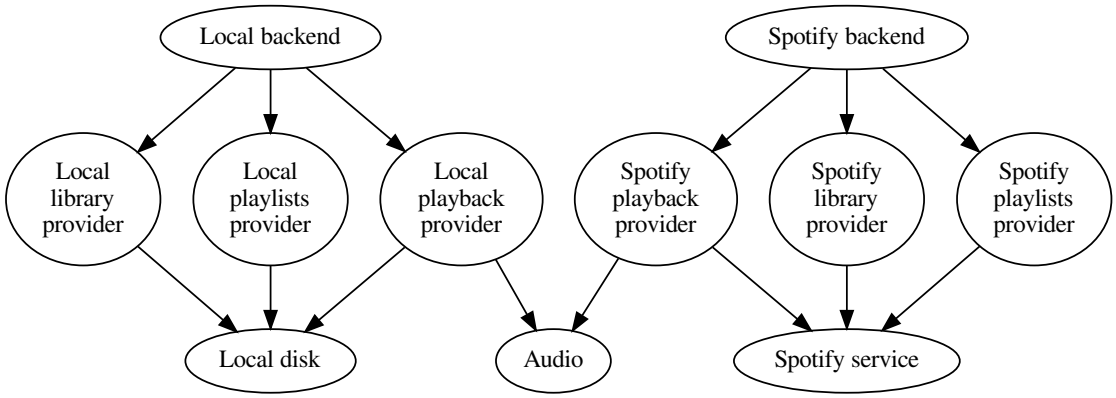
See *Core API* for more details.



Backends

The backends are organized as a set of providers with responsiblity for separate sets of functionality, similar to the core actor.

Anything specific to i.e. Spotify integration or local storage is contained in the backends. To integrate with new music sources, you just add a new backend. See *Backend API* for more details.



Audio

The audio actor is a thin wrapper around the parts of the GStreamer library we use. In addition to playback, it's responsible for volume control through both GStreamer's own volume mixers, and mixers we've created ourselves. If you implement an advanced backend, you may need to implement your own playback provider using the *Audio API*.

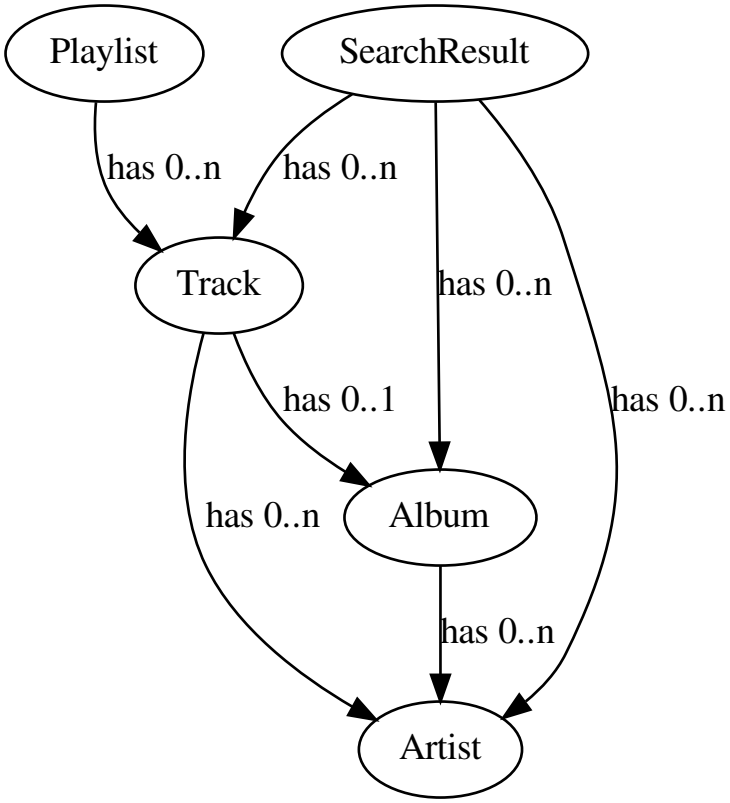
6.3.2 Data models

These immutable data models are used for all data transfer within the Mopidy backends and between the backends and the MPD frontend. All fields are optional and immutable. In other words, they can only be set through the class constructor during instance creation.

If you want to modify a model, use the *copy()* method. It accepts keyword arguments for the parts of the model you want to change, and copies the rest of the data from the model you call it on. Example:

```
>>> from mopidy.models import Track
>>> track1 = Track(name='Christmas Carol', length=171)
>>> track1
Track(artists=[], length=171, name='Christmas Carol')
>>> track2 = track1.copy(length=37)
>>> track2
Track(artists=[], length=37, name='Christmas Carol')
>>> track1
Track(artists=[], length=171, name='Christmas Carol')
```

Data model relations



Data model API

class mopidy.models.**Album**(**args*, ***kwargs*)

Parameters

- **uri** (*string*) – album URI
- **name** (*string*) – album name
- **artists** (list of *Artist*) – album artists
- **num_tracks** (integer or `None` if unknown) – number of tracks in album
- **num_discs** (integer or `None` if unknown) – number of discs in album
- **date** (*string*) – album release date (YYYY or YYYY-MM-DD)
- **musicbrainz_id** (*string*) – MusicBrainz ID
- **images** (*list of strings*) – album image URIs

artists = frozenset([])
A set of album artists. Read-only.

date = None
The album release date. Read-only.

images = frozenset([])
The album image URIs. Read-only.

musicbrainz_id = None
The MusicBrainz ID of the album. Read-only.

name = None
The album name. Read-only.

num_discs = None
The number of discs in the album. Read-only.

num_tracks = None
The number of tracks in the album. Read-only.

uri = None
The album URI. Read-only.

class mopidy.models.**Artist**(**args*, ***kwargs*)

Parameters

- **uri** (*string*) – artist URI
- **name** (*string*) – artist name
- **musicbrainz_id** (*string*) – MusicBrainz ID

musicbrainz_id = None
The MusicBrainz ID of the artist. Read-only.

name = None
The artist name. Read-only.

uri = None
The artist URI. Read-only.

class mopidy.models.**ImmutableObject**(**args*, ***kwargs*)
Superclass for immutable objects whose fields can only be modified via the constructor.

Parameters **kwargs** (*any*) – kwargs to set as fields on the object

copy(***values*)
Copy the model with `field` updated to new value.

Examples:

```
# Returns a track with a new name
Track(name='foo').copy(name='bar')
# Return an album with a new number of tracks
Album(num_tracks=2).copy(num_tracks=5)
```

Parameters **values** (*dict*) – the model fields to modify

Return type new instance of the model being copied

class mopidy.models.**ModelJSONEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)

Automatically serialize Mopidy models to JSON.

Usage:

```
>>> import json
>>> json.dumps({'a_track': Track(name='name')}, cls=ModelJSONEncoder)
'{"a_track": {"__model__": "Track", "name": "name"}}'
```

default(*obj*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

class mopidy.models.**Playlist**(**args*, ***kwargs*)

Parameters

- **uri** (*string*) – playlist URI
- **name** (*string*) – playlist name
- **tracks** (list of *Track* elements) – playlist's tracks
- **last_modified** (*int*) – playlist's modification time in milliseconds since Unix epoch

last_modified = None

The playlist modification time in milliseconds since Unix epoch. Read-only.

Integer, or `None` if unknown.

length

The number of tracks in the playlist. Read-only.

name = None

The playlist name. Read-only.

tracks = ()

The playlist's tracks. Read-only.

uri = None

The playlist URI. Read-only.

class mopidy.models.**Ref**(**args*, ***kwargs*)

Model to represent URI references with a human friendly name and type attached. This is intended for use a lightweight object "free" of metadata that can be passed around instead of using full blown models.

Parameters

- **uri** (*string*) – object URI
- **name** (*string*) – object name
- **type** – object type

ALBUM = u'album'
Constant used for comparison with the *type* field.

ARTIST = u'artist'
Constant used for comparison with the *type* field.

DIRECTORY = u'directory'
Constant used for comparison with the *type* field.

PLAYLIST = u'playlist'
Constant used for comparison with the *type* field.

TRACK = u'track'
Constant used for comparison with the *type* field.

classmethod album(***kwargs*)
Create a *Ref* with `type` *ALBUM*.

classmethod artist(***kwargs*)
Create a *Ref* with `type` *ARTIST*.

classmethod directory(***kwargs*)
Create a *Ref* with `type` *DIRECTORY*.

name = None
The object name. Read-only.

classmethod playlist(***kwargs*)
Create a *Ref* with `type` *PLAYLIST*.

classmethod track(***kwargs*)
Create a *Ref* with `type` *TRACK*.

type = None
The object type, e.g. "artist", "album", "track", "playlist", "directory". Read-only.

uri = None
The object URI. Read-only.

class mopidy.models.**SearchResult**(**args*, ***kwargs*)

Parameters

- **uri** (*string*) – search result URI
- **tracks** (list of *Track* elements) – matching tracks
- **artists** (list of *Artist* elements) – matching artists
- **albums** (list of *Album* elements) – matching albums

class mopidy.models.**TlTrack**(**args*, ***kwargs*)
A tracklist track. Wraps a regular track and it's tracklist ID.

The use of *TlTrack* allows the same track to appear multiple times in the tracklist.

This class also accepts it's parameters as positional arguments. Both arguments must be provided, and they must appear in the order they are listed here.

This class also supports iteration, so your extract its values like this:

```
(tlid, track) = tl_track
```

Parameters

- **tlid** (*int*) – tracklist ID
- **track** (*Track*) – the track

tlid = None
The tracklist ID. Read-only.

track = None
The track. Read-only.

class mopidy.models.**Track**(**args*, ***kwargs*)

Parameters

- **uri** (*string*) – track URI
- **name** (*string*) – track name
- **artists** (list of *Artist*) – track artists
- **album** (*Album*) – track album
- **composers** (*string*) – track composers
- **performers** (*string*) – track performers
- **genre** (*string*) – track genre
- **track_no** (integer or None if unknown) – track number in album
- **disc_no** (integer or None if unknown) – disc number in album
- **date** (*string*) – track release date (YYYY or YYYY-MM-DD)
- **length** (*integer*) – track length in milliseconds
- **bitrate** (*integer*) – bitrate in kbit/s
- **comment** (*string*) – track comment
- **musicbrainz_id** (*string*) – MusicBrainz ID
- **last_modified** (integer or None if unknown) – Represents last modification time

album = None
The track *Album*. Read-only.

artists = frozenset([])
A set of track artists. Read-only.

bitrate = None
The track's bitrate in kbit/s. Read-only.

comment = None
The track comment. Read-only.

composers = frozenset([])
A set of track composers. Read-only.

date = None
The track release date. Read-only.

disc_no = None
The disc number in the album. Read-only.

genre = None
The track genre. Read-only.

last_modified = None
Integer representing when the track was last modified, exact meaning depends on source of track. For local files this is the mtime, for other backends it could be a timestamp or simply a version counter.

length = None
The track length in milliseconds. Read-only.

musicbrainz_id = None
The MusicBrainz ID of the track. Read-only.

name = None
The track name. Read-only.

performers = frozenset([])
A set of track performers‘. Read-only.

track_no = None
The track number in the album. Read-only.

uri = None
The track URI. Read-only.

mopidy.models.**model_json_decoder**(*dct*)
Automatically deserialize Mopidy models from JSON.

Usage:

```
>>> import json
>>> json.loads(
...     '{"a_track": {"__model__": "Track", "name": "name"}}',
...     object_hook=model_json_decoder)
{u'a_track': Track(artists=[], name=u'name')}
```

6.3.3 Backend API

The backend API is the interface that must be implemented when you create a backend. If you are working on a frontend and need to access the backends, see the *Core API* instead.

URIs and routing of requests to the backend

When Mopidy's core layer is processing a client request, it routes the request to one or more appropriate backends based on the URIs of the objects the request touches on. The objects' URIs are compared with the backends' *uri_schemes* to select the relevant backends.

An often used pattern when implementing Mopidy backends is to create your own URI scheme which you use for all tracks, playlists, etc. related to your backend. In most cases the Mopidy URI is translated to an actual URI that GStreamer knows how to play right before playback. For example:

- Spotify already has its own URI scheme (`spotify:track:...`, `spotify:playlist:...`, etc.) used throughout their applications, and thus Mopidy-Spotify simply uses the same URI scheme. Playback is handled by pushing raw audio data into a GStreamer `appsrc` element.

- Mopidy-SoundCloud created it's own URI scheme, after the model of Spotify, and use URIs of the following forms: `soundcloud:search`, `soundcloud:user-...`, `soundcloud:exp-...`, and `soundcloud:set-...`. Playback is handled by converting the custom `soundcloud:..` URIs to `http://` URIs immediately before they are passed on to GStreamer for playback.
- Mopidy differentiates between `file://...` URIs handled by *Mopidy-Stream* and `local:...` URIs handled by *Mopidy-Local*. *Mopidy-Stream* can play `file://...` URIs pointing to tracks and playlists located anywhere on your system, but it doesn't know a thing about the object before you play it. On the other hand, *Mopidy-Local* scans a predefined *`local/media_dir`* to build a meta data library of all known tracks. It is thus limited to playing tracks residing in the media library, but can provide additional features like directory browsing and search. In other words, we have two different ways of playing local music, handled by two different backends, and have thus created two different URI schemes to separate their handling. The `local:...` URIs are converted to `file://...` URIs immediately before they are passed on to GStreamer for playback.

If there isn't an existing URI scheme that fits for your backend's purpose, you should create your own, and name it after your extension's *`ext_name`*. Care should be taken not to conflict with already in use URI schemes. It is also recommended to design the format such that tracks, playlists and other entities can be distinguished easily.

Backend class

class mopidy.backend.**Backend**

Backend API

If the backend has problems during initialization it should raise `mopidy.exceptions.BackendError` with a descriptive error message. This will make Mopidy print the error message and exit so that the user can fix the issue.

Parameters

- **config** (*dict*) – the entire Mopidy configuration
- **audio** (`pykka.ActorProxy` for *mopidy.audio.Audio*) – actor proxy for the audio subsystem

audio = None

Actor proxy to an instance of *mopidy.audio.Audio*.

Should be passed to the backend constructor as the kwarg `audio`, which will then set this field.

library = None

The library provider. An instance of *LibraryProvider*, or `None` if the backend doesn't provide a library.

playback = None

The playback provider. An instance of *PlaybackProvider*, or `None` if the backend doesn't provide playback.

playlists = None

The playlists provider. An instance of *PlaylistsProvider*, or class:*None* if the backend doesn't provide playlists.

uri_schemes = []

List of URI schemes this backend can handle.

