Xintong Hao

[hxtong@bu.edu](mailto:hxtong@bu.edu)

https://github.com/XintongHao/EC500\_C1

**Mopidy issue#1599**

**Introduction of Mopidy**

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.

**Flow diagram**

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. The **core actor** uses the mixer actor to control volume, while the backends use the audio actor to play audio.

digraph overall_architecture {
"Multiple frontends" -> Core
Core -> "Multiple backends"
Core -> Mixer
"Multiple backends" -> Audio
}

**Frontends** expose Mopidy to the external world. They can implement servers for protocols like HTTP, 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.

digraph frontend_architecture {
"HTTP\nfrontend" -> Core
"MPD\nfrontend" -> Core
"MPRIS\nfrontend" -> Core
"Scrobbler\nfrontend" -> Core
}

**The core** is organized as a set of controllers with responsibility 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.

digraph core_architecture {
Core -> "Tracklist\ncontroller"
Core -> "Library\ncontroller"
Core -> "Playback\ncontroller"
Core -> "Playlists\ncontroller"
Core -> "History\ncontroller"

"Library\ncontroller" -> "Local backend"
"Library\ncontroller" -> "Spotify backend"

"Playback\ncontroller" -> "Local backend"
"Playback\ncontroller" -> "Spotify backend"
"Playback\ncontroller" -> Audio

"Playlists\ncontroller" -> "Local backend"
"Playlists\ncontroller" -> "Spotify backend"
}

**The backends** are organized as a set of providers with responsibility 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.

**URIs and routing of requests to the backend**:

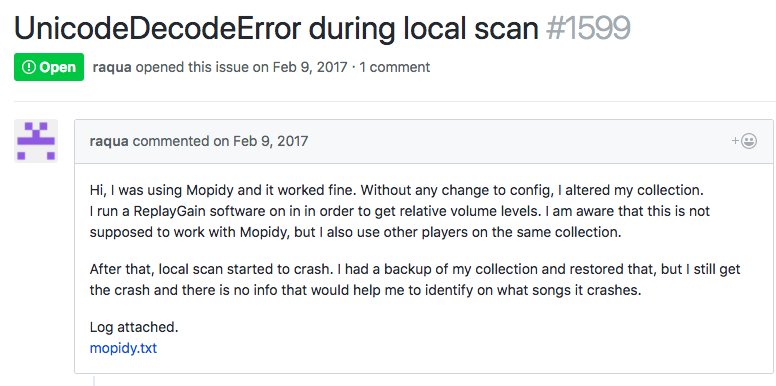
When Mopidy’s core layer is processing a client request, it routes the request to one or more appropriate backends based on the URIs of the objects the request touches on. The objects’ URIs are compared with the backends’ *uri\_schemes* to select the relevant backends.

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.

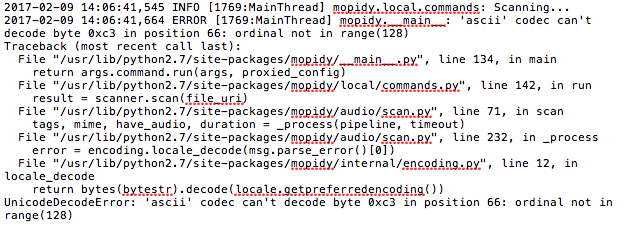
digraph backend_architecture {
"Local backend" -> "Local\nlibrary\nprovider" -> "Local disk"
"Local backend" -> "Local\nplayback\nprovider" -> "Local disk"
"Local backend" -> "Local\nplaylists\nprovider" -> "Local disk"
"Local\nplayback\nprovider" -> Audio

"Spotify backend" -> "Spotify\nlibrary\nprovider" -> "Spotify service"
"Spotify backend" -> "Spotify\nplayback\nprovider" -> "Spotify service"
"Spotify backend" -> "Spotify\nplaylists\nprovider" -> "Spotify service"
"Spotify\nplayback\nprovider" -> Audio
}

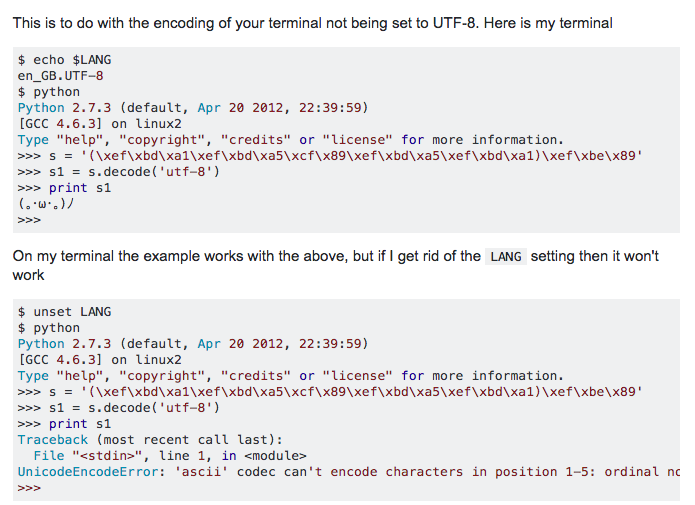
**Mopidy issue#1599**

****

**error description in the *mopidy.txt*:**

****

In the mopidy.txt, the error might be that the user’s locale LANG variable is unset. From an answer of a similar stack overflow issue, it shows how to set LANG by using *$ echo $LANG.*



In the process of [Fix/1599 catch unicode decoding errors in encoding.locale\_decode()](https://github.com/mopidy/mopidy/pull/1633).

At first, the contributor just proposes that *local\_decode()* returns a string to scan.py if it fails with decoding. The error string could be "*Code can't decode content. Check your LANG setting ($ echo $LANG)*.". Then the scanner stops at the track that failed, thus giving better feedback to the user.

However, it didn’t fix the error because it only shows the error handling without replacing with UTF-8 or alternatively codes. What the users want is to set LANG automatically and run their local scan successfully. In order to solve the error of decoding, the contributor edits an *encoding.py*. This exception can happen when locale is unset, which effectively means that ascii char set is used. In the file, when raise the *UnicodeDecodeError*, it will return of a string decoded with *UTF-8* using *errors="replace",* so it's easy to identify positions in the string that couldn't be decoded. The contributor also adds a test file to check that *local\_decode()* can handle ascii locale.

From the whole fix process above, I learnt how to figure out the bugs by searching for similar errors and how to catch exceptions and fix the error by replacing the setting in the exception.

