

Politechnika Poznańska
Wydział Informatyki
Instytut Informatyki

Praca dyplomowa inżynierska

SYSTEM KONTROLI BEZPIECZEŃSTWA – THE GUARD

Mateusz Bartos, 122437
Piotr Falkiewicz, 122537
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. [4]

Spis treści

1	Wstęp	1
2	Architektura systemu	3
2.1	Schemat	3
2.2	Komunikacja	3
2.3	Bezpieczeństwo	6
3	Zbieranie i przetwarzanie danych z czujników	7
3.1	Raspberry Pi	7
3.2	Czujniki	9
3.3	Obsługa wideo	13
4	Rozwiązania chmurowe	17
4.1	Microsoft Azure	17
4.2	Firebase	17
4.3	Aplikacja serwerowa	17
4.4	Baza danych	17
5	Aplikacje klienckie	18
5.1	Funkcje aplikacji	18
5.1.1	Logowanie	18
5.1.2	Monitoring	18
5.1.3	Status czujników	18
5.1.4	Dziennik zdarzeń	18
5.1.5	Ostatnie zagrożenia	18
5.1.6	Ustawienia urządzenia	18
5.2	Aplikacja Android	19
5.3	Aplikacja iOS	20
5.4	Aplikacja internetowa	25
6	Testy funkcjonalne	34
7	Uwagi końcowe	35
	Literatura	36

Rozdział 1

Wstęp

Mimo spadku stopnia przestępcości w ostatnich latach w Polsce i stosunkowo wysokiego poziomu bezpieczeństwa zawsze lepiej mieć przy sobie narzędzia, które pozwolą kontrolować to co dzieje się w naszym domu. Kiedy jesteśmy w pracy, na wakacjach czy też zostawiliśmy nasz dom bez opieki na dłuższy czas chcielibyśmy mieć możliwość podglądu tego co się w nim dzieje i na bieżąco monitorować sytuację. W przypadku włamań zarejestrowany materiał filmowy jest bardzo cenny nie tylko dla nas ale szczególnie dla policji, która jest w stanie odtworzyć zachowania i czasami zidentyfikować sprawcę. Bardzo często słyszy się też o zatruciach tlenkiem węgla, który nazywany jest cichym zabójcą czy wybuchach gazu w mieszkaniu. W takich chwilach człowiek zdaje sobie sprawę z zagrożenia najczęściej kiedy jest już za późno i są małe szanse aby uratować innych lokatorów naszego domu np. po zatruciu CO. Zbudowany system kontroli bezpieczeństwa - The Guard stara się rozwiązać te wszystkie problemy. Naszym celem było stworzenie systemu umożliwiającego analizę danych z czujników pomiarowych, monitorowanie pomieszczeń, w których zamontowano nasz system a także nagrywanie materiału video w momencie wykrycia ruchu i przechowywaniu go bezpiecznie na zewnętrznym serwerze aby był dostępny dla nas w każdym momencie i nie uległ zniszczeniu. Zadaniem systemu jest 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 i 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.

W ramach pracy przygotowano projekt całego systemu, od urządzeń zbierających dane, przez system monitorujący i analizujący zebrane dane, po aplikacje klienckie. Następnie zaimplementowano zaprojektowane wcześniej aplikacje, złożono zestawy urządzeń składających się z Raspberry Pi 3 i czujników, oraz połączono wszystkie elementy w spójny system. 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 reprezentantów, dokumentację sprzętu.

W ramach pracy Mateusz Bartos zrealizował aplikację mobilną przeznaczoną na system Android. Ponadto przygotował maszynę wirtualną w ramach usług oferowanych przez Microsoft Azure. Zaproponował też wykorzystane usługi Google: Firebase Realtime Database, Firebase Storage oraz Firebase Authentication. Piotr Falkiewicz wykonał projekt serwera, obsługującego zapytania aplikacji mobilnych, w oparciu o protokół HTTP oraz jest odpowiedzialny za poprawne przetwarzanie obrazu dostarczanego z urządzeń do aplikacji przy wykorzystaniu modułu Nginx Rtmp. Wprowadził do projektu usługę usługi Firebase Storage. Paweł Szudrowicz w ramach

pracy przygotował urządzenia Raspberry Pi 3 do obsługi czujników i wykonał oprogramowanie pracujące na każdym z nich. Kolejnym zrealizowanym zadaniem był projekt i wykonanie aplikacji mobilnej na urządzenia iOS. W ramach pracy Aleksandra Główczewska zaprojektowała i wykonała aplikację internetową, z wykorzystaniem języka programowania Python i frameworka Django. Ponadto jest odpowiedzialna za wprowadzenie uwierzytelniania użytkowników.

Rozdział 2

Architektura systemu

Wstęp do rozdziału

2.1 Schemat

// Piotr - schemat na podstawie "Praca inżynierska na Google Docs" //tzn?

2.2 Komunikacja

Komunikacja, pomiędzy elementami systemu, odbywa się na zasadach architektury REST. Takie podejście gwarantuje prostotę przesyłanych komunikatów oraz skalowalność w kontekście nowych urządzeń Raspberry, strumieniujących dane, oraz nowych urządzeń korzystających z aplikacji klienckich. Początkowo, projekt był oparty o zapytania GET i 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.

Wprowadzenie tokenów uwierzytelniających (więcej w akapicie nt. Bezpieczeństwa), spowodowało, że wymianę komunikatów oparto tylko i wyłącznie na zapytaniach POST. W zależności od zadania, obsługa zapytania polega na wykonaniu zapytania na bazie danych lub wysłaniu notyfikacji do klienta. Obsługę zapytań można również podzielić, ze względu na zaplanowane źródło zapytania: aplikacja użytkownika lub urządzenie 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 podanych serialu 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 serialu. Jeśli wykryto ruch człowieka, następuje nagranie fragmentu wideo, który zostaje zapisany w bazie danych Firebase Storage, a użytkownik zostaje poinformowany o zajściu i o nagraniu, które może pobrać. Jeśli nie wykryto obecności ludzkiej, notyfikacja zostaje 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 zmianie wartości jednego z czujników.

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, wyświetlaną w podglądzie, w aplikacji użytkownika.

f) Uzbrojenie/rozbroszenie 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, powiązanych z urządzeniem o podanym serialu.

h) Powiązanie aplikacji z kontem użytkownika:

```
Adres: /backend/v1/devices/fcmTokenUpdate  
Zawartość:  
{  
  'email': <użytkownik>,  
  'fcmToken': <ttoken-z-firebase>,  
  'deviceId' : <id_aplikacji>  
}
```

Działanie: Powiązuje aplikację, zarówno mobilną, jak i sesję aplikacji przeglądarkowej, o podanym tokenie, z kontem użytkownika. Dzięki temu, notyfikacje trafiają na wszystkie urządzenia użytkownika.

Wybrane rozwiązanie pozwala na łatwą lokalizację ewentualnego błędu w działaniu systemu, oraz szybką naprawę zaistniałego problemu. Ponadto prosta logika oraz łatwe, krótkie funkcje, obsługujące zapytania, sprawiają, że dalszy rozwój tej części systemu, będzie możliwy bardzo niskim nakładem sił.

2.3 Bezpieczeństwo

Aplikacje wysyłając zapytania do serwera, muszą potwierdzić swoją tożsamość, co dzieje się inaczej w przypadku aplikacji mobilnych i aplikacji webowej.

W przypadku aplikacji mobilnych zastosowano proponowane przez Firebase rozwiązanie JSON Web Tokens. W momencie wysłania zapytania POST do serwera, aplikacja wysyła także unikalny token, który następnie jest przez serwer weryfikowany przy użyciu Firebase Admin SDK.

