Университет ИТМО

Курсовая работа
по дисциплине «Встроенные системы»
на тему «Умный замок»

Работу выполнили:
Возжаев Артем
Хрулёв Виктор
Кокорин Роман
Глушков Дмитрий
Добровицкий Дмитрий
Порядин Арсений
Дерябин Андрей

Группа:
P3410

Санкт-Петербург
2020

Содержание

Цель работы

Разработать умный замок на основе двухфакторной аутентификации, построенный на базе 3 компонентов:

- Android приложение
- Сервер на Kotlin
- Контроллер замка на базе микроконтроллера

Замок выполняет функцию блокировки дверей на предприятии. Замок располагается рядом с предметом управления (дверью). Для управления замком необходимо использовать Android приложение, с помощью которого можно осуществлять открытие дверей, а также заниматься администрированием.

Задачи

1. Разработать архитектуру проекта
2. Реализовать компоненты системы
   2.1. Прошивку для контроллера STM32
   2.2. Мобильное приложение для управления контроллером
   2.3. Сервер для связи мобильного приложения с контроллером
3. Провести интеграционное тестирование реализованной системы
4. Сформировать отчёт по проделанной работе

Описание системы

Была поставлена цель разработать умный замок на основе двухфакторной аутентификации.

Умный замок находится в состоянии “закрыт”. Пользователь умного замка в приложении выбирает замок и нажимает на кнопку открыть. После чего запрос отправляется на сервер, который переводит замок в состояние ввода пароля, пин-код, который случайным образом генерируется на сервере. для ввода отправлятся в Android приложение. Пользователь получает новый код в android приложении и вводит его в умный замок. После проделанных действий умный замок переходит в статус “открыто”. Измените статуса на “закрыто” производится автоматически по таймауту.

Также система предоставляет возможность администрирования, добавление/удаление пользователей, обновлять пароли пользователей, добавление/удаление замков, а также изменение прав пользователей на открытие замков.

Разрабатываемый умный замок на основе двухфакторной аутентификации (далее проект) состоит из следующих **программных** компонентов:

1. Управляющая программа - Обеспечивает управление аппаратной частью умного замка.
2. Android приложение - Обеспечивает взаимодействие пользователя и умного замка, через программный сервер.
3. Программный сервер - Связывает android приложение и управляющую программу.

Проект состоит из следующих **аппаратных** компонентов:

1. Отладочная плата на базе MCU - Аппаратная часть умного замка.
2. Аппаратный сервер - Аппаратная платформа для обеспечения работы программного сервера.

Разрабатываемый умный замок на основе двухфакторной аутентификации (далее проект) состоит из следующих **программных** компонентов:

1. Управляющая программа - Обеспечивает управление аппаратной частью умного замка.
2. Android приложение - Обеспечивает взаимодействие пользователя и умного замка, через программный сервер.
3. Программный сервер - Связывает android приложение и управляющую программу.

Проект состоит из следующих **аппаратных** компонентов:

1. Отладочная плата на базе MCU - Аппаратная часть умного замка.
2. Аппаратный сервер - Аппаратная платформа для обеспечения работы программного сервера.

