

Politechnika Poznańska
Wydział Informatyki
Instytut Informatyki

Praca dyplomowa inżynierska

SYSTEM KONTROLI BEZPIECZEŃSTWA – THE GUARD

Mateusz Bartos, 122437
Piotr Falkiewicz, 122563
Aleksandra Główczewska, 122494
Paweł Szudrowicz, 122445

Promotor
dr inż. Mariusz Nowak

Poznań, 2018 r.

Tutaj przychodzi karta pracy dyplomowej;
oryginał wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

Spis treści

1	Streszczenie	1
2	Wstęp	2
2.1	Cel i zakres pracy	2
2.2	Podział pracy	2
2.3	Struktura pracy	3
3	Architektura systemu	4
3.1	Schemat	4
3.2	Komunikacja	5
3.3	Bezpieczeństwo	5
4	Kosztorysy	7
4.1	Koszty elementów systemu	7
4.2	Koszty użytkowania	7
5	Zbieranie i przetwarzanie danych z czujników	9
5.1	Raspberry Pi	9
5.2	Czujniki	12
5.3	Obsługa wideo	15
6	Rozwiązania chmurowe	18
6.1	Microsoft Azure	18
6.2	Firebase	18
6.3	Aplikacja serwerowa	19
6.4	Baza danych	21
7	Aplikacje klienckie	23
7.1	Funkcje aplikacji	23
7.2	Aplikacja Android	26
7.3	Aplikacja iOS	28
7.4	Aplikacja internetowa	32
8	Testy funkcjonalne	36
9	Uwagi końcowe	39
	Literatura	40
	Spis rysunków	42

Rozdział 1

Streszczenie

System kontroli bezpieczeństwa – The Guard jest to system monitorowania zagrożeń w domu i ostrzegania właścicieli przed niebezpieczeństwem. Pozwala na transmisję obrazu w czasie rzeczywistym z pomieszczeń, w których zainstalowano urządzenia. Zaprojektowano prosty i intuicyjny program obsługi dostępny na każdej z najbardziej popularnych platform. Zdecydowano się na aplikację internetową i dwie aplikacje mobile przeznaczone na system iOS i Android. Dane zbierane są za pomocą urządzeń Raspberry Pi, wyposażonych w szereg czujników i kamery. Jeżeli na jakimkolwiek z czujników pojawią się odczyty odbiegające od normy, system powiadamia wszystkie urządzenia podłączone do konta użytkownika o zagrożeniu. Za poprawne działanie samego systemu, odpowiadają usługi chmurowe MS Azure oraz Google Firebase. Postanowiono, że cały system będzie na licencji open-source, aby każdy mógł pobrać oprogramowanie i edytować je według własnych potrzeb.

Rozdział 2

Wstęp

Inspiracją niniejszego projektu jest chęć stworzenia niezależnego systemu monitoringu wraz z nowoczesnymi mechanizmami powszechnie używanymi w projektach programistycznych. Współczesne systemy kontroli bezpieczeństwa powinny nie tylko nagrywać obraz, ale także analizować go w czasie rzeczywistym i odpowiednio reagować na wykryte zmiany. Na podstawie danych z kamer i czujników system powinien podejmować decyzje o stanie bezpieczeństwa domu i w razie potrzeby alarmować użytkownika o wykrytych zagrożeniach.

2.1 Cel i zakres pracy

System kontroli bezpieczeństwa - The Guard to nasza odpowiedź na przedstawione problemy. Celem pracy było stworzenie systemu umożliwiającego analizę danych z czujników pomiarowych, monitorowanie pomieszczeń, w których zamontowano system, a także nagrywanie materiału wideo w momencie wykrycia ruchu i przechowywanie go bezpiecznie na zewnętrznym serwerze, aby był dostępny dla nas w każdym momencie i nie uległ zniszczeniu. Do zaadań systemu należało także poinformowanie o każdym niebezpieczeństwie właściciela systemu. Priorytetem był prosty i intuicyjny program obsługi, który mógłby być użyty przez każdą osobę, na każdej z najbardziej popularnych platform. Zdecydowano się na aplikację internetową oraz dwie aplikacje mobilne napisane natywnie dla systemu iOS i Android. Ponadto uzgodniono, że rozwiązanie będzie oparte na niezależnych modułach, które będzie można później w łatwy sposób zmodyfikować. Całość pracy oparta jest na licencji open-source, aby użytkownicy mogli nie tylko korzystać z systemu, ale także dowolnie go edytować i dopasowywać do własnych potrzeb.

W ramach pracy przygotowano projekt całego systemu, od urządzeń zbierających dane, przez oprogramowanie monitorujące i analizujące zebrane dane, po aplikacje klienckie. Następnie zaimplementowano zaprojektowane wcześniej aplikacje oraz złożono zestawy urządzeń składających się z Raspberry Pi 3, czujników i kamer. Ze względu na cel pracy oraz wykorzystane technologie i usługi, zespół oparł swoją pracę o dokumentację usług dostępną na stronach internetowych producentów, dokumentację narzędzi dołączoną do odpowiednich repozytoriów, dokumentację sprzętu.

2.2 Podział pracy

- Mateusz Bartos:

Zaprojektowanie architektury systemu, stworzenie aplikacji mobilnej przeznaczonej na system Android, przygotowanie maszyny wirtualnej w ramach usług oferowanych przez chmurę Microsoft Azure.

- Piotr Falkiewicz:

Wykonanie projektu i zaimplementowanie serwera obsługującego aplikacje mobilne w oparciu o protokół HTTP, przetwarzanie strumienia obrazu dostarczanego w czasie rzeczywistym z urządzeń do aplikacji przy wykorzystaniu modułu Nginx RTMP.

- Aleksandra Główczewska:

Projekt i wykonanie aplikacji internetowej wraz z własną obsługą baz danych, oparcie aplikacji na języku Python i bibliotece Django [1], wprowadzenie uwierzytelniania użytkowników.

- Paweł Szudrowicz:

Przygotowanie urządzenia opartego na Raspberry Pi 3, wykonanie oprogramowania na Raspberry Pi 3 obsługującego czujniki i kamerę w czasie rzeczywistym. Ponadto, projekt i wykonanie aplikacji mobilnej przeznaczonej na urządzenia z systemem iOS, implementacja obsługi push notyfikacji wykorzystującej do tego usługę Firebase.

2.3 Struktura pracy

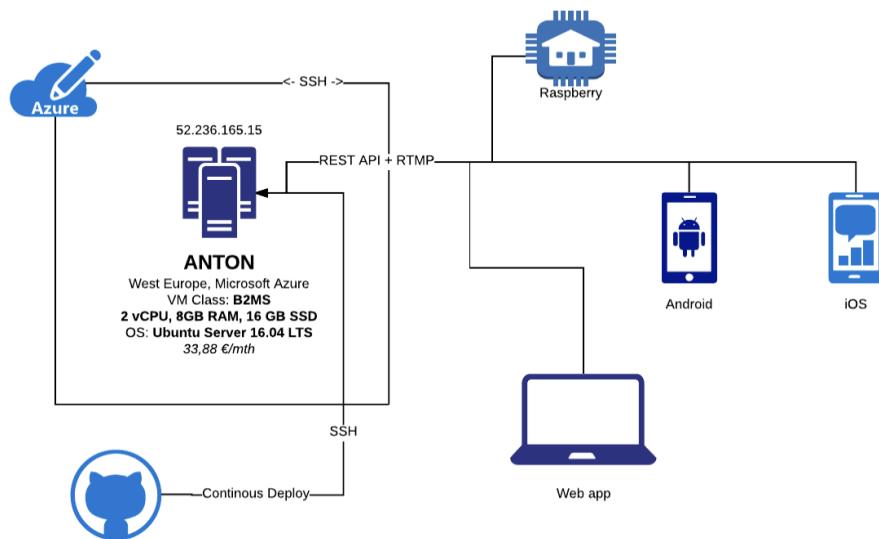
W pierwszym rozdziale opisana jest architektura przygotowanego systemu. Następna część poświęcona jest kosztom utrzymania działającego systemu i wszystkich wymaganych podzespołów do jego poprawnej pracy. Kolejny dział zawiera specyfikację wykorzystanych czujników i schemat poprawnego ich podłączenia, a także informacje dotyczące przetwarzania obrazu. W części tej poruszona jest również kwestia poprawnej instalacji oprogramowania na urządzeniu Raspberry Pi 3. W czwartej części zaprezentowane są użycie rozwiązań chmurowe takie jak Microsoft Azure, Firebase i omówione jest działanie serwera opartego na Django. Aplikacje klienckie są opisane w szóstym rozdziale poniższej pracy. W tym rozdziale wykorzystane są zdjęcia ekranów z działających aplikacji wraz z opisem najważniejszych aspektów ich realizacji. Ostatni rozdział zawiera opis przeprowadzonych testów funkcjonalności.

Rozdział 3

Architektura systemu

Dobre zaprojektowanie architektury systemu jest fundamentalnym zadaniem. Na rynku istnieje wiele infrastruktur opartych na technologii przetwarzania w chmurze, które oferują bardzo podobne funkcjonalności. Należało wybrać te, które dawały najwięcej korzyści przy jak najniższej cenie (kosztorys omówiono w rozdziale 3). Dlatego zdecydowano się na Microsoft Azure. Dodatkowym atutem było to, że zespół miał już doświadczenie z tą usługą. Opis działania poszczególnych elementów systemu zostanie omówiony w dalszej części pracy.

3.1 Schemat



Rysunek 3.1: Schemat systemu [opracowanie własne]

Architektura została przedstawiona na schemacie (rys 2.1). Wszystkie urządzenia klienckie (iOS, Android i Web), jak i Raspberry Pi komunikują się z serwerem ANTON wykupionym i pracującym na platformie Microsoft Azure. Komunikacja pomiędzy klientami a serwerem odbywa się dzięki REST API. Obraz natomiast przesyłany może być za pomocą dwóch protokołów RTMP i HLS. Postanowiono, że transmisja obrazu w systemie The Guard oparta będzie na protokole HLS ze względu na brak możliwości obsługi RTMP na iOS. Priorytetem było zapewnienie identycznych warunków i tych samych doświadczeń użytkownika na wszystkich platformach. Było to głównym powodem całkowitego odrzucenia przesyłania obrazu przy użyciu protokołu RTMP. RTMP posiada jednak w porównaniu do HLS jedną, aczkolwiek bardzo istotną przewagę. Jest to brak opóźnienia

w transmisji obrazu. HLS wysyła dane w małych porcjach. Przed wysłaniem pierwszej części, konieczne jest jej nagranie. To właśnie powoduje kilkunastosekundowe opóźnienie w stosunku do RTMP, który transmituje obraz bezpośrednio. Oba protokoły omówiono dokładniej w rozdziale 4.3.

3.2 Komunikacja

Komunikacja pomiędzy elementami systemu odbywa się na zasadach architektury REST. Wiadomości przesyłane są asynchronicznie, na wskazane wcześniej adresy. Takie podejście gwarantuje prostotę przesyłanych komunikatów oraz skalowalność w kontekście nowych urządzeń Raspberry strumieniujących dane oraz nowych urządzeń korzystających z aplikacji klienckich. Początkowo projekt oparto o zapytania GET i POST. [2]. Wprowadzenie tokenów uwierzytelniających (więcej w akapicie nt. Bezpieczeństwa) spowodowało, że wymianę komunikatów oparto tylko i wyłącznie na zapytaniach POST.

Zapytanie GET Metoda GET pozwala na pobranie dokumentu sieciowego, na postawie zapytania zawartego w adresie URL. Metoda ta jest używana tylko i wyłącznie do pobierania danych z punktu docelowego.

Zapytanie POST W metodzie POST, należy zamieścić wiadomość wewnętrz zapytania HTTP. Odpowiedzią na ten typ zapytania, może być zarówno kod statusu, jak i dane, zwarcane w podobnej postaci jak przy zapytaniu GET.

Wysłanie zapytania na określony adres powoduje uruchomienie specjalnej funkcji na serwerze. Każdy adres ma przypisaną osobną funkcję uruchamianą automatycznie po otrzymaniu zapytania. Funkcje te realizują operacje na bazie danych (CRUD) lub odpowiedzialne są za wysyłanie notyfikacji do wszystkich urządzeń klienta. Przykładowo (rys. 2.2) funkcja obsługująca dodanie nowego urządzenia w bazie danych, która uruchamia się po wysłaniu zapytania na adres /backend/v1/devices/add. Funkcja ta sprawdza czy zapytanie jest typu POST, pobiera przesłane dane i wykorzystuje je do utworzenia nowego rekordu w bazie danych.

```
# PATH /backend/v1/devices/add/
@api_view(['POST'])
@csrf_exempt
def register_rasp(request):
    context = {}
    if request.method == 'POST':
        body_unicode = request.body.decode('utf-8')
        body = json.loads(body_unicode)
        serial = body['serial']
        owner = body['owner']
        authorize_request(body['token'])
        device, created = Rasps.objects.get_or_create(serial=serial, defaults={'owner': owner, 'name': 'Guard', 'isArmed': True})
        if created:
            return JsonResponse(content_type = "application/json",status_code = 200)
    return JsonResponse(content_type = "application/json",status_code = 400)
```

Rysunek 3.2: Funkcja obsługująca dodanie nowego urządzenia [opracowanie własne]

3.3 Bezpieczeństwo

Aplikacje wysyłając zapytania do serwera muszą potwierdzić swoją tożsamość. W aplikacjach mobilnych zastosowano proponowane przez Firebase rozwiązanie JSON Web Tokens. W momencie wysłania zapytania POST do serwera, aplikacja dołącza także unikalny token generowany przez moduł Firebase Auth. Po dotarciu wiadomości do serwera, token ten weryfikowany jest przy użyciu modułu Firebase Admin SDK. Jeśli weryfikacja tokenów przebiegła pomyślnie (oba tokeny

są identyczne) to udzielany jest dostęp do wykonania kodu na serwerze. W przeciwnym wypadku generowany jest błąd.