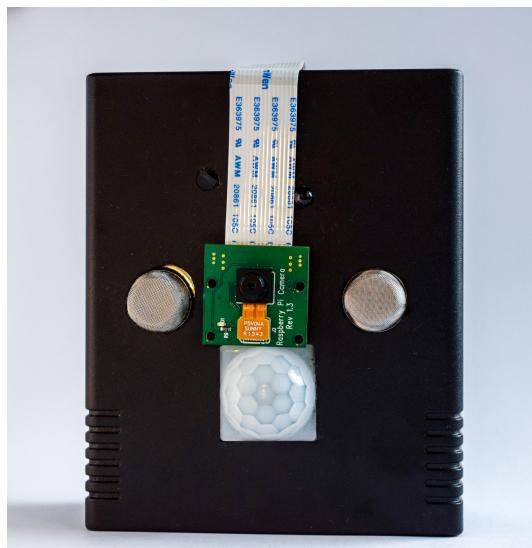
W przypadku aplikacji internetowej, zastosowano wbudowane w bibliotekę Django zabezpieczenia: CSRF token oraz przesyłanie id sesji wraz z zapytaniem. Zabezpieczenie CSRF tokenem uniemożliwia tzw. ‘Cross Site Request Forgery’ tj. ataki w których na stronie, gdzie zalogowany jest użytkownik bez jego wiedzy uruchamiany jest skrypt, najczęściej w języku JavaScript. Następnie, korzystając z faktu, że użytkownik jest zalogowany, wysyłane jest zapytanie na serwer, które może zrobić wszystkie operacje do których upoważniony jest dany użytkownik. CSRF token zapisywany jest w przeglądarce jako ‘ciasteczko’ (eng. cookie) i jest dodawany do danych przesyłanych w momenie kliknięcia przycisku odpowiedzialnego za przesłanie formularza. Następnie wbudowana w serwer Django biblioteka weryfikuje na podstawie zapisanych i przesyłanych danych sesji poprawność tokenu i w przypadku błędu zwraca błąd serwera 403. Ponieważ token przy każdym zapytaniu jest tworzony na nowo na podstawie otwartej sesji, rozwiązanie nie było komfortowe dla użytkowników aplikacji mobilnych: aplikacja musiałaby najpierw ustawić połączenie z serwerem (wysłać zapytanie GET na stronę główną), następnie zalogować się (wysłać zapytanie POST z danymi logowania) oraz zapisywać tokeny i id sesji odsyłane przez serwer. Aby ograniczyć ilość zapytań wysyłanych do serwera, posłużono się powyżej opisaną metodą tokenów JWT.

Rozdział 3

Zbieranie i przetwarzanie danych z czujników

3.1 Raspberry Pi

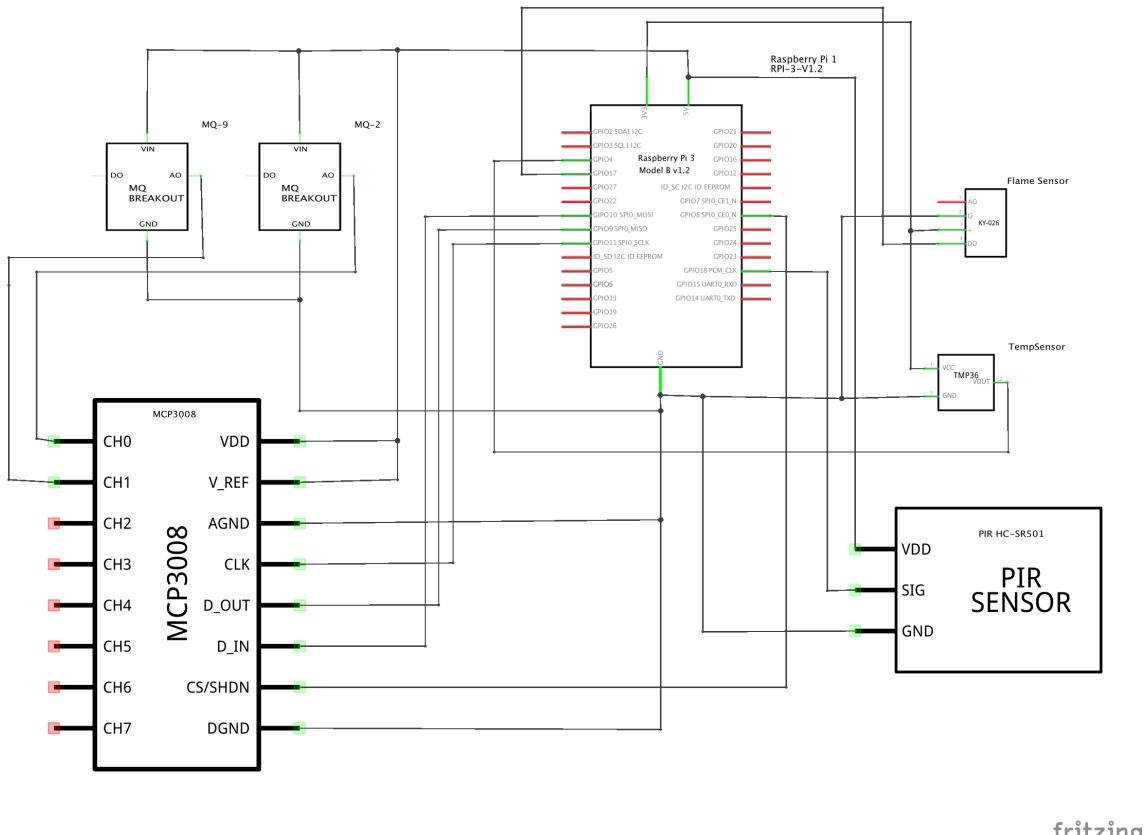
Wszystkie zestawy zbudowano w oparciu o Raspberry Pi 3 v1.2. Zdecydowano się na to rozwiązanie, ponieważ bazuje on na dystrybucji Linuxa, posiada opowiadnie interfejsy i złącza a także zintegrowany moduł WiFi. Minusem w stosunku do konkurencyjnego Arduino jest brak wejść analogowych. Problem rozwiązano dodając zewnętrzny przetwornik A/C. Całość zamknięto w małą plastikową obudowę z wyciętymi otworami na czujniki (rys. 3.1). Schemat budowy układu wykonano w programie Fritzing (rys. 3.2).



RYSUNEK 3.1: Zbudowany zestaw The Guard

Specyfikacja Raspberry Pi 3:

- Procesor 1.2 GHz
- Liczba rdzeni 4. Quad Core
- Pamięć RAM 1 GB
- Pamięć Karta microSD



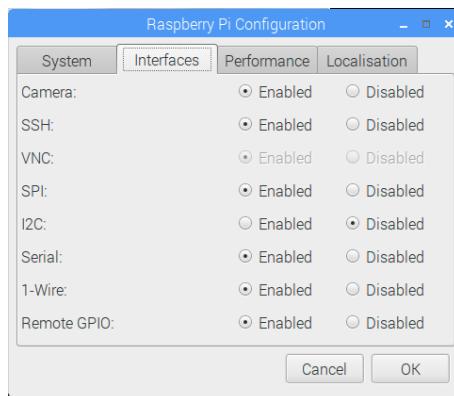
fritzing

RYSUNEK 3.2: Schemat układu The Guard

- 40 GPIO

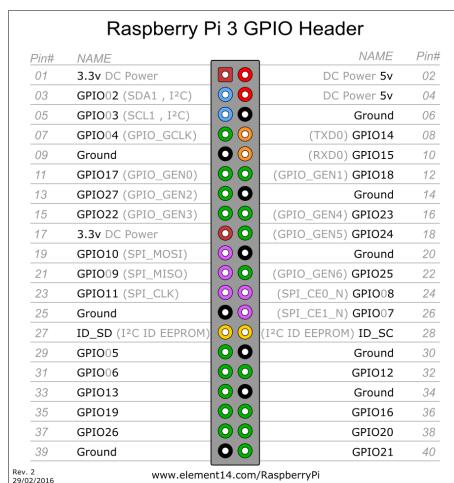
Aby prawidłowo zainstalować oprogramowanie The Guard na dowolnym urządzeniu Raspberry Pi 3 należy wykonać poniższe czynności w terminalu:

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



RYSUNEK 3.3: Ustawienia

Następnym krokiem jest włączenie odpowiednich interfejsów w panelu konfiguracyjnym. Należy zmienić ustawienia zgodnie ze schematem (rys. 3.3). Użyto biblioteki wiringpi do odczytu danych z układów cyfrowych. Należy podkreślić, że numeracja fizycznych pinów (rys. 3.4) i numeracja pinów w wiringPi (rys. 3.5) jest różna i nie zawiera wszystkich dostępnych pinów na urządzeniu. Przykładowo odczyt pinu numer 1 w wiringPi jest równoznaczny z odczytem stanu na pinie numer 12 (GPIO18). Zainstalowane oprogramowanie odpowiedzialne jest za ciągłe monitorowanie



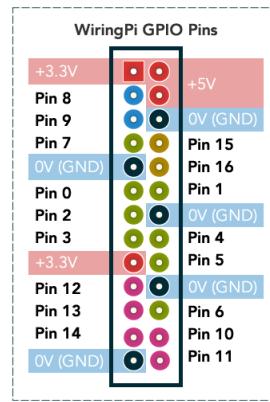
RYSUNEK 3.4: GPIO

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 klienckich. Dodanie akcesorium pomiarowego następuje poprzez wprowadzenie w aplikacji jego numeru seryjnego.