Архитектура системы

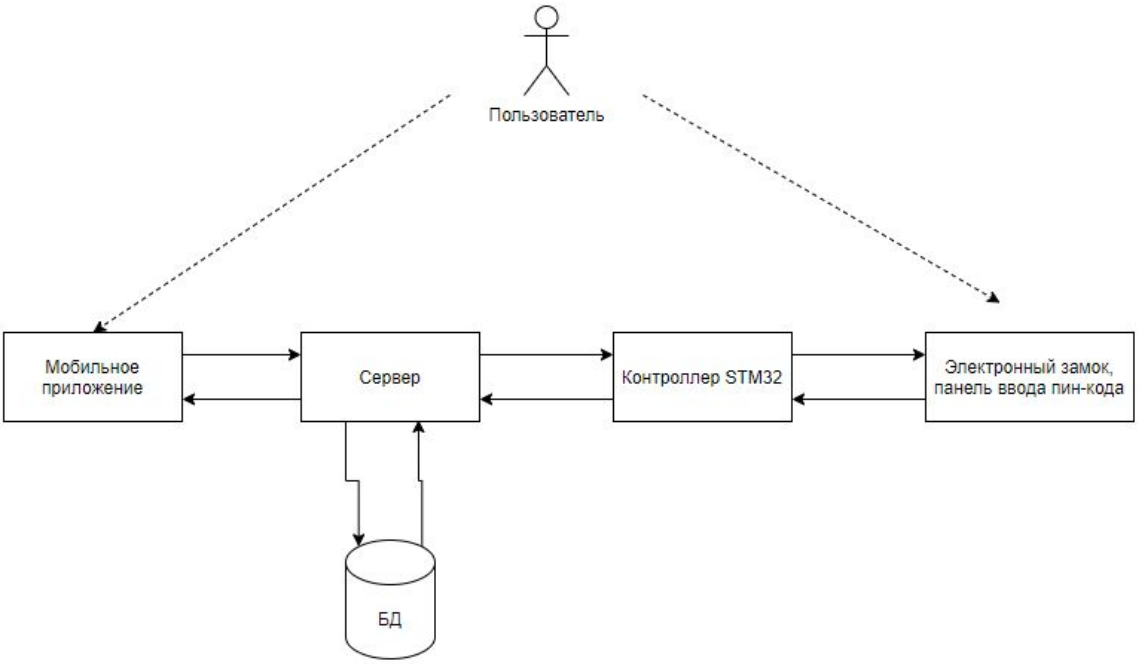

Рисунок 1 - Концептуальная модель системы

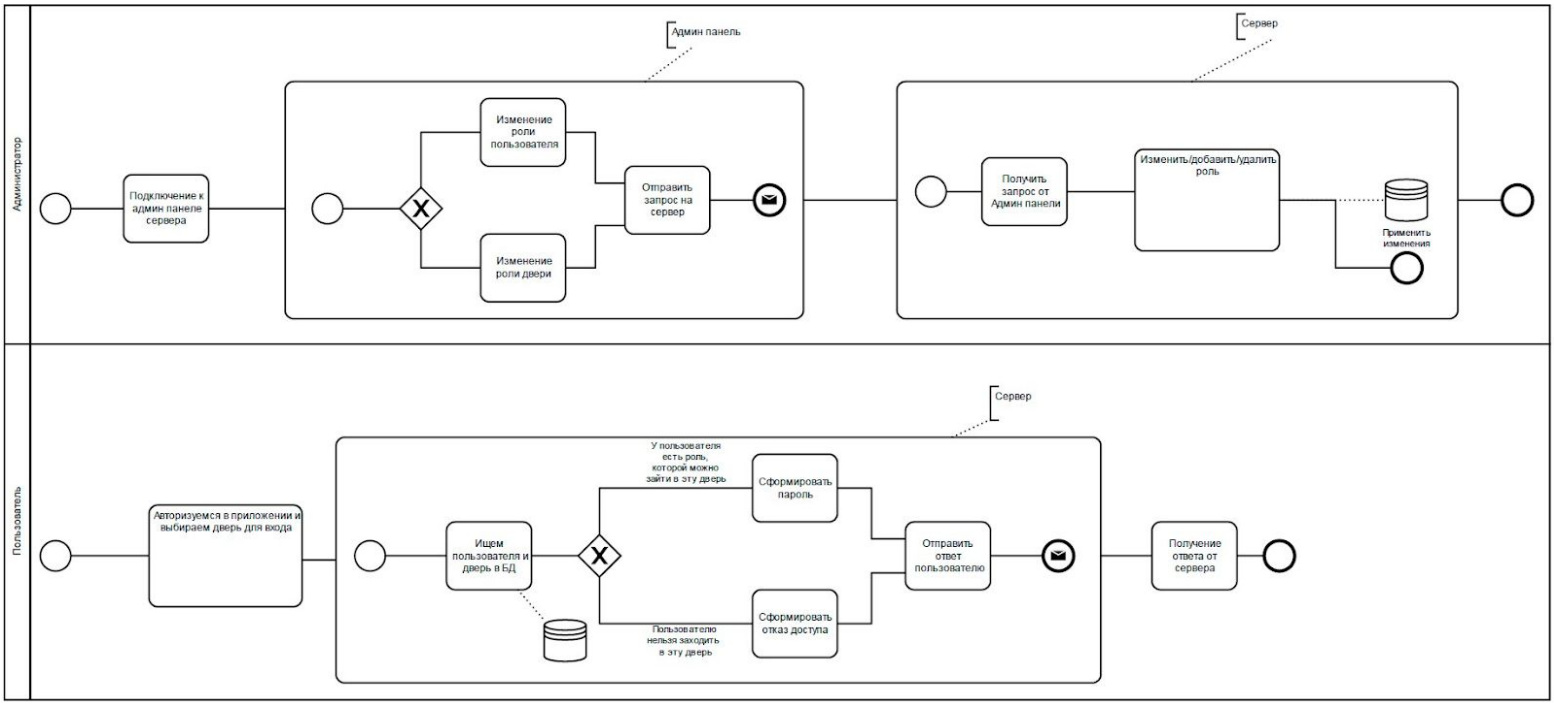

Рисунок 2 - Варианты использования системы администратором и рядовым пользователем

Ход работы

Этап 1.

Первым делом мы разработали архитектуру системы и разбились на три команды (STM32, сервер, клиент).

После этого мы создали репозитории для каждого проекта, настроили всем участникам права, договорились о порядке работы.

Далее каждая команда составила список задач (использовались Github Issues и Kanban-доска там же). После этого составленные задачи были распределены между участниками команд.

Также на данном этапе была определена структура API для взаимодействия сервера с клиентом, а также протокол взаимодействия сервера с контроллером STM32.

Этап 2.

На данном этапе разработка выполнялась параллельно в трёх командах:

Первая команда - Возжаев Артем и Хрулёв Виктор - занималась разработкой приложения для платы STM32.

Вторая команда - Глушков Дмитрий, Добровицкий Дмитрий и Порядин Арсений - занималась разработкой сервера приложений наряду с TCP-сервером.

Третья команда - Дерябин Андрей и Кокорин Роман - занималась разработкой клиентского приложения под Android.

Этап 3.

На данном этапе все разработанные компоненты были собраны в единую систему. Было произведено интеграционное тестирование, в результате которого мы убедились, что система функционирует корректно.

Приложение на плате STM32

Аппаратная часть контроллера замка

За основу была взята плата STM32L152RCT6 на базе микроконтроллера ARM Cortex-M3. Основными функциями которой являются:

- Управление взаимодействием с периферийными устройствами
- Основная логика контроллера на основе машины состояний
- Взаимодействие с сервером через интернет.

Для осуществления доступа в интернет была выбран wi-fi модуль ESP8266 с прошивкой esp-01. Подключение модулей произвели при помощи USART (это позволило передавать команды модулю и получать необходимую информацию от него), а взаимодействие с модулем производилось с помощью AT-команд.

Для общения с пользователем были использованы устройства ввода-вывода, такие как:

- Светодиоды, размещенные на плате - для индикации состояний
- Матричная клавиатура 4x1 - для ввода pin кода, который будет отправлен на сервер
- Для отображения информации о состояниях и введенном pin коде был использован экран LCD1602 (взаимодействие по шине i2c).

Для корректного подключения нужных модулей к STM32 был изучен datasheet [1]. В результате подключение устройств к STM32 осуществляется по следующей схеме:

