

ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ

ДИПЛОМНА РАБОТА

Тема: IoT система за отдалечен достъп, контрол и анализ на устройства

Дипломант:

Огнян Барух

Научен ръководител:

Атанас Атанасов

СОФИЯ

2021

СЪДЪРЖАНИЕ

СЪДЪРЖАНИЕ	3
УВОД	6
ГЛАВА I. ТЕХНОЛОГИИ ЗА РЕАЛИЗАЦИЯ НА СИСТЕМА ОТ СВЪРЗАНИ УСТРОЙСТВА	7
1.1 Лицево засичане	7
1.2 Лицево верифициране	7
1.3 NumPy[1]	8
1.4 Виртуални среди	8
1.5 MQTT[2]	9
1.6 Дигитални близнаци	10
1.7 Raspberry Pi[3]	10
1.8 Bosch IoT Suite[4]	11
1.9 Bosch IoT Hub[5]	11
1.10 Bosch IoT Things[6]	11
1.11 Bosch IoT Edge Services[7]	12
1.12 Bosch IoT Edge Agent[8]	12
1.13 Gateway Device	12
1.14 Умни устройства	12
1.15 Z-Wave[10]	13
1.16 ONVIF[11]	13
1.17 Flutter[12]	13
1.18 Firebase[13]	14
1.19 Firebase Authentication[14]	14
1.20 Cloud Storage for Firebase[15]	14
1.21 Cloud Firestore[16]	15
ГЛАВА II. ИЗИСКВАНИЯ КЪМ ПРИЛОЖЕНИЕТО. СТРУКТУРА НА СИСТЕМАТА ОТ УСТРОЙСТВА	16
2.1 Изисквания към приложението	16
2.1.1 Изисквания към модела за лицево верифициране	16
2.1.2 Изисквания към системата от свързани устройства	16
2.1.3 Изисквания към мобилното приложение	17
2.2 Съображения за избор на програмни средства	18
2.2.1 Алгоритми за машинно самообучение	18
2.2.2 Устройства в системата	18
2.3 Основни алгоритми за лицево засичане и лицево верифициране	19

2.3.1 Алгоритми за лицево засичане	19
2.3.2 Алгоритми за лицево верифициране	19
2.4 Структура на Flutter мобилно приложение	20
2.5 Структура на базата от данни на мобилното приложение	22
2.6 Структура на системата от свързани устройства	26
ГЛАВА III. ПРОГРАМНА РЕАЛИЗАЦИЯ НА СИСТЕМА ОТ СВЪРЗАНИ УСТРОЙСТВА	27
3.1 Файлово и функционално разпределение на приложението	27
3.2 Файлова структура на управляващия код за системата от свързани устройства	30
3.3 Програмна реализация на алгоритъма за лицево верифициране	31
3.3.1 Файлова структура	31
3.3.2 Създаване на изображенията за трениране	31
3.3.3 Помощни функции за лицево засичане	37
3.3.4 Помощни функции за лицево верифициране	44
3.3.5 Трениране на модела за лицево верифициране	45
3.3.6 Изпълнение на алгоритъма за лицево верифициране	51
3.4 Свързване на устройствата към системата	55
3.4.1 Създаване на “Gateway Device”	55
3.4.2 Добавяне на камера към системата от свързани устройства	56
3.4.3 Добавяне на ключалка към системата от свързани устройства	57
3.5 Програмна реализация на управляващия код за устройствата	57
3.5.1 Инициализиране на MQTT клиент	57
3.5.2 Програмна реализация на управляващия код за камерата	58
3.5.3 Програмна реализация на управляващия код за ключалката	59
3.5.4 Програмна реализация на известията	59
3.6 Стартиране на системата от свързани устройства	60
3.7 Файлова структура на мобилното приложение	61
3.8 Програмна реализация на мобилното приложение	63
3.8.1 Структура на страниците	63
3.8.2 Управление на профил	64
3.8.3 Създаване на токен за Bosch IoT Suite	64
3.8.4 Управление на базата от данни	65
3.8.5 Изграждане на начална страница	65
3.8.6 Управление на камери	66
3.8.7 Управление на ключалки	66
3.8.8 Управление на известия	67

ГЛАВА IV. РЪКОВОДСТВО ЗА ПОТРЕБИТЕЛЯ	68
4.1 Стартиране на системата от свързани устройства	68
4.1.1 Подготвяне на Raspberry Pi	68
4.1.2 Инсталиране на функционалния код от Github	68
4.1.3 Създаване на изображения за трениране	69
4.1.4 Трениране на модела за лицево верифициране	69
4.1.5 Създаване на Subscription в Bosch IoT Suite	70
4.1.6 Инсталиране на Bosch IoT Edge Agent	70
4.1.7 Инсталиране на Bosch IoT Edge Services	71
4.1.8 Добавяне на камерата към системата	72
4.1.9 Добавяне на ключалката към системата	72
4.1.10 Създаване на дигитална репрезентация на “Gateway Device”	72
4.1.11 Стартиране на системата от свързани устройства	76
4.2 Стартиране на мобилното приложение	77
4.2.1 iOS	77
4.2.2 Android	77
4.3 Използване на приложението	78
4.3.1 Начална страница	78
4.3.2 Създаване и влизане в профил	79
4.3.3 Управление на камери	80
4.3.4 Управление на ключалки	81
4.3.5 Управление на известия	82
ЗАКЛЮЧЕНИЕ	84
ИЗПОЛЗВАНА ЛИТЕРАТУРА	85

УВОД

През последните години терминът “IoT” (Internet of Things – Интернет на нещата) набира голяма популярност, тъй като решения в тази сфера улесняват много нашето ежедневие. Вече съществуват продукти, които ни позволяват по-лесно да контролираме нашите домове, коли, градини и други. Интернет на нещата навлиза и в проекти, свързани със сигурността, отчитането и предаването на данни, както и с идентификация на лице, глас и пръстов отпечатък.

Машинното самообучение също е основен фактор в множество софтуерни и хардуерни решения. Развитието на машините позволява да бъдат тренирани по-сложни алгоритми с цел постигане на по-точни резултати. Машинното самообучение навлиза в света на технологиите все повече и повече, което ни позволява да заменим човешките усилия с работа на машини. То е използвано както за лични проекти, така и за глобални решения с оглед подобряване на услуги като градски транспорт, имейл, персонални асистенти, преводи и други.

Целта на настоящата дипломна работа е да бъде изградена система, която комбинира две от най-широко използваните технологии и която създава по-лесен и по-сигурен начин за достъп до нашия дом. Основните компоненти на дипломната работа са камера, която засича обекти, ключалка и алгоритъм за машинно самообучение, който проверява дали засечените обекти са хора и дали те попадат в списък с позволени хора, за да отключи вратата и да ги пусне в техния дом.

ГЛАВА I. ТЕХНОЛОГИИ ЗА РЕАЛИЗАЦИЯ НА СИСТЕМА ОТ СВЪРЗАНИ УСТРОЙСТВА

1.1 Лицево засичане

Лицевото засичане е технология, използвана в множество решения за идентифициране на човешки лица в изображения или във видео връзки на живо. То произлиза от засичането на обекти, като в случая търсените обекти са човешки лица. Един алгоритъм за лицево засичане се тренира върху множество изображения на различни човешки лица – различни полове, различни раси, различни черти на лицето. При изпълнение алгоритъмът обхожда пиксел по пиксел даденото изображение и сравнява пикселите със съществуващите снимки на лица, за да открие приликите между тях. В зависимост от приликите между потенциалното засечено лице и снимките на познатите лица, алгоритъмът “взема решение” дали даденият обект е лице или не, като резултатът е процентът сигурност, че разглежданият обект е лице. Съществуват алгоритми, които се тренират по време на изпълнение – при засечено лице алгоритъмът го добавя към множеството от познати лица. По този начин всяко следващо разпознато лице довежда до по-точни резултати. Лицевото засичане намира множество приложения в различни сфери – системи за лицево разпознаване, автоматичен фокус във фотографията, разпознаване на емоции, както и четене по устни. С развитието на алгоритмите и моделите за лицево засичане, то набира все по-голяма популярност в различни решения.

1.2 Лицево верифициране

Лицевото верифициране използва резултатите от лицевото засичане, за да сравни непознато лице с познати такива. За разлика от лицевото разпознаване, което отговаря на въпроса: “Чие е това лице”, лицевото верифициране отговаря на въпроса: “Това ли е правилното лице?”. Един алгоритъм за лицево верифициране сравнява характерните черти на

непознатото и познатите лица – разстоянието между зениците на двете очи, разстоянието между външния и вътрешния ъгъл на всяко око, разстоянието между носа и устата и други. Тази информация се записва като вектор със 128 измерения, а впоследствие се сравняват данните от всички измерения, за да се верифицира непознатото лице. Лицевото верифициране намира широко приложение в системи, свързани със сигурността, като това могат да бъдат както лични системи, така и публични такива. Лицевото верифициране се използва за контрол на достъпа до различни помещения, както и за отключване на нашите смартфони.

1.3 NumPy^[1]

NumPy е най-използваната библиотека за работа с многомерни масиви, написана на Python. Библиотеката предлага множество функционалности, свързани със създаване и обработване на масиви с много измерения. Освен това NumPy има много добре описана документация, която улеснява използването на функционалностите, които библиотеката предлага. Тя позволява запазването на такива масиви в специални файлове с разширение *.npz*, което представлява компресиран асоциативен масив. Друга широко използвана функционалност е че масивът, използван в NumPy, е обект, който съдържа и друга лесно достъпна информация като брой на измеренията, големината на всяко измерение и други.

1.4 Виртуални среди

В много случаи при разработка на приложение на скриптовия език Python се изисква инсталиране на външни библиотеки. Различните приложения обаче може да се нуждаят от различни версии на някоя библиотека, което означава, че при определена изтеглена версия някои приложения няма да функционират правилно. Решението на този проблем е да се създаде т.нар. виртуална среда, която съдържа в себе си необходимите библиотеки за съответното приложение. Най-често всяко приложение има

отделна виртуална среда, за да се предотврати конфликт на версии. Към всяка виртуална среда върви автоматично генериран текстов документ, в който са описани имената и версиите на всички външни библиотеки. Това позволява лесното пренасяне на всяко приложение през публично хранилище – всеки, който иска да стартира такова приложение, трябва да създаде виртуална среда на машината си и да изтегли всички библиотеки, описани в текстовия документ, като за тази цел за изпълняват две команди в конзолата и целият процес отнема до 1 минута.

1.5 MQTT^[2]

MQTT представлява стандартен протокол за комуникация, който намира широко приложение в проектите, свързани с “Интернет на нещата”. MQTT работи на принципа на публикуване и абониране за съобщения, което позволява едно съобщение да достига до множество крайни устройства. Всяко устройство може да се абонира за даден канал и да получава всички съобщения в този канал.

Протоколът дефинира два типа устройства – един брокер и много клиенти. Брокерът е центърът на комуникацията, като той отговаря за получаването на всяко съобщение и изпращането му до всеки абонат. Той контролира кои клиенти са свързани и кои клиенти към кои канали са се абонирали. Клиентите могат да бъдат много и те се свързват към даден брокер по IP. Те могат да се абонират към отделни канали и да получават съобщения, изпратени от други клиенти. В зависимост от приложението съобщенията могат да бъдат криптирани.

Най-силните страни на MQTT са високата скорост и ниската консумация. Една MQTT заявка се изпълнява над 10 пъти по-бързо в сравнение с една HTTP заявка. Освен това този протокол е с малко, но все пак достатъчни възможности, за да се постигнат необходимите функционалности за работа с устройства. MQTT заявките са с много малък размер, което позволява на устройства с ограничени възможности да използват услугите му.

Предимството на принципа на публикуване и абониране за съобщения е, че едно MQTT съобщение може да бъде получено от безброй много абонати, докато при HTTP ще трябва да се изпращат отделни заявки до всеки получател. Има много примери за използването на този протокол извън сферата на Интернет на нещата, като най-големият продукт, използващ MQTT за комуникация, е Messenger на Facebook. Той предоставя бързо изпращане и получаване на съобщения, като се използват много малко ресурси.

1.6 Дигитални близнаци

Дигитален близнак на едно устройство представлява съвкупността от информация, описваща съществуващо устройство. Терминът добива популярност в началото на XIX век с навлизането в ерата на Интернет на нещата. Дигиталният близнак може да се разглежда като база от данни за едно устройство, съдържаща само текущата информация. Съществуването на дигитални репрезентации позволява на хората да следят от разстояние данните от всяко устройство.

1.7 Raspberry Pi^[3]

Raspberry Pi е един от най-популярните микрокомпютри на пазара. То представлява по-немошен, но и по-малък и евтин компютър. Най-често тези микрокомпютри се използват в решения, свързани с Интернет на нещата, тъй като повечето имат вграден Wi-Fi модул и Bluetooth модул, което им позволява да обменят информация с различни устройства и да се създаде система от свързани устройства на сравнително ниска цена. Тези микрокомпютри нямат много памет, нито много добър процесор, но са способни да извършват достатъчно операции, свързани с комуникация и обработка на данни от устройства. Някои от по-новите модели (Raspberry Pi 3 и Raspberry Pi 4) имат възможност да изпълняват и някои сравнително леки модели за машинно самообучение. Всяко Raspberry Pi използва Linux

базирана операционна система, предоставяща на потребителите конзола, чрез която да управляват своя компютър. Едно от най-големите му предимства е, че съществуват множество форуми и видео уроци, които предлагат възможност на по-начинаещи програмисти да създадат свой проект, използвайки Raspberry Pi.

1.8 Bosch IoT Suite^[4]

Bosch IoT Suite представлява платформа за свързване на устройства, тяхното контролиране, както и обработването на данните, които изпраща всяко устройство. Услугите в Bosch IoT Suite са разделени спрямо функционалността си – свързване на устройства, дигитална репрезентация на устройства, управление на устройства, подновяване на софтуера върху устройства и анализ на данните им. Към днешна дата над 15 милиона устройства са свързани към Bosch IoT Suite. Също така над 250 международни проекта използват услугите, предлагани от тази платформа.

1.9 Bosch IoT Hub^[5]

Bosch IoT Hub представлява облачна услуга, която свързва устройства с приложения, които ги управляват. Чрез тази услуга разработчиците на софтуер получават бърз, лесен и най-вече сигурен начин да свържат различни устройства към своите приложения. Тя поддържа няколко протокола за комуникация, част от които са HTTP, MQTT и AMQP.

1.10 Bosch IoT Things^[6]

Bosch IoT Things представлява облачна услуга, която позволява на всеки потребител да съхранява и обработва дигиталните близнаци на устройствата си. Това може да се случва през сайта на Bosch IoT Things или чрез HTTP заявки. Функционалностите на услугата са регистриране и обработване на дигитални близнаци, контролиране на достъпа до тях, както и търсене на дигитални близнаци по динамични стойности.

1.11 Bosch IoT Edge Services^[7]

Bosch IoT Edge Services е услуга, която осъществява връзката към устройствата в една система. Тя поддържа комуникации посредством най-различни протоколи – Bluetooth, ONVIF, ZigBee, Z-Wave и други. Освен това услугата предоставя и други функционалности – история и статистика на данните от устройствата, система за правила за устройствата и други

1.12 Bosch IoT Edge Agent^[8]

Bosch IoT Edge Agent е услуга, която осъществява връзката към облачната услуга Bosch IoT Hub. Комуникацията се осъществява посредством MQTT, като при стартирането на услугата се инициализира MQTT клиент за получаване и изпращане на съобщения към Bosch IoT Hub, както и MQTT брокер за локално управление на съобщенията.

1.13 Gateway Device

“Gateway Device” е машината, върху която има инсталирани услугите Bosch IoT Edge Services и Bosch IoT Edge Agent. Той представлява връзката между устройствата и техните дигитални близнаци. В една IoT система от свързани устройства “Gateway Device” се явява ядро на системата, тъй като свързва всички нейни компоненти.

1.14 Умни устройства

През последните години всички устройства, използвани в ежедневието, биват заменени от умни такива. Това представляват устройства или сензори, които са свързани помежду си и които могат да се използват интерактивно. Съществуват множество протоколи за комуникация между отделни устройства, като най-известните са Bluetooth, Wi-Fi, Z-Wave, ZigBee, ONVIF и други. Умни устройства могат да бъдат телефони, автомобили, часовници, камери, хладилници, ключалки и други. Появата им улеснява човешкото ежедневие и тяхната употреба става необходима във все повече индустрии.

1.15 Z-Wave^[10]

Z-Wave представлява протокол за безжична комуникация, като се използва в решения, свързани с Интернет на нещата. Най-често протоколът се използва, за да се осъществи комуникацията с устройства в дома като автоматични ключалки, автоматични прозорци, автоматични врати за гаражи, термостати и други. В реални условия вълните достигат до 30-40 метра, което позволява пълен контрол в дома и градината.

1.16 ONVIF^[11]

ONVIF е глобална организация, която цели да стандартизира работата с базираните на IP устройства, свързани със сигурността. Създаденият стандарт описва връзката и комуникацията между различни устройства за видеонаблюдение и централни машини, които следят тяхната работа и потока, който записват. Друга основна цел на организацията е да се осигури съвместимостта между продукти на различни компании, за да се постигне по-сигурна и стандартна система.

1.17 Flutter^[12]

Flutter е пакет за разработка на софтуер, създаден от Google. Първата версия излиза през 2017 г. и за кратко време става една от най-използваните платформи за създаване на мобилни приложения. Във Flutter се използва обектно-ориентираният език за програмиране Dart. Основната единица в едно Flutter приложение е т.нар “widget”, което представлява клас за даден компонент в приложението (бутон, параграф, изображение), който има отделни параметри и функционалности. Едно от най-големите предимства на Flutter е, че позволява компилиране за iOS и Android, което означава, че може да се създаде едно приложение на един език за двете операционни системи. Освен това съществуват множество курсове и форуми за Flutter, което помага на по-начинаещи програмисти да направят своите първи стъпки в разработката на мобилни приложения.

1.18 Firebase^[13]

Firebase е платформа, разработена от Google за създаване на всякакъв тип приложения. Платформата съдържа 18 продукта, които улесняват разработката на приложения. Всеки продукт има отделна документация, в която е описан процесът на инсталиране и функционалностите, които предлага. Най-често процесът на инсталиране изисква няколко реда код, което помага на разработчиците да се фокусират върху функционалностите на приложението, което създават.

1.19 Firebase Authentication^[14]

В почти всички приложения се изисква имплементирането на някакъв вид автентикация. Firebase Authentication позволява да се вградят над 10 вида автентикация в приложението – чрез потребителско име и парола, чрез Facebook, Github, Twitter, Apple ID, Microsoft и други. Firebase позволява изграждането на база от данни, в която се пазят потребителските имена и пароли за използване от Firebase Authentication. Този продукт поддържа сесии, което позволява по всяко време да се провери дали някой потребител е влязъл в своя профил.

1.20 Cloud Storage for Firebase^[15]

Firebase Cloud Service е услуга, създадена от Google, която се използва от мобилни и уеб приложения за съхраняване на изображения и видеоклипове. Добавянето на услугата в едно приложение отнема малко време и изисква малко стъпки, но в същото време позволява на разработчиците лесен и удобен начин за съхраняване на снимков материал. Съществуват приложно-програмни интерфейси за много езици за програмиране, което позволява най-различни проекти да използват Cloud Storage.

1.21 Cloud Firestore^[16]

Cloud Firestore е облачна услуга, създадена от Google, която представлява база от данни, която може да се използва в приложения, написани на най-различни програмни езици. Структурата на базата от данни е NoSQL, като информацията в нея се съхранява под формата на дървовидна структура и документи. Услугата позволява на приложения да следят промени в реално време.

ГЛАВА II. ИЗИСКВАНИЯ КЪМ ПРИЛОЖЕНИЕТО. СТРУКТУРА НА СИСТЕМАТА ОТ УСТРОЙСТВА

2.1 Изисквания към приложението

2.1.1 Изисквания към модела за лицево верифициране

- Моделът за лицево верифициране да включва програма, която да позволи на потребителя да отвори камерата на машината си и да снима изображения на потребителя, с които да се тренира алгоритъмът.

- Моделът за лицево верифициране да включва модел за лицево засичане, който да засича всички лица на хора в едно изображение и да ги запазва в масив.

- Процесът на лицево верифициране да прекъсва изпълнението си, ако няма засечени лица в изображението.

