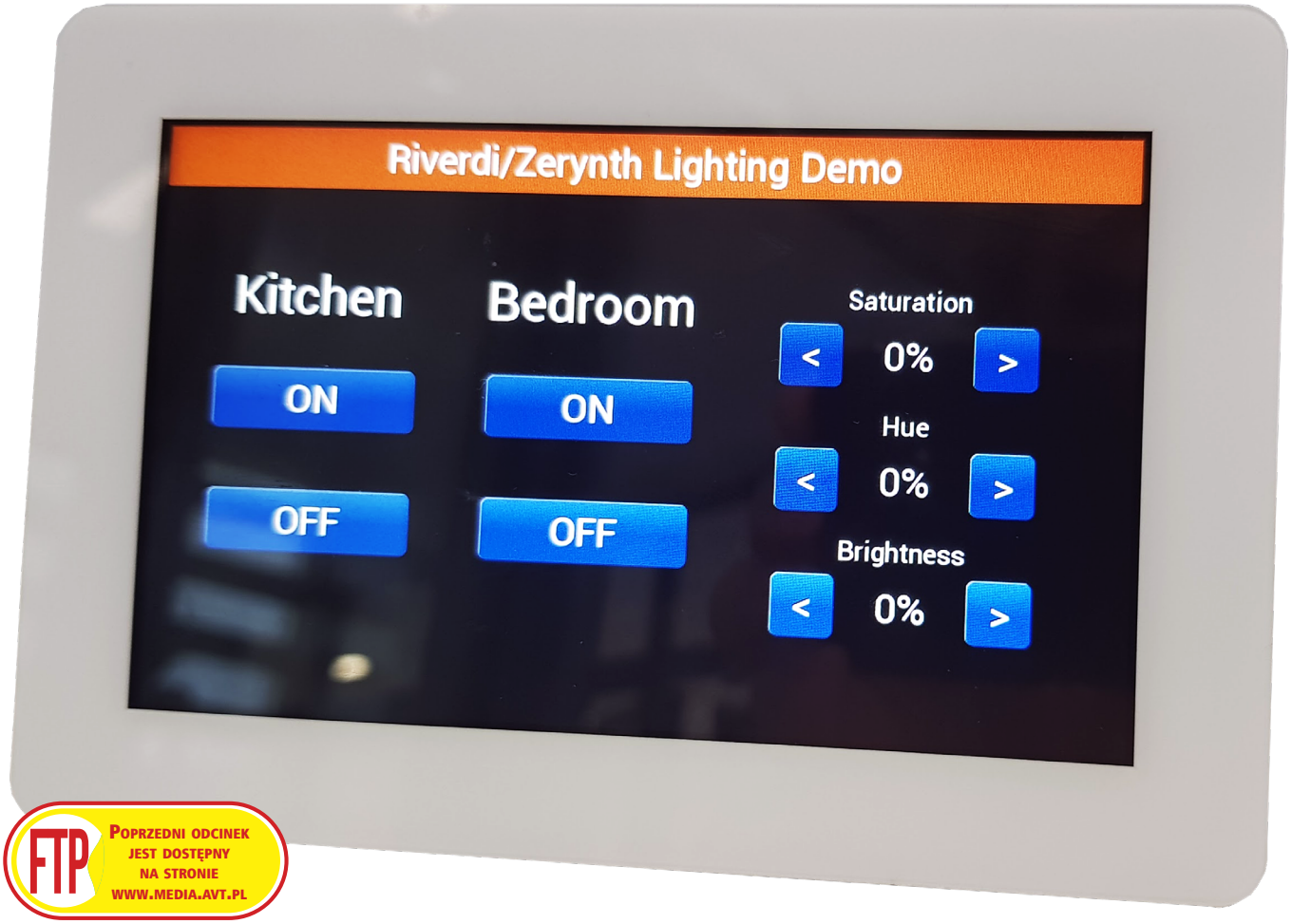


FTP Poprzedni odcinek jest dostępny na stronie www.media.avt.pl

System sterowania oświetleniem Philips Hue

z użyciem środowiska Zerynth oraz języka Python (2)

Prezentujemy prosty system sterowania żarówkami z serii Philips Hue zbudowany przy użyciu zintegrowanego modułu wyświetlacza Riverdi IoT Display oraz środowiska programistycznego Zerynth – umożliwiającego łatwe i szybkie tworzenie aplikacji z użyciem języka Python. W drugiej części artykułu przygotujemy główny panel kontrolny naszego systemu.

