МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «КубГУ»)

Факультет компьютерных технологий и прикладной математики
Кафедра анализа данных и искусственного интеллекта

КУРСОВАЯ РАБОТА

РАЗРАБОТКА МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ НА JAVA С ИСПОЛЬЗОВАНИЕМ ФРЕЙМВОРКА SPRING

Работу выполнил _____ М.В. Сидоренко
(подпись)

Направление подготовки 02.03.03 Математическое обеспечение и администрирование информационных систем курс 3

Направленность (профиль) Технология программирования

Научный руководитель
канд. физ.-мат. наук, доц. _____ Г.В. Калайдина
(подпись)

Нормоконтролер
канд. физ.-мат. наук, доц. _____ Г.В. Калайдина
(подпись)

Краснодар
2023

РЕФЕРАТ

Курсовая работа 88 с., 14 рис., 23 источн., 6 прил.

МИКРОСЕРВИСНАЯ АРХИТЕКТУРА, JAVA, SPRING, SPRING BOOT, SPRING CLOUD, КЭШИРОВАНИЕ ДАННЫХ, REDIS, KAFKA, RABBITMQ REACT.JS, REACT NATIVE, RESTFUL API, POSTGRESQL, MONGODB

Объектом исследования данной работы является разработка и проектирование микросервисной архитектуры на языке программирования Java с использованием фреймворка Spring, а также дополнительного стека технологий, необходимого для поддержания высоконагруженной системы.

Целью работы является разработка эффективной, масштабируемой и отказа устойчивой системы, основанной на принципах микросервисной архитектуры, которая сможет обеспечить бесперебойное уведомления пользователей о чрезвычайных ситуациях.

В процессе работы использованы UML-диаграммы для визуализации и описания системы. А также приложение Figma, для создания дизайна пользовательского интерфейса. Системный стек: Java, Spring, Redis, Kafka, Docker, Kubernetes. Для лучшего проектирования приложения использована методология Domain-Driven Design (DDD). Взаимодействие между микросервисами будет осуществляться посредством RESTful API. Используемые базы данных PostgreSQL, MongoDB. Для разработки веб-клиента выбран фреймворк React и React native для мобильного приложения.

В результате выполнения данной работы разработана эффективная и масштабируемая информационная система, основанная на принципе микросервисной архитектуры, способная обеспечить бесперебойное уведомление пользователей о чрезвычайных ситуациях. Весь процесс проектирования задокументирован и описан в научной работе.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном информационном обществе остро встает вопрос об оповещении пользователей о чрезвычайных ситуациях. Быстрая доставка информации играет критическую роль для защиты жизни и здоровья людей, а также для минимизации материальных потерь. Кризисные ситуации, будь то природные катаклизмы или человеческие действия, могут застать врасплох каждого. В связи с этим возникает необходимость в разработке эффективной системы уведомлений.

Такая система должна быть быстрой и отказоустойчивой, ведь от её работы могут зависеть жизни. Именно поэтому так важно качественно подойти к разработке её архитектуры.

Основной целью данной научной работы являлось проектирование микросервисной архитектуры, способной выдержать большие нагрузки, и обеспечить пользователей оперативной доставкой уведомлений о чрезвычайных ситуациях, а также разработка пользовательского интерфейса, который будет удобен и интуитивно понятен каждому.

Для достижения поставленной цели необходимо было выполнить следующие 10 задач:

1) Изучить основы микросервисной архитектуры и принципы ее организации.

2) Провести аналитику и декомпозицию предметной области информационной системы.

3) Исследовать фреймворк Spring и его роль в разработке микросервисов.

4) Спроектировать работу каждого сервиса.

5) Разобраться с паттернами Энтерпрайз разработки, для правильной организации дизайна системы.

6) Разработать правильную организацию слоя данных.

7) Разработать функциональность кэширования данных с использованием технологии Redis.

8) Исследовать брокеры сообщений, и организовать общение сервисов через них.

9) Изучить роль инфраструктуры и экосистемных технологий в разрботке микросервисных архитектур.

10) Изучить паттерны UI/UX для проектирования современного пользовательского интерфейса.

Теоретическая значимость данной работы заключается в исследовании принципов организации микросервисной архитектуры на Java с использованием фреймворка Spring, а также других инфраструктурных технологий необходимых для разработки такой потенциально высоконагруженной системы. Полученные результаты будут использованы в дальнейшем при реализации данной системы. Кроме того, навыки и знания, полученные в процессе исследования, буду полезны для проектирования других подобных систем.