- Моделът за лицево верифициране да включва програма, която да тренира алгоритъма, използвайки заснетите снимки на познати лица.

- Моделът за лицево верифициране да включва програма, която да приема изображение чрез линк или чрез файл с възможни разширения *.jpg*, *.png* и *.jpeg*. Програмата трябва да засече всички лица в изображението и да провери кои лица са на познати хора. Алгоритъмът трябва да връща масив с познатите засечени лица, “Unknown”, ако сред засечените лица няма нито едно познато, или “No faces detected”, ако няма засечени лица.

2.1.2 Изисквания към системата от свързани устройства

- Да се създаде акаунт в Bosch IoT Suite и да се създаде безплатен Subscription, използвайки пакет Bosch IoT Suite for Device Management.

- Върху Raspberry Pi да се инсталира Bosch IoT Edge Services, за да може да се свържат устройствата в системата към това Raspberry Pi.

- Системата да включва камера, която да засича движение, да прави снимка и да изпраща линк на снимката към Raspberry Pi, за да се верифицират човешките лица, ако има такива.

- Системата да включва ключалка, която да се отключва за 5 секунди, ако моделът за лицево верифициране разпознае лице. Съобщенията към ключалката да се изпращат и получават посредством Z-Wave контролер.

- Върху Raspberry Pi да се инсталира Bosch IoT Edge Agent.

- Да се създаде “Provisioning” на “Gateway Device”, който да позволява автоматично създаване на дигитални близнаци от Raspberry Pi.

- Да се създаде скрипт, който се свързва към локалния MQTT брокер, за да изпраща и получава съобщения към облачната услуга Bosch IoT Suite.

- Да се създаде функция, която извлича изображение от камерата.

- Да се създадат функции, които отключват и заключват ключалката.

2.1.3 Изисквания към мобилното приложение

- Да се имплементира създаване на профил и влизане в съществуващ такъв.

- Да се създаде функция, която изпраща заявка за вземане на токен за достъп от Bosch IoT Suite и го съхранява в рамките на приложението.

- Да може да се изпращат HTTP заявки към Bosch IoT Suite с цел отключване и заключване на ключалката.

- Да може да се получават известия, когато камерата засече движение.

- Да може да се изпраща заявка за вземане на изображение от камерата.

2.2 Съображения за избор на програмни средства

2.2.1 Алгоритми за машинно самообучение

За реализация на алгоритмите за лицево засичане и лицево верифициране се използва скриптовият език Python. Има много причини, защо Python е най-използваният език за реализиране на алгоритми за машинно самообучение. Този език е сред първите 10 на най-използваните в програмирането. Освен това Python е скриптов език и за изпълнението на програма, написана на Python, не се изискват допълнителни процеси като компилация, както при програмни езици като C и Java. Друго предимство на Python е, че може да се поддържа от всякакви операционни системи, дори и на микроконтролери. Освен това съществуват множество библиотеки за машинно самообучение, написани на Python, което позволява на разработчиците на код да създават всякакви приложения, използвайки Python и съществуващите библиотеки.

2.2.2 Устройства в системата

Системата от свързани устройства включва камера и ключалка, свързани към Raspberry Pi посредством Bosch IoT Edge Services. Използва се този микроконтролер, тъй като той разполага с достатъчно добър процесор, за да поддържа връзките към устройствата, връзката към облачната услуга Bosch IoT Hub и да изпълнява тренирания алгоритъм за лицево верифициране.

Камерата, използвана за реализацията на дипломната работа, е Bosch FLEXIDOME IP 4000i. Камерата използва ONVIF за комуникация, което позволява лесното ѝ свързване и контролиране от друга машина. Тя съдържа сензор за движение и сигнализира, когато има засечен обект от сензора. Камерата може да изпраща линк към изображение и линк към видео поток, които могат да се достъпят с автентикация посредством потребителско име и парола.

Ключалката, използвана за реализацията на дипломната работа, е Danalock V3 Smartlock. Тя може да бъде свързана към смартфон или към

смарт часовник и да се контролира от съответното устройство. Ключалката използва протоколите Bluetooth и Z-Wave за комуникация. При използването на Z-Wave се изисква допълнителен Z-Wave контролер, тъй като смартфоните и компютрите нямат вграден такъв. Освен създателите на тази ключалка са разработили и мобилно приложение, което може да се свърже с нея и да я контролира.

2.3 Основни алгоритми за лицево засичане и лицево верифициране

2.3.1 Алгоритми за лицево засичане

Съществуват много алгоритми, създадени и оптимизирани специално за целта да засичат лица в снимки. Според много източници MTCNN е един от най-добрите модели откъм ефективност и постигнати резултати. Този модел се състои от 3 конволюционни невронни мрежи, като освен лица в снимка, те могат да открият и координатите на характеристични точки на лице на човек – двете очи, носа и двата края на устата. MTCNN постига почти перфектни резултати дори в изображения с по-ниско качество или с по-малки лица.

2.3.2 Алгоритми за лицево верифициране

Както при лицевото засичане, така и при лицевото верифициране невронните мрежи водят до най-точни резултати. Един от най-известните модели за лицево верифициране е VGG-Face, разработен от специалисти от Оксфордски университет. Алгоритъмът за лицево верифициране приема изображение, съдържащо лице и създава т.нар. “*face embedding*”, което представлява вектор в 128 измерения, като всяко измерение представлява различна част от този *face embedding*. Този вектор пази информация за много разстояния между характеристичните части на човешкото лице – разстоянието между зениците на двете очи, разстоянието между вътрешния и външния ъгъл на всяко око и други.

При трениране на алгоритъма за лицево верифициране се използва метода “*triplet loss*”. За неговата реализация се използват три изображения на

човешки лица – базова снимка, вярна снимка и грешна снимка. Базовата снимка е снимката, върху която се тренира, вярната снимка е на същото лице, а грешната снимка е на друго лице. По този начин алгоритъмът се “учи” как да разграничава отделните лица по стойностите от различните измерения на вектора *face embedding*. Функцията на *triplet loss* изглежда така:

$$L(A, P, N) = \max(\|f(A) - f(P)\| - \|f(A) - f(N)\| + \alpha, 0)$$

Формула 2.1: Формула за функцията на *triplet loss*

В тази формула A е базовата снимка, P е вярната снимка, а N е грешната. Функцията f представлява функцията за извличане на *face embedding* от изображение на лице, а α е допустимата граница между вярната и грешната снимка. Тази граница се въвежда, за да има по-голямо разграничение между отделни лица и да може да се разграничават хора, които си приличат в лицето. За трениране на алгоритъма за лицево верифициране се използва следната формула:

$$J = \sum_{i=1}^M L(A^{(i)}, P^{(i)}, N^{(i)})$$

Формула 2.2: Функция за трениране на модела за лицево верифициране

Резултатът от тази формула е сумата от резултатите на функцията L с всички снимки от колекцията от снимки за трениране. Този резултат се записва във файл с разширение *.npz* и впоследствие се използва при изпълнението на алгоритъма върху тестови лица.

2.4 Структура на Flutter мобилно приложение

Във Фиг. 2.3 е показана примерна файлова структура на едно Flutter приложение, което е разположено в папката “flutter-application”.

```

flutter-application /
├── android/
│   ├── app/
│   ├── build.gradle
│   └── gradle/
├── build/
├── fonts/
├── ios/
│   ├── Flutter/
│   ├── Podfile
│   ├── Pods/
│   └── Runner/
├── lib/
│   ├── main.dart
│   ├── models/
│   ├── screens/
│   └── services/
├── pubspec.yaml
└── README.md

```

Фиг. 2.3: Примерна файлова структура на Flutter приложение

В папката “android” е поместено приложението за устройства с Android операционна система. В подпапката “app” се намира функционалният код на мобилното приложение, а в подпапката “gradle” се намира кодът за сглобяване на приложението. Файлът “build.gradle” съдържа в себе си константи, които се използват при сглобяването на приложението.

В папката “build” са поместени сглобените приложения за Android OS и iOS, както и сорс код на изтеглените външни библиотеки.

В папката “fonts” се поставят семействата от шрифтове, използвани в приложението.

В папката “ios” се намира приложението за iOS устройства. В подпапката “Flutter” е поместена логиката за сглобяване на мобилното приложение. В подпапката “Pods” се намират необходимите външни библиотеки за сглобяване на приложението, а в подпапката “Runner” се

пазят конфигурациите за стартирането му върху реално устройство. Във файлът “Podfile” се пазят имената и версиите на необходимите външни библиотеки, които при сглобяване се изтеглят в папката “Pods”.

В папката “lib” е поместен основният функционален код на приложението, написан на програмния език Dart. Там се намира кодът, написан от разработчика на едно мобилно приложение. При сглобяване на приложението кодът, написан на Dart, се “превежда” на Swift (програмният език за разработване на приложения за iOS) и Kotlin (програмният език за разработване на приложения за Android OS), за да могат да се стартират приложенията върху двете операционни системи. Подпапката “models” съдържа различни помощни класове, които се използват в приложението. В подпапката “screens” са поместени класовете за различните екрани в мобилното приложение. В подпапката “services” се намират различни помощни класове, свързани с функционалностите на приложението. Файлът “main.dart” е файлът, който съдържа началната страница на приложението и който се изпълнява при неговото стартиране.

2.5 Структура на базата от данни на мобилното приложение

Базата от данни на мобилното приложение е разпределена между двете услуги на Firebase – Cloud Firestore и Cloud Storage. В Cloud Firestore се съхранява информацията за устройствата и за известията, докато в Cloud Storage се съхраняват само изображения. Структурата на двете бази е NoSQL, което означава че данните се съхраняват йерархически под формата на колекции и документи.

Във Firebase Cloud Firestore базата от данни известията се съхраняват в отделна колекция с наименование “notifications”, като в нея се съхраняват документи за всяко известие. Един документ (Фиг. 2.4) съдържа уникален идентификатор на известието, който се използва при запазването и извличането на изображението към известието от Cloud Storage, идентификатор на камерата, час и списък с имената на засечените лица.

Notification	
ID (PK)	int
camera_uid	string
names	array<string>
time	int

Фиг. 2.4: Структура на документ за известието

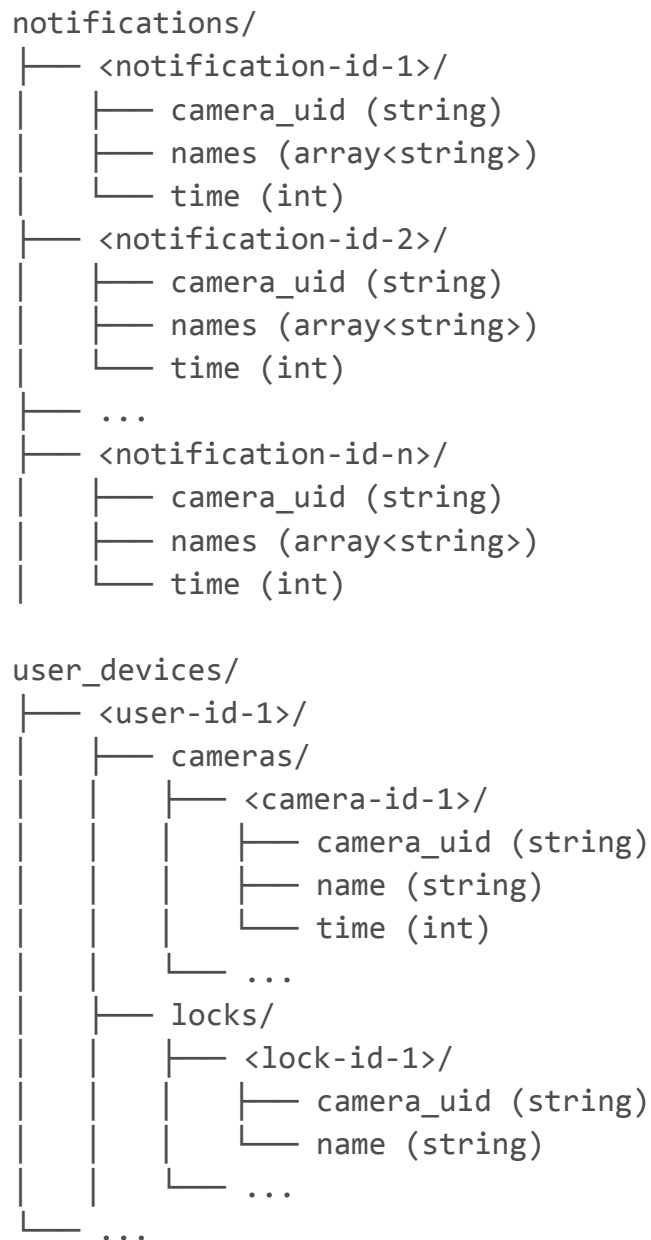
Данните за потребителските устройства се съхранява в колекцията “user_devices”. В нея е създаден отделен документ за всеки потребител, като той съдържа по две колекции – “cameras” и “locks”, които съхраняват данните за камерите и ключалките на потребителя. Структурата на документите за камера и ключалка са изобразени във Фиг. 2.5.

Camera		Lock	
ID (PK)	int	ID (PK)	int
camera_uid	string	lock_uid	string
name	string	name	string
time	int		

Фиг. 2.5: Структура на документите за камера и ключалка

Единствената разлика между документите на камера и ключалка е, че документът за камера съдържа допълнително поле “time”, което обозначава часа, в който е получено изображението след изричното му поискване от потребителя.

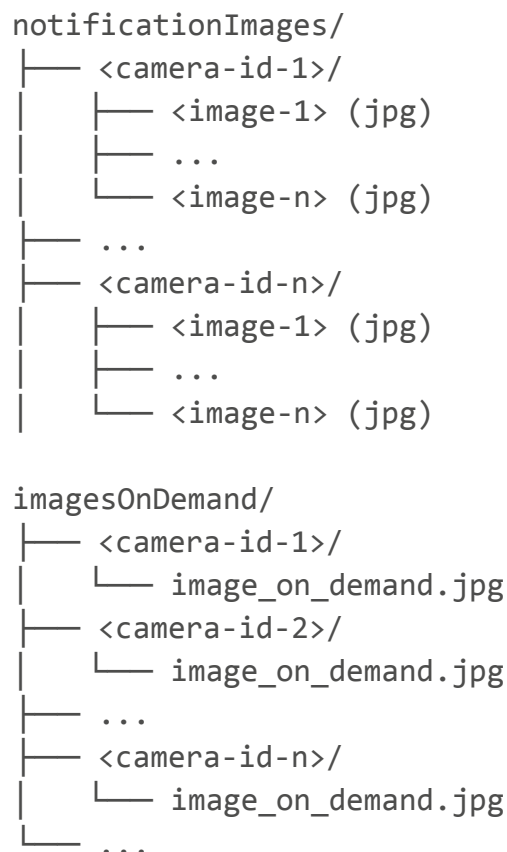
Структурата на цялата база от данни, съхранена в Cloud Firestore е изобразена във Фиг. 2.6.



Фиг. 2.6: Структура на базата от данни в Firebase Cloud Firestore

В документът за камери “camera-id” е уникалният идентификатор, който услугата дава на документа, докато “camera_uid” е идентификатора на дигиталния близък на камерата в Bosch IoT Suite. По този начин повече от един потребител могат да притежават една камера, без да използват само един акаунт.

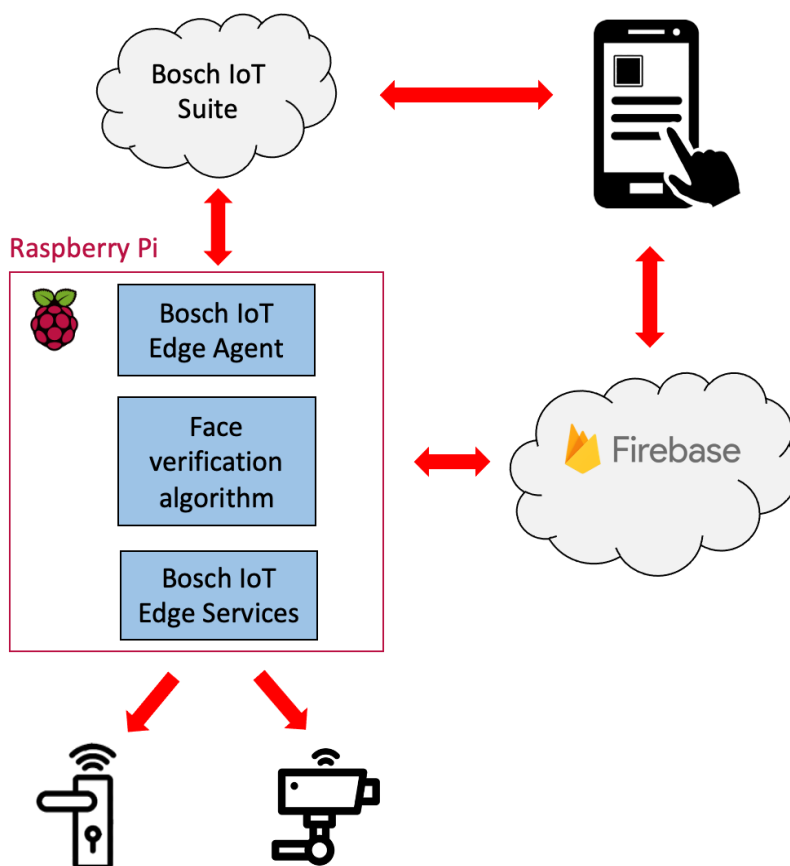
В Cloud Storage изображенията са пазят отново в две колекции – “notificationImages” и “imagesOnDemand”. Те са съставени от множество подколекции, които са за отделните камери. В подколекциите на “notificationImages” се съхраняват всички изображения, които са запазени при изпращането на известие от системата от свързани устройства. В подколекциите на “imagesOnDemand” се съхранява само по едно изображение с наименование “image_on_demand.jpg”, което се обновява, когато потребителя поиска ново изображение от камерата. По този начин се приложението се предпазва от бързо запълване на пространството, предоставено от Firebase в услугата Cloud Storage. Структурата на цялата база от данни, съхранена в Firebase Cloud Storage, е изобразена във *Фиг. 2.7*.



Фиг. 2.7: Структура на базата от данни в Firebase Cloud Storage

2.6 Структура на системата от свързани устройства

Структурата на системата от свързани устройства (Фиг. 2.8) се състои от няколко отделни софтуерни и хардуерни компонента. Основата на системата е микрокомпютъра Raspberry Pi, върху който са инсталирани услугите Bosch IoT Edge Agent, Bosch IoT Edge Services и алгоритъмът за лицево верифициране. Посредством Bosch IoT Edge Services са свързани камерата и ключалката към системата, а Bosch IoT Edge Agent свързва системата с облачната услуга Bosch IoT Suite, където се съхраняват дигиталните близнаци на устройствата. Освен това микрокомпютъра изпраща данни към облачните услуги Firebase Cloud Storage и Cloud Firestore, които се използват от мобилното приложение, така че потребителят да има отдалечен достъп до своите устройства. Той може да ги контролира чрез Bosch IoT Suite, като изпраща съобщения през него към микроконтролера, който изпълнява съответните операции върху устройствата.



Фиг. 2.8: Структура на системата от свързани устройства

ГЛАВА III. ПРОГРАМНА РЕАЛИЗАЦИЯ НА СИСТЕМА ОТ СВЪРЗАНИ УСТРОЙСТВА

3.1 Файлово и функционално разпределение на приложението

Приложението е съставено от няколко отделни модула, както и от няколко конфигурационни файла, които се изпълняват при стартиране на програмата. Основната файлова структура на сорс кода е изобразена във *Фиг. 3.1*.

```
src/  
├── clean.sh  
├── install.sh  
├── iot_system/  
├── mobile/  
├── pc.txt  
└── requirements.txt
```

Фиг 3.1: Файлова структура на кода на проекта

Директориите “*iot_system*” и “*mobile*” са двата модула на проекта – кодът, управляващ схемата от свързани устройства и кодът за мобилното приложение. Файлът “*requirements.txt*” съдържа необходимите външни библиотеки за изпълняване на приложението. В него на отделен ред са дефинирани имената на всички библиотеки, както и техните версии, които да се изтеглят.