Playback provider

class mopidy.backend.**PlaybackProvider**(*audio*, *backend*)

Parameters

- **audio** (actor proxy to an instance of *mopidy.audio.Audio*) – the audio actor
- **backend** (*mopidy.backend.Backend*) – the backend

change_track(*track*)
Swith to provided track.

MAY be reimplemented by subclass.

Parameters **track** (*mopidy.models.Track*) – the track to play

Return type `True` if successful, else `False`

get_time_position()
Get the current time position in milliseconds.

MAY be reimplemented by subclass.

Return type int

pause()
Pause playback.

MAY be reimplemented by subclass.

Return type `True` if successful, else `False`

play(*track*)
Play given track.

MAY be reimplemented by subclass.

Parameters **track** (*mopidy.models.Track*) – the track to play

Return type `True` if successful, else `False`

resume()
Resume playback at the same time position playback was paused.

MAY be reimplemented by subclass.

Return type `True` if successful, else `False`

seek(*time_position*)
Seek to a given time position.

MAY be reimplemented by subclass.

Parameters **time_position** (*int*) – time position in milliseconds

Return type `True` if successful, else `False`

stop()
Stop playback.

MAY be reimplemented by subclass.

Return type `True` if successful, else `False`

Playlists provider

class mopidy.backend.**PlaylistsProvider**(*backend*)

Parameters **backend** (*mopidy.backend.Backend* instance) – backend the controller is a part of

create(*name*)
See *mopidy.core.PlaylistsController.create()*.

MUST be implemented by subclass.

delete(*uri*)
See *mopidy.core.PlaylistsController.delete()*.

MUST be implemented by subclass.

lookup(*uri*)
See *mopidy.core.PlaylistsController.lookup()*.

MUST be implemented by subclass.

playlists
Currently available playlists.

Read/write. List of *mopidy.models.Playlist*.

refresh()
See *mopidy.core.PlaylistsController.refresh()*.

MUST be implemented by subclass.

save(*playlist*)
See *mopidy.core.PlaylistsController.save()*.

MUST be implemented by subclass.

Library provider

class mopidy.backend.**LibraryProvider**(*backend*)

Parameters **backend** (*mopidy.backend.Backend*) – backend the controller is a part of

browse(*uri*)
See *mopidy.core.LibraryController.browse()*.

If you implement this method, make sure to also set *root_directory*.

MAY be implemented by subclass.

find_exact(*query=None*, *uris=None*)
See *mopidy.core.LibraryController.find_exact()*.

MAY be implemented by subclass.

lookup(*uri*)
See *mopidy.core.LibraryController.lookup()*.

MUST be implemented by subclass.

refresh(*uri=None*)
See *mopidy.core.LibraryController.refresh()*.

MAY be implemented by subclass.

root_directory = None
`models.Ref.directory` instance with a URI and name set representing the root of this library's browse tree. URIs must use one of the schemes supported by the backend, and name should be set to a human friendly value.

MUST be set by any class that implements :meth:'LibraryProvider.browse'.

search(*query=None*, *uris=None*)
See *mopidy.core.LibraryController.search()*.

MAY be implemented by subclass.

Backend listener

class mopidy.backend.**BackendListener**
Marker interface for recipients of events sent by the backend actors.

Any Pykka actor that mixes in this class will receive calls to the methods defined here when the corresponding events happen in the core actor. This interface is used both for looking up what actors to notify of the events, and for providing default implementations for those listeners that are not interested in all events.

Normally, only the Core actor should mix in this class.

playlists_loaded()
Called when playlists are loaded or refreshed.

MAY be implemented by actor.

static send(*event*, ***kwargs*)
Helper to allow calling of backend listener events

Backend implementations

See *Backend extensions*.

6.3.4 Core API

The core API is the interface that is used by frontends like *mopidy.http* and *mopidy.mpd*. The core layer is inbetween the frontends and the backends.

class mopidy.core.**Core**(*mixer=None*, *backends=None*)

library = None
The library controller. An instance of *mopidy.core.LibraryController*.

playback = None
The playback controller. An instance of *mopidy.core.PlaybackController*.

playlists = None
The playlists controller. An instance of *mopidy.core.PlaylistsController*.

tracklist = None
The tracklist controller. An instance of *mopidy.core.TracklistController*.

uri_schemes
List of URI schemes we can handle

version
Version of the Mopidy core API

Playback controller

Manages playback, with actions like play, pause, stop, next, previous, seek, and volume control.

class mopidy.core.**PlaybackState**
Enum of playback states.

PAUSED = u'paused'
Constant representing the paused state.

PLAYING = u'playing'
Constant representing the playing state.

STOPPED = u'stopped'
Constant representing the stopped state.

class mopidy.core.**PlaybackController**(*mixer*, *backends*, *core*)

change_track(*tl_track*, *on_error_step=1*)
Change to the given track, keeping the current playback state.

Parameters

- **tl_track** (*mopidy.models.TlTrack* or None) – track to change to
- **on_error_step** (*int, -1 or 1*) – direction to step at play error, 1 for next track (default), -1 for previous track

current_tl_track = None
The currently playing or selected *mopidy.models.TlTrack*, or None.

current_track
The currently playing or selected *mopidy.models.Track*.

Read-only. Extracted from *current_tl_track* for convenience.

mute
Mute state as a True if muted, False otherwise

next()
Change to the next track.

The current playback state will be kept. If it was playing, playing will continue. If it was paused, it will still be paused, etc.

on_end_of_track()
Tell the playback controller that end of track is reached.

Used by event handler in *mopidy.core.Core*.

on_tracklist_change()
Tell the playback controller that the current playlist has changed.

Used by *mopidy.core.TracklistController*.

pause()
Pause playback.

play(*tl_track=None*, *on_error_step=1*)
Play the given track, or if the given track is None, play the currently active track.

Parameters

- **tl_track** (*mopidy.models.TlTrack* or None) – track to play

- **on_error_step** (*int, -1 or 1*) – direction to step at play error, 1 for next track (default), -1 for previous track

previous()
Change to the previous track.

The current playback state will be kept. If it was playing, playing will continue. If it was paused, it will still be paused, etc.

resume()
If paused, resume playing the current track.

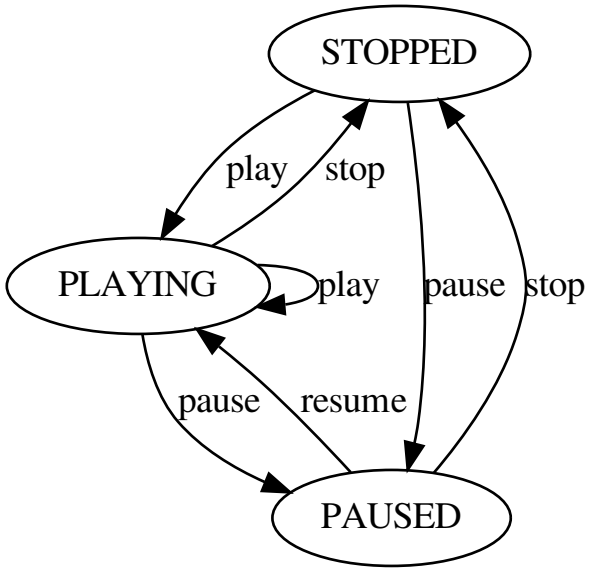
seek(*time_position*)
Seeks to time position given in milliseconds.

Parameters **time_position** (*int*) – time position in milliseconds

Return type `True` if successful, else `False`

state
The playback state. Must be `PLAYING`, `PAUSED`, or `STOPPED`.

Possible states and transitions:



stop(*clear_current_track=False*)
Stop playing.

Parameters **clear_current_track** (*boolean*) – whether to clear the current track _after_ stopping

time_position
Time position in milliseconds.

volume
Volume as int in range [0..100] or `None` if unknown. The volume scale is linear.

Tracklist controller

Manages everything related to the tracks we are currently playing.

class mopidy.core.**TracklistController**(*core*)

add(*tracks=None*, *at_position=None*, *uri=None*)
Add the track or list of tracks to the tracklist.

If `uri` is given instead of `tracks`, the URI is looked up in the library and the resulting tracks are added to the tracklist.

If `at_position` is given, the tracks placed at the given position in the tracklist. If `at_position` is not given, the tracks are appended to the end of the tracklist.

Triggers the *mopidy.core.CoreListener.tracklist_changed()* event.

Parameters

- **tracks** (list of *mopidy.models.Track*) – tracks to add
- **at_position** (int or `None`) – position in tracklist to add track
- **uri** (*string*) – URI for tracks to add

Return type list of *mopidy.models.TlTrack*

clear()
Clear the tracklist.

Triggers the *mopidy.core.CoreListener.tracklist_changed()* event.

consume

True Tracks are removed from the tracklist when they have been played.

False Tracks are not removed from the tracklist.

eot_track(*tl_track*)
The track that will be played after the given track.

Not necessarily the same track as *next_track()*.

Parameters **tl_track** (*mopidy.models.TlTrack* or `None`) – the reference track

Return type *mopidy.models.TlTrack* or `None`

filter(*criteria=None*, ***kwargs*)
Filter the tracklist by the given criterias.

A criteria consists of a model field to check and a list of values to compare it against. If the model field matches one of the values, it may be returned.

Only tracks that matches all the given criterias are returned.

Examples:

```
# Returns tracks with TLIDs 1, 2, 3, or 4 (tracklist ID)
filter({'tlid': [1, 2, 3, 4]})
filter(tlid=[1, 2, 3, 4])

# Returns track with IDs 1, 5, or 7
filter({'id': [1, 5, 7]})
filter(id=[1, 5, 7])
```

(continues on next page)

(continued from previous page)

```
# Returns track with URIs 'xyz' or 'abc'
filter({'uri': ['xyz', 'abc']})
filter(uri=['xyz', 'abc'])

# Returns tracks with ID 1 and URI 'xyz'
filter({'id': [1], 'uri': ['xyz']})
filter(id=[1], uri=['xyz'])

# Returns track with a matching ID (1, 3 or 6) and a matching URI
# ('xyz' or 'abc')
filter({'id': [1, 3, 6], 'uri': ['xyz', 'abc']})
filter(id=[1, 3, 6], uri=['xyz', 'abc'])
```

Parameters **criteria** (*dict, of (string, list) pairs*) – on or more criteria to match by

Return type list of *mopidy.models.TlTrack*

index(*tl_track*)

The position of the given track in the tracklist.

Parameters **tl_track** (*mopidy.models.TlTrack*) – the track to find the index of

Return type `int` or `None`

length

Length of the tracklist.

mark_played(*tl_track*)

Private method used by *mopidy.core.PlaybackController*.

mark_playing(*tl_track*)

Private method used by *mopidy.core.PlaybackController*.

mark_unplayable(*tl_track*)

Private method used by *mopidy.core.PlaybackController*.

move(*start*, *end*, *to_position*)

Move the tracks in the slice `[start:end]` to `to_position`.

Triggers the *mopidy.core.CoreListener.tracklist_changed()* event.

Parameters

- **start** (*int*) – position of first track to move
- **end** (*int*) – position after last track to move
- **to_position** (*int*) – new position for the tracks

next_track(*tl_track*)

The track that will be played if calling *mopidy.core.PlaybackController.next()*.

For normal playback this is the next track in the tracklist. If repeat is enabled the next track can loop around the tracklist. When random is enabled this should be a random track, all tracks should be played once before the tracklist repeats.

Parameters **tl_track** (*mopidy.models.TlTrack* or `None`) – the reference track

Return type *mopidy.models.TlTrack* or `None`

previous_track(*tl_track*)
Returns the track that will be played if calling *mopidy.core.PlaybackController.previous()*.

For normal playback this is the previous track in the tracklist. If random and/or consume is enabled it should return the current track instead.

Parameters **tl_track** (*mopidy.models.TlTrack* or `None`) – the reference track

Return type *mopidy.models.TlTrack* or `None`

random

True Tracks are selected at random from the tracklist.

False Tracks are played in the order of the tracklist.

remove(*criteria=None*, ***kwargs*)
Remove the matching tracks from the tracklist.

Uses *filter()* to lookup the tracks to remove.

Triggers the *mopidy.core.CoreListener.tracklist_changed()* event.

Parameters **criteria** (*dict*) – on or more criteria to match by

Return type list of *mopidy.models.TlTrack* that was removed

repeat

True The tracklist is played repeatedly. To repeat a single track, select both *repeat* and *single*.

False The tracklist is played once.

shuffle(*start=None*, *end=None*)
Shuffles the entire tracklist. If `start` and `end` is given only shuffles the slice `[start:end]`.

Triggers the *mopidy.core.CoreListener.tracklist_changed()* event.

Parameters

- **start** (int or `None`) – position of first track to shuffle
- **end** (int or `None`) – position after last track to shuffle

single

True Playback is stopped after current song, unless in *repeat* mode.

False Playback continues after current song.

slice(*start*, *end*)
Returns a slice of the tracklist, limited by the given start and end positions.

Parameters

- **start** (*int*) – position of first track to include in slice
- **end** (*int*) – position after last track to include in slice

Return type *mopidy.models.TlTrack*

tl_tracks
List of *mopidy.models.TlTrack*.

Read-only.

tracks
List of *mopidy.models.Track* in the tracklist.

Read-only.

version
The tracklist version.

Read-only. Integer which is increased every time the tracklist is changed. Is not reset before Mopidy is restarted.

Playlists controller

Manages persistence of playlists.

class mopidy.core.**PlaylistsController**(*backends*, *core*)

create(*name*, *uri_scheme=None*)
Create a new playlist.

If `uri_scheme` matches an URI scheme handled by a current backend, that backend is asked to create the playlist. If `uri_scheme` is `None` or doesn't match a current backend, the first backend is asked to create the playlist.

All new playlists should be created by calling this method, and **not** by creating new instances of *mopidy.models.Playlist*.

Parameters

- **name** (*string*) – name of the new playlist
- **uri_scheme** (*string*) – use the backend matching the URI scheme

Return type *mopidy.models.Playlist*

delete(*uri*)
Delete playlist identified by the URI.

If the URI doesn't match the URI schemes handled by the current backends, nothing happens.

Parameters **uri** (*string*) – URI of the playlist to delete

filter(*criteria=None*, ***kwargs*)
Filter playlists by the given criterias.