Практическая значимость работы состоит в возможности создания удобной платформы для обмена уведомлениями о чрезвычайных ситуациях, способной оперативно доставлять информацию пользователям. Разработанное приложение может быть использовано различными организациями и государственными структурами, занимающимися обеспечением безопасности и защитой населения.

В данной работе будет использована методология Domain-Driven Design (DDD) и микросервисный подход с применением фреймворка Spring. Также планируется использование нескольких баз данных для правильно распределённой организации хранения информации о пользователях и шаблонах уведомлений.

В следующих главах будут более подробно рассмотрены основные аспекты разработки и масштабирования высоконагруженных распределённых систем.

1 Анализ рынка

В первой главе основной части работы приведены результаты анализа как мирового, так и Российского рынка систем оповещающих пользователей о чрезвычайных ситуациях. Разобрано подробно с какими плюсами и минусами сталкиваются пользователи этих систем.

В первой части первой главы представлено несколько международных аналогов, таких как FEMA Alert System в США, J-Alert в Японии и EU-Alert в странах Европейского союза.

Во второй части этой главы исследование сосредоточено на российских аналогах, таких как система-112, которая объединяет различные службы экстренного реагирования, и система оповещения населения об опасностях, разработанная МЧС России. Обе системы предоставляют оповещение о чрезвычайных ситуациях через различные каналы связи, такие как SMS, громкоговорители, сирены и телевизионные и радиоэфиры.

Затем в заключении первой главы, описано чем отвечает разрабатываемое приложение на плюсы и минусы систем, рассмотренных выше.

В итоге, данная глава основной части работы позволит точно оценить преимущества и недостатки проектируемого приложения по сравнению с аналогами, а также выбрать оптимальный подход к его реализации.

1.1 Международные аналоги

Так как одной из основных функций государства являться защита своего народа, то не удивительно, что при анализе рынка каких-либо систем, связанных с оповещением о чрезвычайных ситуациях, было обнаружено, что крупнейшие системы такого рода, являются на прямую или косвенно государственными разработками, а у любых государственных программ, в

отличие от бизнес-решений, ограниченное финансирование, что в последствие выливается в известные всем недостатки.

1.1.1 FEMA (Federal Emergency Management Agency) Alert System

Федеральное агентство по чрезвычайным ситуациям – американская система оповещения о чрезвычайных ситуациях. Система использует SMS, телевизионные и радиоэфиры для отправки оповещений.

Когда вы добавляете новое местоположение для получения оповещений, вы получаете полный контроль над типами оповещений для мониторинга: наводнения, затопления прибрежных районов и озер, суровые погодные условия (грозы и торнадо), тропическая погода (ураганы и тайфуны), зимняя погода (снег, лед, ледяной дождь), лавины, пожары, экстремальные температуры, морская погода, оповещения об общественных опасностях и многое другое.

В отличие от большинства аналогичных приложений, FEMA также предоставляет экстренные оповещения об эвакуации, гражданской опасности, похищениях детей, опасных материалах, атомных электростанциях, радиационной опасности, перебоях в работе телефона 911, беспорядках, взрывах и многом другом.

FEMA также является приложением для обеспечения готовности к стихийным бедствиям, поскольку здесь представлены советы по технике безопасности в чрезвычайных ситуациях, напоминания о тестировании дымовых сигнализаций и обновлении аварийных комплектов, ресурсы для стихийных бедствий, такие как убежища, и многое другое.

Основным недостатком этой системы, как и большинства подобных, является ограничение в использовании только для оповещений на территории США. Помимо того, пользователи жалуются на устаревший и не удобный пользовательский интерфейс (Рисунок 1.1). В добавок отметим наличие

только старых вариантов оповещения, которые не всегда удобны в современных реалиях.