W przypadku aplikacji internetowej zastosowano wbudowane w bibliotekę Django zabezpieczenia - przesyłanie tokenu CSRF oraz identyfikatora sesji wraz z zapytaniem [3]. Zabezpieczenie CSRF token uniemożliwia tzw. ‘Cross Site Request Forgery’ tj. ataki w których na stronie internetowej, bez wiedzy użytkownika uruchamiany jest skrypt (najczęściej w języku JavaScript). Następnie korzystając z faktu, że użytkownik jest zalogowany strona atakująca podszywa się pod jego konto i wysyła zapytanie do serwera, które może spowodować uruchomienie wszystkich operacji, do których upoważniony jest dany użytkownik. Aby tego uniknąć CSRF token zapisywany jest w przeglądarce jako ‘ciasteczko’ (eng. cookie) i dodawany do danych przesyłanych w momencie wyboru przycisku odpowiedzialnego za przesłanie formularza. Wbudowana w serwer Django biblioteka weryfikuje na podstawie zapisanych i przesłanych danych sesji poprawność tokenu. W przypadku błędu zwraca błąd serwera 403.

Ponieważ token przy każdym zapytaniu jest tworzony na nowo, na podstawie otwartej sesji pozostaje on rozwiązaniem przeznaczonym głównie dla aplikacji przeglądarkowych - powyższe rozwiązanie nie byłoby komfortowe dla użytkowników aplikacji mobilnych: aplikacja musiałaby ustanowić połączenie z serwerem (wysłać zapytanie GET na stronę główną) następnie zalogować się (wysłać zapytanie POST z danymi logowania) oraz zapisywać parę (token, id sesji) odsyłaną przez serwer. Aby ograniczyć ilość zapytań wysyłanych do serwera, w wypadku aplikacji mobilnych posłużono się inną opisaną powyżej metodą tokenów JWT.

Rozdział 4

Kosztorys

4.1 Koszty elementów systemu

W tym rozdziale postarano się o umieszczenie wszystkich kosztów budowy systemu i kosztów jego utrzymania. Koszt budowy systemu zależy od ilości posiadanych urządzeń pomiarowych. Zaprezentowany zostanie kosztorys tylko jednego urządzenia.

- Koszty czujników

Do budowy użyto czujników: MQ-9 [4], MQ-2 [5], DS18B20+ [6], czujnik ruchu [7], czujnik wykrycia płomieni [8]. Na stronach odnośników znajdują się ich ceny ze sklepu Botland.pl.
Sumaryczny koszt: 91,60 zł.

- Koszt kamery

Użyto kamery 5MP Full HD ze wsparciem do modułu Raspberry Pi 3. Cena: 89,00 zł.

- Koszt przetwornika AC

Użyto przetwornika MCP3008 [9]. Jego koszt wynosi: 9,90 zł.

- Koszt Raspberry Pi 3 + karta pamięci 16GB

Ze względu na bardzo dużą ilość dostępnych źródeł, ceny jego nabycia różnią się znacząco.
Cena z Botland.pl: 209,90 zł

Całkowity koszt jednego urządzenia wynosi zatem około 400,40 zł. Istnieje możliwość obniżenia tej kwoty poprzez zakup kamery nagrywającej w mniejszej rozdzielczości.

4.2 Koszty użytkowania

Poza jednorazowymi kosztami związanymi z montażem elementów użytkownik ponosi także koszty miesięczne związane z utrzymaniem części systemu znajdującej się w chmurze.

- Koszty maszyny wirtualnej w chmurze Microsoft Azure

Przy wykorzystaniu maszyny klasy B2MS miesięczny koszt działania wynosi **60,98 €**

- Koszty usług Google Firebase

Usługa jest darmowa jeżeli przestrzegane są poniższe limity:

Firebase Realtime Database

- **100** równoczesnych połączeń,

- **1 GB** przechowywanych danych,

- **10 GB** danych transferowanych w ciągu miesiąca.

Firebase Storage

- **1 GB** danych pobieranych w ciągu dnia,
- **5 GB** przechowywanych danych,
- **20 000** operacji wysyłania danych dziennie,
- **50 000** operacji pobierania danych dziennie.

Rozdział 5

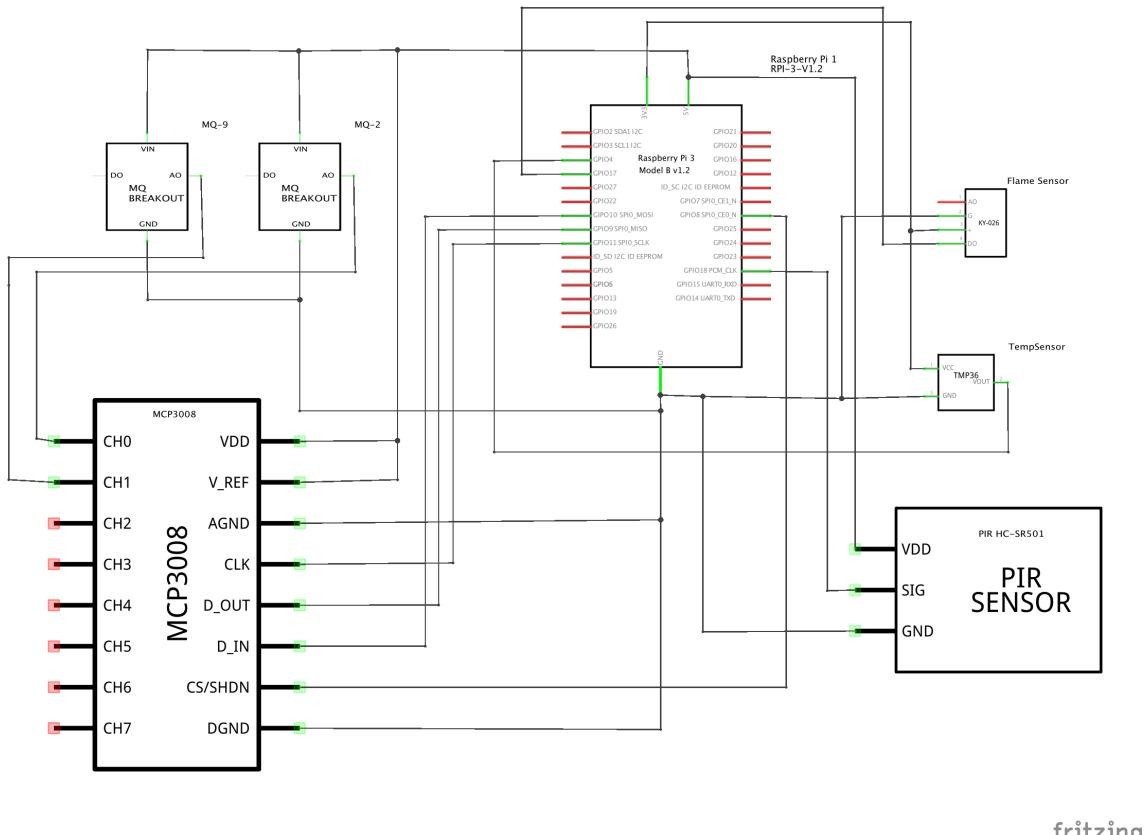
Zbieranie i przetwarzanie danych z czujników

5.1 Raspberry Pi

Wszystkie zestawy zbudowano w oparciu o Raspberry Pi 3 v1.2. Zdecydowano się na to rozwiązanie, ponieważ bazuje na dystrybucji Linuxa, posiada opowiadnie interfejsy i złącza, a także zintegrowany moduł WiFi. Minusem w stosunku do konkurencyjnego Arduino jest brak wejść analogowych. Problem rozwiązało dodając zewnętrzny przetwornik A/C. Całość zamknięto w małą plastikową obudowę z wyciętymi otworami na czujniki (rys. 5.1). Schemat budowy układu wykonano w programie Fritzing (rys. 5.2).



Rysunek 5.1: Zbudowany zestaw The Guard [opracowanie własne]



fritzing

Rysunek 5.2: Schemat układu The Guard [opracowanie własne]

Specyfikacja Raspberry Pi 3 na podstawie strony Botland.com.pl [10].

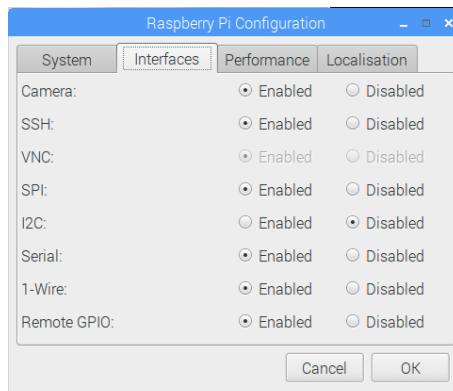
- Procesor 1.2 GHz
- Liczba rdzeni 4. Quad Core
- Pamięć RAM 1 GB
- Karta pamięci microSD (w użytych w tej pracy urządzeniach posłużono się kartami o wielkości 8 i 16 GB).
- 40 GPIO

Aby przygotować dowolne urządzenie Raspberry Pi 3 do poprawnej instalacji oprogramowania The Guard należy wykonać poniższe czynności w terminalu.

1. sudo apt-get install libx264-dev
2. cd /usr/src
3. git clone git://source.ffmpeg.org/ffmpeg.git
4. sudo ./configure --arch=armel --target-os=linux --enable-gpl --enable-libx264 --enable-nonfree
5. sudo make
6. sudo install
7. sudo nano /boot/config.txt

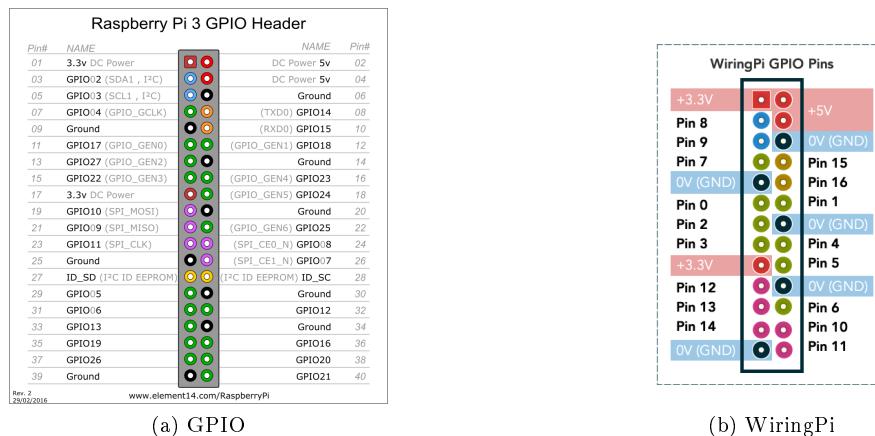
8. w pliku config.txt dopisać Dtoverlay=w1-gpio i Gpiopin=4
9. pip intall wiringpi
10. sudo pip install spidev
11. pip install pyrebase

Następnym krokiem jest włączenie odpowiednich interfejsów w panelu konfiguracyjnym. Należy zmienić ustawienia zgodnie ze schematem (rys. 5.3).



Rysunek 5.3: Ustawienia [opracowanie własne]

Do odczytu danych z układów cyfrowych użyto biblioteki wiringPi. Należy podkreślić, że numeracja fizycznych pinów (rys. 5.4a) i numeracja pinów w wiringPi (rys. 5.4b) jest różna oraz biblioteka wiringPi nie obsługuje wszystkich pinów urządzenia. Przykładowo odczyt pinu numer 1 w wiringPi jest równoznaczny z odczytem stanu pinu numer 12 (GPIO18).



Rysunek 5.4: Porównanie pinów Raspberry Pi 3 [11] z pinami wiringPi [12]

Zainstalowane oprogramowanie odpowiedzialne jest za ciągłe monitorowanie stanów i zbieranie danych z czujników pomiarowych. Po podłączeniu układu do zasilania program jest uruchamiany automatycznie. Pierwszą czynnością jaką wykonuje Raspberry Pi jest wysłanie swojego numeru seryjnego do bazy danych Firebase. Cały proces jest w pełni zautomatyzowany. Dzięki temu użytkownicy od razu mogą dodać urządzenie i przeglądać dane z czujników na aplikacjach klientów. Dodanie akcesorium pomiarowego następuje poprzez wprowadzenie w aplikacji jego numeru seryjnego.

