

# Zastosowania systemów wbudowanych



Politechnika  
Wrocławska

Miłosz Białczak  
Mateusz Gniewkowski  
Beata Szeląg

Prowadzący: dr inż. Marek Woda

07.12.17

# Spis treści

<b>1 Wstęp</b>	<b>3</b>
<b>2 Założenia projektowe</b>	<b>4</b>
<b>3 Technologie</b>	<b>5</b>
<b>4 Projekt Systemu</b>	<b>6</b>
4.1 Projekt systemu . . . . .	6
4.2 Raspberry Pi - Google Assistant . . . . .	7
4.2.1 Spis urządzeń . . . . .	9
4.2.2 Schemat połączeń . . . . .	11
4.3 ESP8266 . . . . .	12
4.3.1 Spis urządzeń . . . . .	14
4.3.2 Schemat połączeń . . . . .	15
4.4 Serwer . . . . .	15
4.5 Aplikacja webowa . . . . .	17
<b>5 Realizacja</b>	<b>19</b>
5.1 Raspberry Pi - Google Assistant . . . . .	19
5.1.1 Projekt fizyczny . . . . .	19
5.1.2 Instalacja Google Assistant . . . . .	20
5.1.3 Opis pliku wejściowego . . . . .	20
5.1.4 Własne komendy . . . . .	23
5.2 ESP8266 . . . . .	26
5.2.1 Projekt fizyczny . . . . .	26
5.2.2 Pierwsze użycie płytki i konfiguracja IDE . . . . .	26
5.2.3 Opis działania programu . . . . .	26
5.3 Serwer . . . . .	28

5.4 Aplikacja webowa . . . . .	29
<b>6 Uwagi i wnioski</b>	<b>31</b>

# Rozdział 1

## Wstęp

Nowoczesne rozwiązania technologiczne pozwalają domowym użytkownikom na budowę systemów wbudowanych dostosowanych do ich potrzeb. Platformy takie jak *Raspberry PI* czy *Arduino*, w połączeniu z wieloma czujnikami dostępnymi na rynku, dają możliwości, które jeszcze kilka lat temu były niewyobrażalne. Projekt, którego dotyczy niniejsza dokumentacja, jest przykładem systemu, który został zrealizowany właśnie dzięki tego typu rozwiązaniom. Będzie on umożliwiał rozmowę ze sztuczną inteligencją udostępnianą przez *Google* (*Google Assistant*). Pozwala ona między innymi na sprawdzenie pogody, swojej skrzynki mailowej, kalendarza i innych usług udostępnianych przez *Google*. Oprócz tego możemy zadać jej właściwe każde pytanie.

System zostanie rozbudowany o możliwość sprawdzenia stanu fizycznej skrzynki pocztowej. Ma ono następować poprzez stronę internetową lub komendę głosową, na którą urządzenie odpowie nam odpowiednim komunikatem. W tym celu zostanie stworzony serwer aplikacji, serwer webowy oraz osobny podsystem oparty na module *ESP8266* pozwalający na zbadanie aktualnego stanu skrzynki. Komunikacja urządzeń będzie wymagała sieci WiFi oraz dostępu do Internetu.

Ten dokument składa się z sześciu rozdziałów. Pierwszy z nich stanowi niewielki wstęp. W drugim zostaną omówione założenia projektowe. W trzecim można znaleźć informacje o wykorzystanych technologiach. W czwartym zostanie opisany projekt poszczególnych elementów systemu. W piątym znajdują się szczegółowe informacje dotyczące realizacji części projektowej. Ostatni rozdział zostanie poświęcony na uwagi i wnioski jakie wyniknęły podczas realizacji projektu.

## Rozdział 2

### Założenia projektowe

Poniżej przedstawiono listę wymagań systemu.

1. System musi pozwalać na werbalną komunikację z *Google Assistant*.
2. System musi pozwalać na sprawdzenie stanu fizycznej skrzynki pocztowej w sposób werbalny i poprzez stronę internetową.
3. Podsystem oparty na *ESP8266* musi mieć możliwość działania na baterijnym źródle zasilania.
4. Podsystem oparty na *ESP8266* musi w sposób rozsądny korzystać ze źródła zasilania.
5. Podsystem oparty na *ESP8266* powinien dawać możliwość ustalenia czasu, w którym dostaliśmy listy.
6. System powinien umożliwiać implementowanie własnych komend.
7. System powinien umożliwiać odtwarzanie muzyki z portalu *YouTube*.

# Rozdział 3

## Technologie

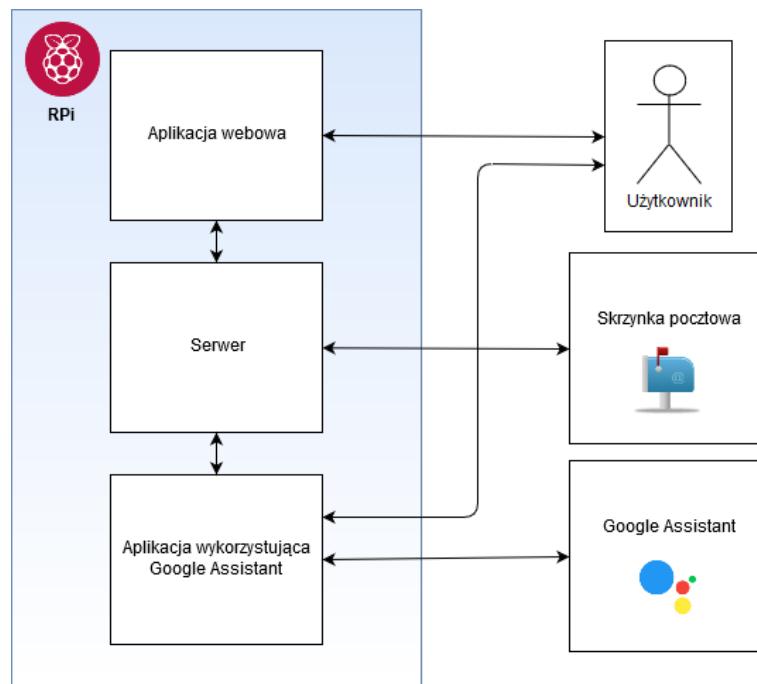
Niżej została przedstawiona lista technologii, które zostaną wykorzystane w systemie. System będzie zbudowany z dwóch urządzeń - *Raspberry Pi* i *ESP8266* oraz szeregu peryferiów do nich podłączonych (ich dokładną listę można znaleźć w kolejnym podrozdziale).