За реализацията на дипломната работа е използван микроконтролерът Raspberry Pi, като върху него се изпълнява алгоритъмът за лицево верифициране, то е свързано към устройствата и към облачните услуги на Bosch. Тъй като Tensorflow версия 2 не е излязла официално за Raspbian OS, процесът за инсталиране на Tensorflow за Raspberry Pi е по-различен от този за Windows или Linux операционни системи. За инсталиране на необходимите външни библиотеки се изпълнява скриптът “*install.sh*”,

изобразен във *Фиг. 3.2*, който инсталира библиотеките в зависимост от операционната система, върху която работи потребителят.

```
#!/bin/bash
if [[ $OSTYPE == "linux-gnueabi" ]]
then
    python3 -m pip install --upgrade pip
    python3 -m pip install virtualenv
    python3 -m virtualenv env
    python3 -m pip install -r requirements.txt
    python3 -m pip install
    https://github.com/bitsy-ai/tensorflow-arm-bin/releases/download/v2.4.0-rc2/tensorflow-2.4.0rc2-cp37-none-linux_armv7l.whl
else
    python3 -m pip install --upgrade pip
    python3 -m pip install pipenv
    pipenv install -r pc.txt
fi
```

Фиг 3.2: Скриптът install.sh

За намиране на операционната система на машината се използва предефинирана константа “OSTYPE”. Независимо от операционната система се инсталира последната версия на PIP (Package Installer for Python), който е най-широко разпространеният инсталатор на пакети за Python. Tensorflow 2 се нуждае от минимална версия 20.0 на PIP, затова се инсталира последната версия, която към момента на разработване на тази дипломна работа е 21.0.1. Скриптът инсталира и виртуална среда, в която ще инсталира всички библиотеки, необходими за изпълнението на приложението. При Raspbian OS библиотеките се инсталират от “requirements.txt”, а за всички останали операционни системи се инсталират от “pc.txt”, който инсталира всичко от “requirements.txt” плюс Tensorflow 2 за Windows или Linux OS. В “install.sh” се инсталира отделно Tensorflow 2 за Raspbian OS.

За удобство на потребителя е създаден скриптът “clean.sh”, изобразен във *Фиг. 3.3*, който премахва виртуалната среда, създадена за проекта и изтрива от машината на потребителя всички инсталирани библиотеки за стартирането на приложението. В този скрипт отново се проверява видът на операционната система, тъй като процесът по деинсталиране на библиотеките се различава.

```
#!/bin/bash
if [[ $OSTYPE == "linux-gnu*" ]]
then
    rm -r env
else
    pipenv --rm
fi
```

Фиг. 3.3: Скриптът clean.sh

Тъй като всички библиотеки, от които приложението зависи, са инсталирани във виртуална среда, то останалите приложения на потребителя няма да бъдат засегнати по никакъв начин.

3.2 Файлова структура на управляващия код за системата от свързани устройства

Основната директория, в която се съхранява функционалният код, управляващ системата от свързани устройства, е с наименование “`iot_system`”. В приложението отделните компоненти са разделени в отделни директории, като има една за алгоритъма за лицево верифициране (“`face_verification`”), една за управляващия код за ключалката (“`door_lock_controller`”), една за управляващия код за камерата (“`camera_controller`”) и една за управление на известията към мобилното приложение (“`notification_controller`”). Освен това папката съхранява и основният скрипт “`main.py`”, отговорен за стартиране на системата. Файловата структура на директорията “`iot_system`” е изобразена във *Фиг. 3.4*.

```
iot_system/  
├── camera_controller/  
├── door_lock_controller/  
├── face_verification/  
├── notification_controller/  
├── models/  
├── __init__.py  
└── main.py
```

Фиг. 3.4: Файлова структура на кода за системата от устройства

Подпапката “`models`” съдържа данните от тренирането на модела за лицево верифициране. Файлът “`__init__.py`” превръща директорията “`iot_system`” в Python пакет, което позволява на всеки скрипт да използва функциите, дефинирани в останалите Python програми. Скриптът “`main.py`” е основният скрипт, който се изпълнява при стартиране на системата от свързани устройства. Този файл е нужен, за да може пакетът “`iot_system`” да бъде най-високото ниво при изпълнението на приложението, което позволява достъп до всички Python програми, разположени в директорията “`iot_system`” и във всяка нейна поддиректория.

3.3 Програмна реализация на алгоритъма за лицево верифициране

3.3.1 Файлова структура

В директорията “face_verification” Python кодът е разпределен в отделни файлове според функционалностите, които изпълнява. Файловата структура на директорията е изобразена във *Фиг. 3.5*:

```
face_verification/  
├── candidate_face_check.py  
├── dataset/  
│   ├── dataset_loader.py  
│   ├── face_embedding_utils.py  
│   ├── face_extract_utils.py  
│   ├── recognize.py  
│   └── train.py
```

Фиг. 3.5: Файлова структура на кода на алгоритъма за лицево верифициране

3.3.2 Създаване на изображенията за трениране

Първата стъпка за създаване на модел за лицево верифициране е да се създаде набор от снимки, върху които да се тренира моделът. Моделът трябва да бъде добре запознат с лицата на хората, на които ще отключва вратата, затова се нуждае от немалък брой изображения. В контекста на тази дипломна работа познатите лица са не повече от 5 и моделът се тренира върху 10 снимки за всяко лице. Всички изображения се съхраняват в директорията “dataset”, като тя е разделена на подпапки, които са имената на хората. При качване на кода в хранилище, се качва празна папката “dataset”, така че всеки, който иска да използва приложението да може да създаде колекция от свои снимки, без да се занимава да трие чужди такива. Python програмата, отговорна за създаване на колекцията от снимки е с името “dataset_loader.py”. При изпълнение на програмата се подават двете имена на човека, който ще си прави снимки, така че неговите снимки да бъдат запазени в папка с неговото име, като има 2 начина, по които могат да се подадат аргументите, които са показани във *Фиг. 3.6*.

```
$ python3 dataset_loader.py -f Ognian -l Baruh
$ python3 dataset_loader.py --fname Ognian --lname Baruh
```

Фиг. 3.6: Примерни команди за стартиране на програмата за създаване на изображения

При изпълнение на програмата с грешни аргументи или без такива се изкарва грешка, изобразена във *Фиг. 3.7*. Грешката показва на потребителя правилните начини за стартиране на програмата.

```
$ python3 dataset_loader.py
Wrong usage!
Correct usage:
$ python3 dataset_loader.py -f <FirstName> -l <LastName>
or
$ python3 dataset_loader.py --fname <FirstName> --lname
<LastName>
```

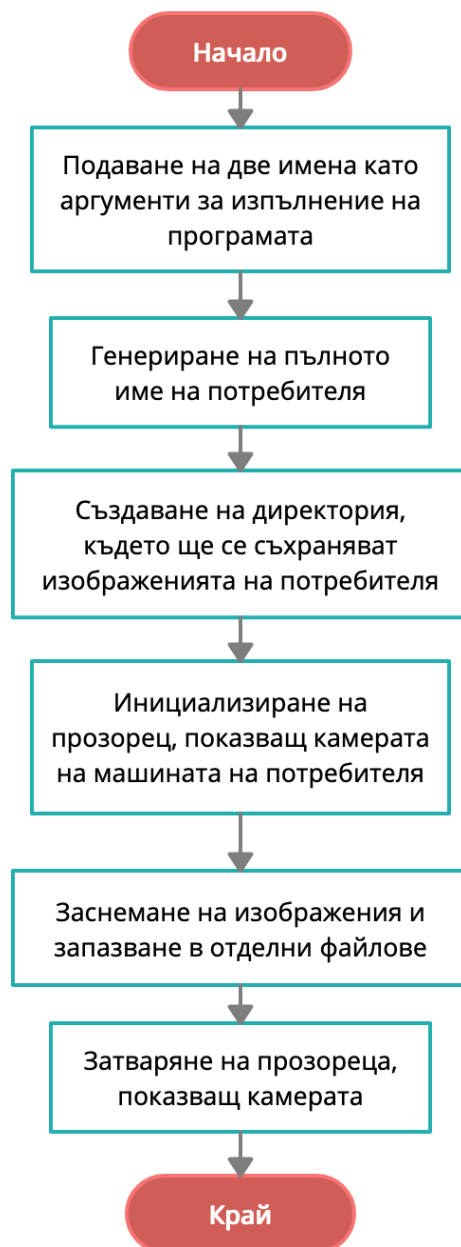
Фиг. 3.7: Съобщение за грешка при изпълнение на програмата за създаване на изображения

При изпълнение на програмата с “-h” или “--help” се изкарва наръчник за стартиране на програмата, който е изобразен във *Фиг. 3.8*.

```
$ py dataset_loader.py -h
-h, --help      ->  shows this screen
-f, --fname     ->  first name of the person
-l, --lname     ->  last name of the person
Note: There can't be two people with the same full name!
```

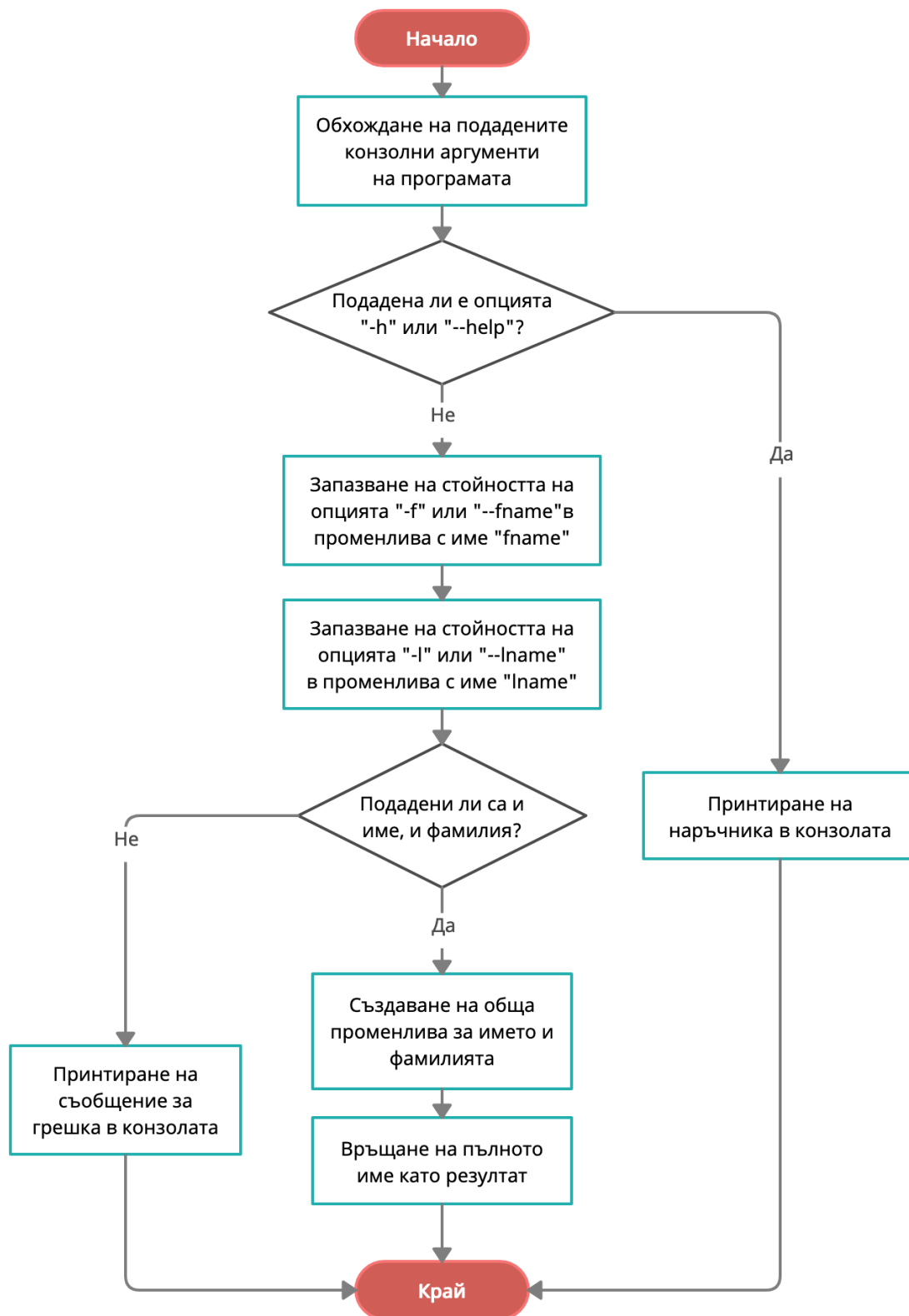
Фиг. 3.8: Наръчник за изпълнение на програмата за създаване на изображения

При нормално изпълнение на програмата процесът за създаване на изображение за трениране е показан на *Фиг. 3.9*.



Фиг. 3.9: Блокова схема на програмата за създаване на изображения

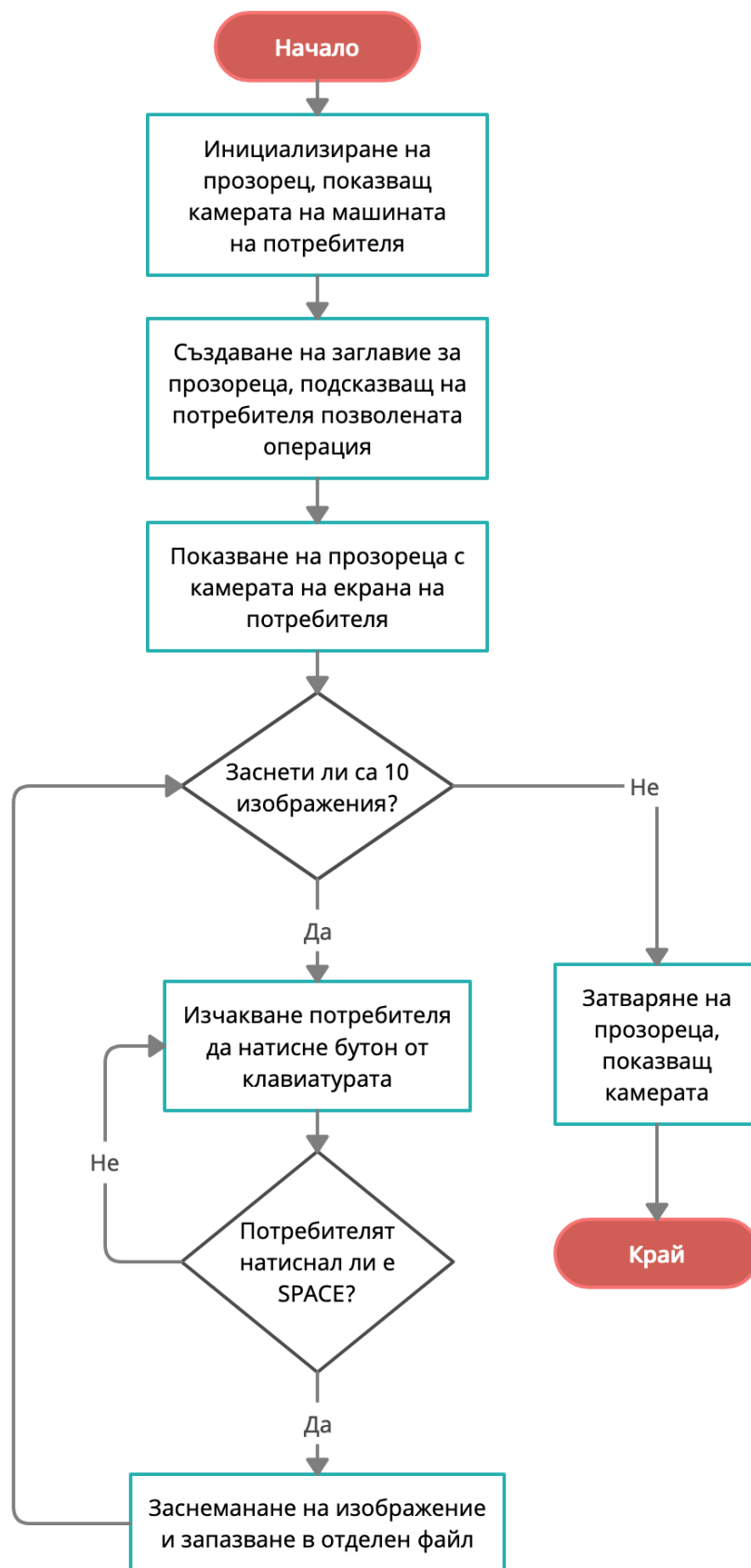
Първата функция, която се изпълнява при стартиране на програмата е функцията “get_name”, показана във Фиг. 3.10, която има за цел да обработи подадените аргументи от потребителя и да генерира пълното име, събирайки първото и последното име на потребителя. Обработката на аргументите се осъществява чрез библиотеката “getopt”, която позволява лесно използване на подадените аргументи.



Фиг. 3.10: Блокова схема на функцията “get_name”

Втората функция е с наименование `“create_dir”` и има за цел да създаде директория, в която ще се пазят снимките на потребителя, като името на тази директория е името на потребителя, което той е подал при стартиране на програмата. Като аргумент на функцията се подава пълното име на потребителя, като в рамките на функцията това име се добавя към константата `“PATH”`, в която е зададен пътят към папката, съдържаща всички изображения за трениране на модела за лицево верифициране. Създаването на директорията става посредством функцията `“mkdir”` от вградената библиотека `“os”`. Като резултат функцията връща пълния път към директорията, където ще се съхраняват потребителските снимки.

Последната функция в програмата е с най-голяма функционалност, като тя е отговорна за снимането на потребителя. За тази цел се използва библиотеката `OpenCV`, която позволява да се отвори отделен прозорец, в който да се покаже камерата на устройството на потребителя (ако потребителят използва настолен компютър, то ще трябва да свърже камера или да превключи на лаптоп). Функцията приема като аргумент пълния път към директорията, където ще се запазят изображенията. Снимките се заснемат с натискане на бутона `“SPACE”`. Програмата изпълнява цикъл 10 пъти, което позволява на потребителя да създаде 10 изображения на своето лице. Те биват запазени в създадената директория за снимки на потребителя, като името на всяко едно от изображенията е `“image_<пореден-номер-на-изображението>.jpg”`. При приключване на програмата се затваря прозорецът с камерата и всички направени снимки се запазват. Наименованието на функцията е `“create_images”` и тя е изобразена във *Фиг. 3.11*.



Фиг. 3.11: Блокова схема на функцията “create_images”

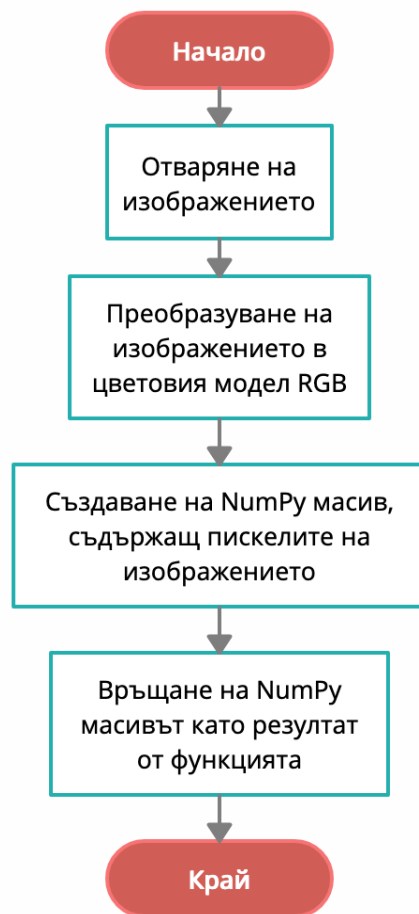
3.3.3 Помощни функции за лицево засичане

При изпълнението на алгоритъм за лицево верифициране или лицево разпознаване първата стъпка е да се отделят всички лица от оригиналната снимка. Този процес включва изваждане на лицето от снимката и изправянето му, така че лицето да не е под определен ъгъл. Резултатът от алгоритъм за лицево засичане са координатите на правоъгълник, в който се намира лицето. Посредством тези координати, правоъгълникът може да бъде отрязан от оригиналната снимка и да бъде запазен в нов файл или да бъде използван по време на изпълнението на програмата. За реализацията на проекта е създаден отделен файл, съдържащ всички функционалности, свързани с алгоритъма за лицево засичане, чието име е `“face_extract_utils.py”`. В жизнения цикъл на приложението този скрипт не се изпълнява директно, а функции му се използват в други програми като помощни функции. `“face_extract_utils.py”` съдържа 5 помощни функции, като те се използват както в тренирането на алгоритъма за лицево верифициране, така и в неговото изпълнение.