W kolejnym kroku, po poprawnym nawiązaniu połączenia z siecią Wi-Fi, możemy przystąpić do utworzenia interfejsu graficznego, umożliwiającego wprowadzenie adresu IP, przypisanego do urządzenia Philips Hue Bridge. Do budowy interfejsu zastosujemy metodę `bt81x.add_keys()`, która umieszcza na ekranie pojedynczy wiersz przycisków – dla przykładowego wywołania:

```
bt81x.add_keys(450, 70, 280, 60, 30, 0, "123")
```

Uzyskany efekt pokazuje **rysunek 4**.

Okno wprowadzania adresu IP jest pierwszym elementem budowanego interfejsu użytkownika, który wymaga obsługi panelu dotykowego. Dzięki możliwości przypisania unikalnego identyfikatora *TAG* do każdego tworzonego obiektu graficznego programista jest zwolniony z obowiązku odczytu surowych danych, w postaci współrzędnych X i Y. Obsługa panelu dotykowego wymaga od programisty wyłącznie wywołania funkcji `bt81x.touch_loop()`, która wskazuje funkcje zwrotne obsługi zdarzenia dotyku. Dla każdej wartości parametru *TAG* istnieje możliwość przypisania wydzielonej funkcji obsługi zdarzenia lub, jak ma to miejsce w omawianej aplikacji, przygotowanie jednej funkcji zwrotnej do obsługi wszystkich zdarzeń (wówczas argument funkcji `bt81x.touch_loop()` przyjmuje wartość –1):

```
bt81x.touch_loop(((-1, pressed), ))
```

Rysunek 4. Pojedynczy wiersz przycisków utworzony z użyciem *bt81x.add_keys()*

Listing 4. Funkcja wyświetlająca interfejs użytkownika z klawiaturą numeryczną

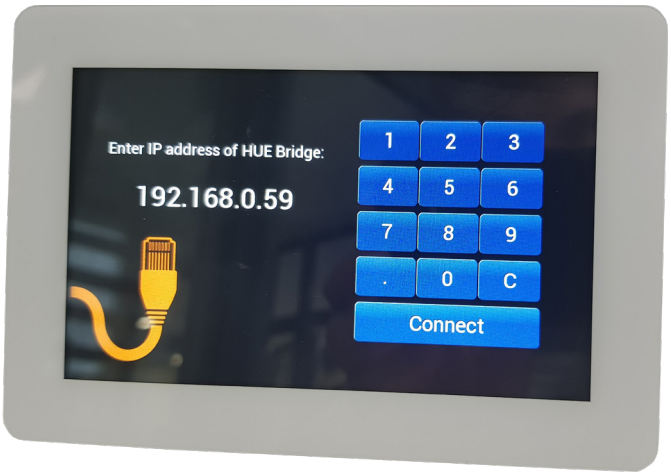
```
def showAddrScreen(ip):

    # start
    bt81x.dl_start()
    bt81x.clear(1, 1, 1)

    # image
    image = bt81x.Bitmap(1, 0, (bt81x.ARGB4, 200 * 2), (bt81x.BILINEAR, bt81x.BORDER, bt81x.BORDER, 200, 200))
    image.prepare_draw()
    image.draw((0, 255), vertex_fmt=0)

    # text
    txt = bt81x.Text(225, 120, 29, bt81x.OPT_CENTERX | bt81x.OPT_CENTERY, "Enter IP address of HUE Bridge:", )
    bt81x.add_text(txt)
    txt.text = ip
    txt.x = 225
    txt.y = 195
    txt.font = 31
    bt81x.add_text(txt)

    # keys
    bt81x.track(450, 350, 280, 60, 0)
    bt81x.add_keys(450, 70, 280, 60, 30, 0, "123")
    bt81x.add_keys(450, 140, 280, 60, 30, 0, "456")
    bt81x.add_keys(450, 210, 280, 60, 30, 0, "789")
    bt81x.add_keys(450, 280, 280, 60, 30, 0, ".0C")

    # connect button
    btn = bt81x.Button(450, 350, 280, 60, 30, 0, "Connect")
    bt81x.tag(1)
    bt81x.add_button(btn)

    bt81x.display()
    bt81x.swap_and_empty()
```