1. **Python** - zarówno serwer jak i aplikacja komunikująca się z *Google Assistant* zostanie napisane w języku *Python*
2. **Flask** - *Flask* jest framework'iem pozwalającym na budowę serwerów aplikacji typu *REST*
3. **C++** - do zaprogramowania modułu *ESP8266* zostanie użyty język *C++*
4. **HTML5 + CSS + JS + AngularJS** - niniejsze technologie zostaną wykorzystane do napisania aplikacji webowej.

# Rozdział 4

## Projekt Systemu

### 4.1 Projekt systemu



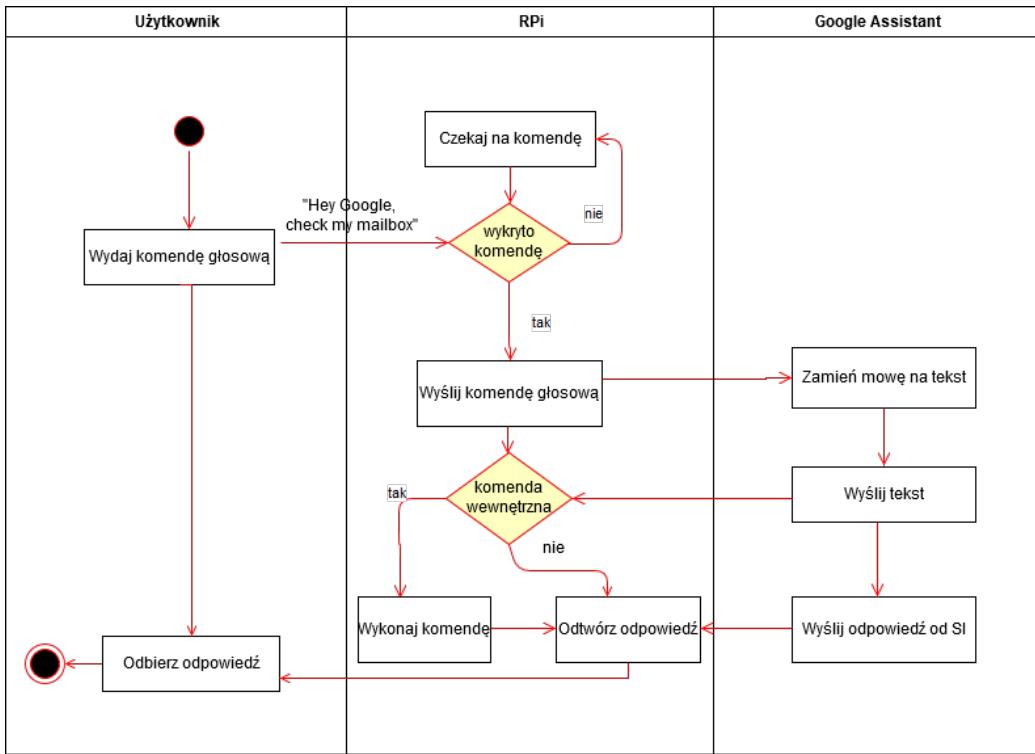
Rysunek 4.1: Schemat działania systemu

Rysunek 4.1 przedstawia ogólny schemat opisywanego projektu. Jego kolejne części są opisane w poniższych sekcjach:

- część RPI + Google Assistant w sekcji *Raspberry Pi - Google Assistant*
- część z ESP8266 w sekcji *ESP8266*
- serwer w sekcji *Serwer*
- aplikacja webowa w sekcji *Aplikacja webowa*

## 4.2 Raspberry Pi - Google Assistant

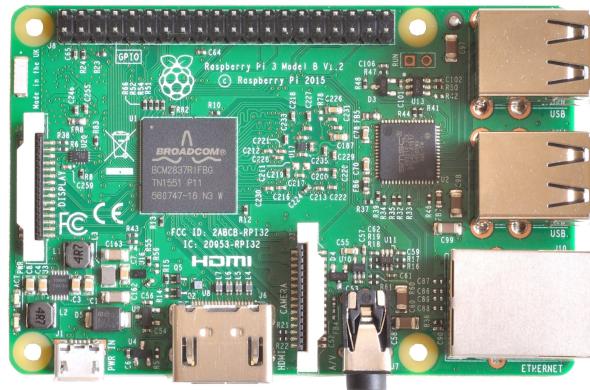
Głównym elementem naszego projektu jest mikrokomputer Raspberry Pi 3 w wersji B. Zostanie na nim zainstalowany serwer aplikacji webowej (omówiony w punkcie 4.4), aplikacja webowa (punkt 4.5) i aplikacja wykorzystująca Google Assistant. W niniejszej części dokumentu zostanie omówiony projekt bezpośredniego połączenia Raspberry Pi z urządzeniami zewnętrznymi (takimi jak głośnik, mikrofon) stanowiącymi razem pewną integralną całość w założeniu mającą znaleźć się w jednej obudowie; oraz projekt działania aplikacji komunikującej się z Google Assistant. Zdalny moduł wykorzystujący ESP8266 zostanie omówiony w innej części tego dokumentu.