3.2 Czujniki

Każdy zestaw składa się z 5 czujników analogowo cyfrowych, jednej kamery i jednego przetwornika AC.

a) Specyfikacja MQ-9 - czujnik tlenku węgla:



RYSUNEK 3.5: WiringPi

- Zasilanie: 5 V
- Pobór prądu: 150 mA
- Temperatura pracy: od -10 do 50 °C
- Wyjścia: analogowe oraz cyfrowe

b) Specyfikacja MQ-2 - czujnik LPG i dymu:

- Zasilanie: 5 V
- Pobór prądu: 150 mA
- Temperatura pracy: od -10 do 50 °C
- Wyjścia: analogowe oraz cyfrowe

c) Specyfikacja czujnika wykrywania płomieni:

- 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:

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

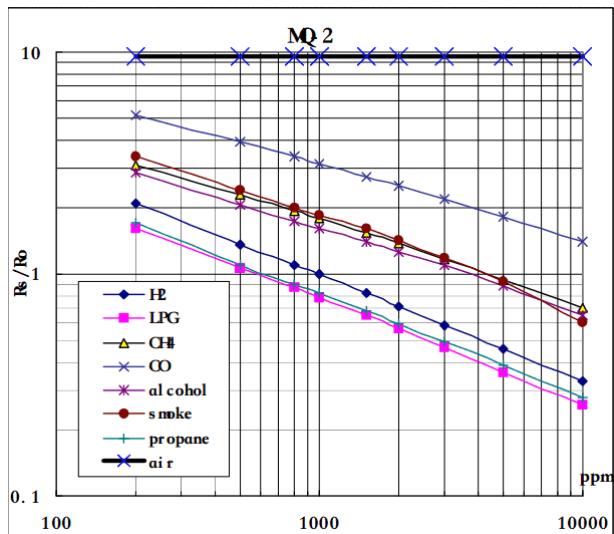
e) Kamera:

- Wykorzystano moduł kamery Raspberry Pi element14
- Kamera 5MP - wspierająca nagrywanie 30 klatek na sekundę w rozdzielcości Full HD

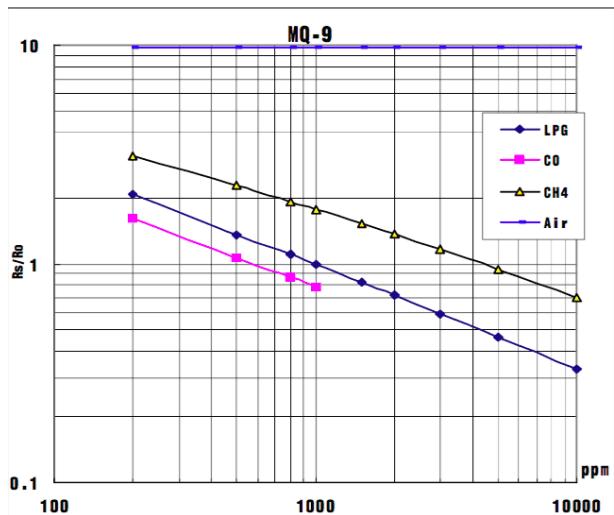
f) Specyfikacja MCP3008 - przetwornik A/C:

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

Na schematach (rys. 3.6, rys. 3.7) przedstawiono charakterystykę czujników analogowych. R_o -



RYSUNEK 3.6: Charakterystyka MQ-2 [1]



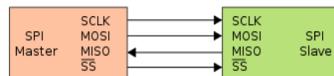
RYSUNEK 3.7: Charakterystyka MQ-9 [2]

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

R_s - jest to opór czujnika w różnych stężeniach gazu. Skoro R_o jest stałe to przy wzroście R_s czułość będzie maleć. Dlatego też im mniejszy stosunek R_s do R_o tym lepiej. Widać, że oba

czujniki reagują na wiele różnych gazów. MQ-2 nazywano czujnikiem LPG a MQ-9 czujnikiem CO ze względu na to, że w stosunku do tych gazów mają najwyższą czułość.

Niestety żaden model Raspberry nie posiada wbudowanego przetwornika analogowo cyfrowego dlatego konieczne było użycie układu zewnętrznego. Wybrano przetwornik MCP3008 ze względu na jego nisko koszt i interfejs SPI, który jest wspierany przez Raspberry Pi. MCP3008 to 10-bitowy przetwornik analogowy cyfrowy. Zasilany jest napięciem 5V. Skoro jest to przetwornik 10-bitowy jest w stanie wykryć 1024 stany. Posiada 8 kanałów jednak w projekcie wykorzystano tylko 2 – dla MQ-9 i MQ-2.



RYSUNEK 3.8: Interfejs SPI [6]

Interfejs SPI: SPI jest to interfejs synchroniczny (rys. 3.8). Może być do niego podłączone wiele urządzeń typu slave, jednak tylko z jednym urządzeniem Master, które generuje zegar. Master poprzez linię SS wybiera urządzenie z którym chce się komunikować.

Interfejs ten 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 przesyłania zegara wygenerowanego przez Raspberry Pi 3

Do komunikacji poprzez ten interfejs wykorzystano bibliotekę spiDev.

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 wtedy 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 i ruchu. Czujnik ruchu detekcję zagrożenia określa przez stan wysoki natomiast czujnik płomieni przez stan niski. W kodzie jednak wykonano instrukcje negacji, aby stan wysoki 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 czujnik ruchu nie wykryje zagrożenia

z powodu braku odczytu we właściwym momencie, ponieważ utrzymuje on 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 jest w stanie odtworzyć całą historię wydarzeń w swoim systemie. Aby zapewnić wydajny i pewny system bezpieczeństwa przy otrzymywaniu 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.

3.3 Obsługa wideo

Protokół RTMP Podstawą funkcji strumieniowania wideo, jest protokół RTMP (Real-Time Message Protocol). Jest to, oparty na protokole TCP, protokół wysyłania obrazu, dźwięku oraz danych. Podstawową jednostką danych, w protokole RTMP, jest wiadomość (ang. Message), której struktura jest zależna od typu strumieniowanych informacji. Wiadomości dzielone są na kawałki (ang. Chunks), które są porcjami gotowymi do transmisji. Zatem strumień RTMP to ostatecznie strumień częstek (ang. Chunk Stream)

Ponadto, wykorzystano protokół HLS (HTTP Live Streaming), zapisywanie odbieranego obrazu, we wskazanej liczbie plików wideo o określonej długoci. Gdy aplikacja kliencka odtwarza strumień wideo, w rzeczywistości odbiera, strumieniowane po kolei, zapisane pliki ts. Wpływ to na opóźnienie odtwarzania, względem rzeczywistości, z jednej strony, a z drugiej, dostarcza płynne wideo.

H264 W pracy wykorzystano kodowanie obrazu koderem H.264. Charakteryzuje go niska złożoność algorytmów kompresji oraz niewielkie opóźnienie, dzięki czemu idealnie nadaje się do zadania związanego z szybkim enkodowaniem obrazu, przed przestreamieniowaniem go dalej. Szybkość i złożoność H.264, idą w parze z jego jakością, która jest o wiele wyższa, niż w starszych rozwiązańach. Cechą charakterystyczną, dla tego typu kodowania wideo, jest użycie klatki kluczowej (ang key-frame, i-frame). Jest to pełna klatka obrazu, w przeciwnieństwie, do następujących po niej danych, które wyrażają różnice między dwoma konsekwencyjnymi klatkami. Pozwala to na zmniejszenie rozmiarów, ostatecznego obrazu wideo.