Listing 5. Funkcja obsługi zdarzeń pochodzących z ekranu dotykowego

```
def pressed(tag, tracked, tp):

    # entering HUE IP
    if (screenLayout == 2):

        global hueIP
        global screenLayout

        # remove last character
        if (tag != 67):

            # max length of IP
        if len(hueIP) >= 15:
            return

        hueIP = hueIP + str(chr(tag))

    else:
        if len(hueIP) > 0:
            hueIP = hueIP[:-1]

        # connect -> go to next screen
        if (tag == 1):
            hueIP = hueIP[:-1]
            screenLayout = 3
```

Listing 6. Wywołanie funkcji showAddrScreen() realizowane w pętli

```
gui.loadImage('gui_addr_hue.png')
while True:

    # enter IP address of HUE gateway
    screenLayout = 2
    while (screenLayout == 2):
        gui.showAddrScreen(hueIP)

    # show spinner (looking for HUE)
    screenLayout = 3
    gui.showSpinner("Looking for HUE Bridge...")

    # check HUE availability
    status = hue.testConnection(hueIP)
    if (status):
        break;
```



Fotografia 3. Interfejs wprowadzania adresu IP dla urządzenia Philips Hue Bridge

Wróćmy do operacji przygotowywania interfejsu graficznego. Ekran wprowadzania adresu IP będzie składał się z 4-wierszowej klawiatury (cyfry 0–9, symbol kropki oraz przycisk umożliwiający skasowanie błędnie wprowadzonych wartości), przycisku *Connect* oraz prostej grafiki. Całość operacji związanych z przygotowaniem interfejsu została zrealizowana w postaci jednej funkcji `showAddrScreen()`, której kod znajduje się na **listingu 4**.

Poprzez zastosowanie metody `bt81x.tag()` w funkcji `showAddrScreen()` do przycisku *Connect* przypisano wartość identyfikatora `TAG` równą 1. Przyciski z klawiatury wygenerowanej za pomocą szeregu wywołań `bt81x.add_keys()` jako wartość pola `TAG` zwracają kody ASCII, odpowiadające oznaczeniom poszczególnych klawiszy. Obsługa wszystkich zdarzeń realizowana jest w funkcji zwrotnej `pressed()`, jak zdefiniowano to w wywołaniu `bt81x.touch_loop()`. Fragment funkcji `pressed()`, związany z obsługą klawiatury, pokazuje **listing 5**.

Wywołanie funkcji `showAddrScreen()` realizowane jest w pętli `while()` (w pliku *main.py*), do momentu wybrania przycisku *Connect* (proces ten jest powtarzany, jeśli wprowadzony adres IP urządzenia jest nieprawidłowy) (**listing 6**). Finalny efekt wywołania funkcji `showAddrScreen()` został pokazany na **fotografii 3**.

Listing 7. Wyświetlenie komunikatu z prośbą o autoryzację użytkownika

```
def showAuthScreen():

    bt81x.dl_start()
    bt81x.clear(1, 1, 1)

    image = bt81x.Bitmap(1, 0, (bt81x.ARGB4, 420 * 2), (bt81x.BILINEAR, bt81x.BORDER, bt81x.BORDER, 420, 480))
    image.prepare_draw()
    image.draw((0, 0), vertex_fmt=0)

    txt = bt81x.Text(590, 220, 28, bt81x.OPT_CENTERX | bt81x.OPT_CENTERY, "Press the push-link button of the Hue", )
    bt81x.add_text(txt)
    txt.text = "bridge you want to connect to"
    txt.x = 590
    txt.y = 260
    bt81x.add_text(txt)

    bt81x.display()
    bt81x.swap_and_empty()
```

Listing 8. Uruchomienie procesu autoryzacji użytkownika

```
gui.showSpinner("Creating new user...")
gui.loadImage('gui_auth_hue.png')
while True:

    # show authScreen
    screenLayout = 5
    gui.showAuthScreen()

    # check status
    username = hue.createUser(hueIP)
    if (username):
        break;
```

Po poprawnym wskazaniu adresu IP urządzenia *Philips Hue Bridge* panel kontrolny przechodzi do etapu autoryzacji (zagadnienia związane bezpośrednio z nawiązaniem połączenia, autoryzacją oraz protokołem komunikacji z *Philips Hue Bridge*, zostaną omówione w dalszej części artykułu). W tym miejscu komunikat wyświetlany dla użytkownika ograniczy się wyłącznie do prostej grafiki oraz krótkiego komunikatu z prośbą o wciśnięcie przycisku umieszczonego na obudowie mostka *Philips* (jest to potwierdzenie, że użytkownik systemu ma fizyczny dostęp do urządzenia). Za wyświetlenie komunikatu oraz grafiki odpowiedzialna będzie funkcja `showAuthScreen()` z pliku *gui.py* (**listing 7**).

