МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Кафедра «Прикладная математика и информатика»
(наименование)

01.03.02 Прикладная математика и информатика
(код и наименование направления подготовки / специальности)

Компьютерные технологии и математическое моделирование
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему «Разработка алгоритма проверки введенных показаний приборов учета коммунальных услуг»

Обучающийся В.В. Огуречников
(Инициалы Фамилия) (личная подпись)

Руководитель канд.пед.наук, доцент, О.М. Гущина
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Консультант канд. пед. наук, доцент, С.А. Гудкова
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2024

Аннотация

Тема выпускной квалификационной работы: «Разработка алгоритма проверки введенных показаний приборов учета коммунальных услуг».

ВКР посвящена разработке ПО для проверки введенных показаний приборов учета коммунальных услуг.

В ходе выполнения исследований по ВКР был проведен сравнительный анализ различных способов построения высоконагруженных систем и анализ методов масштабирования. Была выведена математическая модель для прогнозирования эффективности алгоритма проверки введенных показаний приборов учета коммунальных услуг, с использованием модели акторов, а также разработка и тестирование ПО.

Во введении прописывается актуальность темы, написаны цель и задачи.

В первом разделе рассматривается предметная область и проводится анализ различных методов реализации и масштабирования.

Во втором разделе описана архитектура и структура взаимодействий сущностей программы, способы хранения информации, а также спроектирована система.

В третьем разделе рассматривается разработка ПО и тестирование по критериям валидации.

В заключении представлены результаты выполнения ВКР.

ВКР состоит из введения, трёх разделов, заключения и списка использованной литературы.

Abstract

Title of the graduation thesis: "Development of an algorithm for checking the entered meter readings for utility services."

The thesis is dedicated to the development of software for checking the entered meter readings for utility services. During the research for the thesis, a comparative analysis of various methods for building high-load systems and scaling methods was conducted. A mathematical model was derived to forecast the effectiveness of the algorithm for checking the entered meter readings for utility services using the actor model, as well as the development and testing of the software.

The introduction outlines the relevance of the topic, states the objectives and tasks.

The first section examines the subject area and conducts an analysis of various implementation and scaling methods.

The second section describes the architecture and structure of interactions between program entities, methods of storing information, and designs the system.

The third section covers the development of software and testing according to validation criteria.

The conclusion presents the results of the thesis.

The thesis consists of an introduction, three sections, a conclusion, and a list of references.

Содержание

Введение

Неотъемлемой частью жизни каждого человека является его быт. Каждый день любой из нас живет в мире, где изначально не существовало благ цивилизации, интернета, электричества, в том виде, в котором люди используют его сейчас. Что уж тут говорить про центральное отопление. С течением времени, благодаря великим изобретателям и энтузиастам, люди обеспечили себя теплым жильем и всеми необходимыми благами для безопасного проживания в любое время года. Теперь, с развитием технологий у нас есть теплые батареи, большое количество розеток, которые питают электричеством все необходимые бытовые приборы и гаджеты, ну и само электричество в целом, как явление.

В результате этих изменений комфорт обычного человека вырос до такой степени, что люди, проживающие в городских условиях, больше не думают о том, что им нужно сходить зимой в лес за дровами, растопить печь или нагреть воду для приготовления еды и приема ванной. Все наши проблемы решает современное коммунальное хозяйство.

В следствии всего этого люди стали задумываться о горячей воде или электричестве только в тех случаях, когда нужно отправить в управляющую компанию показания своих счетчиков и оплатить квитанции ЖКХ. Однако, все эти данные должны как-то собираться. Какие – то люди должны сверять эти показания и выставлять счет на оплату, а жильцы в свою очередь должны своевременно передавать свои показания в управляющую компанию.

Эта тема является актуальной, поскольку для сбора этих данных нужны большие мощности. В нашей стране живет более 140 миллионов человек. По данным переписи населения за 2021 год, в городах живет около 75% населения России. Это огромные данные, которые в свою очередь нужно как минимум сверять. Нынешние технологии могут позволить отправлять показания онлайн, но слишком малое количество мощностей и высокая цена оборудования требует эффективного и масштабируемого подхода.

Более того, если передавать показания и проверять данные вручную, это повысит нагрузку на ЖКХ, и со временем она будет расти все больше и больше, ведь население в городах растет с каждым годом. Да и обычным гражданам не удобно каждый месяц лично приходить в управляющую компанию для передачи показаний, а для некоторых людей это и вовсе невозможно.

Объектом исследования бакалаврской работы является разработка оптимального и эффективного алгоритма проверки введенных показаний приборов коммунальных услуг.

Предметом исследования – математическая модель параллельных вычислений, основанная на понятии актора, а также библиотеки и программные продукты для реализации алгоритма.

Целью работы будет являться разработка программного обеспечения для реализации масштабируемого сервиса по приему показаний приборов коммунальных услуг и эффективного алгоритма проверки данных.

Задачи бакалаврской работы включают в себя:

- обзор существующих методов реализации подобных сервисов и сравнительный анализ использования асинхронного и параллельного подхода;
- разработку методологии проектирования ПО и составление математической модели алгоритма;
- разработку сервиса, реализующего алгоритм асинхронной проверки показаний приборов коммунальных услуг;
- тестирование и оценку качества написанного ПО;
- формулирование выводов по исследованию темы, дальнейшие рекомендации по использованию ПО и дальнейших перспектив;

В первом разделе ВКР описана методология разработки алгоритма проверки введенных показаний. Во втором разделе строится архитектура приложения и математическая модель. В третьем разделе описана реализация программного продукта и его тестирование.

1 Описание методологии разработки алгоритма

1.1 Постановка цели и задачи ВКР

Задачей данной работы является разработка алгоритма проверки введенных показаний приборов учета коммунальных услуг для массового использования, способного выдерживать большое количество пользователей и множество запросов. Программное обеспечение подразумевает под собой удобный, масштабируемый и отказоустойчивый инструмент, который избавит жильцов от ручной передачи показаний приборов учета.

Цель этой работы – создать удобный инструмент передачи показаний, который облегчит жизнь пользователям и снизит нагрузку на работников сферы ЖКХ, путем избавления их от ручной валидации большого количества данных. Также наличие алгоритма проверки введенных показаний приборов учета коммунальных услуг, может существенно сказаться на корректности вычислений в лучшую сторону, ведь влияние человеческого фактора на точность валидации сведено к минимуму.

Для достижения этой цели, перед началом написания программного кода, в первую очередь нужно разобраться с тем, каким путем пользователи будут взаимодействовать с сервисом передачи показаний. Необходимо определиться с выбором платформы, под которую будет писаться код. Есть несколько вариантов, нам же нужен тот который покроет максимальное число пользователей и не создаст лишних проблем при реализации.

Для покрытия максимального количества жителей, нужна такая платформа, которой может пользоваться каждый человек, способный выходить в интернет со своего устройства. Разработка под мобильные платформы будет слишком затратной, так как требует больших мощностей, а в некоторых случаях – определенный тип операционной системы, без которой невозможно приступить к разработке программного обеспечения. Разработка нативных клиентов для настольных компьютеров также не подойдет, так как

операционная система самих настольных компьютеров, может быть несовместима с разрабатываемым программным обеспечением, а разработка кроссплатформенных решений, может лишить нас привилегий нативных приложений. Идеальным решением для разработки алгоритма проверки введенных показаний приборов учета коммунальных услуг, может стать подход, подразумевающий под собой разработку Web – приложения для передачи показаний приборов учета. Данный подход позволит облегчить разработку, сузив спектр технологий сугубо до работы с web – сервером и HTML – страницами, а также сэкономит время на разработке нативных клиентов под операционную систему пользователя. Также ситуацию облегчает наличие огромного количества web – серверов и множество литературы для внедрения их в свои проекты и дальнейшей поддержки программного обеспечения

На данный момент времени, использование Web технологий – это самый подходящий вариант для реализации сервиса передачи показаний. Разработка на базе Web позволит программному продукту охватить максимальное количество пользователей в Российской Федерации. Каждый человек, используя любые доступные ему платформы, может открыть браузер и начать взаимодействие с нашим сервисом, при подключении к интернету.

1.2 Популярные подходы к созданию Web-сервиса, реализующего алгоритм проверки введенных показаний

От выбора подхода к реализации зависит легковесность, надежность и наличие возможностей для последующего масштабирования программного продукта.

Возможные методы реализации:

- Rest API;
- SOAP;
- GraphQL.

Широко принятый и эффективный подход для создания Web – приложений – это метод написания, который базируется на принципах архитектуры REST (Representation State Transfer) для распределенных систем World Wide Web. Данный архитектурный стиль очень прост в использовании, ведь в своей основе он построен на клиент – серверной архитектуре с HTTP - протоколом передачи данных. Стиль REST опирается на ресурсы, а точнее уникальные URL, которые по сути являются первичными ключами для единиц данных [16][18]. Стоит также отметить, что данные передаются без применения дополнительных слоёв, а это положительно влияет на ресурсоёмкость реализуемой системы.

При сравнении SOAP и REST, последний предоставляет разработчику простой и более гибкий подход к созданию Web – сервисов, в основном благодаря базировании работы на HTTP методах и поддержке различных форматов данных, таких как JSON и XML, что упрощает взаимный обмен информацией между клиентом и сервером [17]. Ещё REST не требует специальных запросов и имеет более одного URL, в отличии от GraphQL [23]. Это значит разработка и понимание API значительно упрощаются [24]. Также REST лучше подходит для работы в формате отдачи ресурсов к тому, кто их запросил, это может быть удобнее в контексте бизнес-логики.

1.3 Сравнительный анализ способов построения высоконагруженных систем

Сталкиваясь с задачей написания сервиса с алгоритмом проверки введенных показаний приборов учета коммунальных услуг, рано или поздно возникает потребность в том, чтобы система могла выдерживать большие нагрузки. Приборы учета стоят в каждой квартире, следовательно спрос на пользование сервисом будет очень высоким. Необходимо выбрать какими способами решить задачу повышения эффективности алгоритма проверки показаний.

Для решения такой важной проблемы можно прибегнуть к параллельным вычислениям, однако такой подход влечёт за собой определенные трудности. Например, синхронизация доступа к общим ресурсам приведет алгоритм проверки в состояние гонки и блокировкам, что снизит эффективность и может привести к ошибкам в пользовательских данных, а в сфере ЖКХ это недопустимо. Более того, управление множеством параллельных потоков может увеличить сложность и требовать к себе больше ресурсов для контроля. Все эти трудности негативно скажутся на пользовательском опыте, и побудят людей использовать старый добрый способ передавать показания лично в управляющую кампанию.

Альтернативой может стать асинхронный подход к вычислениям и проверке показаний. В отличии от параллелизма асинхронность предполагает выполнение задач независимо друг от друга, без явного ожидания завершения предыдущей задачи или освобождения какого-нибудь ресурса. Это позволит алгоритму проверки введенных показаний эффективно работать на большое количество пользователей сервиса, что в сравнении с параллелизмом даст людям положительный опыт. Передача показаний будет быстрой и удобной.

Наилучшим методом использования асинхронности, является модель акторов. Она поддерживает параллелизм, но из-за своего устройства, а точнее из-за устройства акторов, проблемы параллельных вычислений были решены.

В акторной модели каждый актор имеет свое собственное состояние и исполняется независимо от остальных акторов в системе, что избавляет нас от проблем с гонками данных и состояний. Акторы – это независимые сущности в акторной системе, которые могут принимать решения и общаться с другими акторами через асинхронные сообщения, по принципу «отправил – забыл». В ответ на получаемые сообщения актор может принимать независимые решения, отправлять другие сообщения или же породить нового актора. Сообщения – это простые структуры данных, которые нельзя изменять после создания, то есть они «immutable» [13][14]. Пример взаимодействия акторов представлен на рисунке 1.