Examples:

```
# Returns track with name 'a'
filter({'name': 'a'})
filter(name='a')

# Returns track with URI 'xyz'
filter({'uri': 'xyz'})
filter(uri='xyz')

# Returns track with name 'a' and URI 'xyz'
filter({'name': 'a', 'uri': 'xyz'})
filter(name='a', uri='xyz')
```

Parameters **criteria** (*dict*) – one or more criteria to match by

Return type list of *mopidy.models.Playlist*

lookup(*uri*)
Lookup playlist with given URI in both the set of playlists and in any other playlist sources. Returns `None` if not found.

Parameters **uri** (*string*) – playlist URI

Return type *mopidy.models.Playlist* or `None`

playlists
The available playlists.

Read-only. List of *mopidy.models.Playlist*.

refresh(*uri_scheme=None*)
Refresh the playlists in *playlists*.

If `uri_scheme` is `None`, all backends are asked to refresh. If `uri_scheme` is an URI scheme handled by a backend, only that backend is asked to refresh. If `uri_scheme` doesn't match any current backend, nothing happens.

Parameters **uri_scheme** (*string*) – limit to the backend matching the URI scheme

save(*playlist*)
Save the playlist.

For a playlist to be saveable, it must have the `uri` attribute set. You should not set the `uri` atribute yourself, but use playlist objects returned by *create()* or retrieved from *playlists*, which will always give you saveable playlists.

The method returns the saved playlist. The return playlist may differ from the saved playlist. E.g. if the playlist name was changed, the returned playlist may have a different URI. The caller of this method should throw away the playlist sent to this method, and use the returned playlist instead.

If the playlist's URI isn't set or doesn't match the URI scheme of a current backend, nothing is done and `None` is returned.

Parameters **playlist** (*mopidy.models.Playlist*) – the playlist

Return type *mopidy.models.Playlist* or `None`

Library controller

Manages the music library, e.g. searching for tracks to be added to a playlist.

class mopidy.core.**LibraryController**(*backends*, *core*)

browse(*uri*)
Browse directories and tracks at the given `uri`.

`uri` is a string which represents some directory belonging to a backend. To get the intial root directories for backends pass None as the URI.

Returns a list of *mopidy.models.Ref* objects for the directories and tracks at the given `uri`.

The *Ref* objects representing tracks keep the track's original URI. A matching pair of objects can look like this:

```
Track(uri='dummy:/foo.mp3', name='foo', artists=..., album=...)
Ref.track(uri='dummy:/foo.mp3', name='foo')
```

The *Ref* objects representing directories have backend specific URIs. These are opaque values, so no one but the backend that created them should try and derive any meaning from them. The only valid exception to this is checking the scheme, as it is used to route browse requests to the correct backend.

For example, the dummy library's `/bar` directory could be returned like this:

```
Ref.directory(uri='dummy:directory:/bar', name='bar')
```

Parameters **uri** (*string*) – URI to browse

Return type list of *mopidy.models.Ref*

find_exact(*query=None*, *uris=None*, ***kwargs*)

Search the library for tracks where `field` is `values`.

If the query is empty, and the backend can support it, all available tracks are returned.

If `uris` is given, the search is limited to results from within the URI roots. For example passing `uris=['file:']` will limit the search to the local backend.

Examples:

```
# Returns results matching 'a' from any backend
find_exact({'any': ['a']})
find_exact(any=['a'])

# Returns results matching artist 'xyz' from any backend
find_exact({'artist': ['xyz']})
find_exact(artist=['xyz'])

# Returns results matching 'a' and 'b' and artist 'xyz' from any
# backend
find_exact({'any': ['a', 'b'], 'artist': ['xyz']})
find_exact(any=['a', 'b'], artist=['xyz'])

# Returns results matching 'a' if within the given URI roots
# "file:///media/music" and "spotify:"
find_exact(
    {'any': ['a']}, uris=['file:///media/music', 'spotify:'])
find_exact(any=['a'], uris=['file:///media/music', 'spotify:'])
```

Parameters

- **query** (*dict*) – one or more queries to search for
- **uris** (list of strings or `None`) – zero or more URI roots to limit the search to

Return type list of *mopidy.models.SearchResult*

lookup(*uri*)

Lookup the given URI.

If the URI expands to multiple tracks, the returned list will contain them all.

Parameters **uri** (*string*) – track URI

Return type list of *mopidy.models.Track*

refresh(*uri=None*)

Refresh library. Limit to URI and below if an URI is given.

Parameters **uri** (*string*) – directory or track URI

search(*query=None*, *uris=None*, ***kwargs*)

Search the library for tracks where `field` contains `values`.

If the query is empty, and the backend can support it, all available tracks are returned.

If `uris` is given, the search is limited to results from within the URI roots. For example passing `uris=['file:']` will limit the search to the local backend.

Examples:

```
# Returns results matching 'a' in any backend
search({'any': ['a']})
search(any=['a'])

# Returns results matching artist 'xyz' in any backend
search({'artist': ['xyz']})
search(artist=['xyz'])

# Returns results matching 'a' and 'b' and artist 'xyz' in any
# backend
search({'any': ['a', 'b'], 'artist': ['xyz']})
search(any=['a', 'b'], artist=['xyz'])

# Returns results matching 'a' if within the given URI roots
# "file:///media/music" and "spotify:"
search({'any': ['a']}, uris=['file:///media/music', 'spotify:'])
search(any=['a'], uris=['file:///media/music', 'spotify:'])
```

Parameters

- **query** (*dict*) – one or more queries to search for
- **uris** (list of strings or `None`) – zero or more URI roots to limit the search to

Return type list of *mopidy.models.SearchResult*

Core listener

class mopidy.core.**CoreListener**

Marker interface for recipients of events sent by the core actor.

Any Pykka actor that mixes in this class will receive calls to the methods defined here when the corresponding events happen in the core actor. This interface is used both for looking up what actors to notify of the events, and for providing default implementations for those listeners that are not interested in all events.

mute_changed(*mute*)

Called whenever the mute state is changed.

MAY be implemented by actor.

Parameters **mute** (*boolean*) – the new mute state

on_event(*event*, ***kwargs*)

Called on all events.

MAY be implemented by actor. By default, this method forwards the event to the specific event methods.

Parameters

- **event** (*string*) – the event name

- **kwargs** – any other arguments to the specific event handlers

options_changed()
Called whenever an option is changed.

MAY be implemented by actor.

playback_state_changed(*old_state*, *new_state*)
Called whenever playback state is changed.

MAY be implemented by actor.

Parameters

- **old_state** (string from *mopidy.core.PlaybackState* field) – the state before the change
- **new_state** (string from *mopidy.core.PlaybackState* field) – the state after the change

playlist_changed(*playlist*)
Called whenever a playlist is changed.

MAY be implemented by actor.

Parameters **playlist** (*mopidy.models.Playlist*) – the changed playlist

playlists_loaded()
Called when playlists are loaded or refreshed.

MAY be implemented by actor.

seeked(*time_position*)
Called whenever the time position changes by an unexpected amount, e.g. at seek to a new time position.

MAY be implemented by actor.

Parameters **time_position** (*int*) – the position that was seeked to in milliseconds

static send(*event*, ***kwargs*)
Helper to allow calling of core listener events

track_playback_ended(*tl_track*, *time_position*)
Called whenever playback of a track ends.

MAY be implemented by actor.

Parameters

- **tl_track** (*mopidy.models.TlTrack*) – the track that was played before playback stopped
- **time_position** (*int*) – the time position in milliseconds

track_playback_paused(*tl_track*, *time_position*)
Called whenever track playback is paused.

MAY be implemented by actor.

Parameters

- **tl_track** (*mopidy.models.TlTrack*) – the track that was playing when playback paused
- **time_position** (*int*) – the time position in milliseconds

track_playback_resumed(*tl_track*, *time_position*)
Called whenever track playback is resumed.

MAY be implemented by actor.

Parameters

- **tl_track** (*mopidy.models.TlTrack*) – the track that was playing when playback resumed
- **time_position** (*int*) – the time position in milliseconds

track_playback_started(*tl_track*)
Called whenever a new track starts playing.

MAY be implemented by actor.

Parameters **tl_track** (*mopidy.models.TlTrack*) – the track that just started playing

tracklist_changed()
Called whenever the tracklist is changed.

MAY be implemented by actor.

volume_changed(*volume*)
Called whenever the volume is changed.

MAY be implemented by actor.

Parameters **volume** (*int*) – the new volume in the range [0..100]

6.3.5 Audio API

The audio API is the interface we have built around GStreamer to support our specific use cases. Most backends should be able to get by with simply setting the URI of the resource they want to play, for these cases the default playback provider should be used.

For more advanced cases such as when the raw audio data is delivered outside of GStreamer or the backend needs to add metadata to the currently playing resource, developers should sub-class the base playback provider and implement the extra behaviour that is needed through the following API:

class mopidy.audio.**Audio**(*config*, *mixer*)
Audio output through GStreamer.

emit_data(*buffer_*)
Call this to deliver raw audio data to be played.

Note that the uri must be set to `appsrc://` for this to work.

Returns true if data was delivered.

Parameters **buffer** (`gst.Buffer`) – buffer to pass to appsrc

Return type boolean

emit_end_of_stream()
Put an end-of-stream token on the playbin. This is typically used in combination with *emit_data()*.

We will get a GStreamer message when the stream playback reaches the token, and can then do any end-of-stream related tasks.

get_mute()
Get mute status of the software mixer.

Return type `True` if muted, `False` if unmuted, `None` if no mixer is installed.

get_position()
Get position in milliseconds.

Return type int

get_volume()
Get volume level of the software mixer.

Example values:

0: Minimum volume.

100: Maximum volume.

Return type int in range [0..100]

pause_playback()
Notify GStreamer that it should pause playback.

Return type `True` if successfull, else `False`

prepare_change()
Notify GStreamer that we are about to change state of playback.

This function *MUST* be called before changing URIs or doing changes like updating data that is being pushed. The reason for this is that GStreamer will reset all its state when it changes to `gst.STATE_READY`.

set_appsrc(*caps*, *need_data=None*, *enough_data=None*, *seek_data=None*)
Switch to using appsrc for getting audio to be played.

You *MUST* call *prepare_change()* before calling this method.

Parameters

- **caps** (*string*) – GStreamer caps string describing the audio format to expect
- **need_data** (*callable which takes data length hint in ms*) – callback for when appsrc needs data
- **enough_data** (*callable*) – callback for when appsrc has enough data
- **seek_data** (*callable which takes time position in ms*) – callback for when data from a new position is needed to continue playback

set_metadata(*track*)
Set track metadata for currently playing song.

Only needs to be called by sources such as *appsrc* which do not already inject tags in playbin, e.g. when using *emit_data()* to deliver raw audio data to GStreamer.

Parameters **track** (*mopidy.models.Track*) – the current track

set_mute(*mute*)
Mute or unmute of the software mixer.

Parameters **mute** (*bool*) – Whether to mute the mixer or not.

Return type `True` if successful, else `False`

set_position(*position*)
Set position in milliseconds.

Parameters **position** (*int*) – the position in milliseconds

Return type `True` if successful, else `False`

set_uri(*uri*)
Set URI of audio to be played.

You *MUST* call *prepare_change()* before calling this method.

Parameters **uri** (*string*) – the URI to play

set_volume(*volume*)
Set volume level of the software mixer.

Parameters **volume** (*int*) – the volume in the range [0..100]

Return type `True` if successful, else `False`

start_playback()
Notify GStreamer that it should start playback.

Return type `True` if successfull, else `False`

state = u'stopped'
The GStreamer state mapped to `mopidy.audio.PlaybackState`

stop_playback()
Notify GStreamer that is should stop playback.

Return type `True` if successfull, else `False`

Audio listener

class mopidy.audio.**AudioListener**
Marker interface for recipients of events sent by the audio actor.

Any Pykka actor that mixes in this class will receive calls to the methods defined here when the corresponding events happen in the core actor. This interface is used both for looking up what actors to notify of the events, and for providing default implementations for those listeners that are not interested in all events.

reached_end_of_stream()
Called whenever the end of the audio stream is reached.

MAY be implemented by actor.

static send(*event*, ***kwargs*)
Helper to allow calling of audio listener events

state_changed(*old_state*, *new_state*, *target_state*)
Called after the playback state have changed.

Will be called for both immediate and async state changes in GStreamer.

Target state is used to when we should be in the target state, but temporarily need to switch to an other state. A typical example of this is buffering. When this happens an event with *old=PLAYING, new=PAUSED, target=PLAYING* will be emitted. Once we have caught up a *old=PAUSED, new=PLAYING, target=None* event will be be generated.

Regular state changes will not have target state set as they are final states which should be stable.

MAY be implemented by actor.

Parameters

- **old_state** (string from *mopidy.core.PlaybackState* field) – the state before the change

- **new_state** (string from *mopidy.core.PlaybackState* field) – the state after the change
- **target_state** (string from *mopidy.core.PlaybackState* field or `None` if this is a final state.) – the intended state

Audio scanner

class mopidy.audio.scan.**Scanner**(*timeout=1000*, *min_duration=100*)

Helper to get tags and other relevant info from URIs.

Parameters

- **timeout** – timeout for scanning a URI in ms
- **min_duration** – minimum duration of scanned URI in ms, -1 for all.

scan(*uri*)

Scan the given uri collecting relevant metadata.

Parameters **uri** – URI of the resource to scan.

Returns Dictionary of tags, duration, mtime and uri information.

6.3.6 Audio mixer API

class mopidy.mixer.**Mixer**(*config*)

Audio mixer API

If the mixer has problems during initialization it should raise `mopidy.exceptions.MixerError` with a descriptive error message. This will make Mopidy print the error message and exit so that the user can fix the issue.

Parameters **config** (*dict*) – the entire Mopidy configuration

get_mute()

Get mute state of the mixer.

MAY be implemented by subclass.

Return type `True` if muted, `False` if unmuted, `None` if unknown.

get_volume()

Get volume level of the mixer on a linear scale from 0 to 100.

Example values:

0: Minimum volume, usually silent.

100: Maximum volume.

None: Volume is unknown.

MAY be implemented by subclass.

Return type int in range [0..100] or `None`

name = None

Name of the mixer.

Used when configuring what mixer to use. Should match the *ext_name* of the extension providing the mixer.

set_mute(*mute*)
Mute or unmute the mixer.

MAY be implemented by subclass.