W pliku *main.py*, przed wyświetleniem komunikatu z prośbą o autoryzację, dodajmy krótki ekran pośredni (z zastosowaniem uprzednio przygotowanej funkcji `showSpinner()`). W tym czasie do pamięci GRAM kontrolera BT81x wgrywamy plik graficzny *gui_auth_hue.png*, który jest potrzebny w konstrukcji interfejsu z funkcji `showAuthScreen()`. Wszystkie operacje wykonywane zostaną w pętli `while()`, do momentu prawidłowej autoryzacji użytkownika (funkcja `createUser()` zostanie omówiona w dalszej części artykułu) (**listing 8**).

Opuszczenie pętli `while()` z listingu 8 jest równoznaczne z prawidłowym zakończeniem procesu łączenia z *Philips Hue Bridge*. Przyszedł więc czas na wyświetlenie głównego ekranu sterującego, złożonego z przycisków *ON*/*OFF* dla dwóch żarówek, oraz elementów sterujących dla jednej „kolorowej" żarówki (**fotografia otwierająca**).

Główny panel kontrolny jest zdecydowanie bardziej rozbudowany od poprzednich ekranów aplikacji – każdorazowe dodanie 10 przycisków i 8 elementów tekstowych mocno skomplikowałoby główną funkcję programu, a ewentualne wprowadzanie korekt położeń poszczególnych elementów interfejsu byłoby bardzo uciążliwe. W tym celu wszystkie komponenty głównego interfejsu opisane zostały za pomocą tablicy JSON, a umieszczenie ich na liście pogrupowane w dwie pętle `for()` (osobną dla elementów typu *Button* oraz *Text*). Fragmenty opisów JSON pokazuje **listing 9**.

Listing 9. Opis elementów interfejsu użytkownika w postaci tablicy JSON

```
buttons = [
{
    "tag_id": 2,
    "text": "ON",
    "x_cord": 50,
    "y_cord": 200,
    "width": 170,
    "height": 50,
    "size": 30
},
{
    "tag_id": 3,
    "text": "OFF",
    "x_cord": 50,
    "y_cord": 300,
    "width": 170,
    "height": 50,
    "size": 30
},
/.../
{
    "tag_id": 11,
    "text": ">",
    "x_cord": 675,
    "y_cord": 350,
    "width": 50,
    "height": 50,
    "size": 30
},
]

labels = [
{
    "text": "Kitchen",
    "x_cord": 136,
    "y_cord": 140,
    "size": 31,
    "options": bt81x.OPT_CENTER
},
/.../
{
    "text": "Brightness",
    "x_cord": 622,
    "y_cord": 330,
    "size": 28,
    "options": bt81x.OPT_CENTER
},
]
```

Listing 10. Utworzenie interfejsu użytkownika na podstawie danych umieszczonych w tabeli JSON

```
def showMainMenu(saturation, hue, brightness):

    bt81x.dl_start()
    bt81x.clear(1, 1, 1)

    image = bt81x.Bitmap(1, 0, (bt81x.ARGB4, 800 * 2), (bt81x.BILINEAR, bt81x.BORDER, bt81x.BORDER, 800, 50))
    image.prepare_draw()
    image.draw((0, 0), vertex_fmt=0)

    btn = bt81x.Button(0, 0, 170, 70, 31, 0, "")
    for button in buttons:
        btn.text = button["text"]
        btn.font =  button["size"]
        btn.x = button["x_cord"]
        btn.y = button["y_cord"]
        btn.width =  button["width"]
        btn.height = button["height"]
        bt81x.track(btn.x, btn.y,  button["width"],  button["height"], button["tag_id"])
        bt81x.tag(button["tag_id"])
        bt81x.add_button(btn)

    txt = bt81x.Text(0, 0, 0, 30, "")
    for label in labels:
        txt.text = label["text"]
        txt.x = label["x_cord"]
        txt.y = label["y_cord"]
        txt.font = label["size"]
        txt.options = label["options"]
        bt81x.add_text(txt)

    /.../

    # brightness value label
    txt.text = str(int((brightness/240)*100)) + '%'
    txt.x = 622
    txt.y = 375
    txt.font = 30
    txt.options = bt81x.OPT_CENTER
    bt81x.add_text(txt)

    # display
    bt81x.display()
    bt81x.swap_and_empty()
```