Rysunek 4.2: Schemat działania podsystemu

Powyższy rysunek (4.2) przedstawia schemat działania aplikacji komunikującej się z Google Assistant. Po wydaniu komendy głosowej przez użytkownika jest ona następnie wysyłana do Google Assistant'a. Google Assistant zamienia mowę na tekst, który następnie jest wysyłany do naszego urządzenia. W zależności czy tekstowa wersja komendy została rozpoznana jako komenda wewnętrzna (zdefiniowana przez użytkownika, np. prośba o sprawdzenie skrzynki pocztowej lub o odtworzenie utworu z portalu YouTube) urządzenie wykoną ją i odtworzy nagraną wcześniej odpowiedź (ewentualny słowny komunikat od Google zostanie zignorowany). W przeciwnym przypadku urządzenie będzie czekać aż sztuczna inteligencja dostarczy słowną odpowiedź, a następnie odtworzy ją użytkownikowi.

#### 4.2.1 Spis urządzeń



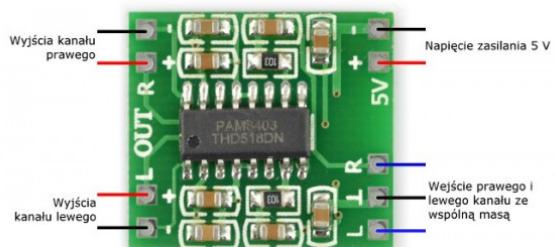
Rysunek 4.3: Raspberry Pi 3 w wersji B



Rysunek 4.4: Zewnętrzna karta dźwiękowa Virtual 7.1 Channel USB



Rysunek 4.5: Głośnik 3W 8Ohm 40x88mm



Rysunek 4.6: Wzmacniacz audio stereo PAM8403 5V 3W - dwukanałowy

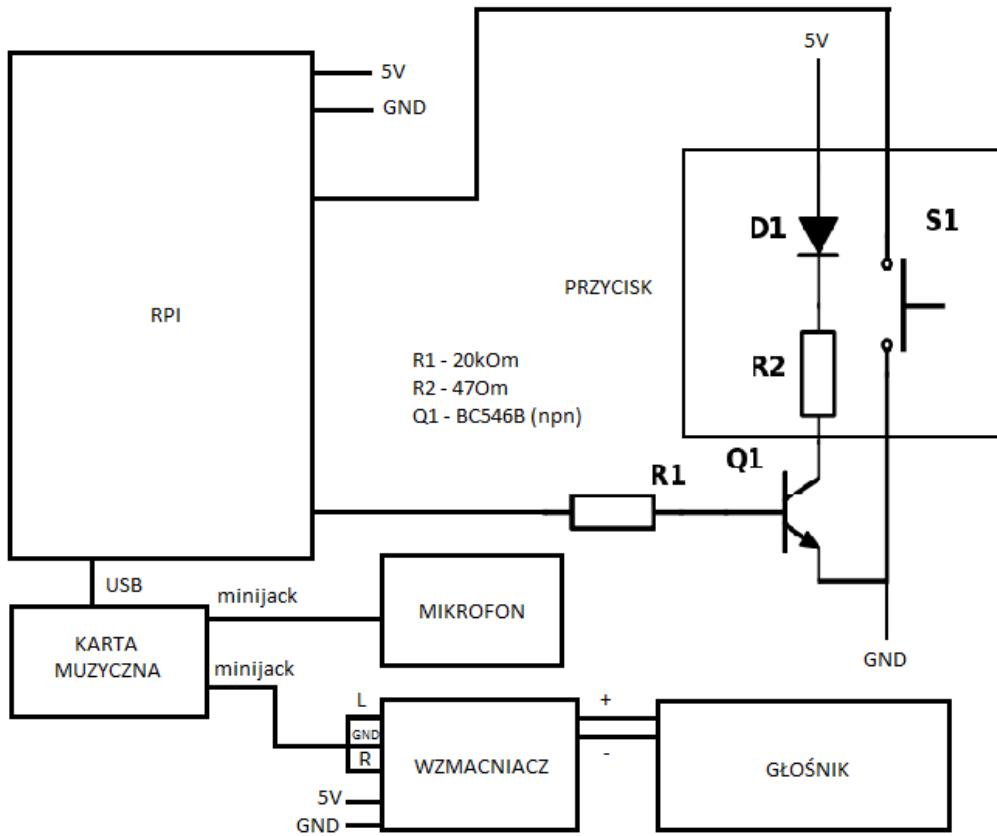


Rysunek 4.7: Rzeczykis Arcade Push Button niebieski z podświetleniem

Dodatkowo zostanie wykorzystany mikrofon wykorzystujący złącze typu *mini jack*.