Parameters **mute** (*bool*) – `True` to mute, `False` to unmute

Return type `True` if success, `False` if failure

set_volume(*volume*)
Set volume level of the mixer.

MAY be implemented by subclass.

Parameters **volume** (*int*) – Volume in the range [0..100]

Return type `True` if success, `False` if failure

trigger_mute_changed(*mute*)
Send `mute_changed` event to all mixer listeners.

This method should be called by subclasses when the mute state is changed, either because of a call to *set_mute()* or because of any external entity changing the mute state.

trigger_volume_changed(*volume*)
Send `volume_changed` event to all mixer listeners.

This method should be called by subclasses when the volume is changed, either because of a call to *set_volume()* or because of any external entity changing the volume.

class mopidy.mixer.**MixerListener**
Marker interface for recipients of events sent by the mixer actor.

Any Pykka actor that mixes in this class will receive calls to the methods defined here when the corresponding events happen in the mixer actor. This interface is used both for looking up what actors to notify of the events, and for providing default implementations for those listeners that are not interested in all events.

mute_changed(*mute*)
Called after the mute state has changed.

MAY be implemented by actor.

Parameters **mute** (*bool*) – `True` if muted, `False` if not muted

static send(*event*, ***kwargs*)
Helper to allow calling of mixer listener events

volume_changed(*volume*)
Called after the volume has changed.

MAY be implemented by actor.

Parameters **volume** (*int in range [0..100]*) – the new volume

Mixer implementations

See *Mixer extensions*.

6.3.7 Frontend API

The following requirements applies to any frontend implementation:

- A frontend MAY do mostly whatever it wants to, including creating threads, opening TCP ports and exposing Mopidy for a group of clients.
- A frontend MUST implement at least one Pykka actor, called the "main actor" from here on.
- The main actor MUST accept two constructor arguments:
  - `config`, which is a dict structure with the entire Mopidy configuration.
  - `core`, which will be an `ActorProxy` for the core actor. This object gives access to the full *Core API*.
- It MAY use additional actors to implement whatever it does, and using actors in frontend implementations is encouraged.
- The frontend is enabled if the extension it is part of is enabled. See *Extension development* for more information.
- The main actor MUST be able to start and stop the frontend when the main actor is started and stopped.
- The frontend MAY require additional config values to be set for it to work.
- Such config values MUST be documented.
- The main actor MUST raise the `mopidy.exceptions.FrontendError` with a descriptive error message if the defined config values are not adequate for the frontend to work properly.
- Any actor which is part of the frontend MAY implement the *`mopidy.core.CoreListener`* interface to receive notification of the specified events.

Frontend implementations

See *Frontend extensions*.

6.3.8 Commands API

class mopidy.commands.**Command**

Command parser and runner for building trees of commands.

This class provides a wraper around `argparse.ArgumentParser` for handling this type of command line application in a better way than argprases own sub-parser handling.

add_argument(**args*, ***kwargs*)

Add an argument to the parser.

This method takes all the same arguments as the `argparse.ArgumentParser` version of this method.

add_child(*name*, *command*)

Add a child parser to consider using.

Parameters **name** (*string*) – name to use for the sub-command that is being added.

exit(*status_code=0*, *message=None*, *usage=None*)

Optionally print a message and exit.

format_help(*prog=None*)

Format help for current parser and children.

format_usage(*prog=None*)

Format usage for current parser.

parse(*args*, *prog=None*)

Parse command line arguments.

Will recursively parse commands until a final parser is found or an error occurs. In the case of errors we will print a message and exit. Otherwise, any overrides are applied and the current parser stored in the command attribute of the return value.

Parameters

- **args** (*list of strings*) – list of arguments to parse
- **prog** (*string*) – name to use for program

Return type argparse.Namespace

run(**args*, ***kwargs*)
Run the command.

Must be implemented by sub-classes that are not simply an intermediate in the command namespace.

set(***kwargs*)
Override a value in the finaly result of parsing.

6.3.9 Extension API

If you want to learn how to make Mopidy extensions, read *Extension development*.

class mopidy.ext.**Extension**
Base class for Mopidy extensions

dist_name = None
The extension's distribution name, as registered on PyPI

Example: `Mopidy-Soundspot`

ext_name = None
The extension's short name, as used in setup.py and as config section name

Example: `soundspot`

get_command()
Command to expose to command line users running `mopidy`.

Returns Instance of a *Command* class.

get_config_schema()
The extension's config validation schema

Returns ExtensionConfigSchema

get_default_config()
The extension's default config as a bytestring

Returns bytes or unicode

setup(*registry*)
Register the extension's components in the extension *Registry*.

For example, to register a backend:

```
def setup(self, registry):
    from .backend import SoundspotBackend
    registry.add('backend', SoundspotBackend)
```

See *Registry* for a list of registry keys with a special meaning. Mopidy will instantiate and start any classes registered under the `frontend` and `backend` registry keys.

This method can also be used for other setup tasks not involving the extension registry. For example, to register custom GStreamer elements:

```
def setup(self, registry):
    from .mixer import SoundspotMixer
    gobject.type_register(SoundspotMixer)
    gst.element_register(
        SoundspotMixer, 'soundspotmixer', gst.RANK_MARGINAL)
```

Parameters **registry** (*Registry*) – the extension registry

validate_environment()
Checks if the extension can run in the current environment.

Dependencies described by `setup.py` are checked by Mopidy, so you should not check their presence here.

If a problem is found, raise `ExtensionError` with a message explaining the issue.

Raises `ExtensionError`

Returns `None`

version = None
The extension's version

Should match the `__version__` attribute on the extension's main Python module and the version registered on PyPI.

class mopidy.ext.**Registry**
Registry of components provided by Mopidy extensions.

Passed to the *setup()* method of all extensions. The registry can be used like a dict of string keys and lists.

Some keys have a special meaning, including, but not limited to:

- `backend` is used for Mopidy backend classes.
- `frontend` is used for Mopidy frontend classes.
- `local:library` is used for Mopidy-Local libraries.

Extensions can use the registry for allow other to extend the extension itself. For example the `Mopidy-Local` use the `local:library` key to allow other extensions to register library providers for `Mopidy-Local` to use. Extensions should namespace custom keys with the extension's *ext_name*, e.g. `local:foo` or `http:bar`.

add(*name*, *cls*)
Add a component to the registry.

Multiple classes can be registered to the same name.

mopidy.ext.**load_extensions**()
Find all installed extensions.

Returns list of installed extensions

mopidy.ext.**validate_extension**(*extension*)
Verify extension's dependencies and environment.

Parameters **extensions** – an extension to check

Returns if extension should be run

6.3.10 Config API

class mopidy.config.**Proxy**(*data*)

mopidy.config.**read**(*config_file*)
Helper to load config defaults in same way across core and extensions

Config section schemas

class mopidy.config.schemas.**ConfigSchema**(*name*)
Logical group of config values that correspond to a config section.

Schemas are set up by assigning config keys with config values to instances. Once setup *deserialize()* can be called with a dict of values to process. For convienience we also support `format()` method that can used for converting the values to a dict that can be printed and *serialize()* for converting the values to a form suitable for persistence.

deserialize(*values*)
Validates the given `values` using the config schema.

Returns a tuple with cleaned values and errors.

serialize(*values*, *display=False*)
Converts the given `values` to a format suitable for persistence.

If `display` is `True` secret config values, like passwords, will be masked out.

Returns a dict of config keys and values.

class mopidy.config.schemas.**LogLevelConfigSchema**(*name*)
Special cased schema for handling a config section with loglevels.

Expects the config keys to be logger names and the values to be log levels as understood by the `LogLevel` config value. Does not sub-class *ConfigSchema*, but implements the same serialize/deserialize interface.

Config value types

class mopidy.config.types.**Boolean**(*optional=False*)
Boolean value.

Accepts `1`, `yes`, `true`, and `on` with any casing as `True`.

Accepts `0`, `no`, `false`, and `off` with any casing as `False`.

class mopidy.config.types.**ConfigValue**
Represents a config key's value and how to handle it.

Normally you will only be interacting with sub-classes for config values that encode either deserialization behavior and/or validation.

Each config value should be used for the following actions:

1. Deserializing from a raw string and validating, raising ValueError on failure.
2. Serializing a value back to a string that can be stored in a config.
3. Formatting a value to a printable form (useful for masking secrets).

`None` values should not be deserialized, serialized or formatted, the code interacting with the config should simply skip None config values.

deserialize(*value*)
Cast raw string to appropriate type.

serialize(*value*, *display=False*)
Convert value back to string for saving.

class mopidy.config.types.**Deprecated**
Deprecated value

Used for ignoring old config values that are no longer in use, but should not cause the config parser to crash.

class mopidy.config.types.**ExpandedPath**(*original*, *expanded*)

class mopidy.config.types.**Hostname**(*optional=False*)
Network hostname value.

class mopidy.config.types.**Integer**(*minimum=None*, *maximum=None*, *choices=None*, *optional=False*)
Integer value.

class mopidy.config.types.**List**(*optional=False*)
List value.

Supports elements split by commas or newlines. Newlines take presedence and empty list items will be filtered out.

class mopidy.config.types.**LogLevel**
Log level value.

Expects one of `critical`, `error`, `warning`, `info`, `debug` with any casing.

class mopidy.config.types.**Path**(*optional=False*)
File system path

The following expansions of the path will be done:

- ~ to the current user's home directory
- `$XDG_CACHE_DIR` according to the XDG spec
- `$XDG_CONFIG_DIR` according to the XDG spec
- `$XDG_DATA_DIR` according to the XDG spec
- `$XDG_MUSIC_DIR` according to the XDG spec

class mopidy.config.types.**Port**(*choices=None*, *optional=False*)
Network port value.

Expects integer in the range 0-65535, zero tells the kernel to simply allocate a port for us.

class mopidy.config.types.**Secret**(*optional=False*, *choices=None*)
Secret string value.

Is decoded as utf-8 and n t escapes should work and be preserved.

Should be used for passwords, auth tokens etc. Will mask value when being displayed.

class mopidy.config.types.**String**(*optional=False*, *choices=None*)
String value.

Is decoded as utf-8 and n t escapes should work and be preserved.

Config value validators

mopidy.config.validators.**validate_choice**(*value*, *choices*)
Validate that `value` is one of the `choices`

Normally called in *deserialize()*.

mopidy.config.validators.**validate_maximum**(*value*, *maximum*)
Validate that `value` is at most `maximum`

Normally called in *deserialize()*.

mopidy.config.validators.**validate_minimum**(*value*, *minimum*)
Validate that `value` is at least `minimum`

Normally called in *deserialize()*.

mopidy.config.validators.**validate_required**(*value*, *required*)
Validate that `value` is set if `required`

Normally called in *deserialize()* on the raw string, _not_ the converted value.

6.3.11 Zeroconf API

class mopidy.zeroconf.**Zeroconf**(*name*, *port*, *stype=None*, *domain=None*, *text=None*)
Publish a network service with Zeroconf.

Currently, this only works on Linux using Avahi via D-Bus.

Parameters

- **name** (*str*) – human readable name of the service, e.g. 'MPD on neptune'
- **port** (*int*) – TCP port of the service, e.g. 6600
- **stype** (*str*) – service type, e.g. '_mpd._tcp'
- **domain** (*str*) – local network domain name, defaults to ''
- **host** (*str*) – interface to advertise the service on, defaults to all interfaces
- **text** (*list of str*) – extra information depending on `stype`, defaults to empty list

publish()
Publish the service.

Call when your service starts.

unpublish()
Unpublish the service.

Call when your service shuts down.

6.3.12 HTTP server side API

The *Mopidy-HTTP* extension comes with an HTTP server to host Mopidy's *HTTP JSON-RPC API*. This web server can also be used by other extensions that need to expose something over HTTP.

The HTTP server side API can be used to:

- host static files for e.g. a Mopidy client written in pure JavaScript,
- host a Tornado application, or

- host a WSGI application, including e.g. Flask applications.

To host static files using the web server, an extension needs to register a name and a file path in the extension registry under the `http:static` key.

To extend the web server with a web application, an extension must register a name and a factory function in the extension registry under the `http:app` key.

For details on how to make a Mopidy extension, see the *Extension development* guide.

Static web client example

To serve static files, you just need to register an `http:static` dictionary in the extension registry. The dictionary must have two keys: `name` and `path`. The `name` is used to build the URL the static files will be served on. By convention, it should be identical with the extension's *`ext_name`*, like in the following example. The `path` tells Mopidy where on the disk the static files are located.

Assuming that the code below is located in the file `mywebclient/__init__.py`, the files in the directory `mywebclient/static/` will be made available at `/mywebclient/` on Mopidy's web server. For example, `mywebclient/static/foo.html` will be available at http://localhost:6680/mywebclient/foo.html.

```
from __future__ import unicode_literals

import os

from mopidy import ext


class MyWebClientExtension(ext.Extension):
    ext_name = 'mywebclient'

    def setup(self, registry):
        registry.add('http:static', {
            'name': self.ext_name,
            'path': os.path.join(os.path.dirname(__file__), 'static'),
        })

    # See the Extension API for the full details on this class
```

Tornado application example

The *Mopidy-HTTP* extension's web server is based on the Tornado web framework. Thus, it has first class support for Tornado request handlers.

In the following example, we create a `tornado.web.RequestHandler` called `MyRequestHandler` that responds to HTTP GET requests with the string `Hello, world! This is Mopidy $version`, where it gets the Mopidy version from Mopidy's core API.

To hook the request handler into Mopidy's web server, we must register a dictionary under the `http:app` key in the extension registry. The dictionary must have two keys: `name` and `factory`.

The `name` is used to build the URL the app will be served on. By convention, it should be identical with the extension's *`ext_name`*, like in the following example.

The `factory` must be a function that accepts two arguments, `config` and `core`, respectively a dict structure of Mopidy's config and a `pykka.ActorProxy` to the full Mopidy core API. The `factory` function must return a list of Tornado request handlers. The URL patterns of the request handlers should not include the `name`, as that will be prepended to the URL patterns by the web server.

