МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет вычислительной техники

Кафедра «Вычислительная техника»

Специальность: 09.05.01 «Применение и эксплуатация автоматизированных систем специального назначения»

Специализация: «Эксплуатация вычислительных машин, комплексов, систем и сетей специального назначения»

ДИПЛОМНЫЙ ПРОЕКТ

на тему

СИСТЕМА МОНИТОРИНГА ПОКАЗАНИЙ СЧЕТЧИКОВ РЕСУРСОВ

Студент _____ 20.06.2023 Лучинкин Глеб Юрьевич
(подпись, дата) (ФИО полностью)

Руководитель _____ 22.06.23 Дубинин В.Н.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр _____ 22.06.23 Бычков А.С.
(подпись, дата) (фамилия, инициалы)

Рецензент
(должность, место работы) _____ 20.06.23 Печерский С.В.
(подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол зас. кафедры от 2.06.2023 г. № 14)

Заведующий кафедрой _____ Митрохин М.А.
(подпись) (фамилия, инициалы)

Работа защищена с отметкой отлично

(протокол заседания ГЭК от 23.06.23 № 7)

Секретарь ГЭК _____ Гурин Е.И.
(подпись) (фамилия, инициалы)

Пенза 2023

Федеральное государственное бюджетное образовательное учреждение
высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет Вычислительной техники

Кафедра Вычислительная техника

«Утверждаю»
Заведующий кафедрой
_____ Митрохин М.А.
«26» апреля 2023 г.

ЗАДАНИЕ
НА ДИПЛОМНЫЙ ПРОЕКТ

1. Студент Лучинкин Глеб Юрьевич гр. 18ВОЭ1 факультета вычислительной техники.
2. Тема работы: «Система мониторинга показаний счетчиков ресурсов».
Тема утверждена приказом ПГУ № 458/о от 26.04.2023 г.
3. Руководитель работы д.т.н., профессор каф. ВТ «ПГУ» Дубинин В.Н.
4. Задание на работу (назначение разработки, исходные данные и т.п.).
Разработать приложение «Система мониторинга показаний счетчиков ресурсов», позволяющее автоматизировать ручной процесс отслежевания расходов коммунальных услуг. Приложение должно обладать графическим и интуитивно понятным интерфейсом, для взаимодействия с пользователем. Система должна предоставлять возможность для взаимодействия с информацией о зданиях и счётчиков ресурсов. Разрабатываемое приложение также должно выполнять задачи по формированию электронных таблиц.
5. Перечень подлежащих разработке вопросов.
Требуется произвести анализ предметных областей: изучить существующие системы, выделить их достоинства и недостатки. Выбрать язык программирования, который подойдет для создания графического интерфейса. Определить архитектуру приложения и вспомогательные средства разработки. Изучить функционал библиотек для формирования отчётов.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1.	Анализ предметной области	10	26.04.2023-28.04.2023	
2.	Анализ требований	10	29.04.2023-30.04.2023	
3.	Выбор инструментов для разработки программы	10	31.04.2023-03.05.2023	
4.	Реализация графического интерфейса	10	04.05.2023-08.05.2023	
5.	Реализация программы	20	09.05.2023-15.05.2023	
6.	Сборка приложения в контейнере	10	16.05.2023-23.05.2023	
7.	Оформление выпускной работы	30	24.05.2023-02.06.2023	

Дата выдачи задания «26» апреля 2023 года.

Руководитель дипломного проекта _____ 22.06.23 Дубинин В.Н.
(подпись, дата)

Задание к исполнению принял студент _____ Лучинкин Г.Ю.
(подпись, дата)

Работу к защите допустить
Декан факультета _____ Фионова Л.Р.
22.06.2023 *(подпись, дата)*

Реферат

Дипломный проект содержит 109 листов, 56 рисунков, 22 таблицы, 4 источника, 3 приложения

СИСТЕМА МОНИТОРИНГА ПОКАЗАНИЙ СЧЕТЧИКОВ РЕСУРСОВ, *DOCKER, CSV, REST, JAVA*

Цель работы - необходимо разработать систему, позволяющую отслеживать показания счетчиков ресурсов и составлять отчеты по полученным показания.

Объектом разработки является клиент-серверное приложение которое выводит выводит показания у счётчиков которые были установлены на определённой локации. На основе этих показаний формируются отчёты, некоторые из них можно выгружать в формате *csv*. Система должна обладать web интерфейсом удобным для пользователя.

Разработка велась в средах программирования *Intelij IDEA* и *VisualStudio Code* на языках программирования *Java* и *JavaScript*.

В результате проделанной работы было разработано приложение, удовлетворяющее поставленной задаче, а именно: удобный пользовательский интерфейс, возможность устанавливать счётчики на локации, просматривать их показания и составлять отчёты.

Созданная система может быть внедрена на предприятие, на которых часто используются коммунальные услуги (вода, электричество) с целью оптимизации и предотвращения переизбытка использования ресурсов.

Содержание

Перечень сокращений и обозначений.....5
Введение.....6
1 Система мониторинга показаний счетчиков ресурсов.....8
1.1 Предпроектный анализ предметной области.....8
1.1.1 Анализ предметной области.....8
1.2.1 Техническое задание.....9
1.2.1.1 Основание для разработки.....9
1.2.1.2 Назначение разработки.....9
1.2.1.3 Требование к программе.....10
1.2.1.4 Требования к надежности.....11
1.3.1 Выбор стека технологий и средств разработки.....11
1.3.1.1 Архитектура приложения.....11
1.3.1.2 Выбор способа взаимодействия компонентов системы.....13
1.3.1.3 Выбор языков программирования для создания системы.....15
2 Технология реализации системы мониторинга показателей счетчиков ресурсов.....18
2.1 Реализация серверной части приложения.....18
2.1.1 Разработка микросервиса *main_server*.....18
2.1.2 Разработка микросервиса *meter_emulator*.....31
2.1.3 Разработка микросервиса *statistic_collector*.....36
2.2 Реализация клиентской части приложения.....40
2.3 Инсталяция средств запуска системы.....54
2.3.1 Установка *Docker*.....54
2.3.2 Запуск системы мониторинга показателей счетчиков ресурсов.....55
3 Безопасность жизнедеятельности. Основные производственные опасные и вредные факторы, воздействующие на программиста.....57
4 Экономическое обоснование разработки.....59
Заключение.....63

Список использованных источников.....64
Приложение А – UML-диаграммы.....65
Приложение А.1 – Диаграмма вариантов использования.....66
Приложение А.2 – Диаграмма последовательности.....67
Приложение Б – Листинг программы.....68
Приложение В – Презентация..... 101

Перечень сокращений и обозначений

В настоящей работе применяют следующие сокращения и обозначения

БД – база данных

ПО – программное обеспечение

ПЭВМ – персональная электронно-вычислительная машина

СУБД – система управления базами данных

API (Application Programming Interface) – программный интерфейс приложения

REST (Representational State Transfer) – архитектурный стиль построения программного интерфейса

JS (Java Script) - язык программирования

UML (Unified Modeling Language) – унифицированный язык моделирования

IoT (Internet of Things) - интернет вещи

UI (User Interface) - пользовательский интерфейс

Введение

В современном мире все стремится к оптимизации производственных и бытовых процессов. Сейчас не составляет труда расплатиться за бензин не подходя к кассе, а используя мобильное приложение, либо сделать покупку через огромное количество маркетплейсов находящихся в свободном доступе в интернете.

С точки зрения любой организации, то их конечной целью является оптимизация своих бизнес-процессов и увеличение эффективности работы. Оптимизация бизнес-процессов - это процесс улучшения и ускорения рабочих процессов в компании. Цель оптимизации бизнес-процессов заключается в повышении эффективности работы бизнеса, уменьшении затрат на производство и услуги, а также в улучшении качества продукции и удовлетворенности клиентов. Чтобы оптимизировать бизнес-процессы, необходимо провести анализ текущих рабочих потоков и выявить проблемные моменты. Затем следует разработать план действий для устранения этих проблем и внедрить новые методы и инструменты, которые помогут ускорить и упростить работу. Оптимизация бизнес-процессов может быть осуществлена с помощью автоматизации некоторых задач, перераспределения обязанностей между сотрудниками, использования новых технологий и методов, а также обучения сотрудников новым навыкам и усовершенствованию системы управления.

В этом контексте системы мониторинга показаний счетчиков ресурсов играют важную роль, позволяя оперативно отслеживать потребление энергоносителей и воды, а также контролировать расходы на эти ресурсы.

Целью данного дипломного проекта является разработка системы мониторинга показаний счетчиков ресурсов, с помощью которой жители квартир смогут:

- просматривать показания счетчиков;
- составлять отчетность по количеству затраченных ресурсов;
- анализировать расход ресурсов.

Дипломный проект системы мониторинга показаний счетчиков ресурсов основан на основанной на использовании современных технологий в области IoT (Internet of Things), но представленные в виде программной реализации базирующейся на микросервисной архитектуре:

- пользовательским интерфейсом является web приложение (сайт), на которое заходит пользователь, и на котором позволяется заводить здания в которых установленны счетчики, прикреплять счетчики к зданию или квартире, включать счетчики и просматривать отчетность о счетчике,

- главный сервис с которым общается web клиент, сервис эмуляции работы счетчиков и сервис составления отчетности написанны на языке программирования Java с использованием фрейморков Spring и Spring Boot,

- в качестве хранилища информации была выбрана реляционная СУБД PostgreSQL.

Будут выявлены требования к системе, проанализированы архитектуры и технологии, используемые при ее разработке, представлены результаты тестирования и экспериментов, а так же будет представлено экономическое обоснование проекта и инструкция по безопасности жизнедеятельности пользователя компьютера.

1 Система мониторинга показаний счетчиков ресурсов

1.1 Предпроектный анализ предметной области

1.1.1 Анализ предметной области

Система мониторинга показаний счётчиков ресурсов является важной частью управления потреблением ресурсов, таких как электроэнергия, газ, вода и т.д. Она позволяет отслеживать и анализировать показания счётчиков, контролировать потребление ресурсов и оптимизировать расходы.

Основной задачей системы мониторинга является сбор и обработка данных о потребляемых ресурсах. Для этого используются различные типы счётчиков, которые передают данные в центральную систему мониторинга через специальные интерфейсы.

Центральная система мониторинга осуществляет сбор данных, их анализ и хранение в базе данных. Пользователи имеют доступ к данным через веб-интерфейс.

В ходе анализа предметной области нужно рассмотреть аналоги системы мониторинга состояния счетчиков ресурсов и выяснить какие у них имеются достоинства и недостатки. Плюсы и минусы аналогов приведены в таблице 1.1.

Таблица 1.1 – Сравнительный анализ аналогов реализуемой программы

Наименование	Плюс решения	Минус решения
Smart Metering	- уменьшение вероятности ошибок при расчете стоимости; - технологическое сопровождение; - сокращение расходов.	- необходимость замены счетчиков; - возможность нарушения конфиденциальности; - учитывает расход только энергоресурсов.
Ecois	- сокращение расходов на энергопотребление; - автоматическое управление системами отопления.	- высокая стоимость установки и обслуживания; - учитывает расход только энергоресурсов.

Сравнение существующих систем мониторинга показаний счетчиков ресурсов показало, что они в основном нацелены на мониторинг расходов энергоресурсов.

Идеей дипломного проекта является разработка системы которая учитывает расход различных ресурсов (горячей и холодной воды, электричества). В результате клиенты получат универсальную систему мониторинга показаний счетчиков ресурсов.

Основные преимущества системы мониторинга показаний счётчиков ресурсов:

- Контроль за потребляемыми ресурсами;
- Оптимизация расходов на ресурсы;
- Идентификация возможных проблем и неисправностей;
- Удалённый доступ к данным о потреблении;
- Автоматическое уведомление о превышении лимитов потребления;
- Учет различных видов ресурсов.

В целом «Система мониторинга показателей счетчиков ресурсов» - это важный инструмент для управления расходами и повышения эффективности использования ресурсовявляется составной функцией и включает в себя возможность добавить или удалить детали для ремонта.

1.2.1 Техническое задание

1.2.1.1 Основание для разработки

Основанием для разработки является задание на дипломное проектирование.

1.2.1.2 Назначение разработки

Web-приложение «Система мониторинга показаний счётчика ресурсов» предназначено для автоматизации мониторинга за показателями счётчиков установленных в здании, а именно съем показателей, получение отчётности по счётчику, заведение счётчиков в системе.

1.2.1.3 Требование к программе

Система должна выполнять следующие функции:

- Создание объекта в котором установлены счётчики;
- Создание счётчика прикреплённого к объекту;
- Создание типов счётчиков;
- Запуск и остановка счётчиков;
- Получение суммы по тарифу за определённый период;
- Получение данных о частоте снятия показаний за период;
- Получение расчётных параметров за определённый период;
- Построение графика расходов за период;
- Составление отчёта по объекту;
- Составление отчётов по счётчику.

Разрабатываемая система должна формировать следующие отчеты:

- Отчёт по объекту;
- Отчёт по счётчику за текущие сутки;
- Отчет за весь период в формате *csv*;
- Отчет за весь период в *PDF* формате.

Система должна обеспечивать ввод, хранение и отображение данных.

Входными данными являются:

- Информация об объекте;
- Информация о счетчике;
- Период дат за который формируются данные о сумме, и показаниях.

Выходными данными являются:

- Сведения об объекте;
- Сведения о счетчике;
- Отчет по счетчику за текущие сутки;
- Отчет за весь период в форматах *CSV* и *PDF*;
- Информация о сумме за период;
- Информация о частоте снятия показаний за период;

- Получение расчетных параметров за определенный период;
- График расходов за период.

1.2.1.4 Требования к надежности

Система должна обеспечивать проверку валидности входящих данных. В случае ввода некорректных данных сообщать об этом, запрещать выполнять определенные функции если не заполнены все входные данные.

Пользователь не должен иметь доступа к исходным файлам приложения, чтобы не менять его исходную программную часть.

1.3.1 Выбор стека технологий и средств разработки

1.3.1.1 Архитектура приложения

В системе используется клиент - серверная архитектура (Рисунок 1.1)

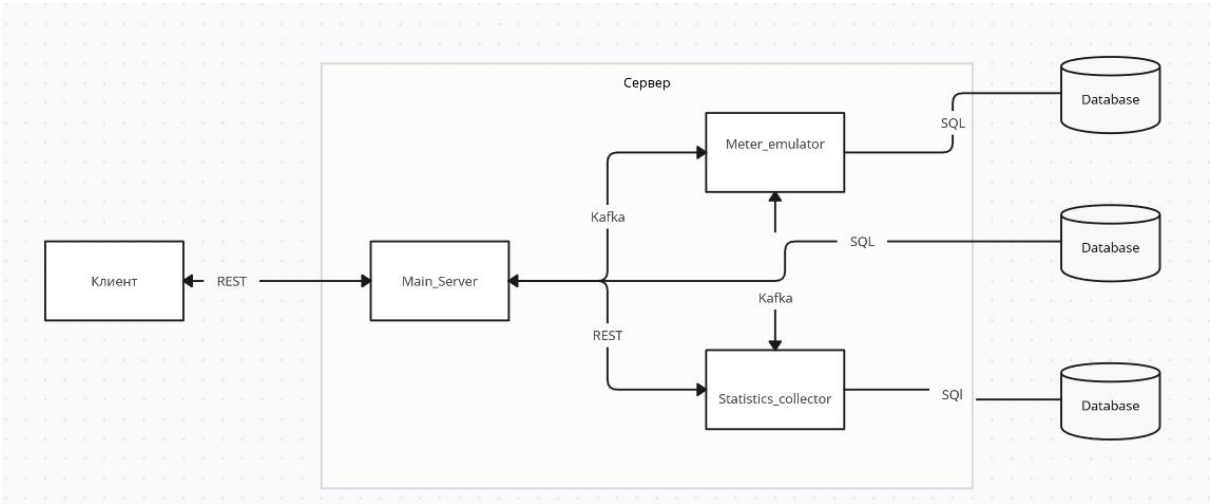


Рисунок 1.1 - Архитектура системы

Клиент-серверная архитектура - это архитектурный подход, который предполагает разделение приложения на две основных состовляющих: клиентскую и серверную. Клиентская часть отвечает за пользовательский интерфейс и взаимодействие с пользователем, а серверная часть обеспечивает обработку данных и бизнес-логику.

Клиенты и серверы могут быть реализованы на различных уровнях

абстракции, например, на уровне операционной системы, на уровне сети или на уровне приложения.

Клиенты и серверы могут работать в разных моделях взаимодействия, таких как клиент-сервер, *peer-to-peer*, *publish-subscribe* и другие.

Главное преимущество клиент-серверной архитектуры заключается в том, что она позволяет обеспечить масштабируемость, надежность и безопасность приложения. Клиенты могут быть легко добавлены или удалены, а серверы могут быть масштабированы горизонтально или вертикально, чтобы удовлетворить потребности приложения.

Кроме того, клиент-серверная архитектура облегчает разработку и поддержку приложения, поскольку она позволяет разделить обязанности между клиентом и сервером, что упрощает процесс тестирования, отладки и оптимизации приложения.

Так же в разрабатываемой системе сервер - это набор микросервисов, такой подход к проектированию систем называется микросервисной архитектурой.

Микросервисная архитектура основывается на создании небольших изолированных компонентов (микросервисов), которые взаимодействуют друг с другом для выполнения конкретных задач. Каждый микросервис работает как отдельное приложение, со своей собственной базой данных и интерфейсом программирования приложений (*API*).

Один из главных принципов микросервисной архитектуры заключается в том, что каждый микросервис должен выполнять только одну функцию. Например, может быть создан микросервис для обработки платежей, другой - для отправки уведомлений, третий - для хранения данных пользователей и так далее.

Такой подход имеет несколько преимуществ. Во-первых, микросервисы обычно проще разрабатывать и поддерживать, чем монолитные приложения. Во-вторых, благодаря изоляции каждого микросервиса изменения в одном из них не повлияют на работу остальной системы. В-третьих, микросервисы

могут распределяться по разным серверам, что позволяет создавать гибкие и масштабируемые системы.

Серверная часть приложения состоит из трех микросервисов:

- *Main_Server* - это основная точка доступа клиента к приложению. Через него происходит взаимодействие с всеми остальными микросервисами, являясь в своем роде неким proxy между клиентом и другими микросервисами. Так же к нему подключена база данных. Используется СУБД *PostgreSQL*.

- *Meter_emulator* - этот микросервис отвечает за эмуляцию работы счетчиков устанавливаемые на объекте через *Main_Server* по средствам *REST* запросов. *Main_Server* и *Meter_Emulator* общаются с помощью брокера сообщений *Kafka*. Так же к нему подключена база данных. Используется СУБД *PostgreSQL*.

- *Statistics_collector* отвечает за сбор и обработку статистики о работе счетчиков. Через него происходит составление всей отчетности, графиков, информации по роботе счетчиков. Связан с *Main_Server* по средствам *REST* запросов, а с *Meter_emulator* при помощи брокера сообщений *Kafka*. Так же к нему подключена база данных. Используется СУБД *PostgreSQL*.

1.3.1.2 Выбор способа взаимодействия компонентов системы.

В разрабатываемой системе клиент и сервер общаются с помощью REST запросов.

REST (*Representational State Transfer*) — это архитектурный стиль, основанный на использовании протокола HTTP для передачи данных между клиентом и сервером.

Основными принципами *REST* являются:

- Использование *HTTP*-методов (*GET*, *POST*, *PUT*, *DELETE* и т.д.) для работы с ресурсами;

- Разделение ресурсов на уникальные идентификаторы *URI* (*Uniform Resource Identifier*);

- Использование гипермедиа (например, ссылок) для связывания ресурсов между собой;

- Отсутствие состояния на сервере: каждый запрос должен содержать всю необходимую информацию для его обработки.

RESTful API предоставляет доступ к ресурсам через *HTTP*-запросы. *GET* используется для получения информации о ресурсе, *POST* для создания нового ресурса, *PUT* для обновления существующего ресурса, а *DELETE* - для удаления ресурса.

REST API обычно возвращают данные в формате *JSON* или *XML*. Эти форматы позволяют легко разбирать ответы сервера и использовать извлеченную информацию в приложении.

REST является очень популярным среди разработчиков веб-приложений, так как он предоставляет простой и эффективный способ обмена данными между клиентом и сервером.

Для взаимодействия микросервисов на стороне сервера была выбрана *Kafka*.

Kafka - это распределенная система, предназначенная для обработки потоковых данных. Она используется для передачи сообщений между различными компонентами системы, в том числе микросервисами.

Взаимодействие микросервисов с помощью *Kafka* может происходить следующим образом: один микросервис создает сообщение и отправляет его в *Kafka,* а другой микросервис получает сообщение из *Kafka* и обрабатывает его.

Для этого каждый микросервис должен быть подписан на соответствующую тему в *Kafka*. Тема - это категория, к которой относятся сообщения, а подписка на тему позволяет микросервисам получать сообщения, опубликованные в этой теме.

Также возможно использование *Kafka* в качестве буфера между микросервисами, что позволяет одновременно уменьшить нагрузку на каждый сервис и гарантировать доставку сообщений в порядке их отправки.

Кроме того, *Kafka* позволяет хранить сообщения в течение определенного времени, что может быть полезно при перезапуске микросервисов или в случае временных проблем с доступностью к сервису.

Использование *Kafka* в взаимодействии микросервисов обеспечивает гибкость, масштабируемость и надежность системы.

1.3.1.3 Выбор языков программирования для создания системы

Для серверной части приложения был выбран язык программирования *Java*. *Java* - это объектно-ориентированный язык программирования позволяющий создавать кросплатформенные приложения.

Преимуществами *Java* в разработке *backend* приложений является:

Надежность: *Java* обеспечивает высокий уровень надежности, что особенно важно для *backend*-приложений, которые должны работать непрерывно и обрабатывать большое количество запросов.

Безопасность: *Java* имеет множество встроенных функций безопасности, обеспечивающих защиту от различных видов атак, таких как инъекции *SQL*, переполнение буфера и другие.

Масштабируемость: *Java* обладает хорошей масштабируемостью, позволяя создавать *backend*-приложения, которые могут обрабатывать большое количество запросов одновременно.

Переносимость: *Java*-код может выполняться на любой платформе, что делает его идеальным выбором для разработки кроссплатформенных backend-приложений.

Богатый набор библиотек и фреймворков: *Java* имеет огромное сообщество разработчиков, которые создают и поддерживают библиотеки и фреймворки, упрощающие разработку *backend*-приложений.

Для создания клиентской части был выбран язык программирования *JavaScript*.

JavaScript является одним из основных языков программирования, используемых для создания *frontend* приложений. Это связано с тем, что *JavaScript* поддерживается всеми современными веб-браузерами и может выполняться непосредственно на стороне клиента без необходимости отправки запросов на сервер.

JavaScript также имеет широкий выбор библиотек и фреймворков, которые облегчают создание интерактивных страниц и упрощают работу с различными элементами интерфейса, такими как кнопки, формы и таблицы.

Большинство современных фреймворков и библиотек, таких как *React*, *Angular* и *Vue*, написаны на *JavaScript*.

Более того, *JavaScript* имеет динамическую типизацию и поддерживает функциональное программирование, что делает его гибким и позволяет быстро прототипировать итерфейсы и легко менять их в процессе разработки. В целом, *JavaScript* - это мощный инструмент для создания интерактивных веб-приложений, который продолжает развиваться и улучшаться со временем.

