Smart Home Receiver - Dokumentacja

Wstęp

Smart Home Receiver to oprogramowanie służące do odbioru sygnałów związanych z inteligentnym domem i egzekucji ich. Jest to możliwe dzięki urządzeniu Raspberry Pi, na którym można to oprogramowanie uruchomić i podpiąć do niego odpowiednie urządzenia. Stanowi ostatni element łańcucha programowego w systemie - jest realizowany na sprzęcie, dzięki czemu każde polecenie z systemu jest realizowane w rzeczywistości poprzez sterowanie napięciem na odpowiednich pinach w zależności od otrzymanego sygnału. Nie narzuca nam korzystania z odpowiednich pinów i urządzeń - wszystko możemy zdefiniować według własnych potrzeb w plikach konfiguracyjnych typu JSON.

Instrukcja

Jest to mimo pozoru bardzo łatwe w obsłudze oprogramowanie. Wystarczy z katalogu głównego uruchomić w konsoli “python main.py” i wyświetli nam się okienko połączeniowe, w którym wpisujemy dane do połączenia z serwerem (adres IP, port). Po zatwierdzeniu powinno nam się pojawić małe, proste okno ze statusami urządzeń podłączonych do Raspberry Pi. Tak wygląda uruchomienie aplikacji, jeżeli chodzi zaś o konfigurację, to wszystko można modyfikować wedle własnych potrzeb w plikach JSON w katalogu Engine/ConfigSetup.

Technologie

Korzystałem z Raspberry Pi 3. Wykorzystane technologie w tej aplikacji to:

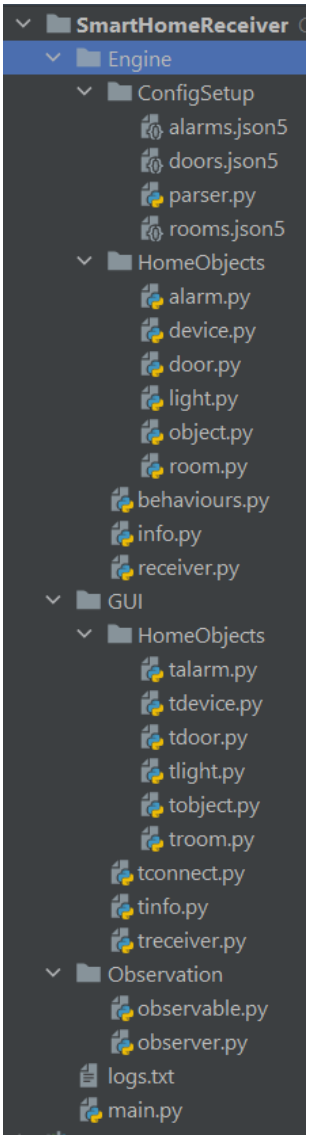
- python - język w którym został napisany odbiornik,
- tkinter - graficzny interfejs aplikacji (uruchamiany poglądowo, istotny na początku podczas połączenia),
- MQTT - protokół transmisji danych, oparty o wzorzec publish/subscribe. W tym przypadku jest wykorzystywany do komunikacji z systemem inteligentnego domu - ten program otrzymuje polecenia i wykonuje je poprzez zmiany na pinach,
- JSON - z plików tego typu wczytywana jest konfiguracja odbiornika,
- wzorzec projektowy Observer - jest pośrednikiem pomiędzy silnikiem aplikacji a jej graficznym interfejsem.

Budowa

Cechą charakterystyczną budowy tego projektu jest struktura katalogowa. W katalogu głównym znajdują się plik główny main.py który odpowiada za uruchomienie odbiornika oraz trzy katalogi:

- Engine - stanowi silnik aplikacji,
- GUI - graficzny interfejs,
- Observation - pośrednik między silnikiem a interfejsem