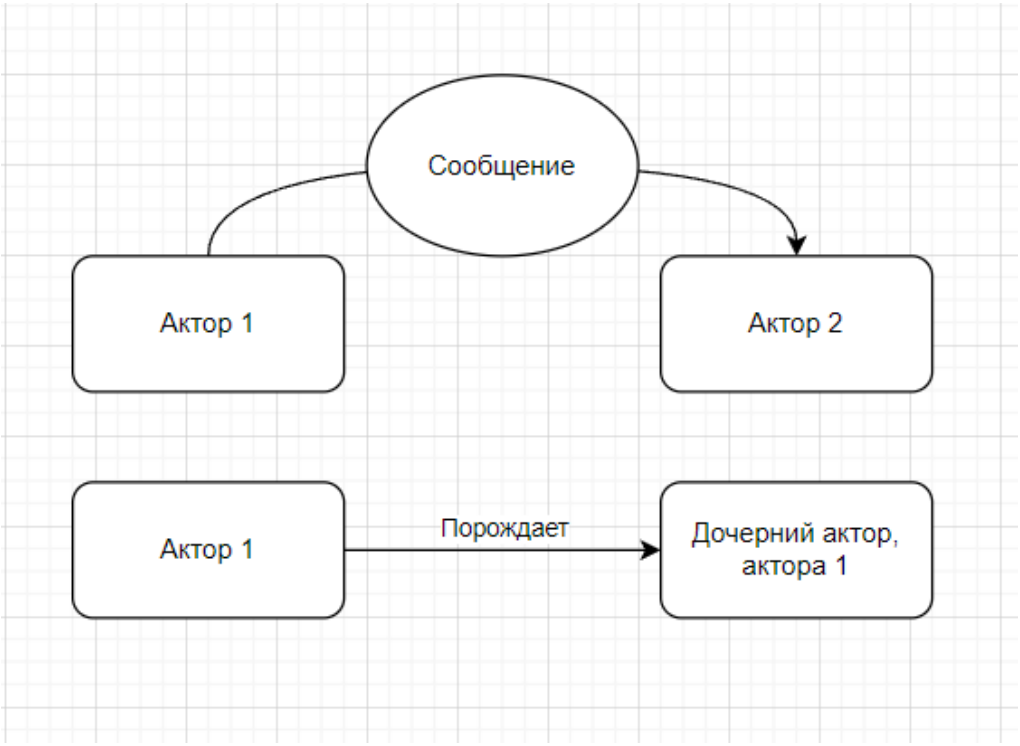


Рисунок 1 – Взаимодействие акторов

Модель акторов обеспечивает более высокий уровень абстракции для написания параллельных и распределенных систем [3]. Это избавляет разработчика от необходимости иметь дело с явной блокировкой и управлением потоками, упрощая написание правильных параллельных систем.

На данный момент времени, наиболее популярной технологией построения модели акторов является фреймворк Akka. Это набор инструментов открытого проекта, разрабатываемого компанией Lightbend. Akka – представляет модель акторов для масштабирования приложений в JVM (Java Virtual Machine) в обоих направлениях – по вертикали и горизонтали [15].

В своей выпускной квалификационной работе я буду использовать Akka в связке с языком программирования Scala. Данный язык программирования разработан для выполнения в JVM и требует существенно меньше кода, в сравнении с Java, для построения модели акторов [4][5]. Также сам фреймворк

Akka написан на Scala, что может избавить от конфликтов и возможных ошибок компиляции во время разработки [2].

Таким образом выбор модели акторов, в сравнении с параллелизмом, представляет более эффективный подход к реализации алгоритма проверки введенных показаний, ведь использует асинхронные сообщения и избавляет от недостатков параллельных вычислений.

1.4 Анализ методов масштабирования

Масштабирование - критически важный аспект проектирования сервиса проверки показаний с алгоритмом валидации пользовательских данных. В связи с огромным распространением приборов учета нагрузка будет расти и важность дальнейшей эффективной работы будет расти вместе с ней. Также необходимо предусмотреть рост сложности приложения если в него будет встроен новый функционал [7].

При росте количества пользователей или функционала сервиса проверки введенных показаний приборов учета коммунальных услуг, в дальнейшем от масштабируемости мы ожидаем два сценария.

Один из наихудших сценариев – когда приходиться платить больше за недоиспользуемые ресурсы. Другой неблагоприятный сценарий – когда сложность сервиса с алгоритмом проверки введенных показаний выходит за допустимые пределы с добавлением новых ресурсов. В связи с этим, возникают две потребности при масштабировании в будущем:

- Сложность должна оставаться на как можно более низком уровне;
- Ресурсы должны использоваться максимально эффективно.

При повышении сложности, разработка новых функций и поддержка старого кода со временем может стать настолько трудоемкой, что будет стоить большого количество человеческих часов работы.

Во-первых, мы ожидаем медленный рост потребности в ресурсах при росте требований к сервису. Данная зависимость изображена на рисунке 2.

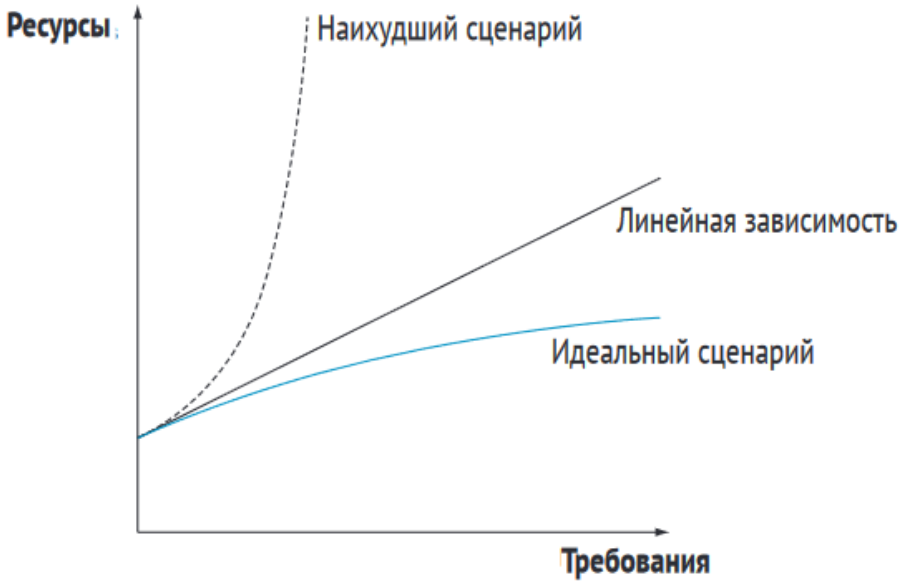


Рисунок 2 – Зависимость требований от ресурсов

Во-вторых, если возникает потребность в наращивании ресурсов, желательно, чтобы сложность оставалась на прежнем уровне, или хотя бы росла как можно медленнее [16]. На рисунке 3 изображена зависимость между объемом ресурсов и сложностью сервиса.

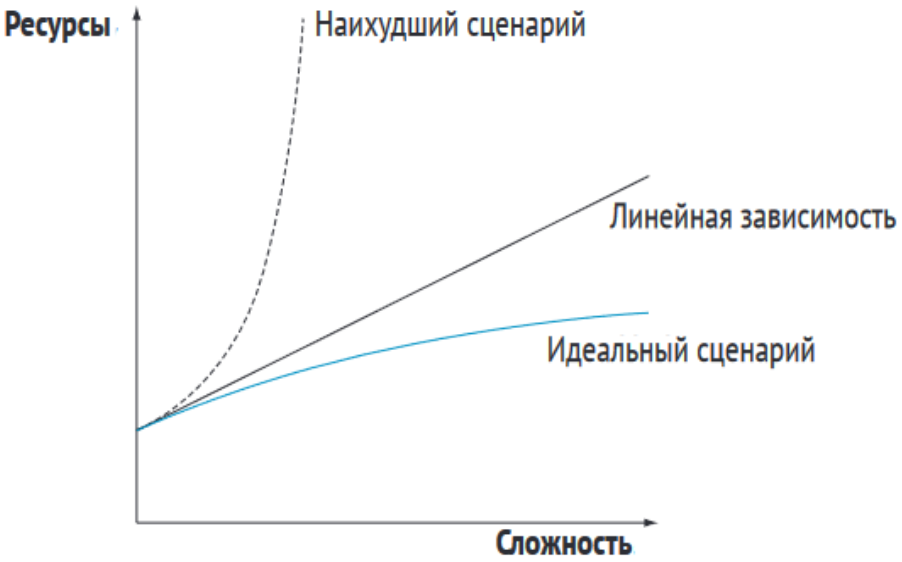


Рисунок 3 – Зависимость сложности сервиса от ресурсов

Для реализации алгоритма проверки введенных показаний приборов учета коммунальных услуг, была выбрана модель акторов, представленная фреймворком Akka.

Рассмотрим и проанализируем два подхода к масштабированию:

– Традиционный подход;

– Подход с использованием Akka.

Используя традиционный подход, мы бы начинали с простого приложения, размещающегося в памяти. Это может быть простой Web – сервис, принимающий от пользователей показания приборов учета коммунальных услуг, например электроэнергия, отопление, холодная и горячая вода. В таком приложении на начальных этапах данные будут храниться в оперативной памяти, благодаря различным структурам данных. Однако с ростом популярности приложения и ростом функционала, рано или поздно придется переходить от оперативной памяти к полноценной базе данных. В ней будут храниться старые показания, даты показаний, а также служебная информация, например даты поверки. Код станет сложнее, ведь теперь все нужные данные и состояния не хранятся в памяти, а значит методы объектов не смогут обращаться к ним непосредственно; вся важная логика переместилась в инструкции к базе данных.

Когда пользователи начнут вводить данные и отправлять их на проверку, придется постоянно опрашивать базу для сравнения новых показаний по всем нужным параметрам, и для каждого параметра будет свой отдельный запрос. Такой подход ведет за собой риски, так как обращения к базе данных, по сути, является вызовом RPC, и почти все стандартные драйверы баз данных, такие как JDBC, используют блокирующий ввод/вывод. То есть одновременно используются потоки выполнения и вызовы RPC. Блокировки памяти, используемы для синхронизации потоков, и блокировки базы данных требуют повышенной осторожности, если возникает ситуация их объединить. Приходиться жестко определять, какие части приложения используют потоки для вертикального масштабирования (увеличение

мощности сервера), а какие – RPC для масштабирования по горизонтали (увеличение количества серверов). Также появилась проблема скорости приложения, ведь алгоритму проверки введенных показаний придется постоянно обращаться к базе для сверки каждого нового показания, ведь не осталось даже какого-то промежуточного состояния, а в случае, если какой-то компонент даст сбой, есть риск затормозить всю систему. На рисунке 4 проиллюстрировано взаимодействие алгоритма с базой.