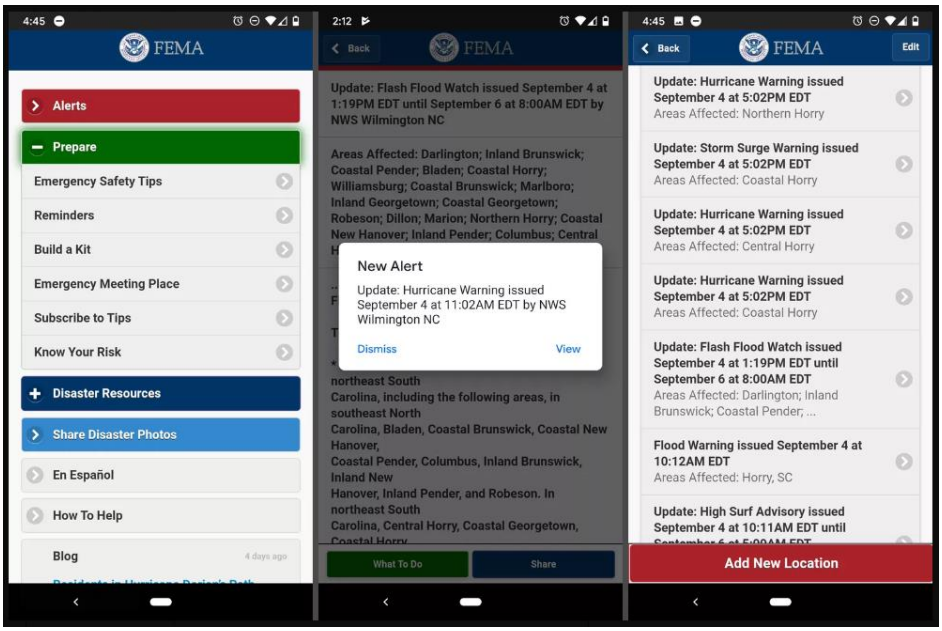

Рисунок 1.1 – Интерфейс приложения «FEMA»

1.1.2 J-Alert

Система оповещения о чрезвычайных ситуациях на базе спутника в Японии. Использует различные каналы связи, такие как мобильные телефоны, телевизионные и радиоэфиры, чтобы доставлять сообщения о надвигающихся опасностях.

Из плюсов можно выделить удивительную скорость реагирования, что очень полезно в сейсмически активном регионе, таком как Япония.

На бумаге эта система должна была быть внедрена очень быстро и повсеместно, но на деле, из-за большой консервативности японской культуры, какой бы развитой страной она не была, со всем, где государство встречается с цифровизацией, в Японии очень плохо. Ни о каком удобстве использования не может быть и речи. Приложений или веб-интерфейсов для пользовательского доступа к системе нет, используются аналоговые варианты связи.

1.1.3 EU-Alert

Система оповещения о чрезвычайных ситуациях в странах Европейского союза. Она основана на стандарте Cell Broadcast (трансляция по сотовой связи) и позволяет доставлять сообщения о чрезвычайных ситуациях на сотовые телефоны пользователей в заданной географической области.

Является стандартом схожим с Американской системой оповещения. Следовательно, имеет схожие отрицательные стороны, но помимо минусов, заимствованных у заокеанских коллег, имеет свои уникальные недостатки. Например, полное отсутствие какого-либо пользовательского интерфейса, оповещения приходят по SMS.

Нельзя не отметить, что у большинства европейских стран, есть своя реализация данного стандарта. Не углубляясь в каждую, можно утверждать, что имея свои локальные плюсы и минусы, эти системы, в общем и целом, склонны иметь одни и те же минусы, описанные выше.

1.1.4 Nextdoor

Заключительная в этой главе международная система, которая представляет собой социальную сеть, основанную на место положении. В отличие от аналогов, описанных выше, является не государственной и более современной разработкой.

Приложение имеет приятный и интуитивно понятный пользовательский интерфейс (Рисунок 1.2), публиковать экстремальные оповещения может любой желающий.

Из минусов: приложение не является строго организованной системой, для оповещения пользователей, а скорее является платформой для удобного уведомления соседей, или рассылкой любой информации на конкретный район.