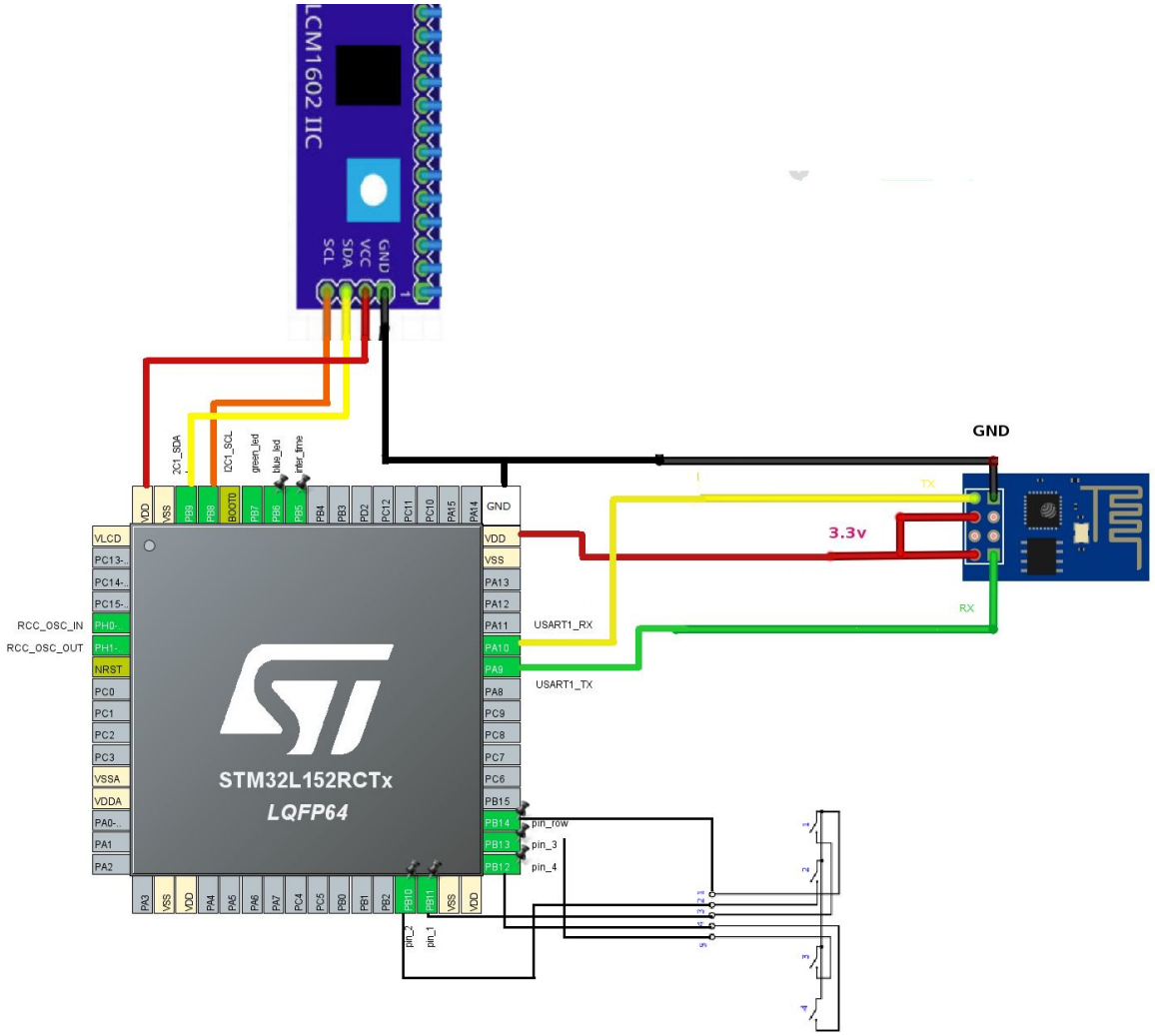

Рисунок 3 - Схема подключения устройства к STM32

Программная часть контроллера замка

Реализация TCP соединения с сервером (посредствам wi-fi модуля esp)

Подключение к серверу реализовано с помощью протокола TCP. Общение с сервером происходит с помощью собственных команд, представленных ниже:

Таблица 1 - Список команд для взаимодействия платы и сервера

Команда	Описание	Отправитель
GOS_HELLO=<ip>, <id>	Оповещение сервера о подключении контроллера	Контроллер замка

GOS_GET	Запрос ввода пароля	Сервер
GOS_OPEN	Запрос на открытие замка	Сервер
GOS_CLOSE	Запрос на закрытие замка	Сервер
GOS_PASS=<password>	Отправка введенного пароля	Контроллер замка
GOS_LOCKED	Подтверждение закрытия замка	Контроллер замка

Т.к. подключение к интернету реализовано через плату ESP8266, общение с которой осуществляется с помощью AT-команд, то для удобства была написана небольшая библиотека. Данная библиотек посредством USART позволяет взаимодействовать с модулем на более высоком уровне абстракции, а именно посредством вызова функций, которые с помощью сформированных AT-команд реализуют следующий функционал:

- Подключение к wi-fi сети
- Подключение к серверу
- Отправка и получение команд (описанных выше)

С реализацией можно ознакомится в Приложении 1 Прошивка STM32 (wifi.c).

Для тестирования данной библиотеки были написаны unit тесты(Приложение 1, unit_tests/), в которых модуль wi-fi подключается к компьютеру посредством адаптера, а вместо вызовов библиотеки HAL используются системные вызовы read и write. Сервер эмулировался TCP сокетом.

Реализация машины состояний с использованием FreeRtos

Для возможности осуществления одновременной работы с сервером и выполнения функции контроллера замка (вывод информации на экран, ввод пин кода, открытие/закрытие замка) было принято решение использовать операционную систему реального времени FreeRtos

FreeRtos был сконфигурирован следующим образом:

Task Name	Priority	Stack Size (...	Entry Function	Code Generat...	Parameter	Allocation	Buffer Name	Control Block...
defaultTask	osPriorityNor...	128	StartDefaultT...	Default	NULL	Dynamic	NULL	NULL
init_task	osPriorityNor...	128	InitTask	Default	NULL	Dynamic	NULL	NULL
close_state_tas	osPriorityNor...	128	CloseStateTask	Default	NULL	Dynamic	NULL	NULL
input_task	osPriorityNor...	128	InputTask	Default	NULL	Dynamic	NULL	NULL
open_task	osPriorityNor...	128	OpenTask	Default	NULL	Dynamic	NULL	NULL

Рисунок 4 - Конфигурация task во FreeRtos

Для реализации функционала контроллера была реализована машина состояний согласно следующей схеме:

Типы сообщений

1) Переход в состояние ввода пороля: GOS_GET (str)
2) Переход в состояние открыто: GOS_OPEN (str)
3) Переход в состояние закрыто: GOS_CLOSE (str)
4) Передача введеного пароля: GOS_PASS=<pin code>
5) Передача сообщения при старте: GOS_HELLO=<ip>,<id>

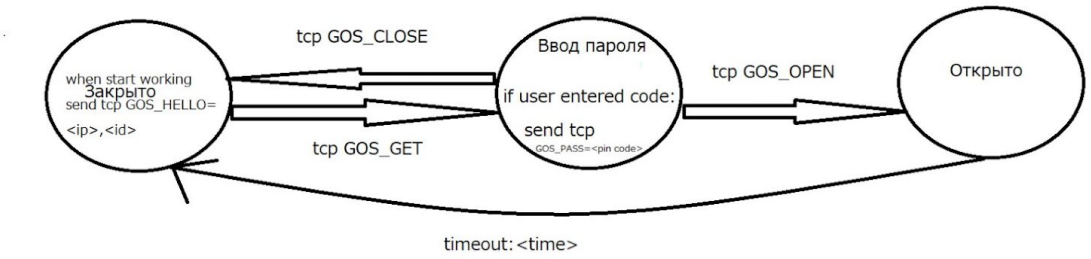

Рисунок 5 - Машина состояний контроллера замка

∨ API	
FreeRTOS API	CMSIS v2
∨ Versions	
FreeRTOS version	10.0.1
CMSIS-RTOS version	2.00
∨ Kernel settings	
USE_PREEMPTION	Enabled
CPU_CLOCK_HZ	SystemCoreClock
TICK_RATE_HZ	1000
MAX_PRIORITIES	56
MINIMAL_STACK_SIZE	128 Words
MAX_TASK_NAME_LEN	16
USE_16_BIT_TICKS	Disabled
IDLE_SHOULD_YIELD	Enabled
USE_MUTEXES	Enabled
USE_RECURSIVE_MUTEXES	Enabled
USE_COUNTING_SEMAPHORES	Enabled
QUEUE_REGISTRY_SIZE	8
USE_APPLICATION_TASK_TAG	Disabled
ENABLE_BACKWARD_COMPATIBILITY	Enabled
USE_PORT_OPTIMISED_TASK_SELECTION	Disabled
USE_TICKLESS_IDLE	Disabled
USE_TASK_NOTIFICATIONS	Disabled
RECORD_STACK_HIGH_ADDRESS	Disabled
∨ Memory management settings	