#### 4.2.2 Schemat połączeń

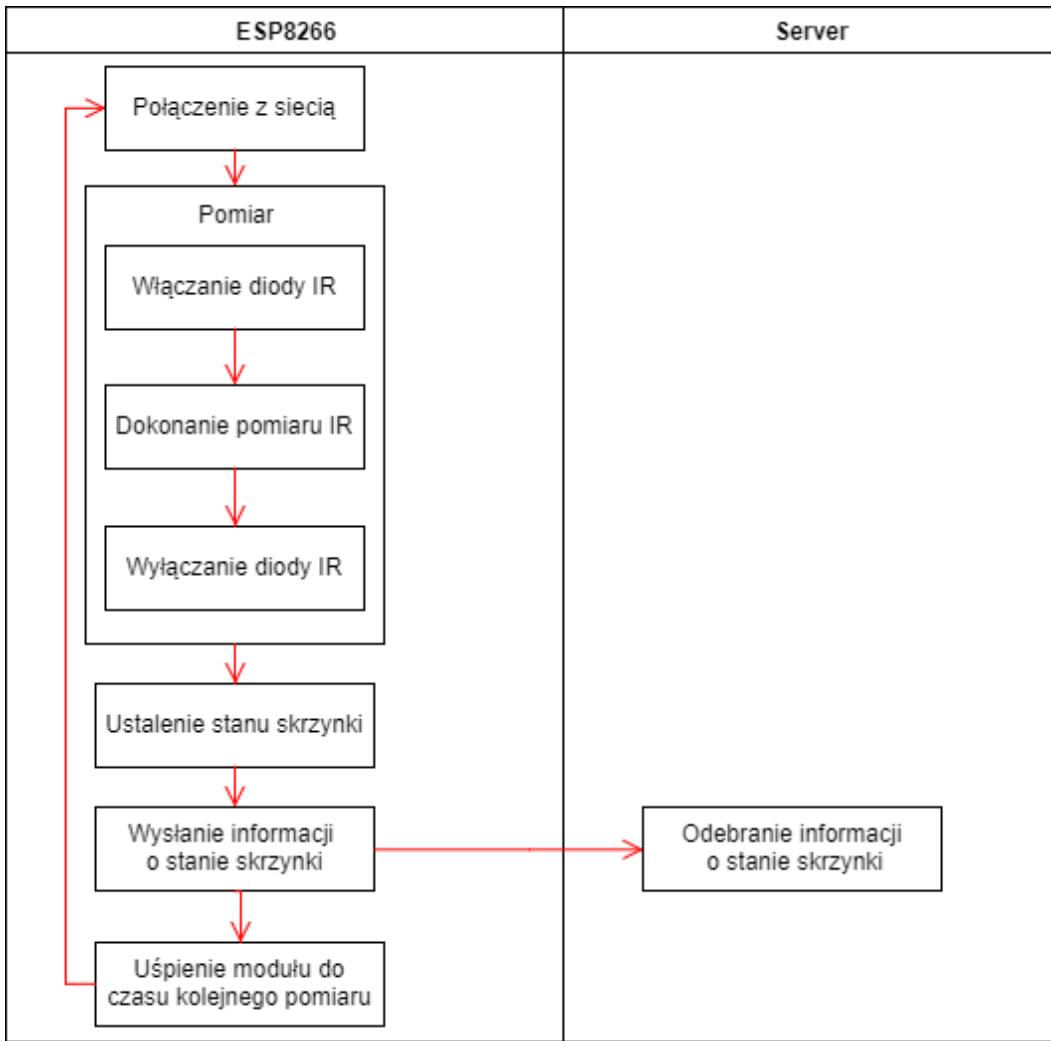
Na poniższym schemacie zaprezentowano układ połączeń wyżej wymienionych elementów. Rezystancje oporników mogą się różnić ze względu na wykorzystany tranzystor i rodzaj diody świecącej. W większości przypadków układ typu „klucz npn” nie będzie potrzebny, gdyż napięcie przewodzenia diody może okazać na tyle niskie, że będzie możliwe zasilenie jej z portu GPIO (niestety nie było tak w naszym przypadku).



Rysunek 4.8: Schemat połączeń peryferiów do Raspberry Pi

### 4.3 ESP8266

Drugim kluczowym elementem naszego projektu jest układ wykorzystujący moduł *ESP8266*. Zostanie do niego podłączony nadajnik IR, odbiornik IR i bateria. Gdy odczyt sygnału, nadawanego przez pierwsze z urządzeń, nie będzie możliwy, znaczy to, że na drodze wiązki znajduje się jakiś przeszkoda - w założeniach ma to być list. Moduł zostanie oprogramowany tak aby, co jakiś ustalony czas, wysyłać do serwera informacje o odczycie. W finalnej wersji układ ma zostać zamontowany w skrzynce.

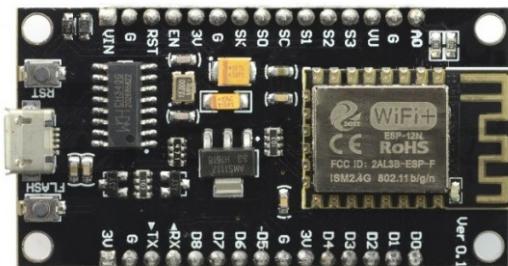


Rysunek 4.9: Diagram stanów modułu ESP8266

Powyższy diagram 4.9 przedstawia schemat działania modułu ESP w projekcie. Po uruchomieniu moduł dokonuje połączenia siecią WiFi. Po potwierdzeniu łączności, moduł rozpoczyna pomiar zajętości skrzynki. Pierwszym krokiem jest włączenie diody IR, następnie pomiar odbioru sygnału IR na czujniku i wyłączenie diody IR. Kolejno jest ustalany stan skrzynki w zależności od uzyskanego wyniku pomiarów. Następnie informacja o stanie skrzynki musi zostać wysłana do serwera. W ostatnim kroku moduł usypia, by po ustalony czasie powrócić do działania rozpoczętym od kroku łączenia.

z siecią.