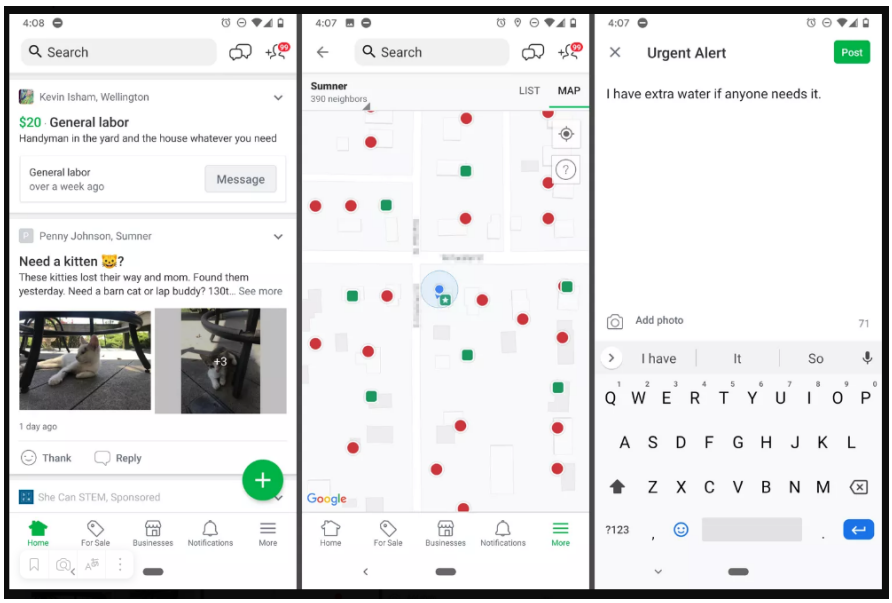

Рисунок 1.2 – Интерфейс приложения «Nextdoor»

1.2 Российские системы оповещения

Проанализировав российский рынок, столкнулся с ещё более не воодушевляющими результатами. Как и в консервативной Японии, у нас тоже есть только государственные программы, реализующие оповещение пользователей о чрезвычайных ситуациях. Как и ожидалось, эти системы очень далеки от необходимой современной цифровизации.

1.2.1 Система-112

Российская система оповещения о чрезвычайных ситуациях, которая объединяет различные службы экстренного реагирования. Система безопасности 112 – это единая система экстренного обслуживания, которая предназначена для приема и обработки вызовов. Система основана на использовании телефонного номера 112, который можно набрать для вызова экстренных служб.

Очевидно, этих мер недостаточно в современном мире для обеспечения точечного, быстрого и постоянного реагирования.

1.2.2 Система оповещения населения об опасностях

Система, разработанная МЧС России, которая обеспечивает оповещение населения о предстоящих опасностях через различные каналы связи, такие как SMS, громкоговорители, сирены и телевизионные и радиоэфиры. Система предоставляет информацию о пожарах, наводнениях, землетрясениях и других чрезвычайных ситуациях.

Здесь мы снова сталкиваемся с теми же минусами, что уже видели во всех устаревших государственных системах по всему миру.

1.3 Итоги анализа рынка

В результате анализа мирового и российского рынка систем оповещения о чрезвычайных ситуациях, можно сделать следующие выводы:

а) Государственные системы по обеспечению оповещения о чрезвычайных ситуациях имеют ограниченное финансирование, что ведет к ограниченным возможностям обновления и улучшения этих систем. Это приводит к устареванию функционала и неудобству использования для пользователей.

б) Международные системы оповещения, такие как FEMA Alert System в США и J-Alert в Японии, имеют ряд полезных функций, таких как быстрая реакция на чрезвычайные ситуации и использование различных каналов связи. Однако они также имеют свои недостатки, включая ограничение в использовании только на определенной территории и неудобный пользовательский интерфейс.

в) Социальные сети, такие как Nextdoor, имя удобный и понятный интерфейс, могут служить платформой для удобного уведомления соседей о локальных ситуациях, но они не являются организованной системой оповещения.

г) Российские системы экстренного реагирования, включая Систему-112 и систему оповещения населения об опасностях, также имеют ряд недостатков, включая ограниченные возможности реагирования и устаревший функционал.

В целом, существующие системы оповещения о чрезвычайных ситуациях имеют приличное количество недостатков, что заставляет тревожно задумываться, а что будет, если вдруг случиться ЧС. Ведь такие ситуации – явление редкое, именно поэтому наши системы недостаточно проверены в боевых условиях. Такие упущения могут по влечь за собой большие жертвы, поэтому необходимо улучшать работу оповещения до того, как мы убедимся на практике, что наши системы работают недостаточно качественно.

Разработка нового приложения должна учитывать все эти недостатки и стремиться решить их, обеспечивая удобство использования для пользователей и быструю реакцию на чрезвычайные ситуации.