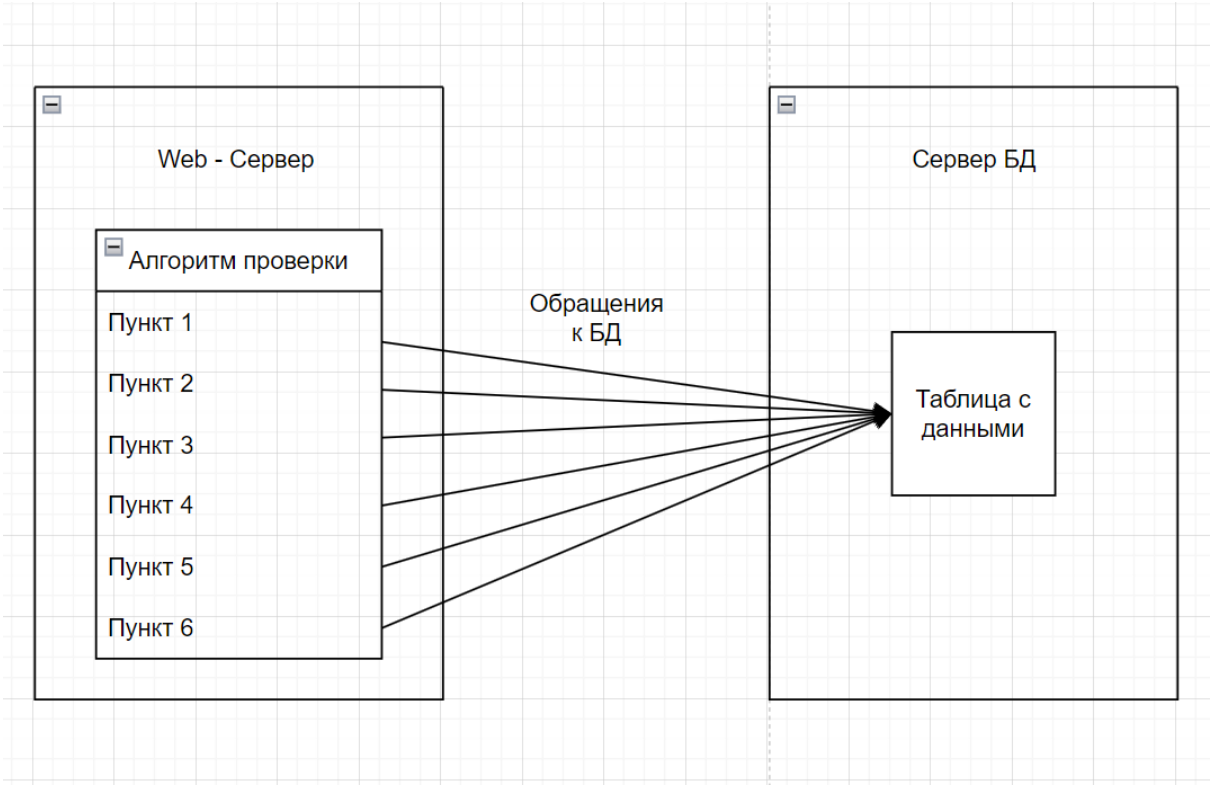


Рисунок 4 – Взаимодействие алгоритма с базой данных

Постоянное предугадывание возможных ошибок и исключений замедляют разработку, увеличивая ее стоимость, а также усложняет код.

Мало того, что сложность увеличилась, стало сложнее добавлять новые интерактивные функции. Опрос базы данных, далеко не лучшая идея для алгоритма проверки введенных показаний приборов учета коммунальных услуг, но других альтернатив нет, ведь вся логика сосредоточена в объектах

доступа к данным, а сама база данных не может послать событие Web – серверу.

Очевидно, что традиционный подход плохо масштабируется. С каждой новой проблемой приходиться менять направление, с риском неконтролируемого роста сложности и высокого спроса на ресурсы.

В подходе с использованием Akka, модель акторов упрощает разработку сервиса и помогает легко масштабировать его при любой возможности, автоматически решая проблемы конкурентного выполнения [7][8]. В таблице 1 приведены различия между подходами.

Таблица 1 – Различия между подходами к масштабированию

Цели	Методы достижения	
	Традиционный	На основе Akka
Масштабирование	Использовать комбинацию потоков выполнения, общего изменяемого состояния в базе данных (CRUD операции) и RPC – вызовов веб-служб.	Акторы посылают и принимают сообщения. Общее изменяемое состояние отсутствует. Выполнением управляют неизменяемые события.
Передача интерактивной информации	Опрос текущей информации	Передача при появлении события
Масштабирование в сети	Синхронные RPC – вызовы, блокирующий ввод/вывод.	Асинхронная передача сообщений, неблокирующий ввод/вывод.
Обработка ошибок	Обработка всех исключений. Работа продолжается, только если все компоненты находятся в рабочем состоянии.	Отказы изолируются, и работа продолжается без отказавших компонентов.

В контексте выпускной квалификационной работы, для реализации алгоритма проверки введенных показаний приборов учета коммунальных услуг, при использовании модели акторов, главной задачей было уменьшить количество обращений к базе данных.

Каждый актор, по определению имеет свое собственное состояние. Именно его можно использовать в качестве промежуточного состояния, а

каждый актор будет закреплен за конкретным лицевым счетом и будет работать только с данными по конкретной квартире. Все обращения к базе данных сводятся к двум обращениям:

– Получить старые данные;

– Сохранить новые (проверенные алгоритмом) данные.

Такой подход значительно облегчает нагрузку на базу данных. Данное взаимодействие представлено на рисунке 5.

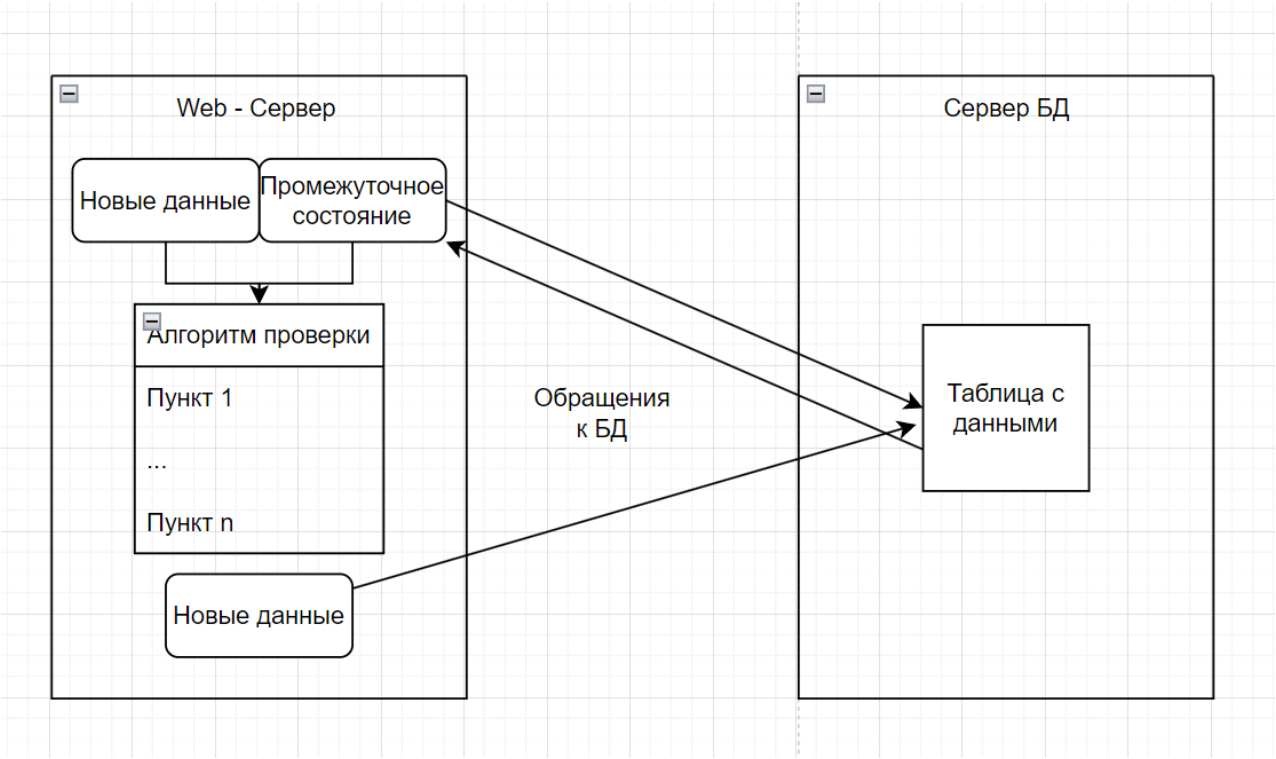


Рисунок 5 – Взаимодействие алгоритма с базой данных, при помощи актора

Как говорилось выше, приложение для поддержки большого количества пользователей должно стать конкурентным. Необходимая модель конкурентного программирования, должна быть работоспособна как на одном, так и на нескольких серверах. Модель акторов использует отправку и прием сообщений как абстракцию, чтобы не зависеть от количества используемых потоков выполнения и серверов [30].

Акторы можно представить как объекты, независимые друг от друга по трем критериям:

– Местоположение;
– Время;
– Интерфейс.

Местоположение – актор не дает никаких гарантий и не имеет никаких ожиданий касательно своего местоположения.

Время – актор не дает никаких гарантий и не имеет никаких ожиданий относительно момента завершения своей работы.

Интерфейс – актор не определяет интерфейса. Актор не имеет никаких ожиданий о том, смогут ли его сообщения понять остальные акторы. Никакие два актора не могут иметь общих данных, они все передаются в сообщениях.

Обеспеченная этими критериями гибкость, играет важную роль в масштабировании сервиса приема показаний. Если в системе имеется достаточное количество процессоров, то акторы могут действовать одновременно, в ином случае друг за другом. Акторы могут находиться по соседству или далеко друг от друга, и в случае ошибки способны получать сообщения, обработать которые не в состоянии.

Зависимость по этим трем критериям отрицательно сказывается на способности сервиса восстанавливаться после отказа и масштабироваться по мере необходимости. Если все же приложение будет зависеть от местоположения, времени и интерфейса как в традиционном подходе, то оно может существовать только в единой среде выполнения и будет полностью выходить из строя, если какой-нибудь его компонент откажет.

Вывод по разделу 1

В первом разделе мы поставили задачи и цели ВКР, выбрали REST для построения Web-сервиса. Также провели сравнительный анализ способов построения высоконагруженных систем и проанализировали методы масштабирования нашего программного продукта.

2 Описание архитектуры системы для работы алгоритма проверки введенных показаний

2.1 Описание требований к системе

Каждая квартира, имеет свой закрепленный лицевой счет. Первым делом, пользователь идентифицирует себя по лицевому счету, вводя его в соответствующее поле. Следующим шагом система должна вывести на экран всю информацию по приборам учета коммунальных услуг, которые имеются в базе по данному лицевому счету, а именно:

- Серийный номер ПУ;
- Наименование услуги;
- Два предыдущих показания;
- Даты двух предыдущих показаний.

Затем, по нажатию специальной кнопки, пользователю открывается форма отправки новых показаний, с такими приборами учета, как:

- Теплоноситель (горячая вода);
- Холодная вода;
- Электроэнергия;
- Отопление.

По нажатию кнопки «Отправить», данные отправляются на валидацию по следующим пунктам:

- Новое показание не меньше предыдущего;
- Не отрицательное;
- Не больше максимального допустимого;
- При истечении срока поверки прием показаний невозможен;
- Количество знаков до и после запятой, не больше, чем у ПУ;
- Если новый расход превышает в 2 раза средний расход за последние 2 месяца, то возникает предупреждение.

Максимальный возможный расход для внесения вычисляется как большее значение двух показателей:

– Среднее десятикратное потребление;
– Нормативное значение.

Нормативные значения составляют:

– Для воды – 36;
– Для электроэнергии – 1350;
– Для отопления – 108.

Все эти требования должен включать в себя сервис с алгоритмом проверки введенных показаний приборов учета коммунальных услуг.

2.2 Способ хранения показаний приборов учета – выбор БД

В ходе выполнения выпускной квалификационной работы было принято решение использовать технологии БД с открытым исходным кодом. Такие решения как Oracle Database и SQL Server можно сразу вычеркнуть и списка возможных технологий для разработки алгоритма проверки введенных показаний приборов учета коммунальных услуг. Эти базы данных имеют высокую стоимость или слабый функционал на бесплатных версия. К тому же – некоторые из них имеют большие проблемы с импортом файлов и потребляют большое количество ресурсов, чего стоит избегать при разработке высоконагруженного сервиса, с большим количеством пользователей и высоким потенциалом к масштабированию, как в вертикальном направлении, так и в горизонтальном.

Одни из «open – source» лидеров в сфере реляционных баз данных, являются MySQL и PostgreSQL [1]. Обе технологии имеют высокий функционал, и зарекомендовали себя как надежные решения для больших и высоконагруженных систем.

Сравнение данных баз данных отображено в таблице 2.