Рисунок 6 - Конфигурация FreeRtos

Описанная выше машина состояний реализована с помощью функционала, который предоставляется операционной системой, а именно все состояния были разбиты на функции, каждая из которых выполняет свою задачу:

- `InitTask` - осуществляется подключение к wi-fi сети и к серверу через es8266, прослушивает tcp соединение для отлавливания команд от сервера, которые отвечают за переход между состояниями.
- `CloseStateTask` - выводит на экран close (переводит в состояние закрыто), уведомление сервера о переходе в закрытое состояние.
- `InputTask` - ввод пин-кода с помощью матричной клавиатуры, вывод информации на экран а также отправка пин-кода на сервер. Также был добавлен таймер на 30 секунд, при истечении которого происходит автоматический переход в состояние закрыто.
- `OpenTask` - выводит на экран Open (что свидетельствует о переходе в состояние открыто). При истечении тайм аута в 3 секунды происходит автоматический переход в закрытое состояние.

Реализацию машины состояний можно увидеть в приложении 1 в файле main.c.

Реализация взаимодействия с устройствами ввода-вывода.

Подключение матричной клавиатуры осуществлялось посредством GPIO портов. В коде мы опрашиваем каждый порт и смотрим, какой из них перешёл в состояние SET. Соответствующий символ записываем в буфер пароля.

```
char read_keypad()
{
  HAL_GPIO_WritePin(GPIOB, pin_row_Pin, GPIO_PIN_SET);

  if ((HAL_GPIO_ReadPin(GPIOB, pin_1_Pin))) {
    while ((HAL_GPIO_ReadPin(GPIOB, pin_1_Pin)));
    return '1';
  }
  if ((HAL_GPIO_ReadPin(GPIOB, pin_4_Pin))) {
    while ((HAL_GPIO_ReadPin(GPIOB, pin_4_Pin)));
    return '4';
  }
  if ((HAL_GPIO_ReadPin(GPIOB, pin_2_Pin))) {
    while ((HAL_GPIO_ReadPin(GPIOB, pin_2_Pin)));
```

```
    return '2';
  }
  if ((HAL_GPIO_ReadPin(GPIOB, pin_3_Pin))) {
    while ((HAL_GPIO_ReadPin(GPIOB, pin_3_Pin)));
    return '3';
  }
}
```

LCD экран подключается через протокол I2C с использованием библиотеки для вывода строк на экран. Данная библиотека позволяет на более высоком уровне работать с I2C шиной посредством функций, которые используют регистры микроконтроллера для передачи символов на экран.

Серверная часть приложения

Общая информация о приложении

Сервер является связующим звеном между платой и клиентским приложением. На стороне сервера реализована вся бизнес-логика сервиса - общение с клиентским приложением через HTTP запросы, общение с приложением на STM32 через TCP соединение, взаимодействие с базой данных.

Одним из требований к серверной части было использование языка в процессе разработки языка Kotlin. Так как он является JVM языком, было решено использовать фреймворк Spring в качестве каркаса приложения.

Таблица 2 - Использованные компоненты Spring

Компонент	Краткое описание
Spring Boot	Инструмент, упрощающий процесс конфигурации и разработки приложений, использующих Spring.
Spring Integration IP	Инструмент, расширяющий программную модель Spring для поддержки Enterprise Integration Patterns. Позволяет использовать легковесное общение с внешними системами с помощью декларативных адаптеров.
Spring Boot Starter Data MongoDB (Spring Data MongoDB)	Диспетчер для использования документо-ориентированной базы данных MongoDB и Spring Data MongoDB.
Spring Boot Starter Web (Spring MVC)	Диспетчер построения веб приложений с помощью Spring MVC. Использует Tomcat в качестве контейнера по умолчанию.

Структура сервера и описание компонентов:,

В схеме ниже представлена схема структурная схема приложения со всеми файлами. В таблице 3 представлено описание структурных компонентов приложения.