Raspberry Pi Do obsługi strumieniowania wideo, po stronie Raspberry Pi, wykorzystywany jest program FFmpeg. Pozwala on na sterowanie strumieniem, od wyboru urządzenia wejściowego, przez statystyki strumienia, po punkt docelowy. Dostęp do unikatowego, dla każdego urządzenia Raspberry, punktu końcowego, gwarantował wcześniejsze pobranie nr seryjnego urządzenia. Poniżej przedstawiono skrypt, wykonujący wymienione funkcje:

```
#!/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 znajdowana jest w nim linia, w której znajduje się wyjątkowy serial urządzenia. Na końcu, z wykorzystaniem potoku, i funkcji cut, wartość ta zostaje przypisana do zmiennej serialid.

W drugiej linii skryptu wykorzystane jest narzędzie linii poleceń Raspberry - raspivid. Pozwala ono pobrać obraz z kamery.

- Pierwszym przełącznikiem jest -o z parametrem -. Oznacza to, że obraz z kamery jest wysyłany na wyjście standardowe.
- Przełącznik -t ustawiony na 0 pozwala przekazywać obraz, z modułu kamery, przez nieokreślony 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 pamięci, w której ma się znaleźć obraz przechwycony w ciągu 1 sekundy. 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.

Drugim polecienniem jest wywołanie narzędzia ffmpeg, połączonego za pomocą potoku, odbierającego, przechwytywany za pomocą funkcji raspivid, obraz i przekazującego go na docelowy punkt końcowy. Za jego pomocą ustala się ostatecznie opcje kodujące obraz i dźwięk w trakcie końcowej transmisji.

- Przełącznik -re pozwala odczytywać dane wejściowe, z oryginalną częstotliwością. Zatem zostają przechwycone ustawienia przełącznika funkcji raspivid -fps 30.
- Opcje -ar, -ac, -acodec, -f, -strict odpowiadają kolejno za: próbkowanie dźwięku, wybór liczby kanałów, kodęk 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 h264.
- 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, obsługiwanej przez serwer o adresie IP 52.236.165.15 na porcie 1936, obraz wysyłany jest na aplikację o nazwie camera i punkt charakteryzowany przez serial urządzenia. Działanie tego elementu opisano w kolejnym punkcie.

Serwer Narzędziem, umożliwiającym obsługę strumieniowania wideo, z wielu źródeł, na wiele urządzeń równoczesnie, jest serwer NGINX. W części projektu, związanej ze strumieniowaniem wideo, wykorzystano moduł nginx-rtmp. Moduł ten pozwala, m. in., na:

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

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

Wszystkie funkcjonalności definiują poniższe 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;
        }
    }
}
```

Powyższe linie powodują, że serwer rtmp, dostępny jest na porcie 1936 (domyslnie porty dla protokołu RTMP, na urządzeniu z systemem z rodziny Ubuntu, to 1935 i 1936). Następnie tworzona jest aplikacja o nazwie camera. Dla aplikacji strumieniujących dane, jest ona dostępna pod adresem: rtmp://<ip-serwera>:1936/camera/<klucz>, gdzie klucz jest wybierany przez aplikację strumieniującą 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 będą 2 tymczasowe fragmenty, o długości 3 sekund, każdy. Pliki te będą 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 związanego z serwerem RTMP.

Wykorzystane funkcjonalności narzędzia Nginx nie wykorzystują w pełni możliwości produktu, jednak projekt nie wymagał korzystania z funkcji serwera proxy.

Rozdział 4

Rozwiązania chmurowe

4.1 Microsoft Azure

Abi zapewnić wysoki poziom bezpieczeństwa oraz dostępności systemu zdecydowano się na skorzystanie z chmury Microsoft Azure.

4.2 Firebase

// Mateusz

4.3 Aplikacja serwerowa

Koncepcja Aplikacja serwerowa, została zaplanowana jako interfejs pomiędzy: urządzeniami periferyjnymi, aplikacjami użytkowników urządzeń mobilnych, bazą danych a usługami chmurowymi. Założeniem, dotyczącym pierwsza części komunikacji: Raspberry - Serwer - Aplikacja, było wykorzystanie zapytań HTTP: GET i POST.

- Zapytanie GET
- Zapytanie POST

final version W ostatecznej wersji, skorzystano z dodatkowych usług, m. in Firebase OAuth2. Uwierzytelnianie użytkowników zaczęło wymagać, by do każdego zapytania, dodać token użytkownika. Spowodowało to, że jedynym typem zapytań, wykorzystywanym w systemie, są zapytania POST.

// Ola

4.4 Baza danych

// Ola

Rozdział 5

Aplikacje klienckie

Na podstawie analizy statystyk dotyczących podziału rynku aplikacji na platformy, dostępnych na stronie statcounter.com¹, 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 webowa.

5.1 Funkcje aplikacji

5.1.1 Logowanie

Ogólny opis funkcjonalności dla iOS, Android i Web

5.1.2 Monitoring

Ogólny opis funkcjonalności dla iOS, Android i Web

5.1.3 Status czujników

Ogólny opis funkcjonalności dla iOS, Android i Web

5.1.4 Dziennik zdarzeń

Ogólny opis funkcjonalności dla iOS, Android i Web

5.1.5 Ostatnie zagrożenia

Ogólny opis funkcjonalności dla iOS, Android i Web

5.1.6 Ustawienia urządzeń

Ogólny opis funkcjonalności dla iOS, Android i Web

¹<http://gs.statcounter.com/os-market-share/mobile-tablet/worldwide/#monthly-201801-201801-bar>

5.2 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.² Kotlin ściśle współpracuje z kodem stworzonym w Javie i w przypadku Androida jest kompilowany do kodu JVM.

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.

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", czyli kod odpowiedzialny za logikę biznesową, połączenie z serwerem i złożone operacje,
- "View", czyli kod odpowiedzialny wyłącznie za poprawne wyświetlanie przygotowanych informacji,
- "Presenter", czyli kod odpowiedzialny za przygotowanie informacji otrzymanych z warstwy model do wyświetlenia 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.

Funkcje i interfejs użytkownika

Aplikacja została zaprojektowana zgodnie z wytycznymi Material Design³. 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.

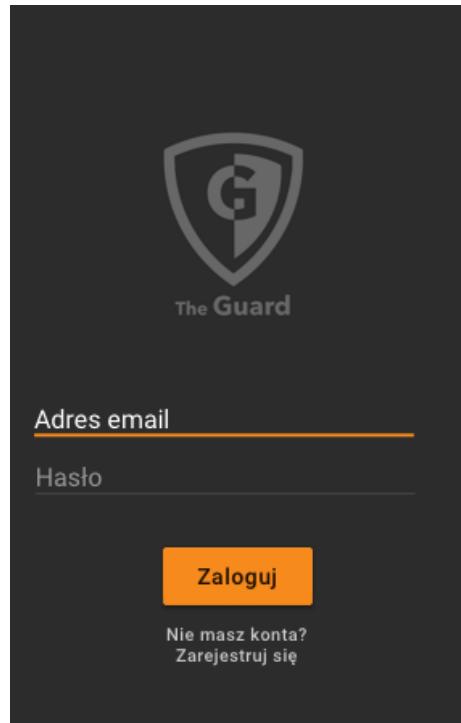
Logowanie Użytkownik loguje się do aplikacji przy użyciu adresu email oraz hasła. Dane te trafiają do Firebase Authotizdtio

Integracje

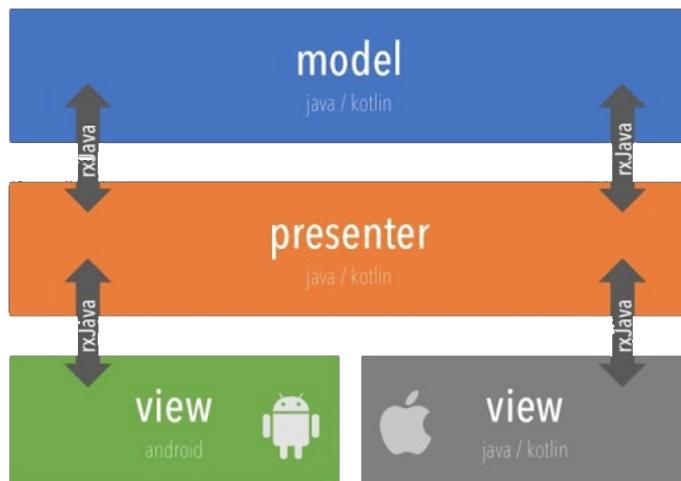
Opis połączenia aplikacji z Firebase, Fabric i innymi bibliotekami.

²<https://twitter.com/Android/status/864911929143197696>

³Learning Material Design: Master Material Design and create beautiful, animated interfaces for mobile and web applications, Kyle Mew, Packt Publishing 2015



RYSUNEK 5.1: Ekran logowania do aplikacji

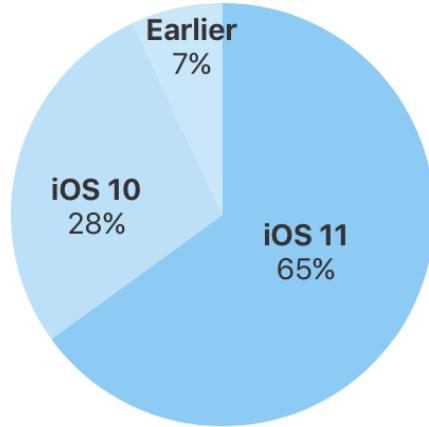


RYSUNEK 5.2: Architektura MVP

5.3 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)⁴ jest w stanie zainstalować oprogramowanie a liczba ta stale rośnie. Aplikacja wspiera zarówno telefony komórkowe iPhone jak i tablety iPad. Napisana w stosunkowo nowym języku Swift (zaprezentowany przez Apple w 2014r na konferencji WWDC) w oparciu o architekturę MVC (Model-View-Controller) wykorzystując przy tym programowanie reaktywne i funkcjonalne. Aplikacja powstała w programie Xcode. Programowanie reaktywne

⁴<https://developer.apple.com/support/app-store/>



RYSUNEK 5.3: Udziały wersji systemu iOS z 18.01.2018

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 background do trybu foreground. Oznacza to, że po wyjściu z aplikacji, ale pozostawiając ją działającą w tle 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 [3]. 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 w tle jest ograniczona (rys. 5.2). Próba oszustwa i wykonywania innej pracy w tle niż zaznaczone zostało w trybie Background Modes.