Таблица 2 – Сравнительный анализ MySQL и PostgreSQL

СУБД	Преимущества	Недостатки
MySQL	Надежность; Простота; Стабильность.	Отсутствие поддержки OLAP и XML; Платная поддержка.
PostgreSQL	Инновации; Масштабируемость; Функциональность; Большой объем обработки; Доступность интерфейсов.	Неполная поддержка; Конфигурация.

В процессе разработки было принято решение остановиться на PostgreSQL. Если MySQL в первую очередь ориентирована на надежность, простоту и стабильность, то PostgreSQL – на инновации и расширенную функциональность. Она разработана в качестве более универсальной системы, способной справляться как с рабочими нагрузками при интенсивных операциях чтения, так и при интенсивной записи, но с немного меньшей производительностью, чем MySQL, которая оптимизирована для больших нагрузок во время чтения [6]. Однако в последних версиях PostgreSQL улучшила свою производительность, особенно в отношении сложных запросов и обработки данных.

Программный продукт, разрабатываемый в рамках выпускной квалификационной работы, опирается на масштабирование системы, и предполагает, что такое свойство является краеугольным для поддержки большого количества пользователей. PostgreSQL удовлетворяет такое требование к нашей системе, ведь эта БД известна своей вертикальной масштабируемостью, то есть она может обрабатывать большие объемы данных и манипулировать вычислительными мощностями путем добавления дополнительных ресурсов на один узел. А горизонтальное масштабирование данная БД поддерживает с помощью шардинга, который позволяет разделять большие массивы данных по нескольким узлам [22].

В следствии всего описанного выше, PostgreSQL является хорошим выбором, поскольку данная БД имеет открытый исходный код, хорошо

масштабируется и предлагает мощную поддержку сложных запросов, текстового поиска и индексации данных. Кроме того, он имеет репутацию очень надежного и безопасного.

2.3 Описание структуры алгоритма проверки показаний

Сервис с алгоритмом проверки введенных показаний приборов учета коммунальных услуг состоит из двух классов акторов (Manager, Validator) и класса, отвечающего за поддержку и связывание маршрутов HTTP (RestApi). Структура взаимодействия этих сущностей представлена на рисунке 6.

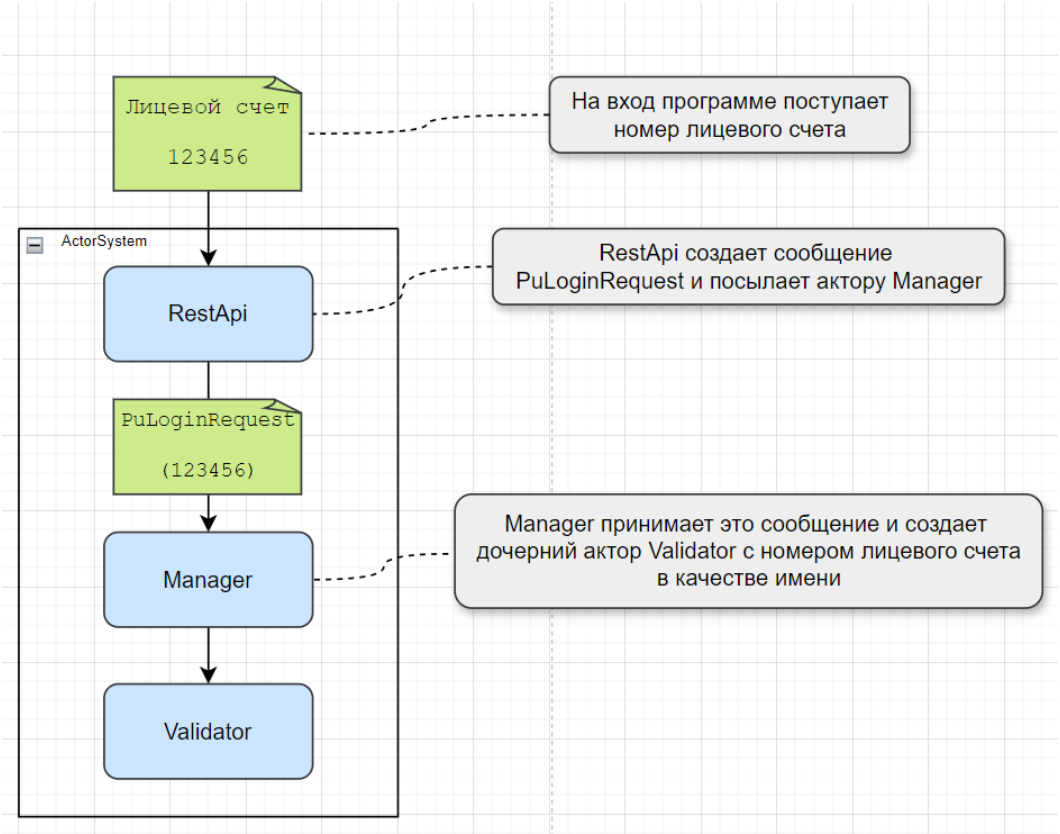


Рисунок 6 – Структура взаимодействия сущностей

RestApi использует актор Manager как прокси, что бы связать работу с пользователем и логику работы алгоритма между собой. RestApi содержит необходимые для работы сервиса маршруты для обработки HTTP – запросов

[10][12]. Маршруты определяют, как должны обрабатываться HTTP – запросы с применением удобного предметно – ориентированного языка (DSL), поддерживаемого модулем Akka HTTP [9][11].

На рисунке 7 – визуализировано данное распределение.

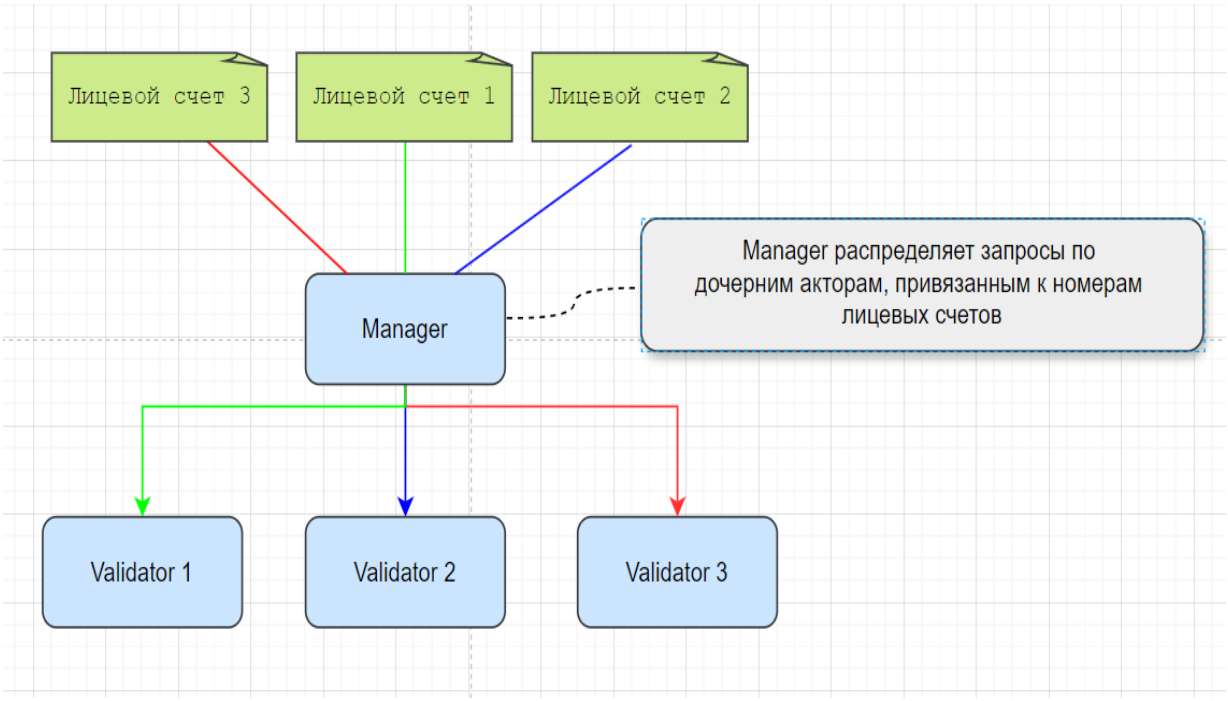


Рисунок 7 – Распределение запросов по дочерним акторам

Актор Manager выполняет функцию распределителя. Его задача делегировать все запросы пользователей дочерним акторам, которые будут отвечать строго за те показания, которые идентифицируются конкретным лицевым счетом. Иначе говоря – один дочерний актор отвечает за один лицевой счет. Распределение, как и любые другие действия выполняется путем отправки сообщений.

Дочерними акторами, порожденными актором Manager, служат экземпляры Validator. Эти акторы главные вычислительные механизмы алгоритма проверки введенных показаний приборов учета коммунальных услуг. Именно они будут работать с данными и хранить в своем состоянии данные пользователей. Актор Validator, следит за выполнением требований к

введенным данным, описанным в разделе 2, пункт 1 и, если все хорошо, новые показаний уже успешно добавляются в базу данных. Также в его задачи входит запрос данных из базы и отправка их на уровень выше.

Запрос данных представлен на рисунке 8, а проверка новых показаний на рисунке 9.

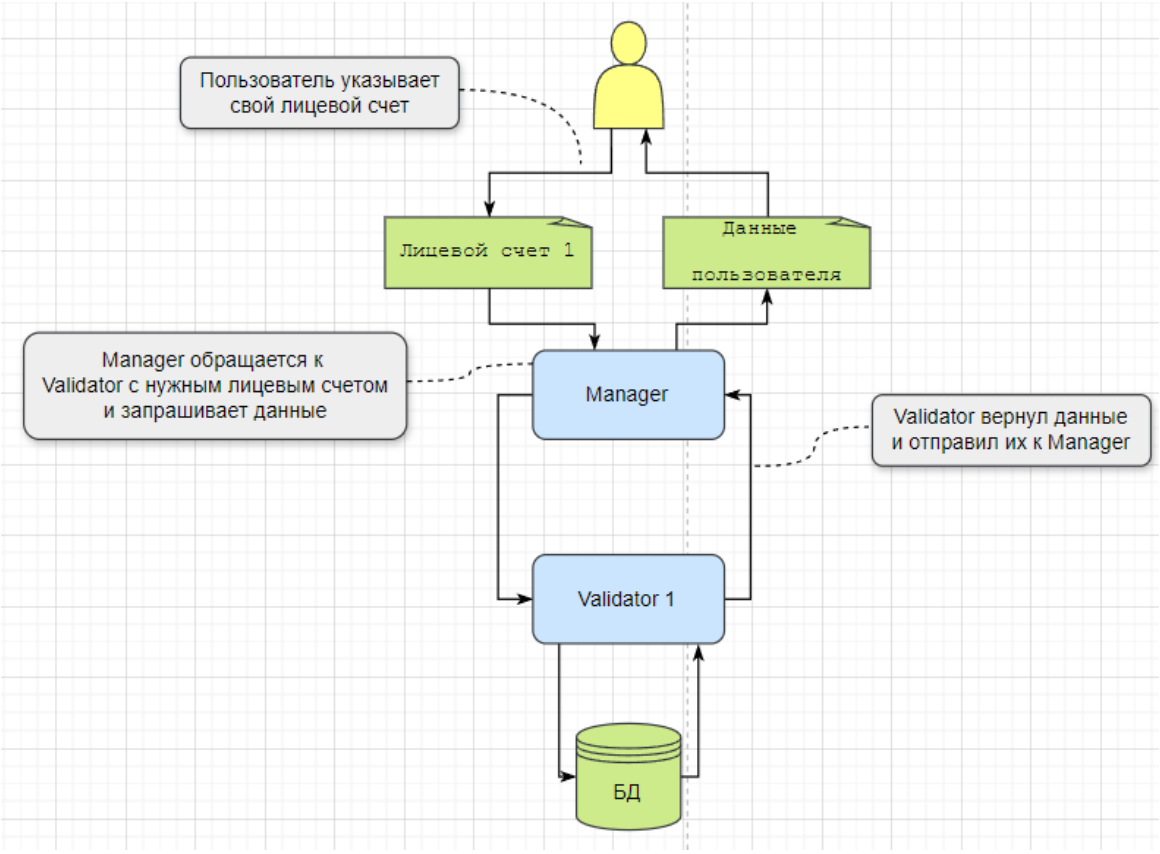


Рисунок 8 – Запрос данных из БД

Как уже говорилось все взаимодействия – запрос данных, проверка показаний и сохранения в БД выполняются путем отправки асинхронных сообщений от актора к актору. При описании класса актора указываются типы сообщений, которые они могут принимать и затем выполнять определенные действия. Список сообщений для Manager и Validator представлены в таблице 3.

