**РАЗРАБОТКА АЛГОРИТМА ОБРАБОТКИ ДАННЫХ**

Алгоритм обработки данных беспроводной системы мониторинга окружающей среды представляет собой модульную конструкцию и включает фоновый процесс для получения данных и отправки их, а также веб-сервис для вывода данных в виде графиков.

**5.1 Разработка фонового процесса для получения и записи данных в базу данных**

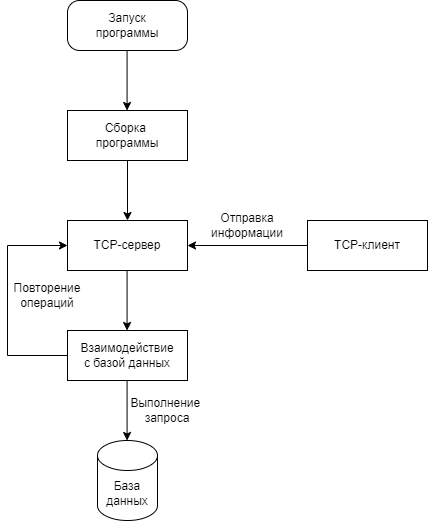


Рисунок 5.1.1 – Схема проектирования фонового процесса

В ходе разработке сборки процесса необходимо предусмотреть смену входных данных для сборки программы, которые позволят менять параметры *TCP*-сервера, менять количество хранимых дынных и строку подключения к в базе данных. Эти параметры позволят гибко настраивать программу и менять настройки программы без вмешательства в код программы.

Одним из важных компонентов программы является *TCP*-сервер, который получает от *TCP*-клиента данные и после передает их системе по занесения их в базу данных. Необходимо спроектировать сервер так, чтобы он проверять существуют ли таблицы или схемы, у которых название будет соответствовать id передатчика, и при необходимости мог создавать их. Реализовать алгоритм внесения данных в базу данных, если лимит параметров в будет превышен.

И последний компонент системы — это взаимодействие с базой данных, использующую технологию *Dapper*. *Dapper* — это легкий и высокопроизводительный микро-*ORM*, разработанный специально для приложений .*NET*. Он обеспечивает простой и эффективный способ сопоставления записей базы данных с объектами .*NET*, позволяя разработчикам взаимодействовать с базами данных с помощью простого и интуитивно понятного кода.

Ключевых преимуществ использования относят:

1. *Dapper* известен своей превосходной производительностью и был создан с учетом быстродействия и обеспечивает минимальные накладные расходы по сравнению с другими *ORM*;
2. Несмотря на то, что *Dapper* предоставляет возможности сопоставления по умолчанию, он также позволяет при необходимости настраивать отображение по своему усмотрению. Такая гибкость позволяет разработчикам легко работать со сложными сценариями или нестандартными структурами баз данных.
3. *Dapper* работает с различными поставщиками баз данных, включая *SQL Server*, *MySQL*, *PostgreSQL*, *SQLite* и другие. Это делает его универсальным выбором для проектов, которым требуется поддержка нескольких платформ баз данных.

Основные компоненты по разработке в фоновом режиме, которые вам понадобятся для разработки программы: сборка проекта, *TCP*-сервер и система по взаимодействию с базой данной.

Проектирование программы начнем с рассмотрения начальных данных, которые нам понадобятся для сборки программы. Начальные данные будем хранить в формате *JSON*. Данный текстовый формат легко читается и записывается человеком. Это позволяет разработчикам и пользователям создавать, редактировать и просматривать файлы входных данных напрямую. Данные в формате *JSON* могут быть легко проанализированы языками программирования с использованием встроенных или сторонних библиотек. *JSON* поддерживает сериализацию и десериализацию, позволяя легко преобразовывать входные данные между их представлением в формате *JSON* и собственными структурами данных в программе.

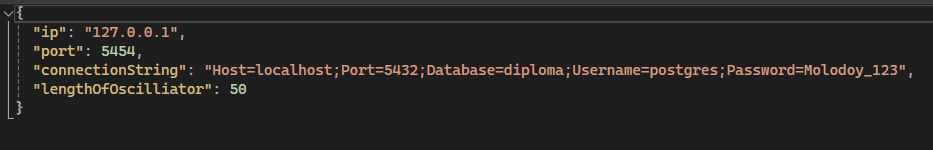


Рисунок 5.1.2 – Данные нужные для

Начальные данные:

* *ip* – это *ip*-адресс, к которому привязан *TCP*-сервер
* *port* - номер порта *TCP*-сервера, числовой идентификатор, который помогает различать различные службы, работающие на одном компьютере
* *connectionString* - строка подключения используется для установления соединения с базовой системой баз данных. Она содержит всю необходимую информацию для подключения Dapper к базе данных, включая адрес сервера, учетные данные для аутентификации, имя базы данных и другие параметры подключения.
* *lengthOfOscilliator* – числовое значение, которое определяет сколько данных может храниться в базе данных.

После запуска, программа считывает данные с файла *config.json* и записывает их в класс *Config*, в котором до конца выполнения программы будут храниться в ней. После эти данные отправляются в класс *Server\_PС*, который представляет *TCP*-сервер.

В *.NET TCP*-сервер — это тип серверного приложения, которое взаимодействует с клиентами по протоколу управления передачей (*TCP*), одному из основных протоколов пакета интернет-протоколов. *TCP* обеспечивает надежную, ориентированную на подключение связь между устройствами в сети. Вот обзор *TCP*-серверов в .*NET*, их преимуществ перед *UDP*, принципов их работы и как устроен *TCP*-сервер:

* Инициализация: на этом этапе вы создается экземпляр прослушивателя *TCP*, указывая *IP*-адрес и номер порта, на котором сервер будет прослушивать входящие соединения.
* Прослушивание: после инициализации вы вызываете метод *Start()* прослушивателя *TCP*, чтобы начать прослушивание входящих подключений по указанному *IP*-адресу и порту.
* Принимайте подключения: как только сервер начинает прослушивание, он переходит в цикл, в котором он непрерывно принимает входящие соединения асинхронно. Когда клиент пытается установить соединение, вызывается метод *AcceptTcpClientAsync()*, который блокирует его до тех пор, пока соединение не будет установлено.
* Обработка соединений: после установления соединения вы обычно создаете новый поток или используете асинхронные методы для обработки взаимодействия с клиентом. Это включает в себя считывание данных с клиента, их обработку и отправку ответов обратно.
* Выключение: когда сервер необходимо выключить, вы вызываете метод *Stop()* прослушивателя *TCP*, чтобы прекратить прослушивание новых подключений. Затем вы закрываете все существующие клиентские подключения и освобождаете все ресурсы, связанные с сервером.