1.3.4 Выбор фреймворков для создания системы

Spring Boot - это фреймворк для создания приложений на *Java*, который является частью *Spring Framework*. Он предоставляет простой и быстрый способ создания самостоятельных приложений, которые могут запускаться на любой платформе с установленной *JVM*.

Micronaut - это современный фреймворк, предназначенный для разработки микросервисов и серверных приложений на *Java*, *Kotlin* и *Groovy*. Он имеет простой *API* и высокую производительность.

Play Framework - это фреймворк для создания веб-приложений на *Java* и *Scala*. Он использует асинхронную модель и имеет мощную систему маршрутизации.

React используется для создания масштабируемых и быстрых пользовательских интерфейсов. *React* предоставляет компонентную модель, которая позволяет разрабатывать приложения как набор независимых компонентов, каждый из которых имеет свой собственный стиль и состояние.

Angular - это фреймворк для разработки масштабируемых приложений. Он предоставляет множество инструментов и функций, таких как двустороннее связывание данных, инъекция зависимостей, маршрутизация и многое другое.

Vue.js - это относительно новый фреймворк, для создания пользовательских интерфейсов. *Vue.js* построен на основе компонентной модели, подобной *React*, но имеет более лаконичный синтаксис и требует меньше дополнительных зависимостей.

Для реализации серверной части приложения был *Spring Boot*, так как он самый популярный и он поддерживает наибольшее количество сторонних библиотек. А для реализации клиентской части был выбран фреймворк *React*, так как он самый популярный и позволяет использовать декларативный подход при создании интерфейсов, что позволяет разработчику только определить как должен выглядеть *UI* (*User interface*) в определенном состоянии, а не заботиться об изменении этого состояния.

2. Технология реализации системы мониторинга показателей счетчиков ресурсов

2.1 Реализация серверной части приложения

Как уже было сказанно в главе 1 в пункте 1.3.1.1 «Архитектура приложения» серверная часть приложения состоит из трех микросервисов (*main_server*, *meter_emulator*, *statistic_collector*). В последующих пунктах будет описано как разрабатывался каждый из них.

2.1.1 Разработка микросервиса *main_server*

Для начала разработки микросервиса нужно определиться с тем, какие сущности будут храниться в базе данных. Для этого опишем логическую модель базы данных.

Таблица 2.1 - Атрибуты сущности «Локация»

Название атрибута	Тип	Обязательный
Идентификатор	Целый	Да
Город	Строковый	Да
Улица	Строковый	Да
Дом	Строковый	Да
Квартира	Строковый	Да

Таблица 2.2 - Атрибуты сущности «Тип счетчика»

Название атрибута	Тип	Обязательный
Идентификатор	Целый	Да
Название	Строковый	Да

Таблица 2.3 - Атрибуты сущности «Счетчик»

Название атрибута	Тип	Обязательный
Идентификатор	Строковый	Да
Начальное показание	Дробный	Да
Серийный номер	Строковый	Да
Дата установки	Временная метка	Да
Активность	Логическое значение	Да
Локация	Целый	Да
Тип счетчика	Целый	Да

Таблица 2.4 - Атрибуты сущности «Показания»

Название атрибута	Тип	Обязательный
Идентификатор	Целый	Да
Счетчик	Целый	Да
Дата снятия показания	Временная метка	Да
Дата сохранения записи	Временная метка	Да
Значение	Дробный	Да
Сырые данные	Строковый	Да

После проектирования логической модели базы данных нам стало известно какие сущности в ней будут храниться.

Далее, на основе логичечкой модели базы данных можно спроектировать физическую модель.

Таблица 2.5 - Поля таблицы «*locations*»

Название поля	Тип	Описание
id	*int*	Первичный ключ, обязательный
city	*varchar(255)*	Обязательный
street	*varchar(255)*	Обязательный
house	*varchar(255)*	Обязательный
flat	*varchar(255)*	Обязательный

Таблица 2.6 - Поля таблицы «*meter_types*»

Название поля	Тип	Описание
id	*int*	Первичный ключ, обязательный
title	*varchar(255)*	Обязательный

Таблица 2.7 - Поля таблицы «*meters*»

Название поля	Тип	Описание
id	*UUID*	Первичный ключ, обязательный
initial_readings	*float*	Обязательный
serial_number	*varchar(255)*	Обязательный
installation_time	*timestamp*	Обязательный
is_active	*bool*	Обязательный
location_id	*id*	Внешний ключ, обязательный
type_id	*id*	Внешний ключ, обязательный

Таблица 2.8 - Поля таблицы «*readings*»

Название поля	Тип	Описание
id	*int*	Первичный ключ, обязательный
meter_id	*UUID*	Внешний ключ, обязательный
read_time	*timestamp*	Обязательный
insert_time	*timstamp*	Обязательный
value	*float*	Обязательный
raw_data	*varchar(255)*	Обязательный

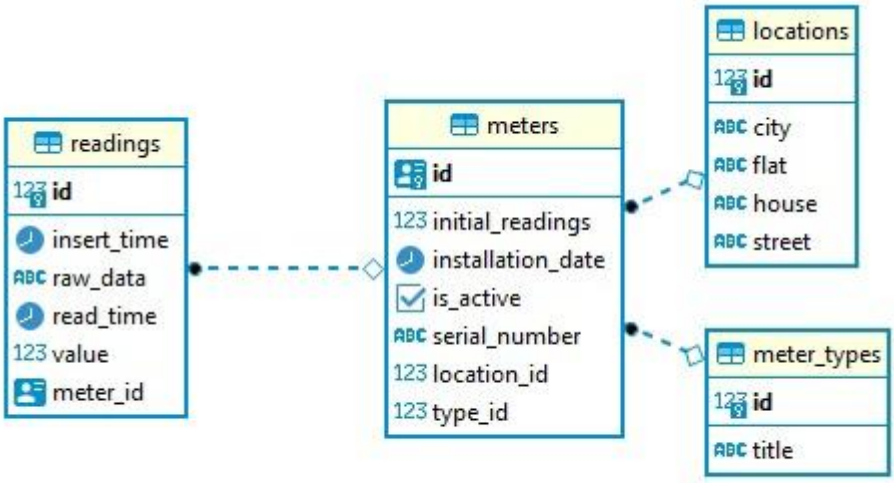


Рисунок 2.1 - Физическая модель базы данных сервиса *main_server*

После проектирования физической модели можно приступать к написанию кода.

Разработка любого приложения с использованием фрейморка *Spring Boot* начинается с создания главного класса, в котором определен метод *main*, запускающий приложение *public static void main(String[] args) {SpringApplication.run(MainServer.class, args);}* (Рисунок 2.2)

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.6.2)

2023-06-14 20:13:17.022  INFO 8880 --- [           main] com.example.kafkameters.MainServer
2023-06-14 20:13:17.026 DEBUG 8880 --- [           main] com.example.kafkameters.MainServer
2023-06-14 20:13:17.027  INFO 8880 --- [           main] com.example.kafkameters.MainServer
2023-06-14 20:13:17.992  INFO 8880 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate
2023-06-14 20:13:18.076  INFO 8880 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate
2023-06-14 20:13:18.962  INFO 8880 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
2023-06-14 20:13:18.974  INFO 8880 --- [           main] o.apache.catalina.core.StandardService
```

Рисунок 2.2 - Запуск приложения *Spring Boot*

Далее нужно описать классы сущностей. В *Spring Boot* это делается с помощью анотации *@Entity* и *@Table*. С помощью этих аннотаций *Spting Boot* понимает к какому аспекту относится данный класс и встраивает его в контекст приложения.

Первым создается класс *Location* - это шаблон здания в котором впоследствии будут зарегистрированы счетчики. Классы в языке программирования *Java* создаются с помощью конструкции *public class <ClassName>*. Чтобы создать класс *Location* нужо написать *public class Location*. Далее идет описание полей таблицы *Locations*. Чтобы понять что поля класса относятся к полям таблицы нужно указать аннотацию *@Column()* и в скобках указать параметр *name*. Получится *@Column(name = "city")*. Для идентификации того, что поле класса является идентификатором таблицы используется аннотация *@Id*. Так же можно указать стратегию генерации значения идентификатора с помощью аннатоции *@GeneratedValue()* указав параметр *@GeneratedValue(strategy = GenerationType.IDENTITY)*. Для таблиц *meter_type*, *meters*, и *readings* тоже нужно создать классы, которые будут описывать их поля. Делается это по аналогии с классом *Location*, но стоит отметить момент описания связей полей таблицы. В *Spring Boot* для этого используются аннотация *@JoinColumn*, с указанием параметра *name*, которое обозначает к какому полю идет отношение. Получится *@JoinColumn(name = "type_id")*. Для указания какая связь используется между таблицами используют аннотации по типу *@OneToOne*. Эта аннотация указывает на то, что связь между таблицей meters и *meter_type* один к одному.

После описания всех сущностей нужно для каждой создать репозиторий, который будет связывать *Java* класс с сущностью в базе данных. Для этого в *Spring Boot* используется интерфейс под названием *JpaRepository<T, ID>*. Это встроенный интерфейс который в своей сигнатуре использует дженерики *T* и *ID*.

Для создания репозитория нужно создать собственный интерфейс, который будет наследовать *JpaRepository*, указав в дженерике класс сущности и тип *id*, *public interface LocationRepository extends JpaRepository<Location, Long>*. Взглянув на реализацию *JpaRepository* можно увидеть, что он содержит в себе описание методов для взаимодействия с базой данных (Рисунок 2.3).

```
@NoRepositoryBean
public interface JpaRepository<T, ID> extends PagingAndSortingRepository<T, ID>, QueryByExampleExecutor<T> {
    List<T> findAll();

    List<T> findAll(Sort sort);

    List<T> findAllById(Iterable<ID> ids);

    <S extends T> List<S> saveAll(Iterable<S> entities);

    void flush();

    <S extends T> S saveAndFlush(S entity);

    <S extends T> List<S> saveAllAndFlush(Iterable<S> entities);

    /** @deprecated */
    @Deprecated
    default void deleteInBatch(Iterable<T> entities) { this.deleteAllInBatch(entities); }

    void deleteAllInBatch(Iterable<T> entities);

    void deleteAllByIdInBatch(Iterable<ID> ids);

    void deleteAllInBatch();

    /** @deprecated */
    @Deprecated
    T getOne(ID id);

    T getById(ID id);

    <S extends T> List<S> findAll(Example<S> example);

    <S extends T> List<S> findAll(Example<S> example, Sort sort);
}
```

Рисунок 2.3 - Интерфейс *JpaRepository*

Через наследование интерфейса *JpaRepository* создаем репозитории для оставшихся сущностей. В результате получим четыре репозитория:

- *LocationRepository;*
- *MeterRepository;*
- *MeterTypeRepository;*
- *ReadingRepository.*

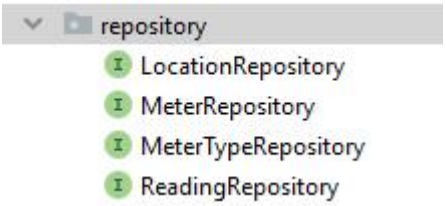


Рисунок 2.4 - Созданные репозитории

После создания репозиториев нужно создать сервисы для каждой сущности. Чтобы взаимодействие клиента и сервера было максимально безопасным, с точки зрения потери данных, нужно создать объекты передачи данных *DTO*. Они бывают двух типов *RequestDTO* и *ResponseDTO*. Через *RequestDTO* клиент передает информацию на сервер, а через *ResponseDTO* получает ответ от сервера. Для создания *RequestDTO* нужно создать класс и назвать его *LocationRqDto*. Это будет класс с описанием полей таблицы locations, но без id, так как он генерируется автоматически. В свою же очередь *ResponseDto* содержит в себе все поля из таблицы. Подобную пару объектов передачи данных нужно создать для всех объектов для которых в последствии будут написаны сервисы (Рисунок 2.5).

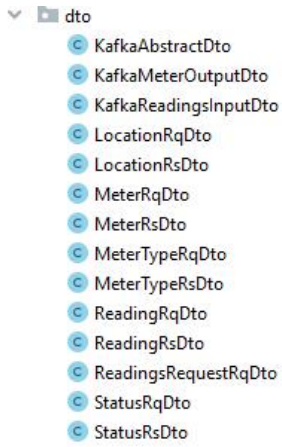


Рисунок 2.5 - Созданные *Data Transfer Objects*

После создания объектов передачи данных нужно сделать мапперы для них. Мапперы будут переводить входные объекты в сущность, а сущность в объекты ответа. Для создания мапперов используется библиотека *ModelMapper*. Перед тем как начать работу с ней нужно ее сконфигурировать. Для этого создадим класс *ModelMapperConfig* и пометим его аннотацией *@Configuration*. Внутри класса создадим метод *modelMapper()*. Внутри метода создаем объект этого маппера и устанавливаем ему конфигурацию (Рисунок 2.6)

```
@Bean
@Scope(SCOPE_PROTOTYPE)
public ModelMapper modelMapper() {
    ModelMapper mapper = new ModelMapper();
    mapper.getConfiguration()
            .setMatchingStrategy(MatchingStrategies.STRICT)
            .setFieldMatchingEnabled(true)
            .setSkipNullEnabled(true);
    return mapper;
}
```

Рисунок 2.6 - Конфигурация *ModelMapper*

После конфигурирования *ModelMapper* нужно создать класс который будет мапить конкретный объект. К примеру создадим класс *LocationMapper*, определим в нем два метода *toEntity* и *toDto* (Рисунок 2.7).

```
/**
 * Преобразователь из входного DTO в сущность
 *
 * @param locationRqDto входной DTO объекта установки
 * @return сущность объекта установки
 */
public Location toEntity(LocationRqDto locationRqDto) { return modelMapper.map(locationRqDto, Location.class); }

/**
 * Преобразователь из сущности в выходной DTO
 *
 * @param location сущность объекта установки
 * @return выходной DTO объекта установки
 */
public LocationRsDto toDto(Location location) { return modelMapper.map(location, LocationRsDto.class); }
```

Рисунок 2.7 - Методы для работы с маппером

Такие классы нужно сделать для всех объектов которые будут сохраняться в базу данных(Рисунок 2.8).

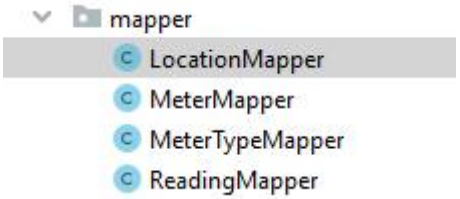


Рисунок 2.8 - Созданные мапперы

После создания всех мапперов и DTO можно приступить к созданию сервисов. Для этого нужно создать интерфейс и описать в нем методы которые впоследствии будут реализованы с точки зрения бизнес логики. К примеру создадим интерфейс *public interface LocationService{}* и опишем в нем методы общения с базой данных. Сигнатура каждого метода такая, что при необходимости в метод передается *RqDto*, а на выход отдается *RsDto*. В следствии этого метод создания записи в базе будет выглядеть так - *LocationRsDto create(LocationRqDto locationRqDto);*. Так же в сервисе описаны методы получения и удаления записей из базы данных:

- *LocationRsDto getById(Long id);*
- *Page<LocationRsDto> getAll(Pageable pageable);*
- *void deleteById(Long id)*.

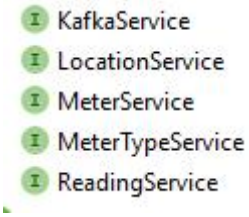


Рисунок 2.9 - Созданные интерфейсы сервисов

Для реализации бизнес логики нужно создать класс, который будет реализовывать созданный интерфейс *public class LocationServiceImpl implements LocationService*. В методе *create* происходит проверка на то, не был ли ранее создан *location* с таким же адресом, если в базе нашлась запись, то выбрасывается исключение оповещающее о том, что такой объект уже создан, *locationRepository.findByCityAndStreetAndHouseAndFlat (locationRqDto.getCity(), locationRqDto.getStreet(), locationRqDto.getHouse(), locationRqDto.getFlat()).ifPresent(location->{ throw new BadRequestException*

(LOCATION_ALREADY_EXISTS);});. Если исключение не было выбрашенно, то, с помощью маппера происходит преобразование входного *DTO* в сущность *Location locationSrc = locationMapper.toEntity(locationRqDto);* После преобразования *DTO* в сущность происходит сохранение записи в базу данных с помощью *locationRepository*, *Location locationDst = locationRepository.save(locationSrc)*; Затем происходит преобразование сохранённой записи в выходное *DTO* и происходит выход из метода return *locationMapper.toDto(locationDst)*;.

В методе *getById* происходит получение записи из базы данных с помощью *locationRepository*. Если запись не была найдена, то происходит выброс исключения, которое оповещает о том, что запись не была найдена *returnlocationRepository.findById(id).map(locationMapper::toDto).orElseThrow(() -> new NotFoundException(String.format(LOCATION_NOT_FOUND, id)));*.

В методе *getAll* происходит получение всех записей из базы данных *return locationRepository.findAll(pageable).map(locationMapper::toDto);*.

В методе *deleteById* происходит проверка на то, есть ли у здания активные счетчики, если есть, то происходит выброс исключения, которое оповещает о том, что к зданию прикреплены счетчики *throw new BadRequestException(LOCATION_HAS_ACTIVE_METERS);*. Если счетчиков нет, то происходит удаление записи из базы данных *locationRepository::delete.*

Подобные сервисы и их реализации нужно создать для каждой сущности (Рисунок 2.10).

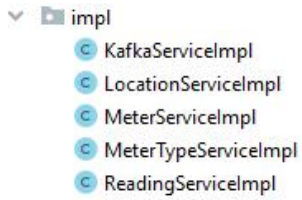


Рисунок 2.10 - Созданные сервисы с описанной бизнес логикой

Разрабатываемый микросервис общается с клиентской частью с помощью *REST API*. Поэтому дальнейший шаг - это создание *API*. Для этого создадим интерфейс *LocationAPI* и пометим его аннотацией *@RequestMapping*, указав путь к данному *API*. Получим *@RequestMapping("/locations")*. В интерфейсе нужно описать методы и прописать к ним пути. Напишем метод *ResponseEntity<LocationRsDto> create(@Valid @RequestBody LocationRqDto locationRqDto)*;. Метод создания записи в базе данных с точки зрения *REST API* является методом *POST*, поэтому метод *create* нужно пометить аннотацией *@PostMapping*. Методы *getById* и *getAll* с точки зрением *REST API* являются методами *GET*, поэтому нужно пометить его аннотацией *@GetMapping*. В случае с методом *getById* в аннотации *@GetMapping* нужно указать путь *@GetMapping("/{id}")*, где *id* это переменная передаваемая в метод. Метод *deleteById* с точки зрения *REST API* является методом *DELETE*, поэтому его нужно пометить аннотацией *@DeleteMapping*, и как в случае с методом *getById* указать *@DeleteMapping("/{id}")*, где *id* это переменная передаваемая в метод *deleteById*.

Подобные интерейфесы описывающие *API* нужно создать для каждого сервиса (Рисунок 2.11).

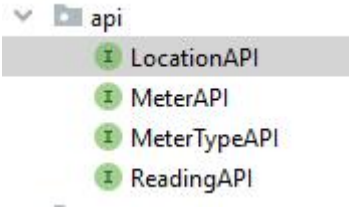


Рисунок 2.11 - Созданные интерфейсы описывающие *API*

Чтобы связать бизнес логику с *API* нужно написать класс контроллер, который будет реализовывать методы описанные в *API*. Для этого нужно создать класс *LocationController*, и реализовать в нем интерфейс *LocationAPI*. В методе *create* происходит вызов метода *create* у созданного ранее *locationService*. После чего происходит возврат результата выполнения метода *locationService.create(locationRqDto)*.

В методе *getById* происходит вызов метода *locationServie.getById()*. Затем

происходит возврат результата выполнения этого метода с помощью ключевого слова *return*.

В методе *getAll* происходит вызов метода *locationServie.getAll()*. Затем происходит возврат результата выполнения этого метода с помощью ключевого слова return.

В методе *deleteById* происходит выполнение метода *locationServie.deleteById()*. Так как этот метод ничего не возвращает, а клиент нужно оповестить о том, что запрос выполнился успешно, то происходит возвращение экземпляра класса *ResponseEntity*, у которого устанавливается статус *accepted return ResponseEntity.accepted().build()*;.

Подобные контроллеры нужно создать для каждого интерфейса, которые описывают *API* (Рисунок 2.12)

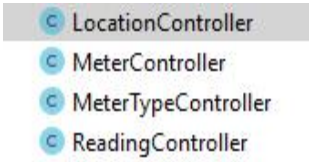


Рисунок 2.12 - Созданные контроллеры

Микросервисы входящие в состав сервера общаются друг с другом с помощью брокера сообщений *Kafka*. Для настройки *Kafka* в микросервисе *main_server* нужно создать два конфигурационных файла. Один для получателя (*Consumer*) сообщений, а другой для отправителя (*Producer*) сообщений.

Создадим класс *KafkaConsumerConfig* и пометим его аннотацией *@Configuration*. Определим в нем метод *consumerConfigs* и установим в нем настройки нужные для получателя сообщений. Настройки кладутся в структуру данных *Map<String, Object>*, где *String* - это ключ, а *Object* - значение. Поэтому с помощью метода *put* добавляем настройки, обозначаем на каком из наших локальных серверах висит *Kafka props.put(ConsumerConfig, kafkaServer)*, затем устанавливаем десериалайзеры для ключа *props.put(ConsumerConfig,LongDeserializer.class)* и значения

props.put(ConsumerConfig, StringDeserializer.class). Далее нужно обозначить к какой группе относится *Consumer props.put(ConsumerConfig, kafkaGroupId)*.

Похожим образом настраивается отправитель сообщений (Рисунок 2.13).

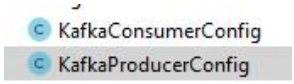


Рисунок 2.13 - Конфигурация брокера *Kafka*

Для того чтобы работать с *Kafka* нужно создать сервис наподобие тех, что были созданы для работы с сущностями базы данных. Был создан интерфейс *KafkaService*, в котором определили три метода:

- *meterCreated();*
- *meterStatusSwitched()*;
- *readingsReceived()*.

Далее был создан класс реализующий интерфейс *KafkaService*, *KafkaServiceImpl*. В нем были реализованы методы интерфейса. В методе meterCreated происходит отправка сообщения о том, что был создан новый счетчик, в методе *meterStatusSwitched* происходит отправка сообщения, содержащее информацию о том, что статус работы счетчика изменился. В свою же очередь метод readingsReceived является слушателем и помечен аннотацией *@KafkaListener*.

Этот метод принимает значения которые генерирует микросервис *meter_emulator* и сохраняет их в базу данных по средствам вызова метода сервиса *readingService.create(RqDto)*.

Таким образом сервис *main_server* считается разработанным (Рисунок 2.14).

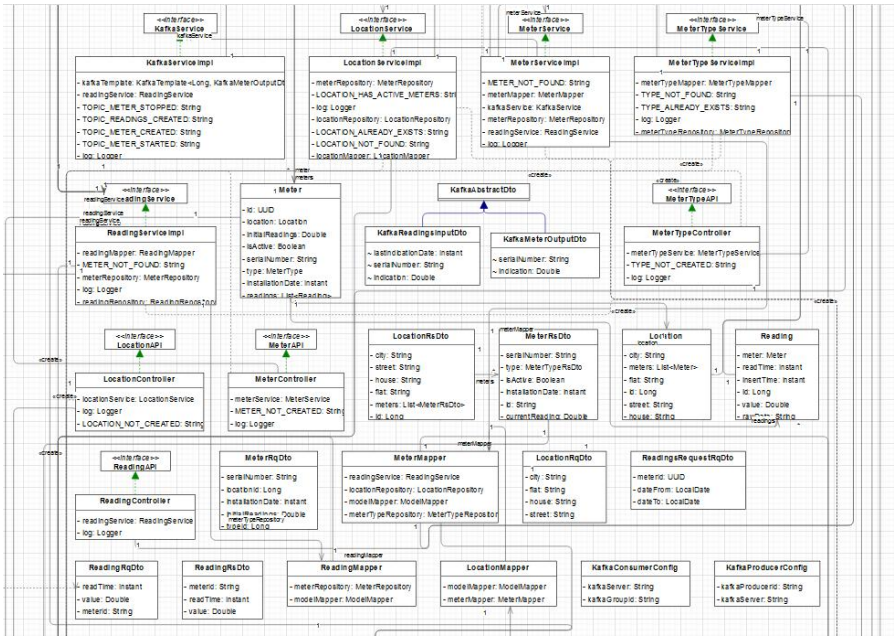


Рисунок 2.14 - Диаграмма классов сервиса main_server

2.1.2 Разработка микросервиса meter_emulator

Для начала разработки микросервиса нужно определиться с тем, какие сущности будут храниться в базе данных. Для этого опишем логическую модель базы данных.

Таблица 2.9 - Атрибуты сущности «Счетчик»

Название атрибута	Тип	Обязательный
Идентификатор	Целый	Да
Серийный номер	Строковый	Да
Показание	Дробный	Да
Статус счетчика	Логическое значение	Да
Дата последнего показания	Временная метка	Да

В микросервисе *meter_emulator* будет всего одна таблица, в которой будут храниться счетчики созданный в микросервисе *main_server*. На основе логической модели создадим физическую.

Таблица 2.10 - Поля таблицы «*Counter*»

Название поля	Тип	Описание
id	*int*	Первичный ключ, обязательный
serial_number	*varchar(255)*	Обязательный
indication	*float*	Обязательный
isActive	*bool*	Обязательный
last_indication_date	*timestamp*	Обязательный

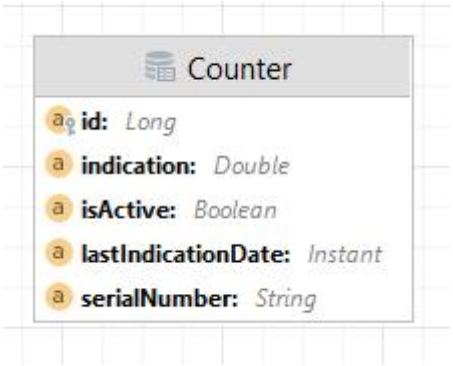


Рисунок 2.15 - Физическая модель базы данных сервиса *meter_emulator*

После проектирования физической модели можно приступать к написанию кода.

Разработка любого приложения с использованием фрейморка *Spring Boot* начинается с создания главного класса, в котором определен метод *main*, запускающий приложение *public static void main(String[] args) {SpringApplication.run(MeterEmulator.class, args);}*

Далее нужно создать классы сущностей. Как это делается подробно было описано в пункте 2.1.1 - создание микросервиса *main_server*. Тут будет тезисно описываться этот этап. Создаем класс *Counter* и помечаем его аннотацией *@Entity*. Далее нужно написать поля класса которые будут относиться к полям таблицы в базе данных. Поле отвечающее за идентификатор нужно пометить аннотацией *@Id*, все остальные поля пометить аннотацией *@Column* и указать у нее атрибут *name*. К примеру поле с аннотацией *@Column(name = "serial_number")* будет относиться к полю serial_number в базе данных.

После описания всех сущностей нужно для каждой создать репозиторий, который будет связывать *Java* класс с сущностью в базе данных. Для этого в *Spring Boot* используется интерфейс под названием *JpaRepository<T, ID>*. Чтобы создать репозиторий нужно создать интерефейс который будет наследовать интерфейс *JpaRepository*. Для этого нужно написать *public interface CounterRepository extends JpaRepository<Counter, Long>*.

Далее нужно описать *DTO* для сущности *Counter*. Для этого создается два класса *RqDto* и *RsDto*. Далее к этим объектам передачи данных и сущности нужно сделать маппер. Маппер делается по такому же принципу как и в пункте 2.1.1.

После создания *DTO* и маппера нужно реализовать сервис по работе с репозиторием. Для этого создается интерфейс под названием *CounterService*, в котором описаны метода для работы с репозиторием:

- *CounterRsDto create(CounterRqDto counterRqDto);*
- *Page<CounterRsDto> findAll(Pageable pageable);*
- *CounterRsDto findById(@NonNull Long id)*;
- *void deleteCounter(@NonNull Long id)*.

Затем нужно создать класс который будет иплементировать этот интерфейс и реализует в нем методы.

Так же в этотм микросервисе нужно настроить брокер сообщений *Kafka*. Создадим класс *KafkaConsumerConfig* и пометим его аннотацией *@Configuration*. Определим в нем метод *consumerConfigs* и установим в нем настройки нужные для получателя сообщений. Настройки кладутся в структуру данных *Map<String, Object>*, где *String* - это ключ, а *Object* - значение. Поэтому с помощью метода *put* добавляем настройки, обозначаем на каком сервере висит *Kafka props.put(ConsumerConfig, kafkaServer)*, затем устанавливаем десериалайзеры для ключа *props.put(ConsumerConfig, LongDeserializer.class)* и значения *props.put(ConsumerConfig, StringDeserializer.class)*. Далее нужно обозначить к какой группе относится *Consumer props.put(ConsumerConfig, kafkaGroupId)*.

Похожим образом настраивается отправитель сообщений (Рисунок 2.16)

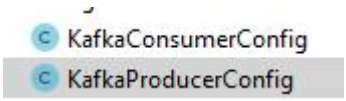


Рисунок 2.16 - Конфигурация брокера *Kafka*

Для того чтобы работать с *Kafka* нужно создать сервис наподобие тех, что были созданы для работы с сущностями базы данных. Был создан интерфейс *KafkaService*, в котором определим четыре метода:

- *void sendResultOfJob(Long id);*
- *void consumeCreatCounterMessage(CounterRqDto counterRqDto);*
- *void consumeStartCounterMessage(CounterRqDto counterRqDto);*
- *void consumeStopCounterMessage(CounterRqDto counterRqDto)*.

Метод *sendResultOfJob* отсылает сгенерированные показания с счетчика в топик, который прослушивает сервис *main_server*. Происходит это с помощью вызова метода *kafkaTemplate.send()*. Метод *consumeCreatCounterMessage* является слушателем сервиса *main_server*. Этот метод помечен аннотацией *@KafkaListener()*. Когда на стороне микросервиса *main_server* создается счетчик, то информация об этом отсылается в сервис *meter_emulator*. Метод *consumeCreatCounterMessage* принимает новый счетчик и сохраняет его к себе в базу данных в таблицу *counter*. Метод *consumeStartCounterMessage* является слушателем сервиса *main_server*. Этот метод помечен аннотацией *@KafkaListener*(). Когда на стороне микросервиса main_server происходит смена статуса работы счетчика на *true*, то информация об этом отсылается в сервис *meter_emulator*. Метод *consumeStartCounterMessage* принимает эту информацию, затем изменяет статус у счетчика и запускает его. Метод *consumeStopCounterMessage* является слушателем сервиса *main_server*. Этот метод помечен аннотацией *@KafkaListener*(). Когда на стороне микросервиса *main_server* происходит смена статуса работы счетчика на *false*, то информация об этом отсылается в сервис *meter_emulator*. Метод *consumeStartCounterMessage* принимает эту

информацию, затем изменяет статус у счетчика и останавливает его.

Для эмуляции работы счетчика была выбрана библиотека под названием *Quartz*. Эта библиотека позволяет создавать *jobs*, которые будут выполняться по расписанию, либо с через определённое количество времени. Для создания такой *job* потребуется создать класс *SheduledJob*, который будет реализовывать встроенный в библиотеку *Quartz* интерфейс *Job* (Рисунок 2.17).