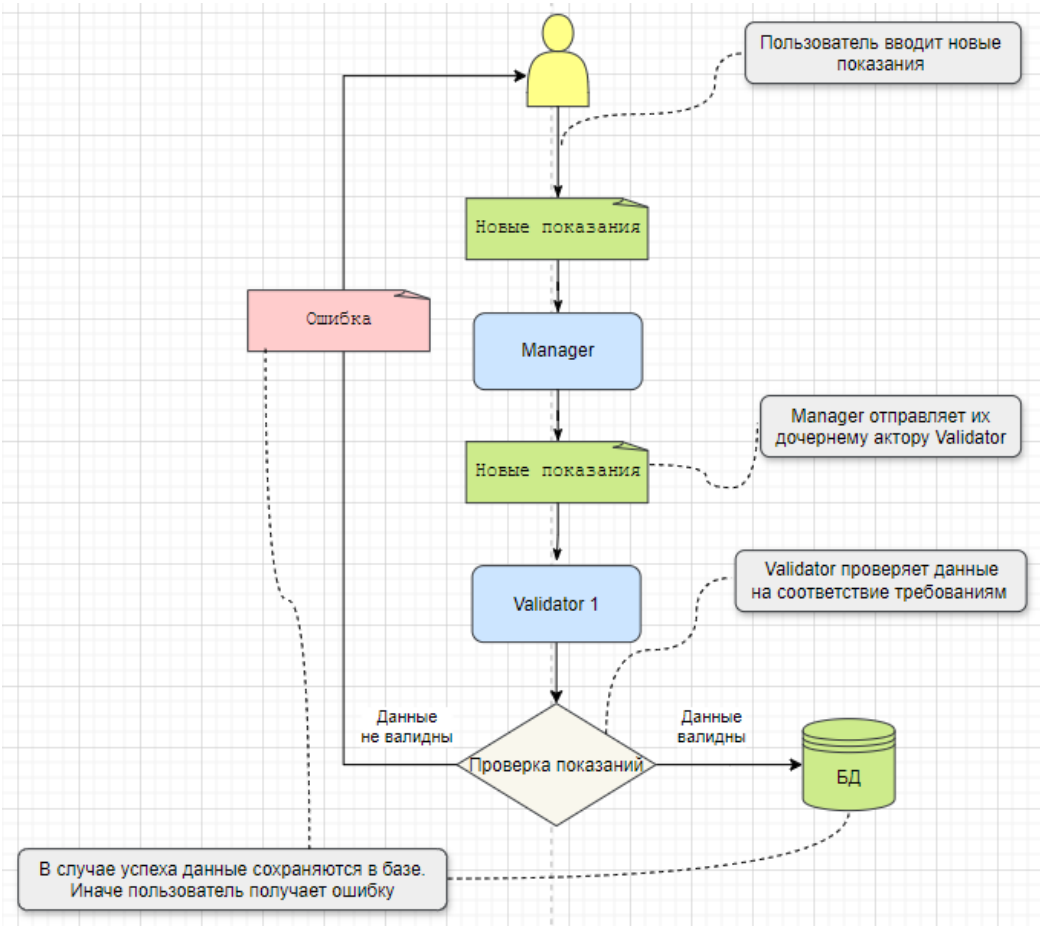


Рисунок 9 – Проверка новых показаний

Таблица 3 – Сообщения для акторов Manager и Validator

Manager	Validator
PuLoginRequest - Запросить у Validator данные по лицевому счету, если актора еще не существует - создать и запросить	PuDataRequest - Запросить и сохранить данные в промежуточном состоянии
PuValidCheck - Отправить в Validator новые показания на проверку	Validating - Проверить данные на соответствие требованиям
ValidGood - Сообщение об успехе проверки	
ValidBad - Сообщение об ошибке	PuCommit - Сохранить новые показания в БД

Реализация данных сообщений в итоговом программном продукте будет играть ключевую роль в построении алгоритма проверки введенных показаний приборов учета коммунальных услуг.

2.4 Математическая модель для алгоритма проверки введенных показаний приборов учета коммунальных услуг

Для предсказания эффективности алгоритма на основе модели акторов, составим следующую математическую модель:

Пусть A – множество всех акторов, M – множество всех возможных сообщений, R – производительность системы (количество запросов, которые система может обрабатывать за единицу времени), T – время обработки одного запроса.

Тогда каждый актор $a \in A$ можно представить как функцию $f: M \times R \to A \times M^* \times R$, где M^* - множество последовательностей сообщений. Функция f определяет поведение актора следующим образом: для каждого сообщения $m \in M$ и текущей производительности $r \in R$, $f(m, r) = (a', M', r')$, где a' - новый актор (возможно тот же самый актор a), M' - последовательность сообщений, которые актор решает отправить, и r' - новая производительность системы.

В случае Manager, при получении сообщения PuLoginRequest, он создает нового актора Validator (если таковой еще не существует для данного лицевого счета), затем увеличивает $\boldsymbol{r}$ на 1 и отправляет актору Validator сообщение PuDataRequest. При получении PuValidCheck, Manager отправляет актору Validator сообщение – Validating.

В случае Validator, при получении сообщения PuDataRequest, он делает запрос в базу данных и сохраняет всю таблицу по своему лицевому счету. При получении сообщения Validating, если все верно, Validator отправляет сам себе сообщение PuCommit, и тогда все новые показания сохраняются в базе данных по конкретному лицевому счету.

Предположим, что алгоритм проверки введенных показаний приборов учета коммунальных услуг будет обрабатывать 10 миллионов запросов в секунду ($R = 10\,000\,000$ запросов/сек), а время обработки одного запроса составляет 0,1 секунды ($T = 0.1$ сек/запрос).

Когда Manager получит сообщение PuLoginRequest ($m =$ PuLoginRequest). В таком случае Manager создаст актора Validator и увеличит r на 1. Таким образом уравнение будет выглядеть так (1):

$$f(\text{PuLoginRequest}, r) = (\text{Validator,PuDataRequest}, r + 1) \quad (1)$$

Затем актор Validator получает сообщение PuDataRequest ($m =$ PuDataRequest). В этом случает Validator делает запрос в БД и сохраняет таблицу по своему лицевому счету в свое состояние. Тут формула принимает следующий вид (2):

$$f(\text{PuDataRequest}, r) = (\text{Validator,PuCommit}, r) \quad (2)$$

Таким образом, мы можем использовать данную математическую модель для предсказания изменения производительность алгоритма проверки показаний в ответ на различные входные сообщения.

Вывод по разделу 2

В данном разделе мы описали требования к системе, определились с базой данных, построили архитектуру и составили математическую модель.

3 Реализация и тестирование программного обеспечения

3.1 Реализация программного обеспечения

Для разработки сервиса с алгоритмом проверки введенных показаний приборов учета коммунальных услуг, был выбран язык программирования Scala и фреймворк Akka для реализации модели акторов.

По сколько сервис по сути является – RESTful приложением, то для реализации необходимо определиться с технологией, которая сможет запустить Web – сервер, и будет снабжена поддержкой методов HTTP.

В фреймворке Akka есть модуль Akka HTTP. Он реализует полный серверный и клиентский HTTP – стек поверх akka-actors и akka-stream. Это не Web-фреймворк, а скорее более общий набор инструментов для построения и поддержки сервисов на основе HTTP [21]. Взаимодействие с браузером, конечно, тоже входит в сферу применения, но не является основной целью Akka HTTP [27][28][29]. За обработку методов HTTP, в алгоритме проверки показаний отвечает класс RestApi. На рисунке А.1 (Приложение А) представлен код данного класса.

RestApi использует актор Manager как прокси, для связи пользовательских запросов и последующих за ними вычислений [25][26]. Реализация актора Manager представлена на рисунке А.3 (Приложение А). За вычисления и обработку запросов отвечает актор Validator. Он обрабатывает запрос и возвращает ответ к пользователю. Актор Validator представлен на рисунке А.2 (Приложение А).

Вся разработка акторной модели базируется на грамотном построении логики отправки сообщений между акторами в системе [19][20]. Чтобы описать логику обработки этих сообщений, актору нужно прописать протокол, в котором будут описываться сами сообщения, которые данный актор может получать. Актор Validator будет принимать сообщения, после получения которых, выполняется:

– Запрос предыдущих показаний из базы данных;
– Валидация введенных показаний;
– Отправка успешно проверенных показаний в базу данных.

Протокол актора Validator, описывающий эти сообщения, изображен на рисунке 10.

```
  Владислав
  case class PuDataRequest()
  Владислав
  case class Validating(meter: Map[String, String], date: LocalDate)
  Владислав
  case class PuCommit(meter: Map[String, String], date: LocalDate)
}
```

Рисунок 10 – Протокол сообщений актора Validator

Сообщения актора Manager должны описывать следующее поведение актора:

– Запрос данных у валидатора с нужным лицевым счетом;
– Проверка введенных показаний у конкретного лицевого счета;
– Выдача пользователю информации о успешном сохранении новых показаний в базу данных;
– Выдача ошибки в случае ошибки в новых показаниях.

Протокол актора Manager показан на рисунке 11.

```
Владислав
case class PuLoginRequest(login: Int)
Владислав
case class PuValidCheck(login: Int, meter: Map[String, String], date: LocalDate)
Владислав
sealed trait ValidatorResponse
Владислав
case object ValidGood extends ValidatorResponse
Владислав
case class ValidBad(error: String) extends ValidatorResponse
```

Рисунок 11 – Протокол сообщений актора Manager

Для взаимодействия акторов, прописать протокол недостаточно. Необходимо реализовать логику обработки данных сообщений – поведение актора.

Реализация поведения актора Manager, предполагает под собой перераспределение запросов на акторы валидатора. При получении сообщений PuLoginRequest и PuValidCheck, актор перенаправляет эти запросы командой forward акторам Validator. Поведение актора Manager отображено на рисунке 12.

```
case PuLoginRequest(login) =>
  context.child(login.toString) match {
    case Some(value) =>
      value forward PuDataRequest
      println(s"MANAGER - data resent to $login validator")
    case None =>
      val valid = createValidator(login)
      println(s"MANAGER - validator $login initialized")
      valid forward PuDataRequest
      println(s"MANAGER - data resent to $login validator")
      println()
  }
case PuValidCheck(login, meter_map, date) => context.child(login.toString) match {
  case Some(value) =>
    value forward Validating(meter_map, date)
    println(s"MANAGER - resent VALIDATING to $login validator")
  case None =>
    val valid = createValidator(login)
    println(s"MANAGER - validator $login initialized")
    valid forward Validating(meter_map, date)
    println(s"MANAGER - resent VALIDATING to $login validator")
    println()
}
```

Рисунок 12 – Поведение актора Manager.

Тем временем, при получении актором Validator сообщения PuDataRequest, происходит запрос к базе через метод – all_meter_date(login), в который передается номер лицевого счета, ответ базы записывается в

промежуточное состояние и отправляется к пользователю. Протокол обработки данного сообщения показан на рисунке 13.

```
case PuDataRequest =>
  pu_date = all_meter_date(login)
  println(s"VALIDATOR - data from $login getting")
  sender() ! pu_date
  println(s"VALIDATOR data from $login was sent")
```

Рисунок 13 – Обработка сообщения PuDataRequest

При получении сообщения Validating, актор проверяет новые показания на соответствие требованиям, описанным в разделе 2. 1 и при наличии нарушений выбрасывает исключения. Блок-схема данного алгоритма представлена на рисунке Б.1 (Приложение Б). Если алгоритм выбросил исключение, то актору Manager отправляется сообщение ValidBad с указанием ошибки, а если новые показания прошли валидацию успешно, то актору Manager отправляется сообщение ValidGood, а также самому себе актор Validator отправляет сообщение PuCommit.