Първите две помощни функции във файла са `“get_pixels_from_file”` и `“get_pixels_from_url”` (Фиг. 3.12), като те служат за отваряне на изображението и запазването на неговите пиксели в триизмерен масив. Единствената разлика между двете функции е, че първата отваря изображението от файл, докато втората отваря изображението от линк. Функцията `“get_pixels_from_file”` приема като аргумент името на файла и връща като резултат пикселите на съответното изображение. `“get_pixels_from_url”` приема като аргумент линк, а за линк по подразбиране се използва линка към изображението от камерата. Първата функция отваря директно изображението посредством функцията `“Image.open()”`, докато втората изпраща GET заявка, за да получи изображението. Заявката се изпраща към специален линк на камерата, който връща като отговор заснетото изображение.

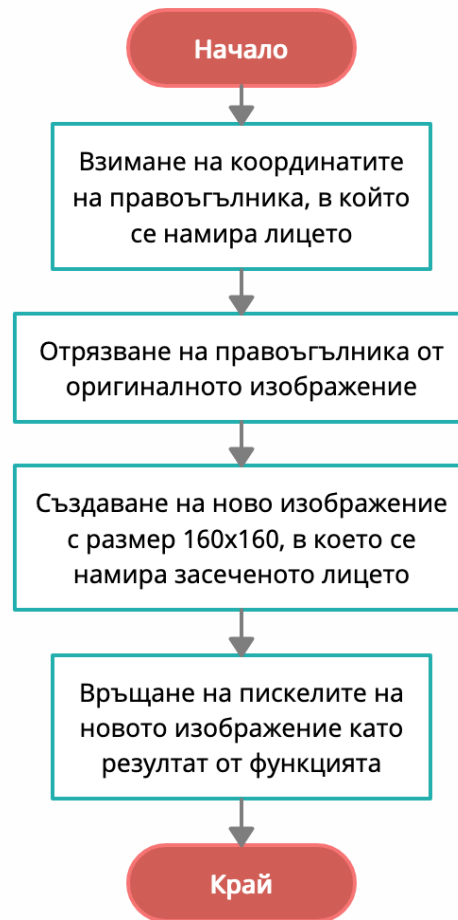
Двете функции за отваряне на изображения конвертират снимката в цветна, за да могат да се използват и черно-бели. Това е необходимо, тъй като всеки модел за машинно самообучение има предварително дефинирани размери на входните данни. Моделът за лицево верифициране очаква изображения с дължина 160 пиксела, ширина 160 пиксела и с 3 стойности за цвят, които има само цветовият модел RGB (Red – червено, Green – зелено, Blue – синьо). В повечето случаи черно-белите снимки имат само една стойност за цвят от 0 до 255, тъй като в цветовия модел RGB сивият цвят се получава при еднакво количество от трите цвята. Преобразувайки едно изображение от черно-бяло в цветно, функциите заменят единичната стойност със списък от 3 такива – по една за червения, зеления и синия цвят. За тази цел се използват функциите, предоставени от класа “Image” от библиотеката “PIL”. Тази библиотека е най-използваната за работа с изображения в Python.

След като изображението е преобразувано в цветовия модел RGB, се изваждат пикселите му и се запазват в специални масив от библиотеката “NumPy”. В тази библиотека масивът се разглежда като клас, който има различни атрибути за брой на измеренията, големина на всяко измерение, брой на всички елементи, тип на елементите и други, както и различни функции, които позволяват по-лесно манипулиране на масива и елементите му. Тези допълнителни функционалности позволяват NumPy масивите да намират широка употреба в моделите за машинно обучение. Освен това моделът за лицево верифициране приема само NumPy масиви, затова преобразуването на изображението в такъв е наложително.



Фиг. 3.12: Блокова схема на функциите `“get_pixels_from_file”` и `“get_pixels_from_url”`

Третата помощна функция във файла се използва за отрязване на лицето от изображението и нейното име е `“crop_face”` (Фиг. 3.13). Тя получава два задължителни аргумента и един незадължителен. Двата задължителни аргумента са координатите на правоъгълника, в който се намира лицето, и пикселите на оригиналното изображение. Последният аргумент е `“required_size”`, чиято стойност по подразбиране е `(160,160)`, тоест изображенията на лицата ще бъдат с дължина 160 пиксела и ширина 160 пиксела. Като резултат функцията връща ново изображение, което съдържа само правоъгълника, в който се намира засеченото лице. За изрязване на лицето от изображението се използва операторът `“:”`, който приема начален и краен елемент на масив и отделя този сегмент от оригиналния масив.



Фиг. 3.13: Блокова схема на функцията “crop_face”

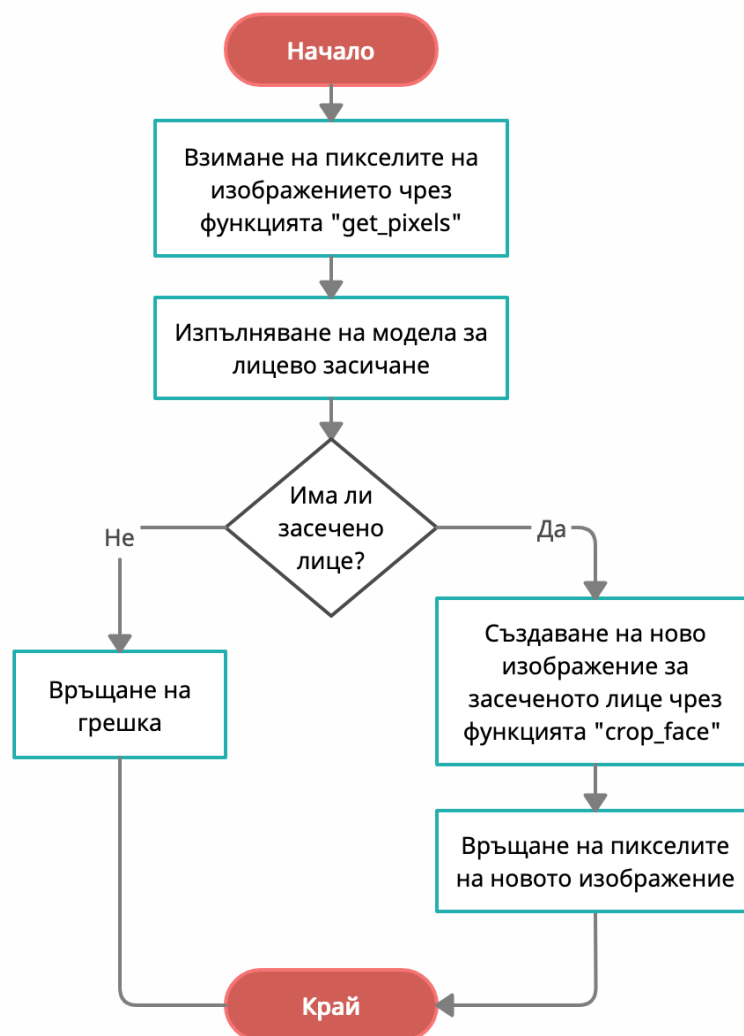
Четвъртата функция във файла се използва само в тренирането на модела за лицево верифициране и нейното име е “extract_single_face”. Тази функция засича само едно лице в изображението и се използва при трениране с цел оптимизиране на бързодействието на процеса на трениране, тъй като обработката на десетки снимки отнема немалко време. Тя приема като аргументи името на файла, където е изображението, и модела за лицево засичане, а връща масив, съдържащ пикселите на засеченото лице. Функцията “extract_single_face” използва “get_pixels”, за да получи пикселите на изображението. Впоследствие се изпълнява моделът за лицево засичане, който връща като резултат координатите на правоъгълника, в който се намира лицето. Засеченото лице се изрязва от оригиналното

изображение посредством функцията “crop_face”. Ако алгоритъмът за лицево засичане върне като резултат празен масив, то това означава, че не е засечено лице в изображението и функцията “extract_single_face” връща грешка. Тъй като тази функция се използва при трениране на модела за лицево верифициране, то е очаквано всяко изображение от колекцията за трениране да съдържа лице, затова при обработването на грешката на потребителската конзола се изписва съобщение, изобразено във *Фиг. 3.14*.

```
$ python3 train.py  
The face detection model didn't detect a face in image_4.jpg in  
Ognian Baruh/
```

Фиг. 3.14: Грешка при липса на засечено лице

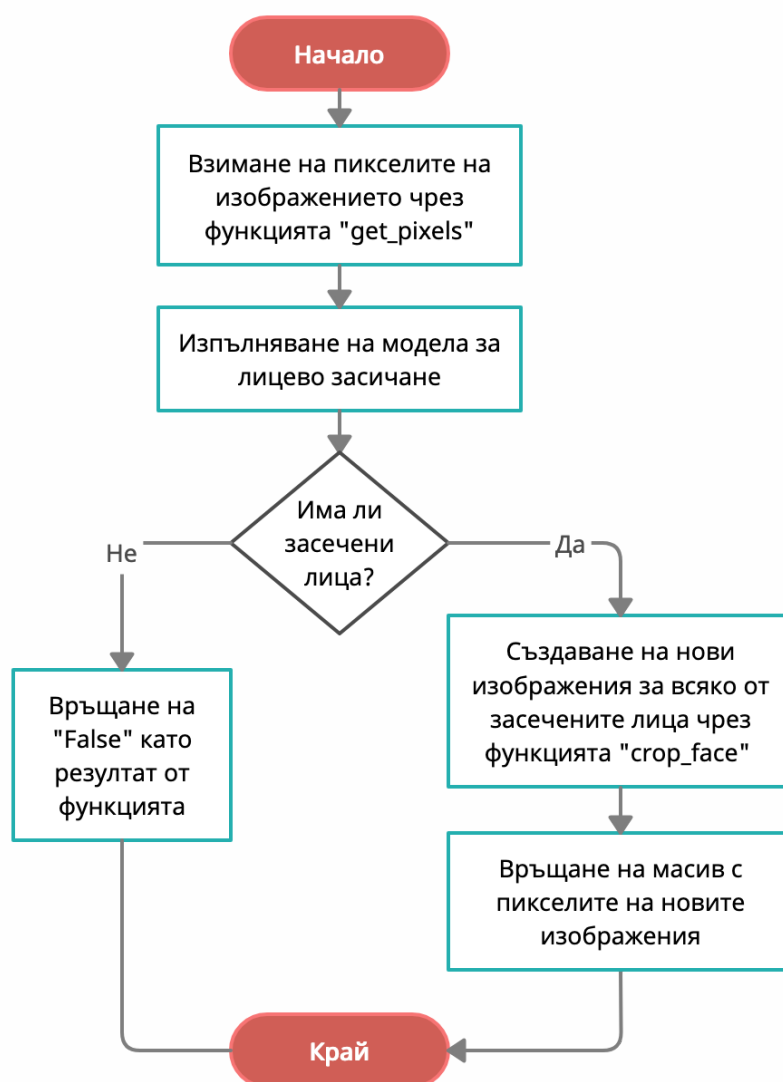
Блоковата схемата на функцията “extract_single_face” е изобразена във *Фиг. 3.15*.



Фиг. 3.15: Блокова схема на функцията “extract_single_face”

Последната функция, с наименование “extract_multiple_faces” (Фиг. 3.16), се използва само при изпълнението на алгоритъма за лицево верифициране, като тя засича множество лица в изображение, а не само едно. Функцията приема името на файла, в което се намира изображението, и модела за лицево засичане и връща масив от нови снимки на отделните лица в оригиналното изображение. Името на файла е незадължителен аргумент и неговата стойност по подразбиране е None. Ако има подадено име на файл, то функцията извлича снимка посредством “get_pixels_from_file”, а ако няма подадено име на файл, се използва функцията “get_pixels_from_url”. Функцията “extract_multiple_faces” се

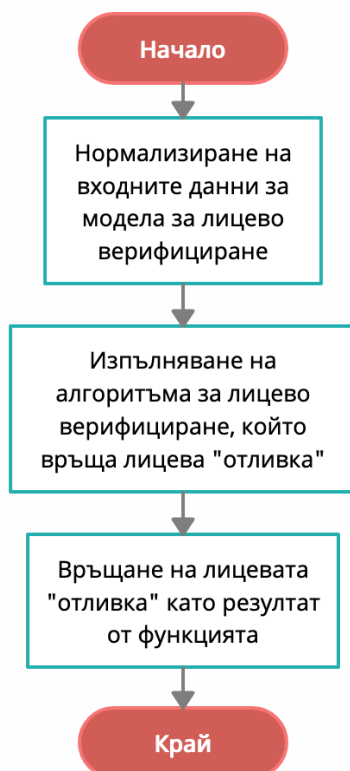
използва за изображения, заснети от камерата, тъй като те могат да съдържат повече от един човек. Ако няма нито едно засечено лице, функцията връща "False", което при обработката връща резултат от изпълнението на модела за лицево верифициране: "No faces detected". Снимките в резултатния масив са от тип NumPy масив, за да могат да бъдат подадени на модела за лицево верифициране, когато трябва да се проверят отделните хора в изображението.



Фиг. 3.16: Блокова схема на функцията "extract_multiple_faces"

3.3.4 Помощни функции за лицево верифициране

При алгоритмите за лицево разпознаване моделът трябва да постави едно лице в едно 128-измерно пространство и да се намери до кое е най-близко, докато при алгоритмите за лицево верифициране моделът трябва да пресметне дали дадено лице е достатъчно близко до някое друго, така че да бъде познато. За да се постави едно лице в 128-измерно пространство се създава т.нар. *face embedding* (“отливка” на лицето). Тази лицева “отливка” представлява 128 стойности, които описват лицето на един човек. Това е начинът, по който машината се “учи” да разпознава или да верифицира лица. След извличане на лице от изображение, това лице се подава на модел за лицево верифициране, който създава тази “отливка” на лицето. Функцията, който създава “отливката” е в отделен файл с име “*get_face_embedding*” (Фиг. 3.17). Функцията приема отрязаната снимка на лице, както и модел за лицево верифициране и връща като резултат “отливката” на лицето. Нейният скелет изглежда по следния начин:

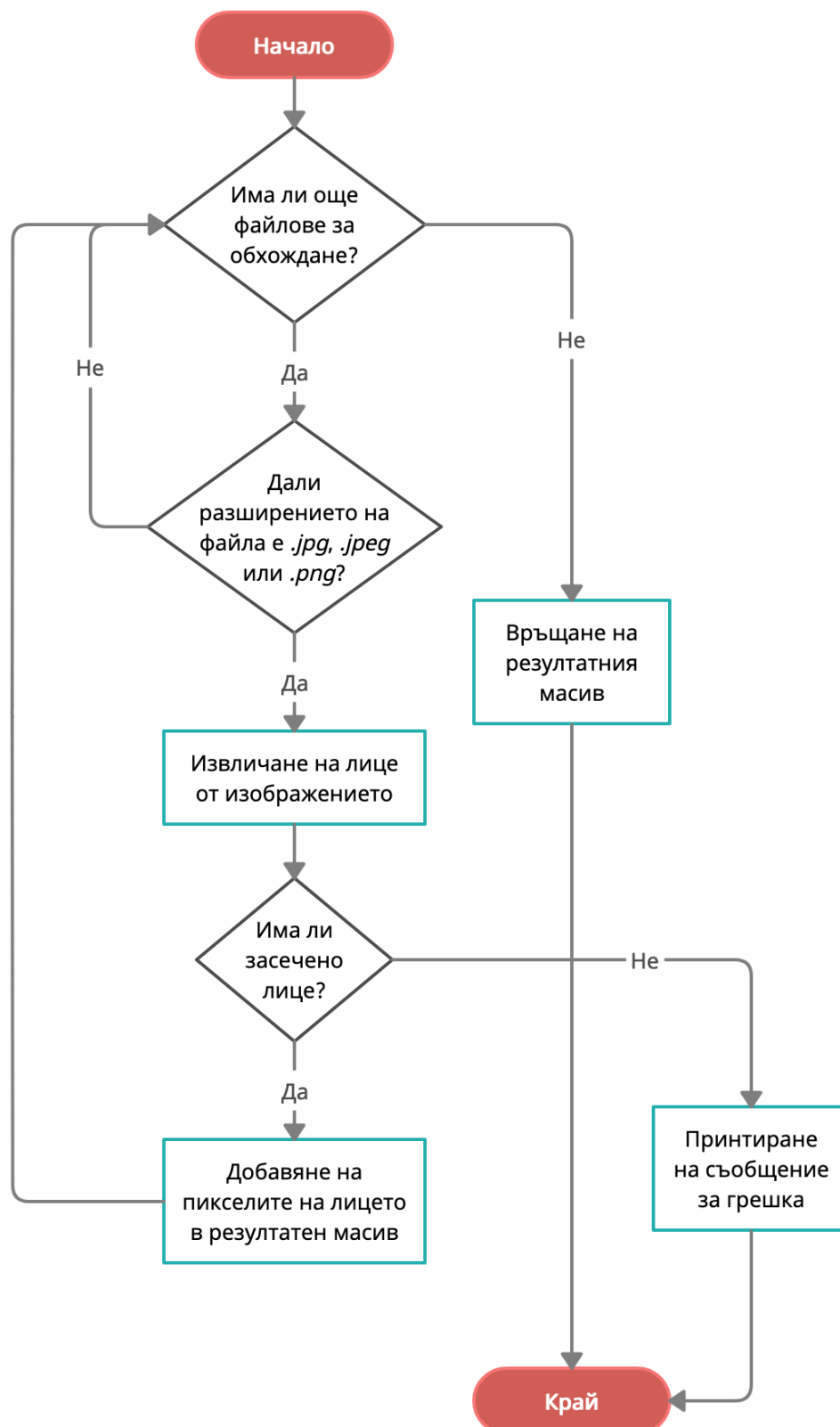


Фиг. 3.17: Блокова схема на функцията “*get_face_embedding*”

3.3.5 Трениране на модела за лицево верифициране

Тренирането на модела за лицево верифициране е най-фундаменталната стъпка за постигането на точни резултати при изпълнение. Важно е да бъдат подадени достатъчно тренировъчни материали, за да се постигне прецизност при изпълнението. Алгоритъмът за лицево верифициране се тества с по 10 изображения на човек, което предразполага за достатъчно голяма прецизност при изпълнението в реална обстановка. Той се “тренира”, като извлича лицевите “отливки” от всяко изображение. Така моделът “знае” къде се намира всяко лице в пространството от 128 измерения и при изпълнение може да се верифицира дали тестваното лице е познато или не. Скриптът за трениране на модела е с име “train.py”, като той съдържа три функции – “load_faces”, “load_dataset” и “save_embeddings”.

Функцията “load_faces” (Фиг. 3.18) извлича лицето от всяко изображение в подадената директория, като всички тези снимки са на един човек, а името на този човек е името на директорията. Функцията приема като аргумент името на директорията и модела за лицево засичане и връща като резултат масив с лицата от всички изображения в директорията. Преди да се извлече лице от изображението, функцията проверява дали файлът е с разширение *.jpg*, *.jpeg* или *.png*, за да е сигурна че се работи с изображение, а не с файл от някакъв друг тип. Ако няма засечено лице в изображението, функцията принтира съобщение за грешка (Фиг. 3.14), тъй като тази функция се изпълнява само в рамките на тренирането на модела за лицево верифициране, а тогава се очаква всяко изображение от колекцията за трениране да съдържа лице. След извличане на всички лица от изображенията в директорията, функцията връща масив, съдържащ тези лица.