#### 4.3.1 Spis urządzeń



Rysunek 4.10: ESP8266 z ModeMCU v3



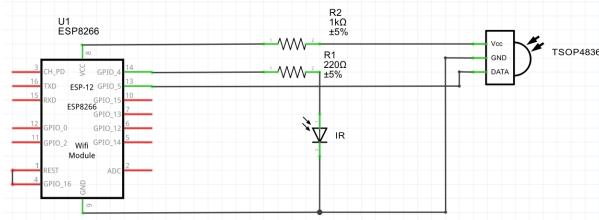
Rysunek 4.11: Odbiornik TSOP4836



Rysunek 4.12: Nadajnik IR LIRED5C

#### 4.3.2 Schemat połączeń

Poniższy schemat przedstawia układ połączeń wyżej wymienionych elementów. Warto zwrócić uwagę na połączenie pinu GPIO\_16 z pinem REST. Pozwoli to przełączac moduł ESP w tryb głębokiego snu - pozwoli to na oszczędzanie baterii.



## 4.4 Serwer

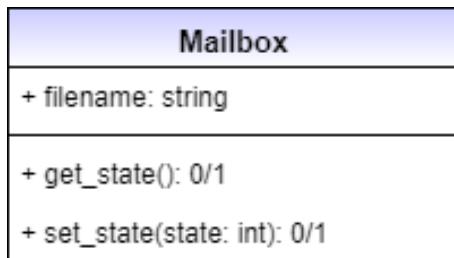
Następną częścią projektu jest serwer oparty o technologię *Flask*. Pozwala on na komunikację pomiędzy *ESP8266*, *Google Assistantem* oraz aplikacją webową.

Serwer jest uruchamiany na *Raspberry PI*, na porcie 5000 i udostępnia klientom dwa *endpointy*. Jeden z nich pozwala na pobranie aktualnego stanu skrzynki pocztowej, drugi - na jego zmianę.

REST API udostępniane przez serwer jest przedstawione w tabeli 4.1.

Tablica 4.1: Serwer REST API

Skrzynka pocztowa /state			
Endpoint	Request	Opis	Dodatkowo
/state	GET	Pobranie informacji o stanie skrzynki	Rezultat: 0 - pusta skrzynka, 1 - pełna
/state/[state]	POST	Zmiana stanu na [state]	state może przyjmować wartości: 0 lub 1



Rysunek 4.13: Klasa Mailbox

Serwer korzysta z klasy Mailbox, która udostępnia dwie metody: do pobrania stanu skrzynki i do ustawienia stanu skrzynki. Więc o obu metodach można znaleźć w rozdziale 5 *Realizacja*. Diagram klasy jest widoczny na rysunku 4.4

Stan skrzynki jest zapisywany w pliku, w postaci:

- 0 - gdy skrzynka jest pusta
- 1 - gdy w skrzynce mamy jakieś wiadomości

W przypadku, jeśli na *Raspberry PI* nie ma wymaganych przez serwer zależności (Flask w wersji 0.10.1) należy wywołać:

---

`pip install -r requirements.txt`

---

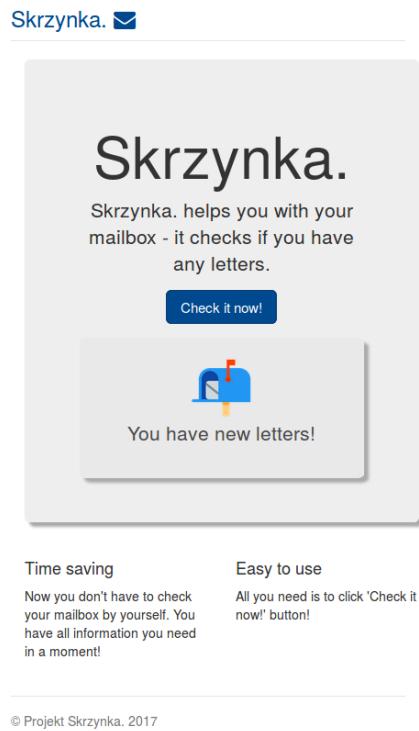
Uruchamianie serwera następuje przez wywołanie:

---

`python run.py`

---

## 4.5 Aplikacja webowa



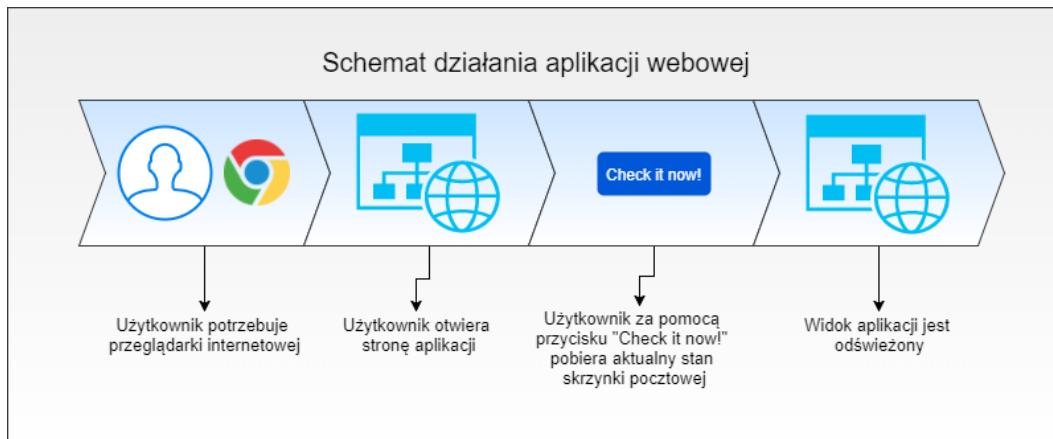
Rysunek 4.14: Aplikacja webowa

Aplikacja webowa przedstawiona na rysunku 4.5 pozwala na sprawdzenie aktualnego stanu skrzynki pocztowej. Logika aplikacji została stworzona w języku *Java Script* wraz z *AngularJS*, natomiast widok to *HTML5* wraz z arkuszami stylów (*CSS*).