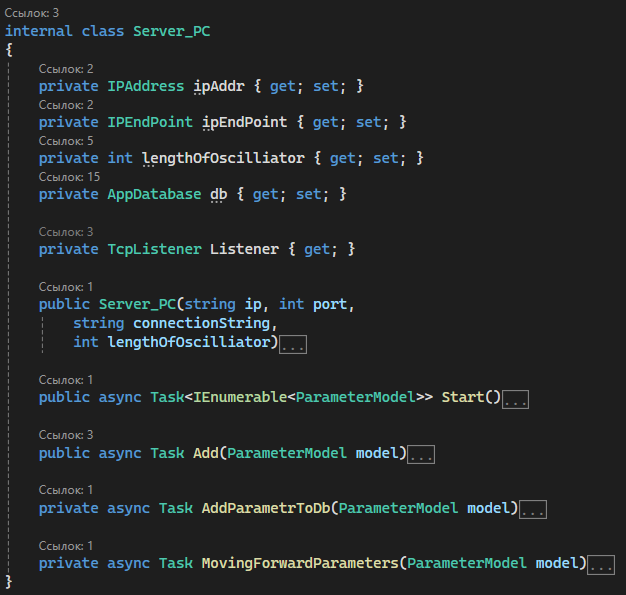


Рисунок 5.1.3 – Преставление класса реализующий *TCP*-сервер

При инициализации экземпляра данного класса свойствам, необходимые для настройки *TCP*-сервера, присвоения пределов хранения данных в базе данных и строка, присваивается значения, которые парсятся из файла *config.json*.

Данный класс состоит из приватных свойств, которые используются для настройки *TCP*-сервера, доступа к базе данных, которая реализована в классе *AppDatabase*, и *Listener* выполняет роль сервера. Он прослушивает входящие подключения по определенному порту. Стоит отметить, что класс *TcpListener* построен на основе класса сокета *Socket*, то есть фактически представляет обертку над классом Socket и позволяет упростить написание некоторых вещей, по сравнению с использованием чистых сокетов.

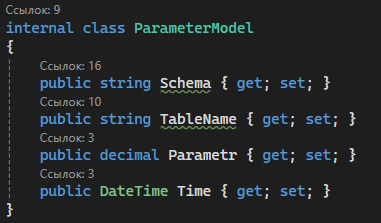


Рисунок 5.1.4 - Класс, представляющий модель для

передачи данных в систему взаимодействия с базой

данных

Основной функционал реализован в асинхронных методах, которые описывают свои основные задачи:

* *Start()* – метод, получающий и возвращающий список классов *ParameterModel*, хранящий данные, полученные от *TCP*-клиента.
* *Add(ParameterModel)* – метод, в котором реализован алгоритм проверки наличия в базе данных схемы датчика и таблиц параметров мониторинга окружающей среды, а также в этом методе реализовано ограничение на количество записей в базе данных
* *AddParametrToDb(ParameterModel)* – метод, который записывает данные в базу данных
* *MovingForwardParameters(ParameterModel)* – метод, который реализует алгоритм записи данных в базу данных, если количество записей превысило ограничение на записи

Использование асинхронных методов открывают ряд преимуществ в сравнении с стандартными методами, а именно:

1. Масштабируемость: асинхронное программирование позволяет приложению обрабатывать большее количество одновременных операций, не блокируя потоки. Это особенно важно для серверных приложений, где масштабируемость необходима для одновременной работы с большим количеством клиентов.
2. Оптимизированное использование ресурсов: избегая блокировки потоков, асинхронные методы позволяют освобождать потоки и возвращать их в пул потоков в ожидании завершения операций ввода-вывода. Это освобождает системные ресурсы, такие как память и центральный процессор, для других задач, что приводит к более эффективному использованию ресурсов.
3. Повышение производительности: асинхронное программирование может привести к повышению производительности, особенно в сценариях, где преобладают операции, связанные с вводом-выводом. Совмещая операции ввода-вывода с другими задачами, ваше приложение может лучше использовать доступные ресурсы и сократить общее время выполнения.

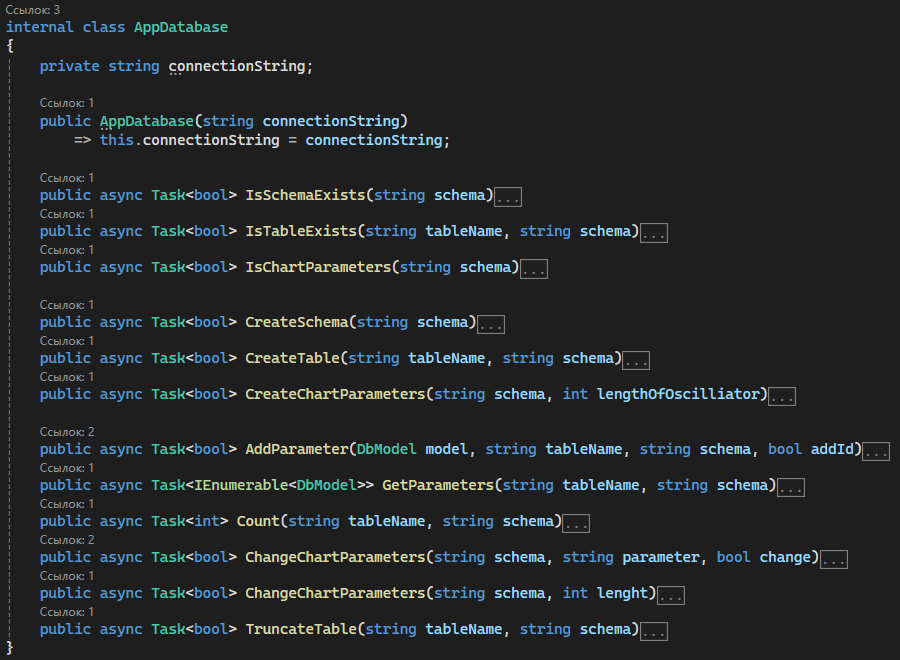


Рисунок 5.1.5 – Преставление класса реализующее взаимодействие с

базой данных

При инициализации экземпляра данного класса свойству строке подключения присваивается значения, которое передается конструктору из файла *config.json*. Строка подключения необходима для подключения к базе данных.

В данном классе используются асинхронные методы, преимущества которых были рассмотрены выше. Также для безопасности и проверки выполнения SQL запросов методы возвращают булевы значения. Все методы можно разделить на три вида: методы проверяющие наличие определенных объектов в базе данных, методы создающие эти объекты и методы, взаимодействующие с данными этих объектов. Рассмотрим все эти виды по отдельности.

Методы проверяющие наличие определенных объектов в базе данных, необходимы для проверки наличия схем, таблиц и свойств параметров. Они были добавлены с целью будущей модернизации программы, если появятся новые параметры мониторинга окружающей среды, и для гибкости выполнения программы. При добавлении нового датчика к ЭВМ не понадобится создавать вручную таблицы и схемы в базе данных, процесс выполнит все сам.

Методы создающие объекты в базе данных были добавлены в дополнение к методам проверяющие наличие определенных объектов в базе данных. После обнаружения отсутствия схем, таблиц в базе данных, процесс создаст их.