```
public interface Job {
    3 implementations
    void execute(JobExecutionContext var1) throws JobExecutionException;
}
```

Рисунок 2.17 - Интерфейс *Job* библиотеки *Quartz*

В этом интерфейсе всего один метод *execute*(), он и будет запускать *job* которая генерирует новые значения у счетчика. В теле этого метода объявлен цикл *for*, внутри тела которого происходит запуск работы счетчика и отсылается только что сгенерированное показание в микросервисы *main_server* и *statistic_collector*.

Таким образом сервис *meter_emulator* считается разработанным (Рисунок 2.18)

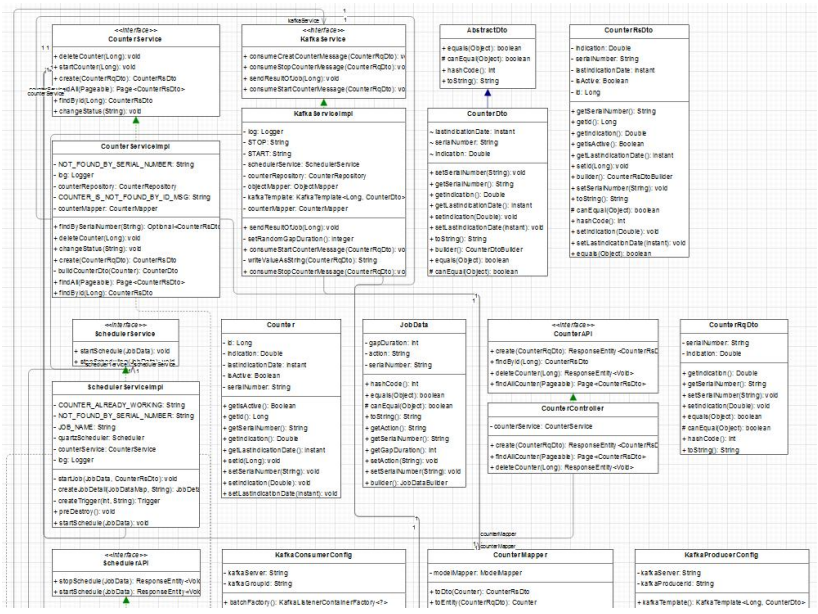


Рисунок 2.18 - Диаграмма классов микросервиса *meter_emulator*

2.1.3 Разработка микросервиса *statistic_collector*

Для начала разработки микросервиса нужно определиться с тем, какие сущности будут храниться в базе данных. Для этого опишем логическую модель базы данных.

Таблица 2.11 - Атрибуты сущности «Частота времени снятия»

Название атрибута	Тип	Обязательный
Идентификатор	Целый	Да
Периодичность	Целый	Да
Частота времени	Целый	Да

Таблица 2.12 - Атрибуты сущности «Тариф»

Название атрибута	Тип	Обязательный
Идентификатор	Целый	Да
Цена	Дробный	Да
Тип счетчика	Строковый	Да

Таблица 2.13 - Атрибуты сущности «Периодичность»

Название атрибута	Тип	Обязательный
Идентификатор	Целый	Да
Устройство	Целый	Да
Дата	Временная метка	Да
Среднее время снятия	Дробное	Да
Последнее показание	Целое	Да
Последнее время снятия	Временная метка	Да

Таким образом, на основе логической модели можно спроектировать физическую модель базы данных.

Таблица 2.14 - Поля таблицы «*delta_time*»

Название поля	Тип	Описание
id	*int*	Первичный ключ, обязательный
periodicity_id	*int*	Внешний ключ, обязательный
delta	*int*	Обязательный

Таблица 2.15 - Атрибуты сущности «*tariff*»

Название поля	Тип	Описание
id	*int*	Первичный ключ, обязательный
price	*float*	Обязательный
type	*varchar(255)*	Обязательный

Таблица 2.16 - Атрибуты сущности «periodicity»

Название поля	Тип	Описание
id	*int*	Первичный ключ, обязательный
device_id	*int*	Обязательный
date	*timestamp*	Обязательный
average	*float*	Обязательный
last_indicator	*int*	Обязательный
last_time	*timestamp*	Обязательный

Схема физической модели базы данных микросервиса *statistic_collector* представлена на рисунке 2.19

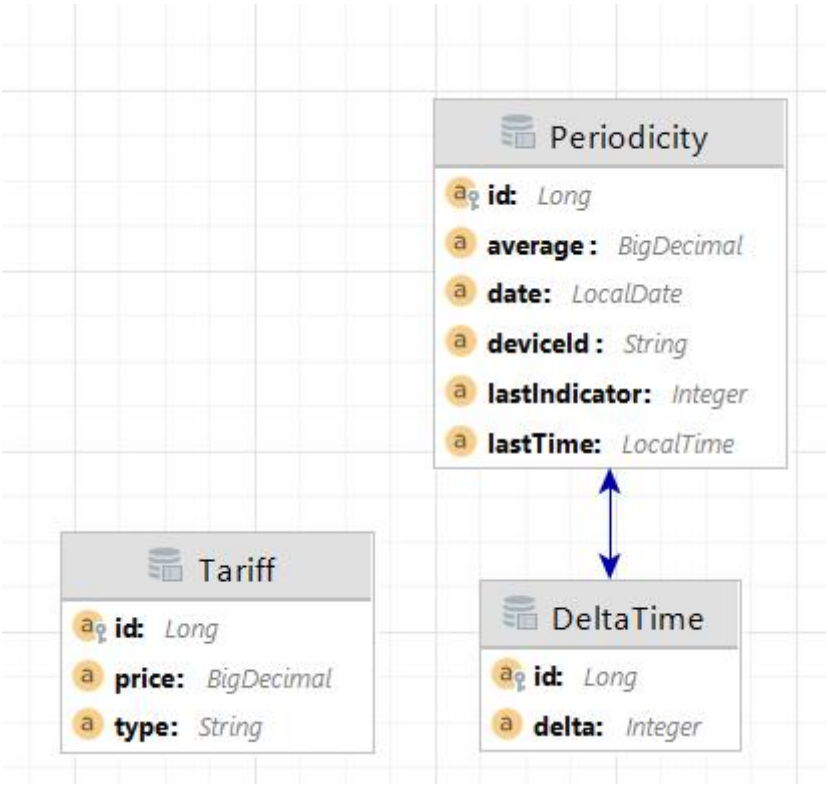


Рисунок 2.19 - Физическая модель базы данных микросервиса *statistic_collector*

После проектирования физической модели можно приступать к написанию кода.

Разработка любого приложения с использованием фрейморка *Spring Boot* начинается с создания главного класса, в котором определен метод *main*, запускающий приложение *public static void main(String[] args) {SpringApplication.run(StatisticCollector.class, args);}* Для этого микросервиса так же нужно описать сущности, сервисы, *DTO*, мапперы и репозитории. Как это делается было уже описано в других пунктах диплома, для этого микросервиса принцип такой же. Самой главное функцией этого сервиса является составление отчетов и генерация файлов с этими отчетами.

Для генерации отчетов в виде файлов и *html* страниц была использована библиотека *FreeMarker*. Чтобы начать работу с *FreeMarker* нужно его сконфигурировать. Для этого нужно создать класс, который будет реализовывать встроенный интерфейс *WebMvcConfigurer*. Этот класс нужно пометить аннотацией *@Configuration*. В конфиге нужно задать путь к

шаблонам на основе которых будут генерироваться отчеты *freeMarkerConfigurer.setTemplateLoaderPath(templateLoaderPath)*, и задать кодировку *freeMarkerConfigurer.setDefaultEncoding("UTF-8")*. Язык разметки на котором описываются шаблоны очень похож на *HTML* или на *XML* (Рисунок 2.20).

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <title>Информация об объекте установки</title>
</head>
<body>
<#if isExist!false>
    <h3>Город: ${city}</h3>
    <h3>Улица: ${street}</h3>
    <h3>Дом: ${house}</h3>
    <h3>Квартира: ${flat!"—"}</h3>
    <h3>Количество счётчиков: <#if meterCount = 0>счётчиков нет<#else> ${meterCount}</#if></h3>
    <#list meterList as meter>
        <h4>Серийный номер: ${meter.serialNumber} (идентификатор: ${meter.id})</h4>
        <h4>Тип: ${meter.type.title}, дата и время установки: ${meter.installationDate?datetime.iso}</h4>
        <h4>Статус счётчика: <#if meter.isActive>включен<#else>выключен</#if>, текущие
            показания: ${meter.currentReading}</h4>
    </#list>
<#else>
    <h2>Статус: ошибка</h2>
    <h3>Не найдена информация об объекте установки</h3>
</#if>
</body>
</html>
```

Рисунок 2.20 - Пример шаблона *freemarker*

После создания всех шаблонов нужно создать класс, который будет обрабатывать эти шаблоны. Для этого создаем класс с помощью конструкции *public class TemplateService*. В нем нужно создать два метода *processTemplate*. Один будет возвращать строку *HTML*, а другой будет возвращать файл. На вход каждому методу передается структура данных *Map<String, Object>*, где *String* - это ключ, а *Object* - значение. По ключу в шаблоне будут доставаться элементы которые будут заменять собой конструкции по тип *${street}*. После реализации этих методов можно сказать, что микросервис *statistic_collector* разработан (Рисунок 2.21).

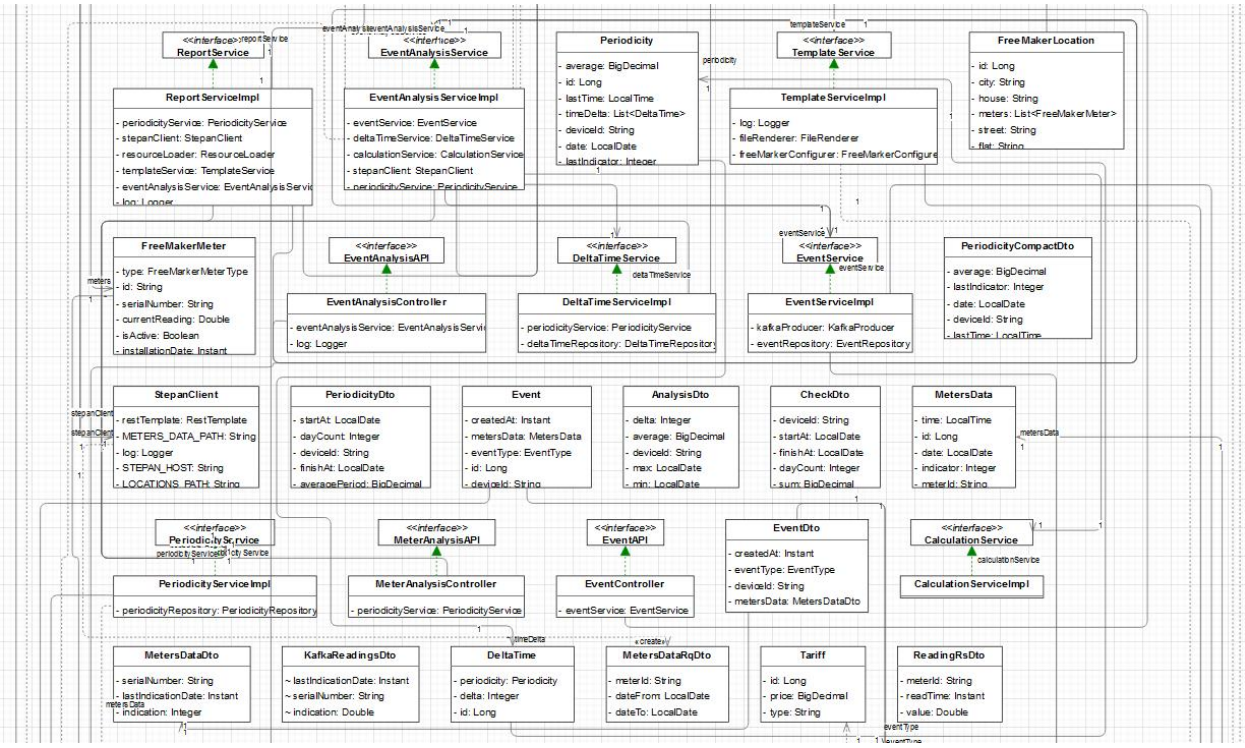


Рисунок 2.21 - Диаграмма классов микросервиса *statistic_collector*

2.2 Реализация клиентской части приложения

Разработка клиентской части велась в среде разработки *VisualStudio Code*, на языке программирования JavaScript, с использованием фреймворка React.

При запуске системы нас встречает главная страница с объектами - это здания в которых установленны счетчики ресурсов (Рисунок 2.22).

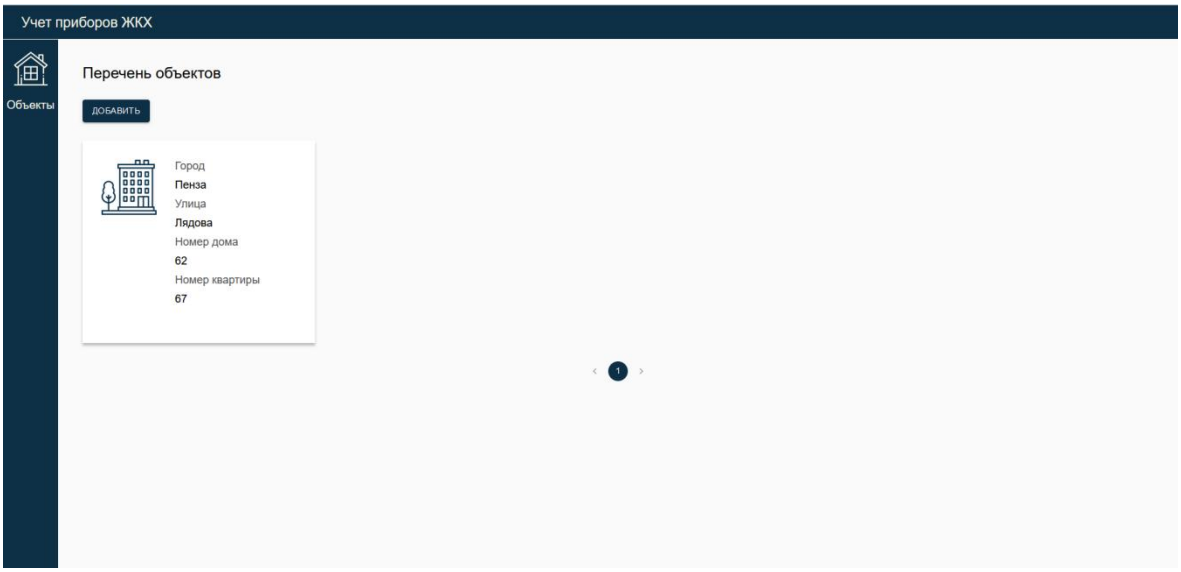


Рисунок 2.22 - Главная страница

При нажатии кнопки «Добавить» откроется окно в котором нужно заполнить информацию о объекте, а затем нажать кнопку «добавить» чтобы создать объект, или «отменить» чтобы вернуться на главную страницу. Если нажать кнопку «добавить» без заполнения полей, то произойдет ошибка и высветится сообщение о том, что нужно заполнить обязательные поля (Рисунок 2.23).