Składa się z trzech głównych części:

- widoku - który określa elementy, jakie są obecne na stronie
- kontrolera - który odpowiada za logikę w aplikacji (wykorzystuje serwis do wysyłania zapytań do serwera)
- serwisu - który wysyła zapytanie **GET /state** na adres serwera

Na stronie dostępny jest przycisk **Check it now!** (ang. *Wypróbowuj już teraz!*), który wywołuje metodę kontrolera odpowiedzialną za wykonanie pytania do serwera przez serwis.



Rysunek 4.15: Schemat działania aplikacji webowej

Rysunek przedstawiony wyżej 4.5 przedstawia schemat działania aplikacji webowej.

Uruchomienie serwera *node.js* z parametrem wskazującym na ścieżkę do pliku **server.js**:

---

```
node server.js
```

---

W przypadku, jeśli na *Raspberry PI* nie ma wymaganych przez aplikację zależności należy wywołać:

---

```
bower install
npm install connect
npm install serve-static
```

---

i później uruchomić serwer *node.js*.

# Rozdział 5

## Realizacja

### 5.1 Raspberry Pi - Google Assistant

#### 5.1.1 Projekt fizyczny

Na podstawie schematu (dostępnego w poprzednim rozdziale) został zbudowany prototyp urządzenia pełniącego rolę asystenta.



Rysunek 5.1: Urządzenie wykorzystujące RPi



Rysunek 5.2: Urządzenie wykorzystujące RPi - wnętrze

### 5.1.2 Instalacja Google Assistant

Podstawowa konfiguracja Google Assistant jest bardzo dobrze opisana na stronie <https://developers.google.com/assistant/sdk/overview> (zakładka „Python”).

### 5.1.3 Opis pliku wejściowego

Zakładając projekt z wykorzystującym Google Assistant SDK dostajemy skrypt realizujący podstawową funkcjonalność asystenta:

---

```
#!/usr/bin/env python

# Copyright (C) 2017 Google Inc.
#
# Licensed under the Apache License, Version 2.0 (the 'License');
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an 'AS IS' BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

```

from __future__ import print_function

import argparse
import os.path
import json

import google.oauth2.credentials

from google.assistant.library import Assistant
from google.assistant.library.event import EventType
from google.assistant.library.file_helpers import existing_file


def process_event(event):
    '''Pretty prints events.
    Prints all events that occur with two spaces between each new
    conversation and a single space between turns of a conversation.
    Args:
        event(Event): The current event to process.
    '''
    if event.type == EventType.ON_CONVERSATION_TURN_STARTED:
        print()

    print(event)

    if (event.type == EventType.ON_CONVERSATION_TURN_FINISHED and
        event.args and not event.args['with_follow_on_turn']):
        print()


def main():
    parser = argparse.ArgumentParser(
        formatter_class=argparse.RawTextHelpFormatter)
    parser.add_argument('--credentials', type=existing_file,
                        metavar='OAUTH2_CREDENTIALS.FILE',
                        default=os.path.join(
                            os.path.expanduser('~/.config'),
                            'google-oauthlib-tool',
                            'credentials.json'
                        ),
                        help='Path to store and read OAuth2 credentials')
    args = parser.parse_args()
    with open(args.credentials, 'r') as f:
        credentials = google.oauth2.credentials.Credentials(token=None,

```

```

        **json.load(f))

with Assistant(credentials) as assistant:
    for event in assistant.start():
        process_event(event)

if __name__ == '__main__':
    main()

```

---

Powyższy kod tworzy obiekt asystenta (wykorzystując w tym procesie dane uwierzytelniające), po czym w pętli zaczyna przetwarzanie zdarzeń (przykładem zdarzenia jest początek konwersacji wywoływany słowami „Hey Google”). Zmieniając implementację metody process\_event możemy wpływać na zachowanie asystenta.

---

```

def process_event(cp, event, assistant):
    '''Pretty prints events.
    Prints all events that occur with two spaces between each new
    conversation and a single space between turns of a conversation.
    Args:
        event(Event): The current event to process.
    '''
    if event.type == EventType.ON_CONVERSATION_TURN_STARTED:
        print()
        gpio.output(22, True)

    print(event)

    if event.type == EventType.ON_RECOGNIZING_SPEECH_FINISHED:
        try:
            if cp.read_command(event.args['text']):
                assistant.stop_conversation()
        except ValueError as e:
            print(e)

    if (event.type == EventType.ON_CONVERSATION_TURN_FINISHED and
        event.args and not event.args['with_follow_on_turn']):
        print()
        gpio.output(22, False)

```

---

Powyższy kod przedstawia zmodyfikowaną wersję metody process\_event. Najważniejszym elementem jest przechwycenie zdarzenia „ON RECOGNIZING SPEECH FINISHED”, w którym można znaleźć nasze słowa zamienione na tekst (ang. Speech To Text). Dzięki temu, odpowiednio przetwarzan-

jąc zdarzenie, możemy zaimplementować własne reakcje systemu. W tym celu stworzyliśmy klasę (opisaną dokładniej w dalszej części dokumentu) „CommandProcessor” (cp). Przyjmuje ona treść naszych słów i szuka odpowiedniej komendy do wywołania - w przypadku znalezienia takowej, „rozmowa” z asystentem jest przerywana (nie usłyszmy odpowiedzi od sztucznej inteligencji). W powyższej metodzie dopisaliśmy również reakcje na zdarzenia „ON CONVERSATION TURN STARTED” i „ON CONVERSATION TURN FINISHED” jest to odpowiednio zapalanie i gaszenie diody.

#### 5.1.4 Własne komendy

W celu umożliwienia stworzenia własnych komend powstał skrypt „commands\_processor.py”:

---