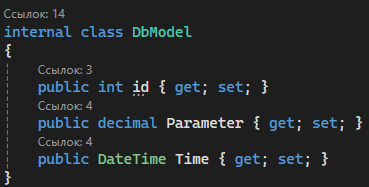


Рисунок 5.1.6 – Класс, представляющий модель объекта

в базе данных

Методы, взаимодействующие с данными этих объектов:

* *AddParameter(DbModel, string, string, bool)* – метод, добавляющий сущность в базу данных по определенную схему в указанную таблицу.
* *GetParameters(string, string)* – метод, возвращающий список сущностей базы данных из определенной схемы и таблицы
* *Count(string, string)* – метод, возвращающий количество записей в базе данных из определенной схемы и таблицы. Метод возвращает целочисленное значение, необходимое для установки ограничения на количество записей
* *ChangeChartParameters(string, string, bool)/ChangeChartParameters(string, int)* – метод, реализован в двух вариантах для выполнения следующих задач: изменения булевых параметров и изменения ограничения записей в базе данных. Первый вариант предназначен для проверки, заполняется ли база новыми параметрами мониторинга окружающей среды, а второй вариант, для перезаписи ограничения на записи.
* *TruncateTable(string, string)* – метод предназначен для очистки определенной таблицы в указанной схеме

**5.2 Рассмотрение структуры базы данных**

В проекте будет использована такая база данных, как *PostgreSql*. *PostgreSQL* — это мощная система управления реляционными базами данных (СУБД) с открытым исходным кодом, известная своей надежностью и обширным набором функций. Вот некоторые ключевые характеристики и возможности *PostgreSQL*:

1. Открытый исходный код: *PostgreSQL* выпускается под лицензией *PostgreSQL*, разрешительной лицензией с открытым исходным кодом, которая позволяет пользователям свободно использовать, изменять и распространять программное обеспечение.
2. Реляционная база данных: *PostgreSQL* — это система реляционных баз данных, которая означает, что она организует данные в таблицы со строками и столбцами и поддерживает *SQL* (язык структурированных запросов) для запроса данных и манипулирования ими.
3. Соответствие требованиям *ACID*: *PostgreSQL* придерживается принципов *ACID* (атомарность, согласованность, изоляция, долговечность), обеспечивая целостность и надежность транзакций даже в сложных многопользовательских средах.
4. Типы данных: *PostgreSQL* предлагает богатый набор встроенных типов данных, включая числовые, текстовые, дату/время, логические, *JSON*, XML, геометрические, сетевой адрес и другие. Кроме того, он поддерживает пользовательские типы данных и составные типы.
5. Расширенные возможности: *PostgreSQL* включает в себя множество расширенных функций, таких как полнотекстовый поиск, индексация, ограничения внешнего ключа, триггеры, представления, хранимые процедуры, репликация, секционирование и поддержка сложных типов данных (массивы, *JSON* и т.д.).
6. Масштабируемость: *PostgreSQL* предназначен для масштабирования от небольших развертываний на одной машине до крупных распределенных корпоративных сред.
7. Кроссплатформенность: *PostgreSQL* является кроссплатформенным и работает в различных операционных системах, включая *Linux*, *Windows*, *macOS* и *Unix*-подобные системы, что делает его подходящим для широкого спектра сред.

*PostgreSQL* — это универсальная и многофункциональная система баз данных, которая хорошо подходит для широкого спектра приложений, от небольших проектов до крупномасштабных корпоративных внедрений в различных отраслях промышленности.

