PedalPi - Application Documentation

Release 1

SrMouraSilva

Mar 15, 2018

Contents

Pedal Pi - Application is a framework for manager the Pedal Pi. Through it is possible loads Pedal Pi Components to provide a Human Machine Interface (HMI) or even have an opening for other software to consume the features of the Pedal Pi.

The components developed use the API (available through Controllers) to manage the resources of the Pedal Pi.

Documentation: http://pedalpi-application.readthedocs.io/

Code: https://github.com/PedalPi/Application

Python Package Index: https://pypi.org/project/PedalPi-Application

License: Apache License 2.0

CHAPTER 1

Running Application

Following are the steps required to set up and run Pedal Pi - Application.

1.1 Prepare ambient

Install with pip

```
pip3 install PedalPi-Application
```

Create the script file that contains the code to run the application (as example `start.py`)

```
from application.application import Application

application = Application(path_data="data/", address='localhost')

application.start()

from signal import pause
try:
    pause()
except KeyboardInterrupt:
    application.stop()
```

Download, compile and install mod-host. Mod-host is a *LV2 host for Jack controllable via socket or command line*. It is developed by Mod Devices, a company that also develops professional equipment for musicians.

```
git clone https://github.com/moddevices/mod-host
cd mod-host
make
make install
```

1.2 Run Application

Start audio process. The required settings for your audio card can vary greatly. I recommend that you try different possibilities in order to minimize the latency and number of *xruns*.

If you do not have any experience with JACK, is recommend the lecture of Demystifying JACK – A Beginners Guide to Getting Started with JACK from **Linux Music Production**.

```
# In this example, is starting a Zoom g3 series audio interface
jackd -R -P70 -t2000 -dalsa -dhw:Series -p256 -n3 -r44100 -s &
mod-host &
```

Finally, start the application

```
python3 start.py
```

CHAPTER 2

Extending

It's possible add or extends the Pedal Pi with addiction of *Component*. A component can provides a Human Machine Interface (HMI) - like Raspberry P0 - or even have an opening for other software to consume the features of the Pedal Pi - like WebService plugin.

See the github Components Project for complete components list.

To add a component in your configuration file, download it and register it before starting the application (`application.start()`):

```
pip3 install PedalPi-<component name>
```

```
from application.Application import Application
application = Application(path_data="data/", address='localhost')

# Loading component
from raspberry_p0.raspberry_p0 import RaspberryP0
application.register(RaspberryP0(application))

# Start application
application.start()

# Don't stop application
from signal import pause
try:
    pause()
except KeyboardInterrupt:
    # Stop components with safety
    application.stop()
```

Each component needs a configuration to work. Pay attention to your documentation for details on how to set it up and use it.

CHAPTER 3

Delegating audio processing to other equipment

The connection with mod-host is over TCP. So it's possible to place a machine to perform the processing and another to provide the control services.

For example, you have a Raspberry Pi B+ and a PC.

- The PC in http://10.0.0.100 will process the audio, then it will execute *jack* process, *mod-host* process and the audio interface will be connected to it.
- The *RPi* will executes *Application* with *Component*, like Raspberry P0 component. Raspberry P0 disposes a simple current pedalboard control.

```
application = Application(path_data="data/", address='10.0.0.100')
```

CHAPTER 4

Creating a component

Subsequently will be added details in the documentation on how to create a component for the Pedal Pi. For now, you can check the blog post Building a Pedal Pi Component - Pedalboard selector

CHAPTER 5

Maintenance

5.1 Test

The purpose of the tests is:

- Check if the notifications are working, since the module plugins manager is responsible for testing the models;
- Serve as a sample basis.

```
make test
make test-details
```

5.2 Generate documentation

This project uses Sphinx + Read the Docs.

You can generate the documentation in your local machine:

```
make install-docs-requirements
make docs

make docs-see
```

CHAPTER 6

Changelog

6.1 Changelog

6.1.1 Version 0.4.1 - released 03/15/18

- Improve PluginsManager dependency: Now supports path releases

6.1.2 Version 0.4.0 - released 02/18/18

- Issue #55 - Fix bank uuid/name
- Issue #58 - Support v0.6.0: Add connection type (audio or midi) in default bank
- Issue #60 - Add makefile
- Issue #46 - Add changelog in docs

6.1.3 Version 0.3.0 - released 05/30/17

- Issue #29 - Secure components close
- Issue #30 - Replace print log to logging
- **Breaking change**: Issues #39 and #5 - Change save method to pluginsmanager (v0.5.0) Autosaver
  - Removed BanksDao -> Using now pluginsmanager Autosaver
  - Removed Database -> Using now pluginsmanager Persistence
  - BanksController, PedalboardController, EffectController, ParamController, CurrentController changes your API
- Issue #41 - Allows current pedalboard is None
- Issue #40 - If current pedalboard index file is wrong, Application now starts with the current pedalboard = None

- Issue #11 - Banks with same name not will be replaced when Application initialize
- Issue #17 - Fixes: Remove bank with current pedalboard will be crash (when reload Application)
- Issue #45 - Add plugins manager v0.5.0 support
  - Removed BanksController, PedalboardController, EffectController, ParamController, NotificationController
  - Implemented `Application.register_observer()`, `Application.unregister_observer()`

6.1.4 Version 0.2.1 - released 04/14/17

- 21fdb32 Issue #30 - Fix move pedalboard notification
- Fix Readme: Pipy render README.rst
- fbb9908 - Add Licenses in __init__.py files

6.1.5 Version 0.2.0 - released 04/05/17

- Initial release

CHAPTER 7

API

Contents:

7.1 PedalPi - Application

7.2 PedalPi - Application - Component

7.2.1 Creating a component

Subsequently will be added details on how to create a component for the Pedal Pi. For now, you can check the blog post Building a Pedal Pi Component - Pedalboard selector

7.2.2 Component

class application.component.component.**Component**(*application*)

close()
Method called when the application is requested to quit. Classes components must implement to safely finish their activities.

init()
Initialize this component

register_observer(*observer*)
Calls `Application.register_observer()`.