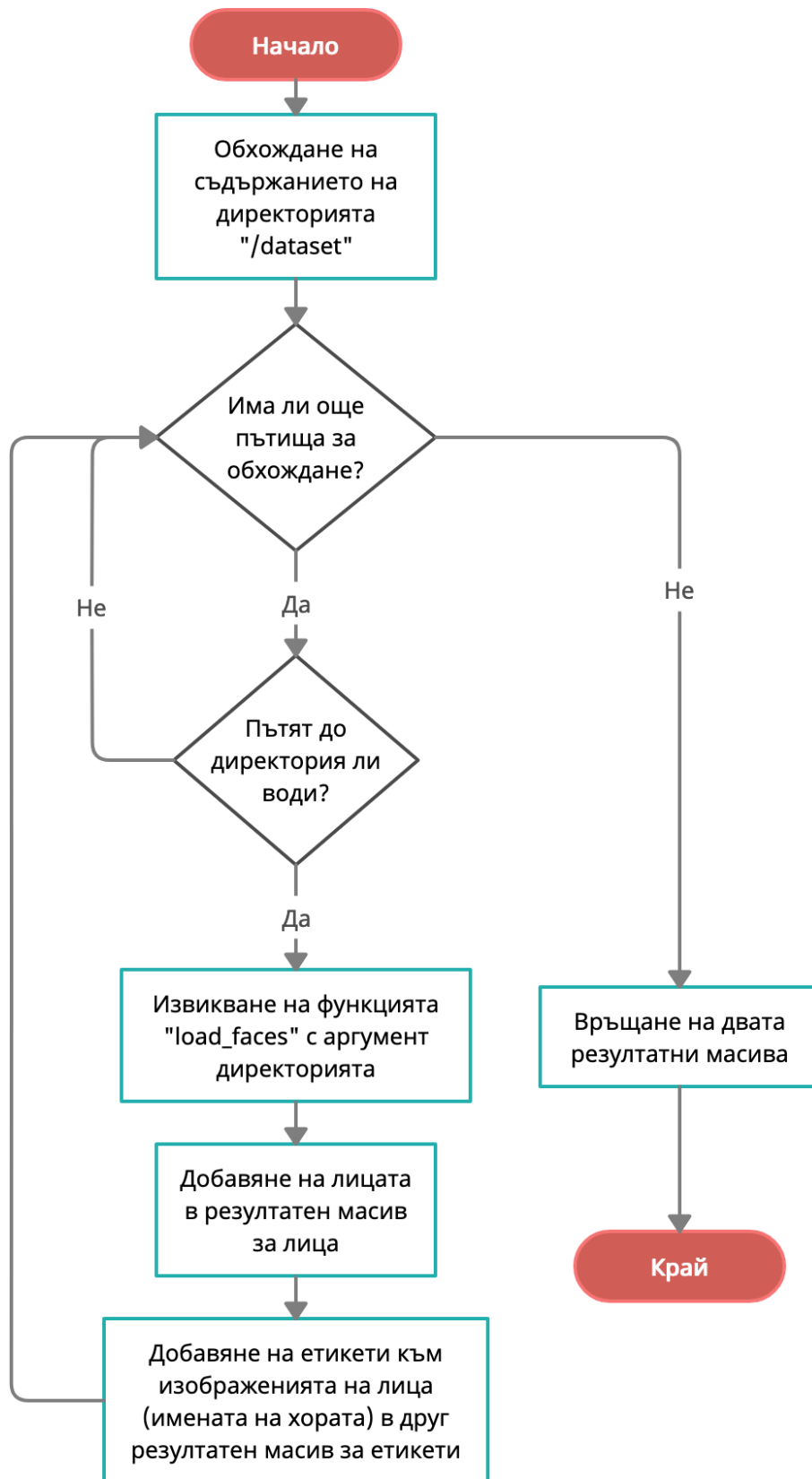


Фиг. 3.18: Блокова схема на функцията “load_faces”

Функцията “load_dataset” (Фиг. 3.20) извиква “load_faces” за всяка поддиректория в папката “dataset”, където се намират всички тренировъчни изображения. Всяка поддиректория представлява изображенията на отделен човек. Аргументите на функцията са името на директорията, в която са запазени всички изображения за трениране, която в контекста на дипломната работа е “dataset”, и моделът за лицево засичане, който се инициализира при стартиране на програмата за трениране и се подава към помощните функции. Функцията връща един асоциативен масив, като ключовете са имената на познатите лица, а стойностите са масиви, съдържащи всички лицеви “отливки” за съответния човек. При обхождане на директорията “dataset” функцията “load_dataset” проверява дали всеки път води до директория, използвайки функцията “isdir” от вградената библиотека “os”. Примерен резултат, който връща функцията, е изобразен във Фиг. 3.19. В него “<face-pixels-from-image-1>” са пикселите на всяко лице, изрязано от изображението, като в един масив се съдържат само снимките на един човек, като името му се използва за ключ.

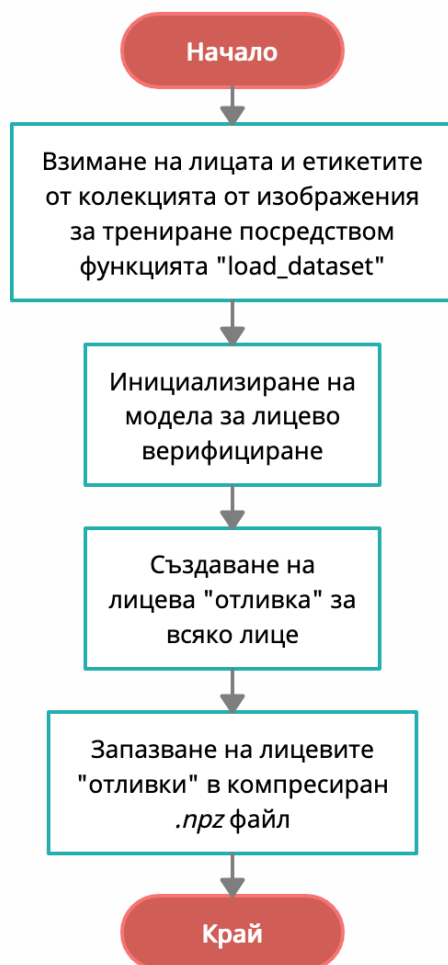
```
{
  Човек-1: [
    <face-pixels-from-image-1>,
    <face-pixels-from-image-2>,
    ...
    <face-pixels-from-image-10>
  ],
  Човек-2: [
    <face-pixels-from-image-1>,
    <face-pixels-from-image-2>,
    ... ,
    <face-pixels-from-image-10>
  ]
}
```

Фиг. 3.19: Примерен резултат от функцията “load_dataset”



Фиг. 3.20: Блокова схема на функцията "load_dataset"

Функцията “`save_embeddings`” (Фиг. 3.21) е отговорна за събирането на всички “отливки” на лицето от колекцията от изображения за трениране и запазването им в специален файл с разширение *.npz*. Тренирането на модела за лицево верифициране е препоръчително да се осъществява на лаптоп или настолен компютър, тъй като използва много ресурси, които един микрокомпютър на теория може да предостави, но процесът ще отнеме прекалено дълго време. Процесът за трениране използва функцията “`load_dataset`”, за да получи всички изображения на лица от колекцията за трениране. Впоследствие се създават лицеви “отливки” за всички лица чрез функцията “`get_face_embedding`”. След изпълнение на функцията, всички лицеви “отливки” се запазват в специален файл с разширение *.npz*. Това представлява архивиран файл, като всичко, запазено в него, се съхранява под формата на асоциативен масив, и при разархивиране на този файл отделните елементи могат да се достъпят с ключовете по подразбиране “`arr_0`”, “`arr_1`”, “`arr_2`” и т.н. Ако са подадени ключове при архивирането, тогава ключовете по подразбиране се заменят с подадените. За архивиране може се използват функциите “`savez`” – за обикновено архивиране, и “`savez_compressed`” – за компресиране преди архивирането, а за разархивиране – “`load`” от библиотеката “`NumPy`”. В контекста на дипломната работа се използва “`savez_compressed`”, като при архивиране се подава ключ “`trainData`” за речника от лицеви “отливки”.



Фиг. 3.21: Блокова схема на процеса на трениране на модела за лицево верифициране

3.3.6 Изпълнение на алгоритъма за лицево верифициране

След като моделът за лицево верифициране е трениран, той може да бъде инициализиран и използван в реална ситуация. В контекста на тази дипломна работа алгоритъмът за лицево верифициране се изпълнява върху изображения, заснети от камера, свързана в системата. Функционалностите за лицевото верифициране са разделени в две функции в два файла – “recognize.py” и “candidate_face_check.py”.

В първия файл се намира функцията “recognize_faces”, която приема като аргументи изображението, заснето от камерата и тренирания модел. Тя извлича всички лица от изображението и им създава лицеви “отливки”, които да се сравнят с познатите лицеви “отливки” от колекцията от изображения за трениране. Ако няма засечено лице в изображението, функцията връща “No faces detected” без въобще да се изпълнява алгоритъмът за лицево верифициране.

Във втория файл е разположена функцията “check_candidate_faces”, която се извиква от “recognize_faces” и сравнява лицата от изображението от камерата с познатите лица, заложили в колекцията. Функцията приема 3 аргумента – речника с всички лицеви “отливки” на тренираните лица, масив с лицевите “отливки” на лицата в изображението от камерата и праг (число от 0 до 1), под който две лица ще бъдат определени като един човек. Всяко лице кандидат се сравнява с познатите лица, като се изчислява косинусът на ъгъла между двата вектора – лицевата “отливка” на лицето кандидат и лицевата “отливка” на познатото лице. За намирането на косинус на два вектора се използва функцията “cosine” (Формула. 3.1) от библиотеката “SciPy”. “SciPy” е библиотека с отворен код за по-сложни математически функции.

$$1 - \frac{u \cdot v}{||u||_2 ||v||_2}$$

Формула 3.1: Формула на функцията “cosine” от библиотеката “SciPy”

Във *Формула 3.1* “*u*” и “*v*” са двата вектора, чиито косинус се търси. В числителя се намира скаларното произведение на двата вектора, а в знаменателя се намира евклидовото разстояние между двата вектора. Тъй като косинус от 0° е 1, то косинусът на двата вектора се изважда от 1, за да може близките вектори да имат стойност близка до 0, а далечните – близка до 1. Така две лица на един и същ човек ще дадат резултат под 0,5 след изпълнение на функцията “cosine”, а две лица на различни хора ще дадат резултат над 0,5. След изследване на резултатите при тестване с различен брой хора и различен брой изображения се достигна до най-точен праг от 0,3 – ако резултатът от функцията “cosine” е по-малък от 0,3, то двете лица са на един и същ човек.

При изпълнение на функцията “check_candidate_faces” се обхождат лицевите “отливки” на кандидатите. Всеки кандидат се сравнява с всеки познат човек, като се изпълнява вложен цикъл, обхождащ всички хора в речника с лицеви “отливки”. След това се изпълнява функцията “cosine” за “отливката” кандидат и всяка от десетте отливки на съответния познат човек и се намира средно аритметично на десетте резултата на функцията “cosine”. Двойният вложен цикъл е изобразен във *Фиг 3.22*. В имплементацията не се използват временни променливи, а се използва силата на скриптовия език Python да се пишат вложени цикли на един ред.

```
инициализиране на масив scores
за всеки елемент candidate в candidate_embeddings
    инициализиране на временен речник temp_dict
    за всяка двойка name, known_embeddings в trainData
        за всеки елемент known в known_embeddings
            извикване на cosine с candidate, known
            извикване на mean с всички резултати от функцията cosine
            запазване на резултата от mean в временна променлива t_mean
            добавяне на name, t_mean в temp_dict
        добавяне на temp_dict в scores
```

Фиг. 3.22: Псевдокод за създаване на асоциативен масив с резултати

Резултатите, съхранявани в променливата “scores” (Фиг. 3.23), се използват, за да се създаде резултатен масив “names”, съдържащ имената на познатите лица, които съвпадат с кандидатите.

```
[
  {
    'Познат-човек-1': 0.20623254179954528,
    'Познат-човек-2': 0.40129605531692521
  },    // Кандидат 1
  {
    'Познат-човек-1': 0.51230716642703621,
    'Познат-човек-2': 0.26734365280065493
  }    // Кандидат 2
]
```

Фиг. 3.23: Примерна стойност на променливата “scores”

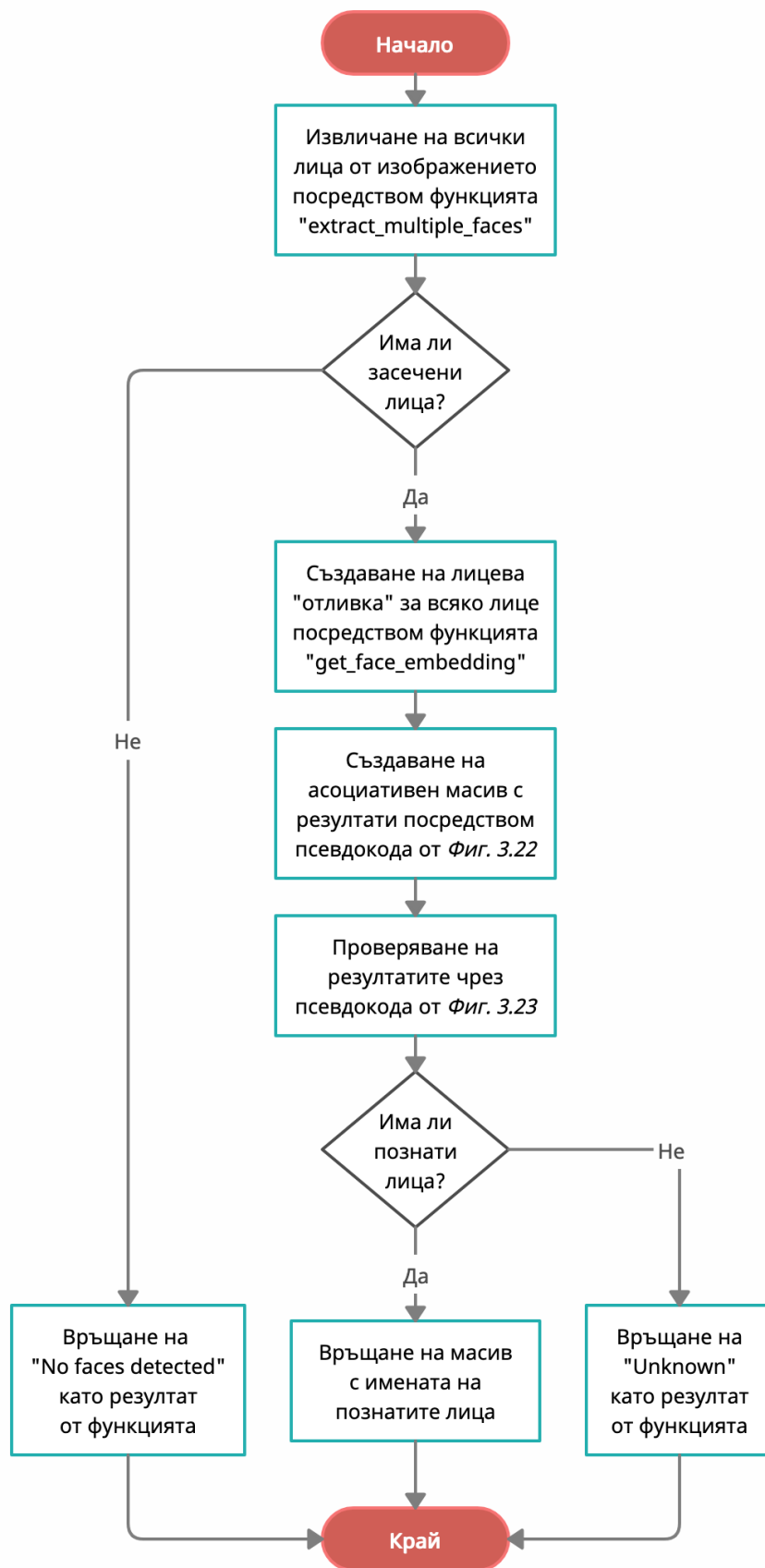
Ако при някои от кандидатите резултатът за някое от познатите лица е под 0,3, то тези две лица съвпадат и името на човека се добавя в резултатния масив. Псевдокодът, проверяващ резултатите е изобразен във Фиг. 3.24.

```
инициализиране на масив names
за всеки елемент score_dict в scores
  за всяка двойка name, score в score_dict
    ако score < threshold
      добавяне на name в names
```

Фиг. 3.24: Псевдокод на проверяване на резултатите от функцията “cosine”

Ако няма нито едно познато лице в изображението, то тогава функцията връща резултат “Unknown”. Ако има познати лица, функцията връща резултатния масив, съдържащ техните имена.

Блоковата схема на процесът за лицево верифициране, имплементиран във функциите “recognize_faces” и “check_candidate_faces” е изобразена във Фиг. 3.25.



Фиг. 3.25 Блокова схема на процеса на лицево верифициране

3.4 Свързване на устройствата към системата

3.4.1 Създаване на “Gateway Device”

“Gateway Device” е основният компонент, който изгражда системата от свързани устройства. За такива устройства могат да се използват компютри или микрокомпютри, като за реализиране на дипломната работа се използва Raspberry Pi. Този микрокомпютър играе ролята на връзка между физическите устройства и техните дигитални репрезентации.

Първата стъпка за създаването на “Gateway Device” е да се създаде профил в облачната услуга Bosch IoT Suite и да се създаде Subscription, използвайки безплатния предоставен план. Безплатната версия на облачната услуга предлага всички необходими функционалности за разработване на личен проект. Subscription може да се създаде на следния линк: <https://accounts.bosch-iot-suite.com/subscriptions/>.

При първо посещение на сайта на Bosch IoT Suite, потребителят се пренасочва към страница за логин, където може да влезе в своя акаунт или да създаде нов такъв. След създаване на акаунт може да се създаде Subscription с безплатен план, като се използва предварително създаденият пакет “Bosch IoT Suite for Device Management”.

След създаване на Subscription върху микрокомпютъра Raspberry Pi се инсталират двете необходими услуги за реализация на дипломната работа – Bosch IoT Edge Services и Bosch IoT Edge Agent, които могат да се намерят в “Edge Downloads”. Услугите се инсталират като архивирани файлове, които трябва да се разархивират и да се копират върху Raspberry Pi. За целта се използва командата “scp”, която предоставя лесно и сигурно копиране на файлове между две машини в една мрежа. Процесът на инсталиране на услугите върху микрокомпютъра е описан в документацията на Bosch IoT Suite – [Наръчник за инсталиране на Bosch IoT Edge Services](#) и [Наръчник за инсталиране на Bosch IoT Edge Agent](#). В двата наръчника има както стъпките, необходими за инсталиране, така и стъпките за стартиране, изключване и деинсталиране на услугите.

Последната стъпка при създаването на “Gateway Device” е да се създаде т.нар “provisioning”, който създава дигитална репрезентация на мрежовия шлюз. Това позволява да се създават автоматично дигитални близнаци на всички устройства, които се свързват към системата. Създаването на “provisioning” е описано подробно в [Наръчник за създаване на "provisioning" за "Gateway Device"](#). След инсталирането и стартирането на Bosch IoT Edge Services и Bosch IoT Edge Agent, както и създаването на “Gateway Device”, може да се пристъпи към добавянето на умните устройства.

3.4.2 Добавяне на камера към системата от свързани устройства

Bosch IoT Edge Services предоставя множество пакети за свързване на устройства, поддържащи различни протоколи за комуникация. Камерата, използвана в системата, работи с ONVIF протокол, затова трябва да се инсталират и стартират пакетите за ONVIF поддръжка. Процесът за тяхното инсталиране е описан в [Наръчник за инсталиране на пакети за ONVIF поддръжка](#). Освен това трябва да се инсталира и уеб конзола, която позволява интерактивно управление на инсталираните пакети и осигурява по-лесно свързване на устройствата към системата, като командите за инсталиране са описани в [Наръчник за инсталиране на уеб конзола](#).

Уеб конзола дава достъп до т.нар. “functional items” (функционални единици), които представят всички функционалности на инсталираните пакети, както и на свързаните устройства. След свързването на устройствата към системата те ще бъдат видими в уеб конзолата и потребителят ще може да изпълнява различни операции върху тях.

За свързването на камерата се използва функционалната единица “ONVIF Discovery”, която открива всички ONVIF камери и техните IP адреси. След това камерата се регистрира и така се създава връзката между нея и микрокомпютъра, върху който е инсталирана услугата Bosch IoT Edge Services. След регистрация, всички функционални единици на камерата ще бъдат достъпни за потребителя в уеб конзолата.

3.4.3 Добавяне на ключалка към системата от свързани устройства

Ключалката, използвана за реализация на дипломната работа, използва Z-Wave за комуникация, но Raspberry Pi 3, няма вграден Z-Wave контролер, затова трябва да се използва отделен такъв, който се вкарва в USB порта на Raspberry Pi. Освен това трябва да се инсталират и пакети за Z-Wave поддръжка, така че микрокомпютърът да може да изпраща съобщения към ключалката като радиовълни. Процесът за инсталиране на тези пакети е показан в [Наръчник за инсталиране на пакети за Z-Wave поддръжка](#).