```
from inspect import signature
import RPi.GPIO as gpio
import time
import subprocess
from subprocess import CalledProcessError, check_output
import _thread
import os

#additional functions
def is_process_alive(process_name):
    try:
        check_output(["pgrep", process_name])
        output = 0
    except subprocess.CalledProcessError as er:
        output = er.returncode

    if output == 0:
        return True
    else:
        return False

def play_sound(filename):
    os.system("aplay sounds/"+filename)

#####
```

```

#custom commands:

def print_hello(text):
    print("hello")

def print_bye(text):
    print("bye")

def print_text(text):
    print(text)

def play_yt(text):

    if is_process_alive("vlc"):
        return

    play_sound("im_on_it.wav")

    playshell = subprocess.Popen(["/usr/local/bin/mpsyt", ""],
                                stdin=subprocess.PIPE, stdout=subprocess.PIPE)

    playshell.stdin.write(bytes('/' + text + '\n\n', 'utf-8'))
    playshell.stdin.flush()

    gpio.setmode(gpio.BCM)
    gpio.setup(23, gpio.IN, pull_up_down=gpio.PUD_UP)

    print("STARTING VLC...")
    while(not is_process_alive("vlc")):
        time.sleep(1)

    print("MUSIC IS PLAYING")
    while(gpio.input(23) and is_process_alive("vlc")):
        time.sleep(1)

    subprocess.Popen(["/usr/bin/pkill", "vlc"], stdin=subprocess.PIPE)
    playshell.kill()

#####
#custom commands:

class CommandProcessor(object):

```

```

commands = [
    ("hello", print_hello),
    ("bye", print_bye),
    ("print", print_text),
    ("play", play_yt)
]

def read_command(self, text):
    try:
        command = next(x for x in self.commands if text.startswith(x[0]))
    except StopIteration as err:
        return False

    text = text.replace(command[0], "", 1)
    text = text.strip()
    method = command[1]

    sig = signature(method)
    length = len(sig.parameters)
    if (length == 0):
        return _thread.start_new_thread(method, ())
    elif (length == 1):
        _thread.start_new_thread(method, (text,))
    else:
        raise ValueError("EXCEPTION: Trying to call a function
                        with more than one parameter")

return True

```

---

Klasa „CommandProcessor” została wspomniana w poprzednim podróziale - przetwarza ona komendy użytkownika decydując o tym, czy były one przeznaczone dla asystenta, czy nie. Zmienna „commands” jest mapą słów kluczowych w komendach na funkcje, które w ramach danej komendy mają być wywołane. Metoda „read\_command” sprawdza czy pierwsze słowa wy powiedzi użytkownika znajdują się w zmiennej „commands”. Jeżeli tak, to w osobnym wątku wywoływana jest odpowiednia funkcja (przykłady funkcji można znaleźć wyżej w kodzie) i zwracana jest wartość True, jeżeli nie to zwracana jest wartość False.

## 5.2 ESP8266

### 5.2.1 Projekt fizyczny

Na podstawie schematu (dostępnego w poprzednim rozdziale) został zbudowany prototyp zbierający dane o stanie zapełnienia skrzynki.

### 5.2.2 Pierwsze użycie płytka i konfiguracja IDE

Przed pierwszym użyciem płytki zalecane jest zaktualizowanie jej firmware'u. Instrukcja wykonania tego znajduje się na stronie <http://hobbyspace.pl/nodemcu-jak-wgrac-firmware/>.

Do oprogramowania płytki zostało wykorzystane Arduino IDE (v.1.8.5). W celu uzyskania możliwości współpracy z wcześniej wspomnianym IDE konieczne było dodanie nowego adresu URL dla menadżera płytka w zakładce preferencje - dzięki temu możliwe było pobranie odpowiednich bibliotek obsługujących moduł *ESP*. Oto wymagany tam link [http://arduino.esp8266.com/stable/package\\_esp8266com\\_index.json](http://arduino.esp8266.com/stable/package_esp8266com_index.json).

### 5.2.3 Opis działania programu

Poniżej przedstawiono program, który ma zostać uruchomiony na płytce *ESP*.

---

```
#include <ESP8266WiFi.h>
#include <ESP8266HTTPClient.h>

const char* ssid = "*****"; // type your ssid
const char* password = "*****"; // type your password
WiFiClient client;

int IRledPin = 4; // Dioda IR nadawcz
int TSOPPin = 5; // Dioda TSOP odbiorcza

// Start setup
void setup() {
    pinMode(TSOPPin, INPUT);
    pinMode(IRledPin, OUTPUT);
    digitalWrite(IRledPin, LOW);
    WiFiConnect();
}
```

```

//Main loop
void loop() {
    measurement();
    ESP.deepSleep(30e6); // 30e6 is 30 microseconds
}

//////
//IR//
//////

// Measurement IR
void measurement()
{
    digitalWrite(IRledPin, HIGH);
    //waiting 10 seconds for IR signal
    int result = pulseIn(TSOPPin,LOW,10000000);
    digitalWrite(IRledPin, LOW);
    reaction(result);
}

// Reaction if something in box
void reaction(int result){
    if(result == 0){
        SendPOST("1");
    }
    else{
        SendPOST("0");
    }
}
///////
//WiFi//
///////

// Connect to WiFi network
void WiFiConnect() {
    WiFi.mode(WIFI_AP_STA);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
    }
}
// Send POST request to URL
void SendPOST(String URL)
{
    String URL = "http://192.168.137.63:5000/state/" + Data;
    HTTPClient http;
}

```

---