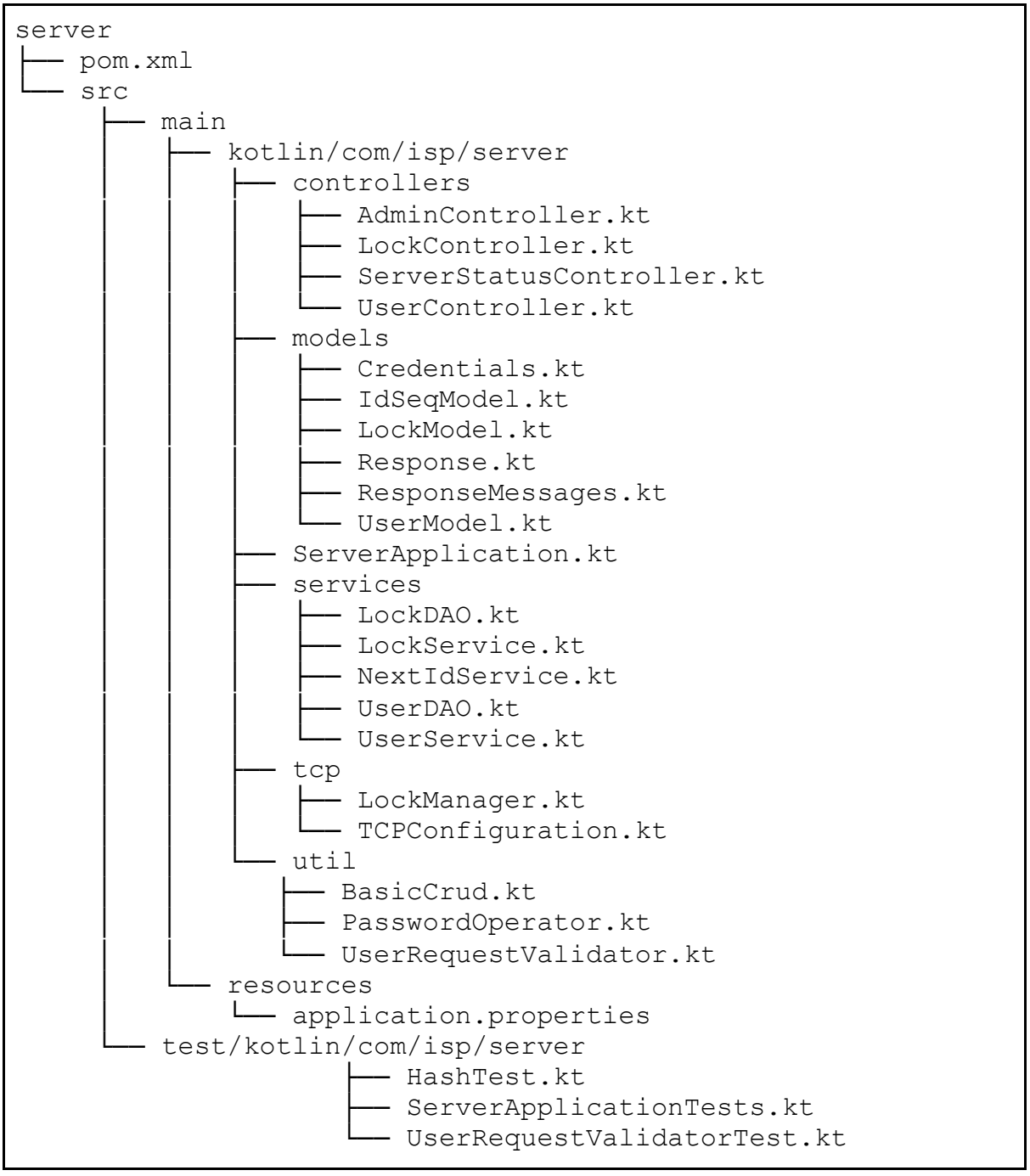

Таблица 3 - Описание основных структурных компонентов проекта

Структурный компонент	Краткое описание
pom.xml	Конфигурационный файл проекта. Содержит информацию о проекте и различных подключенных модулях. Значения из файла используются для билда приложения.

application.properties	Конфигурационный файл. Содержит данные для подключения к базе данных. Значения из файла передаются в рантайм приложения.
ServerApplication.kt	Entry point Spring-приложения
/controllers	Этот пакет содержит REST-контроллеры, реализующие REST интерфейс приложения. Здесь описываются адреса, на которые клиентское приложение может слать HTTP запросы и логика обработчиков полученных запросов.
/models	Этот пакет содержит data class'ы и enum'ы, использующиеся в работе приложения. Data class, помеченной аннотацией @Document используются в ORM для отражение записей из БД в объекты приложения.
/services	Этот пакет содержит сервисы, использующиеся в приложении. Примером такого сервиса могут выступать классы, реализующие логику использования ORM для извлечения и записи данных в БД.
/tcp	Этот пакет содержит Configuration класс для настройки TCP-сервера с целью поддержки общения с платой. Также, в этом пакете содержится класс адаптер, связывающий TCP-конфигурацию с сервисом и контроллерами Spring.
/util	Этот пакет содержат описание вспомогательных функций. Например, функции генерации пароля и генерации хеша пароля.
/test	В этой директории хранятся юнит-тесты компонентов приложения.

Аутентификация в приложении

Для обеспечения безопасности хранения пользовательских данных, при создании пользователя или получения запроса пароль хешируются с использованием алгоритма SHA-256.

Описание разработанного API

Все реализованные эндпоинты можно разделить на 2 категории: которые требуют привилегии администратора (начинаются с /admin/) и которые не требуют соответствующих прав (начинаются с /user/ и /lock/).

В таблице 4 представлен разработанный программный интерфейс с требуемыми форматами запросов и формата возвращаемых ответов. Подразумевается, что все запросы к серверу типа POST, и обязательно содержат в себе информацию о юзере, отправившем этот запрос.

Таблица 4 - Структура запросов и ответов при обращении к API

Path	Request Body	Response Body

/user/login	{
	credentials: {
	user_uid: int,
	password: string
	}
	}
		{
		status: [ok. denied],
		message: string,
		data: {
		privileges: [user, admin]
		}
		}

/user/update	{
	credentials: {
	user_uid: int,
	password: string
	},
	new_password: string
	}
		{
		status: [ok, denied],
		message: string
		}

/lock/all	{
	credentials: {
	user_uid: int,
	password: string
	}
	}
		{
		status: [ok, denied],
		message: string
		data: [{
		lock_uid: int,
		lock_name: string
		}]
		}

/lock/open	{
	credentials: {
	user_uid: int,
	password: string
		{
		status: [ok, denied],
		message: string
		data: [{

	}, lock_uid: int }	lock_uid: int, lock_PIN: string }] }

/lock/status	{
	credentials: {
	user_uid: int,
	password: string
	},
	lock_uid: int
	}
		{
		status: [pending, opened, denied],
		message: string
		}
		
		Описание статусов:
		denied:
		неправильные креды юзера
		неправильный uid замка
		нет доступа к замку
		opened:
		открыто
		pending:
		ожидает ввода PIN

/lock/cancel	{
	credentials: {
	user_uid: int,
	password: string
	},
	lock_uid: int
	}
		{
		status: [ok, denied],
		message: string
		}

/admin/user/all	{
	credentials: {
	user_uid: int,
	password: string
	}
	}
		{
		status: [ok, denied],
		message: string
		data: [{
		uid: int,
		privileges: [user, admin],
		lock_uids: int[],
		name: string,
		surname: string,
		}]
		}

/admin/user/add	{
	credentials: {
	user_uid: int,
	password: string
	},
	name: string,
	surname: string,
	privileges: [user, admin]
	}
		{
		status: [ok, denied],
		message: string
		data: {
		uid: int,
		password: string,
		name: string,
		surname: string,
		privileges: string,
		locks: int[]
		}
		}

/admin/user/delete	{
	credentials: {
	user_uid: int,
	password: string
	},
	target_user_uid: int,
	}
		{
		status: [ok, denied],
		message: string,
		}

/admin/user/update	{
	credentials: {
	user_uid: int,
	password: string
	},
	target_user_uid: int,
	reset_password: bool,
	new_privileges: [user, admin]
	}
		{
		status: [ok, denied],
		message: string,
		data: {
		uid: int,
		password: string,
		name: string,
		surname: string,
		privileges: string
		}
		}

/admin/user/lock/add	{
	credentials: {
	user_uid: int,
	password: string
	},
	lock_uid: int,
	target_user_uid: int
	}
		{
		status: [ok, denied],
		message: string,
		}