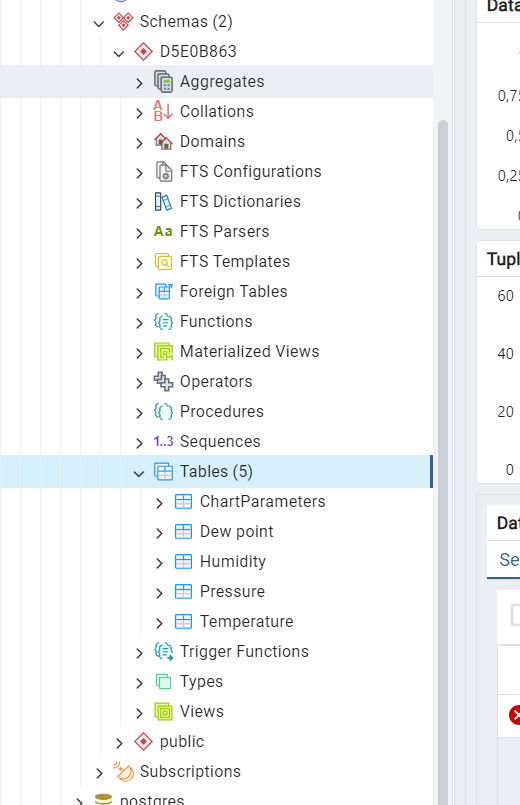


Рисунок 5.2.1 – Представление базы данных, когда

используется один датчик

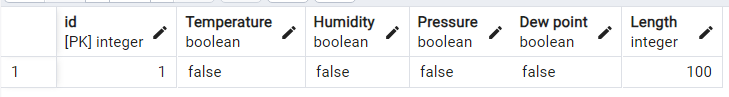


Рисунок 5.2.2 – Представление таблицы свойств графиков

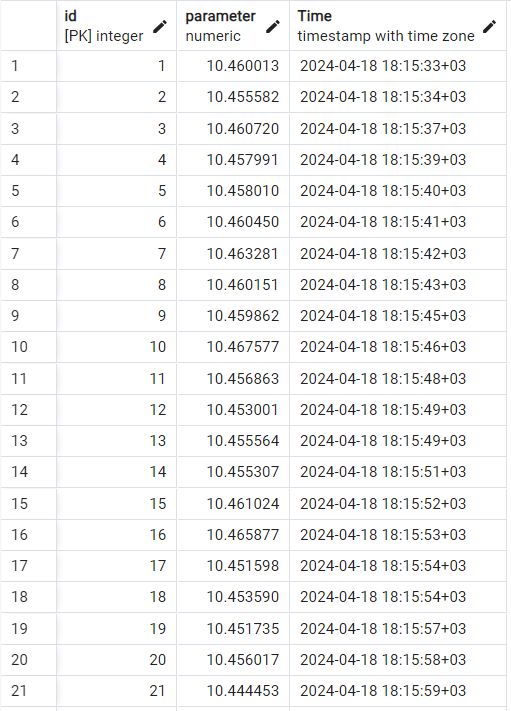


Рисунок 5.2.3 – Представление таблицы точка росы

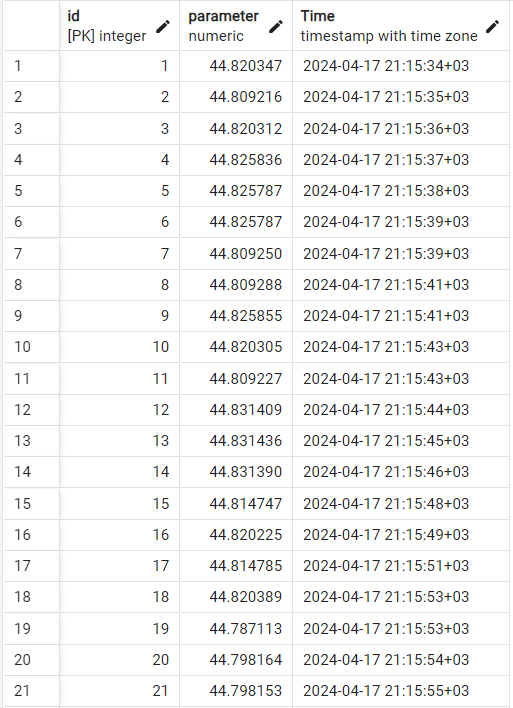


Рисунок 5.2.4 – Представление таблицы влажности

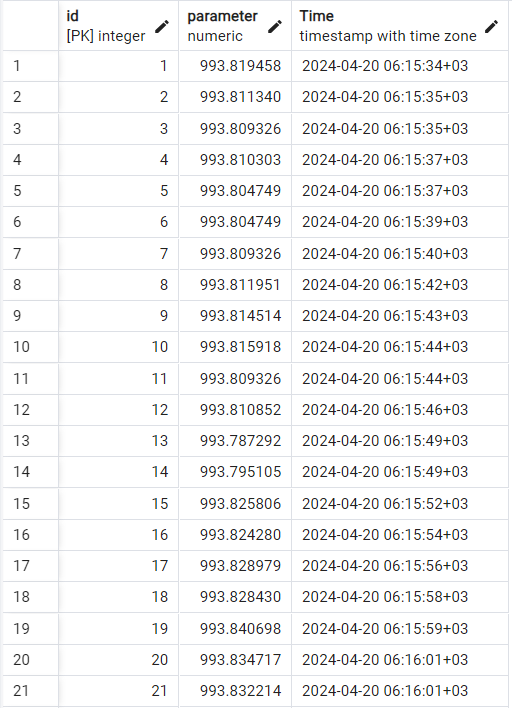


Рисунок 5.2.5 – Представление таблицы давления

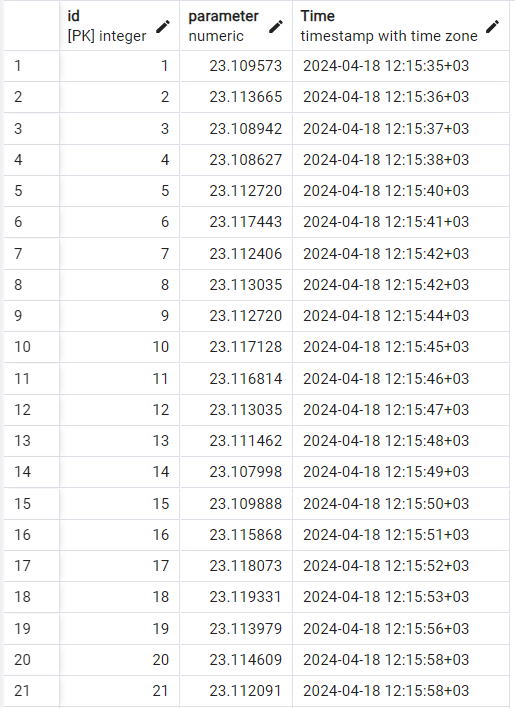


Рисунок 5.2.6 – Представление таблицы температуры

**5.3 Разработка веб-сервиса представления данных**

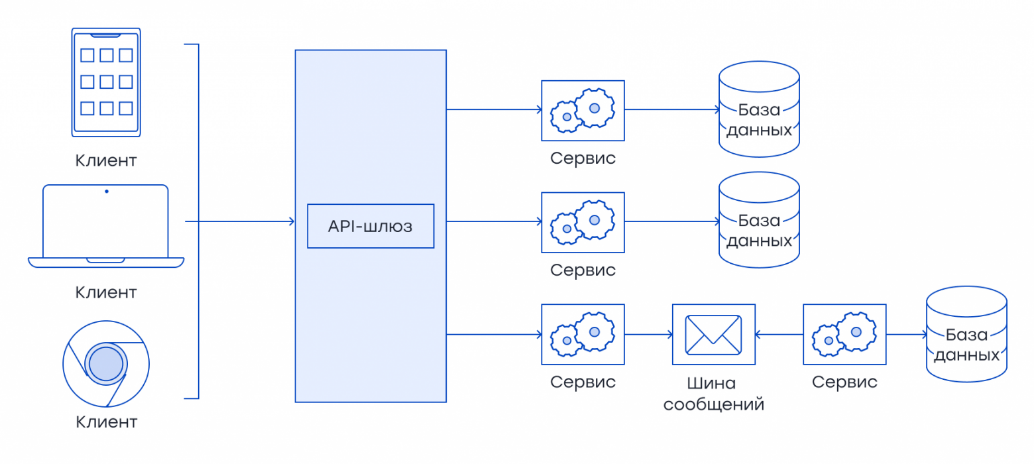


Рисунок 5.3.1 – Архитектура микросервисов

Архитектура микросервисов — это подход к разработке программного обеспечения, при котором одно приложение создается как набор слабо связанных сервисов, каждый из которых представляет определенную бизнес-функцию или возможность. В контексте создания веб-сайтов архитектура микросервисов обладает рядом преимуществ и соображений:

1. Масштабируемость: микросервисы позволяют масштабировать каждый сервис независимо друг от друга, что означает, что вы можете масштабировать только те компоненты, которые требуют дополнительных ресурсов, а не все монолитное приложение. Это особенно полезно для веб-сайтов с различным уровнем трафика или спросом на различные функции.
2. Модульность и гибкость: при использовании микросервисов каждый сервис представляет собой отдельную кодовую базу, часто разрабатываемую и поддерживаемую отдельной командой. Такая модульность позволяет разработчикам работать с меньшими по размеру и более управляемыми кодовыми базами, что упрощает обслуживание, обновления и изменения.
3. Разнообразие технологий: Архитектура микросервисов обеспечивает разнообразие технологий, что означает, что вы можете выбрать наиболее подходящие языки программирования, платформы и базы данных для каждой службы на основе ее конкретных требований. Такая гибкость позволяет вам использовать лучшие инструменты для выполнения каждой задачи и избегать ограничений, связанных с единым технологическим стеком.
4. Автономия и децентрализация: микросервисы способствуют автономии и децентрализации, позволяя командам независимо владеть своими сервисами и управлять ими. Это уменьшает зависимость между командами и ускоряет циклы разработки, поскольку изменения могут вноситься в отдельные сервисы, не затрагивая всего приложения.
5. Устойчивость и изоляция сбоев: поскольку микросервисы разделены и взаимодействуют с помощью упрощенных протоколов, таких как *HTTP* или очереди обмена сообщениями, сбои в работе одной службы с меньшей вероятностью повлияют на работу всего приложения. Изоляция неисправностей позволяет локализовать неполадки в рамках одной службы, сводя к минимуму время простоя и повышая общую устойчивость.
6. Повышенная производительность: микросервисы позволяют повысить производительность за счет параллельной разработки и развертывания сервисов, сокращая время вывода на рынок новых функций и обновлений. Кроме того, сервисы можно оптимизировать по отдельности с точки зрения производительности, масштабируемости и использования ресурсов.

Однако важно отметить, что архитектура микросервисов также сопряжена с такими проблемами, как повышенная сложность развертывания, мониторинга и координации между службами, а также потенциальные проблемы с согласованностью данных и накладными расходами на передачу данных.