Parameters **observer** (*ApplicationObserver*) – The observer who will receive the changes notifications

unregister_observer(*observer*)
Calls `Application.unregister_observer()`.

Parameters **observer** (*ApplicationObserver*) – The observer who will not receive further changes notification

7.2.3 ApplicationObserver

7.2.4 CurrentPedalboardObserver

7.3 PedalPi - Application - Controller

7.3.1 Notification scope

pluginsmanager can notifies they changes. As an example, if a connection between effects is created, plugins manager notifies its observers about the change.

This is how `ModHost` and `Autosaver` know when a change occurs.

These observers work passively: they only receive updates, not using the pluginsmanager api to change the state of the application.

Man-Machine Interfaces are usually active: they need to change the state of the application. As an example, a button that leaves bypass an effect. They also need to receive notifications, so that the information presented to the user can be updated in accordance with changes made by other interfaces.

In these cases, is necessary in a change notifiers all except the one who caused the change.

As example, a multi-effects uses Raspberry-P1 for physical management and WebService for a controller with Apk controller. If they uses only *pluginsmanager*, a toggle status effect change in a Raspberry-P1 will informs WebService and unreasonably Raspberry-P1.

A quick review will be given ahead. For more details, see the pluginsmanager observer documentation.

pluginsmanager has a solution to this problem. Defining a observer:

```
class MyAwesomeObserver(UpdatesObserver):

    def __init__(self, message):
        self.message = message

    def on_bank_updated(self, bank, update_type, **kwargs):
        print(self.message)

    # Defining others abstract methods
    ...
```

Using:

```
>>> observer1 = MyAwesomeObserver("Hi! I am observer1")
>>> observer2 = MyAwesomeObserver("Hi! I am observer2")
>>>
>>> manager = BanksManager()
>>> manager.register(observer1)
>>> manager.register(observer2)
>>>
>>> bank = Bank('Bank 1')
>>> manager.banks.append(bank)
"Hi! I am observer1"
"Hi! I am observer2"
```

```
>>> with observer1:
>>>     del manager.banks[0]
"Hi! I am observer2"
>>> with observer2:
>>>     manager.banks.append(bank)
"Hi! I am observer1"
```

Using **application**, the process changes a bit. Because pluginsmanager does not support the current pedalboard change notifications, clients should extend from `ApplicationObserver`, a specialization that adds this functionality:

```
class MyAwesomeObserver(ApplicationObserver):

    def __init__(self, message):
        self.message = message

    def on_current_pedalboard_changed(self, pedalboard, **kwargs):
        print('Pedalboard changed!')

    def on_bank_updated(self, bank, update_type, **kwargs):
        print(self.message)

    # Defining others abstract methods
    ...
```

To correctly register ApplicationObserver, you must use `Application.register_observer()` (or *`Component.register_observer()`*):

```
>>> observer1 = MyAwesomeObserver("Hi! I am observer1")
>>> observer2 = MyAwesomeObserver("Hi! I am observer2")
>>>
>>> application.register_observer(observer1)
>>> application.register_observer(observer2)
```

Note: Registering directly to the pluginsmanager will result in not receiving updates defined by `ApplicationObserver`

Using:

```
>>> manager = application.manager
>>>
>>> bank = Bank('Bank 1')
>>> manager.banks.append(bank)
"Hi! I am observer1"
"Hi! I am observer2"
>>> with observer1:
>>>     del manager.banks[0]
"Hi! I am observer2"
>>> with observer2:
>>>     manager.banks.append(bank)
"Hi! I am observer1"
```

Warning: The operations performed by PluginsManager **are not atomic**. This architectural constraint was based on the experienced experience that one user will use the system at a time. In this way, try not to abuse the concurrence.

If you are having problems while doing this, let us know.

7.3.2 Controller

class application.controller.controller.**Controller**(*application*)
Abstract class for Application controllers.

Extends to offer functionalities for this API. Remember to manually register the extended class in `Application` (in private `_load_controllers` method)

Parameters **application** (*Application*) – `Application` instance

close()
The *close* method is called by `Application` when application termination is requested

configure()
The *configure* method is called by `Application` for initialize this object

7.3.3 ComponentDataController

7.3.4 CurrentController

7.3.5 DeviceController

7.3.6 PluginsController

7.4 PedalPi - Application - Dao

Dao classes provide a means to persist information.

Warning: When creating a component, the model informations are persisted by `Autosaver` class.

Warning: If you need persists and load any data, use the `ComponentDataController`.

7.4.1 ComponentDao

7.4.2 CurrentDao

7.4.3 PluginsDao

Index