Smart Home Receiver - Documentation

Introduction

Smart Home Remote Control is an application used as smart home commands receiver and executor. It is possible because of the device Raspberry Pi which allows us to run this program and connect smart home devices. This app is placed on the end of the smart home program system chain - it means that Raspberry with this app are dedicated to control all connected devices at home. What more, there's no markup on our smart scheme - we can setup everything like we really want to.

Instruction

This ip is quite easy to use. All you need to know is that, you can run this program on Raspberry Pi by typing "python main.py" in terminal from main directory of project. Next step is to enter correct IP address and port, then you will be able to see all devices statuses. If you want to setup devices to the program, you should check Engine/ConfigSetup directory.

Technologies

My platform was Raspberry Pi 3. Used technologies in this application:

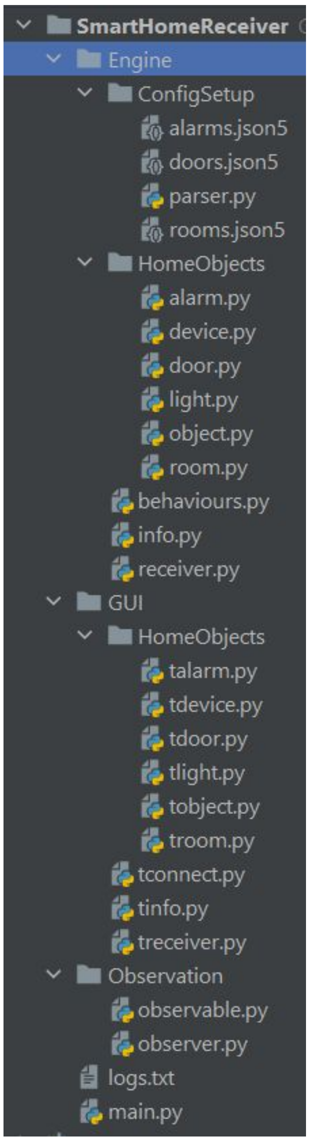
- python - native language,
- tkinter - graphic interface of the application,
- MQTT - data transmission protocol based on publish/subscribe pattern. In this case, protocol is used to communication with smart home system - remote control publishes commands in specific topic and server sends them to all topic subscribers,
- JSON - from this type of files, configuration is loaded,
- Observer pattern - it is an intermediary between app engine and graphic interface

Construction

In main directory there is a main file named "main.py" which is used to start the receiver and there are also three other directories:

- Engine - an app engine,
- GUI - graphical interface for application,
- Observation - it connects the engine with graphical interface.

The most important directory is Engine. Without an Engine, the receiver can’t run, but without GUI or Observer, it can still be run. If you want to run this app with Engine only, you have to implement integration between user and engine via console. Engine includes JSON files with all the configuration and parser.py file, which converts JSON objects into objects based on classes from HomeObjects directory. This directory has all class definitions which represents objects to control. Object class in object.py is a root of any other class - Device, Alarm, Door inherit from Object but Light does not - it’s because of differences between this class and Object class. Other files from engine directory provide grounds for receiver functionality. Info class informs only about connection (port and IP address) and provides entering these data. Receiver class is a root of this application - it connects with server, start your application and receives specific commands from the server. Behaviours definitions are placed in Behaviours class. They describe what to do when you get the message from the server. This specific construction of application is a necessary condition of receiver running and sufficient condition when you have an integration via console written. It is known that a typical user of this app doesn't have to be a software developer, that’s why GUI is written here which is defined in the GUI directory. It is responsible for displaying all devices statuses. HomeObject directory includes tkinter class versions with reference to class objects from HomeObjects in the Engine directory. TInfo class displays everything that the Info object contents. TConnectWindow (TConnect file) is the class which represents a start window where you need to enter the correct server IP address and port. After clicking the “start” button, you will be able to see a receiver panel which is defined in TReceiver class. Everything could be fine but there is no connection between engine and graphic interface - when something changes in Engine, GUI doesn’t know that it is changed. The solution for this problem is Observer Pattern included in the Observation directory. Engine objects are Observable objects (inheritance) but GUI objects are Observer objects. When something changes in engine objects, all observer objects will be notified (in this case - GUI objects which will change their data on the panel). Thanks to this pattern, I got not only efficient upgrade possibilities but also engine and GUI severability. This is a good solution when you want to make a lot of complicated modifications - if the engine had been written to cooperate with a graphic interface, GUI technology replacement would have been impossible without engine changes.



Development

Thanks to construction, which is described above, this application is ready for changes. For example you can write GUI in another technology like PyQt, where you don't have to use Observer pattern because every event you can hold by signals and slots. The only thing you have to remember is to delete the observable heritance in Engine/HomeObjects classes and this is the only one thing from external directories and classes which is used in implementation of the receiver engine. If you want to just extend this application (not modify), it will be possible thanks to the construction also. For example - to add new functionality, you need to create a new Class in a new file, implement all needed functionality (Obj heritance can simplify something), set it in Receiver class, create GUI equivalent (optionally) and just use it as you want (but remember to set it in JSON, without pinout it isn't possible to use it).To sum app, application is written not only to enable modifications but also to allow you to extend it and develop it in your way.

Author:

Łukasz Wolski