Данный сервис будет состоять из двух микросервисов. Первый представляет из себя систему с доступом к базе данных параметров, а вторая система уже сам сайт с визуальным интерфейсом, сервисами для обращения к другим микросервисам, динамеческое обновление элементов страницы и место хранения javascript библиотек.

Рассмотрим структуру приложения первого микросервиса.

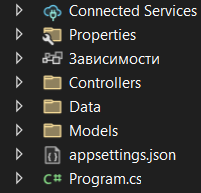


Рисунок 5.3.2 – Структура приложения первого микросервиса

* *Properties*: узел, который содержит некоторые настройки проекта. В частности, в файле *launchSettings.json* описаны настройки запуска проекта, например, адреса, по которым будет запускаться приложение.
* *Controllers*: центральный компонент, который обеспечивает связь между пользователем и приложением, представлением и хранилищем данных. Он содержит логику обработки запроса пользователя. Контроллер получает вводимые пользователем данные и обрабатывает их.
* *Data*: размещаются специфичные классы для работы с разными технологиями доступа к данным. Здесь также хранятся репозитории, через которые уровень бизнес-логики взаимодействует с базой данных.
* *Models*: описывает используемые в приложении данные, а также логику, которая связана непосредственно с данными, например, логику валидации данных.
* *Program.cs*: главный файл приложения, с которого и начинается его выполнение. Код этого файла настраивает и запускает веб-приложение

Рассмотрим ближе основные компоненты этого микросервиса.

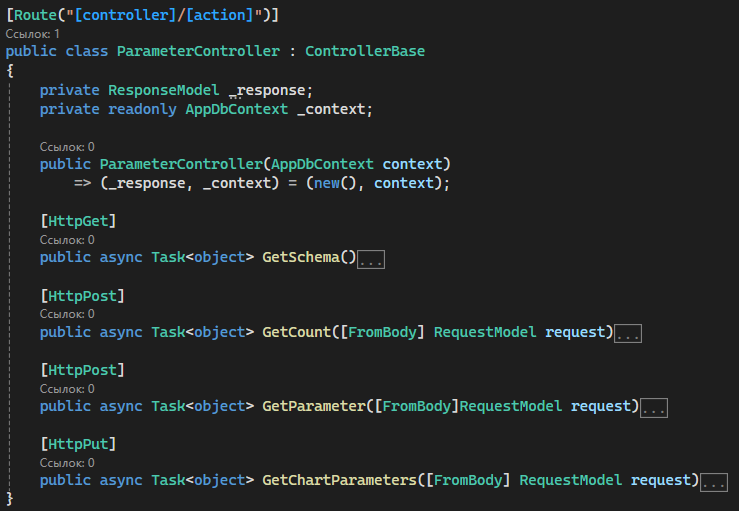


Рисунок 5.3.3 – Класс представляющий контроллер

микросервиса

Этот контроллер имеет два приватных поля: *\_response* и \_*context*. Первый используется для отправки ответа на запрос, который состоит из свойств: *IsSuccess* – успешно ли завершился запрос; Result – необязательное свойство, отвечающее за отправку данных запроса; *DisplayMessages* и *ErrorMessages* – описывают исключение произошедшее в процессе выполнения запроса. Второе поле представляет класс, реализующий технологию Dapper, взаимодействующий с базой данных.

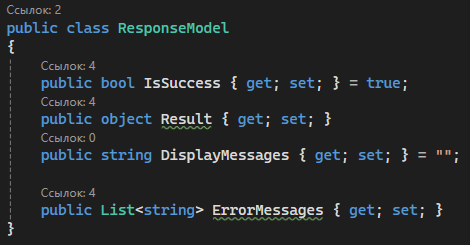


Рисунок 5.3.4 – Класс, представляющий модель

ответа на запрос

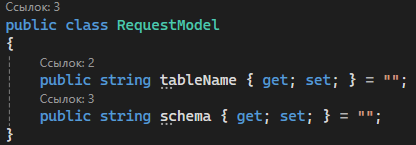


Рисунок 5.3.5 – Класс, представляющий модель

тела запроса

Каждый метод контроллера является асинхронным, что позволит повысить скорость отклика, производительности и масштабируемости. Также методы возвращают объекты, что после облегчит конвертацию в нужный класс.

* *GetSchema()* – возвращает все схемы датчиков расположенных в базе данных
* *GetCount(RequestModel)* – возвращает количество записей в определенной схеме в заданной таблице
* *GetParameter(RequestModel)* – возвращает определенные параметры мониторинга окружающей среды
* *GetChartParameters(RequestModel)* – возвращает свойства определенного датчика

Данный контроллер реализует такие методы запросов как:

* *GET*: запросить некоторый конкретный ресурс.
* *POST*: отправить данные.
* *PUT*: изменения в указанный ресурс по указанному расположению

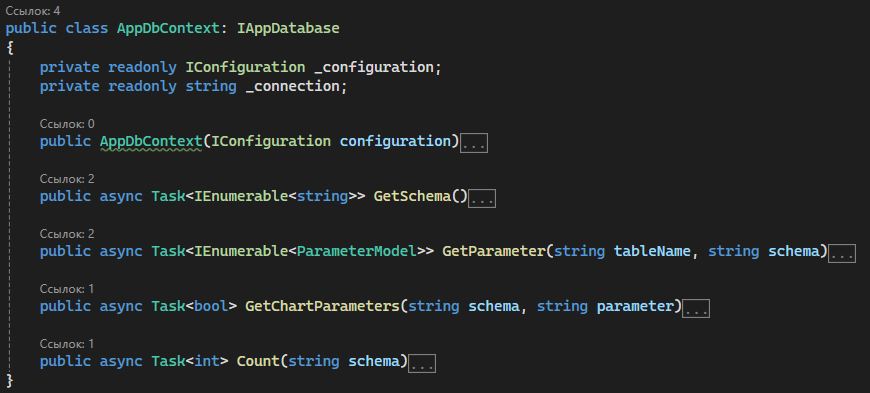


Рисунок 5.3.6 – Класс, реализующий взаимодействия

с базой данных

Данный класс имеет два поля: \_*configuration* и \_*connection*. Первое поле реализует интерфейс базовых настроек приложения, используя который можно получить из файла *appsettings.json* необходимые данные. Второе поле - строка необходимая для подключения к базе данных.

* *GetSchema()* – возвращает список схем датчиков в базе данных
* *GetParameter(string, string)* – возвращает список параметров находящихся в определенной таблице по указанной схеме
* *GetChartParameters(string, string)* – возвращает свойство указанной схемы датчика определенного параметра
* *Count(string)* – возвращает ограничение по записям в базе данных

Перейдем ко второму микросервису, представляющий интерфейс взаимодействия с пользователем. Рассмотрим структуру микросервиса:

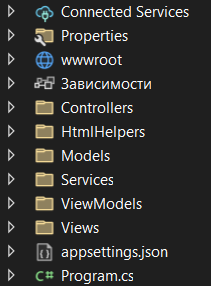


Рисунок 5.3.7 – Структура микросервиса взаимодействующий с пользователем

Данный микросервис реализует *MVC* паттерн, который состоит из трех основных компонентов: *Models*, *Views*, *Controllers*. Это конструктивный шаблон, используемый для разделения пользовательского интерфейса данных и логики приложения. Этот шаблон помогает добиться разделения задач.