При получении сообщения PuCommit, актор Validator вызывает метод new_meter_add, в который обернуто взаимодействие с базой данных, для сохранения новый показаний. Обработка этого сообщения показана на рисунке 14.

```
case PuCommit(meter_map, date_of_reading) => new_meter_add(login, meter_map, date_of_reading)
```

Рисунок 14 – Обработка сообщения PuCommit

Приведенные выше протоколы и реализации поведения акторов, являются базисами, благодаря которым, программный продукт удовлетворяет всем требованиям, предъявленным к нему ранее.

3.2 Тестирование программного обеспечения

Для тестирования программного продукта использован метод функционального тестирования.

Функциональное тестирование – это метод тестирования программного обеспечения, который проверяет функциональность приложения или системы на соответствие заданным требованиям. Он включает в себя тестирование функций и функций программного обеспечения, чтобы убедиться, что оно работает должным образом и соответствует потребностям пользователя

Цель функционального тестирования – убедиться, что программная система выполняет все функции, которые она должна выполнять, и не выполняет функции, для которых она не предназначена.

Пользовательский интерфейс (UI) — это визуальная и интерактивная часть программного приложения или Web-сайта, которая позволяет пользователям взаимодействовать с системой. Он включает макет, дизайн и функциональность кнопок, меню, форм и других элементов, с которыми пользователи взаимодействуют для выполнения задач и достижения целей. Хорошо продуманный пользовательский интерфейс интуитивно понятен, прост в навигации и визуально привлекателен, что может повысить вовлеченность и удовлетворенность пользователей.

Первая вкладка (рисунок 15) отображает форму ввода лицевого счета для получения данных пользователя и его предыдущих показаний.

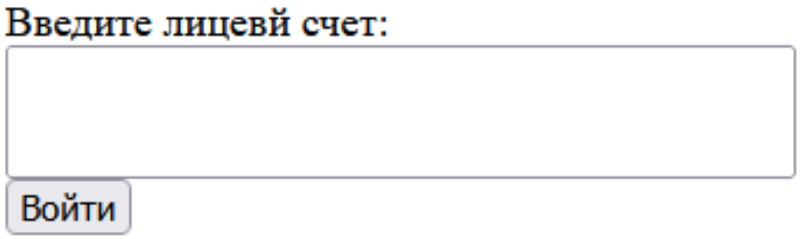


Рисунок 15 – Форма ввода лицевого счета

После ввода лицевого счета и последующего нажатия кнопки «Войти», открывается вкладка с выгруженными из БД данными по введенному лицевому счету (рисунок 16).

Наименование услуги	Серийный номер ПУ	Предыдущие показания 1	Дата предыдущих показаний 1	Предыдущие показания 2	Дата предыдущих показаний 2	Предыдущие показания 3	Дата предыдущих показаний 3
Теплоноситель (Горячая вода)	1124408	37.50000	2023-08-21	39.00000	2023-09-22	40.00000	2023-10-23
Холодная вода	1124529	259.00000	2023-08-22	266.00000	2023-09-22	270.00000	2023-10-23
Электроэнергия	52043483	20098.00000	2023-08-25	20196.00000	2023-09-25	20296.00000	2023-10-21
Отопление	499080	29.50000	2023-02-21	29.70000	2023-03-25	29.70000	2023-04-25

Ввести новые показания

Рисунок 16 – Предыдущие показания пользователя

Если пользователь намерен передать новые показания приборов учета коммунальных услуг, он нажимает кнопку «Ввести новые показания» и переходит на вкладку с формой ввода новых показаний (рисунок 17).

Под его лицевой счет, создается новый актор, либо берется существующий, если таковой уже использовался в системе акторов.

Введите показания счетчика

Теплоноситель (Горячая вода):

Холодная вода:

Электроэнергия:

Отопление:

Отправить

Рисунок 17 – Форма ввода новых показаний

Для проведения тестирования, будут введены произвольные данные, затем алгоритм должен проверить их на соответствие требованиям.

Для поля «Отопление», в целях тестирования критерия срока поверки счетчиков, было установлено значение – «2023-05-17».

Введем следующие показания:

– Теплоноситель (Горячая вода) – 44;
– Холодная вода – 276;
– Электроэнергия – 22222;
– Отопление – 40.

Все остальные показания, кроме показаний «Отопление», не должны выдать ошибки, единственная ошибка, которая должна появиться, это ошибка срока поверки. Результат показан на рисунке 18.

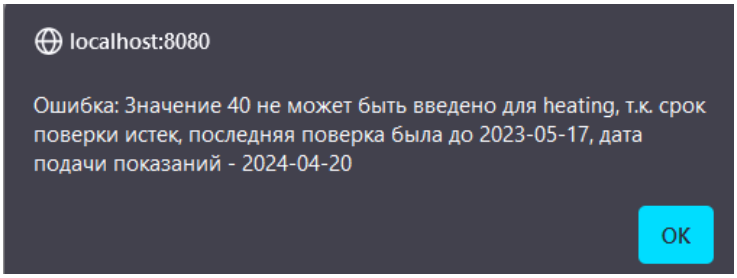


Рисунок 18 – Ошибка срока поверки счетчиков

При изменении срока поверки в БД на корректный, данные успешно сохраняются.

Если какое-то значение будет отрицательным, алгоритм выдаст ошибку, показанную на рисунке 19.

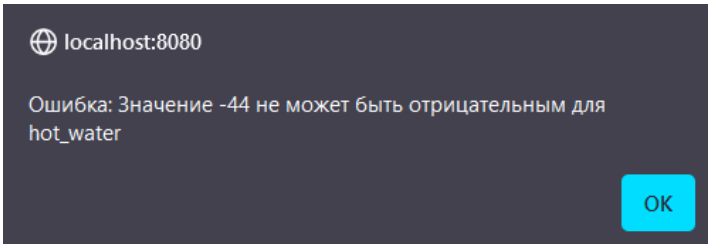


Рисунок 19 – Ошибка отрицательного значения

Если мы попробуем ввести значение меньше предыдущего, например для поля «Холодная вода», то выдаст ошибку, показанную на рисунке 20.

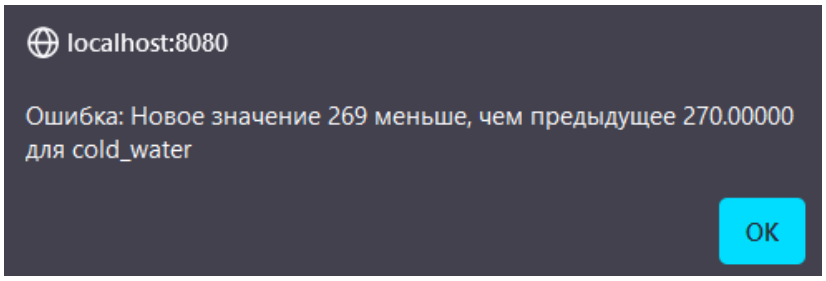


Рисунок 20 – Ошибка меньшего значения

Существует критерий, которые говорит о том, что новое введенное показание не должно быть больше максимального допустимого значение. Это максимально допустимое значение для каждого прибора учета свое, и вычисляется как большее значение двух показателей: среднее десятикратное потребление и нормативное значение.

Введем аномальное значение показание прибора учета – 999999, в поле «Электроэнергия» и получим следующую ошибку, показанную на рисунке 21.

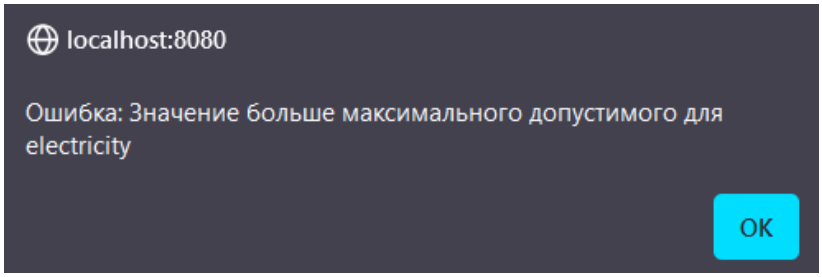


Рисунок 21 – Ошибка максимального допустимого значения

Также алгоритм должен проверять введенное количество цифр как до разделительной точки, так и после. Несоблюдение данного требование должно вызвать ошибку (рисунок 22, 23).

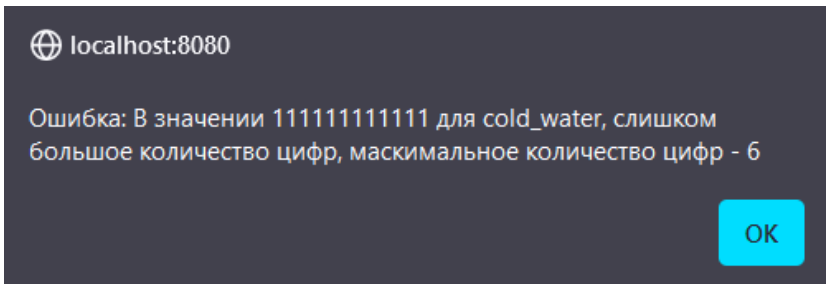


Рисунок 22 – Ошибка количества цифр до точки

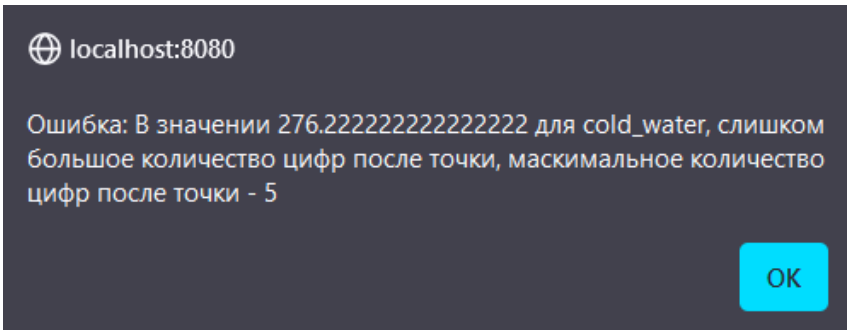


Рисунок 23 – Ошибка количества цифр после точки

Если новый расход превышает в 2 раза средний расход за последние 2 месяца, то всплывает следующее окно (рисунок 24).

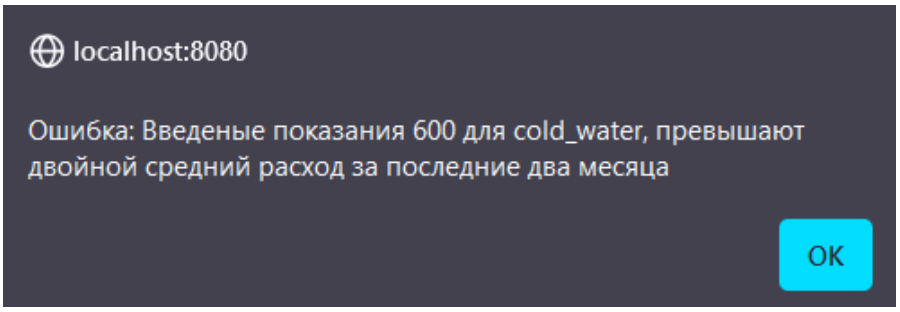


Рисунок 24 – Окно превышения среднего расхода

После проведения тестирования, были получены подтверждения работоспособности программного обеспечения.