5.2 Czujniki

Każdy zestaw składa się z 5 czujników, jednej kamery oraz jednego przetwornika analogowo-cyfrowego (AC).

a) Specyfikacja MQ-9 - czujnik tlenku węgla[4].

- Zasilanie: 5 V
- Pobór prądu: 150 mA
- Temperatura pracy: od -10 do 50 °C
- Wyjścia: analogowe oraz cyfrowe (do pomiarów użyto wyjścia analogowego)

b) Specyfikacja MQ-2 - czujnik LPG i dymu [5].

- Zasilanie: 5 V
- Pobór prądu: 150 mA
- Temperatura pracy: od -10 do 50 °C
- Wyjścia: analogowe oraz cyfrowe (do pomiarów użyto wyjścia analogowego)

c) Specyfikacja czujnika wykrywania płomieni [8].

- Zasilanie: 3.3 V
- Zakres wykrywanej fali: 760 do 1100nm
- Kąt detekcji: od 0 do 60 stopni
- Temperatura pracy: od -25 do 85 °C

d) Specyfikacja DS18B20 - czujnik temperatury [6].

- Zasilanie: 3.3 V
- Zakres pomiarowy: od -55 do 125 °C

e) Kamera:

- Wykorzystano moduł kamery Raspberry Pi
- Kamera 5MP - wspierająca nagrywanie 30 klatek na sekundę w rozdzielcości Full HD

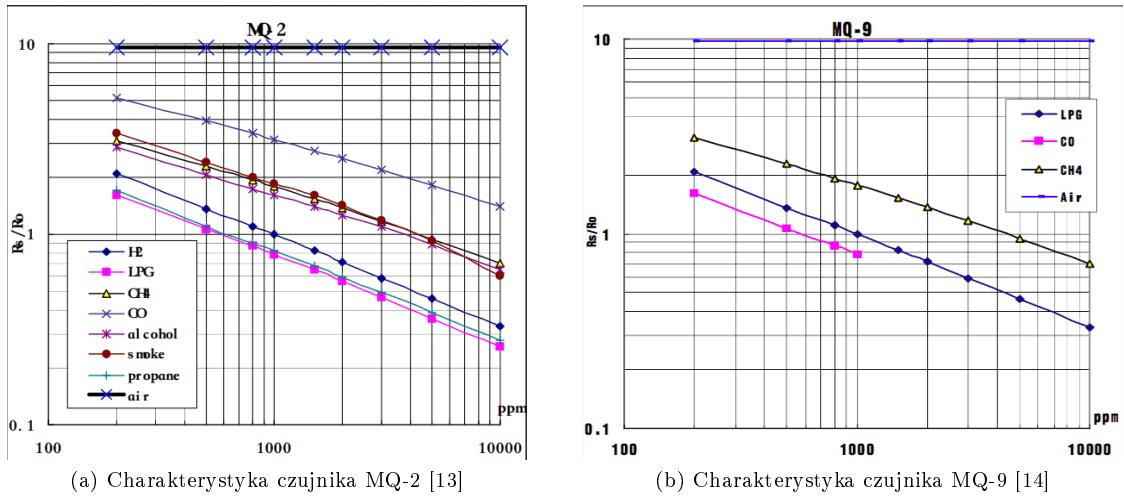
f) Specyfikacja MCP3008 - przetwornik A/C [9].

- Zasilanie: od 2.7V do 5.5V
- Pobór prądu: 0.5 mA
- Interfejs komunikacyjny: SPI
- Liczba kanałów: 8
- Rozdzielcość: 10bit

g) Specyfikacja czujnika ruchu PIR HC-SR501 [7].

- Zasilanie: od 4.5V do 20V
- Pobór prądu w stanie czuwania: 50 μ A
- Zakres pomiarowy: maks. 7 m
- Kąt widzenia: do 100°

Na wykresach (rys. 5.5a, rys. 5.5b) przedstawiono charakterystyki czułości czujników.



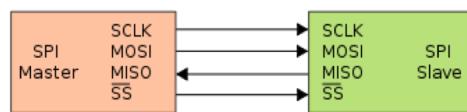
Rysunek 5.5: Charakterystyka czujników MQ-2 oraz MQ-9.

R_o - jest to stała wartość oporu czujnika przy 1000 ppm H₂ w czystym powietrzu.

R_s - jest to opór czujnika w różnych stężeniach gazu.

Przy założeniu stałej wartości R_o , przy wzroście R_s czułość czujnika maleje - im mniejszy stosunek R_s do R_o tym dokładniejszy pomiar wykonywany przez czujnik. Na schemacie 5.5 widać, że obydwa czujniki reagują na wiele różnych gazów. MQ-2 nazwano w tej pracy czujnikiem LPG, a MQ-9 czujnikiem CO ze względu na to, że w stosunku do tych gazów mają najwyższą czułość.

Żaden z modeli Raspberry nie posiada wbudowanego przetwornika analogowo-cyfrowego dlatego konieczne było użycie układu zewnętrznego. Wybrano przetwornik MCP3008 ze względu na jego niski koszt i komunikację poprzez interfejs SPI, który jest wspierany przez Raspberry Pi. MCP3008 to 10-bitowy przetwornik analogowo-cyfrowy zasilany napięciem 5V. Przetwornik 10-bitowy jest w stanie rozróżnić 1024 stany. Posiada 8 kanałów jednak w projekcie wykorzystano tylko 2 – dla MQ-9 i MQ-2.



Rysunek 5.6: Interfejs SPI [15]

Interfejs SPI: SPI jest to interfejs synchroniczny (rys. 5.6). Może być do niego podłączone wiele urządzeń typu Slave, jednak są połączone tylko z jednym urządzeniem Master, które generuje

sygnał zegarowy. Urządzenie typu Master poprzez linię SS wybiera urządzenie z którym chce się komunikować.

Interfejs SPI zawiera jeszcze 3 linie.

1. MOSI (ang. Master Output Slave Input):

Poprzez tę linię wysyłane są dane z Raspberry Pi do przetwornika analogowo cyfrowego MCP3008.

2. MISO (ang. Master Input Slave Output):

Poprzez tę linię wysyłane są dane z przetwornika AC do układu Master, czyli w naszym przypadku Raspberry Pi 3.

3. SCLK (ang. Serial Clock) :

Ta linia wykorzystywana jest do przesłania zegara wygenerowanego przez Raspberry Pi 3.

Do komunikacji poprzez ten interfejs wykorzystano bibliotekę SpiDev [16].

Każdy układ monitoruje wskaźniki pomiarowe z czujników analogowych i cyfrowych. W przypadku wykrycia wskazań, które w znaczący sposób odbiegają od normy informuje właściciela o zagrożeniu. Informacja ta wysyłana jest do wszystkich urządzeń(smartfony, tablety itp), które posiada właściciel. Analizując dane z czujników analogowych w czystym powietrzu, które wynoszą odpowiednio:

czujnik MQ-9: od 0.15 do 0.2,

czujnik MQ-2: od 0.05 do 0.15,

przyjęto, że granicą wysłania notyfikacji do użytkownika jest przekroczenie progu 0.3. Wartości te to znormalizowane dane z przetwornika AC, który jak już wcześniej wspomniano wykrywa 1024 stany. Odczytywane wartości bezpośrednio na wyjściu przetwornika MCP3008 dla czujnika MQ-9 w czystym powietrzu to około 170. Stąd $170/1024 = 0.166$. Wysłanie notyfikacji wiąże się z otrzymaniem wartości większej niż 308.

Czujniki cyfrowe wykorzystane w pracy informują o wykryciu płomieni lub ruchu. Czujnik ruchu wykrycie zagrożenia określa przez stan wysoki, natomiast czujnik płomieni przez stan niski. Przy implementacji zanegowano sygnał odbierany z czujnika płomieni, aby stan wysoki zawsze informował o niebezpieczeństwie, a stan niski reprezentował jego brak. Na czujnikach znajduje się potencjometr, za pomocą którego dowolnie można ustawić jego czułość. Odczyt danych następuje nieprzerwanie co 2 sekundy. Nie należy obawiać się, że aplikacja nie odczyta zagrożenia z powodu zbyt krótkiego trwania sygnału wysokiego w bazie danych, ponieważ czujnik utrzymuje stan wysoki przez 5 sekund po wykryciu ruchu.

Oprogramowanie wysyła także informacje z czujników do bazy danych Firebase. Zastosowanie takiej bazy daje możliwość monitorowania wszystkich danych w czasie rzeczywistym na aplikacjach klienckich. Dodatkowo w przypadku zagrożenia czyli przekroczeniu progu, o którym mowa wyżej wysyłana jest push notyfikacja do urządzeń użytkownika, a informacja o zagrożeniu zapisywana jest w bazie danych Django. Każdy użytkownik jest w stanie odtworzyć całą historię wydarzeń w swoim systemie.

Aby zapewnić wydajny i pewny system bezpieczeństwa przy otrzymaniu wysokich wartości na czujnikach zapisywany jest czas zdarzenia. Każda kolejna notyfikacja zostanie wysłana po upływie 10 minut od poprzedniej przy założeniu, że stan na czujniku nadal jest wysoki.

5.3 Obsługa wideo

Protokół RTMP Podstawą funkcji strumieniowania wideo jest protokół RTMP (ang. Real-Time Message Protocol). Jest to oparty na protokole TCP protokół wysyłania obrazu, dźwięku oraz danych. [17] Podstawową jednostką danych w protokole RTMP jest wiadomość (ang. Message), której struktura jest zależna od typu strumieniowanych informacji. Wiadomości dzielone są na części (ang. Chunks), które są gotowe do transmisji. Strumień RTMP to ostatecznie strumień fragmentów wiadomości (ang. Chunk Stream) [18]

Ponadto wykorzystano protokół HLS (HTTP Live Streaming), którego cechą jest zapisywanie odbieranego obrazu w plikach o określonej długości. Gdy aplikacja kliencka odtwarza strumień wideo, w rzeczywistości odbiera strumieniowane po kolej zapisane pliki .ts. Zaletą takiego rozwiązania jest płynność odbieranego wideo, a jego wadą opóźnienie w transmisji.

H264 W pracy wykorzystano kodowanie obrazu zgodnie ze standardem H.264. Charakteryzuje go niska złożoność algorytmów kompresji oraz niewielkie opóźnienie dzięki czemu idealnie nadaje się do zadań związanych z szybkim enkodowaniem obrazu przed wysłaniem. [19] Cechą charakterystyczną dla tego typu kodowania wideo jest użycie klatki kluczowej (ang keyframe, i-frame). Jest to pełna klatka obrazu, podczas gdy następujące po niej dane wyrażają różnice między dwoma tą a następnymi klatkami. Rozwiązywanie to pozwala to na zmniejszenie rozmiarów ostatecznego obrazu wideo.

Raspberry Pi Do obsługi odbioru strumienia wideo po stronie Raspberry Pi wykorzystywane jest narzędzie ffmpeg, które pozwala na przechwytywanie obrazu z kamery, ustawianie własności wysyłanego obrazu oraz punktu docelowego na który ten obraz będzie przesłany.

Pobranie numeru seryjnego urządzenia i użycie go, jako fragment adresu końcowego, gwarantuje stworzenie unikatowego dla każdego urządzenia adresu. Poniżej przedstawiono skrypt wykonujący wymienione zadania. Do pobrania obrazu z kamery Raspberry pi posłużono się narzędziem raspivid. [20]

```
#!/bin/bash
serial_id=$(cat /proc/cpuinfo | grep Serial | cut -d ' ' -f 2)
raspivid -o - -t 0 -fps 30 -b 1000000 | ffmpeg -re -ar 44100 -ac 2
-acodec pcm_s16le -f s16le -i /dev/zero -f h264 -i - -vcodec copy -g 60
-strict experimental
-f flv rtmp://52.236.165.15:1936/camera/${serial_id}
```

Pierwszą czynnością wykonywaną w skrypcie jest otwarcie pliku /proc/cpuinfo. Następnie znajdująca jest w nim linia, w której znajduje się unikalny serial urządzenia. Następnie, z wykorzystaniem potoku i funkcji cut wartość ta zostaje przypisana do zmiennej serialid.

- Pierwszym przełącznikiem jest -o oraz parametr -. Oznacza to, że obraz z kamery jest wysyłany na wyjście standardowe.
- Przełącznik -t ustawiony na 0 pozwala przekazywać obraz z modułu kamery przez nieokrelony czas. Aby przestać pobierać wideo, należy użyć przerwania za pomocą sygnału SIGINT (obsługiwanego w terminalu skrótem klawiszowym CTRL+C).
- Opcja -fps pozwala wskazać liczbę przechwytywanych klatek w ciągu sekundy. Tutaj wykorzystano maksymalne możliwości wybranego modułu kamery.