Используя шаблон *MVC* для веб-сайтов, запросы направляются к контроллеру, который отвечает за работу с моделью для выполнения действий и/или получения данных. Контроллер выбирает представление для отображения и предоставляет ему модель. Представление отображает окончательную страницу на основе данных в модели.

*Models*: Модель представляет данные и бизнес-логику приложения. Она инкапсулирует данные и поведение приложения, обычно включая операции с базой данных, проверку данных и бизнес-правила. В *ASP.NET MVC* модели часто представлены классами, которые определяют структуру данных и методы взаимодействия с ними.

*Views*: Представление отвечает за представление пользовательского интерфейса пользователям приложения. Оно отображает данные из модели в формате, подходящем для отображения, обычно с использованием *HTML*, *CSS* и клиентских языков сценариев, таких как JavaScript. Представления в *ASP.NET MVC* обычно реализуются с использованием синтаксиса *Razor*, который позволяет разработчикам смешивать серверный код с *HTML* для динамической генерации контента.

*Controllers*: Контроллер действует как посредник между моделью и представлением, обрабатывая вводимые пользователем данные, обрабатывая запросы и соответствующим образом обновляя модель. Контроллеры в *ASP.NET MVC* отвечают за маршрутизацию входящих запросов к соответствующим действиям, выполнение бизнес-логики и выбор подходящего представления для отображения ответа. Контроллеры реализованы в виде классов, содержащих методы действий, каждый из которых соответствует определенной конечной точке запроса.

В папке *wwwroot* располагаются все библиотеки *JS* и код, которые реализует динамическое изменение страницы сайта. В *Servives* располагаются классы для отправки *HTTP* запросов другим микросервисам. И в папке *ViewsModels* располагается модель для отправки запросов на сторону клиента и для выведения данных, находящихся в этой модели.

Рассмотрим основные компоненты данного микросервиса.

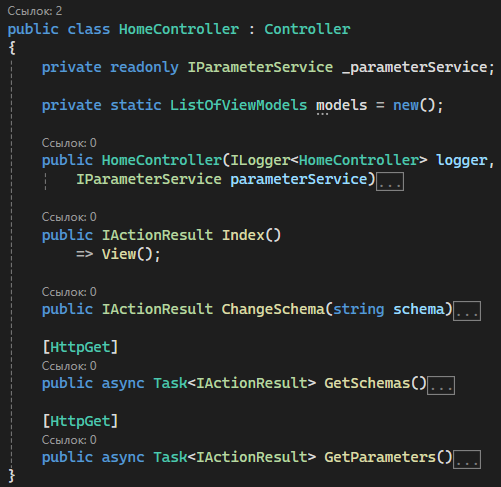


Рисунок 5.3.7 – Главный контроллер данного

микросервиса

Контроллер имеет два поля \_*parameterService* и *models*. Первое поле реализует интерфейс сервиса для отправки *HTTP* запросов выше рассмотренному микросервису. Второе поле представляет статическое поле модели ответа на запросы со стороны клиента и хранит в себе данные необходимые для вывода графиков или моментальных значений параметров окружающей среды.

Также данный контроллер имеет методы необходимые для обработки данных и возвращения ответа на запрос:

* *Index()* – метод, который перебрасывает пользователя на начальную страницу.
* *ChangeSchema(string)* – метод, который изменяет воспроизведение данных датчика
* *GetSchemas()* – метод, который возвращает id всех используемых датчиков
* *GetParameters()* – метод, реализующий алгоритм возвращения всех данных, необходимых для построения графиков и представления мгновенных параметров окружающей среды

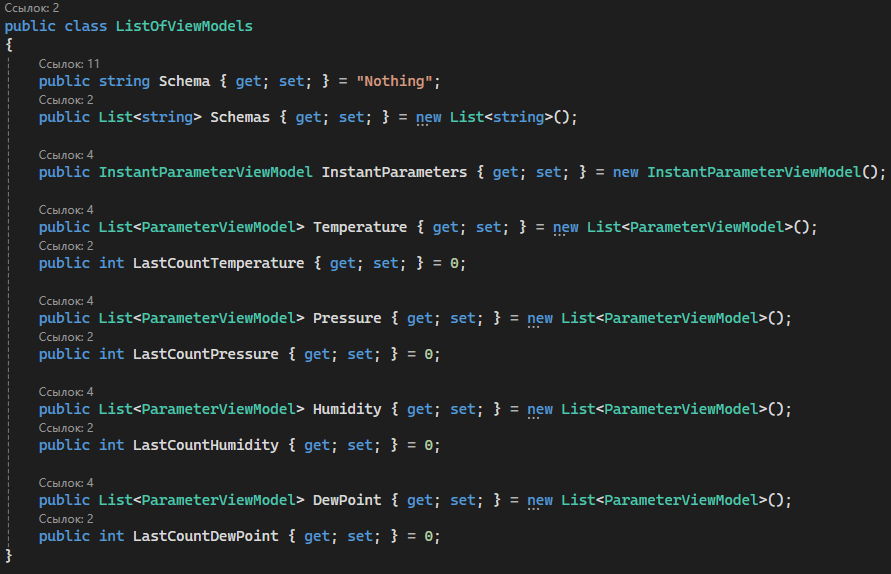


Рисунок 5.3.8 – Модель ответа на запрос

клиента

Рассмотрим компонент реализующий метод отправки *HTTP* запросов другому микросервису. Этот компонент состоит из двух интерфейсов и двух классов реализующие эти интерфейсы. В интерфейсах заложены основные методы для реализаций классов, которые необходимо обязательно реализовать.

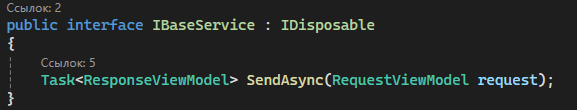


Рисунок 5.3.9 – Интерфейс базового сервиса

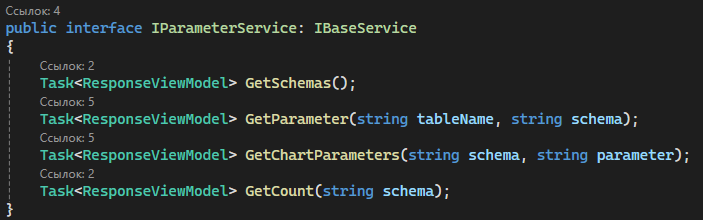


Рисунок 5.3.10 – Интерфейс сервиса параметров

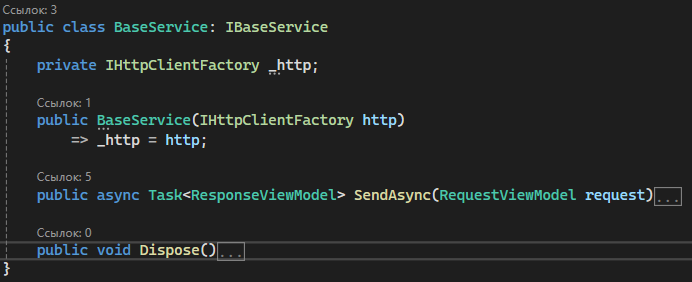


Рисунок 5.3.10 – Класс реализующий

базовый интерфейс сервисов

Данный сервис имеет поле, которое используется в методе *SendAsync*, и необходимо для построения запроса и получения ответа от стороннего микросервиса. Также данный класс использует метод *Dispose()* для полной очистки объекта после завершения его использования.