Za parsowanie poszczególnych elementów opisu JSON, dodanie ich do listy oraz jej finalne wyświetlenie odpowiada funkcja `showMainMenu()`, umieszczona w pliku *gui.py* (**listing 10**).

Cykliczne wywołanie funkcji `showMainMenu()` umieszczone zostało w pliku *main.py*, w głównej pętli programu, która aktualizuje wartości prezentowane w interfejsie graficznym, sprawdza status zmiennych, reprezentujących stany sterowanych żarówek, oraz, w przypadku wystąpienia zmian (wynikających z obsługi panelu dotykowego i obsługi zdarzeń w funkcji zwrotnej `pressed()`), wysyła komunikaty do urządzenia Philips Hue Bridge (**listing 11**). Zaktualizowany fragment funkcji `pressed()` znajduje się na **listingu 12**.

Udało się do tej pory skutecznie pominąć wszystkie aspekty

Listing 11. Główna pętla programu sterowania systemem Philips Hue

```
while True:

 gui.showMainMenu(saturation_old_value, hue_old_value, brightness_old_value)

 if (bulb_1_new_value != bulb_1_old_value):
    hue.turnLight(hueIP, username ,"4", bulb_1_new_value)
    bulb_1_old_value = bulb_1_new_value

 if (bulb_2_new_value != bulb_2_old_value):
    hue.turnLight(hueIP, username ,"2", bulb_2_new_value)
    bulb_2_old_value = bulb_2_new_value

 if (saturation_new_value != saturation_old_value):
    hue.changeColor(hueIP,username,"3",True, saturation_new_value,brightness_new_value, hue_new_value)
    saturation_old_value = saturation_new_value

 if (hue_new_value != hue_old_value):
    hue.changeColor(hueIP,username,"3",True,saturation_new_value, brightness_new_value, hue_new_value)
    hue_old_value = hue_new_value

 if (brightness_new_value != brightness_old_value):
    hue.changeColor(hueIP,username,"3", True,saturation_new_value,brightness_new_value, hue_new_value)
    brightness_old_value = brightness_new_value
```

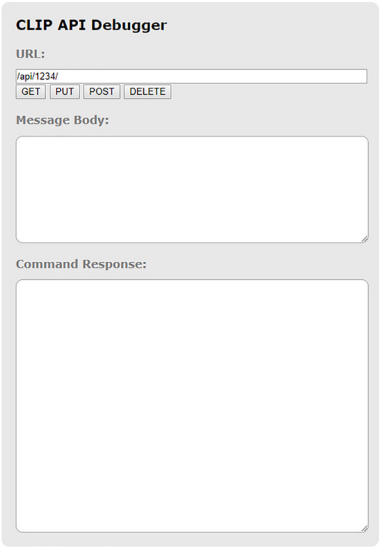
Listing 12. Zaktualizowania funkcja obsługi zdarzeń pochodzących z ekranu dotykowego

```
def pressed(tag, tracked, tp):

    # entering HUE IP
    if (screenLayout == 2):
    /.../
    # mainmenu screen
    elif (screenLayout == 7):

        global bulb_1_new_value
        global bulb_2_new_value

        # on/off switching
        if (tag == 2):
            bulb_1_new_value = True
        elif (tag == 3):
            bulb_1_new_value = False
        elif (tag == 4):
            bulb_2_new_value = True
        elif (tag == 5):
            bulb_2_new_value = False
        elif (tag == 6):
            if (saturation_new_value  > 0):
                saturation_new_value -= 24
        elif (tag == 7):
            if (hue_new_value  > 0):
                hue_new_value -= 6550
        /…/
        elif (tag == 11):
            if (brightness_new_value < 240):
                brightness_new_value += 24
```