- Ostatnią opcją, wykorzystaną w pobieraniu obrazu z kamery, jest bitrate, tzn wielkość jednostki pamięci w której ma się znaleźć obraz przechwycony w ciągu 1 sekundy strumienia. Ustawienie opcji -b na 1000000 oznacza, że 1 sekunda wideo, może zajmować 125 kilobajtów pamięci. Jest to szczególnie istotna informacja w kontekście transmisji obrazu poza urządzenie przy wykorzystaniu łącza internetowego.

Drugim polecienniem jest wywołanie narzędzia ffmpeg. Odbiera ono za pomocą potoku przechwytywany obraz i przekazuje go na docelowy punkt końcowy. Za jego pomocą ustalane są ostateczne opcje kodujące obraz w trakcie transmisji.

- Przełącznik -re pozwala odczytywać dane wejściowe z oryginalną częstotliwością. (W powyższym skrypcie znajdują się przechwycone ustawienia przełącznika narzędzia raspivid -fps 30).
- Opcje -ar, -ac, -acodec, -f, -strict odpowiadają kolejno za: próbkowanie dźwięku, wybór liczby kanałów, kodk audio, format dźwięku oraz wybór eksperymentalnego sposobu kodowania. Wymuszenie wykorzystania, jako wejścia strumienia dźwięku, na /dev/zero, oznacza, że strumień ten zostaje wypełniony wartościami pustymi. Zatem opcje transmisji dźwięku są nieistotne.
- Przełącznik -vcodec ustala kodęk wideo. W pracy wykorzystano standard kodowania H.264.
- Następnie ustalone zostało wejście obrazu. Przełącznik -i - powoduje, że narzędzie ffmpeg przechwytuje, dzięki potokowi, obraz przekazywany funkcją raspivid.
- Opcja -g 60 oznacza, że tzw klatka kluczowa (ang. keyframe) pojawia się co 60 klatek (w tej sytuacji co 2 sekundy).
- Przełącznik -f w przypadku strumienia obrazu z kamery wymusza format nadawanego wideo.
- Ostatnim elementem polecenia jest podanie punktu docelowego dla strumienia. Za pomocą protokołu RTMP, obługiwanego przez serwer o adresie IP 52.236.165.15 na porcie 1936, obraz wysyłany jest na aplikację o nazwie camera i punkt charakteryzowany przez numer seryjny urządzenia. Działanie tego elementu opisano w kolejnym punkcie.

Serwer Narzędziem umożliwiającym obsługę proxy jest serwer nginx. W części projektu związanej ze strumieniowaniem wideo wykorzystano moduł nginx-rtmp [21]. Umożliwia on obsługę obrazu transmitowanego z wielu źródeł, na wiele urządzeń równocześnie. Moduł ten pozwala na realizację wielu funkcji, a niektóre z nich przedstawiono poniżej.

- Tworzenie dynamicznych punktów końcowych, dla urządzeń strumieniujących obraz.
- Zmianę parametrów przechwytywanego obrazu .
- Zapisywanie nagrani po stronie serwera.
- Tworzenie punktów nadających, dla aplikacji odtwarzających strumień.

Ponadto pozwala na utworzenie aplikacji HLS przechowującej tymczasowo obraz, zanim zostanie on przesłany dalej. Pozwala to na uniknięcie opóźnień między kolejnymi klatkami obrazu.

Wszystkie funkcjonalności są zdefiniowane poprzez ustawienia pliku konfiguracyjnego, którego lokalizacja to /usr/local/nginx/conf/nginx.conf:

```

rtmp {
    server {
        listen 1936;
        chunk_size 4096;
        application camera {
            hls on;
            hls_path /mnt/hls/;
            hls_fragment 2;
            hls_playlist_length 3;
            allow publish all;
            allow play all;
            live on;
            record off;
        }
    }
}

```

Powyższy plik konfiguracyjny powoduje, że serwer rtmp dostępny jest na porcie 1936 (domyślne porty dla protokołu RTMP na urządzeniach z systemem z rodziną Ubuntu to 1935 i 1936). Następnie tworzona jest aplikacja o nazwie camera. Dla aplikacji wysyłających dane, jest ona dostępna pod adresem: `rtmp://<ip-serwera>:1936/camera/<klucz>`, gdzie klucz jest wybierany przez aplikację i tworzony dynamicznie, gdy tylko urządzenie zacznie nadawać dane pod wskazany adres. Ustawienia aplikacji decydują o tym, że dla urządzeń odtwarzających wideo, dostępne jest ono dzięki aplikacji HLS - opcja hls on. Na szczególną uwagę zasługują linie: hls fragment 2 oraz hls playlist length 3, które wskazują na to, że po stronie serwera nagrywane są 2 tymczasowe fragmenty, o długości 3 sekund każdy. Pliki te są przechowywane w folderze `/mnt/hls/`, a ich nazwę jednoznacznie będzie wskazywać klucz strumienia.

```

http {
    server {
        listen      80;
        location /hls {
            types {
                application/vnd.apple.mpegurl m3u8;
                video/mp2t ts;
            }
            root /mnt/;
        }
    }
}

```

Konfiguracja ta udostępnia aplikację HLS. Dla aplikacji klienckich, obraz będzie dostępny pod adresem: `http://<ip-serwera>:80/hls/<klucz>.m3u8`, o czym decyduje konfiguracja części związanej z serwerem RTMP.

Rozdział 6

Rozwiązania chmurowe

6.1 Microsoft Azure

Aby zapewnić wysoki poziom bezpieczeństwa oraz dostępności systemu zdecydowano się na skorzystanie z chmury Microsoft Azure. Wykorzystany został serwer wirtualny, na którym zainstalowano system Ubuntu 16.04 LTS. Dostęp do serwera jest możliwy wyłącznie przy użyciu protokołu SSH.

Parametry maszyny wirtualnej 'Anton'.

- **Klasa:** B2MS
- **Lokalizacja:** Zachodnia Europa
- **Moc obliczeniowa:** 2 vCPU
- **Pamięć RAM:** 8 GB
- **Dysk SSD:** 16 GB
- **Ilość dysków danych:** 4
- **Ilość operacji wejścia/wyjścia na sekundę:** 4800
- **Koszt:** 60,98 € miesięcznie

Po skonfigurowaniu środowisk deweloperskich zarówno dla części obsługującej aplikacje mobilne oraz części odpowiedzialnej za przetwarzanie obrazu, wybrany katalog podłączono do repozytorium kodu korzystającego z systemu kontroli wersji ‘git’, który znajduje się w serwisie GitHub.com. W ten sposób bieżące zmiany były dokonywane na komputerach deweloperów, którzy za pomocą repozytorium publikowali nowe wersje aplikacji serwerowej w chmurze.

6.2 Firebase

W celu usprawnienia działania aplikacji klienckich skorzystano także z usług platformy Firebase. Platforma Firebase jest częścią chmury Google Cloud i oferuje rozwiązania przedstawione poniżej.

- **Firebase Auth** - usługę bezpiecznej autoryzacji użytkowników.
- **Firebase Storage** - usługę wygodnego przechowywania plików.

- **Firebase Realtime Database** - usługę bazy danych aktualizowanej w czasie rzeczywistym.

Usługi autoryzacji użytkowników zostały wykorzystane nie tylko w aplikacjach klienckich, ale także na serwerze do autoryzacji tokenów z nadchodzących żądań klientów. Usługa przechowywania plików została wykorzystywana do przesyłania fragmentów nagrani, na których wykryto ruch spowodowany przez człowieka. Usługa bazy danych została wykorzystana do prezentowania w czasie rzeczywistym wartości czujników w aplikacjach klienckich.

6.3 Aplikacja serwerowa

Serwer obsługujący aplikacje mobilne wykonano w języku Python korzystając z rozwiązań Django z wykorzystaniem modułu Django-Rest-Framework. Umożliwia on tworzenie punktów końcowych (ang. endpoint) które, zgodnie z rozdziałem nt. komunikacji mogły odbierać zapytania POST. Obsługę zapytań można podzielić ze względu na źródło wysłania wiadomości: od aplikacji użytkownika lub od urządzenia Raspberry. W pierwszej kolejności przedstawione zostaną wiadomości wymieniane na linii Raspberry - Serwer.

a) Rejestracja Raspberry Pi.

```
Adres: /backend/v1/devices/add
Zawartość:
{
  'serial': <serial-urządzenia>,
  'name': <nazwa-urządzenia>,
  'token': 'jwt.token.from.client'
}
```

Działanie: Raspberry, o podanym numerze seryjnym i nazwie zostaje dodane do bazy danych urządzeń.

b) Wykrycie ruchu:

```
Adres: /backend/v1/PIRnotification
Zawartość:
{
  'serial': <serial-urządzenia>,
  'message': <wiadomość>,
  'token': 'jwt.token.from.client'
}
```

Działanie: Po odebraniu informacji o wykryciu ruchu, następuje pobranie klatki ze strumienia obrazu nadawanego przez Raspberry o wskazanym numerze seryjnym. Jeżeli na pobranej klatce wykryto człowieka, uruchamiana jest funkcja nagrywająca 30 sekundowy fragment wideo, który zostaje zapisany w bazie danych Firebase Storage. Użytkownik zostaje poinformowany o zajściu zdarzenia z informacją zawartą w polu 'message'. Notyfikacja zostanie wysłana jeżeli wykrycie ruchu zostało spowodowane przez człowieka. W każdym innym przypadku zostanie zignorowana.

c) Wykrycie zmian na czujniku:

Adres: /backend/v1/notification

Zawartość:

```
{
  'serial': <serial-urządzenia>,
  'sensorType': <typ-czujnika>,
  'value': <wartość>,
  'token': 'jwt.token.from.client'
}
```

Działanie: Informuje serwer o wykryciu zagrożenia na jednym z czujników. Serwer następnie wyszukuje wszystkich klientów, którzy posiadają urządzenie o podanym numerze seryjnym, który wykrył niebezpieczne wskazania na czujniku i wysyła do nich powiadomienie za pomocą push notyfikacji.

Następne zapytania dotyczą polecień wysyłanych z aplikacji użytkownika.

d) Pobranie urządzeń użytkownika:

Adres: /backend/v1/get

Zawartość:

```
{
  'owner': <użytkownik>,
  'token': 'jwt.token.from.client'
}
```

Działanie: Zwraca listę urządzeń użytkownika.

e) Zmiana nazwy urządzenia:

Adres: /backend/v1/devices/changeRaspName

Zawartość:

```
{
  'serial': <serial-urządzenia>,
  'name': <nowa-nazwa>,
  'token': 'jwt.token.from.client'
}
```

Działanie: Zmienia nazwę urządzenia, wyświetlana w aplikacji użytkownika. Przyjęto, że nazwa ta powinna oznaczać miejsce, w którym znajduje się urządzenie.

f) Uzbrojenie/rozbrojenie urządzenia:

Adres: /backend/v1/devices/changeIsArmed

Zawartość:

```
{
  'serial': <serial-urządzenia>,
  'armed': <nowy-stan>,
  'token': 'jwt.token.from.client'
}
```

Działanie: Ustala czy nowe powiadomienia związane z urządzeniem dalej będą wysyłane do aplikacji.

g) Pobranie listy notyfikacji:

Adres: /backend/v1/devices/getNotifications

Zawartość:

```
{  
  'serial': <serial-urządzenia>,  
  'token': 'jwt.token.from.client'  
}
```

Działanie: Pobiera listę notyfikacji (historię zdarzeń) powiązanych z urządzeniem o podanym numerze seryjnym.

h) Powiązanie aplikacji z kontem użytkownika:

Adres: /backend/v1/devices/fcmTokenUpdate

Zawartość:

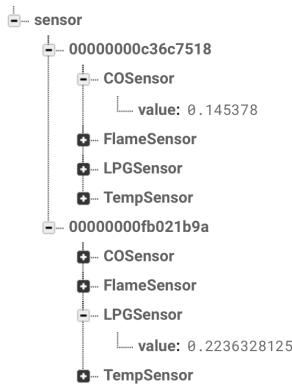
```
{  
  'email': <użytkownik>,  
  'fcmToken': <token-z-firebase>,  
  'deviceId' : <id_aplikacji>  
}
```

Działanie: Powiązuje aplikację mobilną o podanym tokenie z kontem użytkownika. Dzięki temu, notyfikacje trafiają na wszystkie urządzenia użytkownika.

6.4 Baza danych

System operuje na dwóch oddzielnych bazach danych. Pierwsza z nich zawiera wyłącznie dane z czujników pomiarowych, druga całą resztę. Dane z czujników pomiarowych aktualizowane są bardzo często i wymagają natychmiastowego asynchronicznego przesłania tej informacji do klientów. Było to powodem przeniesienia tej tabeli do zewnętrznego systemu, który posiada już takie rozwiązania. Zdecydowano się na usługę Firebase, ponieważ oferuje właśnie te dwie bardzo ważne funkcjonalności. Dane niewymagające częstych zmian znajdują się w bazie danych Django.