Выводы по разделу 3

В заключительной части выпускной квалификационной работы было разработано и протестировано программное обеспечение. Для реализации этого ПО был выбран язык программирования Scala, который зарекомендовал себя как современный функциональный язык, прекрасно подходящий для создания масштабируемых и высокопроизводительных систем.

В качестве основной архитектурной концепции был использован фреймворк Akka, который предоставляет мощную абстракцию акторов. Эта модель акторов и асинхронных сообщений позволила организовать эффективное взаимодействие различных компонентов приложения, таких как модули валидации данных, работы с базой данных и взаимодействия с пользовательским интерфейсом.

Для реализации RESTful API был задействован модуль Akka HTTP. Это гибкое и производительное решение упростило задачи, связанные с обработкой HTTP-запросов, маршрутизацией, преобразованием данных и другими аспектами работы веб-сервисов.

В качестве системы управления базами данных был выбран PostgreSQL - мощная реляционная СУБД с широкими возможностями. Для сериализации данных использовался фреймворк SprayJson, который органично интегрируется со Scala и обеспечивает эффективную передачу данных между различными компонентами системы.

После реализации программного обеспечения оно было протестировано. Были проверены требования по валидации вводимых показаний приборов учета коммунальных услуг, обеспечена стабильная работа пользовательского интерфейса и взаимодействие с базой данных. Таким образом, разработанное ПО продемонстрировало свою работоспособность и соответствие поставленным задачам.

Заключение

Выпускная квалификационная работа была посвящена разработке алгоритма для проверки введенных показаний приборов учета коммунальных услуг.

В ходе выполнения ВКР была проведена исследовательская работа, в рамках которой были изучены существующие методы разработки веб-приложений. Особое внимание было уделено сравнительному анализу подходов к созданию высоконагруженных приложений с использованием асинхронного программирования, параллелизма и методов масштабирования. На основе этого анализа были определены наиболее эффективные решения для реализации поставленной задачи.

Далее были сформулированы четкие требования к алгоритму валидации введенных показаний, а также разработана архитектура взаимодействия различных сущностей в системе. Кроме того, была выведена математическая модель для прогнозирования эффективности работы алгоритма проверки данных, построенного на основе модели акторов.

В заключительной части работы было разработано и всесторонне протестировано программное обеспечение. Для реализации этого ПО был выбран язык программирования Scala, который использовался в связке с фреймворком Akka и модулем Akka HTTP. Это позволило организовать эффективное взаимодействие различных компонентов приложения, таких как модули валидации данных, работы с базой данных и взаимодействия с пользовательским интерфейсом.

В качестве системы управления базами данных был выбран PostgreSQL, а для сериализации данных использовался фреймворк SprayJson. Такое сочетание технологий обеспечило надежное хранение и передачу данных в рамках построенного RestAPI-приложения.

В результате выполнения выпускной квалификационной работы было разработано программное обеспечение, решающее две основные задачи:

– Создание масштабируемого сервиса для приема показаний приборов учета коммунальных услуг. Данный сервис позволяет эффективно обрабатывать большие объемы данных, поступающих от абонентов, благодаря использованию современных подходов к разработке высоконагруженных приложений, таких как асинхронное программирование, параллелизм и масштабирование.
– Реализация эффективного алгоритма проверки (валидации) введенных пользователями показаний. Разработанный алгоритм, основанный на математической модели, позволяет выявлять ошибки и несоответствия в данных, обеспечивая высокую достоверность информации, поступающей в систему учета коммунальных услуг.

Таким образом, в ходе выполнения выпускной квалификационной работы были полностью решены все поставленные задачи, что позволило достичь основной цели - создания программного обеспечения, отвечающего современным требованиям к масштабируемости, производительности и качеству данных в области учета коммунальных услуг.

Список используемой литературы

1. Готовим полнотекстовый поиск в Postgres. Часть 1 [Электронный ресурс] // Хабр. URL: http://habr.com/ru/articles/442170/. – (дата обращения: 01.01.2024).
2. Изучаем Scala [Электронный ресурс] // Scala Documentation. URL: http://docs.scala-lang.org/ru/. – (дата обращения: 10.03.2024).
3. Модель акторов в компьютерных науках [Электронный ресурс] // Wikipedia. URL: http://en.wikipedia.org/wiki/Actor_model. – (дата обращения: 03.01.2024).
4. Начало работы [Электронный ресурс] // Scala Documentation. URL: http://docs.scala-lang.org/ru/getting-started/index.html. – (дата обращения: 28.02.2024).
5. Онлайн курсы (MOOCs) от Scala Center [Электронный ресурс] // Scala Documentation. URL: http://docs.scala-lang.org/ru/online-courses.html. – (дата обращения: 22.02.2024).
6. Полная документация PostgreSQL на русском языке [Электронный ресурс] // Postgres Professional. URL: http://medium.com/postgres-professional/%D0%BF%D0%BE%D0%BB%D0%BD%D0%B0%D1%8F-%D0%B4%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%86%D0%B8%D1%8F-postgresql-%D0%BD%D0%B0-%D1%80%D1%83%D1%81%D1%81%D0%BA%D0%BE%D0%BC-%D1%8F%D0%B7%D1%8B%D0%BA%D0%B5-fd418c1a151. – (дата обращения: 22.02.2024).
7. Рестенбург, Баккер, Уильямс: Akka в действии. Manning Publications.
8. AKKA definition and meaning [Электронный ресурс] // Collins English Dictionary. URL: http://www.collinsdictionary.com/dictionary/english/akka. – (дата обращения: 18.02.2024).

9. Akka HTTP [Электронный ресурс] // Akka Documentation. URL: http://doc.akka.io/docs/akka-http/current. – (дата обращения: 12.02.2024).

10. Akka HTTP [Электронный ресурс] // Akka Documentation. URL: http://doc.akka.io/docs/akka-http/current. – (дата обращения: 15.03.2024).

11. Akka HTTP [Электронный ресурс] // GitHub. URL: http://github.com/akka/akka-http. – (дата обращения: 20.03.2024).

12. Akka HTTP [Электронный ресурс] // GitHub. URL: http://github.com/akka/akka-http. – (дата обращения: 12.01.2024).

13. Akka Resources [Электронный ресурс] // Lightbend. URL: http://www.lightbend.com/akka/resources. – (дата обращения: 07.01.2024).

14. Akka Resources [Электронный ресурс] // Lightbend. URL: http://www.lightbend.com/akka/resources. – (дата обращения: 20.02.2024).

15. Akka: Сборка высоко параллельных, распределенных и устойчивых … [Электронный ресурс] // Akka. URL: http://akka.io. – (дата обращения: 18.02.2024).

16. Developing Reactive Microservices with Akka HTTP [Электронный ресурс] // Reintech. URL: http://reintech.io/blog/developing-reactive-microservices-with-akka-http. – (дата обращения: 07.01.2024).

17. GitHub - spray/spray-json: A lightweight, clean and simple JSON … [Электронный ресурс] // GitHub. URL: http://github.com/spray/spray-json. – (дата обращения: 18.02.2024).

18. HTTP Model [Электронный ресурс] // Akka HTTP. URL: http://doc.akka.io/docs/akka-http/current/common/http-model.html. – (дата обращения: 10.03.2024).

19. Learn Scala [Электронный ресурс] // Scala Documentation. URL: http://docs.scala-lang.net/. – (дата обращения: 03.01.2024).

20. Online Courses (MOOCs) from The Scala Center [Электронный ресурс] // Scala Documentation. URL: http://docs.scala-lang.org/online-courses.html. – (дата обращения: 22.02.2024).

21. Online Resources [Электронный ресурс] // Scala Documentation. URL: http://docs.scala-lang.org/learn.html. – (дата обращения: 01.01.2024).

22. PostgreSQL: The world’s most advanced open source database [Электронный ресурс] // PostgreSQL. URL: http://www.postgresql.org. – (дата обращения: 01.01.2024).

23. Resources [Электронный ресурс] // Akka.NET Documentation. URL: http://getakka.net/community/online-resources.html. – (дата обращения: 28.02.2024).

24. spray-json [Электронный ресурс] // GitHub. URL: http://github.com/spray/spray-json. – (дата обращения: 10.03.2024).

25. spray-json Support [Электронный ресурс] // spray. URL: http://spray.readthedocs.io/en/latest/documentation/spray-httpx/spray-json-support.html. – (дата обращения: 03.01.2024).

26. spray-json Support [Электронный ресурс] // spray. URL: http://spray.readthedocs.io/en/latest/documentation/spray-httpx/spray-json-support.html. – (дата обращения: 15.03.2024).

27. Terminology and Concepts [Электронный ресурс] // Akka.NET Documentation. URL: http://getakka.net/articles/concepts/terminology.html. – (дата обращения: 22.02.2024).

28. Terminology and Concepts [Электронный ресурс] // Akka.NET Documentation. URL: http://getakka.net/articles/concepts/terminology.html. – (дата обращения: 12.01.2024).

29. The Streaming-first HTTP server/module of Akka [Электронный ресурс] // GitHub. URL: http://github.com/akka/akka-http. – (дата обращения: 20.02.2024).

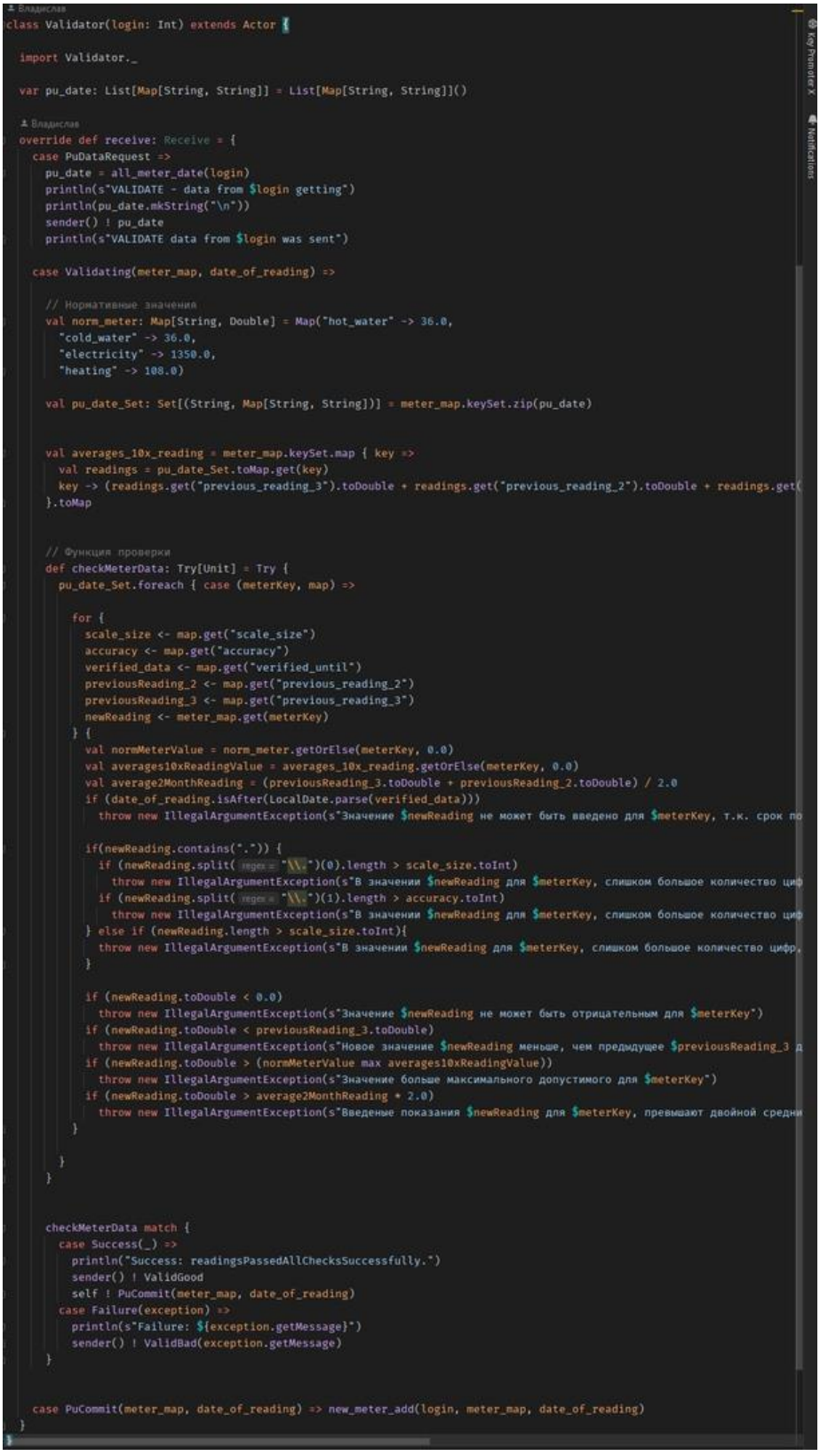
30. Understanding the Actor Model [Электронный ресурс] // Hardly Working. URL: http://hardlyworking.dev/posts/understanding-the-actor-model/. – (дата обращения: 07.01.2024).