When the extension is installed, Mopidy will respond to requests to http://localhost:6680/mywebclient/ with the string `Hello, world! This is Mopidy $version`.

```
from __future__ import unicode_literals

import os

import tornado.web

from mopidy import ext


class MyRequestHandler(tornado.web.RequestHandler):
    def initialize(self, core):
        self.core = core

    def get(self):
        self.write(
            'Hello, world! This is Mopidy %s' %
            self.core.get_version().get())


def my_app_factory(config, core):
    return [
        ('/', MyRequestHandler, {'core': core})
    ]


class MyWebClientExtension(ext.Extension):
    ext_name = 'mywebclient'

    def setup(self, registry):
        registry.add('http:app', {
            'name': self.ext_name,
            'factory': my_app_factory,
        })

    # See the Extension API for the full details on this class
```

WSGI application example

WSGI applications are second-class citizens on Mopidy's HTTP server. The WSGI applications are run inside Tornado, which is based on non-blocking I/O and a single event loop. In other words, your WSGI applications will only have a single thread to run on, and if your application is doing blocking I/O, it will block all other requests from being handled by the web server as well.

The example below shows how a WSGI application that returns the string `Hello, world! This is Mopidy $version` on all requests. The WSGI application is wrapped as a Tornado application and mounted at http://localhost:6680/mywebclient/.

```
from __future__ import unicode_literals

import os

import tornado.web
import tornado.wsgi
```

(continues on next page)

(continued from previous page)

```
from mopidy import ext


def my_app_factory(config, core):

    def wsgi_app(environ, start_response):
        status = '200 OK'
        response_headers = [('Content-type', 'text/plain')]
        start_response(status, response_headers)
        return [
            'Hello, world! This is Mopidy %s\n' %
            self.core.get_version().get()
        ]

    return [
        ('(.*)', tornado.web.FallbackHandler, {
            'fallback': tornado.wsgi.WSGIContainer(wsgi_app),
        }),
    ]


class MyWebClientExtension(ext.Extension):
    ext_name = 'mywebclient'

    def setup(self, registry):
        registry.add('http:app', {
            'name': self.ext_name,
            'factory': my_app_factory,
        })

    # See the Extension API for the full details on this class
```

API implementors

See *Web extensions*.

6.3.13 HTTP JSON-RPC API

The *Mopidy-HTTP* extension makes Mopidy's *Core API* available using JSON-RPC over HTTP using HTTP POST and WebSockets. We also provide a JavaScript wrapper, called *Mopidy.js*, around the JSON-RPC over WebSocket API for use both from browsers and Node.js. The *Mopidy-API-Explorer* extension, can also be used to get you familiarized with HTTP based APIs.

> **Warning:** API stability
>
> Since the HTTP JSON-RPC API exposes our internal core API directly it is to be regarded as **experimental**. We cannot promise to keep any form of backwards compatibility between releases as we will need to change the core API while working out how to support new use cases. Thus, if you use this API, you must expect to do small adjustments to your client for every release of Mopidy.
>
> From Mopidy 1.0 and onwards, we intend to keep the core API far more stable.

HTTP POST API

The Mopidy web server accepts HTTP requests with the POST method to http://localhost:6680/mopidy/rpc, where the `localhost:6680` part will vary with your local setup. The HTTP POST endpoint gives you access to Mopidy's full core API, but does not give you notification on events. If you need to listen to events, you should probably use the WebSocket API instead.

Example usage from the command line:

```
$ curl -d '{"jsonrpc": "2.0", "id": 1, "method": "core.playback.get_state"}' http://
↪localhost:6680/mopidy/rpc
{"jsonrpc": "2.0", "id": 1, "result": "stopped"}
```

For details on the request and response format, see *JSON-RPC 2.0 messages*.

WebSocket API

The Mopidy web server exposes a WebSocket at http://localhost:6680/mopidy/ws, where the `localhost:6680` part will vary with your local setup. The WebSocket gives you access to Mopidy's full API and enables Mopidy to instantly push events to the client, as they happen.

On the WebSocket we send two different kind of messages: The client can send *JSON-RPC 2.0 requests*, and the server will respond with JSON-RPC 2.0 responses. In addition, the server will send *event messages* when something happens on the server. Both message types are encoded as JSON objects.

If you're using the API from JavaScript, either in the browser or in Node.js, you should use *Mopidy.js JavaScript library* which wraps the WebSocket API in a nice JavaScript API.

JSON-RPC 2.0 messages

JSON-RPC 2.0 messages can be recognized by checking for the key named `jsonrpc` with the string value `2.0`. For details on the messaging format, please refer to the JSON-RPC 2.0 spec.

All methods (not attributes) in the *Core API* is made available through JSON-RPC calls over the WebSocket. For example, *`mopidy.core.PlaybackController.play()`* is available as the JSON-RPC method `core.playback.play`.

The core API's attributes is made available through setters and getters. For example, the attribute *`mopidy.core.PlaybackController.current_track`* is available as the JSON-RPC method `core.playback.get_current_track`.

Example JSON-RPC request:

```
{"jsonrpc": "2.0", "id": 1, "method": "core.playback.get_current_track"}
```

Example JSON-RPC response:

```
{"jsonrpc": "2.0", "id": 1, "result": {"__model__": "Track", "...": "..."}}
```

The JSON-RPC method `core.describe` returns a data structure describing all available methods. If you're unsure how the core API maps to JSON-RPC, having a look at the `core.describe` response can be helpful.

Event messages

Event objects will always have a key named `event` whose value is the event type. Depending on the event type, the event may include additional fields for related data. The events maps directly to the *`mopidy.`*

core.CoreListener API. Refer to the *CoreListener* method names is the available event types. The *CoreListener* method's keyword arguments are all included as extra fields on the event objects. Example event message:

```
{"event": "track_playback_started", "track": {...}}
```

6.3.14 Mopidy.js JavaScript library

We've made a JavaScript library, Mopidy.js, which wraps the *WebSocket API* and gets you quickly started with working on your client instead of figuring out how to communicate with Mopidy.

> **Warning:** API stability
>
> Since the Mopidy.js API exposes our internal core API directly it is to be regarded as **experimental**. We cannot promise to keep any form of backwards compatibility between releases as we will need to change the core API while working out how to support new use cases. Thus, if you use this API, you must expect to do small adjustments to your client for every release of Mopidy.
>
> From Mopidy 1.0 and onwards, we intend to keep the core API far more stable.

Getting the library for browser use

Regular and minified versions of Mopidy.js, ready for use, is installed together with Mopidy. When the HTTP extension is enabled, the files are available at:

- http://localhost:6680/mopidy/mopidy.js
- http://localhost:6680/mopidy/mopidy.min.js

You may need to adjust hostname and port for your local setup.

Thus, if you use Mopidy to host your web client, like described above, you can load the latest version of Mopidy.js by adding the following script tag to your HTML file:

```
<script type="text/javascript" src="/mopidy/mopidy.min.js"></script>
```

If you don't use Mopidy to host your web client, you can find the JS files in the Git repo at:

- `mopidy/http/data/mopidy.js`
- `mopidy/http/data/mopidy.min.js`

Getting the library for Node.js or Browserify use

If you want to use Mopidy.js from Node.js or on the web through Browserify, you can install Mopidy.js using npm:

```
npm install mopidy
```

After npm completes, you can import Mopidy.js using `require()`:

```
var Mopidy = require("mopidy");
```

Getting the library for development on the library

If you want to work on the Mopidy.js library itself, you'll find a complete development setup in the `js/` dir in our repo. The instructions in `js/README.md` will guide you on your way.

Creating an instance

Once you have Mopidy.js loaded, you need to create an instance of the wrapper:

```
var mopidy = new Mopidy();
```

When you instantiate `Mopidy()` without arguments, it will connect to the WebSocket at `/mopidy/ws/` on the current host. Thus, if you don't host your web client using Mopidy's web server, or if you use Mopidy.js from a Node.js environment, you'll need to pass the URL to the WebSocket end point:

```
var mopidy = new Mopidy({
    webSocketUrl: "ws://localhost:6680/mopidy/ws/"
});
```

It is also possible to create an instance first and connect to the WebSocket later:

```
var mopidy = new Mopidy({autoConnect: false});
// ... do other stuff, like hooking up events ...
mopidy.connect();
```

When creating an instance, you can specify the following settings:

autoConnect Whether or not to connect to the WebSocket on instance creation. Defaults to true.

backoffDelayMin The minimum number of milliseconds to wait after a connection error before we try to reconnect. For every failed attempt, the backoff delay is doubled until it reaches `backoffDelayMax`. Defaults to 1000.

backoffDelayMax The maximum number of milliseconds to wait after a connection error before we try to reconnect. Defaults to 64000.

callingConvention Which calling convention to use when calling methods.

If set to "by-position-only", methods expect to be called with positional arguments, like `mopidy.foo.bar(null, true, 2)`.

If set to "by-position-or-by-name", methods expect to be called either with an array of position arguments, like `mopidy.foo.bar([null, true, 2])`, or with an object of named arguments, like `mopidy.foo.bar({id: 2})`. The advantage of the "by-position-or-by-name" calling convention is that arguments with default values can be left out of the named argument object. Using named arguments also makes the code more readable, and more resistent to future API changes.

Note: For backwards compatibility, the default is "by-position-only". In the future, the default will change to "by-position-or-by-name". You should explicitly set this setting to your choice, so you won't be affected when the default changes.

New in version 0.19: (Mopidy.js 0.4)

console If set, this object will be used to log errors from Mopidy.js. This is mostly useful for testing Mopidy.js.

webSocket An existing WebSocket object to be used instead of creating a new WebSocket. Defaults to undefined.

webSocketUrl URL used when creating new WebSocket objects. Defaults to `ws://<document.location.host>/mopidy/ws`, or `ws://localhost/mopidy/ws` if `document.location.host` isn't available, like it is in the browser environment.

Hooking up to events

Once you have a Mopidy.js object, you can hook up to the events it emits. To explore your possibilities, it can be useful to subscribe to all events and log them:

```
mopidy.on(console.log.bind(console));
```

Several types of events are emitted:

- You can get notified about when the Mopidy.js object is connected to the server and ready for method calls, when it's offline, and when it's trying to reconnect to the server by looking at the events `state:online`, `state:offline`, `reconnectionPending`, and `reconnecting`.
- You can get events sent from the Mopidy server by looking at the events with the name prefix `event:`, like `event:trackPlaybackStarted`.
- You can introspect what happens internally on the WebSocket by looking at the events emitted with the name prefix `websocket:`.

Mopidy.js uses the event emitter library BANE, so you should refer to BANE's short API documentation to see how you can hook up your listeners to the different events.

Calling core API methods

Once your Mopidy.js object has connected to the Mopidy server and emits the `state:online` event, it is ready to accept core API method calls:

```
mopidy.on("state:online", function () {
    mopidy.playback.next();
});
```

Any calls you make before the `state:online` event is emitted will fail. If you've hooked up an errback (more on that a bit later) to the promise returned from the call, the errback will be called with a `Mopidy.ConnectionError` instance.

All methods in Mopidy's *Core API* is available via Mopidy.js. The core API attributes is *not* available, but that shouldn't be a problem as we've added (undocumented) getters and setters for all of them, so you can access the attributes as well from JavaScript. For example, the *mopidy.core.PlaybackController.state* attribute is available in JSON-RPC as the method `core.playback.get_state` and in Mopidy.js as `mopidy.playback.getState()`.

Both the WebSocket API and the JavaScript API are based on introspection of the core Python API. Thus, they will always be up to date and immediately reflect any changes we do to the core API.

The best way to explore the JavaScript API, is probably by opening your browser's console, and using its tab completion to navigate the API. You'll find the Mopidy core API exposed under `mopidy.playback`, `mopidy.tracklist`, `mopidy.playlists`, and `mopidy.library`.

All methods in the JavaScript API have an associated data structure describing the Python params it expects, and most methods also have the Python API documentation available. This is available right there in the browser console, by looking at the method's `description` and `params` attributes:

```
console.log(mopidy.playback.next.params);
console.log(mopidy.playback.next.description);
```

JSON-RPC 2.0 limits method parameters to be sent *either* by-position or by-name. Combinations of both, like we're used to from Python, isn't supported by JSON-RPC 2.0. To further limit this, Mopidy.js currently only supports passing parameters by-position.

Obviously, you'll want to get a return value from many of your method calls. Since everything is happening across the WebSocket and maybe even across the network, you'll get the results asynchronously. Instead of having to pass callbacks and errbacks to every method you call, the methods return "promise" objects, which you can use to pipe the future result as input to another method, or to hook up callback and errback functions.

```
var track = mopidy.playback.getCurrentTrack();
// => ``track`` isn't a track, but a "promise" object
```

Instead, typical usage will look like this:

```
var printCurrentTrack = function (track) {
    if (track) {
        console.log("Currently playing:", track.name, "by",
            track.artists[0].name, "from", track.album.name);
    } else {
        console.log("No current track");
    }
};

mopidy.playback.getCurrentTrack()
    .done(printCurrentTrack);
```

The function passed to `done()`, `printCurrentTrack`, is the callback that will be called if the method call succeeds. If anything goes wrong, `done()` will throw an exception.

If you want to explicitly handle any errors and avoid an exception being thrown, you can register an error handler function anywhere in a promise chain. The function will be called with the error object as the only argument:

```
mopidy.playback.getCurrentTrack()
    .catch(console.error.bind(console));
    .done(printCurrentTrack);
```

You can also register the error handler at the end of the promise chain by passing it as the second argument to `done()`:

```
mopidy.playback.getCurrentTrack()
    .done(printCurrentTrack, console.error.bind(console));
```

If you don't hook up an error handler function and never call `done()` on the promise object, when.js will log warnings to the console that you have unhandled errors. In general, unhandled errors will not go silently missing.

The promise objects returned by Mopidy.js adheres to the CommonJS Promises/A standard. We use the implementation known as when.js. Please refer to when.js' documentation or the standard for further details on how to work with promise objects.

Cleaning up

If you for some reason want to clean up after Mopidy.js before the web page is closed or navigated away from, you can close the WebSocket, unregister all event listeners, and delete the object like this:

```
// Close the WebSocket without reconnecting. Letting the object be garbage
// collected will have the same effect, so this isn't strictly necessary.
mopidy.close();

// Unregister all event listeners. If you don't do this, you may have
// lingering references to the object causing the garbage collector to not
// clean up after it.
mopidy.off();

// Delete your reference to the object, so it can be garbage collected.
mopidy = null;
```

Example to get started with

1. Make sure that you've installed all dependencies required by *Mopidy-HTTP*.
2. Create an empty directory for your web client.
3. Change the *http/static_dir* config value to point to your new directory.
4. Start/restart Mopidy.
5. Create a file in the directory named `index.html` containing e.g. "Hello, world!".
6. Visit http://localhost:6680/ to confirm that you can view your new HTML file there.
7. Include Mopidy.js in your web page:

   ```
   <script type="text/javascript" src="/mopidy/mopidy.min.js"></script>
   ```

8. Add one of the following Mopidy.js examples of how to queue and start playback of your first playlist either to your web page or a JavaScript file that you include in your web page.