Baza danych Firebase Baza danych firebase opiera się na strukturze JSON czyli na elementach klucz-wartość. Nie istnieje tu podział na kolumny i wiersze jak w przypadku standardowej relacyjnej bazy danych. Dane z czujników zapisywane są wewnątrz elementu nadzawanego w strukturze, reprezentującego numer seryjny urządzenia Raspberry Pi, z którego pochodzą dane (rys. 6.1). W polu 'value' znajdują się wartości zmierzone na konkretnym czujniku. Za aktualizację tej struktury odpowiada oprogramowanie uruchomione na Raspberry.



Rysunek 6.1: Struktura bazy danych Firebase [opracowanie własne]

Baza danych Django została zaimplementowana w SQLite3 i poza specyfcznymi dla biblioteki tabelami zawiera:

- **backend_fcmtokens** przechowuje trzy wartości: aktualny token, id urządzenia oraz nazwę konta (e-mail). Na podstawie tokenu zachodzi autoryzacja urządzeń mobilnych.
- **backend_notification** tabela ta reprezentuje historię zdarzeń w systemie. Pojedynczy rekord zawiera: rodzaj zagrożenia, informacje z którego urządzenia pochodzą, datę, wiadomość przeslaną razem z notyfikacją oraz adres url do nagranego wideo (jeśli takie zostało dołączone do wiadomości).
- **backend_rasps** zawiera informację nt. urządzenia: jego numer seryjny, nazwę, konto do którego jest przypisane, stan (czy urządzenie przesyła notyfikacje).
- **users_user** - jest to tabela zawierająca dane użytkowników logujących się w aplikacji internetowej (w szczególności ich login i hasło, identyczny z tymi przechowywanymi w usłudze Firebase Auth).

Rozdział 7

Aplikacje klienckie

Na podstawie analizy statystyk dotyczących podziału rynku aplikacji na platformy[22], zdecydowano się na stworzenie 3 klientów systemu The Guard, które pozwolą możliwie największej grupie osób na korzystanie z systemu:

- aplikacja mobilna na system Android,
- aplikacja mobilna na system iOS,
- aplikacja internetowa.

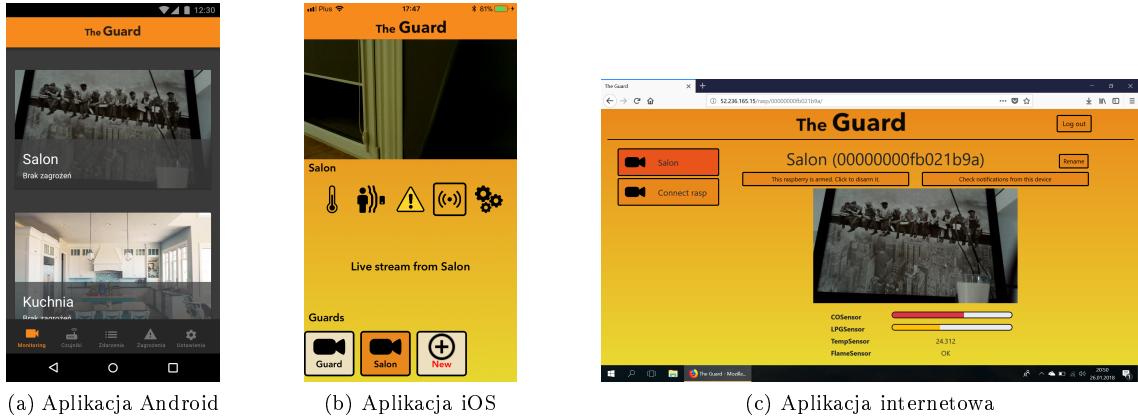
7.1 Funkcje aplikacji

Logowanie Do prawidłowego przejścia do ekranu głównego aplikacji niezbędne jest posiadanie konta. Po rejestracji użytkownika lub po pomyślnym uwierzytelnieniu jeśli konto zostało już wcześniej założone następuje pobranie wszystkich danych użytkownika, jego podłączonych urządzeń i przejście do głównego panelu z dostępem do wszystkich niżej omówionych funkcji.



Rysunek 7.1: Widok logowania [opracowanie własne]

Monitoring Aplikacja pobiera i wyświetla obraz na żywo z zaznaczonego urządzenia podłączonego do konta użytkownika.



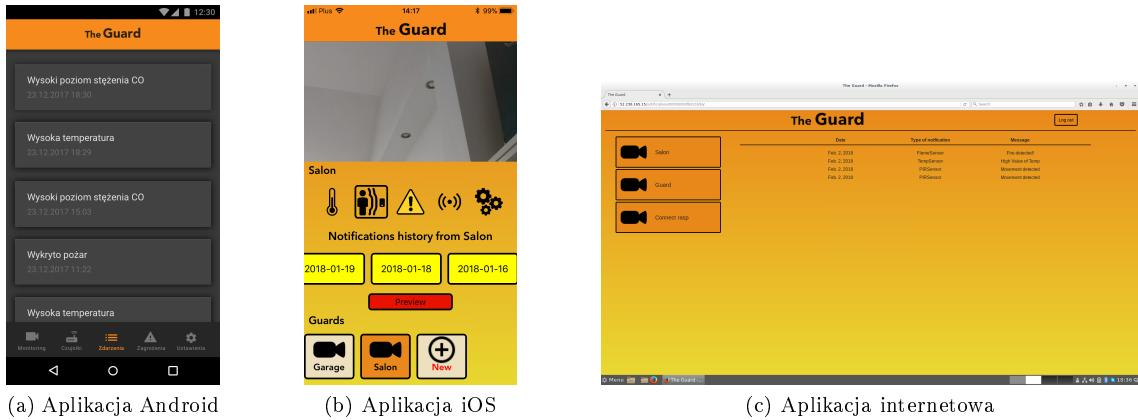
Rysunek 7.2: Widok monitoringui [opracowanie własne]

Status czujników Po przejściu do tej sekcji użytkownik otrzymuje bieżące dane z wszystkich czujników z zaznaczonego urządzenia. Na podstawie koloru prezentowanej wartości z czujnika użytkownik może analizować zagrożenie. Kolor zielony reprezentuje bezpieczne odczyty na czujnikach, kolor pomarańczowy średnie, kolor czerwony natomiast oznacza bardzo wysoki poziom niebezpieczeństwa.



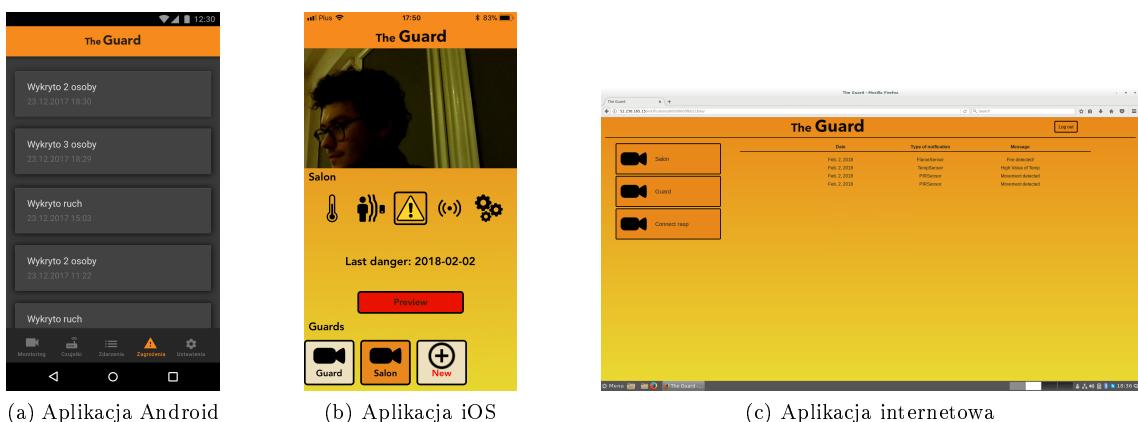
Rysunek 7.3: Widok czujników [opracowanie własne]

Dziennik zdarzeń W tej sekcji użytkownik ma dostęp do historii zdarzeń w systemie.



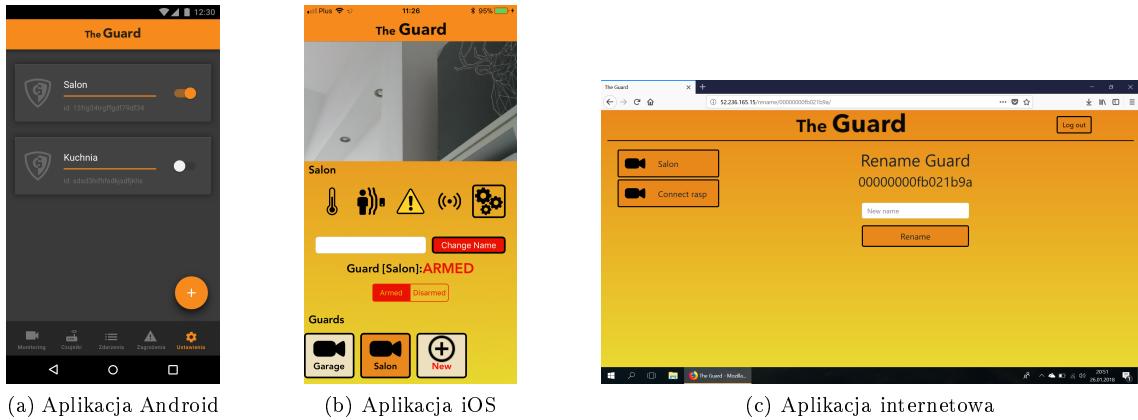
Rysunek 7.4: Widok zdarzeń [opracowanie własne]

Ostatnie zagrożenia Prezentacja ostatniego nagranego zagrożenia. Służy do szybkiego przejrzenia ostatniego niebezpieczeństwa i prezentuje ostatni nagrany materiał video.



Rysunek 7.5: Widok zagrożeń [opracowanie własne]

Ustawienia urządzenia Użytkownik ma możliwość zmiany nazwy urządzenia, które zazwyczaj reprezentuje miejsce, w którym się znajduje. Istnieje również możliwość uzbrojenia i wyłączenia każdego urządzenia. Sprowadza się to do tego, że w przypadku zaznaczenia opcji "Disarmed" użytkownik nie otrzyma kolejnych notyfikacji o zagrożeniach. Opcja ta może okazać się przydatna w momencie uszkodzenia któregoś z modułów i tym samym błędnych danych wysyłanych z czujników.



Rysunek 7.6: Widok ustawień [opracowanie własne]

7.2 Aplikacja Android

Wybór narzędzi Do stworzenia aplikacji mobilnej na system Android użyto języka Kotlin - języka stworzonego przez firmę JetBrains, który 17 maja 2017 roku został uznany przez Google jako oficjalny język programowania aplikacji na platformę Android.[23] Kotlin ściśle współpracuje z kodem stworzonym w Javie i w przypadku Androida jest kompilowany do kodu JVM.[24]

Skorzystano ze środowiska Android Studio w wersji 3.0.1, do automatyzacji budowy projektu został wykorzystany Gradle w wersji 4.1.

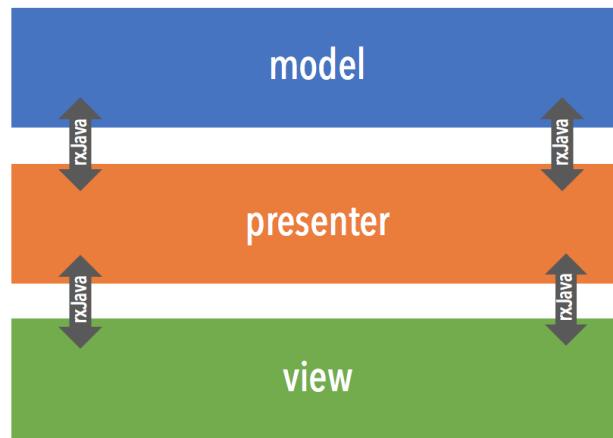
Aplikacja skierowana jest na urządzenia z systemem Android od wersji Lollipop 5.0 (o numerze SDK większym niż 20), który został wydany 12.12.2014 r. Ograniczenie wersji spowodowane jest możliwością użycia bardziej zaawansowanych komponentów, niedostępnych dla niższych wersji. W styczniu 2018 r. oficjalne statystyki informują o tym, że około 80,7 % wszystkich urządzeń z systemem Android na świecie ma wersję 5.0 lub wyższą. [25]

Architektura Aplikacja The Guard dla systemu Android została stworzona zgodnie z założeniami architektury Model View Presenter. Architektura MVP zakłada rozdzielenie kodu źródłowego aplikacji na 3 kategorie.

- Model - kod odpowiedzialny za logikę biznesową, połączenie z serwerem i złożone operacje biznesowe.
- View - kod odpowiedzialny wyłącznie za poprawne wyświetlanie przygotowanych informacji.
- Presenter - kod odpowiedzialny za przygotowanie informacji otrzymanych z warstwy model do wyświetlania w warstwie View.

Największą zaletą architektury MVP jest możliwość wygodnego testowania logiki aplikacji (w warstwie Presenter) oraz zastosowanie programowania reaktywnego przy użyciu biblioteki RxKotlin.