Najważniejszym katalogiem jest Engine. Bez niego aplikacja nie ma szans zadziałać. Bez graficznego interfejsu i pośrednika w postaci wzorca Observer aplikację można uruchomić, jednak będzie to wymagało dodatkowego wysiłku - należy ustalić adres i port połączenia, co jest możliwe domyślnie z poziomu interfejsu użytkownika. Sam silnik składa się na katalog odpowiedzialny za ładowanie konfiguracji odbiornika - są to pliki formatu JSON, które prezentują schemat oraz plik parser.py, który konwertuje obiekty formatu JSON na obiekty typu takiego jak te zdefiniowane w katalogu HomeObjects. Ten katalog z kolei zawiera definicje różnych klas na których pracuje aplikacja. Klasa Object w pliku object.py stanowi korzeń innych klas - Device, Light, Alarm, Door dziedziczą po klasie Object, przy czym Light nie rozszerza jej, ze względu na dużą różnorodność - nie mamy tutaj do czynienia z trybem “on/off” tylko ze zmianą jasności. Pozostałe pliki z katalogu silnika stanowią istotną rolę w funkcjonalności odbiornika. Klasa Info odpowiada za ustalenie danych połączenia. Klasa Receiver to rdzeń aplikacji - odpowiada za połączenie z serwerem, uruchomienie oraz odbiór poleceń. Definicje zachowań w przypadku odbioru polecenia, znajdują się w klasie Behaviours. Tak skonstruowany silnik jest warunkiem koniecznym działania pilota i wystarczającym w przypadku gdy mamy napisaną również integrację z użytkownikiem za pomocą konsoli. Wiadomym jest, że jednak typowym użytkownikiem nie będzie programista tylko również osoba niedoświadczona w informatyce, zatem rozwiązanie konsolowe nie jest najlepszym rozwiązaniem. Na pomoc przychodzi tutaj folder GUI - odpowiada on za wpisanie danych do połączenia oraz poglądowego wyświetlenia statusów podłączonych urządzeń. Katalog HomeObjects zawiera tkinter’owe wersje klas z referencjami na obiekty klas z HomeObjects w Engine. TInfo to klasa, która wyświetla dane z klasy Info z silnika. TConnectWindow (plik TConnect) to klasa która stanowi okno łączenia się z serwerem. Jest to pierwsze okno jakie pojawia nam się po uruchomieniu receivera. Tam wpisujemy adres IP oraz port serwera, a po potwierdzeniu tego przyciskiem “start”, pojawia nam się okno odbiornika, którego implementacja zawarta jest w klasie TReceiver. Zarówno w tym jak i innym przypadku, pliki katalogu GUI muszą korzystać z plików silnika, co jednak jeżeli wartości obiektów z silnika dynamicznie zmieniają się a nie widac tego w interfejsie? Rozwiązaniem tego problemu jest właśnie wzorzec projektowy Observer zawarty w katalogu Observation. Obiektami typu Observable są obiekty utworzony na podstawie klas silnika, zaś obiektami typu Observer są obiekty utworzone na podstawie klas graficznego interfejsu. W przypadku kiedy zmieni się jakaś wartość na obiektach z silnika, wszyscy obserwatorzy się o tym dowiedzą (w tym przypadku obiekty interfejsu) i na moment kiedy otrzymują to powiadomienie, zmieniają dane na panelu odbiornika. Dzięki zastosowaniu tego wzorca, uzyskałem nie tylko sprawną aktualizację danych, ale również rozłączność silnika aplikacji oraz silnika graficznego. Jest to szczególnie dobre rozwiązanie w przypadku, kiedy chcielibyśmy dokonać poważnej modyfikacji aplikacji - o tym w następnej sekcji.



Rozwój

Dzięki budowie, która została opisana powyżej, aplikacja jest gotowa do zmian. Nawet jeżeli nie są one konieczne, to bardzo łatwo jest ich dokonać. Separacja interfejsu graficznego i silnika umożliwia nam zmianę technologii w jakiej napiszemy GUI - gdyby silnik został napisany tak aby współpracował z silnikiem graficznym to wtedy zmiana graficznego interfejsu nie byłaby możliwa bez zmian w silniku. Jednak nie to jest sercem aplikacji, a nasz silnik, zatem co z rozwojem silnika? W każdej chwili możemy dodać kolejne typy urządzenia w bardzo prosty sposób - tworzymy klasę w nowym pliku, która dziedziczy po Obj (lub nie, zależy od kategorii urządzenia) i implementuje odpowiednie metody służące do sterowania sygnałami na pinach. Następnie tworzymy graficzny odpowiednik tej klasy (wszystko w odpowiednich katalogach) i definiujemy jak ma być prezentowane to na mini-interfejsie. Z kolei jeżeli chcielibyśmy dodać jakąś nową funkcjonalność do utworzonych już typów urządzeń w systemie, wystarczy ją zdefiniować w odpowiedniej klasie tak aby była zgodna z projektem - należy pamiętać że jest to odbiornik realizowany na Raspberry Pi, więc cała magia odbywa się finalnie na pinach. Jakiekolwiek powyższe zmiany wprowadzane w aplikacji powinny wystarczyć do prawidłowego działania aplikacji, ewentualnie mogą wymagać drobnych zmian w klasie Receiver. Podsumowując - jest to prosty moduł, który realizuję nam swego rodzaju “translację” poleceń w postaci linijek słów na konkretne zdarzenia rzeczywiste i w dodatku jest jak najbardziej przystosowany do zmian.

Autor:

Łukasz Wolski