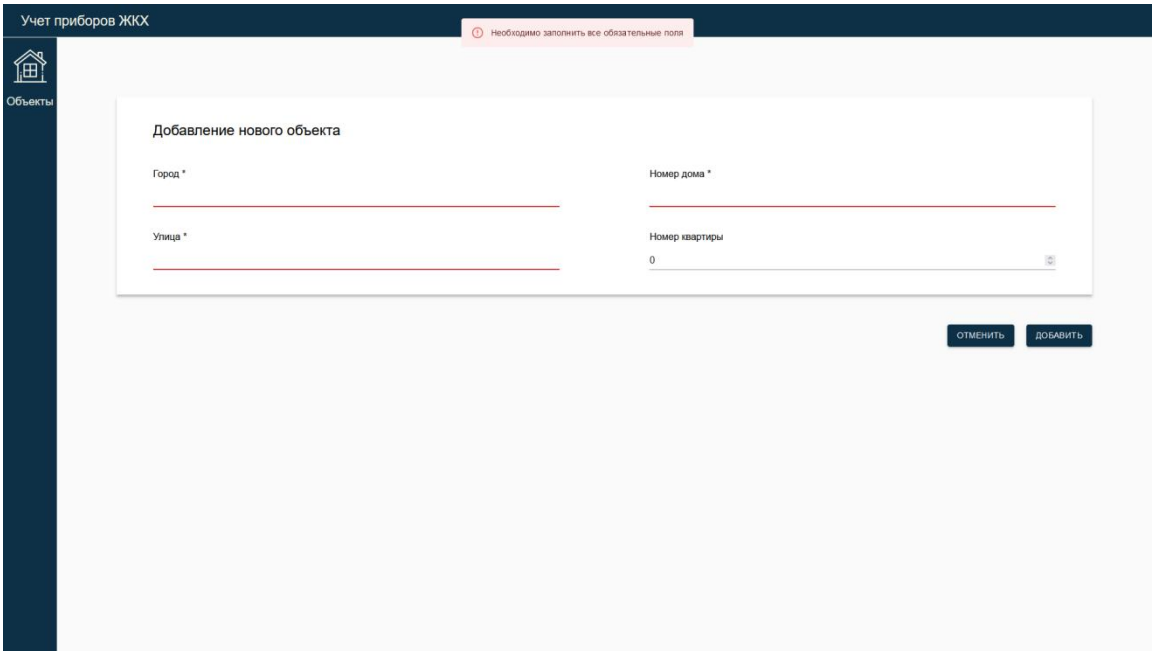


Рисунок 2.23 - Некорректное заполнение формы при добавлении объекта

Если все было заполнено верно (Рисунок 2.24), то при нажатии кнопки «добавить» мы увидим что на главном экране с объектами появилась еще одна панель (Рисунок 2.25).

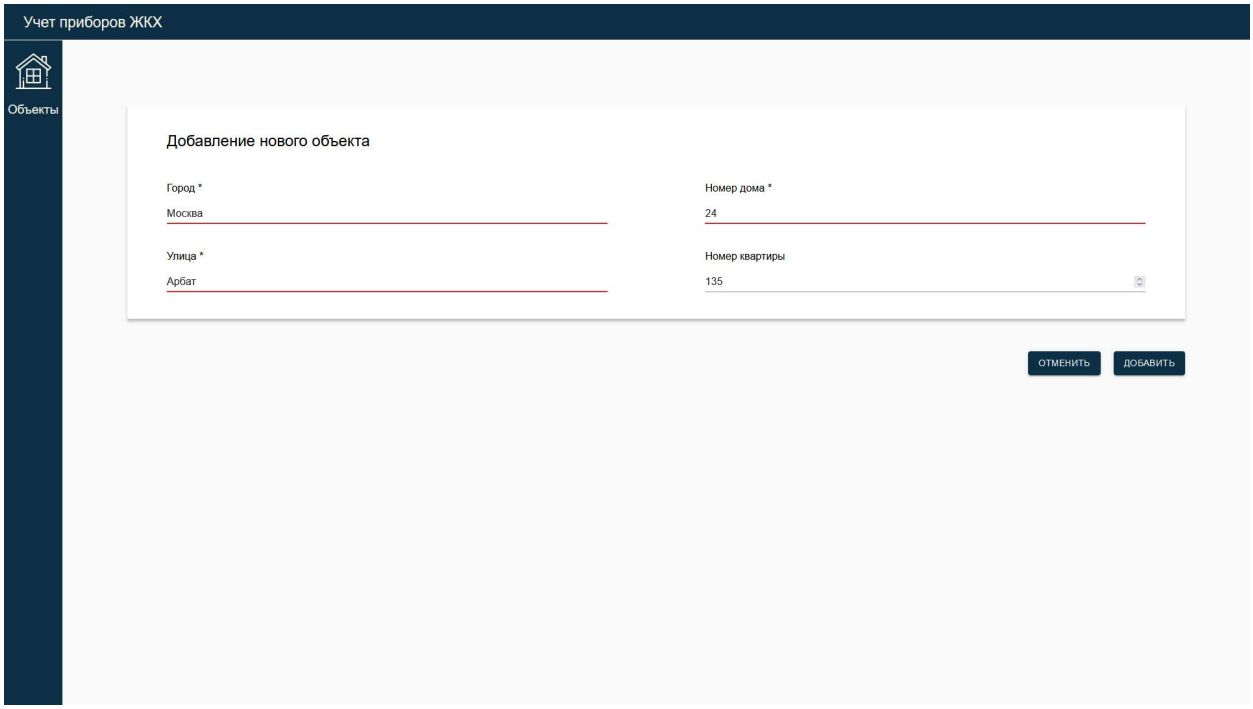


Рисунок 2.24 - Корректное заполнение формы при добавлении объекта

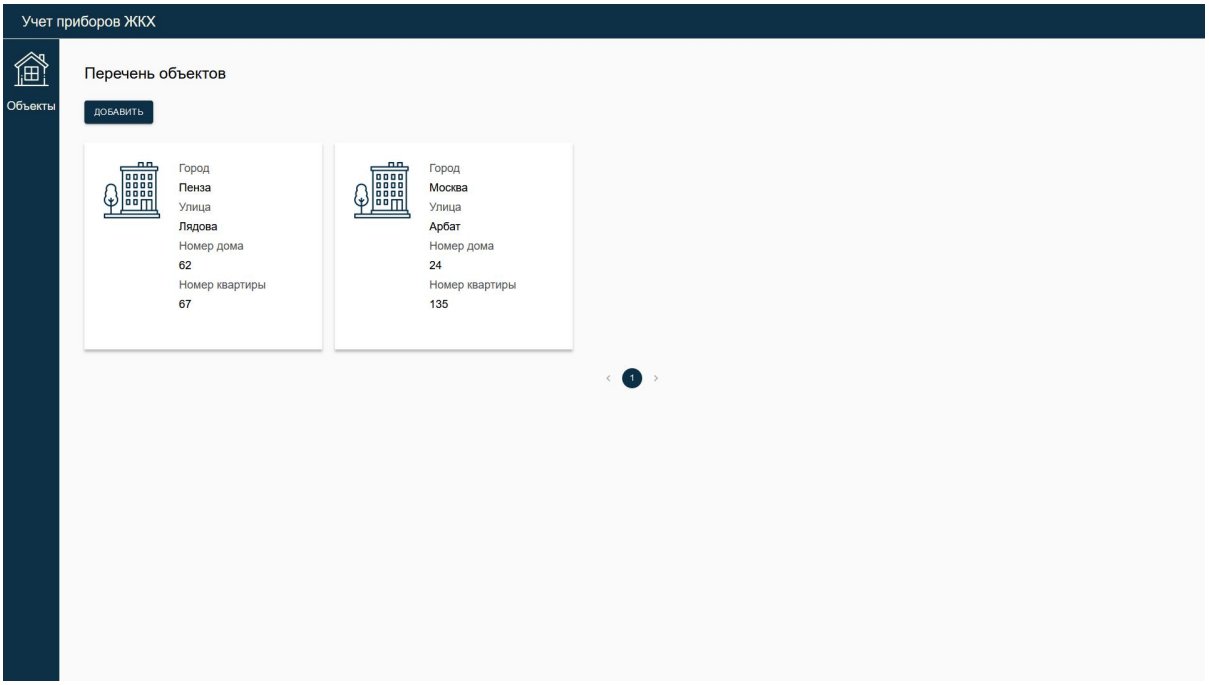


Рисунок 2.25 - Главный экран с только что добавленным объектом

При нажатии на панель с только что созданным объектом откроется другое окно с информации о нем (Рисунок 2.26)

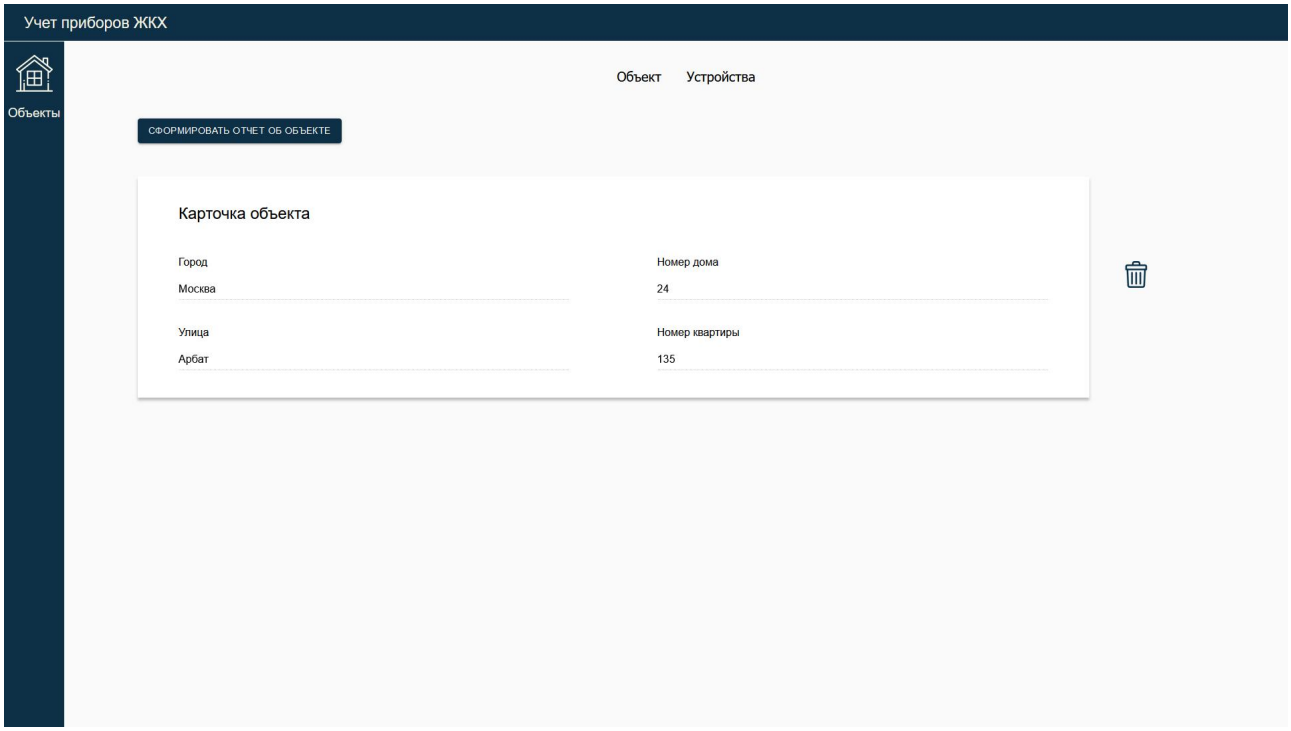


Рисунок 2.26 - информация о объекте

При нажатии на знак «мусорка» можно удалить объект, но перед тем как удалить у пользователя спрашивают о действительности его намерений удалить объект (Рисунок 2.27). Если согласится, то пользователя вернет на главную страницу, на которой он увидит что объект был удален (Рисунок 2.28).

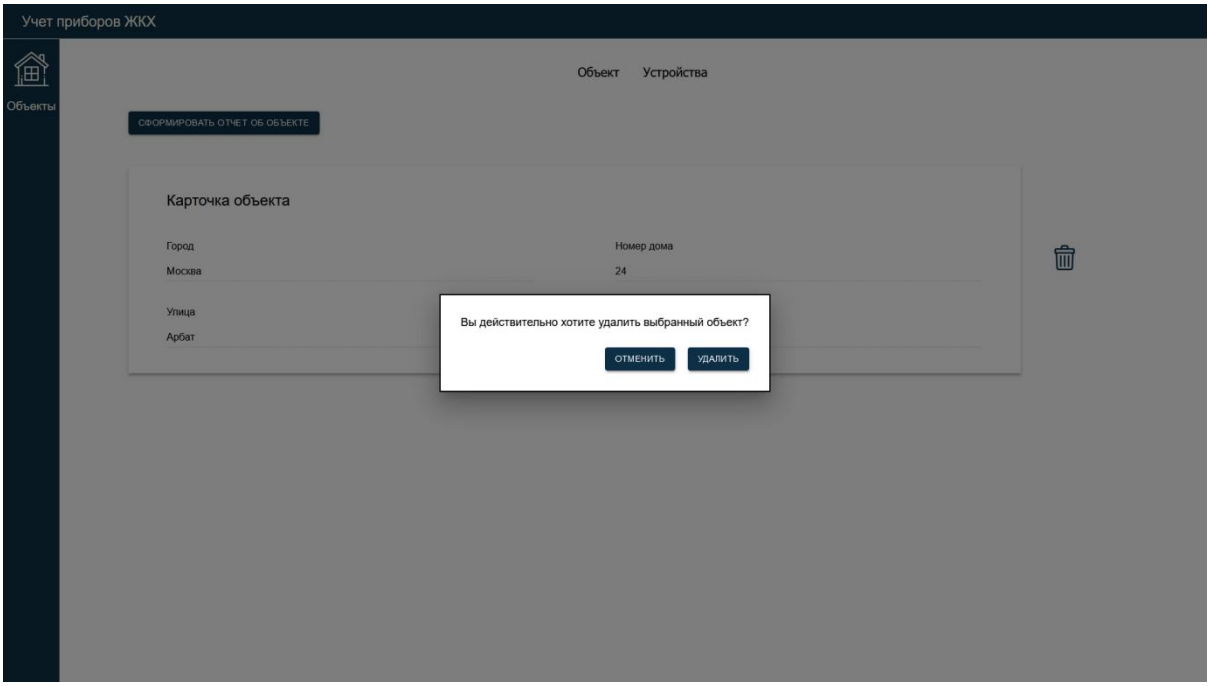


Рисунок 2.27 - Попытка удаления

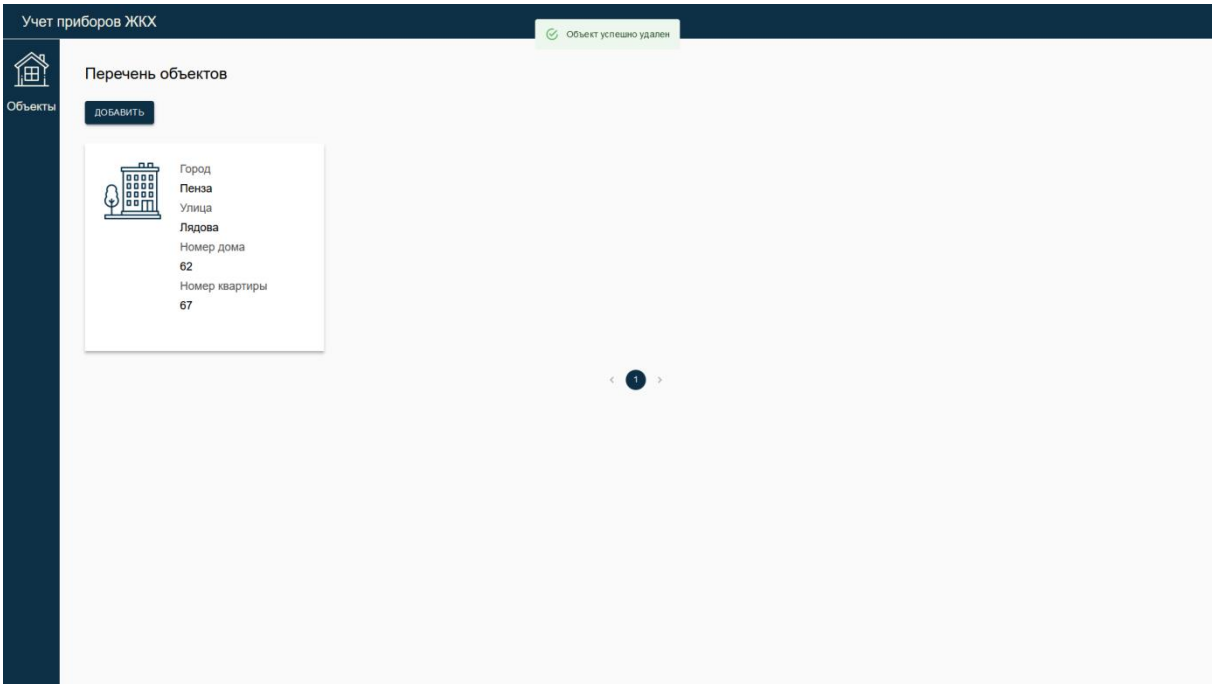


Рисунок 2.28 - Удаление объекта

При нажатии на кнопку «сформировать отчёт об объекте» откроется html страница с информацией о объекте (Рисунок 2.29).

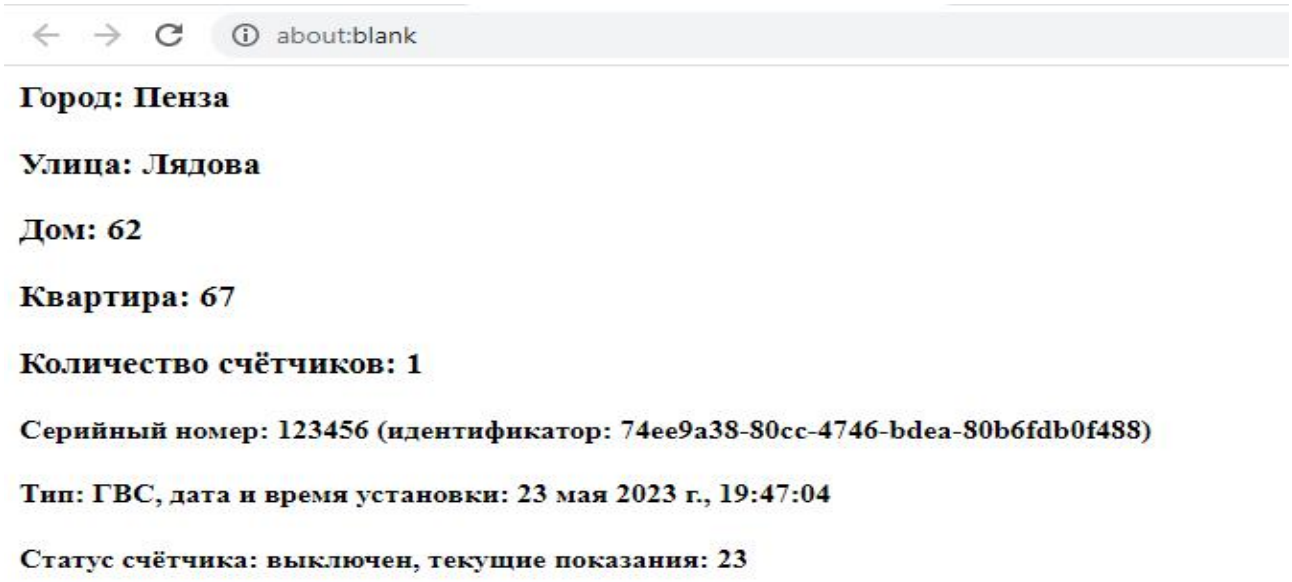


Рисунок 2.29 - Отчет об объекте

Если перейти на вкладку устройства то мы увидим пустую страницу (Рисунок 2.30), потому что на объекте еще не установленно ни одного счетчика. Чтобы добавить счетчик нужно нажать на кнопку «добавить», после чего откроется другое окно, в котором нужно ввести информацию о счетчике (Рисунок 2.31).

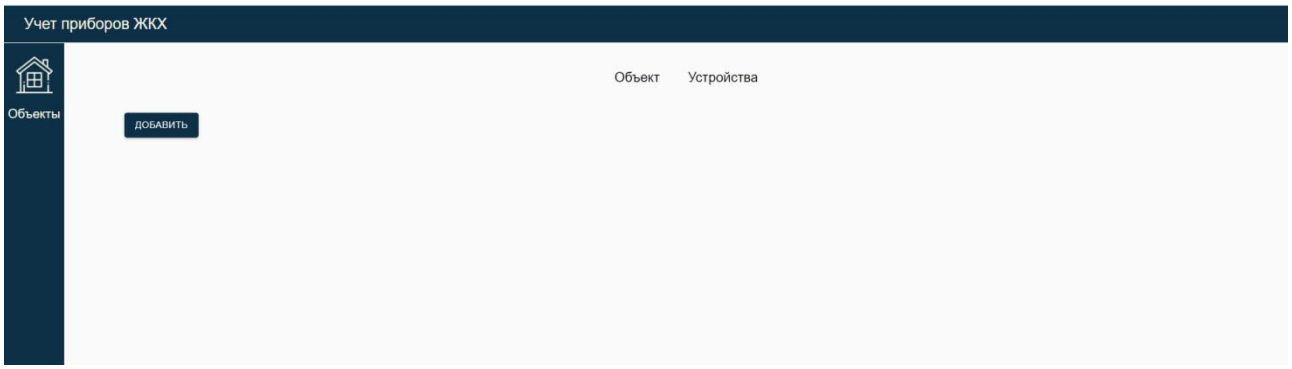


Рисунок 2.30 - Вкладка «устройства» без устройств

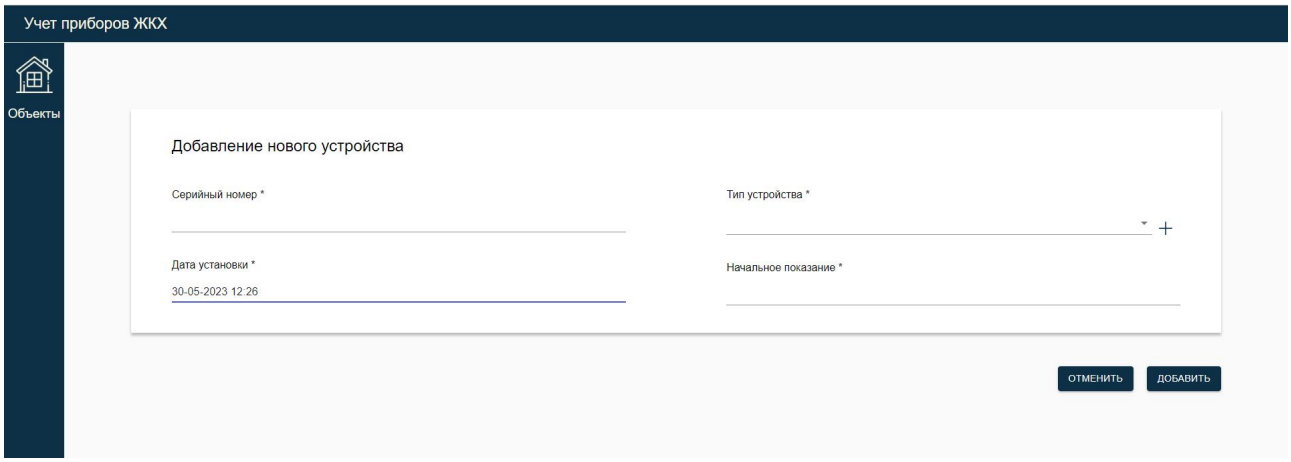


Рисунок 2.31 - Добавление устройства

По умолчанию дата установки выставляется сегодняшняя, но если нажать на поле с выбором даты то появится календарь на котором можно выбрать нужную дату (Рисунок 2.32).