/admin/user/lock/delete	{
	credentials: {
	user_uid: int,
	password: string
	},
	lock_uid: int,
	target_user_uid: int,
	}
		{
		status: [ok, denied],
		message: string,
		}

/admin/lock/all	{
	credentials: {
	user_uid: int,
	password: string
	}
	}
		{
		status: [ok, denied],
		message: string
		data: [{
		uid: int,
		name: string
		}]
		}

/admin/user/lock/all	{
	credentials: {
	user_uid: int,
	password: string
	},
	target_user_uid: int,
	}
		{
		status: [ok, denied],
		message: string,
		data: [{
		uid: int,
		name: string
		}]
		}

/admin/lock/rename	{
	credentials: {
	user_uid: int,
	password: string
	},
	lock_uid: int,
	lock_new_name: string
	}
		{
		status: [ok, denied],
		message: string,
		}

/admin/lock/delete	{
	credentials: {
	user_uid: int,
	password: string
	},
	lock_uid: int
		{
		status: [ok, denied],
		message: string,
		}

	}	

Конфигурация TCP

Для настройки TCP-сервера на стороне сервера приложений было решено использовать Spring Integration Integration Patterns.

Данный инструмент предоставляет адаптеры каналов для приема и отправки сообщений по интернет-протоколам. Предусмотрены адаптеры UDP (User Datagram Protocol) и TCP (Transmission Control Protocol). Каждый адаптер обеспечивает одностороннюю связь по базовому протоколу. Кроме того, Spring Integration обеспечивает простые входящие и исходящие TCP-шлюзы. Они используются, когда требуется двусторонняя связь.

Использовались адаптеры входящих и исходящих каналов TCP:

- `TcpSendingMessageHandler` отправляет сообщения по протоколу TCP.
- `TcpReceivingChannelAdapter` получает сообщения по протоколу TCP.

Сама же конфигурация базового соединения обеспечивается с помощью фабрик соединений. Предусмотрены два типа фабрик: фабрика клиентских соединений и фабрика серверных соединений. Фабрика клиентских соединений устанавливают исходящие соединения. Фабрика соединений серверов прослушивают входящие соединения. Последняя (`TcpNetServerConnectionFactory`) использовалась в нашем проекте.

При этом, ссылаться на одну фабрику соединений может максимум один адаптер каждого типа.

Для реализации двусторонней произвольной связи было решено использовать адаптеры в связке с *TCP Connection ID* - уникальным идентификатором соединения, позволяющим различать сообщения от разных клиентов и отправлять произвольные сообщения нужному клиенту, используя адаптеры TCP-каналов. Таким образом, была реализована поддержка параллельного взаимодействия сервера с более чем одной платой.

Для примера приведём реализацию функции отправки произвольных сообщений клиенту и функции, определяющей адаптер используемого при отправки исходящего канала.

```
// Send arbitrary message
fun sendMessage(message: String, TCPConnId : String) {
      outputChannel().send(MessageBuilder
            .withPayload(message)
            .setHeader(IpHeaders.CONNECTION_ID, TCPConnId)
            .build())
}

// Outbound channel adapter
@Bean
@ServiceActivator(inputChannel = "outputChannel")
fun tcpSendingMessageHandler(cf:
AbstractServerConnectionFactory?)
: TcpSendingMessageHandler? {
      val tcpSendingMessageHandler = TcpSendingMessageHandler()
      tcpSendingMessageHandler.setConnectionFactory(cf)
      return tcpSendingMessageHandler
}
```

Выбор используемой базы данных

В качестве Базы Данных была выбрана MongoDB — документоориентированная система управления базами данных, являющаяся классическим примером No-SQL систем и хранящая данные в формате BSON (бинарные JSON-подобные документы).

Причиной такого выбора послужило несколько факторов:

1) Так как проект было необходимо разработать с нуля за достаточно короткий срок, почти до самого релиза у нас не было четкой структуры документов и эта структура постоянно менялась. В этом случае отлично проявляется главная особенность No-SQL баз данных - отсутствие требований к строгой структуре данных и необходимости после изменения этой структуры форматировать базу данных.
2) Те данных, которые мы храним, являются слабосвязанными между собой и поэтому нет необходимости в соединении таблиц или внешних ключах.

Структура коллекций

В таблицах 5 и 6 представлены структуры используемых коллекций. Коллекция lock содержит информацию о замках, коллекция user содержит информация о пользователях системы.

Таблица 5 - Структура коллекции lock

Имя коллекции	lock
Структура коллекции	uid: Number name: String TCPConnId: String ip: String

Таблица 6 - Структура коллекции user

Имя коллекции	user
Структура коллекции	uid: Number password: String name: String surname: String privileges: String lock_uids: Array<Number>

Мобильное приложение

Для реализации мобильного приложения был выбран фреймворк React Native с использованием Expo Managed workflow. В качестве основного языка программирования использовался Typescript.

Сначала было создано пустое приложение и составлен список экранов, которые необходимо реализовать. После этого были нарисованы wireframe-шаблоны экранов и подобрана библиотека UI компонентов React Native Paper. Внешний вид экранов был согласован с другими участниками проекта.

Затем были реализованы все экраны по отдельности и без функционала.

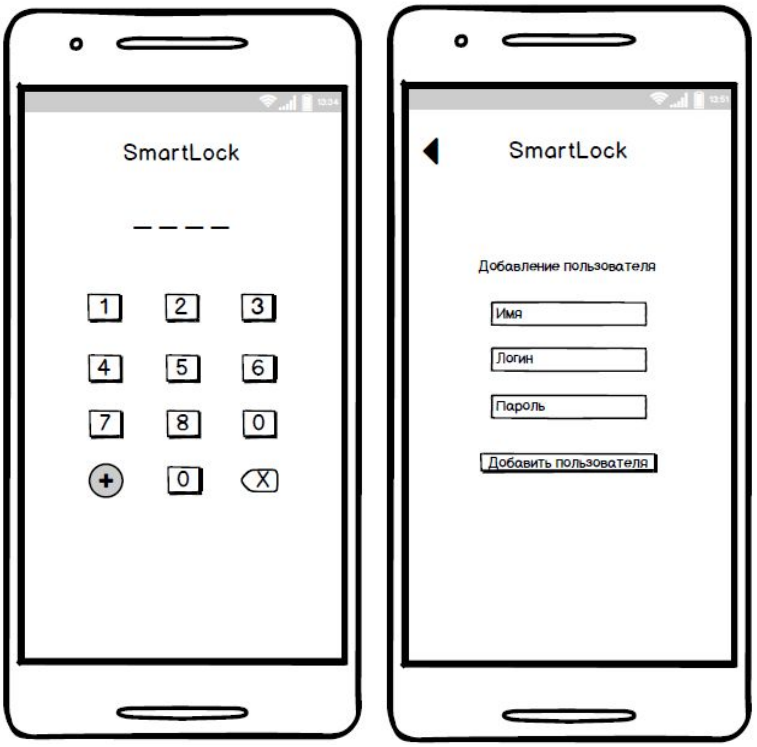

Рисунок 7 - Спроектированные макеты экранов

После этого были реализованы: класс для взаимодействия с API сервера по HTTP, класс для хранения, шифрования и извлечения данных из Localstorage, а также подключена и настроена библиотека Redux (с использованием библиотеки React-Redux).