związane z komunikacją z mostkiem Philips Hue. Czas na poświęcenie kilku chwil temu urządzeniu oraz sposobom jego komunikacji ze światem. W najpopularniejszej konfiguracji Philips Hue Bridge pełni funkcję pośrednika w komunikacji pomiędzy aplikacją mobilną a urządzeniami końcowymi. Twórcy systemu Philips Hue poszli jednak o krok dalej i zamiast tworzyć zamknięty system na linii aplikacja mobilna – mostek, zdecydowali się na przygotowanie otwartego i dobrze udokumentowanego API, umożliwiając tym samym programistom tworzenie własnych urządzeń sterujących i zarządzających pracą systemu. Komplet informacji związanych z tworzeniem własnych aplikacji został udostępniony na stronie producenta: https://bit.ly/3qPCh7d.

Kolejnym ułatwieniem dla programistów jest przygotowanie prostego interfejsu, umożliwiającego przetestowanie API i działania systemu bez tworzenia dedykowanej aplikacji. Interfejs ten – nazwany *CLIP API Debugger* – udostępniono pod adresem: *https://<bridge-ip-address>/debug/clip.html*

Adres mostka Philips można ustalić, wykorzystując protokół UPnP, poprzez stronę https://bit.ly/3mfuexi lub odczyt konfiguracji domowego routera. Wygląd okna interfejsu *CLIP API Debugger* pokazuje **rysunek 5**.



Rysunek 5. Interfejs panelu *CLIP API Debugger*

Komunikacja z mostkiem realizowana jest za pomocą wywołań HTTPS oraz REST API, dlatego okno interfejsu zostało podzielone na pole adresu *URL*, wyboru typu metody (*GET, PUT, POST* lub *DELETE*) oraz sekcji *BODY*, w której umieszczane są zapytania w formacie JSON. Zanim przejdziemy do realizacji zapytań sterujących pracą żarówek, niezbędne jest uprzednie utworzenie nowego użytkownika oraz jego autoryzacja. Aby utworzyć nowego użytkownika, wykonajmy następujące wywołanie:

URL: */api*
Body: *{"devicetype":"my_hue_app#IoT Display"}*
Method: *POST*

W odpowiedzi uzyskamy:

```
[
   {
   "error": {
      "type": 101,
      "address": "",
      "description": "link button not pressed"
   }
   }
]
```

Aby zwiększyć bezpieczeństwo systemu, niezbędna jest odpowiednia autoryzacja użytkownika. W otrzymanym komunikacie użytkownik jest proszony o wciśnięcie przycisku, umieszczonego

Listing 13. Utworzenie nowego użytkownika z wykorzystaniem REST API dla modułu Philips Hue Bridge

```
def createUser(ip):

    try:
        response = requests.post("http://" + ip + "/api", json={"devicetype": "my_hue_app#IoT Display"})

        js = json.loads(response.content)
        return js[0]["success"]["username"];

    except Exception as e:
        return None;
```

Listing 14. Oczekiwanie na poprawne utworzenie i autoryzację użytkownika

```
while True:

    # show authScreen
    screenLayout = 5
    gui.showAuthScreen()

    # check status
    username = hue.createUser(hueIP)
    if (username):
        # todo save user and password to flash
        break;
```

na obudowie urządzenia – ten zabieg ma na celu potwierdzenie, że osoba próbująca uzyskać dostęp do sterowania oświetleniem ma również fizyczny dostęp do mostka sterującego. Wciśnięcie przycisku oraz ponowne wysłanie powyższego zapytania POST zostanie potwierdzone poniższym komunikatem:

```
[
   {
   "success": {
      "username": "JxeQj8Xvi1zLDBnDz09w1aCjWA80rDx-
bQNYUlOCn"
   }
   }
]
```

Wartość pola *username* to nowa nazwa autoryzowanego użytkownika, która będzie wykorzystywana w dalszej komunikacji. Wróćmy na chwilę do kodu naszej aplikacji i zastanówmy się, jak zrealizować powyższe zadania z wykorzystaniem języka *Python* i zestawu bibliotek dostarczanych przez środowisko *Zerynth*. Z pomocą w tym zadaniu przychodzi moduł *requests*. Programistom *Python* nazwa ta jest zapewne dobrze znana – moduł pod kątem API jest inspirowany dość popularnym modułem o tej samej nazwie. Umożliwia szybką implementację obsługi HTTP oraz wywołań *GET, POST, PUT, DELETE*, w postaci pojedynczych funkcji. Pełna dokumentacja API dla modułu *requests* została udostępniona pod adresem: https://bit.ly/3qPFwM4.