```

//Specify request destination
http.begin(URL);
//Specify content-type header
http.addHeader("Content-Type", "text/plain");
//Send the request (returning code
int httpCode = http.POST(Data);
//Get the response payload
String payload = http.getString();
http.end();
}

```

---

W głównej pętli programu wywołujemy dwie funkcje: *measurement* i *deepSleep*. Pierwsza z nich jest funkcją własną odpowiedzialną za pomiar. Uruchamia ona nadajnik i odbiornik podczerwieni, po czym, na podstawie zebranych danych, ustala, czy w skrzynce znajdują się listy (list powinien przecinać wiązkę podczerwieni). Następnie informacja o stanie skrzynki jest wysyłana do serwera (funkcja *reaction*). Druga metoda jest metodą systemową, która (jak zostało to opisane w poprzednim rozdziale) pozwala uśpić płytke na określony czas.

### 5.3 Serwer

Serwer uruchamia się za pomocą skryptu **run.py**, który wywołuje metodę **run** z frameworku *Flask* i wygląda następująco:

---

```

if __name__ == "__main__":
    app.run(debug=True, host='<ip>', processes=5)

```

---

Zwiększoną liczbą procesów była niezbędna, aby serwer poprawnie obsługiwał kolejne, nowo przychodzące połączenia od *ESP8266*.

Następnie określono *endpointy* zwracające stan skrzynki (**GET /state**) oraz zmieniające go (**POST /state/[state]**):

---

```

@app.route("/state")
def get_state():
    response = Response(mailbox.get_state())
    response.headers['Access-Control-Allow-Origin'] = '*'
    return response

@app.route("/state/<state>", methods=['POST'])
def state_changed(state):

```

```
mailbox.set_state(state)
return mailbox.get_state()
```

---

Mailbox jest klasą, która pozwala na odczyt i zapis do pliku przechowującego informacje o skrzynce pocztowej.

Dodanie nagłówka **Access-Control-Allow-Origin** do zapytania pobierającego stan, było niezbędne do tego, aby aplikacja webowa mogła bez przeszkołd pobierać dane. Jest to spowodowane **Cross Domain Policy**, które zwykle blokuje takie akcje.

## 5.4 Aplikacja webowa

Składa się z trzech głównych części:

- widoku - który określa elementy, jakie są obecne na stronie
- kontrolera - który odpowiada za logikę w aplikacji (wykorzystuje serwis do wysyłania zapytań do serwera)
- serwisu - który wysyła zapytanie **GET /state** na adres serwera

Wysyłanie zapytania wygląda następująco:

---

```
function getMailboxStatus() {
    return $http.get("http://<server_ip>:5000/state")
        .then(handleResponse())
        .catch(handleError('An error occurred
            while getting user data by session.'));
}
```

---

Serwer zwraca 0 albo 1 i w zależności od tej wartości na stronie jest wyświetlany komunikat o dostępnych listach lub o ich braku (wraz ze stosownym rysunkiem).

Funkcja kontrolera, która jest wywoływana po naciśnięciu przycisku:

---

```
function check() {
    MainService.getMailboxStatus().then(function (response) {
        if (response.data == 1) {
            vm.message = "You have new letters!";
            vm.logo_path = "static/img/logo.png";
        } else {
            vm.message = "You don't have any letters...";
    })
}
```

---

```
        vm.logo_path = "static/img/logo2.png";
    }
})
}
```

---

- *MainService* - serwis w aplikacji webowej.
- *getMailboxStatus()* - wywołanie metody wysyłającej zapytanie do serwera

W ramach funkcji **check()** przedstawionej na powyższym listeningu, w zależności od odpowiedzi, wyświetlamy komunikat *You have new letters!* i rysunek skrzynki pocztowej pełnej listów lub *You don't have any letters...* z rysunkiem pustej skrzynki.

Ważną kwestią jest uruchomienie aplikacji webowej z wykorzystaniem serwera *node.js*. W tym celu został stworzony prosty skrypt **server.js**, który uruchamia aplikację na porcie 8080, w następujący sposób:

```
var connect = require('connect');
var serveStatic = require('serve-static');
connect().use(serveStatic(__dirname)).listen(8080, function(){
    console.log("Server running on 8080");
});
```

---

# Rozdział 6

## Uwagi i wnioski

W efekcie pracy nad projektem powstał działający prototyp systemu w pełni realizujący zadania przed nim stawiane. Dzięki możliwości definiowania własnych komend głosowych i dołączaniu kolejnych podsystemów, można rozbudować projekt do rozmiarów pozwalających na stworzenie np. inteligentnego domu. Jedną z niewielu wad, którą posiada system, jest brak funkcjonalności pozwalającej na zamianę pisma na mowę (ang. *Text to Speech* - wszelkie słowne odpowiedzi urządzenia są uprzednio nagrane i odtwarzane w razie potrzeby. Rozwiązaniem problemu mogłoby być zastosowanie osobnego narzędzia implementującego tę funkcję, jednak istnieje prawdopodobieństwo, że *Google Assistant* będzie posiadał taką możliwość w przyszłości. Kolejnym elementem jaki należy poprawić w projekcie jest stworzenie odpowiedniej obudowy dla głównego urządzenia - kartonowe pudełko jest mało estetyczne i podane na uszkodzenia.

Możliwości jakie dają współczesne technologie mogą być z powodzeniem wykorzystywane w wielu dziedzinach. Do stworzenia własnego systemu wbudowanego wystarcza podstawowa wiedza z zakresu programowania i elektroniki, co (wraz ze stosunkowo niskim kosztem podzespołów) pozwala na tworzenie domowym użytkownikom rozwiązań dostosowanych do ich potrzeb.