Warstwy komunikują się między sobą w sposób reaktywny - przy użyciu strumieni wydarzeń. Przykładowo klasa warstwy Presenter odpowiedzialna za wyświetlanie obrazu z kamery wykorzystuje klasę warstwy Model do asynchronicznej komunikacji z API.[26]



Rysunek 7.7: Struktura MVP [opracowanie własne]

Interfejs użytkownika Aplikacja została zaprojektowana zgodnie z wytycznymi Material Design. [27] Do nawigacji po funkcjach aplikacji służy panel na dole ekranu - "Bottom Bar". Zanim będzie on widoczny, użytkownik musi najpierw zalogować się (lub zarejestrować) przy użyciu adresu email oraz hasła.

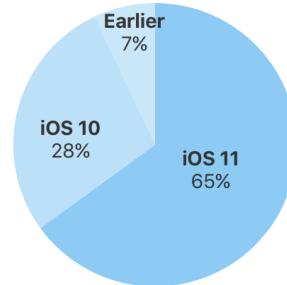
Wykorzystane biblioteki

- **com.squareup.retrofit2** - biblioteka służąca do wygodnej komunikacji z API,
- **io.reactivex:rxandroid** - biblioteka zapewniająca *Scheduler* platformy Android dla kodu reaktywnego,
- **com.jakewharton.rxbinding** - biblioteka zapewniająca nakładki na widoki Androida generujące zdarzenia dla programowania reaktywnego,
- **com.github.zurche:plain-pie** - biblioteka wyświetlająca diagramy na podstawie danych z czujników,
- **com.github.jacek-marchwicki.recyclerview-changes-detector** - biblioteka umożliwiająca automatyczną zmianę widoków po detekcji zmiany zbioru wyświetlnych danych,
- **com.google.code.gson** - biblioteka umożliwiająca serializację i deserializację obiektów,
- **com.google.firebaseio** - biblioteki służące do obsługi usług Firebase.

Programowanie reaktywne Aktualizacja danych z czujników, obsługa interfejsu użytkownika, zapytania API oraz komunikacja z usługami Firebase zrealizowane są zgodnie z paradygmatami programowania reaktywnego. *Observables* tworzą zdarzenia, które mogą zostać modyfikowane przez liczne operatory, a następnie powodować reakcje, które zostały do nich *zasubskrybowane*.

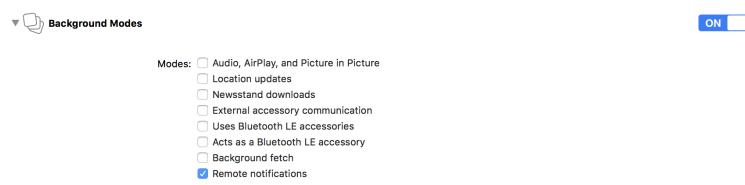
7.3 Aplikacja iOS

Aplikacja przeznaczona jest na urządzenia z systemem operacyjnym iOS od wersji 10.0. Nie wspiera ona wcześniejszych wersji ze względu na nowe funkcje, które Apple wprowadziło wraz z pojawieniem się iOS 10.0 (m.in. klasa UNUserNotificationCenter). Jednak 93% wszystkich obecnych użytkowników tego systemu (rys. 5.1) jest w stanie zainstalować oprogramowanie a liczba ta stale rośnie. Aplikacja wspiera zarówno telefony komórkowe iPhone jak i tablety iPad.



Rysunek 7.8: Udziały wersji systemu iOS z 18.01.2018 [28]

Napisana w stosunkowo nowym języku Swift (zaprezentowany przez Apple w 2014 roku) w oparciu o architekturę MVC (Model-View-Controller) wykorzystując przy tym programowanie reaktywne i funkcyjonalne. Aplikacja powstała w programie Xcode. Programowanie reaktywne zrealizowano przy pomocy biblioteki RxSwift. Ten paradygmat programowania związany jest z pojęciem obserwatora i sekwencji obserwowań. Każdy obserwator wywołując funkcję 'subscribe' na elemencie obserwowanym otrzymuje informację o każdej zmianie na tym obiekcie. RxSwift wykorzystano m.in. w celu wznowienia streamu obrazu z kamery w momencie przejścia aplikacji z trybu pracy w tle do trybu aktywnego. Oznacza to, że po wyjściu z aplikacji i po ponownym jej uruchomieniu tracono obraz ze streamu. Przyczyną jest polityka Apple, która nie zaleca aby aplikacje pracowały w tle i domyślnie wyłącza każdą taką aktywność. Ma to na celu przedłużenie żywotności baterii i optymalizacji całego systemu poprzez ograniczenie ilości zajmowanych zasobów [29]. Oczywiście istnieje możliwość włączenia pracy w tle, jednakże konieczne jest aktywowanie trybu "Background Modes" i zaznaczenie konkretnej aktywności, którą chcielibyśmy wykonywać. Lista dozwolonych czynności możliwych do realizacji jest jednak ograniczona (rys. 5.2).



Rysunek 7.9: Tryby pracy w tle [opracowanie własne]

Próba oszustwa i wykonywania innej pracy w tle niż zaznaczona zostanie wychwycona w procesie weryfikacji przed jej publikacją na platformie Apple Store. Dzięki programowaniu reaktywnemu problem wznowienia podglądu obrazu został rozwiązany co prezentuje poniższy kod:

```
let appDelegate = UIApplication.shared.delegate as! AppDelegate
appDelegate.inBackground.asObservable().subscribe(onNext: { (value) in
    if let streamView = self.streamView {
        if let player = self.currentPlayer {
```

```

        if value == false {
            self.streamVideoFrom(urlString: self.currentUrlString!)
            print("Enter foreground")
        } else {
            print("Enter background")
            streamView.layer.sublayers?.forEach({ (layer) in
                layer.removeFromSuperlayer()
            })
        }
    }
}).disposed(by: disposeBag)

```

Zmienna 'inBackground', która jest zmienną obserwowlaną, ustawiana jest w oddzielnej klasie AppDelegate (klasa, która zapewnia poprawną interakcję z systemem iOS) na wartość true w chwili przejścia do trybu pracy w tle i na wartość false w przeciwnym wypadku. Klasa, w której wywoływany jest funkcja 'subscribe' jest obserwatorem tej zmiennej. Kod wewnątrz funkcji subscribe uruchamiany jest przy każdej zmianie wartości 'inBackground' i wznowia ponownie stream po każdym ponownym uruchomieniu programu. 'Programowanie funkcyjonalne natomiast polega na traktowaniu funkcji jako obiektu. Oznacza to, że mogą być one zapisywane, kopowane i przekazywane tak samo jak wszystkie inne obiekty. Mogą być używane jako parametry innych funkcji.' [30, p. 172]. Wykorzystane są w miejscach gdzie konieczne jest przekształcanie danych:

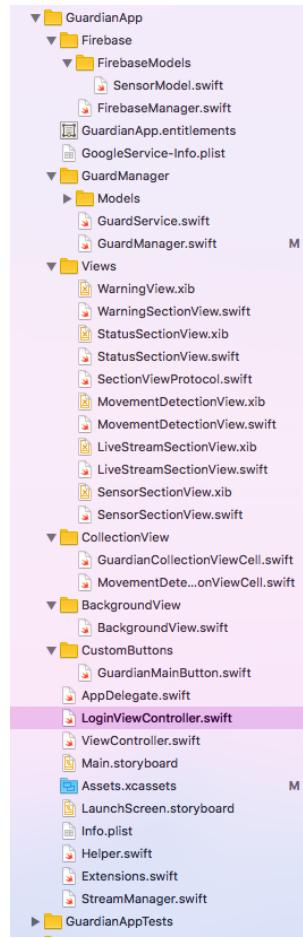
```

lastNotification = notifications.array.sorted(by: { (n1, n2) -> Bool in
    n1.date > n2.date
}).filter({ (notif) -> Bool in return notif.type == "PIRSensor"}).first

```

Na tablicy z notyfikcjami zastosowano szereg kolejnych funkcji: posortowano je malejąco według daty, przefiltrowano w taki sposób aby wybrać tylko te o typie 'PIRSensor' czyli te pochodzące z czujnika ruchu. Na sam koniec wybrano tylko jeden pierwszy element z wybranych i wynik wpisano do zmiennej lastNotification. Każda kolejna wywoływana funkcja np. filter, odbiera wynik poprzedniej.

Strukturę kodu (rys. 7.10) podzielono na kilka osobnych, logicznych części. Folder Firebase zawiera model bazy danych czujników, które zapisane są na serwerach Firebase. W folderze GuardManager znajdują się elementy odpowiedzialne za komunikację REST-ową z serwerem Django i modele bazy danych znajdującej się na naszym serwerze. Folder Views jest zbiorem widoków, które wczytywane są w zależności, w której sekcji się znajdujemy (opis sekcji niżej). ViewController.swift jest głównym kontrolerem zarządzającym widokami i modelami. Odpowiada za załadowanie odpowiedniego widoku i prezentację danych z odpowiedniej sekcji. W folderze GuardianAppTests napisane zostały testy jednostkowe, które sprawdzają poprawność przekształcania danych typu JSON (odpowiedź serwera) do obiektów zdefiniowanych w folderze GuardManager/Models. Klasy, których nazwy kończą się na Manager oznaczają obiekty typu Singleton. Celem takiego wzorca jest zapewnienie istnienia tylko jednej instancji w całej aplikacji i globalnego dostępu do tego obiektu. GuardManager, który odpowiada za pobieranie danych z bazy danych - taki obiekt nie powinien być utworzony więcej niż jeden raz, gdyż wszystkie klasy, które z niego korzystają nie potrzebują kolejnych instancji tej klasy. W ten sposób zapewniono, że zawsze odwołujemy się do tego samego obiektu.



Rysunek 7.10: Struktura aplikacji [opracowanie własne]

Instalacja zewnętrznych bibliotek odbywa się za pomocą CocoaPods. Jest to menadżer zależności dzięki któremu szybko możemy wyszukać i zainstalować wymagane oprogramowanie. Wszystkie użyte zależności przedstawiono poniżej:

```
pod 'Moya'
pod 'MBProgressHUD', '~> 1.0'
pod 'RxSwift',     '~> 4.0'
pod 'RxCocoa',     '~> 4.0'
pod 'IHKeyboardAvoiding'
pod 'Moya-SwiftyJSONMapper'
pod 'Firebase/Core'
pod 'Firebase/Messaging'
pod 'Firebase/Auth'
pod 'Firebase/Database'
pod 'M13ProgressSuite'
```

Moya używana jest do asynchronicznej REST-owej komunikacji z serwerem Django. SwiftyJSONMapper przydatna okazuje się do przekształcenia odpowiedzi serwera w postaci JSON do wcześniej zdefiniowanego modelu. MBProgressHUD umożliwia wyświetlanie ekranu ładowania podczas pobierania informacji z serwera. RxSwift i RxCocoa to biblioteki do programowania reaktywnego. Moduły Firebase itp. służą do komunikacji z serwerami Firebase. Ostatni 'pod M13ProgressSuite' wykorzystano do rysowania wykresów i animowanych elementów graficznych w systemie iOS. Po

uruchomieniu aplikacji pierwszym widokiem jest ekran logowania i rejestracji użytkowników (rys 6.1b). Po prawidłowym uwierzytelnieniu użytkownika uzyskiwany jest dostęp do głównego widoku aplikacji. W górnej części możliwy jest wybór 5 sekcji: sekcja czujników, sekcja historii notyfikacji, sekcja ostatnich zagrożeń przy wykryciu ruchu, sekcja monitoringu na żywo, sekcja ustawień. Wszystkie te sekcje dotyczą konkretnego urządzenia wybranego na pasku u dołu ekranu. Funkcje każdej z nich zostały opisane w rozdziale 6.1, tutaj zostaną zaprezentowane jedynie szczegółowe implementacyjne i rzuty ekranów z wersji na iOS. Przy pierwszym uruchomieniu nie istnieje żadne urządzenie przypisane do naszego konta użytkownika. Aby dodać pierwsze i kolejne stacje, od których chcemy otrzymywać notyfikacje o zagrożeniach a także śledzić i monitorować informacje z czujników należy wybrać przycisk 'New' plusikiem w dolnej części ekranu. Pojawi się okno z prośbą o wpisanie numeru identyfikującego urządzenie. Po chwili dodany 'Guard' będzie widoczny w na liście.