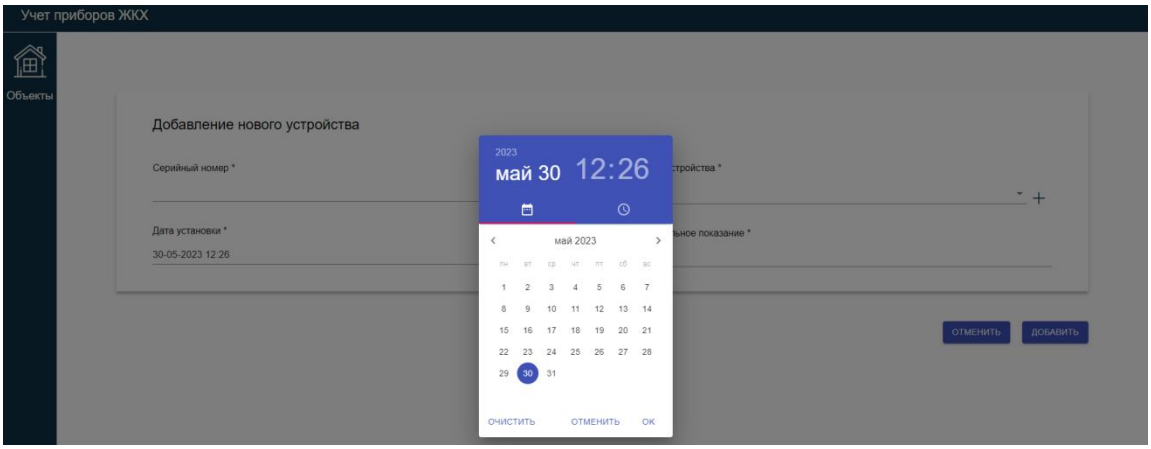


Рисунок 2.32 - Выбор даты установки

Так же если оставить обязательные поля незаполненными, то придет оповещение о том, что нужно их заполнить (Рисунок 2.33).

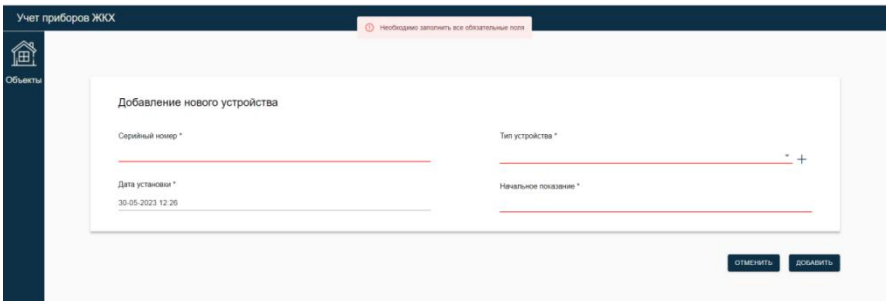


Рисунок 2.33 - Оповещение о обязательности заполнения полей

При заполнении поля «Тип устройства» необходимо выбрать тип из предложенного списка (Рисунок 2.34), либо добавить новый (Рисунок 2.35).

Рисунок 2.34 - Выбор из уже существующих типов устройств

Рисунок 2.35 - Добавление нового типа устройства

После корректного заполнения всех полей нужно нажать на кнопку «добавить», после чего к объекту будет прикреплено устройство (Рисунок 2.36). Видим что у него статус работы выключен и показания пока что равны 0.

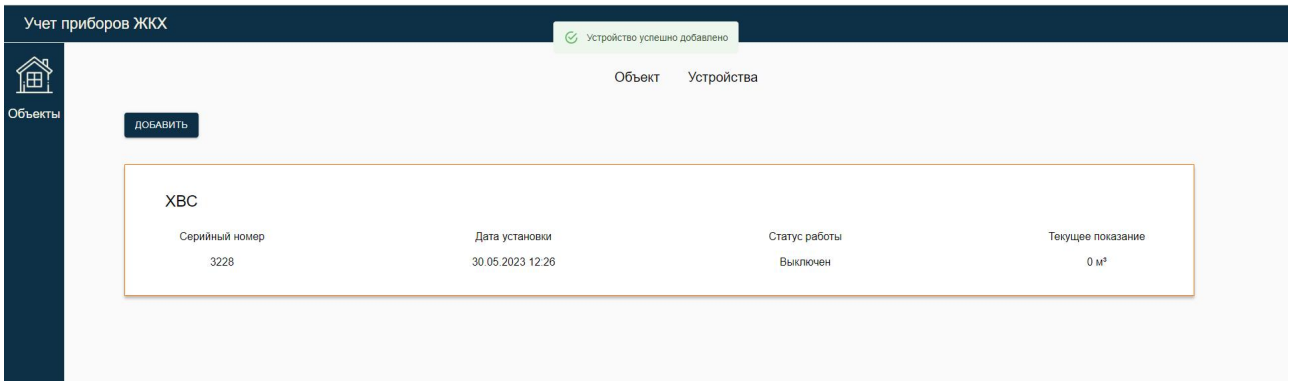


Рисунок 2.36 - Добавленное устройство

Затем можно нажать на добавленное устройства, после чего откроется другое окно с более подробной информацией (Рисунок 2.37).

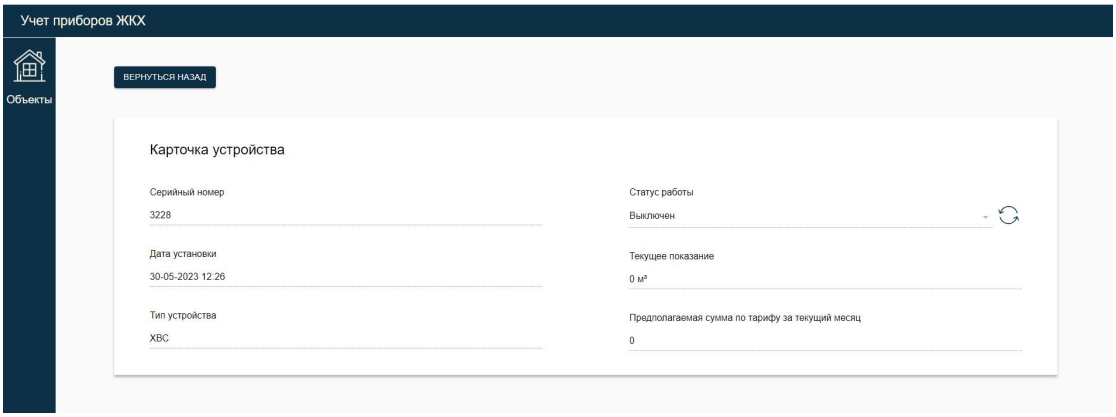


Рисунок 2.37 - Карточка устройства

На этой карточке можно увидеть всю информацию о счетчике, а самое главное на этой вкладке его можно запустить изменив статус (Рисунок 2.38).

Рисунок 2.38 - Изменение статуса работы устройства

После изменения статуса работы на «Включен» можно увидеть сообщение о том, что статус работы изменен, и текущее показание начало меняться (Рисунок 2.39).

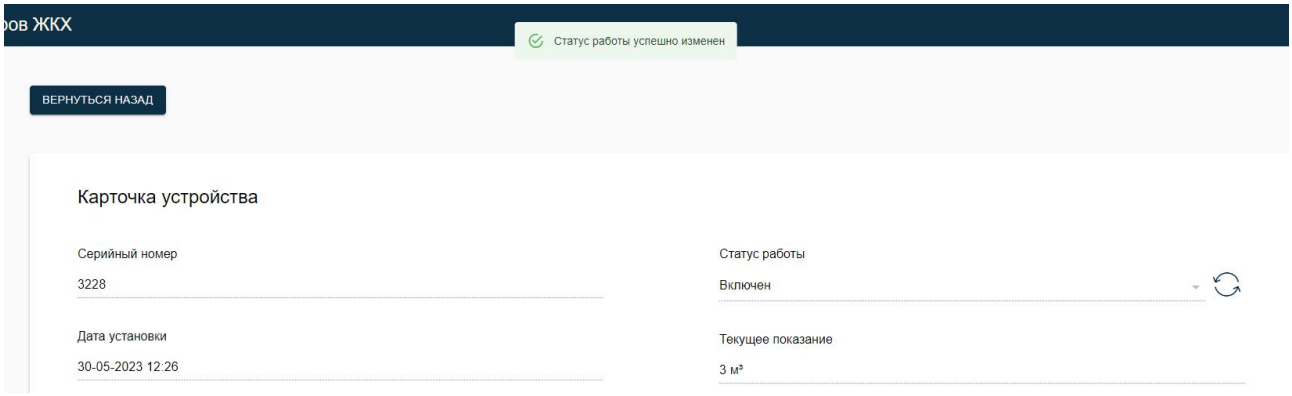


Рисунок 2.39 - Результат изменения статуса работы

Спустя небольшое количество времени можно увидеть что показание устройства увеличивается (Рисунок 2.40). Чтобы счетчик перестал генерировать значения нужно изменить его статус работы на «Выключен» (Рисунок 2.41).

Рисунок 2.40 - Показание спустя время

Рисунок 2.41 - Выключение устройства

Ниже, на этой же вкладке, можно увидеть ячейки в которых можно узнать ту или иную статистику по счетчику (Рисунок 2.42).

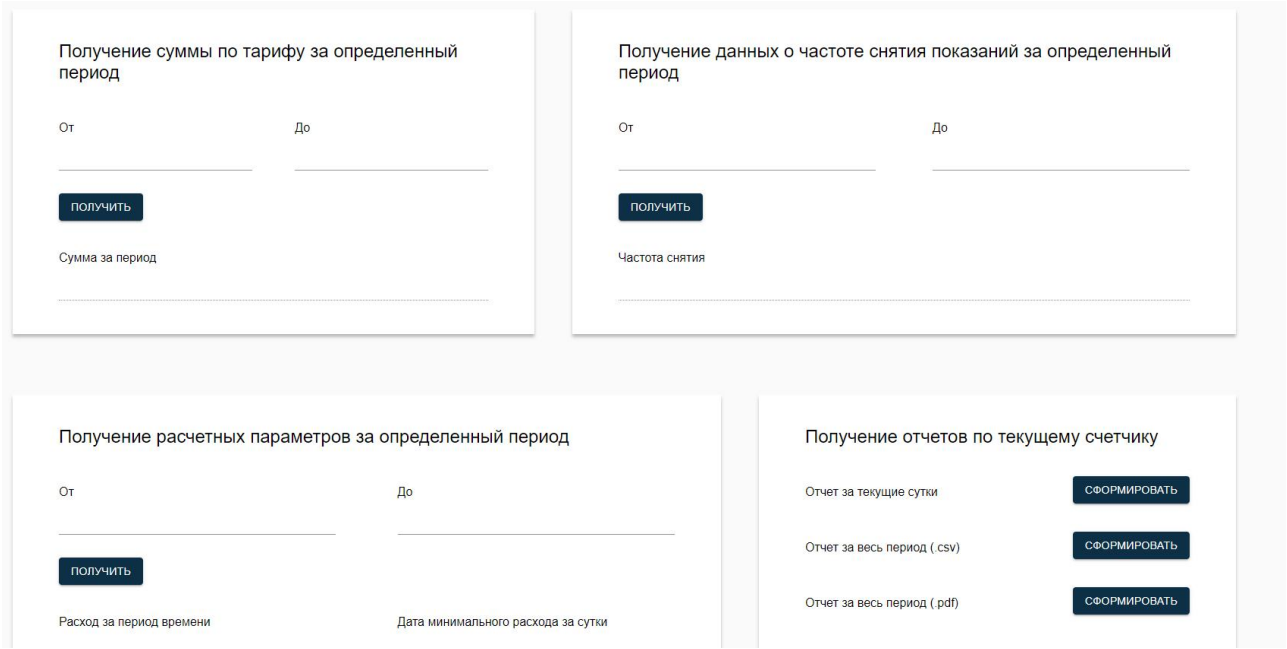


Рисунок 2.42 - Статистика и отчетность

Чтобы узнать сумму по тарифу за определенный период нужно установить этот период и нажать кнопку «Получить» (Рисунок 2.43).

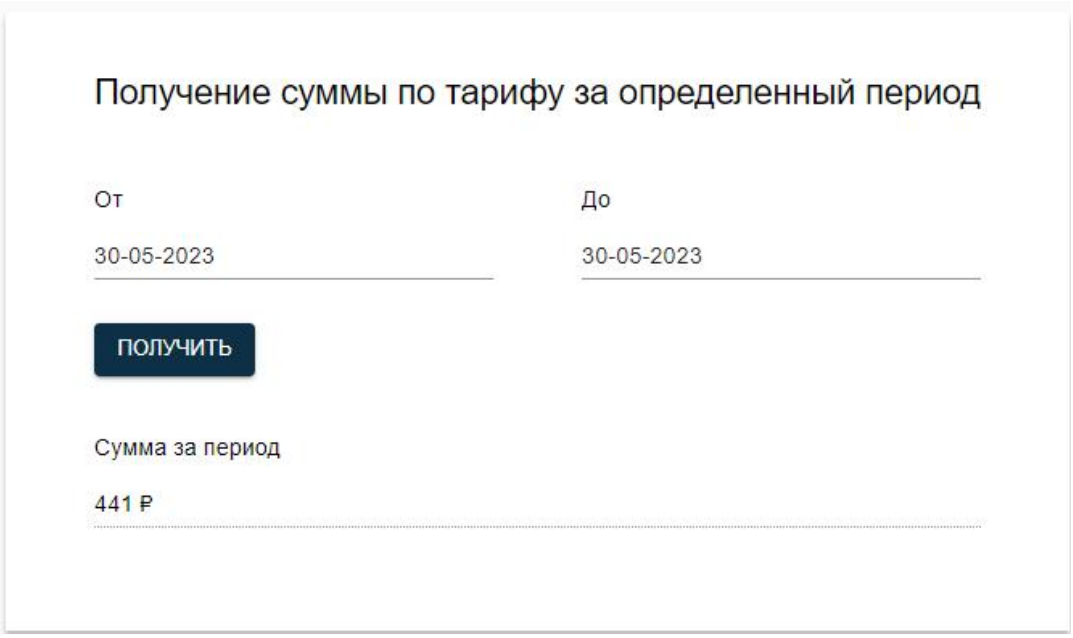


Рисунок 2.43 - Получение суммы по тарифу

Так же можно получить данные о частоте снятия показаний за определенный период, для этого так же нужно указать период (Рисунок 2.44).

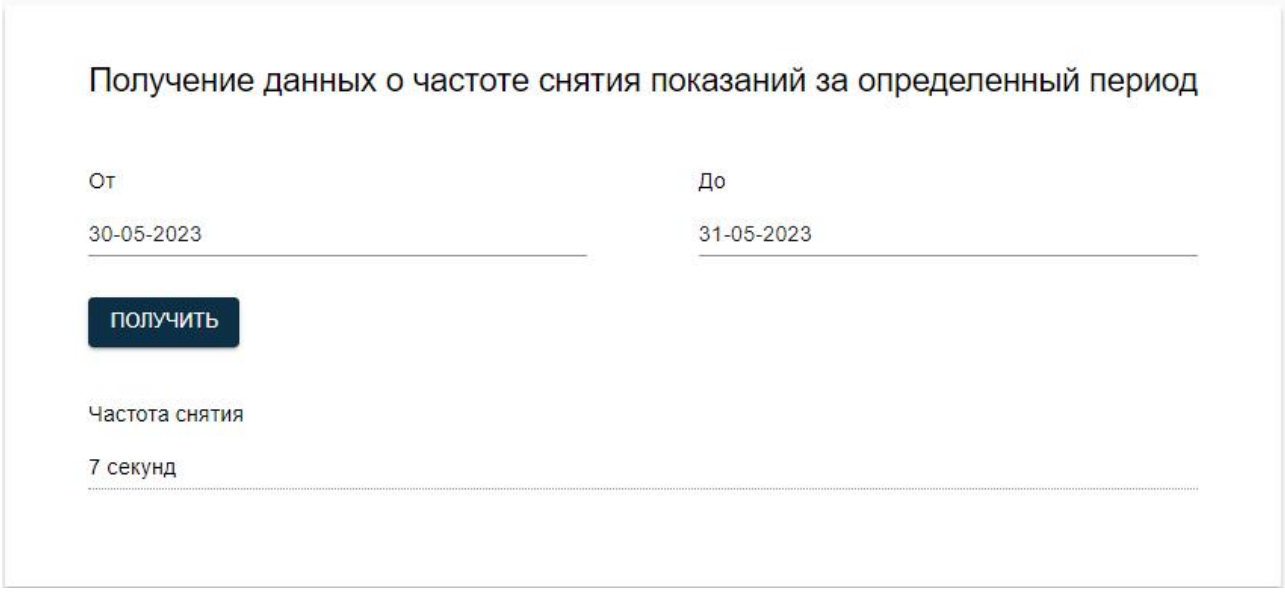


Рисунок 2.44 - Получение частоты снятия показаний

Еще реализована возможность получения расчетных параметров за определенный период (Рисунок 2.45) На этой панели можно увидеть такие показатели как расход за период, дата когда расходов было меньше всего, дату когда расходов было больше всего, и средний расход за сутки.

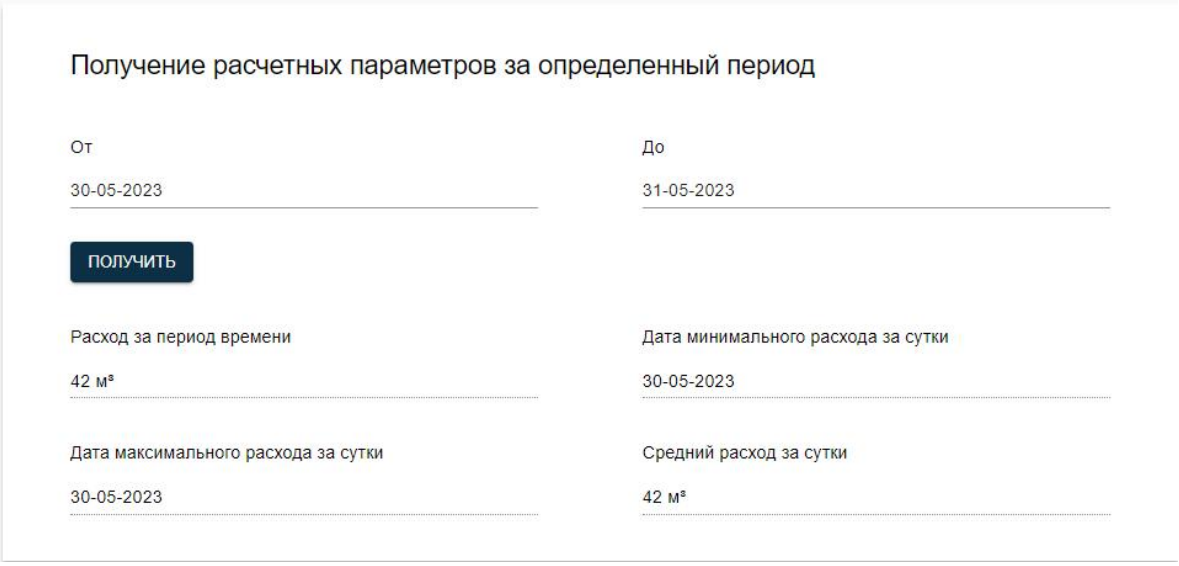


Рисунок 2.45 - Получение расчетных параметров за определенный период

Ниже так же можно составить график расходов за определенный период (Рисунок 2.46).

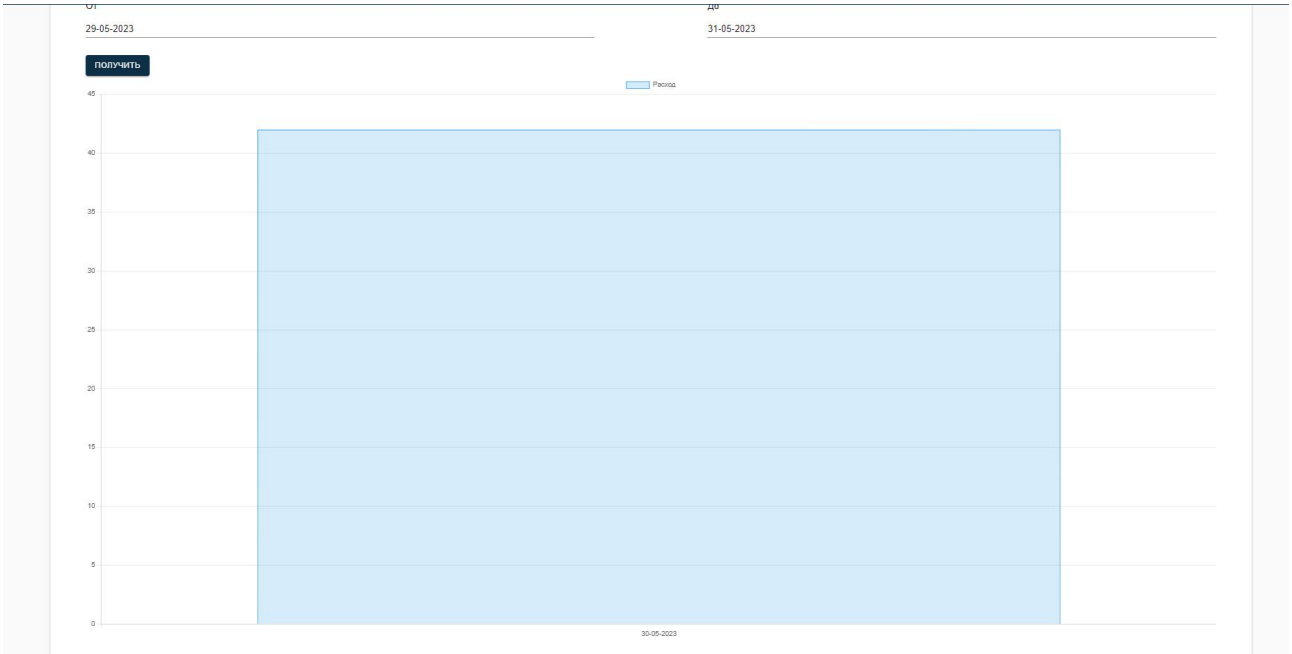


Рисунок 2.46 - График расходов за период

Так же на вкладке устройства можно увидеть панель с разными видами отчетности (Рисунок 2.47).

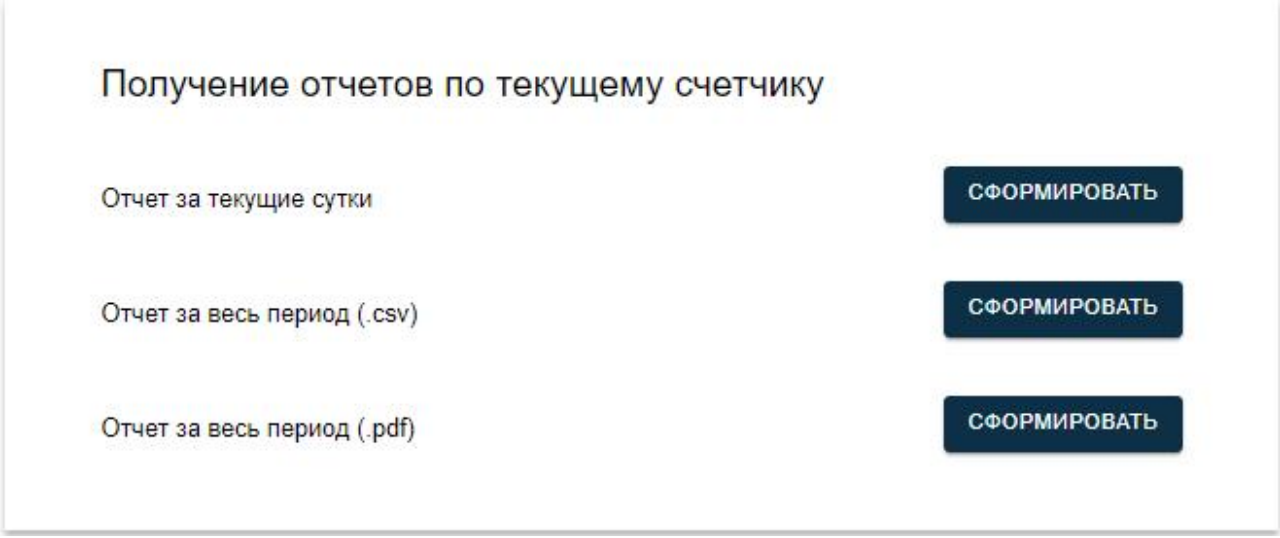


Рисунок 2.47 - Отчеты по счетчику

При нажатии на кнопку «сформировать» у отчета за весь период (*.csv*) произойдет скачивание файла в формате *csv* (Рисунок 2.48). Он содержит информацию о устройстве за весь период, когда было последнее показание, текущее показание и среднее время снятия показаний (Рисунок 2.49).