Funkcję `createUser()`, bazującą na wywołaniach z modułu *request*, odpowiedzialną za utworzenie nowego użytkownika, pokazuje **listing 13**. Funkcja `createUser()` z listingu 13 wykonywana jest w pętli, aż do momentu odczytu wartości *success*, w odpowiedzi zwracanej przez mostek. Do głównej pętli programu funkcja ta zwraca wartość odczytaną z pola *username*, która zostanie wykorzystana w kolejnych wywołaniach sterujących stanem żarówek (**listing 14**).

Uzyskanie nazwy autoryzowanego użytkownika umożliwia nam odczyt konfiguracji systemu, tj. listy podłączonych żarówek, ich stanu, numerów identyfikacyjnych, itp.:

URL: */api/<username>/lights*
Body: *[none]*
Method: *GET*

Fragment odpowiedzi systemu dla konfiguracji z trzema żarówkami pokazuje **listing 15.**

Aktualny stan pracy poszczególnych żarówek można sprawdzić poprzez dodanie w zapytaniu GET numeru adresu identyfikacyjnego urządzenia, np:

URL: */api/<username>/lights/**2***
Body: *[none]*
Method: *GET*

Zmiana stanu standardowej żarówki realizowana jest poprzez wywołanie metody PUT:

URL: */api/<username>/lights/2/state*
Body: *{"on":false}*
Method: *PUT*

Zmiana stanu kolorowej żarówki wymaga od użytkownika określenia nie tylko

Listing 15. Fragment odpowiedzi systemu dla konfiguracji z trzema żarówkami

```
{
    "2": {
        "state": {
            "on": false,
            "bri": 254,
            "alert": "select",
            "mode": "homeautomation",
            "reachable": false
        },
        /.../
        "type": "Dimmable light",
        "name": "Hue white lamp 2",
        "modelid": "LWB010",
        "manufacturername": "Philips",
        "productname": "Hue white lamp",
        /.../
    },
    "3": {
        "state": {
            "on": false,
            "bri": 24,
            "hue": 45850,
            "sat": 240,
            "effect": "none",
            "xy": [
    0.1666,
    0.1016
            ],
            "ct": 153,
            "alert": "select",
            /.../
        },
        "swupdate": {
            "state": "readytoinstall",
            "lastinstall": "2019-07-04T12:22:06"
        },
        "type": "Extended color light",
        "name": "Hue color lamp 1",
        "modelid": "LCT015",
        "manufacturername": "Philips",
        "productname": "Hue color lamp",
        },
        /.../
    },
    "4": {
        "state": {
            "on": false,
            "bri": 254,
            "alert": "select",
            "mode": "homeautomation",
            "reachable": false
        },
        /.../
        "type": "Dimmable light",
        "name": "Hue white lamp 3",
        "modelid": "LWB010",
        "manufacturername": "Philips",
        "productname": "Hue white lamp",
        /.../
    }
}
```

jej nowego stanu pracy, ale również określenia wartości koloru, jasności oraz nasycenia:

URL: */api/<username>/lights/2/state*
Body: *{"on":true, "sat":254, "bri":254,"hue":10000}*
Method: *PUT*

Na bazie powyższego API, z użyciem modułu *requests*, przygotowano funkcje turnLight() oraz changeColor() (**listing 16**).

Pełny kod źródłowy aplikacji został udostępniony pod adresem: *http://bit.ly/3inwPok*.

Łukasz Skalski
contact@lukasz-skalski.com

Listing 16. Funkcje sterujące stanem żarówek Philips Hue

```
def turnLight(ip,user,num,state):

    try:
        addr = "http://" + ip + "/api/" + user + "/lights/" + num + "/state"
        requests.put(addr, json={"on":state})
    except Exception as e:
        return None;


def changeColor(ip,user,num,state, sat, bri, hue):

    try:
        addr = "http://" + ip + "/api/" + user + "/lights/" + num + "/state"
        requests.put(addr, json={"on":state, "sat":sat, "bri":bri, "hue":hue})
    except Exception as e:
        return None;
```