След свързване на ключалката към системата функционалностите на ключалката ще станат видими и достъпни за потребителя под формата на функционални единици, които му позволяват да изпълнява всички възможни операции върху ключалката.

3.5 Програмна реализация на управляващия код за устройствата

3.5.1 Инициализиране на MQTT клиент

При стартиране на Bosch IoT Edge Agent, който осигурява връзка между създадения “Gateway Device” и облачната услуга Bosch IoT Suite, се инсталира и стартира локален MQTT брокер. Той е отговорен за разпределянето на съобщенията локално върху мрежовия шлюз. Инициализирането на MQTT клиент се осъществява в основния скрипт “main.py”, намиращ се в директорията “iot_system”. За създаването на MQTT клиент се използва външната библиотека “paho”. Създава се клиент с име “iot_system”, като се използва класът “Client” от “paho”.

Преди да се инициализира връзката между клиента и брокера се задават две “callback” функции – “on_connect” и “on_message”. Първата “callback” функция се извиква при създаване на връзката между клиента и брокера. В нея са поместени “subscribe” функции, чрез които клиентът се абонира за един или за няколко канала. Каналите, към които се абонира приложението, са въведени като константи в скрипта. Моделът, по който се изграждат техните имена, е описан подробно в документацията на [Eclipse Hono](#). По

този начин MQTT клиентът ще получава съобщенията, за които се е абонира, като при получаването на всяко съобщение се изпълнява функцията “on_message”. Каналите за получаване на съобщения в имплементацията на тази дипломна работа са 3 – съобщения за отключване на ключалката, съобщения за заключване на ключалката и събитие за засичане на обект. В зависимост от канала се изпълнява различна функционалност. Двете “callback” функции се подават като атрибути на инстанцията на класа “Client”.

Свързването на клиента към брокера се изпълнява посредством функцията “connect” на класа “Client”, като тя приема като аргументи IP адреса на брокера (в случая – IP адреса на Raspberry Pi), порт – 1883 и максимално време, в което при липса на съобщения връзката се разпада – 60 секунди. Според спецификацията на MQTT протокола, клиентът изпраща през малки интервали PING заявки към брокера, за да е сигурен, че връзката все още съществува. Ако няма нито една заявка, изпратена в рамките на 60 секунди, то двете страни приемат, че връзката е загубена и спират да изпращат съобщения. След установяване на връзката се изпълнява функцията “loop_forever” върху инстанцията на класа “Client”, която осигурява безкрайното изпълнение на програмата.

3.5.2 Програмна реализация на управляващия код за камерата

Един от каналите, за който се абонира приложението, е за събитието за засичане на обект. При това събитие функционалната единица “Detector” изпраща съобщение, за да информира всички абонати, че има засечен обект. При получаване на такова съобщение скриптът “main.py” изпълнява функцията “recognize_faces” от файла “recognize.py”, която засича всички лица в изображението, заснето от камерата. Ако функцията е върнала като резултат нещо различно от “Unknown” или “No faces detected”, то тогава има засечено познато лице и се отключва ключалката за 5 секунди. Към управляващия код на камерата е добавен скриптът

“screenshot_control.py” в директорията “camera_controller”. Той е разделен на 4 функции. Първата функция “open_image_from_url” отваря изображението от камерата, тъй като тя го записва на специален линк. Втората функция, с наименование “send_image_to_cloud_storage”, изпраща заснетото изображение до Firebase Cloud Storage, като използва първата функция, за да отвори снимката. Третата функция “send_time_of_image_to_firestore” изпраща данните към изображението към Firebase Cloud Firestore, а последната функция “send_screenshot” изпълнява целия процес за изпращане на изображение към мобилното приложение.

3.5.3 Програмна реализация на управляващия код за ключалката

Другите два канала, за които се абонира приложението, са командите за отключване и заключване на вратата. В зависимост от командата се изпълнява различна функция, като двете функции са имплементирани във файла “lock_control.py” в директорията “door_lock_controller”. Функцията за отключване на ключалката е с наименование “unlock_door”, а за заключване – “lock_door”. Двете функции изпращат заявка към функционалната единица “DoorLock” посредством предоставеното от Bosch IoT Edge Services REST API. Тези заявки стигат до ключалката чрез Bosch IoT Edge Services, изпълнявайки желаната от потребителя операция. В документацията на функционалната единица са описани възможните команди към ключалката – стойност 0 я отключва, а стойност 255 я заключва.

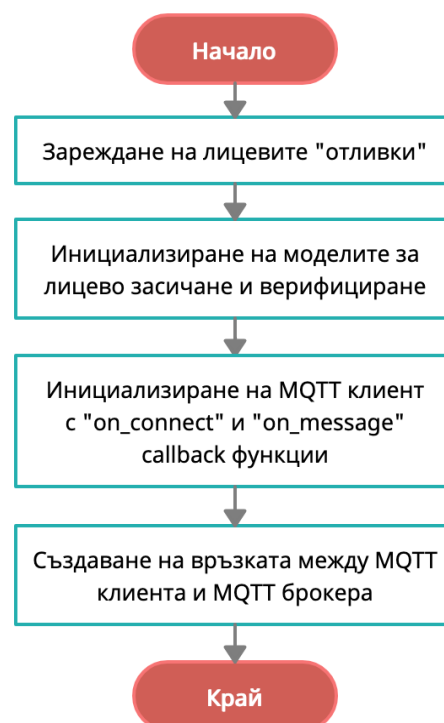
3.5.4 Програмна реализация на известията

При засичане на движение от камерата и преминаване на изображението през алгоритъма за лицево верифициране, системата изпраща известие към мобилното приложение, съдържащо заснетото изображение, датата и резултатът от модела за лицево верифициране. Изпращането на известието е имплементирано в файлът “notification_control.py” в директорията

“notification_controller”. Скриптът е разделен на 4 отделни функции. Първата функция е отговорна за изпращането на изображението към Firebase Cloud Storage и е с наименование “send_image_to_cloud_storage”. Втората функция (“send_update_to_firestore”) изпраща датата и резултата от модела за лицево верифициране в Firebase Cloud Firestore. Третата функция (“send_cloud_notification”) изпраща HTTP заявка към Firebase Cloud Messaging, съдържащ резултата от изпълнението на алгоритъма за лицево верифициране, като облачната услуга на Firebase е отговорна за разпространяването на известието до всички мобилни устройства. Последната функция (“send_notification”) извиква предходните три, за да осъществи процеса за изпращане на известие.

3.6 Стартиране на системата от свързани устройства

За стартиране на системата от свързани устройства се изпълнява скриптът “main.py”, като процесът е изобразен във *Фиг. 3.26*.



Фиг. 3.26: Блокова схема на процеса на стартиране на системата от устройства

3.7 Файлова структура на мобилното приложение

Файловата структура на мобилното приложение, която се намира в поддиректорията “lib” на директорията “mobile” е изобразена във *Фиг. 3.27*. То е разделено на 3 основни компонента – модели (“models”), екрани (“screens”) и услуги (“services”). Във файла “colors.dart” са записани константи на всички цветове, използвани за изграждане на потребителския интерфейс на приложението. Файлът “main.dart” е началният файл, който се изпълнява при стартиране на мобилното приложение.

```
lib/  
├── models/  
├── screens/  
├── services/  
├── colors.dart  
└── main.dart
```

Фиг. 3.27: Файлова структура на мобилното приложение

Папката “models” съдържа само един файл – “user.dart”, който съдържа клас за потребител, който пази неговият уникален номер. Този клас се използва при автентикацията, за да може да се проверява дали потребителят е влязъл в своя профил.

Директорията “screens” съдържа поддиректории за всеки екран в приложението, който е достъпен за потребителя. Всяка поддиректория на екран е разделена на две части – папка “components”, съдържаща отделните компоненти от страницата и основен файл с името на поддиректорията, който комбинира всички компоненти. Файлът “wrapper.dart” се използва, за да се контролира коя страница се показва в зависимост от това дали потребителят е влязъл в своя профил. Файловата структура на директорията “screens” е изобразена във *Фиг. 3.28*.

```
screens/
├── authenticate/
├── cameras/
├── home/
├── locks/
├── notifications/
├── register/
├── signin/
└── wrapper.dart
```

Фиг. 3.28: Файлова структура на директорията “screens”

Директорията “services” съдържа различни файлове с помощни функции за отделните страници в приложението. Нейната структура е изобразена във *Фиг. 3.29*.

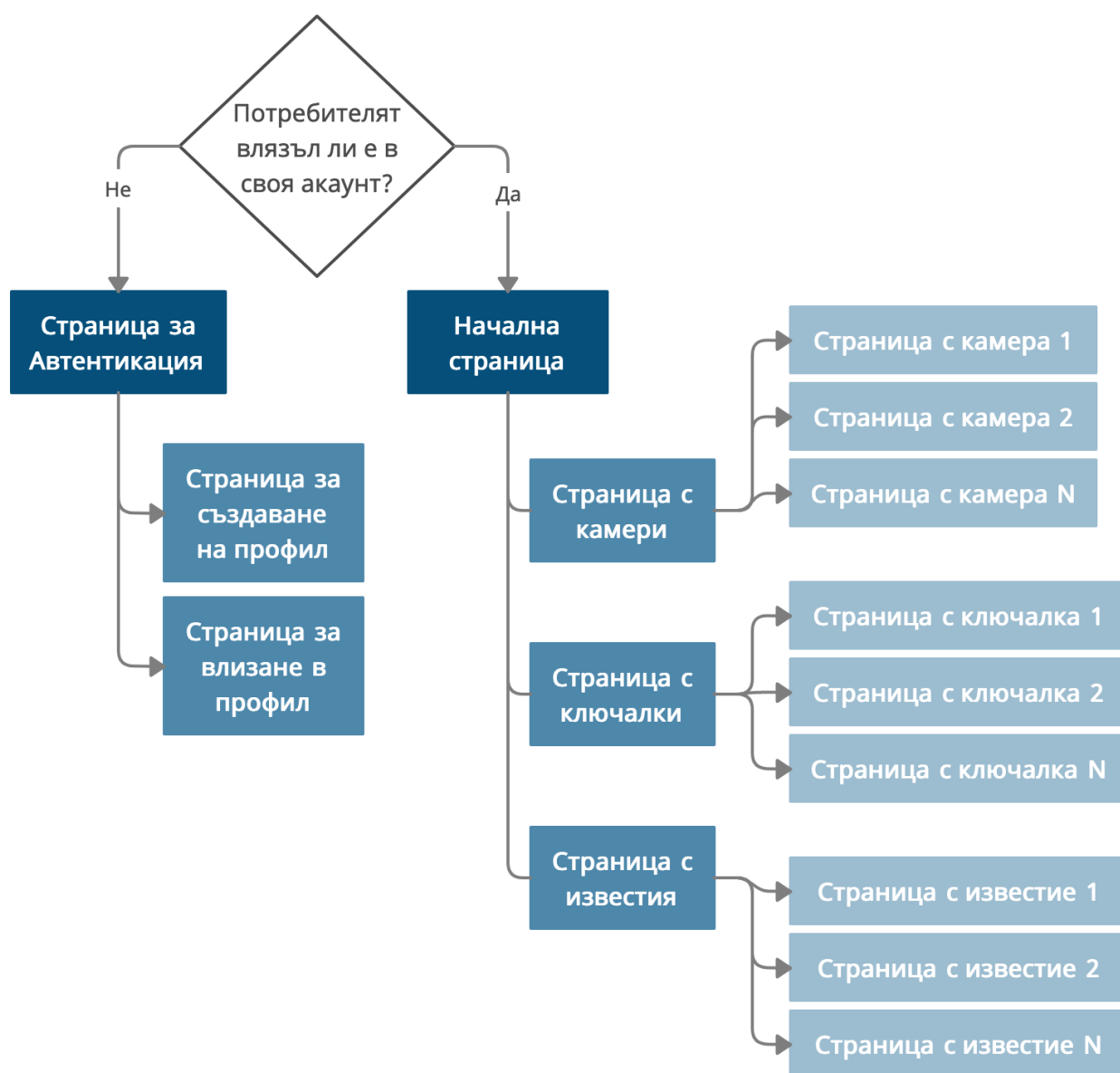
```
services/
├── auth.dart
├── camera.dart
├── database.dart
├── lock.dart
├── storage.dart
└── token.dart
```

Фиг. 3.29: Файлова структура на директорията “services”

3.8 Програмна реализация на мобилното приложение

3.8.1 Структура на страниците

Структурата на страниците в мобилното изображение е изобразена във Фиг. 3.28.



Фиг. 3.28: Структура на страниците в мобилното приложение

3.8.2 Управление на профил

Страниците, отговорни за създаването и влизането в профил са разположени в поддиректориите “authenticate”, “register” и “signin”. на директорията “screens”. Функционалният код, управляващ процесът на управление на профилите е поместен във файла “auth.dart” в папката “services”. Той се състои от класът “AuthService”, който съдържа 3 функции – “signIn”, “register” и “signOut”. И трите функции използват Firebase Authentication, за да автентикират потребителя, като потребителските имена и пароли се пазят в специална база от данни, предоставена от Firebase Authentication. При регистрация и влизане в профил се запазва уникалният идентификатор на потребителя, за да могат да се достъпят неговите камери и ключалки. Той се пази в т.нар. Flutter Secure Storage, което представлява нещо подобно на сесия в уеб приложенията, където може да се съхранява и достъпва различна информация, свързана с потребителя. При инициализация на класът “AuthService” се създава stream, през който се изпраща съобщение, когато потребителят влезе или излезе от своя акаунт, така че да се покаже правилната страница. Файлът “wrapper.dart” показва или страницата за автентикация, или началната страница в зависимост от това дали потребителят е влязъл в своя профил.

3.8.3 Създаване на токен за Bosch IoT Suite

Облачната услуга Bosch IoT Suite съхранява дигиталните близнаци на милиони потребителски устройства, затова трябва да се създаде начин да се контролира достъпът до тях. При създаването на дигитален близнак се създава тайна парола, достъпна само за потребителя. За да се достъпват устройствата чрез HTTP заявки трябва да се генерира токен за достъп. Този токен впоследствие се използва, за да се изпращат съобщения към дигиталните близнаци на устройствата. Функционалностите, свързани със създаването и съхраняването на токена, са имплементирани във файла “token.dart”. Той съдържа 4 функции за управление на токените от Bosch

IoT Suite. “_generateToken” изпраща заявка към Bosch IoT Suite за получаване на токен, който впоследствие го запазва в Flutter Secure Storage. Следващата две функции са отговорни за декодирането на токена, за да се извлече времето на валидност, тъй като един токен от Bosch IoT Suite е валиден 60 минути. Последната функция е “getToken”, която връща токен, който да се използва при изпращане на съобщения към дигиталните близнаци. Тя проверява дали има валиден токен и ако няма, създава такъв посредством функцията “_generateToken”.

3.8.4 Управление на базата от данни

Управлението на заявките към Firebase Cloud Firestore се осъществява във файла “database.dart”. Той съдържа 3 функции, които взимат всички документи в колекциите “notifications”, “cameras” и “locks”. Документите за известия се филтрират по уникалния идентификатор на потребителя и се подреждат по дата в низходящ ред. Документите за камери и ключалки се филтрират само по уникалния идентификатор на потребителя. При извличане на документите се създава “stream”, който следи за промени в базата от данни, които да ги изпрати към приложението.

3.8.5 Изграждане на начална страница

Началната страница се състои от тяло и навигационна лента в дъното на страницата. Навигационната лента показва една от три страници – списък с камерите на потребителя, списък с ключалките на потребителя и списък с всички известия, като по подразбиране се показва страницата с камерите. Файлът, в който е описан потребителския интерфейс на началната страница се намира в директорията “home” и е с наименование “home.dart”. За реализацията на страницата се използва “StatefulWidget”, който позволява да се създава състояние на страницата и да се обнови динамично. В горния десен ъгъл на страницата е добавен бутон, чрез който потребителят може да излезе от своя акаунт.

3.8.6 Управление на камери

Една от подстраниците на началната страница е списъкът с всички камера, които притежава потребителя. Те се взимат от базата от данни и се показват на екрана като бутони. Списъкът с камери се генерира от файла `“cameras_list.dart”`. Той представлява `“StatefulWidget”`, в който списъкът от камери се създава посредством `“FutureBuilder”`. Този Dart компонент се инициализира на екрана и чака за промени в списъка с камери, за да ги отрази в потребителския интерфейс. По този начин се осигурява динамичност на мобилното приложение. Всеки бутон се показва с името на камерата, така че потребителят да може да разграничава своите устройства. При натискане върху някои от бутоните на камера се отваря нова страница, която показва последното взето изображение от камерата, както и бутон `“Get Image”`. За вземане на изображение се използва функцията `“sendGetImageMessage”`, дефинирана във файла `“camera.dart”` на директорията `“services”`. Тя изпраща съобщение към дигиталната репрезентация на камерата в Bosch IoT Suite с цел вземане на снимка от камерата. Изображението се изпраща в Cloud Storage, а уникалният му идентификатор се изпраща в Cloud Firestore. Мобилното приложение слуша за промени в Cloud Firestore, за да може да извлече изображението от базата от данни в реално време и да обнови потребителския интерфейс. Файлът, в който е описана страницата на всяка камера е с наименование `“camera.dart”` и се намира в поддиректорията `“cameras”` на директорията `“screens”`.

3.8.7 Управление на ключалки

Следващата подстраница е списъкът с ключалките на потребителя. Функционалностите на тази подстраница са аналогични на тези от подстраницата за камерите. Файловете за списъка от камери и всяка камера са поставени в поддиректорията `“locks”` на директорията `“screens”` и техните наименования са съответно `“locks_list.dart”` и `“lock.dart”`.

Страницата на всяка ключалка съдържа два бутона – “Lock” и “Unlock”. При натискането върху тях се изпълняват съответно функциите “sendLockMessage” и “sendUnlockMessage”, дефинирани във файла “lock.dart” в директорията “services”. Те изпращат съобщения към Bosch IoT Suite, които се обработват от микроконтролерът Raspberry Pi в рамките на системата и отключва или заключва ключалката.

3.8.8 Управление на известия

Последната подстраница на началната страница съдържа списък от всички получени известия, като тя е имплементирана във файла “notifications_list.dart” в поддиректорията “notifications” в директорията “screens”. Списъкът от известия е имплементиран по подобен начин, както списъците от камери и ключалки, като се използва функцията “getNotifications” от файла “database.dart”, за да се извлече stream от известие, които да се покажат на екрана. Бутоните на всяко известие съдържа заглавие – имената на засечените лица, “Unknown”, ако засеченото лице не е познато, или “No faces detected”, ако засеченият обект не е човек. Освен това бутоните съдържат и подзаглавие, което е часът, в който е засечен обектът. При натискане върху бутонът на някое от известията се отваря нова страница, която показва имената на засечените хора, часът, както и изображение, което е било използвано при изпълнението на алгоритъма за лицево верифициране. Файлът, който описва страницата на всяко известие, е с наименование “notification.dart”, като той се намира в поддиректорията “notifications” на папката “screens”.

ГЛАВА IV. РЪКОВОДСТВО ЗА ПОТРЕБИТЕЛЯ

4.1 Стартиране на системата от свързани устройства

4.1.1 Подготвяне на Raspberry Pi

За “Gateway Device” е задължително да се използва Raspberry Pi 3 или Raspberry Pi 4, тъй като те имат достатъчен мощен процесор и USB порт за Z-Wave контролера. Първо трябва да се достъпи конзолата на микрокомпютъра, като това може да стане с командата “ssh” или чрез програми като PuTTY или MobaXterm.

```
$ ssh pi@<ip-address>
```

Ако микрокомпютърът е свързан с кабел към вашата машина, то тогава IP адресът му ще бъде “raspberrypi.local”. Ако е свързан към интернет, тогава ще трябва да разберете IP адресът му от списъка с устройства, свързани към вашия маршрутизатор. След като имате достъп до конзолата на вашето Raspberry Pi, създайте директория, където ще се инсталира проекта.