Риуснок 2.48 - Скачивание отчета

	Идентификатор устройства	Дата	Последнее время события	Последнее пока	Среднее время между показаниями
2	ed06d9df-1796-4a7e-ab40-93d85d327af1	2023-05-30	12:50:34	42	7

Рисунок 2.49 - Отчет за весь период по счетчику

Так же можно скачать подобный отчет в формате *PDF*. После нажатия на кнопку «Сформировать» начнется скачивание файла в формате *PDF* (Рисунок 2.50).

Отчет по счетчику ed06d9df-1796-4a7e-ab40-93d85d327af1

Идентификатор устройства	Дата	Последнее время события	Последнее показание	Среднее время между показаниями
ed06d9df-1796-4a7e-ab40-93d85d327af1	2023-05-30	12:50:34	42	7

Рисунок 2.50 - Отчет по устройству в формате PDF

2.3 Инсталяция средств запуска системы

Для запуска системы мониторинга показателей счетчиков ресурсов необходимо установить средство контейнеризации *Docker*, так как все написанные сервисы были собраны в контейнер.

Docker - это платформа для разработки, доставки и запуска приложений в контейнерах. Контейнеры представляют собой легковесные, переносимые и автономные окружения, которые могут содержать все необходимое для запуска приложения, включая зависимости, библиотеки и другие файлы.

В контейнере собраны все сервисы отвечающие за серверную часть, база данных *PostgreSQL*, брокер сообщений *Kafka*, и хранилище очередей *Zookeper*, которое нужно для работы *Kafka*.

2.3.1 Установка *Docker*

Для установки докера на компьютер с операционной системой *Windows* нужно скачать файл *Installer.exe* с официального сайта. Далее нужно запустить этот файл и выбрать все нужные компоненты для настройки.

После установки нас встретит главный экран (Рисунок 2.51)

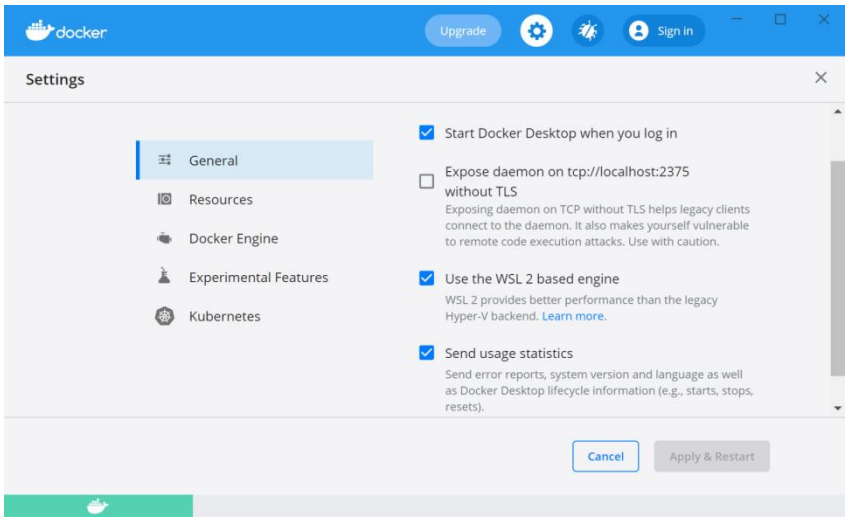


Рисунок 2.51 - *Docker desktop*

Чтобы проверить, что все правильно установилось, нужно открыть *PowerShell* от имени администратора и ввести команду *docker version* (Рисунок 2.52)

```
PS C:\Users\vtkac> docker version
Client: Docker Engine - Community
 Cloud integration: 1.0.7
 Version:           20.10.2
 API version:       1.41
 Go version:        go1.13.15
 Git commit:        2291f61
 Built:             Mon Dec 28 16:14:16 2020
 OS/Arch:           windows/amd64
 Context:           default
 Experimental:      true

Server: Docker Engine - Community
 Engine:
  Version:          20.10.2
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.13.15
  Git commit:       8891c58
  Built:            Mon Dec 28 16:15:28 2020
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.4.3
  GitCommit:        269548fa27e0089a8b8278fc4fc781d7f65a939b
 runc:
  Version:          1.0.0-rc92
  GitCommit:        ff819c7e9184c13b7c2607fe6c30ae19403a7aff
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
PS C:\Users\vtkac>
```

Рисунок 2.52 - Проверка корректности установки

2.3.2 Запуск системы мониторинга показателей счетчиков ресурсов

Для запуска нужно иметь файл *docker-compose.yml.* В этом файле описаны все контейнеры (Рисунок 2.53).

```
docker-compose.yml – Блокнот
Файл  Правка  Формат  Вид  Справка
version: '3'
services:
  postgres:
    container_name: postgres_stepan
    #    build:
    #      context: ./docker-data/postgres/init
    #      dockerfile: postgres.Dockerfile
#    volumes:
      #      - ./pgsql:/var/lib/postgresql/data
#      - ./docker-data/postgres/init/init.sql:/docker-entrypoint-initdb.d/init.sql
    image: postgres:12
    environment:
      POSTGRES_DB: kafkameter
      POSTGRES_USER: kafkameter
      POSTGRES_PASSWORD: kafkameter
    ports:
      - "5433:5432"
    networks:
      - stepan-gleb-network

  zookeeper:
    container_name: zookeeper_stepan
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - "22181:2181"
    networks:
      - stepan-gleb-network

  kafka:
    container_name: kafka_stepan
    image: confluentinc/cp-kafka:latest
    depends_on:
      - zookeeper
    ports:
      - "29092:29092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092,PLAINTEXT_HOST://kafka_stepan:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    networks:
      - stepan-gleb-network
```

Рисунок 2.53 - *docker-compose.yml*

Для выполнения этого файла нужно в терминале написать *docker-compose up* (Рисунок 2.54)

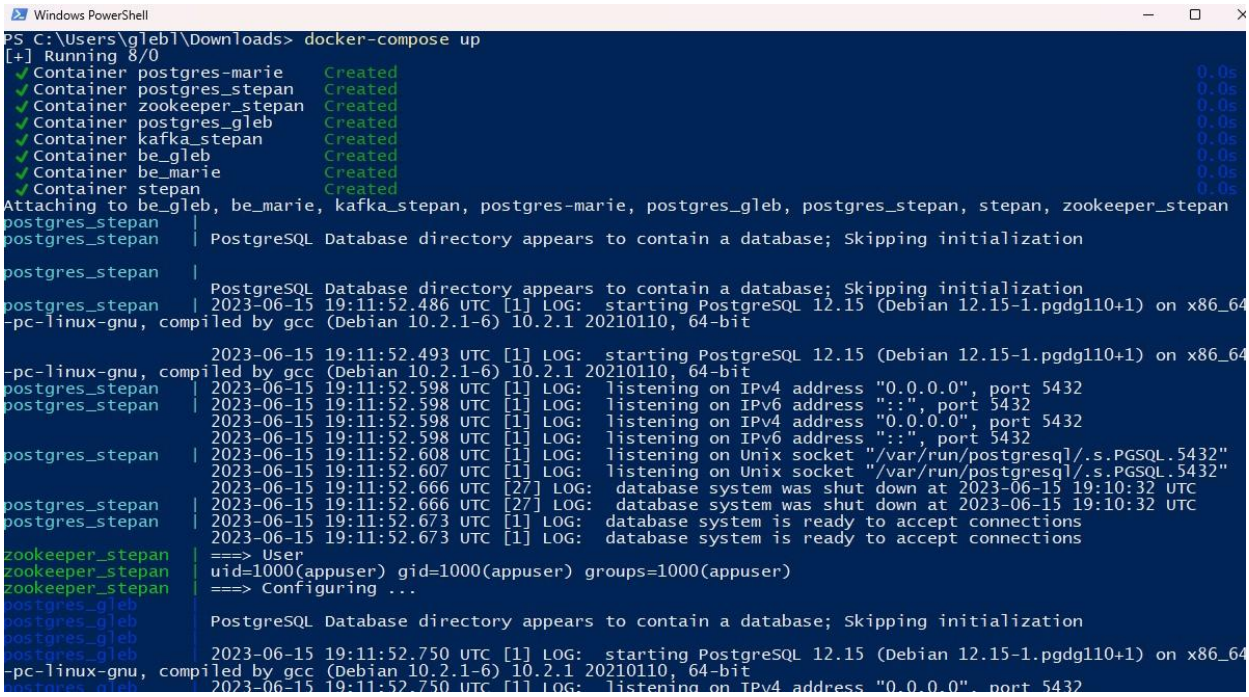


Рисунок 2.54 - запуск приложения.

После выполнения этой команды откроется браузер и на странице с адресом *localhost*:3000 откроется приложение.

3. Безопасность жизнедеятельности. Основные производственные опасные и вредные факторы, воздействующие на программиста.

Программисты, как правило, работают в офисных условиях и не подвергаются многим опасным и вредным производственным факторам, которые могут негативно повлиять на здоровье. Однако, их работа связана со многими факторами, которые могут снижать уровень безопасности жизнедеятельности.

Основными производственными опасными факторами, воздействующими на программиста, могут быть:

- Неблагоприятная обстановка в рабочем помещении, такая как плохое освещение, шум и пыль;

- Сидячий образ жизни и длительное сидение за компьютером, что может привести к различным заболеваниям, таким как болезни опорно-двигательного аппарата, нарушения кровообращения и дыхания;

- Риск возникновения стрессовых ситуаций, так как работа программиста связана с высокой ответственностью, необходимостью постоянного обучения и частыми сроками выполнения задач;

- Риск возникновения профессиональных заболеваний, таких как синдром карпального канала, который может возникнуть при частом использовании компьютерной мыши или клавиатуры.

Чтобы снизить риск снижения уровня безопасности жизнедеятельности необходимо выполнять комплекс профилактических работ. К примеру чтобы улучшить освещение необходимо правильно расположить рабочее место относительно источника света, а так же подобрать подходящую мощность и тип лампы. Это поможет избежать серьезных проблем с зрение. Так же следует регулярно проводить уборку в помещении чтобы снизизить запыленность. Это поможет избежать вызова аллергических реакций, а так же уменьшит негативное воздействие на органы дыхания.

Для профилактики сидячего образа жизни необходимо следить за своим физическим состоянием и уделять время физической активности. Рекомендуется проводить не менее 150 минут умеренной или 75 минут интенсивной активности в неделю. Это может быть любая физическая нагрузка, которая подходит для вашего уровня физической подготовки, например, ходьба, бег, плавание.

Также для профилактики сидячего образа жизни следует следить за питанием, чтобы избежать ожирения и других заболеваний, связанных с питанием. Важно также выполнять упражнения для растяжки мышц и суставов, особенно на шее, спине и плечах, чтобы снизить риск болей в этих областях.

При длительном сидении за компьютером следует следить за правильной позой тела и установкой рабочего места. Рабочее кресло должно быть удобным и подходящим по размеру, а стол - правильной высоты, чтобы избежать напряжения в шее и спине (Рисунок 4.1). Необходимо также делать перерывы каждые 30 минут-1 час для разминки и растяжки мышц.

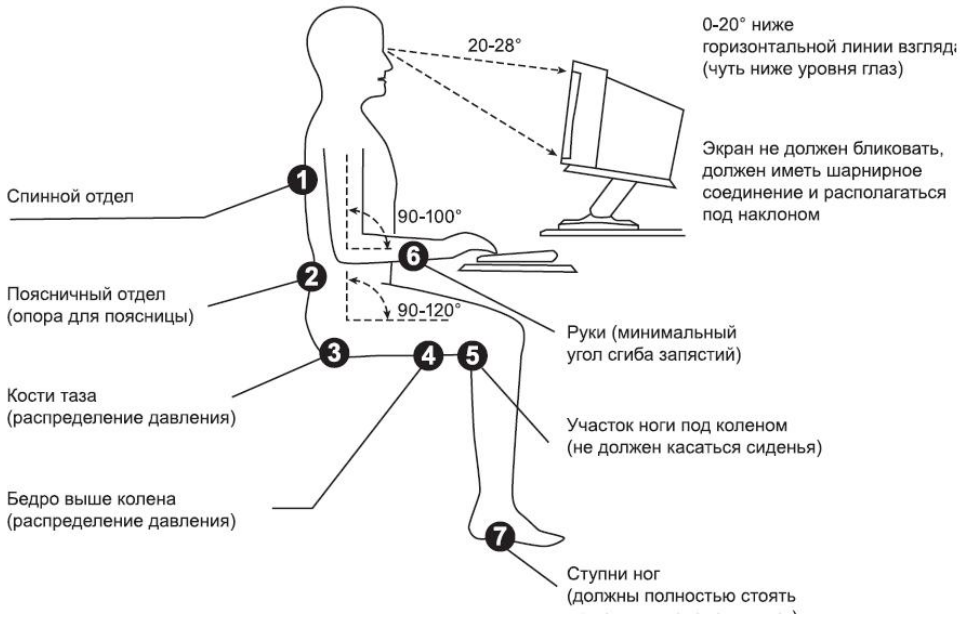


Рисунок 3.1 - Инструкция по правильному сидению за рабочим местом

4. Экономическое обоснование разработки

Разработка системы мониторинга показаний счетчиков ресурсов осуществлялась в рабочей зоне open space площадью 300 кв.м. Содержание помещения в месяц стоит приблизительно 250000 рублей. Это с учетом самой аренды и уборки помещения.

Для разработки были закуплены вычислительные машины, а лицензионное программное обеспечение было выбрано таким образом, что все средства разработки являлись бесплатными.

Необходимое оборудование и программное обеспечение использованные при разработке представлены в таблице 4.1 и 4.2.

Таблица 4.1 - Использованное оборудование

Оборудование	Кол-во	Ориентировочная цена за единицу, руб
Ноутбук Acer Nitro 5	4	80000
Мышь	4	250

Таблица 4.2 - Использованное лицензионное программное обеспечение

Лицензионное программное обеспечение	Источник	Цена, рублей	Срок действия лицензии, дней
ОС Linux Ubuntu	сайт производителя	0	бессрочная
Docker	сайт производителя	0	бессрочная

В разработке системы мониторинга показаний счетчиков ресурсов осуществлялась командой разработчиков. Состав команды и информация о их заработной плате представлена в таблице 4.3.

Таблица 4.3 - Состав команды

Специалист	Количество, человек	Сумма заработной платы, рублей
Backend разработчик	2	110000
Frontend разработчик	1	40000
Тестировщик	1	35000

Себестоимость разработки системы мониторинга показаний счетчиков ресурсов включает следующие статьи затрат:

- затраты на приобретение оборудования;
- затраты на дополнительные материалы;
- затраты на заработную плату;
- затраты на обслуживание.

Затраты на оборудование в период разработки составят 321000 рублей на основании данных в таблице 4.1.

Затраты на программное обеспечение в период разработки составят 0 рублей и представлены в таблице 4.2.

Затраты на дополнительные материалы в период разработки составят 0 рублей в соответствии с условием что потребности в закупке нет.

Затраты на заработную плату включают затраты на основную и дополнительную заработную плату разработчиков программного обеспечения и отчислениях на страховые взносы.

Затраты на основную заработную плату разработчиков определяются исходя из установленной заработной платы, количества специалистов и их занятости в процессе разработки программного продукта (продолжительность работы в днях).

Дополнительная заработная плата составляет 15% от затрат на основную заработную плату разработчиков.

С учетом выходных дней при 5-днейвной рабочей неделе количество рабочих дней в месяце в среднем составляет 21 день.

Затраты на основную и дополнительную заработную плату разработчиков программного обеспечения представлены в таблице 4.4.

Таким образом, затраты на заработную плату разработчиков составят 555000 рублей.

Отчисления на страховые взносы по нормам, установленным законодательством в ПРФ, ФОМС, ФСС РФ вычисляются исходя из данных представленных ниже.

Таблица 4.4 – Затраты на основную и дополнительную заработную плату разработчиков

Исполнитель	Количество, человек	Сумма заработной платы, рублей	Период занятости, дней	Основная заработная плата, рублей	Дополнительная заработная плата, рублей	Всего рублей
Backend разработчик	2	110000	63	330000	49500	379500
Frontend разработчик	1	40000	63	120000	18000	138000
Тестировщик	1	35000	63	105000	15750	120750
Итого	4	185000	189	555000	83250	638250

Отчисления на страховые взносы производится согласно «Налоговому кодексу Российской Федерации (часть вторая)» от 05.08.2000 № 117-ФЗ (ред. от 26.03.2022).

Действующие тарифы на страховые взносы работодателей установлены статьей 34 НК РФ и составляют: ОПС (обязательное пенсионное страхование) – 22%, ВНиМ (страхование временной нетрудоспособности и в связи с материнством) – 2,9%, ОМС (обязательное медицинское страхование) – 5,1%.

Общая сумма отчислений на страховые взносы составит 166500 рублей.

Общая сумма затрат является суммой: затрат на оплату труда включая налоги и сборы, затраты на закупку оборудования и лицензионного ПО и составляет 1125750 рублей.

Заключение

В ходе выполнения дипломного проекта был произведен анализ предметной области, и на основе анализа было сформулировано техническое задание на разработку системы мониторинга показаний счетчиков ресурсов, а так же был подобран стек технологий. В соответсвии с техническим заданием была разработана система мониторинга показаний счетчиков ресурсов на языках программирования Java и JavaScript. Графический интерфейс был реализован с помощью фреймворка React.

Разработанная система позволяет отслеживать показания счетчиков и составлять отчетность по этим показаниям.

Список использованных источников

1. Автор Bill Bejeck. «Kafka Streams in Action: Real-time apps and microservices with the Kafka Streams API». стр 48-75.

2. Автор: Герберт Шилдт: Java полное руководство, стр 1403-1429

3. Авторы Райэн Шайверс и Рэнди Шайверс. "Building Microservices with Spring Boot". стр 104-159

4. Автор Stoyan Stefanov. "React: Up & Running: Building Web Applications". стр 100-189.

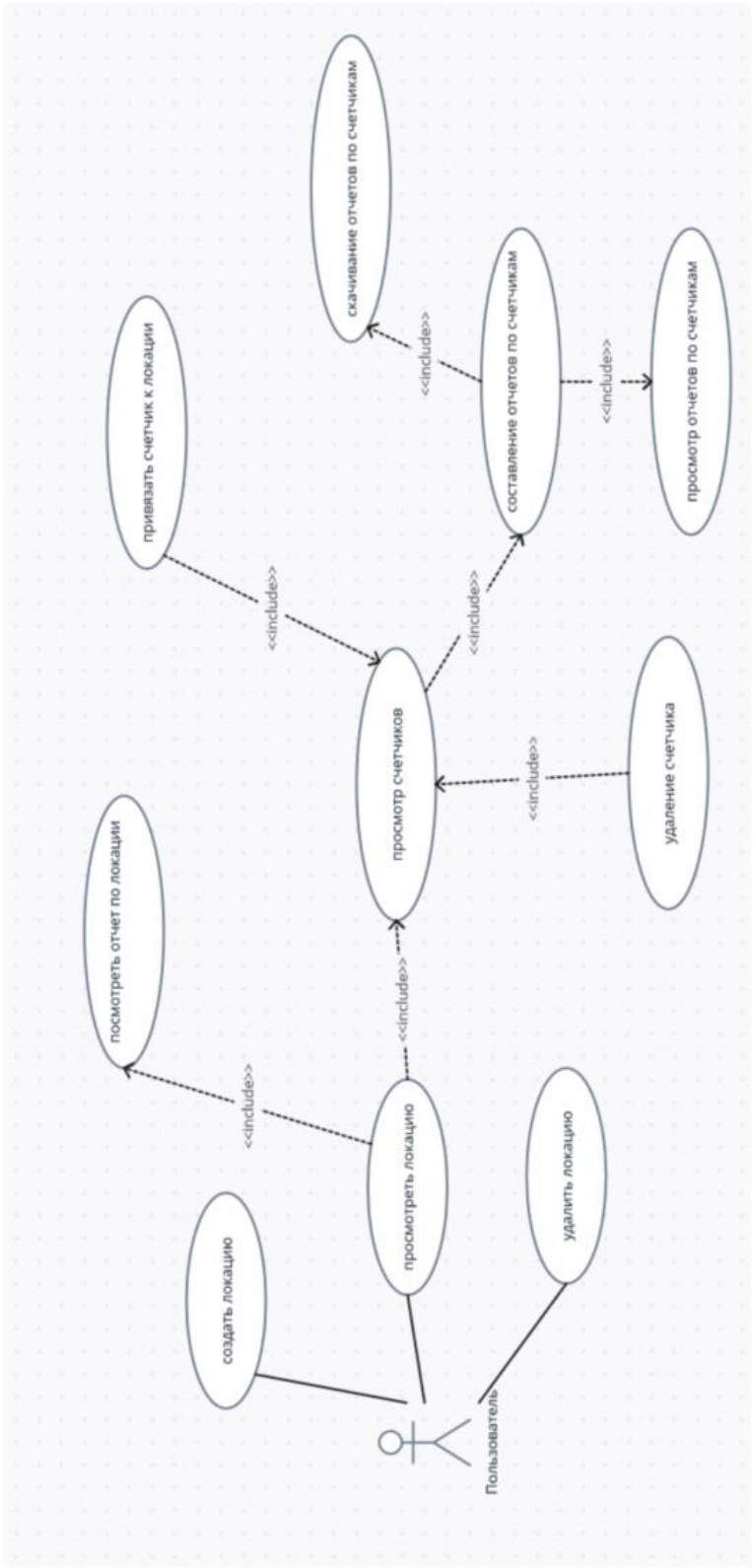
Приложение А

(обязательное)

Система мониторинга показателей счетчиков ресурсов

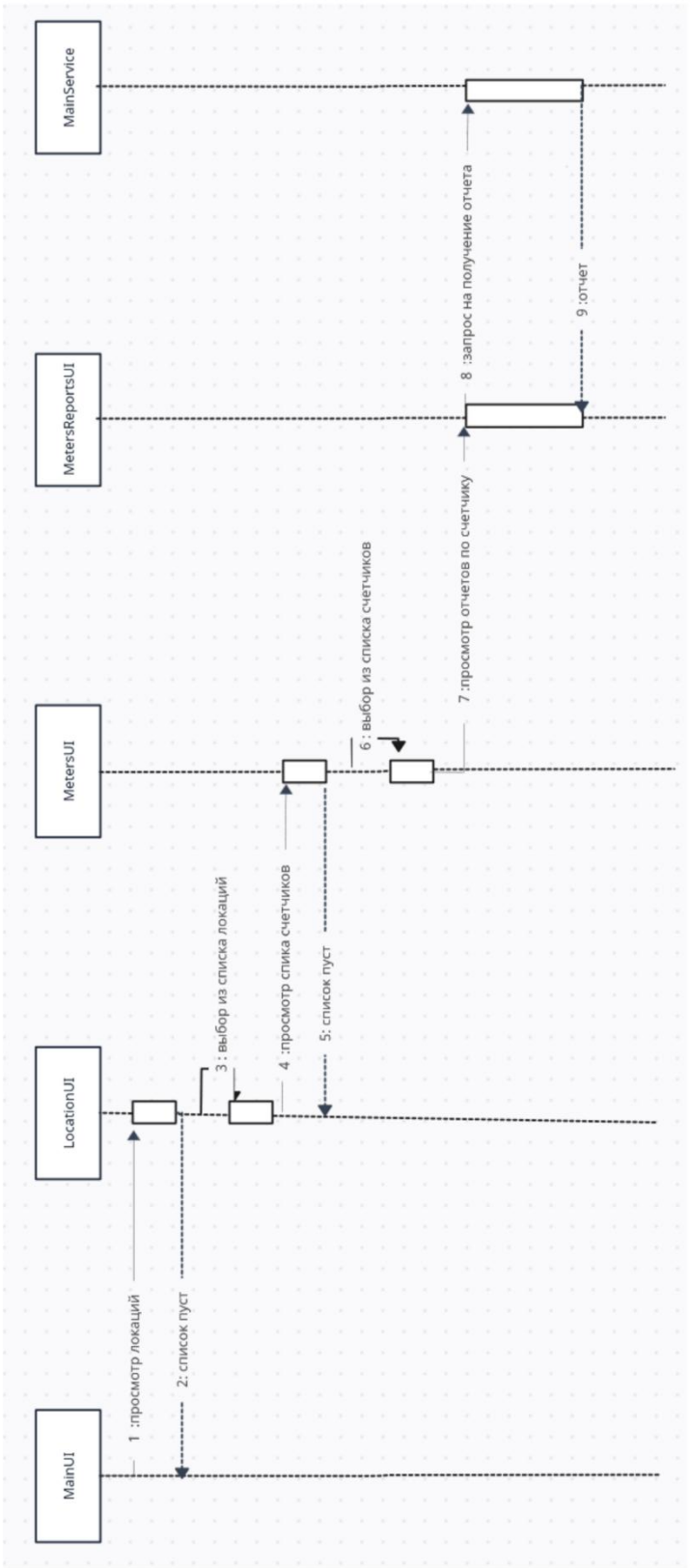
UML-диаграммы

Приложение А.1 – Диаграмма вариантов использования



					ПГУ 1.090501.06.002 ДЗ		
Изм.	Лист	№ докум.	Подпись	Дата			
Разраб.		Дубинин			Система мониторинга показаний счетчиков ресурсов	Лит.	Лист	Листов
Провер.		Дубинин В.Н						
Н. Контр.		Бычков А.С.				ФВТ, каф. ВТ гр. 18ВОЭ1		
Утверд.		Матросов М.А.						