Приложение А

Главные сущности программы

```
class RestApi(system: ActorSystem, timeout: Timeout) extends RestRoutes {

  implicit val requestTimeout: Timeout = timeout

  Владислав
  implicit def executionContext: ExecutionContextExecutor = system.dispatcher

  Владислав
  override def createManager(): ActorRef = {
    val man = system.actorOf(Manager.props, Manager.name)
    println("MANAGER initialized")
    man
  }


}

Владислав *
trait RestRoutes extends ManagerApi with MeterMarshalling {

  import StatusCodes._

  Владислав
  def routes: Route = authenticate ~ user_data_route ~ add_new_meter_date


  val authenticate: Route =
    path( pm = "user" / "authentication") {
      get {
        complete(
          HttpEntity(ContentTypes.`text/html(UTF-8)`,
            loginUser.user_login
          )
        )
      }
    }
  val user_data_route: Route = {
    path( pm = "user" / Segment) { login =>
      get {
        onSuccess(puLoginRequest(login.toInt)) { result =>
          complete(OK,
            HttpEntity(ContentTypes.`text/html(UTF-8)`,
              user_login_data.jsonToTable(result, login.toInt)))
        }
      }
    }
  }


  val add_new_meter_date: Route = {
    pathPrefix( pm = "user" / Segment) { login =>
      path( pm = "add-meter") {
        get {
          complete(
            HttpEntity(ContentTypes.`text/html(UTF-8)`,
              add_meter_data.add_to_validator(login)
            )
          )
        }
      } ~
        path( pm = "validating") {
          post {
            entity(as[MeterData]) { meterData =>
              val dataMap: Map[String, String] = Map(
                "hot_water" -> meterData.hot_water,
                "cold_water" -> meterData.cold_water,
                "electricity" -> meterData.electricity,
                "heating" -> meterData.heating
              )
              val day_of_reading: LocalDate = LocalDate.now()
              println(s"map was get $dataMap ")


              onSuccess(puValidCheck(login.toInt, dataMap, day_of_reading)) {
                case Manager.ValidGood => complete(StatusCodes.OK -> dataMap)
                case Manager.ValidBad(exception) =>
                  complete(HttpResponse(StatusCodes.BadRequest, entity = exception))
              }

            }

          }
        }
    }
  }
```

Рисунок А.1 – Сущность RestApi



```
class Validator(login: Int) extends Actor {

  import Validator._

  var pu_date: List[Map[String, String]] = List[Map[String, String]]()

  override def receive: Receive = {
    case PuDataRequest =>
      pu_date = all_meter_date(login)
      println(s"VALIDATE - data from $login getting")
      println(pu_date.mkString("\n"))
      sender() ! pu_date
      println(s"VALIDATE data from $login was sent")

    case Validating(meter_map, date_of_reading) =>

      // Нормативные значения
      val norm_meter: Map[String, Double] = Map("hot_water" -> 36.0,
        "cold_water" -> 36.0,
        "electricity" -> 1350.0,
        "heating" -> 108.0)

      val pu_date_Set: Set[(String, Map[String, String])] = meter_map.keySet.zip(pu_date)


      val averages_10x_reading = meter_map.keySet.map { key =>
        val readings = pu_date_Set.toMap.get(key)
        key -> (readings.get("previous_reading_3").toDouble + readings.get("previous_reading_2").toDouble + readings.get(
      }.toMap


      // Функция проверки
      def checkMeterData: Try[Unit] = Try {
        pu_date_Set.foreach { case (meterKey, map) =>

          for {
            scale_size <- map.get("scale_size")
            accuracy <- map.get("accuracy")
            verified_data <- map.get("verified_until")
            previousReading_2 <- map.get("previous_reading_2")
            previousReading_3 <- map.get("previous_reading_3")
            newReading <- meter_map.get(meterKey)
          } {
            val normMeterValue = norm_meter.getOrElse(meterKey, 0.0)
            val averages10xReadingValue = averages_10x_reading.getOrElse(meterKey, 0.0)
            val average2MonthReading = (previousReading_3.toDouble + previousReading_2.toDouble) / 2.0
            if (date_of_reading.isAfter(LocalDate.parse(verified_data)))
              throw new IllegalArgumentException(s"Значение $newReading не может быть введено для $meterKey, т.к. срок по

            if(newReading.contains(".")) {
              if (newReading.split( regex = "\\.")(0).length > scale_size.toInt)
                throw new IllegalArgumentException(s"В значении $newReading для $meterKey, слишком большое количество ци
              if (newReading.split( regex = "\\.")(1).length > accuracy.toInt)
                throw new IllegalArgumentException(s"В значении $newReading для $meterKey, слишком большое количество ци
            } else if (newReading.length > scale_size.toInt){
              throw new IllegalArgumentException(s"В значении $newReading для $meterKey, слишком большое количество цифр,
            }

            if (newReading.toDouble < 0.0)
              throw new IllegalArgumentException(s"Значение $newReading не может быть отрицательным для $meterKey")
            if (newReading.toDouble < previousReading_3.toDouble)
              throw new IllegalArgumentException(s"Новое значение $newReading меньше, чем предыдущее $previousReading_3 д
            if (newReading.toDouble > (normMeterValue max averages10xReadingValue))
              throw new IllegalArgumentException(s"Значение больше максимального допустимого для $meterKey")
            if (newReading.toDouble > average2MonthReading * 2.0)
              throw new IllegalArgumentException(s"Введеные показания $newReading для $meterKey, превышают двойной средни
            }

          }
        }


      checkMeterData match {
        case Success(_) =>
          println("Success: readingsPassedAllChecksSuccessfully.")
          sender() ! ValidGood
          self ! PuCommit(meter_map, date_of_reading)
        case Failure(exception) =>
          println(s"Failure: ${exception.getMessage}")
          sender() ! ValidBad(exception.getMessage)
      }


    case PuCommit(meter_map, date_of_reading) => new_meter_add(login, meter_map, date_of_reading)
  }
}
```

Рисунок А.2 – Актор Validator

```scala
Владислав
object Manager {
  val name = "Manager"

  Владислав
  def props(implicit timeout: Timeout): Props = Props(new Manager())


  Владислав
  case class PuLoginRequest(login: Int)

  Владислав
  case class PuValidCheck(login: Int, meter: Map[String, String], date: LocalDate)

  Владислав
  case class PuLoginCommit(login: Int)

  Владислав
  sealed trait ValidatorResponse

  Владислав
  case object ValidGood extends ValidatorResponse

  Владислав
  case class ValidBad(error: String) extends ValidatorResponse


}

Владислав
class Manager(implicit timeout: Timeout) extends Actor {

  import Manager._


  Владислав
  def createValidator(login: Int): ActorRef = context.actorOf(Validator.props(login), login.toString)

  Владислав
  override def receive: Receive = {
    case PuLoginRequest(login) =>
      context.child(login.toString) match {
        case Some(value) =>
          value forward PuDataRequest
          println(s"MANAGER - data resent to $login validator")
        case None =>
          val vali = createValidator(login)
          println(s"MANAGER - validator $login initialized")
          vali forward PuDataRequest
          println(s"MANAGER - data resent to $login validater")
          println()

      }


    case PuValidCheck(login, meter_map, date) => context.child(login.toString) match {
      case Some(value) =>
        value forward Validating(meter_map, date)
        println(s"MANAGER - resent VALIDATING to $login validator")
      case None =>
        val vali = createValidator(login)
        println(s"MANAGER - validator $login initialized")
        vali forward Validating(meter_map, date)
        println(s"MANAGER - resent VALIDATING to $login validat0r")
        println()

    }
    case PuLoginCommit(login) => ???
  }
}
```

Рисунок А.3 – Актор Manager

Приложение Б

Блок схема алгоритма проверки показаний

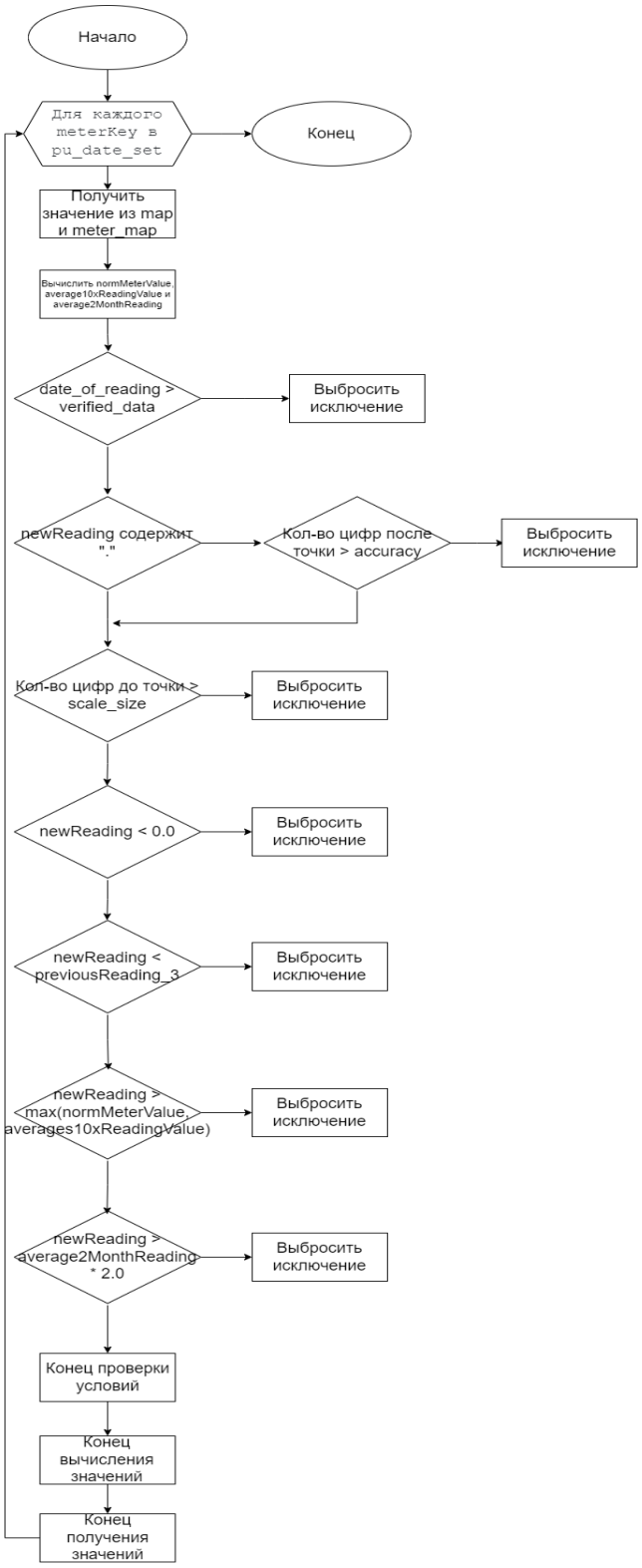


Рисунок Б.1 – Блок схема алгоритма