4.1.2 Инсталиране на функционалния код от Github

- Инсталира се върху машината на потребителя функционалния код за системата от [публичното хранилище](#) в Github.
- Разархивира се проекта и се навигира до папката “src”.

```
$ cd Ognian-Baruh-ELSYS-Thesis-2021/src
```

- Изпълнява се bash скрипта “install.sh”, който ще инсталира всички необходими външни библиотеки във виртуална среда. Стартира се виртуалната среда с командата “pipenv shell”.

```
$ ./install.sh  
$ pipenv shell
```

4.1.3 Създаване на изображения за трениране

- Стартира се програмата за създаване на изображения, като се подава първо име и фамилия. Конзолата трябва да има достъп до камера на машината на потребителя, за да може да се създадат изображения. След като се инициализира прозорец, показващ камерата, се създават 10 изображения, натискайки бутона “SPACE”, като леко се върти и премества лицето на потребителя. Изображенията се пазят в директорията “dataset”, в поддиректорията с името му. Потребителят може ръчно да добавя или премахва изображения или да изтрие цялата поддиректория с негови снимки.

```
$ cd iot_system/face_verification  
$ python3 dataset_loader.py -f Ognian -l Baruh
```

4.1.4 Трениране на модела за лицево верифициране

- Програмата за трениране на алгоритъма се намира в директорията “face_verification”.
- Изпълнява се програмата “train.py”. Тя ще създаде архивиран файл с наименование “embedding.npz” в директорията “models”, който съдържа всички лицеви “отливки”.

```
$ python3 train.py
```

- След като моделът е трениран, функционалният код трябва да се копира върху Raspberry Pi. Трябва да се копира папката “src” с цялото нейно съдържание, като трябва да има установена връзка с него предварително. Ако няма, се следват стъпките в т. 4.1.1, за да се създаде такава.

```
$ cd ../../..  
$ scp -r src pi@<ip-address>:/home/pi/<path-to-project>
```

- След като функционалният код е успешно копиран върху Raspberry Pi, може да се пристъпи към стартирането ѝ. Последната стъпка върху машината е да се изключи виртуалната среда.

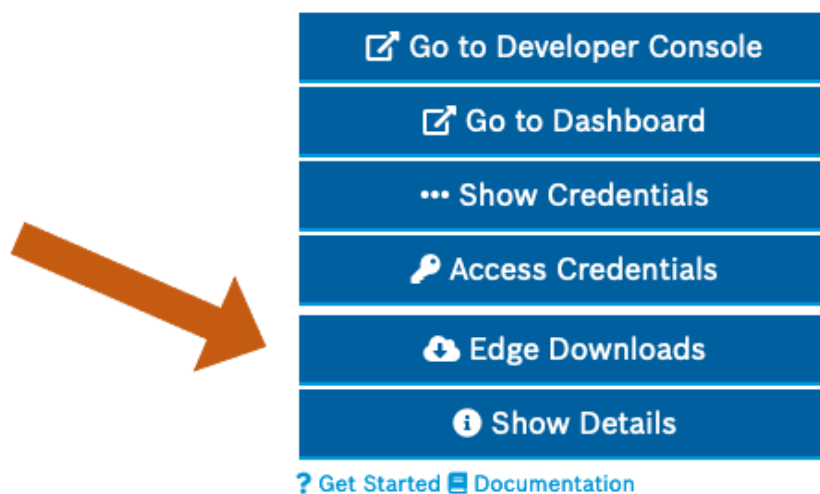
```
$ exit
```

4.1.5 Създаване на Subscription в Bosch IoT Suite

- Навигира се до сайта на [Bosch IoT Suite](#). Ако потребителят не е влязъл в профила си, ще бъде пренасочен към друга страница, където ще може да се регистрира или да влезе в своя акаунт.
- Създава се нов Subscription, използвайки безплатния план на предварително създаденият пакет Bosch IoT Suite for Device Management.
- Задава се име на новия Subscription и се натиска бутона “Subscribe”.

4.1.6 Инсталиране на Bosch IoT Edge Agent

- Натиска се бутона “Edge Downloads” в новосъздадения Subscription (Фиг. 4.1).



Фиг. 4.1: “Edge Downloads” на Subscription

- Изтегля се “Edge Agent for Raspberry Pi”.
- Разархивира се инсталираната услуга.
- Копира се Bosch IoT Edge Agent до микрокомпютъра, ако е сигурно, че машината има връзка с него. Ако няма връзка, се създава такава, използвайки указанията в т. 4.1.1.

```
$ scp -r com.bosch.iot.edge.agent.assemblies.dist-pack_
linux_arm_raspbian pi@<ip-address>:/home/pi/<path-to-
project-directory>
```

- Следва се [наръчника за инсталиране на Bosch IoT Edge Agent](#), за да се инсталира услугата върху потребителското Raspberry Pi.

4.1.7 Инсталиране на Bosch IoT Edge Services

- Отново в страницата с Edge Downloads се изтегля “Edge Services Runtime for Raspberry Pi”.
- Разархивира се инсталираната услуга.
- Копира се Bosch IoT Edge Services до микрокомпютъра, ако е сигурно, че машината има връзка с него.

```
$ scp -r target-image-iot-edge-services-linux-arm-
raspbian pi@<ip-address>:/home/pi/<path-to-project-
directory>
```

- Следва се [наръчника за инсталиране на Bosch IoT Edge Services](#), за да се инсталира услугата върху потребителското Raspberry Pi.

4.1.8 Добавяне на камерата към системата

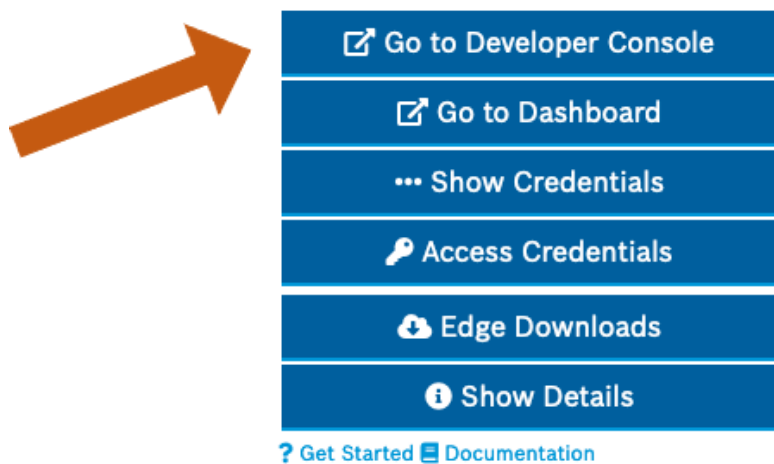
- Стартира се Bosch IoT Edge Services.
- Инсталира се уеб конзола, следвайки [наръчника за инсталиране на уеб конзола](#), за да може да се достъпят функционалните единици на инсталираните пакетите.
- Инсталират се пакетите за ONVIF поддръжка, следвайки [наръчника за инсталиране на ONVIF поддръжка](#).
- Отваря се в браузър линка “http://<host>:<port>/system/console”, като се влиза в профил с потребителско име “admin” и парола “admin”.
- Използва се [наръчника за регистриране на ONVIF устройства](#), за да се регистрира потребителската камера.

4.1.9 Добавяне на ключалката към системата

- Стартира се Bosch IoT Edge Services, ако вече не е.
- Инсталират се пакетите за Z-Wave поддръжка, следвайки [наръчника за инсталиране на Z-Wave поддръжка](#).
- Отваря се в браузър линка “http://<host>:<port>/system/console”, като се влиза в профил с потребителско име “admin” и парола “admin”.
- Използва се [наръчника за регистриране на Z-Wave устройства](#), за да се регистрира потребителската ключалка.

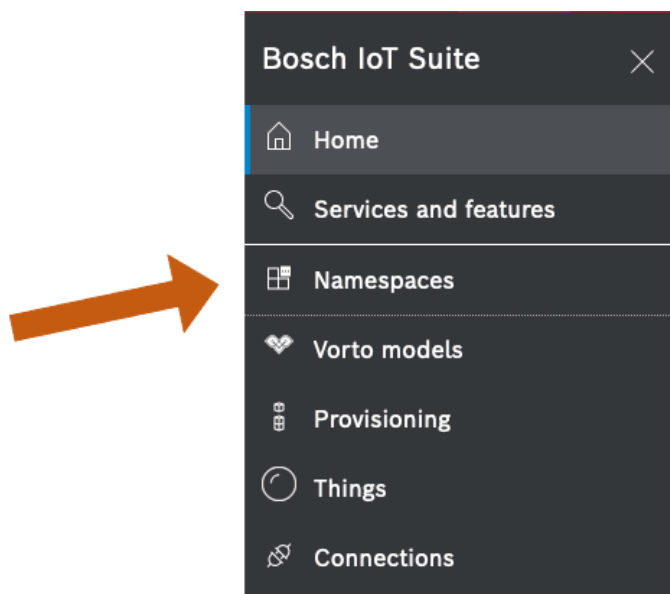
4.1.10 Създаване на дигитална репрезентация на “Gateway Device”

- Навигира се до потребителския [Subscription](#) в Bosch IoT Suite.
- Натиска се бутона “Go to Developer Console” (Фиг. 4.2), за да се достъпи конзолата на потребителския Subscription, където ще има достъп до своите дигитални близнаци.



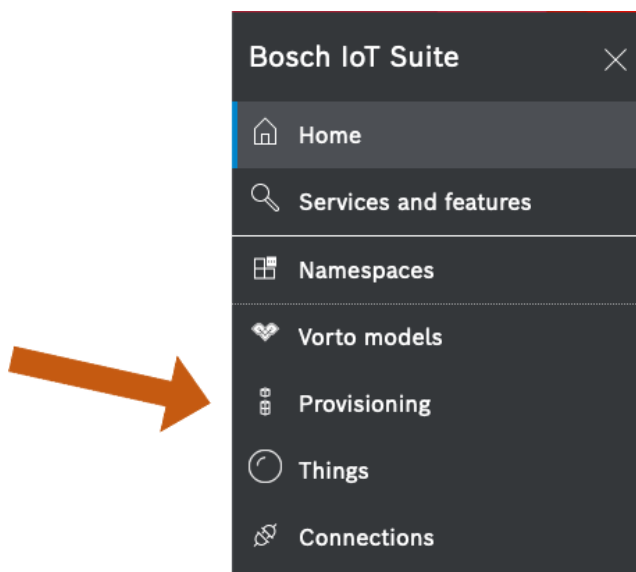
Фиг. 4.2: “Go to Developer Console” на Subscription

- Натиска се бутона “Namespaces” (Фиг. 4.3) и се създава нов Namespace, следвайки указаните стъпки, така че потребителските дигитални близнаци да бъдат свързани с неговия Subscription в Bosch IoT Suite.



Фиг. 4.3: “Namespaces” в конзолата на Subscription

- Натиска се бутона “Provisioning” (Фиг. 4.4).



Фиг. 4.4: “Provisioning” в конзолата на Subscription

- Създава се “provisioning” за мрежови шлюз, като се използват за пример стойностите на полетата, показани във Фиг. 4.5.

Settings ⓘ

Presets ⓘ

Besides starting from scratch, you can either use a Vorto information model or load a template

Start from scratch (default) ▾

Select how the device is connected ⓘ

Gateway device ▾

Gateway authorities ⓘ

☒ Enable automatic provisioning of edge devices

Thing ID

Set the ID of the thing, device and policy

Namespace ⓘ
finalyearproj (default) ▾

Name ⓘ
example-name

Device authentication

Select the credential type used to connect the device to the Bosch IoT Hub

Username and password ▾

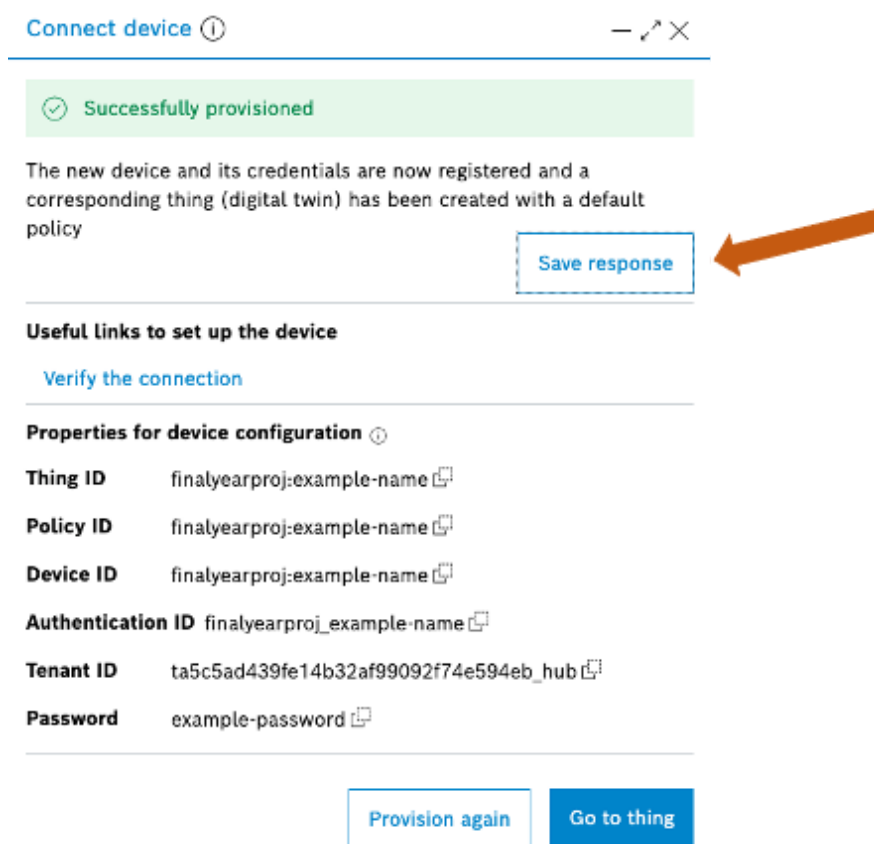
Define a password. The generated (recommended) username can be changed in the next step

Plaintext password ⓘ
example-password

Next

Фиг. 4.5: Примерни стойности на полетата във формата за създаване на “provisioning” на мрежови шлюз

- Запазва се файла “provisioning.json” посредством бутона “Save response” (Фиг. 4.6).



Фиг. 4.6: Бутон “Save response” за запазване на файла “provisioning.json”

- Копира се изтегления файл “provisioning.json” върху потребителското Raspberry Pi посредством командата “scp” от машина му и се поставя в директорията, където се намира услугата Bosch IoT Edge Agent.

```
$ scp provisioning.json pi@<ip-address>:/home/pi/<path  
-to-project>/com.bosch.iot.edge.agent.assemblies.dist-pack  
_linux_arm_raspbian
```

4.1.11 Стартиране на системата от свързани устройства

• Тази стъпка се изпълнява само върху Raspberry Pi, затова потребителят трябва да се свърже към него, използвайки указанията в т. 4.1.1.

• Навигира се до директорията, в която е поставен функционалния код на системата, както и услугите Bosch IoT Edge Agent и Bosch IoT Edge Services.

```
$ cd <path-to-project>
```

• Стартира се Bosch IoT Edge Agent, следвайки [наръчника за стартиране на Bosch IoT Edge Agent](#).

• Стартира се Bosch IoT Edge Services.

```
$ cd target-image-iot-edge-services-linux-arm-raspbian/  
osgi/bin/vms/jdk  
$ ./server.sh
```

• Създава се нова “ssh” връзка към потребителското Raspberry Pi, използвайки указанията в т. 4.1.1.

• Навигира се до папката, в която е инсталиран функционалния код на проекта. Инсталират се необходимите библиотеки.

```
$ cd <path-to-project>/src  
$ ./install.sh
```

• Стартира се системата посредством скрипта “main.py”.

```
$ python3 main.py
```

4.2 Стартиране на мобилното приложение

4.2.1 iOS

Тъй като iOS не поддържа инсталирането на приложения посредством изпълним файл, мобилното приложение трябва да се инсталира върху машината, за да може да се стартира върху iPhone.

- Инсталирайте приложението от [публичното хранилище в Github](#).
- Отворете проекта в приложението Xcode. Той се намира в директорията “src/mobile/ios/”.
- Свържете вашия iPhone към вашата машина.
- Стартирайте приложението като release версия върху вашия телефон. Процесът може да отнеме няколко минути, но накрая ще можете да достъпите приложението на вашия iPhone по всяко време.

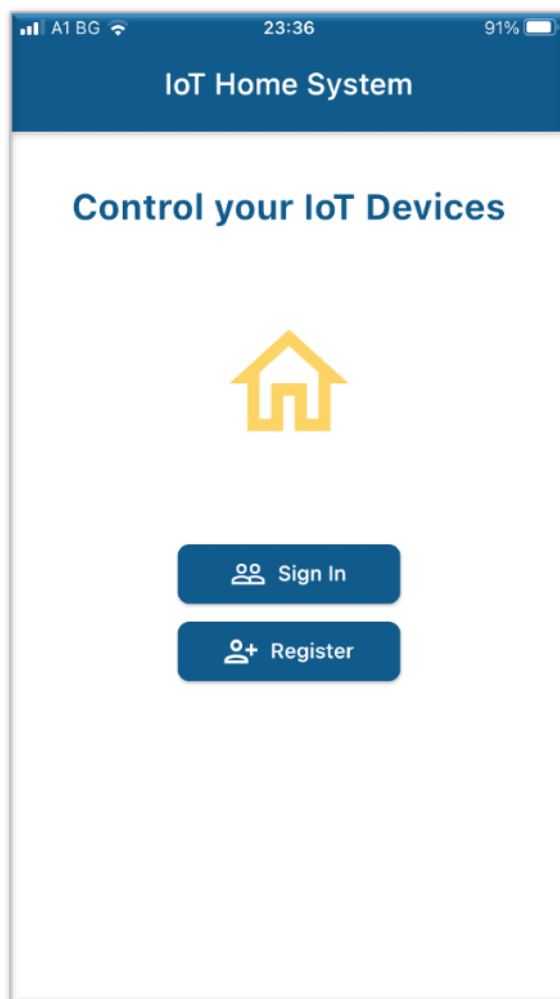
4.2.2 Android

- Инсталирайте приложението от [публичното хранилище в Github](#).
- Навигирайте до директорията “src/mobile/android/”.
- Вътре в нея се намира изпълним файл с разширение “.apk”.
- Инсталирайте “.apk” файла върху вашето Android устройство, като това може да отнеме до няколко минути, но накрая ще можете по всяко време да използвате мобилното приложение по всяко време.

4.3 Използване на приложението

4.3.1 Начална страница

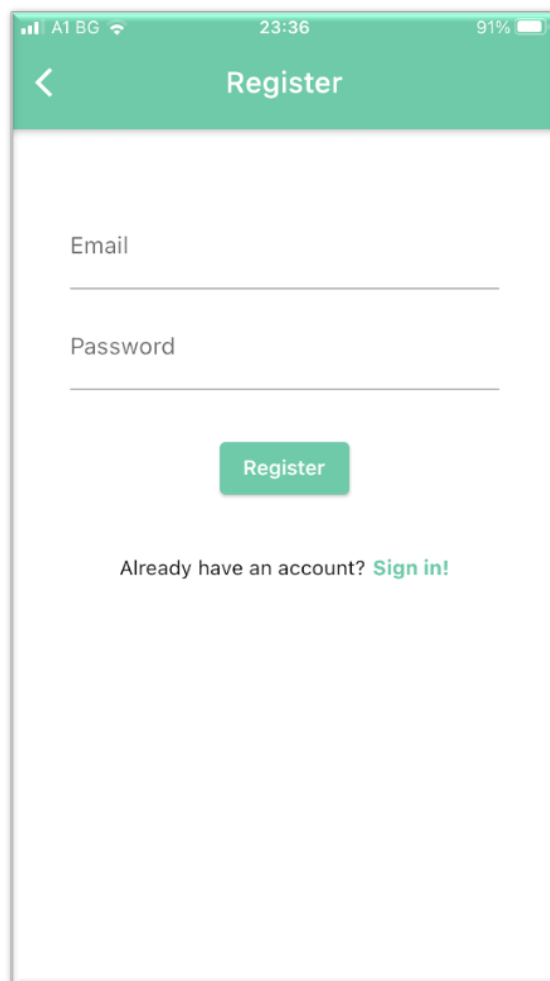
При стартиране на приложението се попада на начална страница (Фиг. 4.7), върху която има бутони за създаване на профил или влизане в такъв.