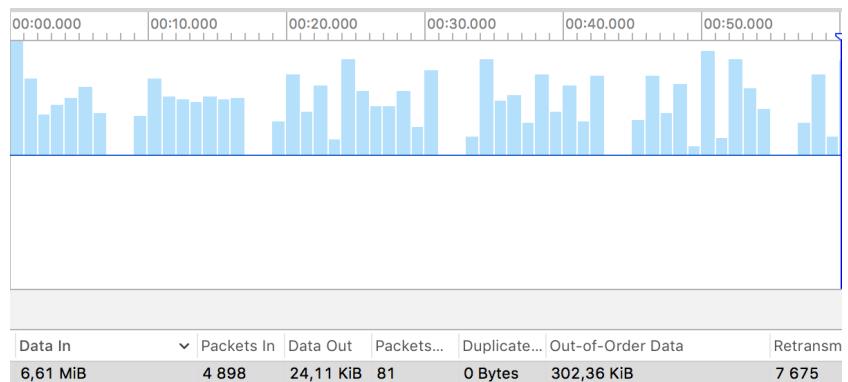
Sekcja czujników: Jest to jedna z najważniejszych sekcji aplikacji (rys 6.3b). Otrzymuje ona dane z czujników w czasie rzeczywistym i prezentuje je użytkownikowi. Implementacja funkcjonalności prezentowania zagrożenia na konkretnym czujniku przy użyciu kolorów zrealizowana została przy pomocy modelu HSV, który w przeciwieństwie do RGB pozwala na bardzo proste przejście z jednego koloru do kolejnego poprzez zmianę tylko jednego parametru. Zmieniając parametr Hue zmieniamy barwę przy stałym nasyceniu i jasności. Wartość tego parametru równa 120° odpowiada kolorowi zielonemu, kolor czerwony to 0° . Przekształcając wartość otrzymaną z czujników, która jest z zakresu [0-1] na wartość z przedziału [120-0] otrzymano wspomniany efekt. Poniżej przedstawiono fragment konwersji danych z czujników na kolor w modelu HSV, gdzie zmienna sensors[0] reprezentuje czujnik LPG.

```
UIColor(hue: CGFloat(0.33 - (sensors[0].value * 0.33)),  
saturation: 1, brightness: 1, alpha: 1)
```

Sekcja historii notyfikacji: Po zaznaczeniu daty reprezentującej moment wystąpienia zagrożenia i wybraniu przycisku 'preview' prezentowana jest informacja o miejscu niebezpieczeństwa i jego rodzaju. (rys 6.4b).

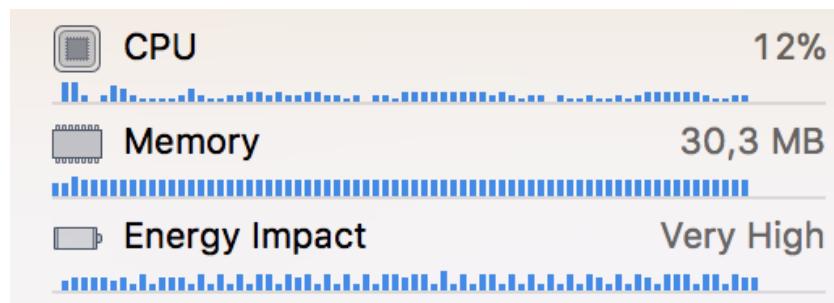
Sekcja monitoringu: Sekcja odpowiedzialna za prawidłowy odbiór obrazu z kamery zaznaczonej w dolnej części ekranu (rys. 6.2b). Okno, w którym odbywa się transmisja ustawiono w taki sposób, aby bez względu na rozmiar telefonu utrzymywało proporcję 16:9. Pozbyto się dzięki temu czarnych ramek lub braku części transmitowanego obrazu. W tym samym oknie prezentowane są nagrane materiały filmowe z historii zdarzeń, które zakrywają obraz nadawany na żywo. Przejście do tej sekcji wznawia pobieranie obrazu z urządzeń w czasie rzeczywistym.

Przeprowadzono kilka testów aplikacji pod pełnym obciążeniem za pomocą programu Instruments. Szczególnie interesująco przedstawia się zużycie sieci podczas streamu obrazu. Widać, że w ciągu jednej minuty pobrano 6,61MB a wysłano jedynie 24,11Kb (rys. 7.11). Obraz pobierany jest tylko wtedy kiedy aplikacja jest aktywna. W ciągu godziny działania aplikacji pobierze ona około 400MB danych. Jednak dla zapewnienia komfortu użytkowania i płynnego streamu obrazu zalecane jest posiadanie łącza umożliwiającego transfer danych na poziomie min. 200KB/s.



Rysunek 7.11: Zużycie sieci podczas transmisji obrazu [opracowanie własne]

Przeprowadzono także test na zużycie pamięci RAM i zużycie procesora. Te jednak są niewielkie i wynoszą odpowiednio 25MB pamięci RAM i średnio 1 procent zużycia procesora. Zużycie procesora wzrasta do poziomu ok. 15 procent tylko w momencie pobierania nagranego obrazu z serwera. Wtedy też zużycie pamięci RAM jest o około 5MB większe i wynosi około 30MB (rys. 7.12).



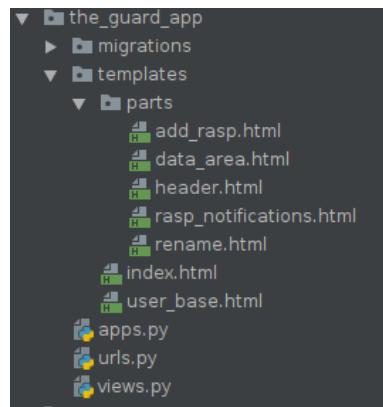
Rysunek 7.12: Zużycie procesora i pamięci RAM przy największym obciążeniu [opracowanie własne]

Testy przeprowadzono na iPhone 6S i iPadzie Pro.

7.4 Aplikacja internetowa

Aplikacja internetowa przeznaczona jest dla użytkowników wszystkich systemów operacyjnych i została przetestowana w przeglądarce Firefox. Rozmieszczenie komponentów aplikacji różni się od tego zastosowanego w aplikacjach IOS oraz Android - spowodowane jest to inną rozdzielczością ekranu.

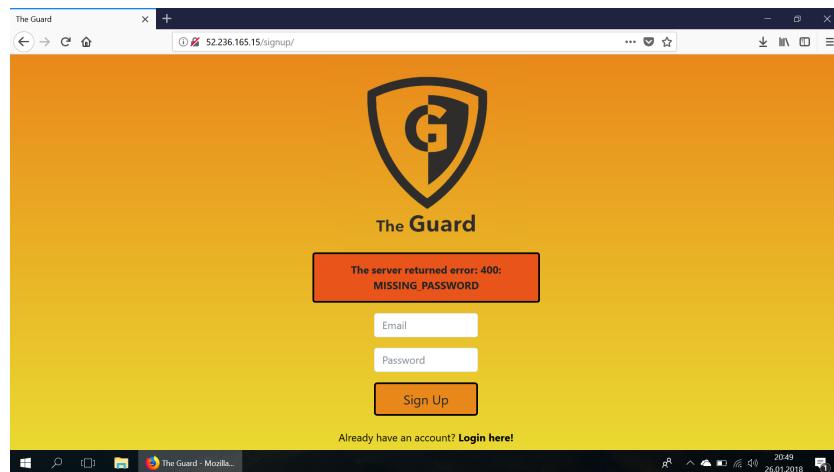
Struktura projektu przedstawiona jest na rysunku 7.13. Folder migrations zawiera pliki tworzone automatycznie przez bibliotekę Django. W folderze templates stworzone są szablony plików html specyficzne dla danych widoków aplikacji. Na uwagę zasługują także pliki urls.py oraz views.py. W pierwszym znajdują się adresy url używane przez aplikację oraz informacja nt. jaki widok powinien być wyrenderowany dla użytkownika w momencie otwarcia przez niego w przeglądarce danego linku. W pliku views.py znajdują się funkcje odpowiedzialne za renderowanie widoków aplikacji oraz łączenie z bazami danych.



Rysunek 7.13: Struktura projektu aplikacji internetowej
[opracowanie własne]

Implementacja Django - połączenie z bazą danych: Biblioteka Django posiada wbudowane rozwiązania umożliwiające pobieranie informacji z bazy danych projektu dzięki czemu aplikacja internetowa nie wysyła zapytań na określone dla aplikacji mobilnych porty, tylko komunikuje się bezpośrednio z bazą danych. Rozwiążanie to umożliwia uniezależnienie aplikacji internetowej od stanu portów oraz zmniejsza liczbę potrzebnych zapytań wysyłanych do serwera. Aplikacja internetowa przeznaczona jest dla użytkowników wszystkich systemów operacyjnych i została przetestowana w przeglądarce Firefox w systemach operacyjnych Microsoft Windows 10 (Firefox 58.0.1) oraz Linux Debian 9 (Firefox ESR 52.5.2 64 bit). Do stworzenia aplikacji użyto języków programowania Python 3, JavaScript oraz framework'u Django, natomiast frontend jest oparty na bibliotece Bootstrap oraz JQuery. Połączenie z bazą danych Firebase zaimplementowano za pomocą Firebase Web Api. Rozmieszczenie komponentów aplikacji różni się od tego zastosowanego w aplikacjach IOS oraz Android - spowodowane jest to inną rozdzielcością ekranu.

Panel logowania / rejestracji użytkownika: Panele logowania oraz rejestracji użytkownika są do siebie bardzo podobne - jedyna ich różnica jest w nazwie i funkcjonalności. Obydwa panele składają się z loga aplikacji oraz formularza w którym trzeba podać adres email i hasło. W przypadku panelu logowania, dane są weryfikowane i jeśli są poprawne użytkownik zostaje zalogowany. Jeżeli użytkownik chce zarejestrować konto, sprawdzana jest poprawność adresu email, a następnie tworzony jest konto w usłudze FireBase Auth. W przypadku błędu, jest on wyświetlany powyżej formularza (rys. 7.14)



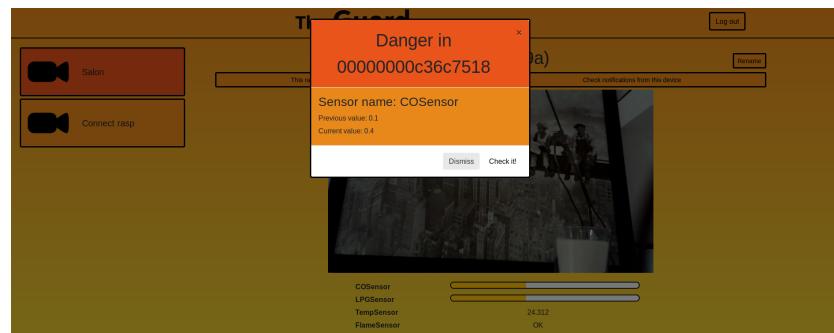
Rysunek 7.14: Widok logowania z przykładowym błędem w aplikacji internetowej [opracowanie własne]

Menu wyboru urządzenia: Po prawidłowym zalogowaniu do aplikacji, użytkownik może zobaczyć listę swoich urządzeń, dodać nowe oraz zaczyna dostawać powiadomienia w razie wykrytego zagrożenia. W przypadku kliknięcia przycisku ‘Connect rasp’, użytkownik zostaje przekierowany do widoku umożliwiającego rejestrację nowego urządzenia. Po wprowadzeniu numeru seryjnego urządzenia oraz jego nazwy, zostaje dodany do baz danych. Po wybraniu urządzenia, informacje nt. jego stanu będą wyświetlane po prawej stronie okna, która w momencie zalogowania jest pusta. Urządzenia w menu są rozpoznawane na podstawie ich nazw.

Widok konkretnego urządzenia: Po wybraniu z menu konkretnego urządzenia, użytkownik zostaje przekierowany na stronę pojedyńczego urządzenia (rys. 7.2). Pod nazwą urządzenia i jego numerem seryjnym wyświetlany jest aktualny obraz z kamery oraz stan czujników. Dzięki zastosowaniu nasłuchiwanego na bazie danych Firebase, zmiany są na bieżąco wyświetlane na stronie. Użytkownik ma możliwość po naciśnięciu odpowiedniego przycisku:

- Zmienić nazwę urządzenia - po kliknięciu na przycisk rename znajdujący się obok nazwy urządzenia, użytkownik zostanie przekierowany do panelu zmiany nazwy (rys. 7.6).
- Wyłączyć / włączyć alerty
- Zobaczyć notyfikacje danego urządzenia - poprzez kliknięcie na przycisk ‘Check notifications from this device’, użytkownik zostanie przekierowany do widoku listy notyfikacji danego urządzenia (rys. 7.4).

Wyświetlanie notyfikacji: W każdym z widoków aplikacji internetowej w czasie rzeczywistym sprawdzane są notyfikacje z bazy danych z pomocą Firebase WebApi. W przypadku wykrycia zmiany uznawanej za niebezpieczną za pomocą skryptów przeglądarki (JavaScript + JQuery) wyświetlany jest monit informujący o niebezpiecznym zdarzeniu, co pokazane jest na rysunku 7.15.



Rysunek 7.15: Informacja o niebezpiecznym zdarzeniu - nagłe zmianie wartości odczytanej przez czujnik [opracowanie własne]

Rozdział 8

Testy funkcjonalne

Test nr 1 Zakres testu:

Użytkownik po odebraniu notyfikacji, ma skontrolować zagrożenie - aplikacja internetowa

Tester: Piotr Falkiewicz

Autor aplikacji: Aleksandra Główczewska

Przebieg testu:

1. Użytkownik otwiera aplikację.
2. Tester odebrał informację o zagrożeniu.
3. Po naciśnięciu przycisku 'Check It!' użytkownik został przeniesiony na podstronę związaną z notyfikacją.