   "Imperative" style:

   ```
   var trackDesc = function (track) {
       return track.name + " by " + track.artists[0].name +
           " from " + track.album.name;
   };

   var queueAndPlay = function (playlistNum, trackNum) {
       playlistNum = playlistNum || 0;
       trackNum = trackNum || 0;
       mopidy.playlists.getPlaylists().then(function (playlists) {
           var playlist = playlists[playlistNum];
           console.log("Loading playlist:", playlist.name);
           return mopidy.tracklist.add(playlist.tracks).then(function (tlTracks) {
               return mopidy.playback.play(tlTracks[trackNum]).then(function () {
                   return mopidy.playback.getCurrentTrack().then(function (track) {
                       console.log("Now playing:", trackDesc(track));
                   });
               });
           });
       })
       .catch(console.error.bind(console)) // Handle errors here
       .done();                            // ...or they'll be thrown here
   };
   ```

(continues on next page)

(continued from previous page)

```
var mopidy = new Mopidy();              // Connect to server
mopidy.on(console.log.bind(console));  // Log all events
mopidy.on("state:online", queueAndPlay);
```

Approximately the same behavior in a more functional style, using chaining of promises.

```
var get = function (key, object) {
    return object[key];
};

var printTypeAndName = function (model) {
    console.log(model.__model__ + ": " + model.name);
    // By returning the playlist, this function can be inserted
    // anywhere a model with a name is piped in the chain.
    return model;
};

var trackDesc = function (track) {
    return track.name + " by " + track.artists[0].name +
        " from " + track.album.name;
};

var printNowPlaying = function () {
    // By returning any arguments we get, the function can be inserted
    // anywhere in the chain.
    var args = arguments;
    return mopidy.playback.getCurrentTrack()
        .then(function (track) {
            console.log("Now playing:", trackDesc(track));
            return args;
        });
};

var queueAndPlay = function (playlistNum, trackNum) {
    playlistNum = playlistNum || 0;
    trackNum = trackNum || 0;
    mopidy.playlists.getPlaylists()
        // => list of Playlists
        .fold(get, playlistNum)
        // => Playlist
        .then(printTypeAndName)
        // => Playlist
        .fold(get, 'tracks')
        // => list of Tracks
        .then(mopidy.tracklist.add)
        // => list of TlTracks
        .fold(get, trackNum)
        // => TlTrack
        .then(mopidy.playback.play)
        // => null
        .then(printNowPlaying)
        // => null
        .catch(console.error.bind(console))  // Handle errors here
        // => null
        .done();                             // ...or they'll be thrown here
};
```

(continues on next page)

(continued from previous page)

```
var mopidy = new Mopidy();              // Connect to server
mopidy.on(console.log.bind(console));  // Log all events
mopidy.on("state:online", queueAndPlay);
```

9. The web page should now queue and play your first playlist every time you load it. See the browser's console for output from the function, any errors, and all events that are emitted.

6.4 Module reference

6.4.1 `mopidy.local` – Local backend

For details on how to use Mopidy's local backend, see *Mopidy-Local*.

class mopidy.local.**Library**(*config*)
Local library interface.

Extensions that wish to provide an alternate local library storage backend need to sub-class this class and install and configure it with an extension. Both scanning and library calls will use the active local library.

Parameters **config** – Config dictionary

ROOT_DIRECTORY_URI = u'local:directory'
URI of the local backend's root directory.

This constant should be used by libraries implementing the *Library.browse()* method.

add(*track*)
Add the given track to library.

Parameters **track** (*Track*) – Track to add to the library

begin()
Prepare library for accepting updates. Exactly what this means is highly implementation depended. This must however return an iterator that generates all tracks in the library for efficient scanning.

Return type *Track* iterator

browse(*uri*)
Browse directories and tracks at the given URI.

The URI for the root directory is a constant available at *Library.ROOT_DIRECTORY_URI*.

Parameters **path** (*string*) – URI to browse.

Return type List of *Ref* tracks and directories.

clear()
Clear out whatever data storage is used by this backend.

Return type Boolean indicating if state was cleared.

close()
Close any resources used for updating, commit outstanding work etc.

flush()
Called for every n-th track indicating that work should be committed. Sub-classes are free to ignore these hints.

Return type Boolean indicating if state was flushed.

load()
(Re)load any tracks stored in memory, if any, otherwise just return number of available tracks currently available. Will be called at startup for both library and update use cases, so if you plan to store tracks in memory this is when the should be (re)loaded.

Return type int representing number of tracks in library.

lookup(*uri*)
Lookup the given URI.

Unlike the core APIs, local tracks uris can only be resolved to a single track.

Parameters **uri** (*string*) – track URI

Return type *Track*

name = None
Name of the local library implementation, must be overriden.

remove(*uri*)
Remove the given track from the library.

Parameters **uri** (*str*) – URI to remove from the library/

search(*query=None*, *limit=100*, *offset=0*, *exact=False*, *uris=None*)
Search the library for tracks where `field` contains `values`.

Parameters

- **query** (*dict*) – one or more queries to search for
- **limit** (*int*) – maximum number of results to return
- **offset** (*int*) – offset into result set to use.
- **exact** (*bool*) – whether to look for exact matches
- **uris** (list of strings or `None`) – zero or more URI roots to limit the search to

Return type *SearchResult*

6.4.2 mopidy.mpd – MPD server

For details on how to use Mopidy's MPD server, see *Mopidy-MPD*.

MPD tokenizer

mopidy.mpd.tokenize.**split**(*line*)
Splits a line into tokens using same rules as MPD.

- Lines may not start with whitespace
- Tokens are split by arbitrary amount of spaces or tabs
- First token must match *[a-z][a-z0-9_]**
- Remaining tokens can be unquoted or quoted tokens.
- Unquoted tokens consist of all printable characters except double quotes, single quotes, spaces and tabs.
- Quoted tokens are surrounded by a matching pair of double quotes.
- The closing quote must be followed by space, tab or end of line.

- Any value is allowed inside a quoted token. Including double quotes, assuming it is correctly escaped.
- Backslash inside a quoted token is used to escape the following character.

For examples see the tests for this function.

MPD dispatcher

class mopidy.mpd.dispatcher.**MpdContext**(*dispatcher*, *session=None*, *config=None*, *core=None*)

This object is passed as the first argument to all MPD command handlers to give the command handlers access to important parts of Mopidy.

core = None
The Mopidy core API. An instance of *mopidy.core.Core*.

dispatcher = None
The current *MpdDispatcher*.

events = None
The active subsystems that have pending events.

lookup_playlist_from_name(*name*)
Helper function to retrieve a playlist from its unique MPD name.

lookup_playlist_name_from_uri(*uri*)
Helper function to retrieve the unique MPD playlist name from its uri.

password = None
The MPD password

refresh_playlists_mapping()
Maintain map between playlists and unique playlist names to be used by MPD

session = None
The current mopidy.mpd.MpdSession.

subscriptions = None
The subsytems that we want to be notified about in idle mode.

class mopidy.mpd.dispatcher.**MpdDispatcher**(*session=None*, *config=None*, *core=None*)

The MPD session feeds the MPD dispatcher with requests. The dispatcher finds the correct handler, processes the request and sends the response back to the MPD session.

handle_request(*request*, *current_command_list_index=None*)
Dispatch incoming requests to the correct handler.

MPD protocol

rather incomplete with regards to data formats, both for requests and responses. Thus, we have had to talk a great deal with the the original MPD server using telnet to get the details we need to implement our own MPD server which is compatible with the numerous existing MPD clients.

mopidy.mpd.protocol.**BOOL**(*value*)
Convert the values 0 and 1 into booleans.

class mopidy.mpd.protocol.**Commands**
Collection of MPD commands to expose to users.

Normally used through the global instance which command handlers have been installed into.

add(*name*, *auth_required=True*, *list_command=True*, ***validators*)
Create a decorator that registers a handler and validation rules.

Additional keyword arguments are treated as converters/validators to apply to tokens converting them to proper Python types.

Requirements for valid handlers:

- must accept a context argument as the first arg.
- may not use variable keyword arguments, `**kwargs`.
- may use variable arguments `*args` *or* a mix of required and optional arguments.

Decorator returns the unwrapped function so that tests etc can use the functions with values with correct python types instead of strings.

Parameters

- **name** (*string*) – Name of the command being registered.
- **auth_required** (*bool*) – If authorization is required.
- **list_command** (*bool*) – If command should be listed in reflection.

call(*tokens*, *context=None*)
Find and run the handler registered for the given command.

If the handler was registered with any converters/validators they will be run before calling the real handler.

Parameters

- **tokens** (*list*) – List of tokens to process
- **context** (*MpdContext*) – MPD context.

mopidy.mpd.protocol.**ENCODING = u'UTF-8'**
The MPD protocol uses UTF-8 for encoding all data.

mopidy.mpd.protocol.**INT**(*value*)
Converts a value that matches [+-]?d+ into and integer.

mopidy.mpd.protocol.**LINE_TERMINATOR = u'\n'**
The MPD protocol uses `\n` as line terminator.

mopidy.mpd.protocol.**RANGE**(*value*)
Convert a single integer or range spec into a slice

`n` should become `slice(n, n+1)` `n:` should become `slice(n, None)` `n:m` should become `slice(n, m)` and `m > n` must hold

mopidy.mpd.protocol.**UINT**(*value*)
Converts a value that matches d+ into an integer.

mopidy.mpd.protocol.**VERSION = u'0.17.0'**
The MPD protocol version is 0.17.0.

mopidy.mpd.protocol.**commands = <mopidy.mpd.protocol.Commands object>**
Global instance to install commands into

mopidy.mpd.protocol.**load_protocol_modules**()
The protocol modules must be imported to get them registered in *commands*.

Audio output

mopidy.mpd.protocol.audio_output.**disableoutput**(*context*, *outputid*)
musicpd.org, audio output section:

```
disableoutput
```

Turns an output off.

mopidy.mpd.protocol.audio_output.**enableoutput**(*context*, *outputid*)
musicpd.org, audio output section:

```
enableoutput
```

Turns an output on.

mopidy.mpd.protocol.audio_output.**outputs**(*context*)
musicpd.org, audio output section:

```
outputs
```

Shows information about all outputs.

mopidy.mpd.protocol.audio_output.**toggleoutput**(*context*, *outputid*)
musicpd.org, audio output section:

```
toggleoutput {ID}
```

Turns an output on or off, depending on the current state.

Channels

mopidy.mpd.protocol.channels.**channels**(*context*)
musicpd.org, client to client section:

```
channels
```

Obtain a list of all channels. The response is a list of "channel:" lines.

mopidy.mpd.protocol.channels.**readmessages**(*context*)
musicpd.org, client to client section:

```
readmessages
```

Reads messages for this client. The response is a list of "channel:" and "message:" lines.

mopidy.mpd.protocol.channels.**sendmessage**(*context*, *channel*, *text*)
musicpd.org, client to client section:

```
sendmessage {CHANNEL} {TEXT}
```

Send a message to the specified channel.

mopidy.mpd.protocol.channels.**subscribe**(*context*, *channel*)
musicpd.org, client to client section:

```
subscribe {NAME}
```

Subscribe to a channel. The channel is created if it does not exist already. The name may consist of alphanumeric ASCII characters plus underscore, dash, dot and colon.

mopidy.mpd.protocol.channels.**unsubscribe**(*context*, *channel*)
musicpd.org, client to client section:

`unsubscribe {NAME}`

Unsubscribe from a channel.

Command list

mopidy.mpd.protocol.command_list.**command_list_begin**(*context*)
musicpd.org, command list section:

To facilitate faster adding of files etc. you can pass a list of commands all at once using a command list. The command list begins with `command_list_begin` or `command_list_ok_begin` and ends with `command_list_end`.

It does not execute any commands until the list has ended. The return value is whatever the return for a list of commands is. On success for all commands, `OK` is returned. If a command fails, no more commands are executed and the appropriate `ACK` error is returned. If `command_list_ok_begin` is used, `list_OK` is returned for each successful command executed in the command list.

mopidy.mpd.protocol.command_list.**command_list_end**(*context*)
See *command_list_begin()*.

mopidy.mpd.protocol.command_list.**command_list_ok_begin**(*context*)
See *command_list_begin()*.

Connection

mopidy.mpd.protocol.connection.**close**(*context*)
musicpd.org, connection section:

`close`

Closes the connection to MPD.

mopidy.mpd.protocol.connection.**kill**(*context*)
musicpd.org, connection section:

`kill`

Kills MPD.

mopidy.mpd.protocol.connection.**password**(*context*, *password*)
musicpd.org, connection section:

`password {PASSWORD}`

This is used for authentication with the server. `PASSWORD` is simply the plaintext password.

mopidy.mpd.protocol.connection.**ping**(*context*)
musicpd.org, connection section:

`ping`

Does nothing but return `OK`.

Current playlist

mopidy.mpd.protocol.current_playlist.**add**(*context*, *uri*)
musicpd.org, current playlist section:

`add {URI}`

Adds the file `URI` to the playlist (directories add recursively). `URI` can also be a single file.

Clarifications:

- `add ""` should add all tracks in the library to the current playlist.

mopidy.mpd.protocol.current_playlist.**addid**(*context*, *uri*, *songpos=None*)
musicpd.org, current playlist section:

`addid {URI} [POSITION]`

Adds a song to the playlist (non-recursive) and returns the song id.