Далее была реализована авторизация с помощью логина/пароля и выбор отображаемого набора экранов в зависимости от прав авторизованного пользователя.

В конце, был реализован функционал всех оставшихся экранов - список замков, отображение PIN-кода для ввода на клавиатуре замка, управление пользователями (для администраторов), настройки входа пользователя (смена пароля, вход с помощью PIN-кода вместо пары логин+пароль).

Рисунок 8 - Экран ввода пин-кода для входа

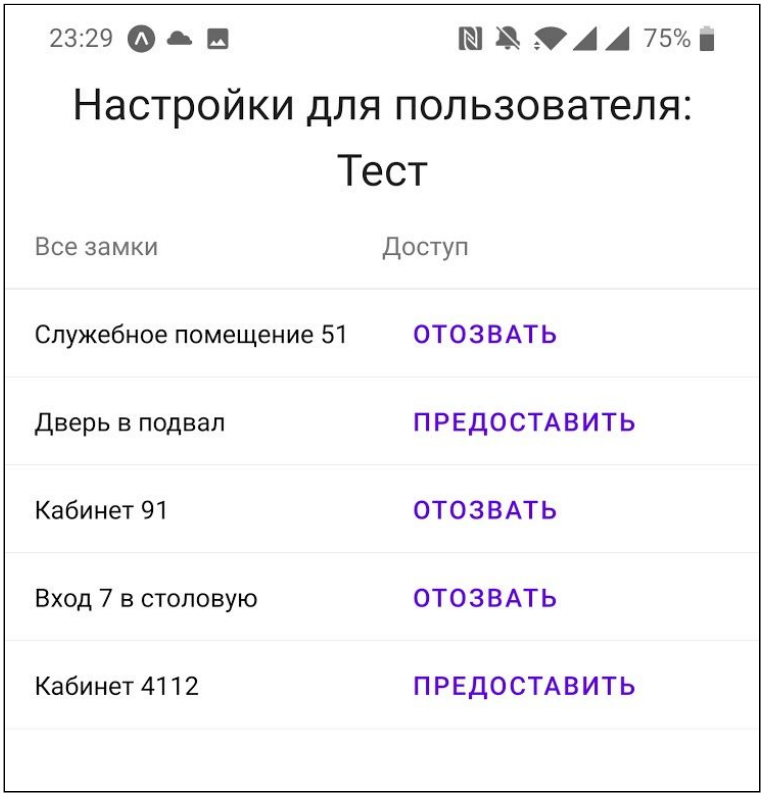

Рисунок 9 - Экран настройки доступа к замкам для пользователя

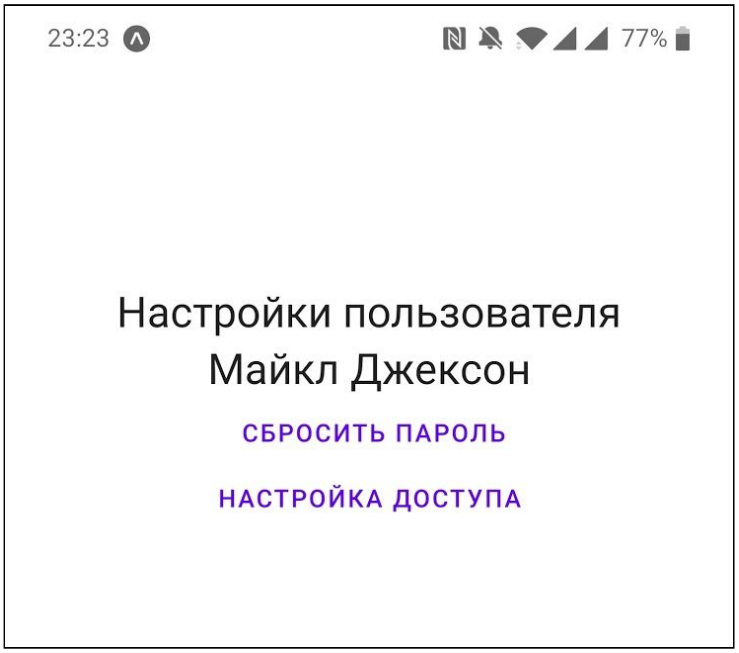

Рисунок 10 - Экран настройки пользователя

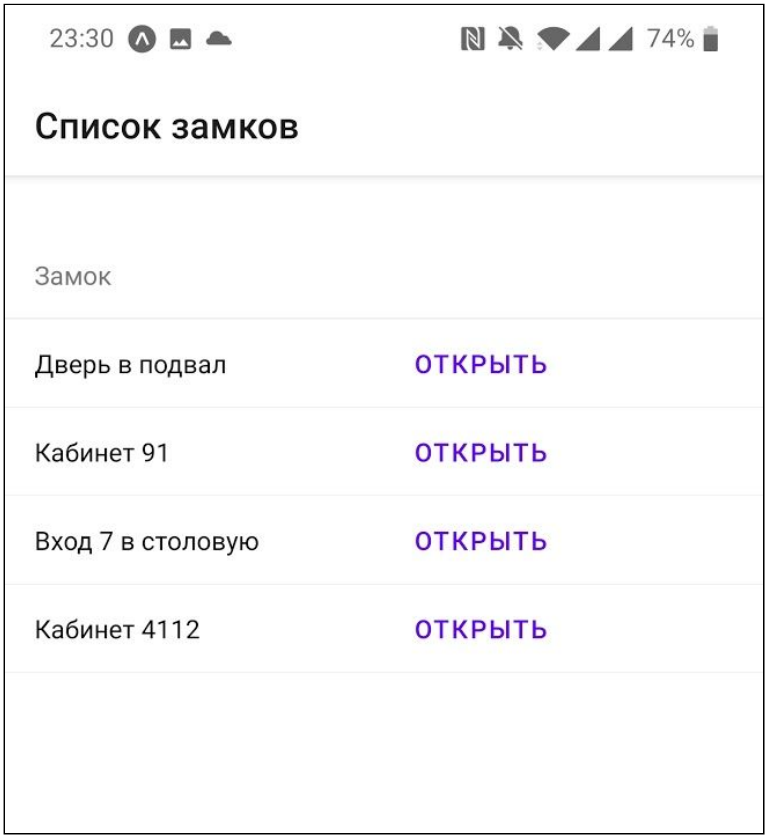

Рисунок 11 - Список замков для пользователя

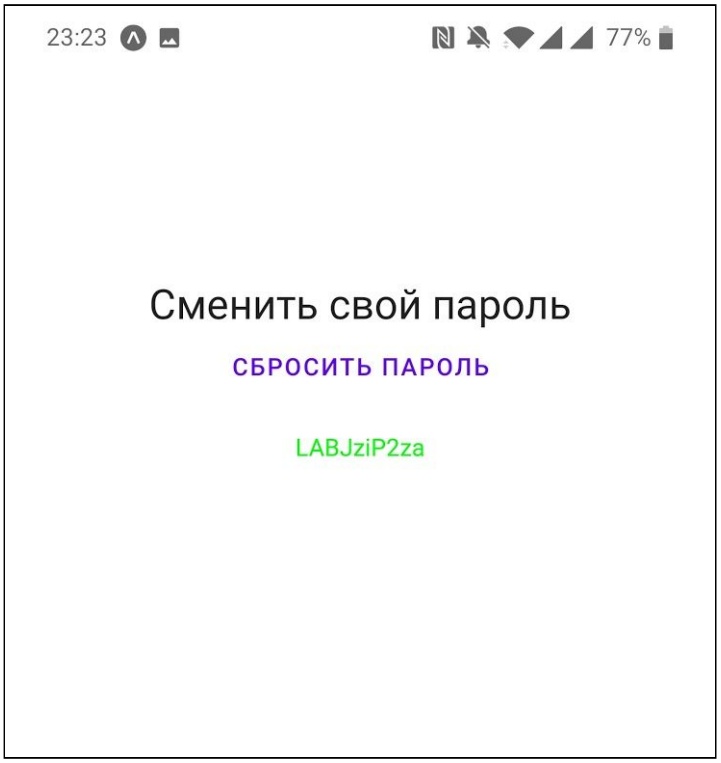

Рисунок 12 - Экран сброса пароля админом

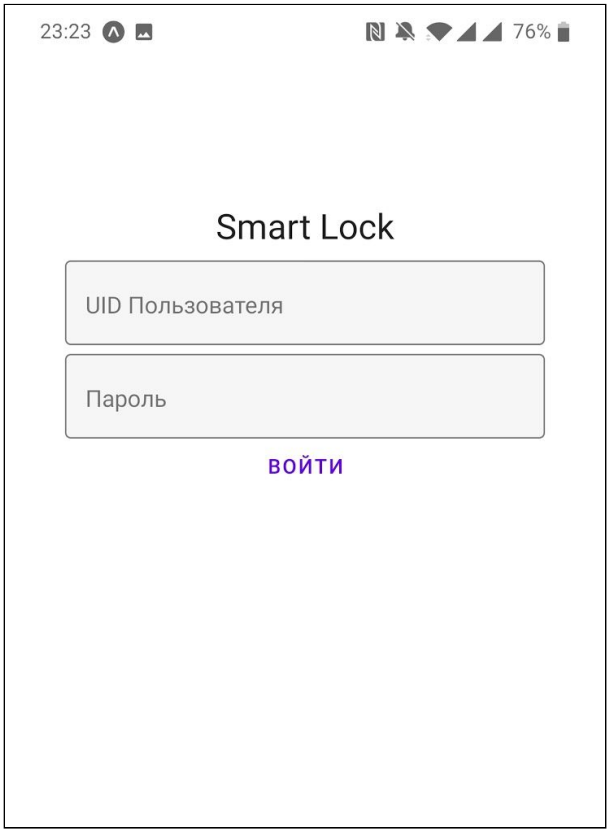

Рисунок 10 - Начальный экран входа