RYSUNEK 5.4: Tryby pracy w tle

czona 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 zapewnie 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 ponowym uruchomieniu programu. "Programowanie funkcjonalne 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." [5]. 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

```

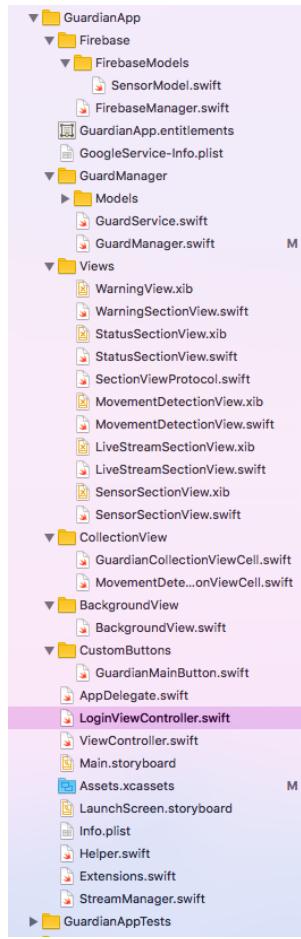
Na tablicy z notyfikacjami zastosowano szereg kolejnych funkcji: posortowano je malejąco według daty, przefiltrowano w taki sposób aby wybrać tylko te o typie 'PIRSensor' czyli te pochodzące z czujnika ruchu. Na sam koniec wybrano tylko jeden pierwszy element z wybranych i wynik wpisano do zmiennej lastNotification.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

Strukturę kodu (rys. 5.3) 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. 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'