Приложение А.2 – Диаграмма последовательности



					ПГУ 1.090501.06.003 Д13				
Изм	Лист	№ докум.	Подпис	Дата					
Разраб.		Лучихин			Система мониторинга показаний счетчиков ресурсов		Лит.	Лист	Листов
Провер.		Дубинин В.Н							
Н. Контр.		Бычков А.С.					ФВТ, каф. ВТ гр. 18ВОЭ1		
Утверд.		Митрохин М.А.							

Приложение Б

(обязательное)

Приложение для обработки онлайн-заказов на базе облачных сервисов

Листинг программы

Модуль LocationController.java

```
package com.example.kafkameters.rest;

import com.example.kafkameters.rest.api.LocationAPI;
import com.example.kafkameters.rest.dto.LocationRqDto;
import com.example.kafkameters.rest.dto.LocationRsDto;
import
com.example.kafkameters.rest.exception.BadRequestException;
import com.example.kafkameters.service.LocationService;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RestController;
import
org.springframework.web.servlet.support.ServletUriComponentsBuilde
r;

import java.util.Optional;

@Slf4j
@RestController
@RequiredArgsConstructor
public class LocationController implements LocationAPI {

    private static final String LOCATION_NOT_CREATED =
"Объект не создан";

    private final LocationService locationService;

    @Override
    public ResponseEntity<LocationRsDto> create(LocationRqDto
locationRqDto) {
        return
Optional.ofNullable(locationService.create(locationRqDto))
```

```
                    .map(created -> {
                        var                 url                 =
ServletUriComponentsBuilder.fromCurrentRequest()
                                .path("/{id}")
                                .buildAndExpand(created.getId())
                                .toUri();
                        return
ResponseEntity.created(url).body(created);
                    })
                    .orElseThrow(()            ->            new
BadRequestException(LOCATION_NOT_CREATED));
        }

        @Override
        public LocationRsDto getById(Long id) {
            return locationService.getById(id);
        }

        @Override
        public Page<LocationRsDto> getAll(Pageable pageable) {
            return locationService.getAll(pageable);
        }

        @Override
        public ResponseEntity<?> deleteById(Long id) {
            locationService.deleteById(id);
            return ResponseEntity.accepted().build();
        }
    }
```

Модуль MeterController.java

```
package com.example.kafkameters.rest;

import com.example.kafkameters.rest.api.MeterAPI;
import com.example.kafkameters.rest.dto.MeterRqDto;
import com.example.kafkameters.rest.dto.MeterRsDto;
import com.example.kafkameters.rest.dto.StatusRqDto;
```

```
import com.example.kafkameters.rest.dto.StatusRsDto;
import
com.example.kafkameters.rest.exception.BadRequestException;
import com.example.kafkameters.service.MeterService;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RestController;
import
org.springframework.web.servlet.support.ServletUriComponentsBuilde
r;

import java.util.Optional;
import java.util.UUID;

@Slf4j
@RestController
@RequiredArgsConstructor
public class MeterController implements MeterAPI {
    private static final String METER_NOT_CREATED = "Счётчик
не создан";
    private final MeterService meterService;
    @Override
    public    ResponseEntity<MeterRsDto>    create(MeterRqDto
meterRqDto) {
        return
Optional.ofNullable(meterService.create(meterRqDto))
                .map(created -> {
                    var                 url                 =
ServletUriComponentsBuilder.fromCurrentRequest()
                            .path("/{id}")
                            .buildAndExpand(created.getId())
                            .toUri();
```

```
                        return
ResponseEntity.created(url).body(created);
                    })
                    .orElseThrow(()              ->              new
BadRequestException(METER_NOT_CREATED));
        }

        @Override
        public MeterRsDto getById(UUID id) {
            return meterService.getById(id);
        }

        @Override
        public StatusRsDto switchStatus(StatusRqDto statusRqDto,
UUID id) {
            return              meterService.switchMeterStatus(id,
statusRqDto);
        }

        @Override
        public Page<MeterRsDto> getAll(Pageable pageable) {
            return meterService.getAll(pageable);
        }

        @Override
        public ResponseEntity<?> deleteById(UUID id) {
            meterService.deleteById(id);
            return ResponseEntity.accepted().build();
        }
    }
```

Модуль MeterTypeController.java

```
package com.example.kafkameters.rest;

import com.example.kafkameters.rest.api.MeterTypeAPI;
import com.example.kafkameters.rest.dto.MeterTypeRqDto;
import com.example.kafkameters.rest.dto.MeterTypeRsDto;
```

```
import
com.example.kafkameters.rest.exception.BadRequestException;
import com.example.kafkameters.service.MeterTypeService;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RestController;
import
org.springframework.web.servlet.support.ServletUriComponentsBuilde
r;

import java.util.Optional;

@Slf4j
@RestController
@RequiredArgsConstructor
public class MeterTypeController implements MeterTypeAPI {
    private static final String TYPE_NOT_CREATED = "Тип
счётчика не создан";

    private final MeterTypeService meterTypeService;

    @Override
    public ResponseEntity<MeterTypeRsDto>
create(MeterTypeRqDto meterTypeRqDto) {
        return
Optional.ofNullable(meterTypeService.create(meterTypeRqDto))
                .map(created -> {
                    var url =
ServletUriComponentsBuilder.fromCurrentRequest()
                            .path("/{id}")
                            .buildAndExpand(created.getId())
                            .toUri();
```

```
                        return
ResponseEntity.created(url).body(created);
                    })
                    .orElseThrow(()              ->              new
BadRequestException(TYPE_NOT_CREATED));
        }

        @Override
        public MeterTypeRsDto getById(Long id) {
            return meterTypeService.getById(id);
        }

        @Override
        public Page<MeterTypeRsDto> getAll(Pageable pageable) {
            return meterTypeService.getAll(pageable);
        }

        @Override
        public ResponseEntity<?> deleteById(Long id) {
            meterTypeService.deleteById(id);
            return ResponseEntity.accepted().build();
        }
    }
```

Модуль ReadingController.java

```
    @Slf4j
    @RestController
    @RequiredArgsConstructor
    public class ReadingController implements ReadingAPI {

        private final ReadingService readingService;
        @Override
        public ResponseEntity<?> getReadings(ReadingsRequestRqDto
readingsRequestRqDto) {
            var                    readings                    =
readingService.getSingleMeterReadingsForPeriod(readingsRequestRqDt
o);
```

```
            return new ResponseEntity<>(readings, HttpStatus.OK);
        }
    }
```

Модуль CounterService.java

```
package com.example.quartz_sheduler.service.impl;

import com.example.quartz_sheduler.entities.Counter;
import com.example.quartz_sheduler.mapper.CounterMapper;
import
com.example.quartz_sheduler.repository.CounterRepository;
import com.example.quartz_sheduler.rest.dto.CounterRqDto;
import com.example.quartz_sheduler.rest.dto.CounterRsDto;
import
com.example.quartz_sheduler.rest.dto.kafka_dto.CounterDto;
import
com.example.quartz_sheduler.rest.exception.NotFoundException;
import com.example.quartz_sheduler.service.CounterService;
import com.example.quartz_sheduler.service.KafkaService;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Service;

import javax.transaction.Transactional;
import java.time.Instant;
import java.util.Optional;

@Slf4j
@Service
@RequiredArgsConstructor
public class CounterServiceImpl implements CounterService {
```

```
        private static final String
COUNTER_IS_NOT_FOUND_BY_ID_MSG = "Счетчик с %s = %d не найден";
        private static final String NOT_FOUND_BY_SERIAL_NUMBER =
"Счетчик с %s = %s не найден";

        private final CounterRepository counterRepository;
        private final CounterMapper counterMapper;

        @Override
        @Transactional
        public CounterRsDto create(CounterRqDto counterRqDto) {
            var src = counterMapper.toEntity(counterRqDto);
            var dst = counterRepository.save(src);
            return counterMapper.toDto(dst);
        }

        @Override
        @Transactional
        public Page<CounterRsDto> findAll(Pageable pageable) {
            return counterRepository.findAll(pageable)
                    .map(counterMapper::toDto);
        }

        @Override
        @Transactional
        public CounterRsDto findById(@NonNull Long id) {
            return counterRepository.findById(id)
                    .map(counterMapper::toDto)
                    .orElseThrow(()              ->              new
NotFoundException(String.format(COUNTER_IS_NOT_FOUND_BY_ID_MSG,
"id", id)));
        }

        @Override
        @Transactional
```

```
    public Optional<CounterRsDto> findBySerialNumber(@NonNull
String serialNumber) {
        return
counterRepository.findBySerialNumber(serialNumber)
                .map(counterMapper::toDto);
    }

    @Override
    public void deleteCounter(@NonNull Long id) {
        counterRepository.findById(id)
                .ifPresentOrElse(counterRepository::delete,
                        () -> {
                            throw                          new
NotFoundException(String.format(COUNTER_IS_NOT_FOUND_BY_ID_MSG,
"id", id));
                        });
    }

    @Override
    @Transactional
    public void startCounter(@NonNull Long id) {
        Counter                  counter                  =
counterRepository.findById(id).orElseThrow();
        var incrementedIndication = counter.getIndication() +
1;
        counter.setIndication(incrementedIndication);
        counter.setLastIndicationDate(Instant.now());
        counterRepository.save(counter);
    }

    @Override
    @Transactional
    public void changeStatus(@NonNull String serialNumber) {
        Counter                  counter                  =
counterRepository.findBySerialNumber(serialNumber).orElseThrow(
```

```
                        ()                    ->                    new
NotFoundException(String.format(NOT_FOUND_BY_SERIAL_NUMBER,
                                "serialNumber",
                                serialNumber)));

            counter.setIsActive(!counter.getIsActive());
        }

        private CounterDto buildCounterDto(Counter counter) {
            return CounterDto.builder()
                    .serialNumber(counter.getSerialNumber())
                    .indication(counter.getIndication())
                    .lastIndicationDate(counter.getLastIndication
Date())
                    .build();
        }
    }
```

Модуль KafkaService.java

```
    package com.example.quartz_sheduler.service.impl;

    import com.example.quartz_sheduler.entities.Counter;
    import com.example.quartz_sheduler.mapper.CounterMapper;
    import
com.example.quartz_sheduler.repository.CounterRepository;
    import com.example.quartz_sheduler.rest.dto.CounterRqDto;
    import
com.example.quartz_sheduler.rest.dto.kafka_dto.CounterDto;
    import com.example.quartz_sheduler.scheduler.JobData;
    import com.example.quartz_sheduler.service.KafkaService;
    import com.example.quartz_sheduler.service.SchedulerService;
    import com.fasterxml.jackson.core.JsonProcessingException;
    import com.fasterxml.jackson.databind.ObjectMapper;
    import lombok.RequiredArgsConstructor;
    import lombok.extern.slf4j.Slf4j;
```

```
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

import java.util.Random;

@Slf4j
@Service
@RequiredArgsConstructor
public class KafkaServiceImpl implements KafkaService {

    private static final String START = "start";
    private static final String STOP = "stop";

    private final KafkaTemplate<Long, CounterDto>
kafkaTemplate;
    private final ObjectMapper objectMapper;
    private final SchedulerService schedulerService;
    private final CounterRepository counterRepository;
    private final CounterMapper counterMapper;

    @Override
    public void sendResultOfJob(Long id) {
        Counter counter =
counterRepository.findById(id).orElseThrow();
        CounterDto counterDto = CounterDto.builder()
                .serialNumber(counter.getSerialNumber())
                .indication(counter.getIndication())
                .lastIndicationDate(counter.getLastIndication
Date())
                .build();
        kafkaTemplate.send("readings.created", counterDto);
    }

    @Override
```

```
        @KafkaListener(topics = "meter.created", groupId =
"myGroup", containerFactory = "singleFactory")
        public void consumeCreatCounterMessage(CounterRqDto
counterRqDto) {
            log.info("=> consumed {}",
writeValueAsString(counterRqDto));
            var src = counterMapper.toEntity(counterRqDto);
            counterRepository.save(src);
        }

        @Override
        @KafkaListener(topics = "meter.started", groupId =
"myGroup", containerFactory = "singleFactory")
        public void consumeStartCounterMessage(CounterRqDto
counterRqDto) {
            JobData jobData = JobData.builder()
                    .action(START)
                    .serialNumber(counterRqDto.getSerialNumber())
                    .gapDuration(setRandomGapDuration())
                    .build();

            schedulerService.startSchedule(jobData);
        }

        @Override
        @KafkaListener(topics = "meter.stopped", groupId =
"myGroup", containerFactory = "singleFactory")
        public void consumeStopCounterMessage(CounterRqDto
counterRqDto) {
            JobData jobData = JobData.builder()
                    .action(STOP)
                    .serialNumber(counterRqDto.getSerialNumber())
                    .build();
            schedulerService.stopScheduling(jobData);
        }
```

```
        private String writeValueAsString(CounterRqDto dto) {
            try {
                return objectMapper.writeValueAsString(dto);
            } catch (JsonProcessingException e) {
                e.printStackTrace();
                throw new RuntimeException("Writing value to JSON
failed: " + dto.toString());
            }
        }

        private Integer setRandomGapDuration() {
            Random random = new Random();
            return random.nextInt((5 - 1) + 1) + 5;
        }
    }
```

Модуль SchedulerService.java

```
    package com.example.quartz_sheduler.service.impl;

    import com.example.quartz_sheduler.rest.dto.CounterRsDto;
    import
com.example.quartz_sheduler.rest.exception.CounterAlreadyWorkingEx
ception;
    import
com.example.quartz_sheduler.rest.exception.NotFoundException;
    import com.example.quartz_sheduler.scheduler.JobData;
    import com.example.quartz_sheduler.scheduler.ScheduledJob;
    import com.example.quartz_sheduler.service.CounterService;
    import com.example.quartz_sheduler.service.KafkaService;
    import com.example.quartz_sheduler.service.SchedulerService;
    import lombok.RequiredArgsConstructor;
    import lombok.extern.slf4j.Slf4j;
    import org.quartz.*;
    import org.springframework.stereotype.Service;
```

```
import javax.annotation.PreDestroy;
import java.time.LocalDateTime;
import java.time.OffsetDateTime;
import java.util.Date;

@Slf4j
@Service
@RequiredArgsConstructor
public class SchedulerServiceImpl implements SchedulerService
{

    private static final String NOT_FOUND_BY_SERIAL_NUMBER =
"Счетчик с %s = %s не найден";
    private static final String COUNTER_ALREADY_WORKING =
"Счетчик %s = %s уже работает";
    private static final String JOB_NAME = "job name";

    private final Scheduler quartzScheduler;
    private final CounterService counterService;

    @PreDestroy
    public void preDestroy() {
        try {
            quartzScheduler.shutdown();
        } catch (SchedulerException e) {
            log.error("scheduler throws exception" + e);
            e.printStackTrace();
        }
    }

    @Override
    public void startSchedule(JobData data) {
        var                    counter                    =
counterService.findBySerialNumber(data.getSerialNumber())
                .orElseThrow(
```

```
                                () -> new
NotFoundException(String.format(NOT_FOUND_BY_SERIAL_NUMBER,
                                        "serialNumber",
                                        data.getSerialNumber()))
                        );

                if (Boolean.FALSE.equals(counter.getIsActive())) {
                    startJob(data, counter);
                } else {
                    throw new
CounterAlreadyWorkingException(String.format(COUNTER_ALREADY_WORKI
NG,
                            "serialNumber",
                            counter.getSerialNumber()
                    ));
                }
            }

            @Override
            public void stopScheduling(JobData data) {
                try {
                    quartzScheduler.pauseJob(JobKey.jobKey(JOB_NAME,
data.getSerialNumber()));

counterService.changeStatus(data.getSerialNumber());
                } catch (SchedulerException e) {
                    log.error("scheduler throws exception: " + e);
                    e.printStackTrace();
                }
            }

            private void startJob(JobData data, CounterRsDto counter)
{

                JobDataMap dataMap = new JobDataMap();
```

```
            dataMap.put(counter.getSerialNumber(),
counter.getId());

            var      detail      =      createJobDetail(dataMap,
data.getSerialNumber());
            var  trigger  =  createTrigger(data.getGapDuration(),
data.getSerialNumber());

            try {
                if
(!quartzScheduler.checkExists(JobKey.jobKey(JOB_NAME,
data.getSerialNumber()))) {
                    quartzScheduler.start();

counterService.changeStatus(data.getSerialNumber());
                    quartzScheduler.scheduleJob(detail, trigger);
                }
                else {

quartzScheduler.resumeJob(JobKey.jobKey(JOB_NAME,
data.getSerialNumber()));

counterService.changeStatus(data.getSerialNumber());
                }
            } catch (SchedulerException e) {
                log.error("scheduler throws exception: " + e);
                e.printStackTrace();
            }
        }

        private JobDetail createJobDetail(JobDataMap jobDataMap,
String jobGroup) {
            return JobBuilder.newJob(ScheduledJob.class)
                    .withIdentity(JOB_NAME, jobGroup)
                    .usingJobData(jobDataMap)
                    .build();
```

```
        }

        private Trigger createTrigger(int gapDuration, String
jobGroup) {
            return
TriggerBuilder.newTrigger().withIdentity(JOB_NAME, jobGroup)
                    .startAt(Date.from(LocalDateTime.now().toInst
ant(OffsetDateTime.now().getOffset())))
                    .withSchedule(SimpleScheduleBuilder.simpleSch
edule()
                            .withIntervalInSeconds(gapDuration)
                            .repeatForever())
                    .build();
        }
    }
```

Модуль EventAnalysisController.java

```
    package com.example.test.controller;

    import com.example.test.controller.dto.AnalysisDto;
    import com.example.test.controller.dto.CheckDto;
    import com.example.test.controller.dto.DeltaDto;
    import com.example.test.service.EventAnalysisService;
    import lombok.extern.slf4j.Slf4j;
    import org.springframework.beans.factory.annotation.Autowired;
    import org.springframework.web.bind.annotation.RestController;

    import java.time.LocalDate;
    import java.util.List;

    @RestController
    @Slf4j
    public class EventAnalysisController implements
EventAnalysisAPI {
```

```
        private final EventAnalysisService eventAnalysisService;

        @Autowired
        public       EventAnalysisController(EventAnalysisService
eventAnalysisService) {
            this.eventAnalysisService = eventAnalysisService;
        }

        @Override
        public CheckDto getSum(String deviceId, LocalDate startAt,
LocalDate finishAt) {
            log.info("GET /sum id=" + deviceId);

            return eventAnalysisService.getSum(deviceId, startAt,
finishAt);
        }

        @Override
        public AnalysisDto getAnalysis(String deviceId, LocalDate
startAt, LocalDate finishAt) {
            log.info("GET /analysis id=" + deviceId);

            return     eventAnalysisService.getAnalysis(deviceId,
startAt, finishAt);
        }

        @Override
        public CheckDto estimateMonth(String deviceId) {
            log.info("GET /estimate id=" + deviceId);

            return eventAnalysisService.estimateMonth(deviceId);
        }

        @Override
        public  List<DeltaDto>  getGraphicData(String  deviceId,
LocalDate startAt, LocalDate finishAt) {
```

```
            log.info("GET /graphic id=" + deviceId);

            return  eventAnalysisService.getGraphicData(deviceId,
startAt, finishAt);
        }
    }
```

Модуль EventController.java

```
    package com.example.test.controller;

    import com.example.test.controller.dto.EventDto;
    import com.example.test.entity.EventType;
    import com.example.test.service.EventService;
    import org.springframework.beans.factory.annotation.Autowired;
    import org.springframework.data.domain.Page;
    import org.springframework.data.domain.Pageable;
    import org.springframework.web.bind.annotation.RestController;

    import java.time.LocalDate;

    @RestController
    public class EventController implements EventAPI {

        private final EventService eventService;

        @Autowired
        public EventController(EventService eventService) {
            this.eventService = eventService;
        }

        @Override
        public Page<EventDto> getAll(String deviceId, LocalDate
startAt, LocalDate finishAt, EventType eventType, Pageable
pageable) {
```

```
            return eventService.getAllByFilter(deviceId, startAt,
finishAt, eventType, pageable);
        }
    }
```