Рисунок 5.3.11 – Класс реализующий

сервис параметров

Данный класс вызывает асинхронный метод из базового интерфейса и отправляет ему параметр *RequestViewModel* с заполненными свойствами, необходимые для отправки нужного запроса микросервису и получение от него ответ на запрос.

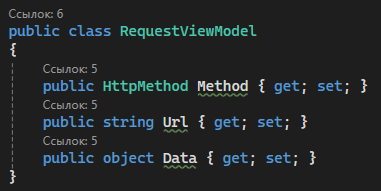


Рисунок 5.3.12 – Модель, описывающая

отправляемый запрос

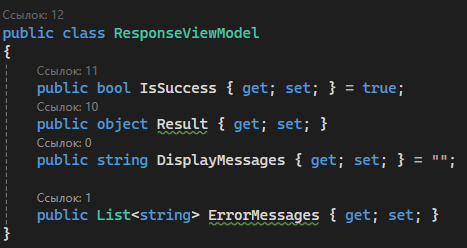


Рисунок 5.3.13 – Модель, описывающая

ответ на запрос

И в последнюю очередь рассмотрим пользовательский интерфейс данного приложения, его функциональность и возможности.

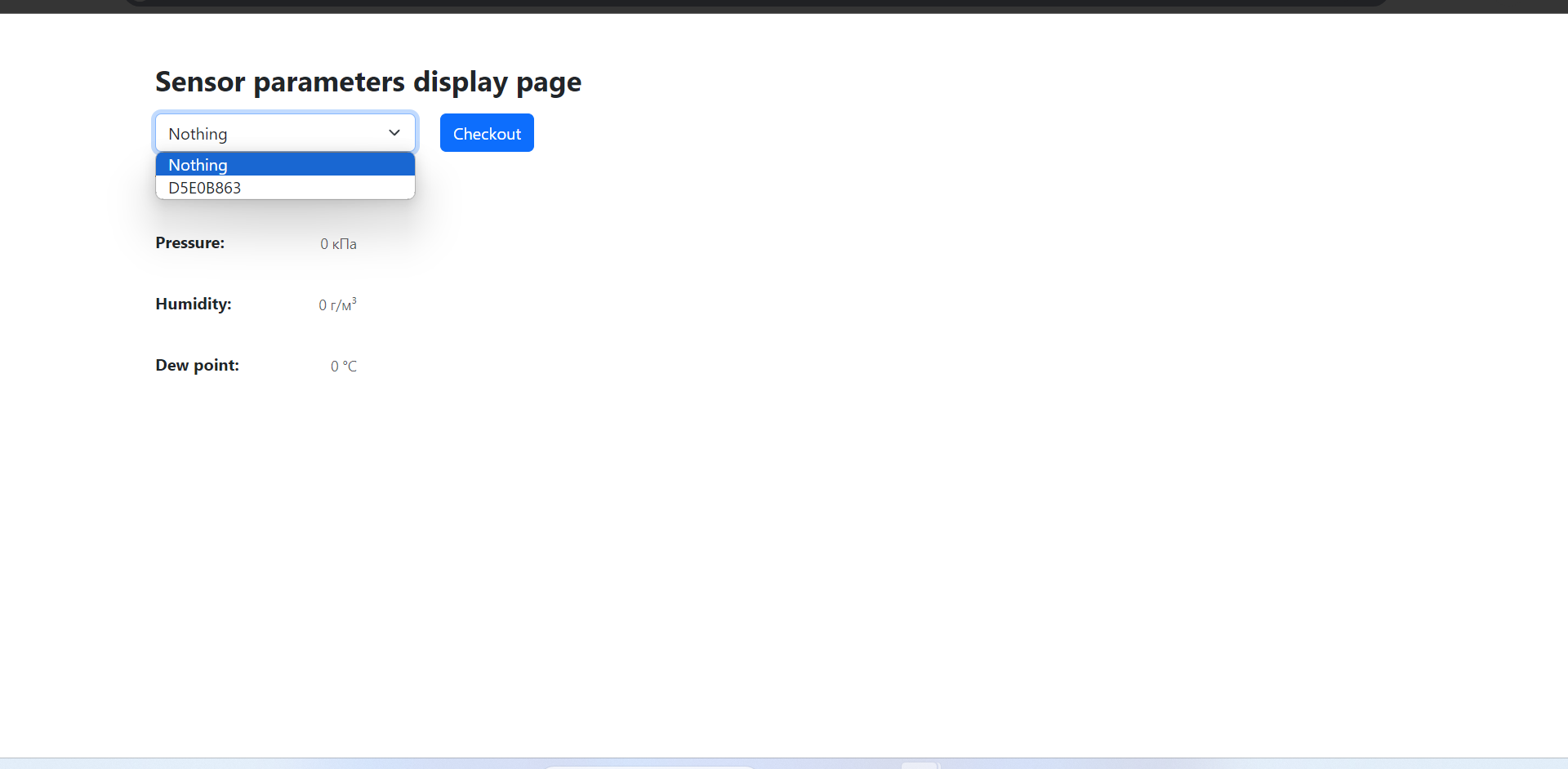


Рисунок 5.3.14 – Стартовая страница сайта при первом входе

При первом входе нам будет представлен список датчиков, графики которых мы хотим рассмотреть. При выборе датчика и клика на кнопку “*Checkout*” нам будет продемонстрированы графики всех параметров мониторинга окружающей среды и их мгновенные значения.