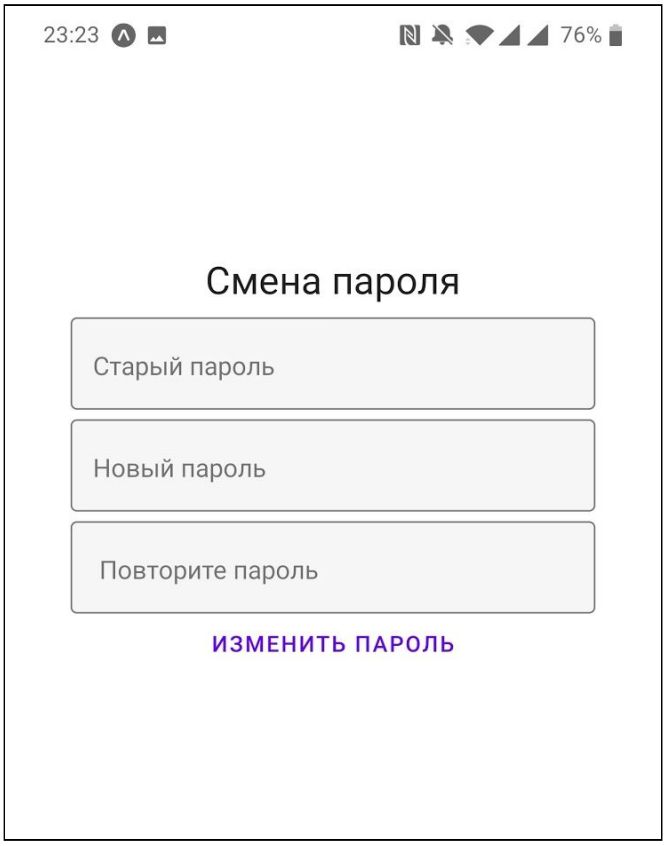

Рисунок 13 - Экран смены пароля

Результаты работы и заключение

В результате была получена работающая система, позволяющая открывать электронный замок с помощью двухфакторной аутентификации: id+пароль пользователя, вводимый через мобильное приложение и псевдослучайным образом генерируемый pin-код для ввода на клавиатуре замка.

Приложение 1. Исходный код

Исходный код и история изменений проекта доступны в репозиториях:

- Сервер: https://github.com/ES-10-01/ISP-Server
- Прошивка STM32: https://github.com/ES-10-01/ISP-STM32
- Мобильное приложение: https://github.com/ES-10-01/ISP-Client

Также архивы с исходными кодами приложены к отчёту в дополнительных файлах.

Приложение 2. Использованная документация

1. https://www.st.com/en/microcontrollers-microprocessors/stm32l152rc.html
2. https://docs.spring.io/spring-integration/reference/html/ip.html