Модуль ReportController.java

```
package com.example.test.controller;

import com.example.test.controller.dto.ReportType;
import com.example.test.service.ReportService;
import io.swagger.v3.oas.annotations.tags.Tag;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.FileSystemResource;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
@Slf4j
@RequestMapping("/reports")
@Tag(name = "Отчеты")
public class ReportController {

    private final ReportService reportService;

    @Autowired
    public ReportController(ReportService reportService) {
        this.reportService = reportService;
    }
```

```
        @GetMapping(value = "/{meterId}")
        public String getMeterReport(Model model, @PathVariable
String meterId) {
            log.info("GET /report/{id}: id={}", meterId);

            reportService.getReportToday(model, meterId);

            return "report";
        }

        @GetMapping(value = "/{meterId}/all")
        @ResponseBody
        public ResponseEntity<FileSystemResource>
getMeterReportAll(@PathVariable String meterId, ReportType
reportType) {
            log.info("GET /report/{id}/all: id={}, reportType={}",
meterId, reportType.name());

            var file = reportService.getReportFull(meterId,
reportType);
            var reportName = "report-" + meterId +
(reportType.equals(ReportType.CSV) ? ".csv" : ".pdf");

            return ResponseEntity.ok()
                    .header("Content-Disposition", "attachment;
filename=" + reportName)
                    .contentLength(file.length())
                    .contentType(MediaType.parseMediaType((report
Type.equals(ReportType.CSV) ?
                            "text/csv;charset=UTF-8" :
"application/pdf")))
                    .body(new FileSystemResource(file));
        }

        @GetMapping(value = "/locations/{locationId}")
```

```
        public String getLocationInfo(Model model, @PathVariable
Long locationId) {
            log.info("GET /report/locations id=" + locationId);

            reportService.getLocationReport(model, locationId);

            return "location";
        }

        @GetMapping("/sum/{meterId}")
        @ResponseBody
        public String getSumByDay(@PathVariable String meterId) {
            log.info("GET /report/sum id=" + meterId);

            return reportService.getSumToday(meterId);
        }

    }
    import { useState } from 'react';
    import * as yup from 'yup';

    import      {      useAddMeterTypeMutation      }      from
'../requests/requests';

    const schema = yup.object().shape({
        newType: yup.string().required(),
    });

    type UseAddDevicesInfoProps = {
        refetch: () => void;
    };

    const     useAddDevicesInfo     =     ({     refetch     }:
UseAddDevicesInfoProps) => {
        const [newType, setNewType] = useState('');
        const [newTypeError, setNewTypeError] = useState(false);
```

```
    const [trigger] = useAddMeterTypeMutation();

    const [visible, setVisible] = useState(false);
    const handleChangeVisible = () => {
        setVisible(true);
    };

    const handleChangeNewType = (
        event: React.ChangeEvent<HTMLInputElement>
    ) => {
        setNewType(event.target.value);
    };

    const handleAddNewType = () => {
        setNewTypeError(false);
        schema
            .validate(
                {
                    newType,
                },
                { abortEarly: false }
            )
            .then(async (valid) => {
                if (valid) {
                    await trigger(newType);
                    refetch();
                    setVisible(false);
                    setNewType('');
                }
            })
            .catch((err: any) => {
                Object?.values(err.errors).forEach((v:   any)
=> {
                    switch (v.split(' ')[0]) {
                        case 'newType':
```

```
                                setNewTypeError(true);
                                break;
                            default:
                                break;
                        }
                    });
                });
        };

        const handleCancelAddType = () => {
            setVisible(false);
            setNewTypeError(false);
        };

        return {
            newType,
            visible,
            newTypeError,
            handleCancelAddType,
            handleAddNewType,
            handleChangeNewType,
            handleChangeVisible,
        };
    };

    export default useAddDevicesInfo;
    import { AddInputInfo } from
'components/molecules/AddInputInfo/UI/AddInputInfo';

    import styles from './AddObjectInfo.module.css';

    type AddObjectInfoProps = {
        city: string;
        street: string;
        house: string;
        flat: string;
```

```
        handleChangeCity:                                    (event:
React.ChangeEvent<HTMLInputElement>) => void;
        handleChangeStreet:                                  (event:
React.ChangeEvent<HTMLInputElement>) => void;
        handleChangeNumberHouse: (
            event: React.ChangeEvent<HTMLInputElement>
        ) => void;
        handleChangeNumberFlat: (
            event: React.ChangeEvent<HTMLInputElement>
        ) => void;
        houseError?: boolean;
        cityError?: boolean;
        streetError?: boolean;
    };

    export function AddObjectInfo({
        city,
        street,
        house,
        flat,
        handleChangeCity,
        handleChangeStreet,
        handleChangeNumberHouse,
        handleChangeNumberFlat,
        houseError,
        cityError,
        streetError,
    }: AddObjectInfoProps) {
        return (
            <div className={styles.Card}>
                <div className={styles.CardBlock}>
                    <AddInputInfo
                        text="Город *"
                        value={city}
                        onChange={handleChangeCity}
                        error={cityError}
```

```
                    />
                    <AddInputInfo
                        text="Улица *"
                        value={street}
                        onChange={handleChangeStreet}
                        error={streetError}
                    />
                </div>
                <div className={styles.CardBlock}>
                    <AddInputInfo
                        text="Номер дома *"
                        value={house}
                        onChange={handleChangeNumberHouse}
                        error={houseError}
                    />
                    <AddInputInfo
                        text="Номер квартиры"
                        value={flat || ''}
                        onChange={handleChangeNumberFlat}
                        type="number"
                    />
                </div>
            </div>
        );
    }

    AddObjectInfo.defaultProps = {
        houseError: false,
        cityError: false,
        streetError: false,
    };

    import { Button } from '@mui/material';
    import { DeviceInfoCard } from
'components/organisms/DeviceInfoCard/UI/DeviceInfoCard';
    import { Link } from 'react-router-dom';
```

```
import styles from './AllDevicesCards.module.css';

interface IDevice {
    id: string;
    serialNumber: string;
    type: {
        id: number;
        title: string;
    };
    installationDate: string;
    isActive: boolean;
    currentReading: number;
}

interface IAllDevicesCardsProps {
    currentId: string;
    devices: Array<IDevice>;
}

export function AllDevicesCards({ currentId, devices }:
IAllDevicesCardsProps) {
    return (
        <div className={styles.DeviceInfoCard}>
            <div className={styles.ButtonDeviceCard}>
                <Link
to={`/objects/${currentId}/add_device`}>
                    <Button
variant="contained">Добавить</Button>
                </Link>
            </div>
            <div>
                {devices?.map((el) => (
                    <Link

to={`/objects/${currentId}/device/${el.id}`}
```

```
                            key={`DevicePage${el.id}`}
                        >
                            <DeviceInfoCard
                                title={String(el.type.title)}
                                key={el.id}
                                serialNumber={el.serialNumber}
                                date={el.installationDate}
                                status={el.isActive}
                                indication={el.currentReading}
                            />
                        </Link>
                    ))}
                </div>
            </div>
        );
    }
    import { Button } from '@mui/material';
    import { Titles } from 'components/atoms/Titles/UI/Titles';
    import { AddDatePickerInfo } from
'components/molecules/AddDatePickerInfo/UI/AddDatePickerInfo';

    import useChartCheck from '../logic/useChartCheck';
    import styles from './ChartCheck.module.css';

    export function ChartCheck() {
        const {
            fromDateError,
            toDateError,
            fromDate,
            toDate,
            chartRef,
            handleChangeFromDate,
            handleChangeToDate,
            handleGetParamsForChart,
        } = useChartCheck();
        return (
```

```
            <div className={styles.wrapper}>
                <Titles text="График расходов"
typographyType="cardTitle" />
                <div className={styles.wrapperDatePickers}>
                    <div className={styles.wrapperPicker}>
                        <AddDatePickerInfo
                            error={fromDateError}
                            text="От"
                            type="DatePicker"
                            value={fromDate}
                            onChange={handleChangeFromDate}
                        />
                    </div>
                    <div className={styles.wrapperPicker}>
                        <AddDatePickerInfo
                            error={toDateError}
                            text="До"
                            type="DatePicker"
                            value={toDate}
                            onChange={handleChangeToDate}
                            minDate={fromDate}
                        />
                    </div>
                </div>
                <Button variant="contained"
onClick={handleGetParamsForChart}>
                    Получить
                </Button>
                <canvas ref={chartRef} />
            </div>
        );
    }
    import { AddInputInfo } from
'components/molecules/AddInputInfo/UI/AddInputInfo';
    import { AddSelectInfo } from
'components/molecules/AddSelectInfo/UI/AddSelectInfo';
```

```
    import dayjs from 'dayjs';

    import checkImg from '../assets/check3.png';
    import changeImg from '../assets/exchange.png';
    import                useCurrentDeviceInfo                from
'../logic/useCurrentDeviceInfo';
    import styles from './CurrentDeviceInfo.module.css';

    export function CurrentDeviceInfo() {
        const {
            data,
            result,
            active,
            isActiveStatus,
            disableImg,
            handleChangeActiveStatus,
            handleChangeDisabledStatus,
        } = useCurrentDeviceInfo();
        return (
            <div className={styles.Card}>
                <div className={styles.CardBlock}>
                    <AddInputInfo
                        disabled
                        text="Серийный номер"
                        value={data?.serialNumber ?? ''}
                    />
                    <AddInputInfo
                        disabled
                        text="Дата установки"

value={`${dayjs(data?.installationDate).format(
                                'DD-MM-YYYY HH:mm'
                        )}`}
                    />
                    <AddInputInfo
                        disabled
```

```
                        text="Тип устройства"
                        value={data?.type?.title ?? ''}
                    />
                </div>
                <div classNameName={styles.CardBlock}>
                    <div
classNameName={styles.changeStatusContainer}>
                        <AddSelectInfo
                            disabled={!disableImg}
                            text="Статус работы"
                            value={String(active)}
                            onChange={handleChangeActiveStatus}
                            source={isActiveStatus}
                        />
                        <div
                            onClick={handleChangeDisabledStatus}

onKeyPress={handleChangeDisabledStatus}
                            role="button"
                            tabIndex={0}
                            classNameName={styles.imgContainer}
                        >
                            <img
                                src={disableImg   ?   checkImg   :
changeImg}

                                alt="change"
                                classNameName={styles.img}
                            />
                        </div>
                    </div>
                    <AddInputInfo
                        disabled
                        text="Текущее показание"
                        value={
                            data?.currentReading
                                ? `${data?.currentReading} м³`
```

```
                                    : '0 м³'
                                }
                            />
                            <AddInputInfo
                                disabled
                                text="Предполагаемая сумма по тарифу за
текущий месяц"
                                value={data?.currentReading}
                            />
                        </div>
                    </div>
                );
            }
```

Приложение В

(обязательное)

Сервис мониторинга показаний счетчиков ресурсов

Презентация

Система мониторинга показаний счетчиков ресурсов

Работу выполнил:
студент гр. 18ВОЭ1 Лучинкин Глеб Юрьевич
Научный руководитель:
к.т.н., доцент каф ВТ Дубинин Виктор Николаевич

Пенза, 2023

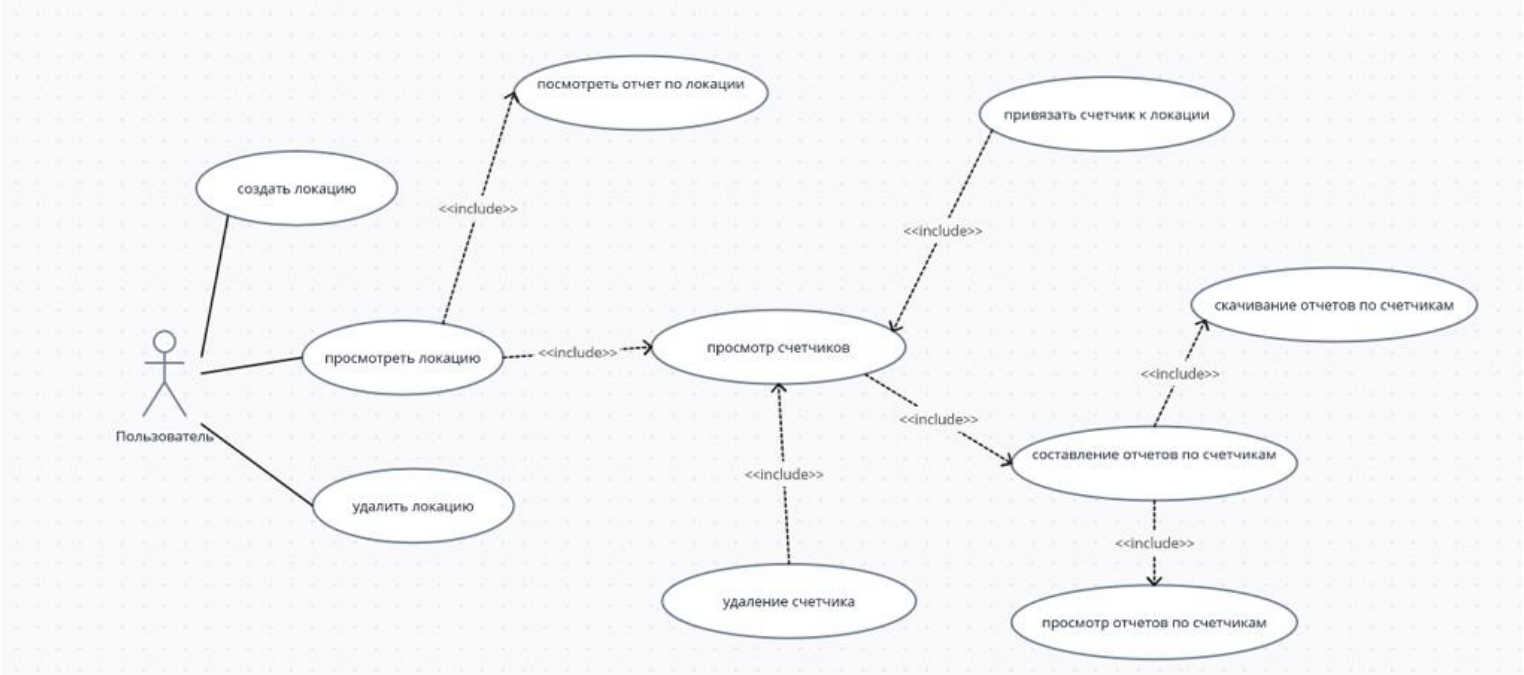
Цель работы

Цель работы - разработать систему мониторинга показаний счетчиков ресурсов.

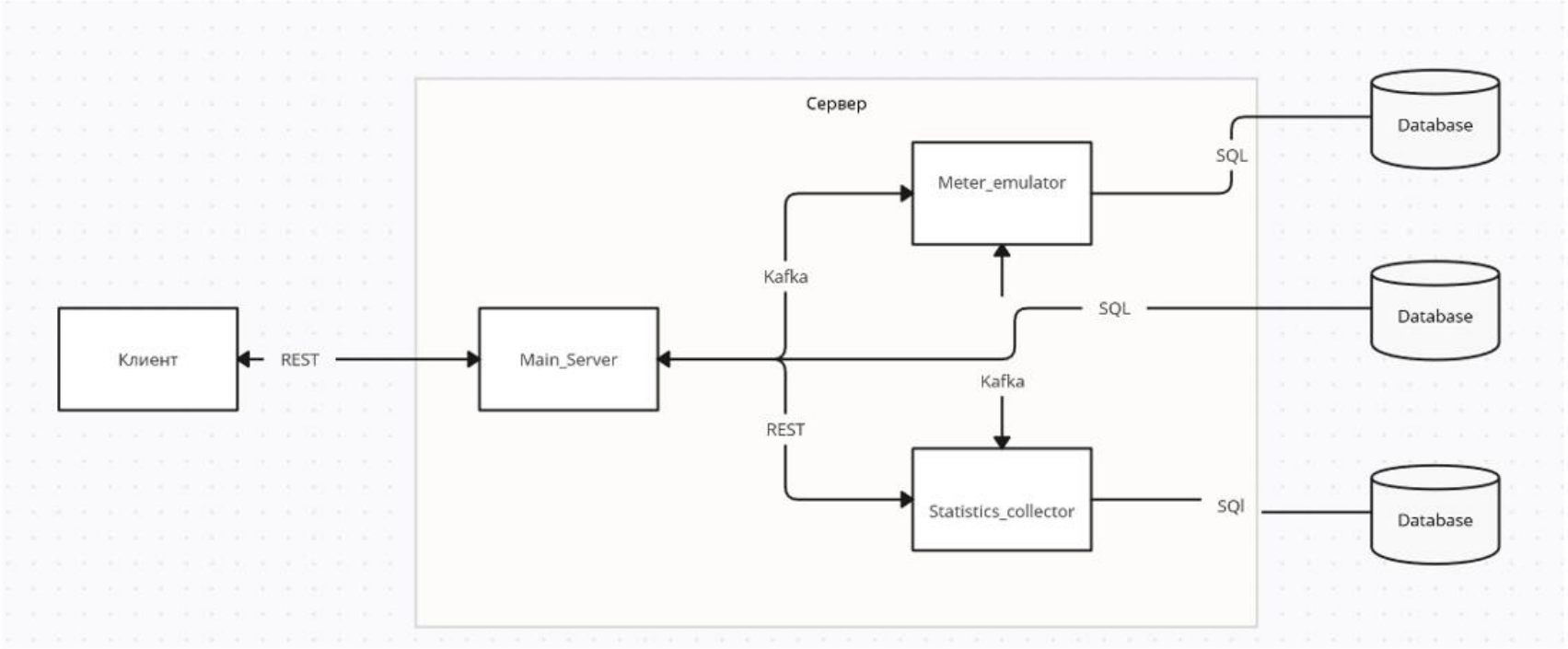
В итоге должно получиться клиент - серверное приложение на базе микросервисной архитектуры, позволяющее регестрировать счетчики на определенной локации, и получать отчеты на основе показаний.

Стек технологий
JS
HTML
CSS
spring
BOOT
docker
HIBERNATE
Java
kafka
Quartz

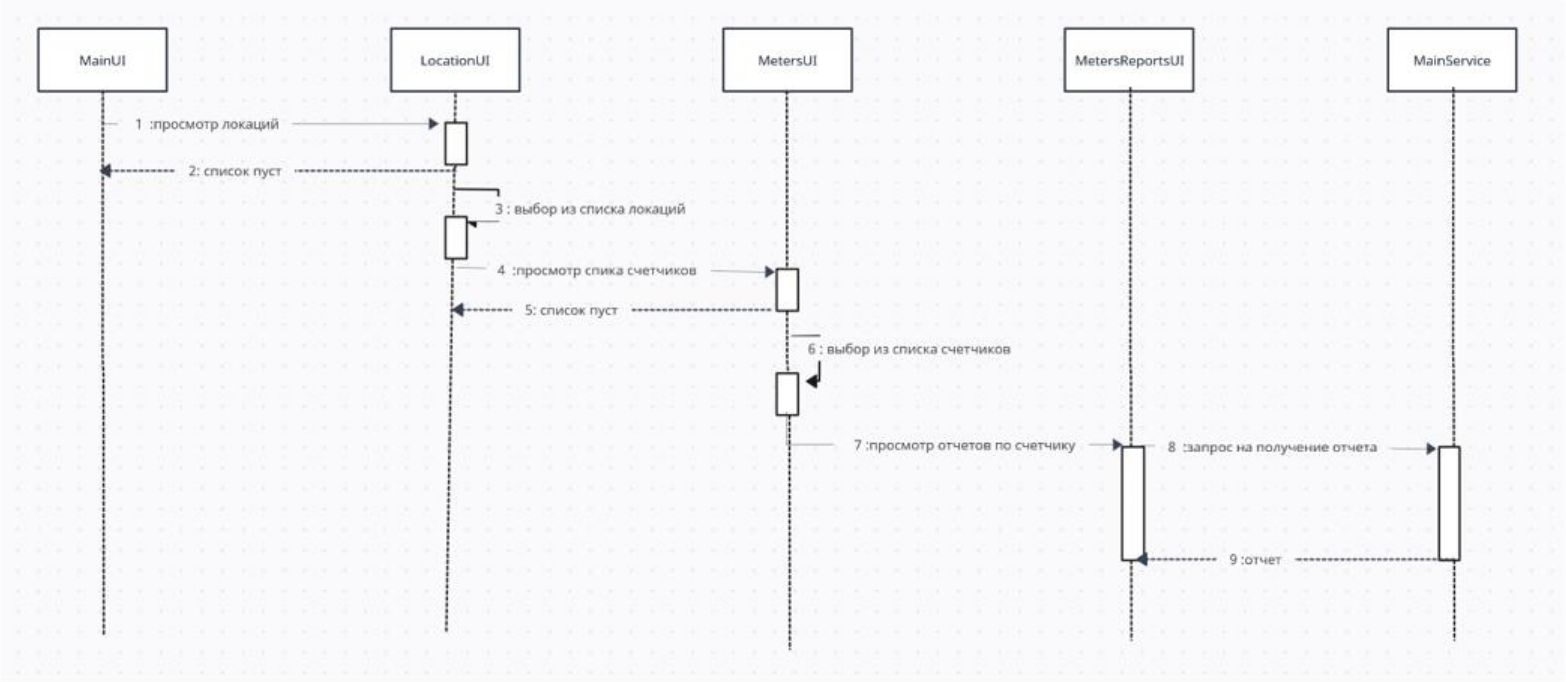
Функции приложения



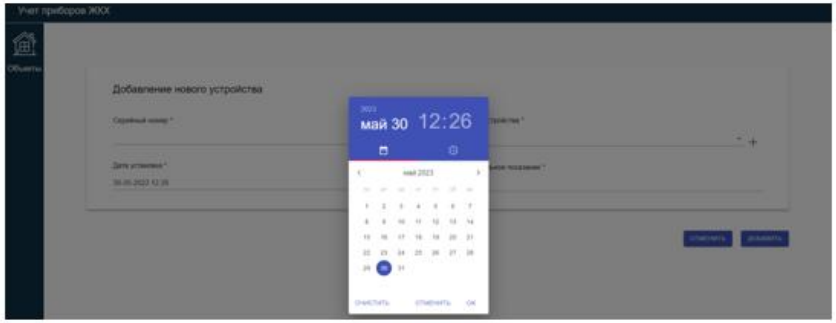
Архитектура приложения

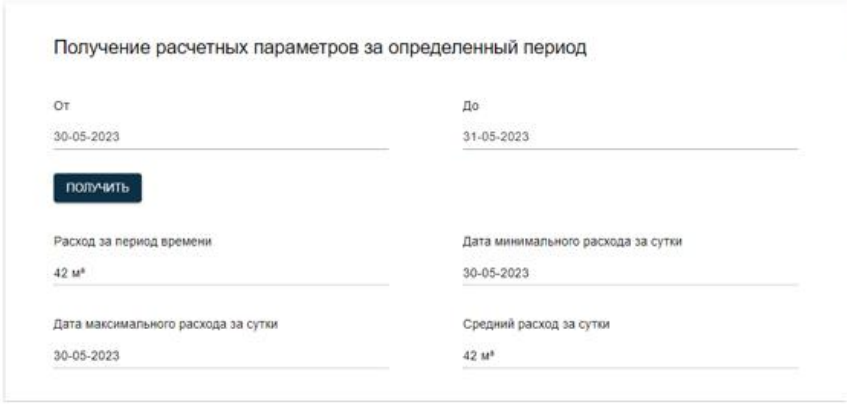


Работа приложения на примере получения отчета



Скриншоты приложения





Заключение

В ходе разработки дипломного проекта была разработана система мониторинга показаний счетчиков ресурсов на языке программирования Java и JavaScript.

Разработанная система позволяет:

- Добавлять данные о локации
- Составлять отчет по локации
- Регестрировать в локации счетчики
- Запускать счетчики
- Генерироваться отчеты по счетчикам