```



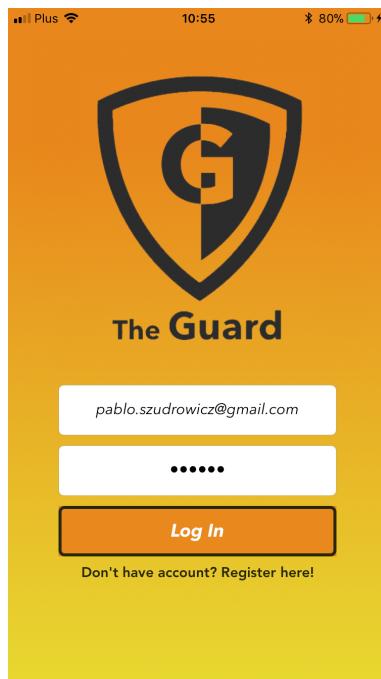
RYSUNEK 5.5: Struktura aplikacji

```

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 Firebas/Core itp. służą do komunikacji z serwerami Firebase. Ostatni 'pod M13ProgressSuite' służy do rysowania wykresów i animowanych elementów graficznych w systemie iOS. Po uruchomieniu aplikacji pierwszym widokiem jest ekran logowania i rejestracji użytkowników (rys 5.4). 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 streamu na żywo, sekcja ustawień. Wszystkie te sekcje dotyczą konkretnego urządzenia wybranego w pasku na dole ekranu. 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



RYSUNEK 5.6: Ekran logowania

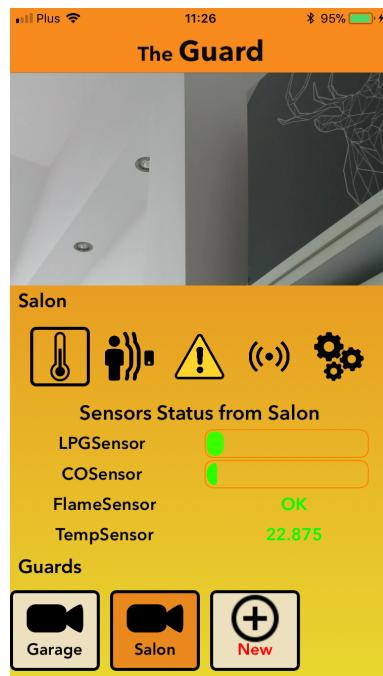
a także śledzić i monitorować informacje z czujników należy wybrać przycisk 'Newz plusikiem w dolnej części ekranu. Pojawi się okno z prośbą o wpisanie numeru identyfikującego urządzenie. Po chwili dodany "Guard" będzie widoczny w na liście.

Sekcja czujników: Jest to jedna z najważniejszych sekcji aplikacji (rys 5.5). Otrzymuje ona dane z czujników w czasie rzeczywistym i prezentuje je użytkownikowi. W zależności od koloru prezentowanej wartości z czujnika użytkownik analizuje zagrożenie. Kolor zielony reprezentuje bezpieczne i prawidłowe odczyty na czujnikach, kolor pomarańczowy średnie, kolor czerwony reprezentuje bardzo wysoki poziom niebezpieczeństwa. Implementacja tej funkcjonalności zrealizowana została przy pomocy modelu HSV, który w przeciwieństwie do RGB pozwala na bardzo proste przejście z jednego koloru do kolejnego poprzez zmienę 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 zmienne sensors[0] reprezentuje czujnik LPG.

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

Sekcja historii notyfikacji: W tej sekcji użytkownik ma dostęp do historii zdarzeń w systemie (rys 5.6). Po zaznaczeniu daty reprezentującej moment wystąpienia zagrożenia i wybranu przycisku 'preview' prezentowana jest informacja o miejscu nie bezpieczeństwa i jego rodzaju.

Sekcja ustawień: 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



RYSUNEK 5.7: Sekcja czujników

nie otrzyma kolejnych notyfikacji o zagrożeniach (rys. 5.7). Opcja ta może okazać się przydatna w momencie uszkodzenia któregoś z modułów i tym samym błędnych danych wysyłanych z czujników.

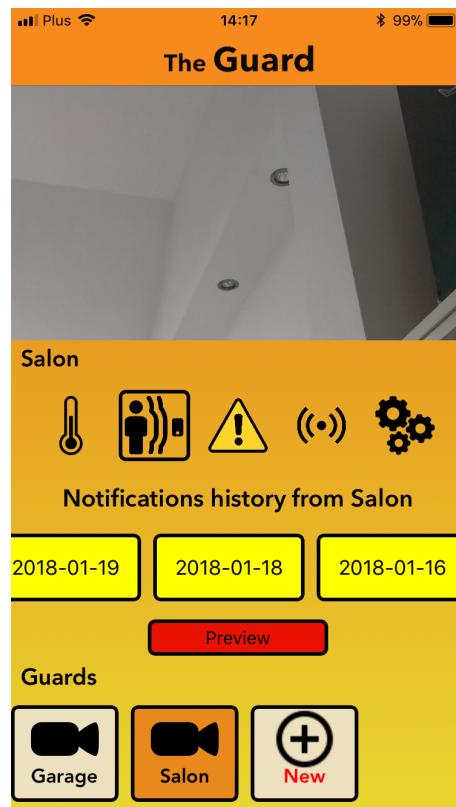
Sekcja streamu: Sekcja odpowiedzialna za prawidłowy odbiór obrazu z kamery zaznaczonej w dolnej części ekranu. Okno, w którym odbywa się stream ustawiono w taki sposób, aby bez względu na rozmiar telefonu utrzymywało proporcję 16:9. Pozbyto się dzięki temu czarnych ramek lub braku części obrazu ze streamu.

Sekcja ostatnich zagrożeń: Sekcja prezentuje ostatnie nagrane zagrożenie. Służy do szybkiego przeglądu ostatniego niebezpieczeństwa i prezentuje ostatni nagrany materiał video.

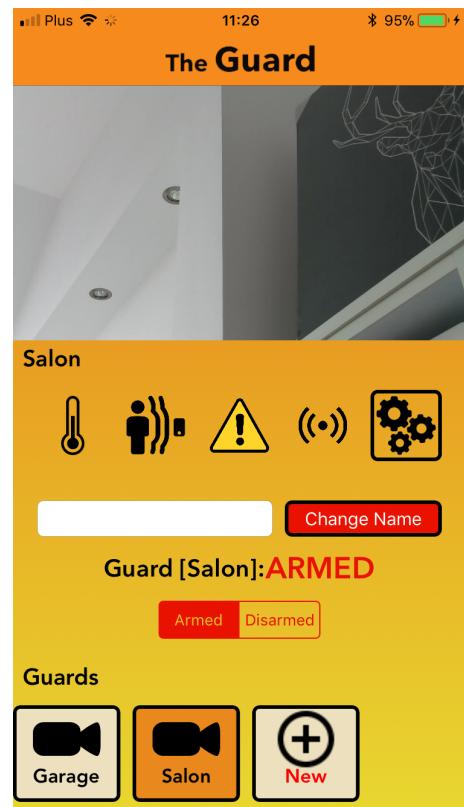
Przeprowadzono kilka testów aplikacji pod pełnym obciążeniem za pomocą programu Instruments. Szczególnie interesująco przedstawia się zużycie sieci podczas streamu obrazu. Widać, że w ciągu jednej minuty pobrano 6,61MB a wysłano jedynie 24,11KB (rys. 5.8). Obraz pobierany jest tylko wtedy kiedy aplikacja 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. 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. 5.9). Testy przeprowadzono na iPhone 6S i iPadzie Pro.

5.4 Aplikacja internetowa

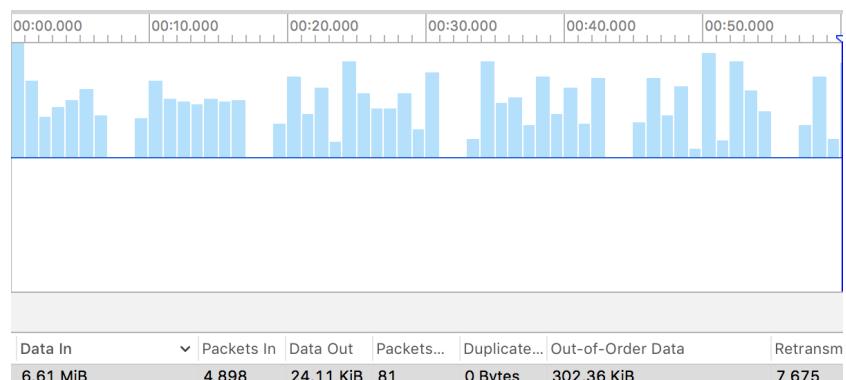
Aplikacja webowa przeznaczona jest dla użytkowników wszystkich systemów operacyjnych i została przetestowana w przeglądarce Firefox. Rozmieszczenie komponentów aplikacji różni się od



RYSUNEK 5.8: Sekcja historii notyfikacji