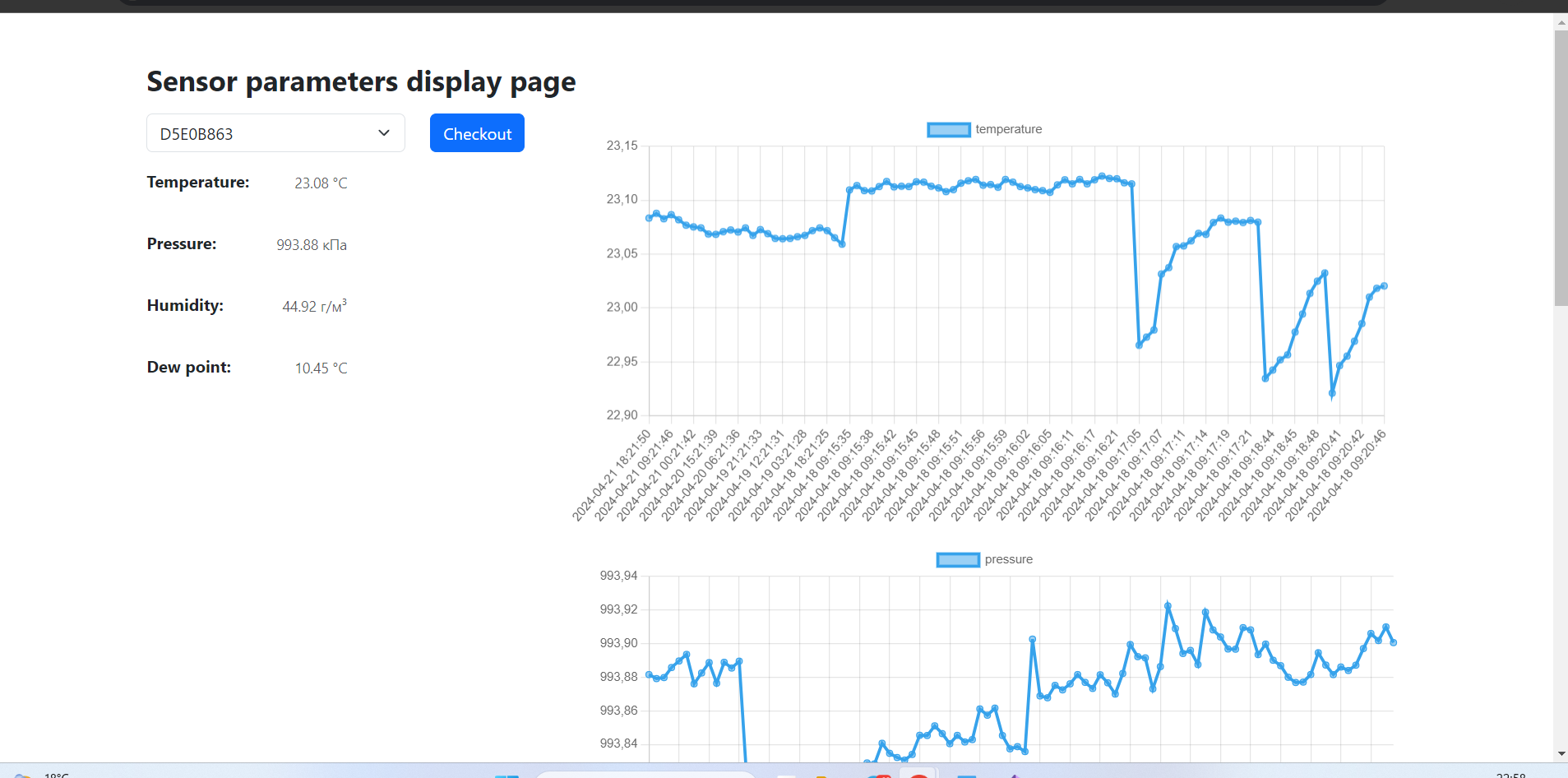


Рисунок 5.3.15 – Стартовая страница при выборе датчика

Но данная система не будет показывать параметры в режиме реального времени, если фоновый процесс не будет включен, так как параметры из датчика не записываются в базу данных. Давайте запустим фоновый процесс и посмотрим на результат.

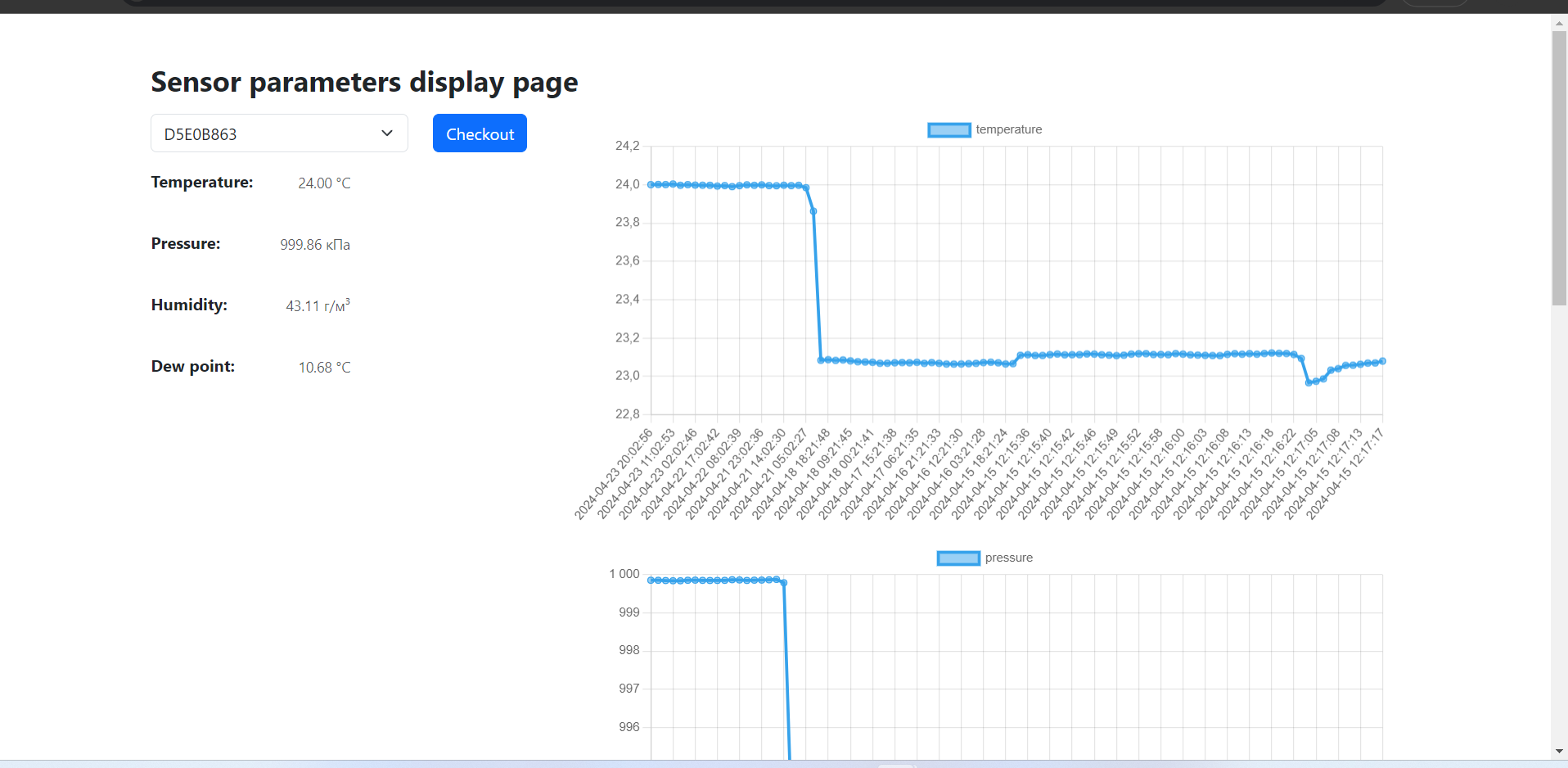


Рисунок 5.3.16 – Стартовая страница при запуске всех программ