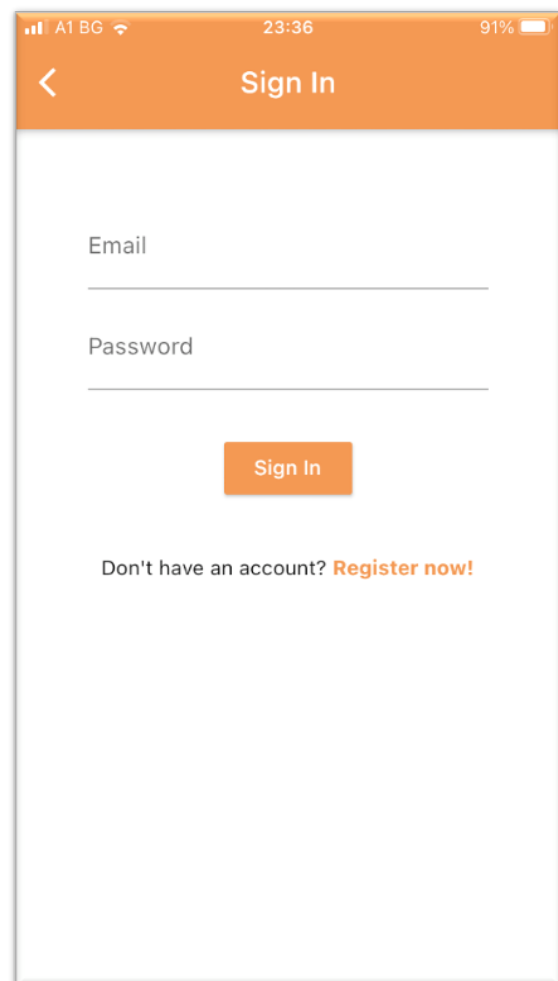
Фиг. 4.7: Начална страница

4.3.2 Създаване и влизане в профил

При създаване или влизане в профил от потребителя се изисква да въведе своя имейл и своята парола. При създаването на профил паролата трябва да е поне 6 символа. И двете форми изкарват съобщение за грешка ако някое от полетата липсва или ако имейлът е грешно форматиран. Страниците за създаване на профил и влизане в профил са изобразени съответно във *Фиг. 4.8* и *Фиг. 4.9*.

The screenshot shows a mobile application interface for a registration page. At the top, there is a green header bar with a back arrow on the left and the word "Register" in the center. Below the header, the page has a white background. There are two input fields: "Email" and "Password", each with a horizontal line underneath. Below these fields is a green button with the text "Register". At the bottom of the page, there is a link that says "Already have an account? Sign in!" where "Sign in!" is in green.

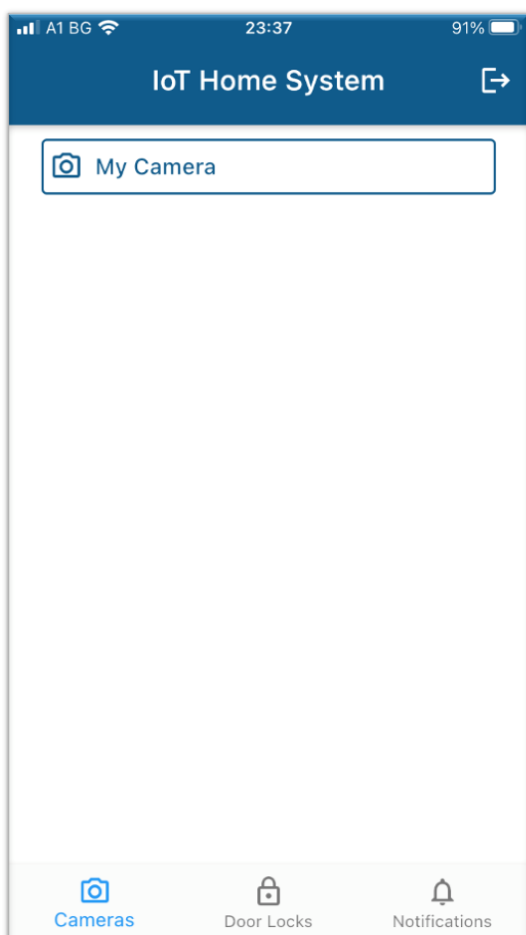
Фиг. 4.8: Страница за създаване на профил

The screenshot shows a mobile application interface for a sign-in page. At the top, there is an orange header bar with a back arrow on the left and the words "Sign In" in the center. Below the header, the page has a white background. There are two input fields: "Email" and "Password", each with a horizontal line underneath. Below these fields is an orange button with the text "Sign In". At the bottom of the page, there is a link that says "Don't have an account? Register now!" where "Register now!" is in orange.

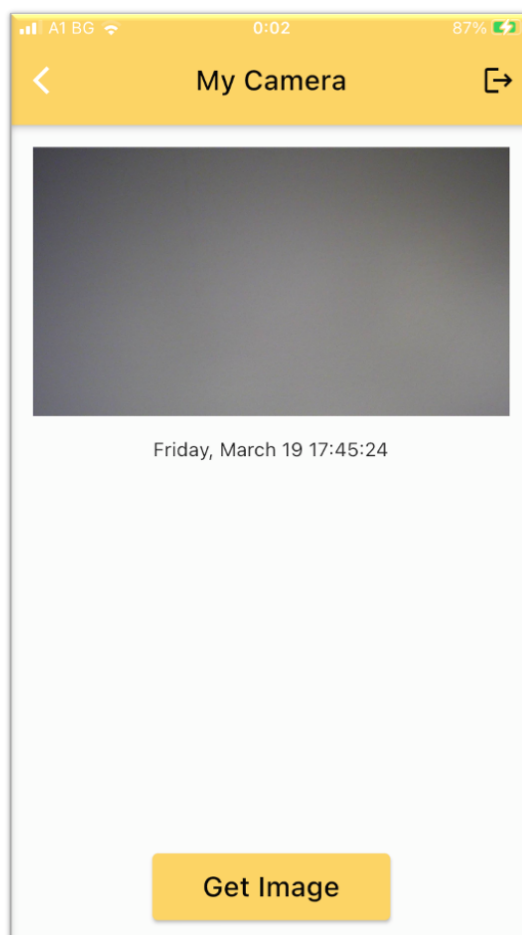
Фиг. 4.9: Страница за влизане в профил

4.3.3 Управление на камери

При влизане в профил на екрана на потребителя се показва списък от камери (Фиг. 4.10), като всяка камера се характеризира със своето име. При натискане върху някой от бутоните за камери, приложението отваря нова страница (Фиг. 4.11), която зарежда последната снимка, която потребителят е пожелал да види от камерата. В дъното на страницата се намира бутон, който изпраща заявка към системата от свързани устройства, за да се вземе нова снимка. Изображението се изпраща от системата във Firebase Cloud Storage, като се изпраща и часът, в който е направено изображението във Firebase Cloud Firestore. Страницата слуша за обновяване на данните в Cloud Firestore, за да вземе новото изображение и новия час.



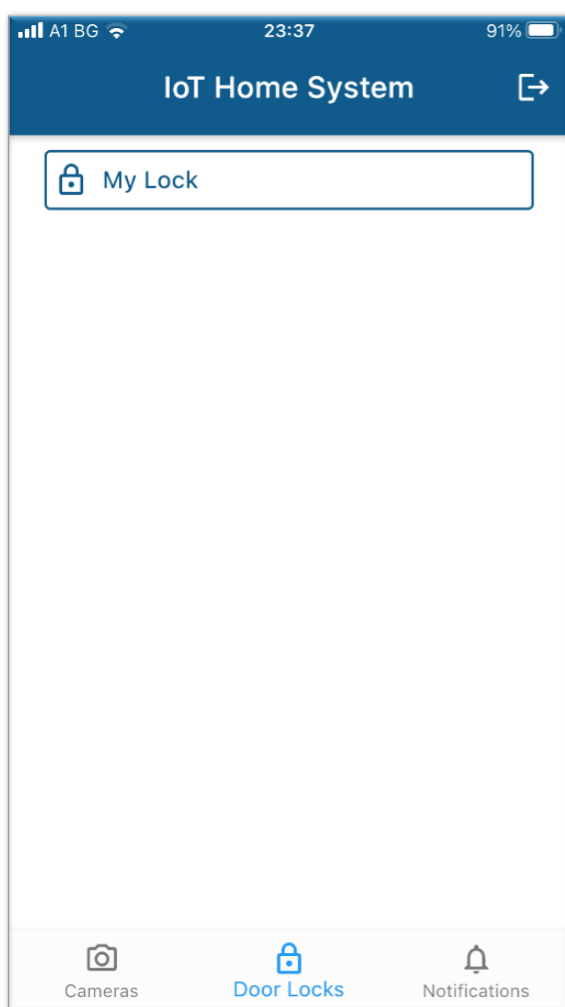
Фиг. 4.10: Страница със списък от камери



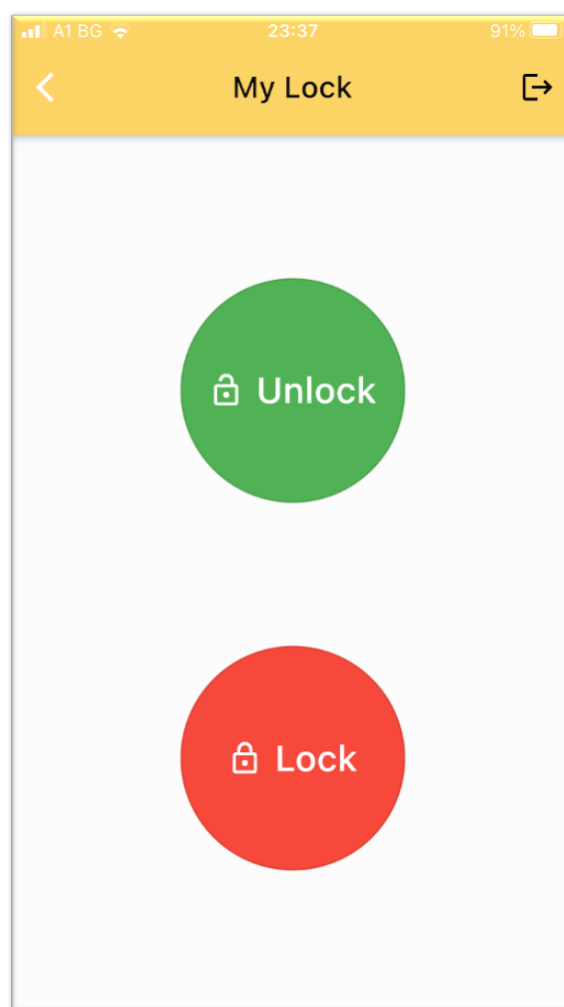
Фиг. 4.11: Страница на една камера

4.3.4 Управление на ключалки

Във втората подстраница се намира списъкът с потребителски ключалки (Фиг. 4.12), като всяка ключалка се характеризира с име. При натискане на бутон за някоя ключалка приложението отваря нова страница (Фиг. 4.13) със заглавие името на камерата. В тялото на страницата са поместени два бутона – “Lock” и “Unlock”. Чрез тях могат да се изпратят съобщения до ключалката през Bosch IoT Suite. По този начин потребителят има отдалечен достъп до своите устройства.



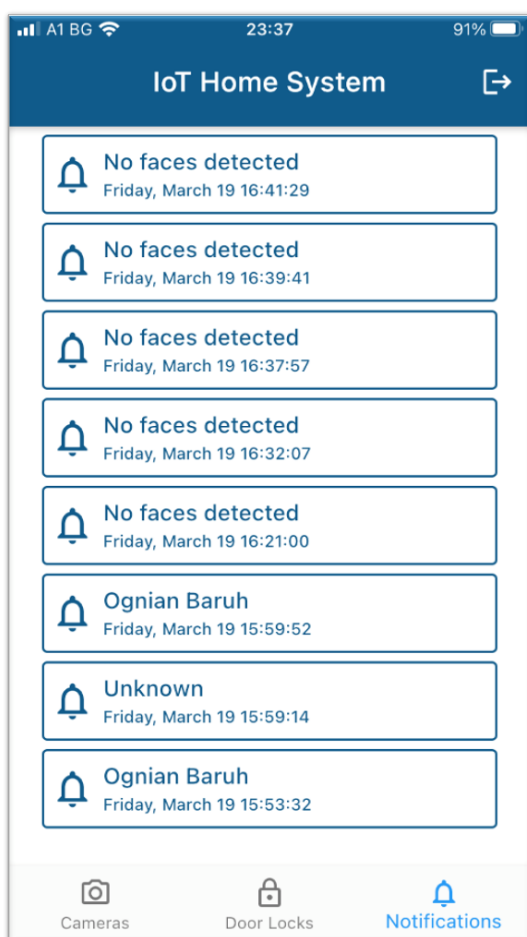
Фиг. 4.12: Страница със списък от ключалки



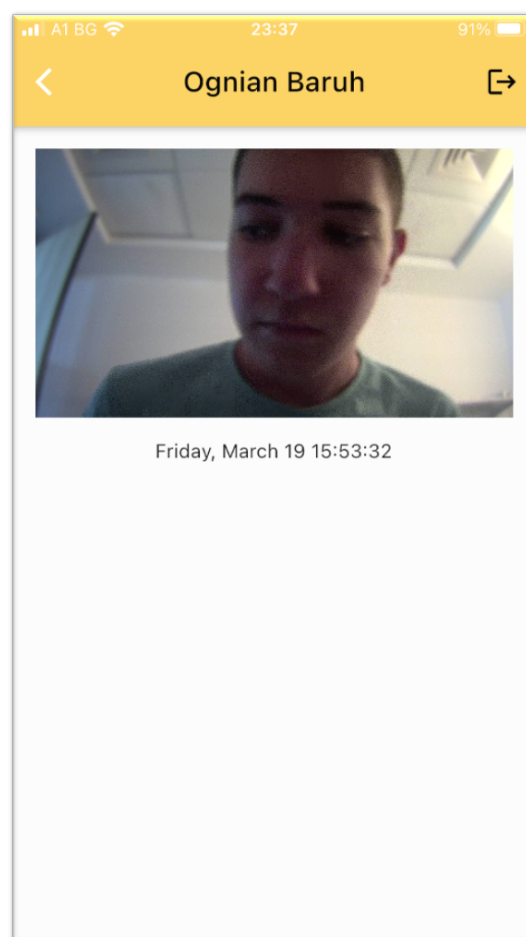
Фиг. 4.13 Страница на една ключалка

4.3.5 Управление на известия

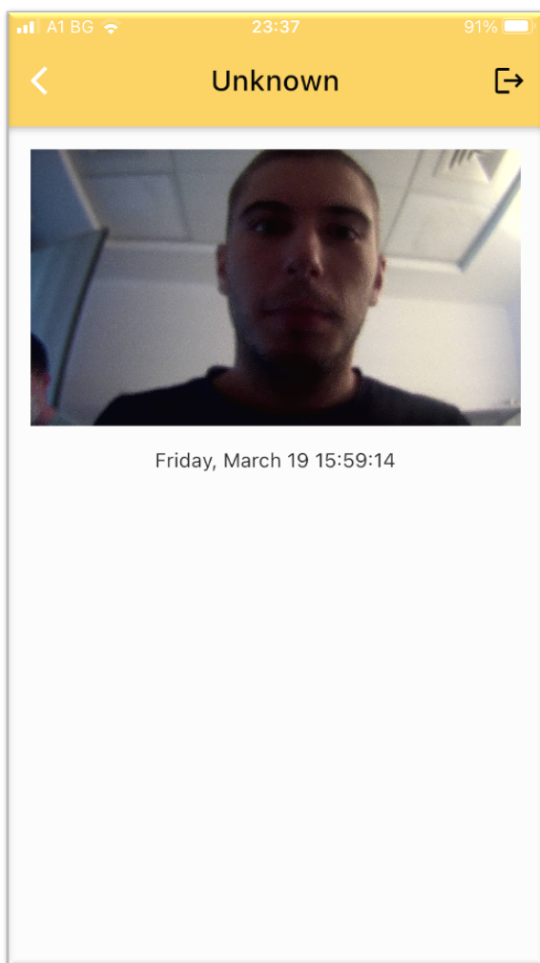
Последната подстраница съдържа списък с всички известия, които потребителят е получавал от неговите камери (Фиг. 4.14). Те се получават, когато някоя камера изпълни моделът за лицево верифициране. Всяко известие в списъка е бутон със заглавие засечените лица и с подзаглавие часа, когато е засечен обектът. При натискането на някой от бутоните се отваря нова страница, която показва и изображението, върху което е изпълнен моделът за лицево верифициране. Страницата на едно известие може да има едно от три вида заглавия – имена на засечените лица (Фиг. 4.15), “Unknown” (Фиг. 4.16), ако засеченото лице е непознато, или “No faces detected” (Фиг. 4.17), ако засеченият обект не е лице на човек.



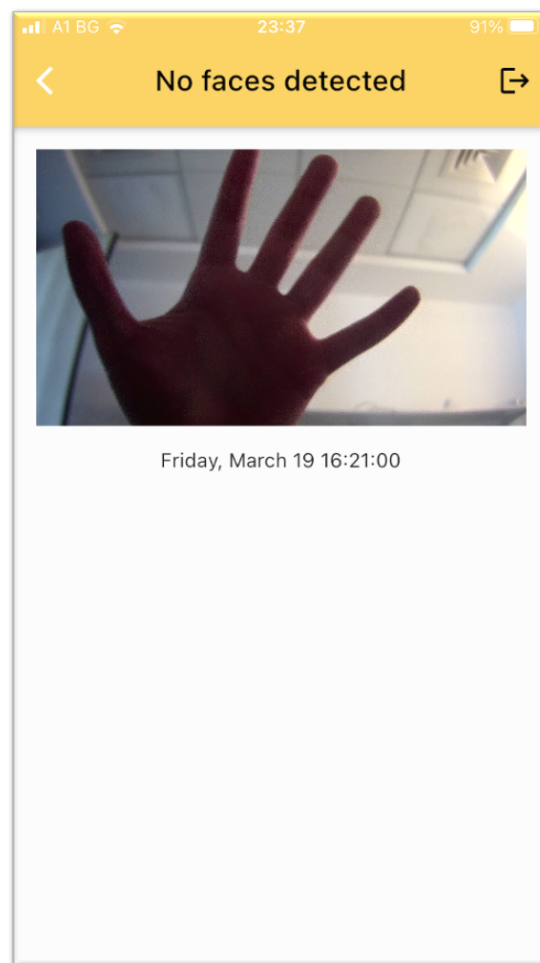
Фиг. 4.14: Страница със списък от известия



Фиг. 4.15: Страница с известие с резултат име на засеченото лице



Фиг. 4.16: Страница с известие с резултат "Unknown"



Фиг. 4.17: Страница с известие с резултат "No faces detected"

ЗАКЛЮЧЕНИЕ

Системата от свързани устройства покрива всички изисквания, описани в заданието. Използването ѝ е лесно и не изисква почти никакви усилия. Към нея могат да бъдат добавени много и различни умни устройства, които да могат да бъдат контролирани от мобилното приложение.

Мобилното приложение покрива всички функционалности, заложи в заданието, както и допълнителни такива. Използването му е много интуитивно и осигурява на всеки потребител бърз и удобен начин да контролира своите устройства. Приложението може да бъде развито в много посоки – да се добави възможност за регистриране на устройства, да поддържа повече видове устройства. Използването на Flutter за реализация на мобилното приложение позволява то да се използва върху смартфони с различни операционни системи без никакви разлики в предоставените функционалности.

ИЗПОЛЗВАНА ЛИТЕРАТУРА

1. [Официален сайт на NumPy](#)
2. [Официален сайт на MQTT](#)
3. [Официална страница на Raspberry Pi](#)
4. [Официална документация на Bosch IoT Suite](#)
5. [Официална документация на Bosch IoT Hub](#)
6. [Официална документация на Bosch IoT Things](#)
7. [Основи на Bosch IoT Edge Services](#)
8. [Основи на Bosch IoT Edge Agent](#)
9. [Официална документация на Bosch IoT Edge](#)
10. [Официална страница на Z-Wave](#)
11. [Официална страница на ONVIF](#)
12. [Официална страница на Flutter](#)
13. [Официална страница на Firebase](#)
14. [Официална страница на Firebase Authentication](#)
15. [Официална страница на Firebase Cloud Storage](#)
16. [Официална страница на Firebase Cloud Firestore](#)
17. [Официална документация на Eclipse Ditto](#)