RYSUNEK 5.9: Sekcja ustawień



RYSUNEK 5.10: Zużycie sieci podczas streamu

tego zastosowanego w aplikacjach IOS oraz Android - spowodowane jest to inną rozdzielcością ekranu.

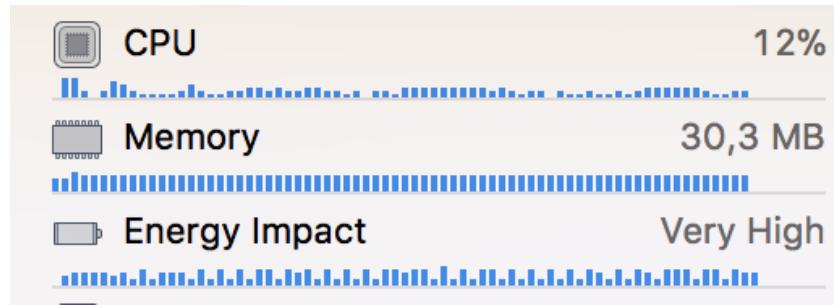
Panel logowania / rejestracji użytkownika:

Menu wyboru urządzenia:

Główny panel aplikacji:

Panel rejestracji urządzenia:

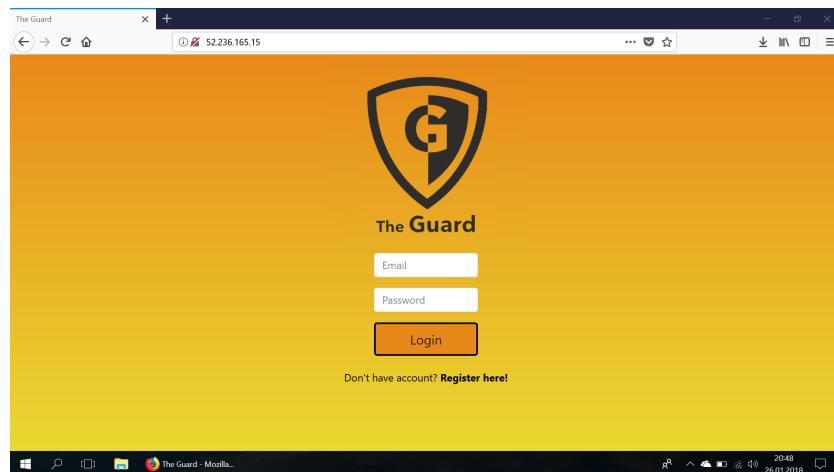
Panel notyfikacji:



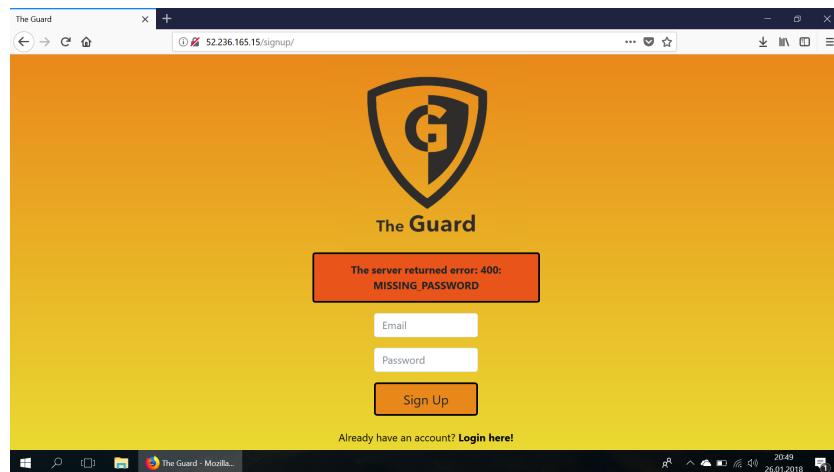
RYSUNEK 5.11: Zużycie procesora i RAM podczas największego obciążenia

Implementacja Django - połączenie z bazą danych: Biblioteka Django posiada wbudowane rozwiązania umożliwiające pobieranie informacji z bazy danych projektu dzięki czemu aplikacja webowa nie wysyła zapytań na określone dla aplikacji mobilnych porty, tylko komunikuje się bezpośrednio z bazą danych. Rozwiązywanie to umożliwia uniezależnienie aplikacji webowej 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 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. 5.19)

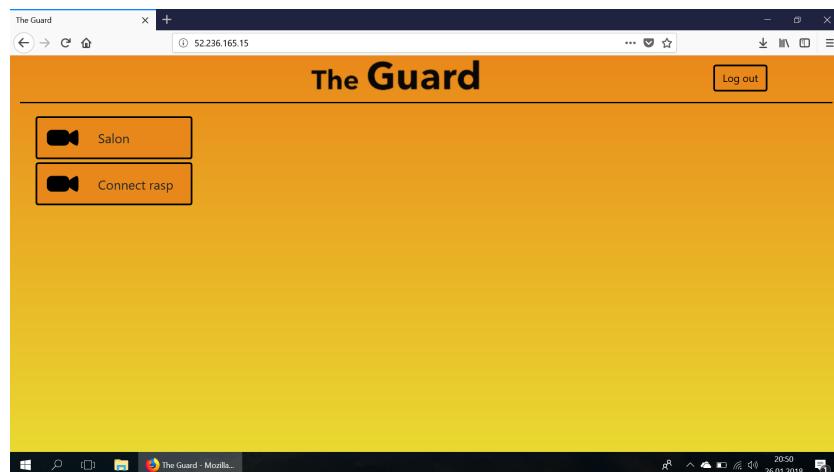


RYSUNEK 5.12: Strona logowania w aplikacji webowej



RYSUNEK 5.13: Strona logowania - przykładowa obsługa błędu (użytkownik nie podał hasła)

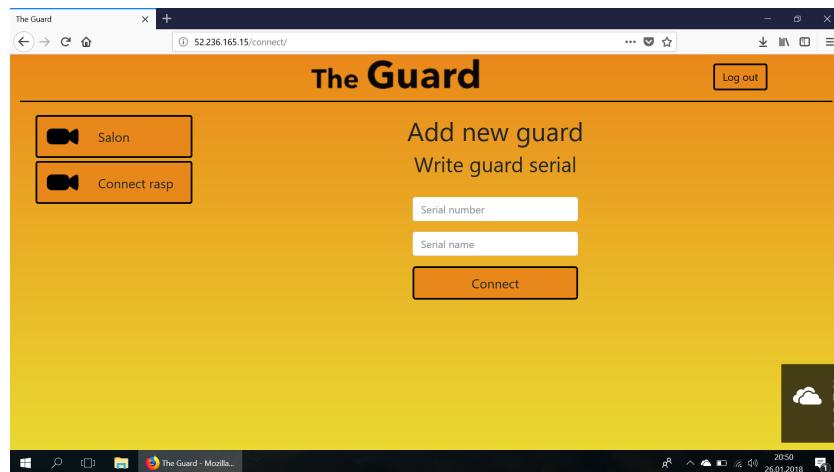
Menu wyboru urządzenia: Po prawidłowym zalogowaniu do aplikacji, użytkownik może zobaczyć listę swoich urządzeń, dodać nowe oraz zaczyna dostawać powiadomienia w razie wykrytego zagrożenia. W przypadku kliknięcia przycisku ‘Connect rasp’, użytkownik zostaje przekierowany do widoku umożliwiającego rejestrację nowego urządzenia(rys. 5.22). 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 (rys. 5.21). Urządzenia w menu są rozpoznawane na podstawie ich nazw.



RYSUNEK 5.14: Strona główna aplikacji webowej

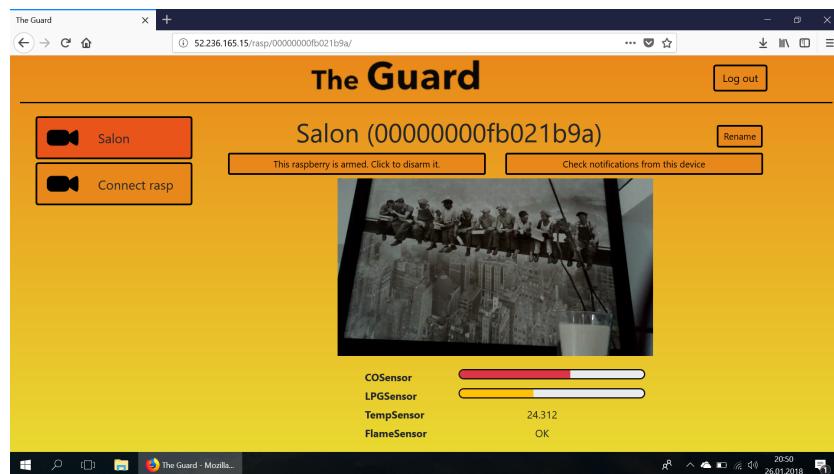
Widok konkretnego urządzenia: Po wybraniu z menu konkretnego urządzenia, użytkownik zostaje przekierowany na stronę pojedyńczego urządzenia (rys. 5.23). Pod nazwą urządzenia i jego numerem seryjnym wyświetlany jest aktualny obraz z kamery oraz stan czujników. Dzięki zastosowaniu nasłuchiwanego na bazie 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. 5.24).
- Wyłączyć / włączyć alerty



RYSUNEK 5.15: Panel rejestracji nowego urządzenia

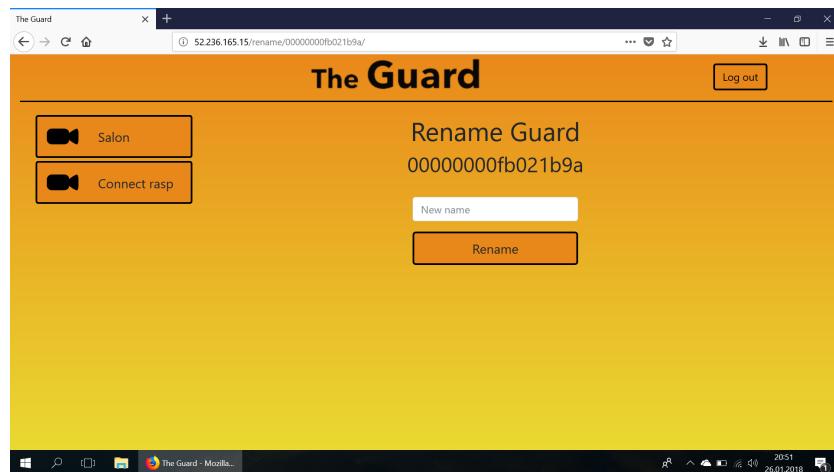
- Zobaczyć notyfikacje danego urządzenia - poprzez kliknięcie na przycisk ‘Check notifications from this device’, użytkownik zostanie przekierowany do widoku listy notyfikacji danego urządzenia (rys. 5.25).



RYSUNEK 5.16: Widok konkretnego urządzenia

Wyświetlanie notyfikacji: W każdym z widoków aplikacji webowej 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 5.25.

Implementacja Django - połaczanie z bazą danych: Biblioteka Django posiada wbudowane rozwiązania umożliwiające pobieranie informacji z bazy danych projektu dzięki czemu aplikacja webowa nie wysyła zapytań na określone dla aplikacji mobilnych porty, tylko komunikuje się bezpośrednio z bazą danych. Rozwiążanie to umożliwia uniezależnienie aplikacji webowej 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 oraz Linux Debian 9 (Firefox ESR 52.5.2 64 bit). Do stworzenia aplikacji użyto języków programowania Python 3,



RYSUNEK 5.17: Panel zmiany nazwy urządzenia

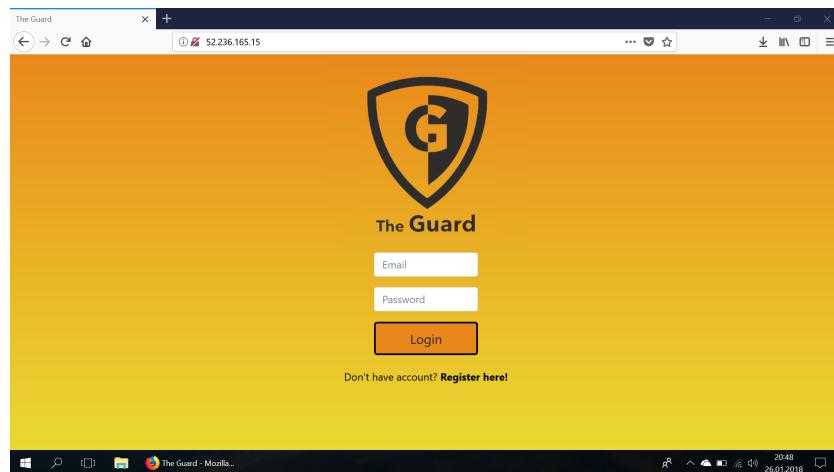
Date	Type of notification	Message
Jan. 26, 2018	PIRSensor	JP2
Jan. 26, 2018	PIRSensor	Test alert

RYSUNEK 5.18: Panel notyfikacji urządzenia

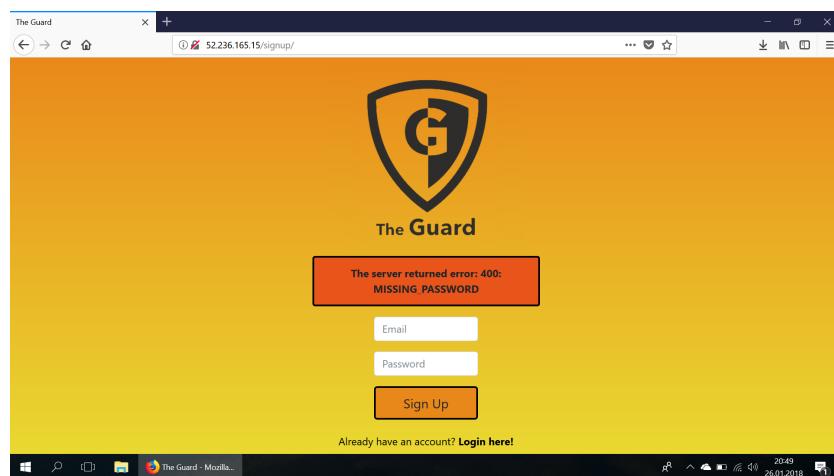
JavaScript oraz framework'u Django, natomiast frontend jest oparty na bibliotece Bootstrap oraz JQuery. Połaczanie z bazą danych Firebase zaimplementowano za pomocą Firebase Web API. Rozmieszczenie komponentów aplikacji różni się od tego zastosowanego w aplikacjach IOS oraz Android - spowodowane jest 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. 5.19)