Wniosek:

Funkcjonalność zaimplementowana prawidłowo

Test nr 2 Zakres testu:

Użytkownik dodaje nowe urządzenie do swojego konta - aplikacja iOS

Tester: Aleksandra Główczewska

Autor aplikacji: Paweł Szudrowicz

Przebieg testu:

1. Użytkownik otwiera aplikację.
2. Użytkownik wybiera przycisk 'New' w dolnej części ekranu
3. Użytkownik wprowadza numer seryjny urządzenia, które zamierza dodać
4. Po naciśnięciu przycisku 'Send' użytkownik otrzymał informację o sukcesie.

Wniosek:

Funkcjonalność zaimplementowana prawidłowo

Test nr 3 Zakres testu:

Użytkownik ma otrzymać transmisję obrazu z urządzenia - aplikacja iOS

Tester: Piotr Falkiewicz

Autor aplikacji: Paweł Szudrowicz

Przebieg testu:

1. Użytkownik otwiera aplikację

2. Użytkownik zaznacza urządzenie, z którego zamierza odbierać obraz
3. Po chwili w górnej części ekranu pojawił się transmitowany obraz

Wniosek:

Funkcjonalność zaimplementowana prawidłowo

Test nr 4 Zakres testu:

Użytkownik ma pobrać nagranie z bazy danych z ostatniego zagrożenia - aplikacja iOS

Tester: Mateusz Bartos

Autor aplikacji: Paweł Szudrowicz

Przebieg testu:

1. Użytkownik otwiera aplikacje
2. Użytkownik zaznacza urządzenie, z którego zamierza pobrać materiał
3. Użytkownik przechodzi do sekcji 'ostatnie zagrożenie'
4. Użytkownik wybiera przycisk 'preview'
5. Na ekranie pojawia się ekran ładowania
6. Po chwili w górnej części ekranu pojawiło się pobrane nagranie

Wniosek:

Funkcjonalność zaimplementowana prawidłowo

Test nr 5 Zakres testu:

Użytkownik ma rozbroić urządzenie - aplikacja internetowa

Tester: Piotr Falkiewicz

Autor aplikacji: Aleksandra Główczewska

Przebieg testu:

1. Użytkownik otwiera aplikację
2. Użytkownik odbiera informację o zagrożeniu
3. Użytkownik przechodzi do sekcji urządzenia
4. Użytkownik wybiera przycisk 'Disarm it!'
5. Aplikacja przestaje wyświetlać powiadomienia o zagrożeniu

Wniosek:

Funkcjonalność zaimplementowana prawidłowo

Test nr 6 Zakres testu:

Użytkownik dodaje nowe urządzenie do swojego konta - aplikacja Android

Tester: Paweł Szudrowicz

Autor aplikacji: Mateusz Bartos

Przebieg testu:

1. Użytkownik otwiera aplikację.
2. Użytkownik wybiera sekcję ustawienia korzystając z przycisków na dole ekranu
3. Użytkownik wprowadza numer seryjny urządzenia, które zamierza dodać
4. Użytkownik widzi dodane urządzenie w pozostałych sekcjach aplikacji

Wniosek:

Funkcjonalność zaimplementowana prawidłowo

Test nr 7 Zakres testu:

Użytkownik może sprawić status czujników - aplikacja Android

Tester: Paweł Szudrowicz

Autor aplikacji: Mateusz Bartos

Przebieg testu:

1. Użytkownik otwiera aplikacje
2. Użytkownik wybiera sekcję "Czujniki"
3. Użytkownik znajduje dane z czujników w wybranych pomieszczeniach

Wniosek:

Funkcjonalność zaimplementowana prawidłowo

Rozdział 9

Uwagi końcowe

Zrealizowano wszystkie cele postawione sobie na początku pracy. Wykonany system działa i jest przystosowany do dalszej modyfikacji. Istnieje możliwość wymiany czujników z rodziny MQ na inne bez konieczności zmian w oprogramowaniu uruchomionym na Raspberry, ze względu na podobny charakter ich działania. W następnej fazie zalecane byłoby zaprojektowanie nowej obudowy na czujniki aby poprawić wygląd urządzeń. Zdecydowano, że cały system będzie na licencji open-source, aby każdy mógł pobrać oprogramowanie i edytować je według własnych potrzeb. Mimo, że taktowanie procesora Raspberry Pi 3 nie jest wysokie i nie poradziłby on sobie sam z zadaniami przedstawionymi na początku pracy, to dzięki zastosowaniu zewnętrznych serwerów i usług na platformie Microstof Azure udało się wykonać w pełni działający system bezpieczeństwa. Przeniesiono bardzo obciążające zadania, takie jak przetwarzanie obrazu, na zewnętrzne platformy, które wyposażone są w znacznie bardziej zaawansowane podzespoły. Wykorzystanie natomiast usług Firebase, z którego korzystają też takie firmy jak Trivago czy Shazam, pozwoliło na szybką implementację zaawansowanych funkcji m.in uwierzytelnianie użytkowników czy aktualizację zmian z bazy danych w czasie rzeczywistym. Konieczne jest dalsze rozwijanie projektu w przyszłości. Wykorzystane rozwiązania pozwalają m. in. na zbieranie większej ilości danych, co z kolei generuje kolejne wyzwania, z którymi będzie należało się zmierzyć.

Literatura

- [1] Dokumentacja django-rest-framework. <http://www.django-rest-framework.org/>. [Dostęp: 15.01.2018].
- [2] Richard N. Taylor Roy T. Fielding. *Principled Design of the Modern Web Architecture*. Information and Computer Science University of California, Irvine, 2002.
- [3] Dokumentacja systemu ochrony tokenem csrf w django. [://docs.djangoproject.com/en/2.0/ref/csrf/](http://docs.djangoproject.com/en/2.0/ref/csrf/). [Dostęp: 15.01.2018].
- [4] Specyfikacja czujnika mq-9. <https://botland.com.pl/czujniki-gazu/3029-czujnik-tlenku-wegla-i-latwopalnych-gazow-mq-9-modul-niebieski.html>. [Dostęp: 15.01.2018].
- [5] Specyfikacja czujnika mq-2. <https://botland.com.pl/czujniki-gazu/5521-czujnik-lpg-propanu-i-wodoru-mq-2-modul-waveshare.html>. [Dostęp: 15.01.2018].
- [6] Specyfikacja czujnika ds18b20+. <https://botland.com.pl/czujniki-temperatury/1506-modul-z-czujnikiem-temperatury-ds18b20-.html>. [Dostęp: 15.01.2018].
- [7] Specyfikacja pir-hc-sr501. <https://botland.com.pl/czujniki-ruchu/1655-czujnik-ruchu-pir-hc-sr501-zielony.html>. [Dostęp: 15.01.2018].
- [8] Specyfikacja czujnika płomieni. <https://botland.com.pl/czujniki-temperatury/4461-czujnik-plomieni-760-1100nm-analogowy.html>. [Dostęp: 15.01.2018].
- [9] Specyfikacja przetwornika a/c. <https://botland.com.pl/przetworniki/2358-przetwornik-ac-mcp3008-ip-10-bitowy-8-kanalowy-spi-dip.html>. [Dostęp: 15.01.2018].
- [10] Specyfikacja raspberry pi 3 model b. <https://www.raspberrypi.org/products/raspberry-pi-3-model-b>. [Dostęp: 15.01.2018].
- [11] element14.com. Gpio. <https://www.element14.com/community/docs/DOC-73950/1/raspberry-pi-3-model-b-gpio-40-pin-block-pinout>. [Dostęp: 15.01.2018].
- [12] 14core.com. Wiringpi. <https://www.14core.com/configure-clibrary-wiringpi/>. [Dostęp: 15.01.2018].
- [13] Dokumentacja mq-2. <https://www.pololu.com/file/0J309/MQ2.pdf>. [Dostęp: 15.01.2018].
- [14] Dokumentacja mq-9. <http://www.haoyuelectronics.com/Attachment/MQ-9/MQ9.pdf>. [Dostęp: 15.01.2018].

- [15] Wikipedia, wolna encyklopedia. Serial peripheral interface. https://pl.wikipedia.org/wiki/Serial_Peripheral_Interface. [Dostęp: 15.01.2018].
- [16] Kod oraz opis biblioteki spidev. <https://github.com/doceme/py-spidev>. [Dostęp: 15.01.2018].
- [17] Jianxiao Xi Xuerong Gou Pengyu Zhao, Jianwei Li. *A Mobile Real-time Video System Using RTMP*. 4th International Conference on Computational Intelligence and Communication Networks, 2012.
- [18] Caihong Wang Xiaohua Lei, Xiuhua Jiang. *Design and Implementation of Streaming Media Processing Software based on RTMP*. 5th International Congress on Image and Signal Processing, 2012.
- [19] Yakubu S. Baguda. *H264/AVC features & functionalities suitable for wireless video transmission*. IEEE 2008 IFIP International Conference on Wireless and Optical Communications Networks, 2008.
- [20] Dokumentacja raspivid. <https://www.raspberrypi.org/documentation/usage/camera/raspicam/raspivid.md>. [Dostęp: 15.01.2018].
- [21] Dokumentacja nginx-rtmp. <https://github.com/arut/nginx-rtmp-module/>. [Dostęp: 15.01.2018].
- [22] Statcounter.com. Udział systemów w rynku sieciowego ruchu mobilnego. <http://gs.statcounter.com/os-market-share/mobile-tablet/worldwide/#monthly-201801-201801-bar>. [Dostęp: 15.01.2018].
- [23] Google. Oficjalny komunikat google nt wsparcia języka kotlin. <https://twitter.com/Android/status/864911929143197696>. [Dostęp: 15.01.2018].
- [24] Igor Wojda Marcin Moskala. *Android Development with Kotlin*. Packt Publishing, 2017.
- [25] Google. Dokumentacja platformy android. <https://developer.android.com>. [Dostęp: 15.01.2018].
- [26] Ben Christensen Tomasz Nurkiewicz. *Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications*. O'Reilly Media, 2016.
- [27] Kyle Mew. *Learning Material Design: Master Material Design and create beautiful, animated interfaces for mobile and web applications*. Packt Publishing, 2015.
- [28] Procentowy udział wersji ios w rynku. <https://developer.apple.com/support/app-store/>. [Dostęp: 31.01.2018].
- [29] Apple. Background execution. <https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html>. [Dostęp: 15.01.2018].
- [30] Paul Hudson. Pro swift. <https://itunes.apple.com/us/book/pro-swift/id1111033310?mt=11>, 2016.

Spis rysunków

3.1 Schemat systemu [opracowanie własne]	4
3.2 Funkcja obsługująca dodanie nowego urządzenia [opracowanie własne]	5
5.1 Zbudowany zestaw The Guard [opracowanie własne]	9
5.2 Schemat układu The Guard [opracowanie własne]	10
5.3 Ustawienia [opracowanie własne]	11
5.4 Porównanie pinów Raspberry Pi 3 [11] z pinami wiringPi [12]	11
5.5 Charakterystyka czujników MQ-2 oraz MQ-9.	13
5.6 Interfejs SPI [15]	13
6.1 Struktura bazy danych Firebase [opracowanie własne]	22
7.1 Widok logowania [opracowanie własne]	23
7.2 Widok monitoringui [opracowanie własne]	24
7.3 Widok czujnikowi [opracowanie własne]	24
7.4 Widok zdarzeń [opracowanie własne]	25
7.5 Widok zagrożeń [opracowanie własne]	25
7.6 Widok ustawień [opracowanie własne]	26
7.7 Struktura MVP [opracowanie własne]	27
7.8 Udziały wersji systemu iOS z 18.01.2018 [28]	28
7.9 Tryby pracy w tle [opracowanie własne]	28
7.10 Struktura aplikacji [opracowanie własne]	30
7.11 Zużycie sieci podczas transmisji obrazu [opracowanie własne]	32
7.12 Zużycie procesora i pamięci RAM przy największym obciążeniu [opracowanie własne]	32
7.13 Struktura projektu aplikacji internetowej [opracowanie własne]	33
7.14 Widok logowania z przykładowym błędem w aplikacji internetowej [opracowanie własne]	34
7.15 Informacja o niebezpiecznym zdarzeniu - nagłe zmianie wartości odczytanej przez czuj- nik [opracowanie własne]	35



© 2018 Mateusz Bartos, Piotr Falkiewicz, Aleksandra Głowczewska, Paweł Szudrowicz

Instytut Informatyki, Wydział Informatyki
Politechnika Poznańska

Skład przy użyciu systemu L^AT_EX.

BibT_EX:

```
@masterthesis{ key,  
    author = "Mateusz Bartos \and Piotr Falkiewicz \and Aleksandra Głowczewska \and Paweł  
Szudrowicz",  
    title = "{System kontroli bezpieczeństwa - The Guard}",  
    school = "Poznan University of Technology",  
    address = "Poznań\n, Poland",  
    year = "2018",  
}
```