`URI` is always a single file or URL. For example:

```
addid "foo.mp3"
Id: 999
OK
```

Clarifications:

- `addid ""` should return an error.

mopidy.mpd.protocol.current_playlist.**addtagid**(*context*, *tlid*, *tag*, *value*)
musicpd.org, current playlist section:

`addtagid {SONGID} {TAG} {VALUE}`

Adds a tag to the specified song. Editing song tags is only possible for remote songs. This change is volatile: it may be overwritten by tags received from the server, and the data is gone when the song gets removed from the queue.

mopidy.mpd.protocol.current_playlist.**clear**(*context*)
musicpd.org, current playlist section:

`clear`

Clears the current playlist.

mopidy.mpd.protocol.current_playlist.**cleartagid**(*context*, *tlid*, *tag*)
musicpd.org, current playlist section:

`cleartagid {SONGID} [TAG]`

Removes tags from the specified song. If TAG is not specified, then all tag values will be removed. Editing song tags is only possible for remote songs.

mopidy.mpd.protocol.current_playlist.**delete**(*context*, *position*)
musicpd.org, current playlist section:

`delete [{POS} | {START:END}]`

Deletes a song from the playlist.

mopidy.mpd.protocol.current_playlist.**deleteid**(*context*, *tlid*)
musicpd.org, current playlist section:

`deleteid {SONGID}`

Deletes the song `SONGID` from the playlist

mopidy.mpd.protocol.current_playlist.**move_range**(*context*, *position*, *to*)
musicpd.org, current playlist section:

```
move [{FROM} | {START:END}] {TO}
```

Moves the song at `FROM` or range of songs at `START:END` to `TO` in the playlist.

mopidy.mpd.protocol.current_playlist.**moveid**(*context*, *tlid*, *to*)
musicpd.org, current playlist section:

```
moveid {FROM} {TO}
```

Moves the song with `FROM` (songid) to `TO` (playlist index) in the playlist. If `TO` is negative, it is relative to the current song in the playlist (if there is one).

mopidy.mpd.protocol.current_playlist.**playlist**(*context*)
musicpd.org, current playlist section:

```
playlist
```

Displays the current playlist.

Note: Do not use this, instead use `playlistinfo`.

---

mopidy.mpd.protocol.current_playlist.**playlistfind**(*context*, *tag*, *needle*)
musicpd.org, current playlist section:

```
playlistfind {TAG} {NEEDLE}
```

Finds songs in the current playlist with strict matching.

GMPC:

- does not add quotes around the tag.

mopidy.mpd.protocol.current_playlist.**playlistid**(*context*, *tlid=None*)
musicpd.org, current playlist section:

```
playlistid {SONGID}
```

Displays a list of songs in the playlist. `SONGID` is optional and specifies a single song to display info for.

mopidy.mpd.protocol.current_playlist.**playlistinfo**(*context*, *parameter=None*)
musicpd.org, current playlist section:

```
playlistinfo [[SONGPOS] | [START:END]]
```

Displays a list of all songs in the playlist, or if the optional argument is given, displays information only for the song `SONGPOS` or the range of songs `START:END`.

ncmpc and mpc:

- uses negative indexes, like `playlistinfo "-1"`, to request the entire playlist

mopidy.mpd.protocol.current_playlist.**playlistsearch**(*context*, *tag*, *needle*)
musicpd.org, current playlist section:

```
playlistsearch {TAG} {NEEDLE}
```

Searches case-sensitively for partial matches in the current playlist.

GMPC:

- does not add quotes around the tag
- uses `filename` and `any` as tags

mopidy.mpd.protocol.current_playlist.**plchanges**(*context*, *version*)
musicpd.org, current playlist section:

`plchanges {VERSION}`

Displays changed songs currently in the playlist since `VERSION`.

To detect songs that were deleted at the end of the playlist, use `playlistlength` returned by status command.

MPDroid:

- Calls `plchanges "-1"` two times per second to get the entire playlist.

mopidy.mpd.protocol.current_playlist.**plchangesposid**(*context*, *version*)
musicpd.org, current playlist section:

`plchangesposid {VERSION}`

Displays changed songs currently in the playlist since `VERSION`. This function only returns the position and the id of the changed song, not the complete metadata. This is more bandwidth efficient.

To detect songs that were deleted at the end of the playlist, use `playlistlength` returned by status command.

mopidy.mpd.protocol.current_playlist.**prio**(*context*, *priority*, *position*)
musicpd.org, current playlist section:

`prio {PRIORITY} {START:END...}`

Set the priority of the specified songs. A higher priority means that it will be played first when "random" mode is enabled.

A priority is an integer between 0 and 255. The default priority of new songs is 0.

mopidy.mpd.protocol.current_playlist.**prioid**(*context*, **args*)
musicpd.org, current playlist section:

`prioid {PRIORITY} {ID...}`

Same as prio, but address the songs with their id.

mopidy.mpd.protocol.current_playlist.**shuffle**(*context*, *position=None*)
musicpd.org, current playlist section:

`shuffle [START:END]`

Shuffles the current playlist. `START:END` is optional and specifies a range of songs.

mopidy.mpd.protocol.current_playlist.**swap**(*context*, *songpos1*, *songpos2*)
musicpd.org, current playlist section:

`swap {SONG1} {SONG2}`

Swaps the positions of `SONG1` and `SONG2`.

mopidy.mpd.protocol.current_playlist.**swapid**(*context*, *tlid1*, *tlid2*)
musicpd.org, current playlist section:

`swapid {SONG1} {SONG2}`

Swaps the positions of `SONG1` and `SONG2` (both song ids).

Music database

mopidy.mpd.protocol.music_db.**count**(*context*, **args*)
musicpd.org, music database section:

```
count {TAG} {NEEDLE}
```

Counts the number of songs and their total playtime in the db matching `TAG` exactly.

GMPC:

- does not add quotes around the tag argument.
- use multiple tag-needle pairs to make more specific searches.

mopidy.mpd.protocol.music_db.**find**(*context*, **args*)
musicpd.org, music database section:

```
find {TYPE} {WHAT}
```

Finds songs in the db that are exactly `WHAT`. `TYPE` can be any tag supported by MPD, or one of the two special parameters - `file` to search by full path (relative to database root), and `any` to match against all available tags. `WHAT` is what to find.

GMPC:

- does not add quotes around the field argument.
- also uses `find album "[ALBUM]" artist "[ARTIST]"` to list album tracks.

ncmpc:

- does not add quotes around the field argument.
- capitalizes the type argument.

ncmpcpp:

- also uses the search type "date".
- uses "file" instead of "filename".

mopidy.mpd.protocol.music_db.**findadd**(*context*, **args*)
musicpd.org, music database section:

```
findadd {TYPE} {WHAT}
```

Finds songs in the db that are exactly `WHAT` and adds them to current playlist. Parameters have the same meaning as for `find`.

mopidy.mpd.protocol.music_db.**list_**(*context*, **args*)
musicpd.org, music database section:

```
list {TYPE} [ARTIST]
```

Lists all tags of the specified type. `TYPE` should be `album`, `artist`, `albumartist`, `date`, or `genre`.

`ARTIST` is an optional parameter when type is `album`, `date`, or `genre`. This filters the result list by an artist.

Clarifications:

The musicpd.org documentation for `list` is far from complete. The command also supports the following variant:

```
list {TYPE} {QUERY}
```

Where `QUERY` applies to all `TYPE`. `QUERY` is one or more pairs of a field name and a value. If the `QUERY` consists of more than one pair, the pairs are AND-ed together to find the result. Examples of valid queries and what they should return:

`list "artist" "artist" "ABBA"` List artists where the artist name is "ABBA". Response:

```
Artist: ABBA
OK
```

`list "album" "artist" "ABBA"` Lists albums where the artist name is "ABBA". Response:

```
Album: More ABBA Gold: More ABBA Hits
Album: Absolute More Christmas
Album: Gold: Greatest Hits
OK
```

`list "artist" "album" "Gold: Greatest Hits"` Lists artists where the album name is "Gold: Greatest Hits". Response:

```
Artist: ABBA
OK
```

`list "artist" "artist" "ABBA" "artist" "TLC"` Lists artists where the artist name is "ABBA" *and* "TLC". Should never match anything. Response:

```
OK
```

`list "date" "artist" "ABBA"` Lists dates where artist name is "ABBA". Response:

```
Date:
Date: 1992
Date: 1993
OK
```

`list "date" "artist" "ABBA" "album" "Gold: Greatest Hits"` Lists dates where artist name is "ABBA" and album name is "Gold: Greatest Hits". Response:

```
Date: 1992
OK
```

`list "genre" "artist" "The Rolling Stones"` Lists genres where artist name is "The Rolling Stones". Response:

```
Genre:
Genre: Rock
OK
```

GMPC:

- does not add quotes around the field argument.

ncmpc:

- does not add quotes around the field argument.
- capitalizes the field argument.

`mopidy.mpd.protocol.music_db.`**`listall`**`(`*context*, *uri=None*`)`

musicpd.org, music database section:

`listall [URI]`

Lists all songs and directories in `URI`.

`mopidy.mpd.protocol.music_db.`**`listallinfo`**(*context*, *uri=None*)
musicpd.org, music database section:

`listallinfo [URI]`

Same as `listall`, except it also returns metadata info in the same format as `lsinfo`.

`mopidy.mpd.protocol.music_db.`**`lsinfo`**(*context*, *uri=None*)
musicpd.org, music database section:

`lsinfo [URI]`

Lists the contents of the directory `URI`.

When listing the root directory, this currently returns the list of stored playlists. This behavior is deprecated; use `listplaylists` instead.

MPD returns the same result, including both playlists and the files and directories located at the root level, for both `lsinfo`, `lsinfo ""`, and `lsinfo "/"`.

`mopidy.mpd.protocol.music_db.`**`readcomments`**(*context*, *uri*)
musicpd.org, music database section:

`readcomments [URI]`

Read "comments" (i.e. key-value pairs) from the file specified by "URI". This "URI" can be a path relative to the music directory or a URL in the form "file:///foo/bar.ogg".

This command may be used to list metadata of remote files (e.g. URI beginning with "http://" or "smb://").

The response consists of lines in the form "KEY: VALUE". Comments with suspicious characters (e.g. newlines) are ignored silently.

The meaning of these depends on the codec, and not all decoder plugins support it. For example, on Ogg files, this lists the Vorbis comments.

`mopidy.mpd.protocol.music_db.`**`rescan`**(*context*, *uri=None*)
musicpd.org, music database section:

`rescan [URI]`

Same as `update`, but also rescans unmodified files.

`mopidy.mpd.protocol.music_db.`**`search`**(*context*, **args*)
musicpd.org, music database section:

`search {TYPE} {WHAT} [...]`

Searches for any song that contains `WHAT`. Parameters have the same meaning as for `find`, except that search is not case sensitive.

GMPC:

- does not add quotes around the field argument.
- uses the undocumented field `any`.
- searches for multiple words like this:

```
search any "foo" any "bar" any "baz"
```

ncmpc:

- does not add quotes around the field argument.
- capitalizes the field argument.

ncmpcpp:

- also uses the search type "date".
- uses "file" instead of "filename".

mopidy.mpd.protocol.music_db.**searchadd**(*context*, **args*)
musicpd.org, music database section:

```
searchadd {TYPE} {WHAT} [...]
```

Searches for any song that contains `WHAT` in tag `TYPE` and adds them to current playlist.

Parameters have the same meaning as for `find`, except that search is not case sensitive.

mopidy.mpd.protocol.music_db.**searchaddpl**(*context*, **args*)
musicpd.org, music database section:

```
searchaddpl {NAME} {TYPE} {WHAT} [...]
```

Searches for any song that contains `WHAT` in tag `TYPE` and adds them to the playlist named `NAME`.

If a playlist by that name doesn't exist it is created.

Parameters have the same meaning as for `find`, except that search is not case sensitive.

mopidy.mpd.protocol.music_db.**update**(*context*, *uri=None*)
musicpd.org, music database section:

```
update [URI]
```

Updates the music database: find new files, remove deleted files, update modified files.

`URI` is a particular directory or song/file to update. If you do not specify it, everything is updated.

Prints `updating_db: JOBID` where `JOBID` is a positive number identifying the update job. You can read the current job id in the `status` response.

Playback

mopidy.mpd.protocol.playback.**consume**(*context*, *state*)
musicpd.org, playback section:

```
consume {STATE}
```

Sets consume state to `STATE`, `STATE` should be 0 or 1. When consume is activated, each song played is removed from playlist.

mopidy.mpd.protocol.playback.**crossfade**(*context*, *seconds*)
musicpd.org, playback section:

```
crossfade {SECONDS}
```

Sets crossfading between songs.

mopidy.mpd.protocol.playback.**mixrampdb**(*context*, *decibels*)
musicpd.org, playback section:

```
mixrampdb {deciBels}
```

Sets the threshold at which songs will be overlapped. Like crossfading but doesn't fade the track volume, just overlaps. The songs need to have MixRamp tags added by an external tool. 0dB is the normalized maximum volume so use negative values, I prefer -17dB. In the absence of mixramp tags crossfading will be used. See http://sourceforge.net/projects/mixramp

mopidy.mpd.protocol.playback.**mixrampdelay**(*context*, *seconds*)

musicpd.org, playback section:

```
mixrampdelay {SECONDS}
```

Additional time subtracted from the overlap calculated by mixrampdb. A value of "nan" disables MixRamp overlapping and falls back to crossfading.

mopidy.mpd.protocol.playback.**next_**(*context*)

musicpd.org, playback section:

```
next
```

Plays next song in the playlist.

MPD's behaviour when affected by repeat/random/single/consume:

Given a playlist of three tracks numbered 1, 2, 3, and a currently playing track `c`. `next_track` is defined at the track that will be played upon calls to `next`.

Tests performed on MPD 0.15.4-1ubuntu3.

repeat	random	single	consume	c = 1	c = 2	c = 3	Notes
T	T	T	T	2	3	EOPL	
T	T	T	.	Rand	Rand	Rand	[1]
T	T	.	T	Rand	Rand	Rand	[4]
T	T	.	.	Rand	Rand	Rand	[4]
T	.	T	T	2	3	EOPL	
T	.	T	.	2	3	1	
T	.	.	T	3	3	EOPL	
T	.	.	.	2	3	1	
.	T	T	T	Rand	Rand	Rand	[3]
.	T	T	.	Rand	Rand	Rand	[3]
.	T	.	T	Rand	Rand	Rand	[2]
.	T	.	.	Rand	Rand	Rand	[2]
.	.	T	T	2	3	EOPL	
.	.	T	.	2	3	EOPL	
.	.	.	T	2	3	EOPL	
.	.	.	.	2	3	EOPL	

- When end of playlist (EOPL) is reached, the current track is unset.
- [1] When *random* and *single* is combined, `next` selects a track randomly at each invocation, and not just the next track in an internal prerandomized playlist.
- [2] When *random* is active, `next` will skip through all tracks in the playlist in random order, and finally EOPL is reached.
- [3] *single* has no effect in combination with *random* alone, or *random* and *consume*.
- [4] When *random* and *repeat* is active, EOPL is never reached, but the playlist is played again, in the same random order as the first time.

mopidy.mpd.protocol.playback.**pause**(*context*, *state=None*)

musicpd.org, playback section:

> pause {PAUSE}
>
> Toggles pause/resumes playing, PAUSE is 0 or 1.