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(rys. 5.22). Po wprowadzeniu numeru seryjnego urządzenia oraz jego nazwy, zostaje dodany do baz danych. Po wybraniu urządzenia,



RYSUNEK 5.19: Strona logowania w aplikacji webowej

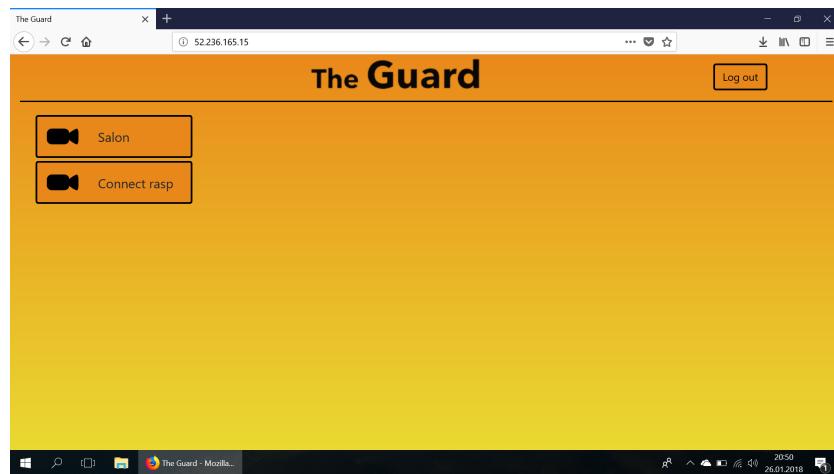


RYSUNEK 5.20: Strona logowania - przykładowa obsługa błędu (użytkownik nie podał hasła)

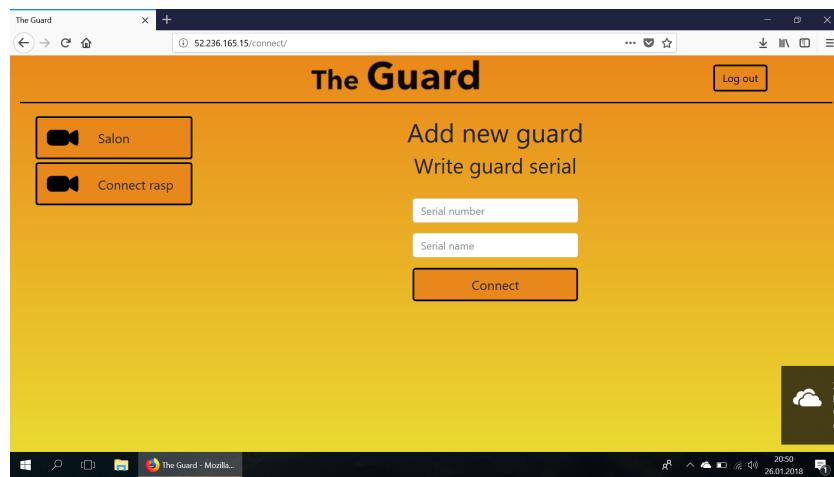
informacje nt. jego stanu będą wyświetlane po prawej stronie okna, która w momencie zalogowania jest pusta (rys. 5.21). Urządzenia w menu są rozpoznawane na podstawie ich nazw.

Widok konkretnego urządzenia: Po wybraniu z menu konkretnego urządzenia, użytkownik zostaje przekierowany na stronę pojedynczego urządzenia (rys. 5.23). 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. 5.24).
- 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. 5.25).



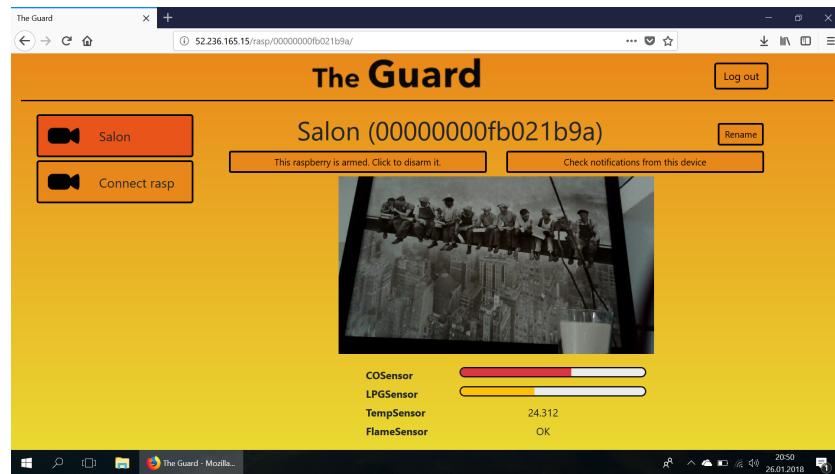
RYSUNEK 5.21: Strona główna aplikacji webowej



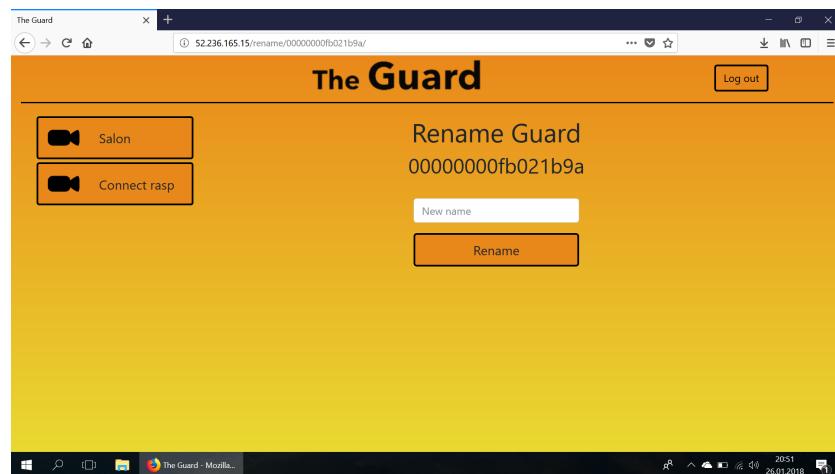
RYSUNEK 5.22: Panel rejestracji nowego urządzenia

Wyświetlanie notyfikacji: W każdym z widoków aplikacji webowej 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 5.25.

Implementacja Django - połączenie z bazą danych: Biblioteka Django posiada wbudowane rozwiązania umożliwiające pobieranie informacji z bazy danych projektu dzięki czemu aplikacja webowa nie wysyła zapytań na określone dla aplikacji mobilnych porty, tylko komunikuje się bezpośrednio z bazą danych. Rozwiążanie to umożliwia uniezależnienie aplikacji webowej od stanu portów oraz zmniejsza liczbę potrzebnych zapytań wysyłanych do serwera.



RYSUNEK 5.23: Widok konkretnego urządzenia



RYSUNEK 5.24: Panel zmiany nazwy urządzenia

Date	Type of notification	Message
Jan. 26, 2018	PIRSensor	JP2
Jan. 26, 2018	PIRSensor	Test alert

RYSUNEK 5.25: Panel notyfikacji urządzenia

Rozdział 6

Testy funkcjonalne

Testy: * Rejestracja+dodanie raspa * Aplikacji (3x, dla każdej): ** Sprawdzenie zagrożenia (skok wartości jednego z czujników) ** Sprawdzenie obrazu z kamery na wybranym raspie ** Odtworzenie nagrania video

Rozdział 7

Uwagi końcowe

Zakończenie pracy zwane również Uwagami końcowymi lub Podsumowaniem powinno zawierać ustosunkowanie się autora do zadań wskazanych we wstępie do pracy, a w szczególności do celu i zakresu pracy oraz porównanie ich z faktycznymi wynikami pracy. Podejście takie umożliwia jasne określenie stopnia realizacji założonych celów oraz zwrócenie uwagi na wyniki osiągnięte przez autora w ramach jego samodzielnej pracy.

Integralną częścią pracy są również dodatki, aneksy i załączniki np. płyty CDROM zawierające stworzone w ramach pracy programy, aplikacje i projekty.

Literatura

- [1] Dokumentacja mq-2. <https://www.pololu.com/file/0J309/MQ2.pdf>. [Online].
- [2] Dokumentacja mq-9. <http://www.haoyuelectronics.com/Attachment/MQ-9/MQ9.pdf>. [Online].
- [3] Apple. Background execution. <https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html>. [Online].
- [4] John Doe. *The Book without Title*. Dummy Publisher, 2100.
- [5] Paul Hudson. Pro swift, 2016. [Online, str. 172].
- [6] Wikipedia, wolna encyklopedia. Serial peripheral interface. https://pl.wikipedia.org/wiki/Serial_Peripheral_Interface. [Online].



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

Instytut Informatyki, Wydział Informatyki
Politechnika Poznańska

Skład przy użyciu systemu LATEX.

BibTEX:

```
@mastersthesis{ 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, Poland",  
    year = "2018",  
}
```