MPDroid:

- Calls pause without any arguments to toogle pause.

mopidy.mpd.protocol.playback.**play**(*context*, *tlid=None*)

musicpd.org, playback section:

> play [SONGPOS]
>
> Begins playing the playlist at song number SONGPOS.

The original MPD server resumes from the paused state on play without arguments.

Clarifications:

- play "-1" when playing is ignored.
- play "-1" when paused resumes playback.
- play "-1" when stopped with a current track starts playback at the current track.
- play "-1" when stopped without a current track, e.g. after playlist replacement, starts playback at the first track.

BitMPC:

- issues play 6 without quotes around the argument.

mopidy.mpd.protocol.playback.**playid**(*context*, *tlid*)

musicpd.org, playback section:

> playid [SONGID]
>
> Begins playing the playlist at song SONGID.

Clarifications:

- playid "-1" when playing is ignored.
- playid "-1" when paused resumes playback.
- playid "-1" when stopped with a current track starts playback at the current track.
- playid "-1" when stopped without a current track, e.g. after playlist replacement, starts playback at the first track.

mopidy.mpd.protocol.playback.**previous**(*context*)

musicpd.org, playback section:

> previous
>
> Plays previous song in the playlist.

MPD's behaviour when affected by repeat/random/single/consume:

> Given a playlist of three tracks numbered 1, 2, 3, and a currently playing track c. previous_track is defined at the track that will be played upon previous calls.
>
> Tests performed on MPD 0.15.4-1ubuntu3.

repeat	random	single	consume	c = 1	c = 2	c = 3
T	T	T	T	Rand?	Rand?	Rand?
T	T	T	.	3	1	2
T	T	.	T	Rand?	Rand?	Rand?
T	T	.	.	3	1	2
T	.	T	T	3	1	2
T	.	T	.	3	1	2
T	.	.	T	3	1	2
T	.	.	.	3	1	2
.	T	T	T	c	c	c
.	T	T	.	c	c	c
.	T	.	T	c	c	c
.	T	.	.	c	c	c
.	.	T	T	1	1	2
.	.	T	.	1	1	2
.	.	.	T	1	1	2
.	.	.	.	1	1	2

- If `time_position` of the current track is 15s or more, `previous` should do a seek to time position 0.

mopidy.mpd.protocol.playback.**random**(*context*, *state*)
musicpd.org, playback section:

`random {STATE}`

Sets random state to `STATE`, `STATE` should be 0 or 1.

mopidy.mpd.protocol.playback.**repeat**(*context*, *state*)
musicpd.org, playback section:

`repeat {STATE}`

Sets repeat state to `STATE`, `STATE` should be 0 or 1.

mopidy.mpd.protocol.playback.**replay_gain_mode**(*context*, *mode*)
musicpd.org, playback section:

`replay_gain_mode {MODE}`

Sets the replay gain mode. One of `off`, `track`, `album`.

Changing the mode during playback may take several seconds, because the new settings does not affect the buffered data.

This command triggers the options idle event.

mopidy.mpd.protocol.playback.**replay_gain_status**(*context*)
musicpd.org, playback section:

`replay_gain_status`

Prints replay gain options. Currently, only the variable `replay_gain_mode` is returned.

mopidy.mpd.protocol.playback.**seek**(*context*, *tlid*, *seconds*)
musicpd.org, playback section:

`seek {SONGPOS} {TIME}`

Seeks to the position `TIME` (in seconds) of entry `SONGPOS` in the playlist.

Droid MPD:

- issues `seek 1 120` without quotes around the arguments.

mopidy.mpd.protocol.playback.**seekcur**(*context*, *time*)
musicpd.org, playback section:

```
seekcur {TIME}
```

Seeks to the position `TIME` within the current song. If prefixed by '+' or '-', then the time is relative to the current playing position.

mopidy.mpd.protocol.playback.**seekid**(*context*, *tlid*, *seconds*)
musicpd.org, playback section:

```
seekid {SONGID} {TIME}
```

Seeks to the position `TIME` (in seconds) of song `SONGID`.

mopidy.mpd.protocol.playback.**setvol**(*context*, *volume*)
musicpd.org, playback section:

```
setvol {VOL}
```

Sets volume to `VOL`, the range of volume is 0-100.

Droid MPD:

- issues `setvol 50` without quotes around the argument.

mopidy.mpd.protocol.playback.**single**(*context*, *state*)
musicpd.org, playback section:

```
single {STATE}
```

Sets single state to `STATE`, `STATE` should be 0 or 1. When single is activated, playback is stopped after current song, or song is repeated if the `repeat` mode is enabled.

mopidy.mpd.protocol.playback.**stop**(*context*)
musicpd.org, playback section:

```
stop
```

Stops playing.

Reflection

mopidy.mpd.protocol.reflection.**commands**(*context*)
musicpd.org, reflection section:

```
commands
```

Shows which commands the current user has access to.

mopidy.mpd.protocol.reflection.**config**(*context*)
musicpd.org, reflection section:

```
config
```

Dumps configuration values that may be interesting for the client. This command is only permitted to "local" clients (connected via UNIX domain socket).

mopidy.mpd.protocol.reflection.**decoders**(*context*)
musicpd.org, reflection section:

```
decoders
```

Print a list of decoder plugins, followed by their supported suffixes and MIME types. Example response:

```
plugin: mad
suffix: mp3
suffix: mp2
mime_type: audio/mpeg
plugin: mpcdec
suffix: mpc
```

Clarifications:

- ncmpcpp asks for decoders the first time you open the browse view. By returning nothing and OK instead of an not implemented error, we avoid "Not implemented" showing up in the ncmpcpp interface, and we get the list of playlists without having to enter the browse interface twice.

mopidy.mpd.protocol.reflection.**notcommands**(*context*)
musicpd.org, reflection section:

```
notcommands
```

Shows which commands the current user does not have access to.

mopidy.mpd.protocol.reflection.**tagtypes**(*context*)
musicpd.org, reflection section:

```
tagtypes
```

Shows a list of available song metadata.

mopidy.mpd.protocol.reflection.**urlhandlers**(*context*)
musicpd.org, reflection section:

```
urlhandlers
```

Gets a list of available URL handlers.

Status

mopidy.mpd.protocol.status.**SUBSYSTEMS = [u'database', u'mixer', u'options', u'output', u'pl**
Subsystems that can be registered with idle command.

mopidy.mpd.protocol.status.**clearerror**(*context*)
musicpd.org, status section:

```
clearerror
```

Clears the current error message in status (this is also accomplished by any command that starts playback).

mopidy.mpd.protocol.status.**currentsong**(*context*)
musicpd.org, status section:

```
currentsong
```

Displays the song info of the current song (same song that is identified in status).

mopidy.mpd.protocol.status.**idle**(*context*, **subsystems*)
musicpd.org, status section:

`idle [SUBSYSTEMS...]`

Waits until there is a noteworthy change in one or more of MPD's subsystems. As soon as there is one, it lists all changed systems in a line in the format `changed: SUBSYSTEM`, where `SUBSYSTEM` is one of the following:

- `database`: the song database has been modified after update.
- `update`: a database update has started or finished. If the database was modified during the update, the database event is also emitted.
- `stored_playlist`: a stored playlist has been modified, renamed, created or deleted
- `playlist`: the current playlist has been modified
- `player`: the player has been started, stopped or seeked
- `mixer`: the volume has been changed
- `output`: an audio output has been enabled or disabled
- `options`: options like repeat, random, crossfade, replay gain

While a client is waiting for idle results, the server disables timeouts, allowing a client to wait for events as long as MPD runs. The idle command can be canceled by sending the command `noidle` (no other commands are allowed). MPD will then leave idle mode and print results immediately; might be empty at this time.

If the optional `SUBSYSTEMS` argument is used, MPD will only send notifications when something changed in one of the specified subsystems.

mopidy.mpd.protocol.status.**noidle**(*context*)
See `_status_idle()`.

mopidy.mpd.protocol.status.**stats**(*context*)
musicpd.org, status section:

`stats`

Displays statistics.

- `artists`: number of artists
- `songs`: number of albums
- `uptime`: daemon uptime in seconds
- `db_playtime`: sum of all song times in the db
- `db_update`: last db update in UNIX time
- `playtime`: time length of music played

mopidy.mpd.protocol.status.**status**(*context*)
musicpd.org, status section:

`status`

Reports the current status of the player and the volume level.

- `volume`: 0-100 or -1
- `repeat`: 0 or 1
- `single`: 0 or 1

- `consume`: 0 or 1
- `playlist`: 31-bit unsigned integer, the playlist version number
- `playlistlength`: integer, the length of the playlist
- `state`: play, stop, or pause
- `song`: playlist song number of the current song stopped on or playing
- `songid`: playlist songid of the current song stopped on or playing
- `nextsong`: playlist song number of the next song to be played
- `nextsongid`: playlist songid of the next song to be played
- `time`: total time elapsed (of current playing/paused song)
- `elapsed`: Total time elapsed within the current song, but with higher resolution.
- `bitrate`: instantaneous bitrate in kbps
- `xfade`: crossfade in seconds
- `audio`: sampleRate‘‘:bits‘‘:channels
- `updatings_db`: job id
- `error`: if there is an error, returns message here

Clarifications based on experience implementing

- `volume`: can also be -1 if no output is set.
- `elapsed`: Higher resolution means time in seconds with three decimal places for millisecond precision.

Stickers

`mopidy.mpd.protocol.stickers.`**`sticker`**(*context*, *action*, *field*, *uri*, *name=None*, *value=None*)

musicpd.org, sticker section:

`sticker list {TYPE} {URI}`

Lists the stickers for the specified object.

`sticker find {TYPE} {URI} {NAME}`

Searches the sticker database for stickers with the specified name, below the specified directory (`URI`). For each matching song, it prints the `URI` and that one sticker's value.

`sticker get {TYPE} {URI} {NAME}`

Reads a sticker value for the specified object.

`sticker set {TYPE} {URI} {NAME} {VALUE}`

Adds a sticker value to the specified object. If a sticker item with that name already exists, it is replaced.

`sticker delete {TYPE} {URI} [NAME]`

Deletes a sticker value from the specified object. If you do not specify a sticker name, all sticker values are deleted.

Stored playlists

mopidy.mpd.protocol.stored_playlists.**listplaylist**(*context*, *name*)
musicpd.org, stored playlists section:

```
listplaylist {NAME}
```

Lists the files in the playlist `NAME.m3u`.

Output format:

```
file: relative/path/to/file1.flac
file: relative/path/to/file2.ogg
file: relative/path/to/file3.mp3
```

mopidy.mpd.protocol.stored_playlists.**listplaylistinfo**(*context*, *name*)
musicpd.org, stored playlists section:

```
listplaylistinfo {NAME}
```

Lists songs in the playlist `NAME.m3u`.

Output format:

Standard track listing, with fields: file, Time, Title, Date, Album, Artist, Track

mopidy.mpd.protocol.stored_playlists.**listplaylists**(*context*)
musicpd.org, stored playlists section:

```
listplaylists
```

Prints a list of the playlist directory.

After each playlist name the server sends its last modification time as attribute `Last-Modified` in ISO 8601 format. To avoid problems due to clock differences between clients and the server, clients should not compare this value with their local clock.

Output format:

```
playlist: a
Last-Modified: 2010-02-06T02:10:25Z
playlist: b
Last-Modified: 2010-02-06T02:11:08Z
```

Clarifications:

- ncmpcpp 0.5.10 segfaults if we return 'playlist: ' on a line, so we must ignore playlists without names, which isn't very useful anyway.

mopidy.mpd.protocol.stored_playlists.**load**(*context*, *name*, *playlist_slice=slice(0, None, None)*)
musicpd.org, stored playlists section:

```
load {NAME} [START:END]
```

Loads the playlist into the current queue. Playlist plugins are supported. A range may be specified to load only a part of the playlist.

Clarifications:

- `load` appends the given playlist to the current playlist.
- MPD 0.17.1 does not support open-ended ranges, i.e. without end specified, for the `load` command, even though MPD's general range docs allows open-ended ranges.

- MPD 0.17.1 does not fail if the specified range is outside the playlist, in either or both ends.

mopidy.mpd.protocol.stored_playlists.**playlistadd**(*context*, *name*, *uri*)
musicpd.org, stored playlists section:

```
playlistadd {NAME} {URI}
```

Adds `URI` to the playlist `NAME.m3u`.

`NAME.m3u` will be created if it does not exist.

mopidy.mpd.protocol.stored_playlists.**playlistclear**(*context*, *name*)
musicpd.org, stored playlists section:

```
playlistclear {NAME}
```

Clears the playlist `NAME.m3u`.

mopidy.mpd.protocol.stored_playlists.**playlistdelete**(*context*, *name*, *songpos*)
musicpd.org, stored playlists section:

```
playlistdelete {NAME} {SONGPOS}
```

Deletes `SONGPOS` from the playlist `NAME.m3u`.

mopidy.mpd.protocol.stored_playlists.**playlistmove**(*context*, *name*, *from_pos*, *to_pos*)
musicpd.org, stored playlists section:

```
playlistmove {NAME} {SONGID} {SONGPOS}
```

Moves `SONGID` in the playlist `NAME.m3u` to the position `SONGPOS`.

Clarifications:

- The second argument is not a `SONGID` as used elsewhere in the protocol documentation, but just the `SONGPOS` to move *from*, i.e. `playlistmove {NAME} {FROM_SONGPOS} {TO_SONGPOS}`.

mopidy.mpd.protocol.stored_playlists.**rename**(*context*, *old_name*, *new_name*)
musicpd.org, stored playlists section:

```
rename {NAME} {NEW_NAME}
```

Renames the playlist `NAME.m3u` to `NEW_NAME.m3u`.

mopidy.mpd.protocol.stored_playlists.**rm**(*context*, *name*)
musicpd.org, stored playlists section:

```
rm {NAME}
```

Removes the playlist `NAME.m3u` from the playlist directory.

mopidy.mpd.protocol.stored_playlists.**save**(*context*, *name*)
musicpd.org, stored playlists section:

```
save {NAME}
```

Saves the current playlist to `NAME.m3u` in the playlist directory.

CHAPTER 7

Indices and tables

- genindex
- modindex

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

Z