1.4 Современный ответ на старую проблему

Проведя детальный анализ рынка, вывод о необходимости развития данной области напрашивается сам. Успешно разработанная информационная система – это та, которая полностью отвечает всем запросом пользователя. Именно поэтому при проектировании данного приложения я полагался на Domain-Driven Design (DDD) - методологию проектирования и разработки программного обеспечения, которая ставит предметную область цифровизируемой модели в центр внимания [1].

Семь основных требований исходя из плюсов и минусов аналогов:

1) Наличие доступа к системе через веб-интерфейс или мобильное приложение.

2) Приятный для использования и интуитивно понятный пользовательский интерфейс.

3) Современный методы получения уведомления.

4) Централизованная общая глобальная система, но при этом возможность точно настраиваемых локальных оповещений.

5) Возможность пользователям создавать свои локальные оповещения, для поддержания ещё более точного и оперативного сообщения о чрезвычайной ситуации.

6) Наличие некого сообщества, для более активного социального взаимодействия.

7) Быстрая и бесперебойная доставка уведомлений.

Все вышеперечисленные пункты являются результатом прямого анализа фичей, которые нравились пользователям предыдущих систем, а также анализа недостатков этих систем, и поиска их альтернатив.

При этом, эти 7 пунктов, легли в основу формирования разрабатываемой мной системы. Это приложение будет представлять собой платформу для удобной организации оповещений граждан любой страны, любого города и района. Система буде иметь такую структуру, что будет удобна в независимости от местонахождения пользователя и зоны мониторинга сообществ. И будет позволять реализовывать как глобальные рассылки на уровне страны, так и локальные рассылки для пользователя на уровне его семьи.

Суть проста – система основана на следующих трёх принципах социальной сети:

1) В системе есть две основные сущности с разными возможностями:

а) Первая сущность – пользователь. В его возможности входит добавление в друзья других пользователей, объединение их в группы для удобных рассылок, создание своих локальных шаблонов рассылок. Ещё более важная возможность, подписаться на сообщество, которое мониторит конкретную чрезвычайную ситуацию на конкретной локации (это может быть страна, а может конкретный город).

б) Вторая сущность – сообщество. Это частная или государственная организация, которая занимается мониторингом различных чрезвычайных

ситуаций на различных локациях. Они являются основой безопасности большого кол-ва граждан, ведь делают важные рассылки на всю свою аудиторию.

2) Такая организация системы позволяет ей быть удобной как для пользователей, так и для организаций, и более того, позволяет стать всем ныне существующим государственным системам своей частью, и заполучить в свои руки возможность не только аналогового уведомления, но также и такие варианты как: пуш-уведомления, уведомления через чат-боты, уведомления по электронной почте.

3) При этом система даёт пользователям возможность самим поучаствовать в организации своей безопасности, а также безопасности окружающих, ведь бывают ситуации, когда проблема очень локальна, и о ней вам скорее сообщит ваш сосед, чем какая-то организация.

Поэтому в следующих главах я подробно разберу весь используемый инструментарий, необходимый для построения дизайна такой серьёзной системы, а также пошагово пройдусь по этапам её разработки.

2 Анализ и выбор актуальных технологий

В данной главе задокументирован процесс проведения анализа существующих технологий и методик, необходимых для разработки информационной системы основанной на микросервисной архитектуре. Цель этого этапа проектирования состояла в том, чтобы выбрать наиболее подходящие и актуальные технологии для разрабатываемого приложения, которые будут обеспечивать ее эффективную работу и масштабируемость.

В ходе анализа рассматривались различные аспекты, начиная с выбора языка программирования, который является фундаментом разработки, заканчивая инфраструктурными технологиями, для удобного развёртывания и дальнейшей поддержки этой системы. Ещё одним важным аспектом является выбор брокера данных, который обеспечивает коммуникацию между сервисами и поддерживает асинхронную обработку сообщений. Также были изучены вопросы выбора базы данных и работы с кэшем, так как эти компоненты играют важную роль в эффективной работе микросервисных систем.

В результате этих исследований были определенны наиболее подходящие технологии и инструменты для разрабатываемого приложения.

2.1 Архитектура системы

Прежде чем перейти к конкретным технологиям и техническим аспектам, важно было разобраться с общими принципами и методиками проектирования микросервисных систем.

Micro Service Architecture (MSA) – Микросервисная архитектура – принципиальная организация распределенной системы на основе микросервисов и их взаимодействия друг с другом и со средой по сети, а также принципов, направляющих проектирование архитектуры, её создание и эволюцию [1].

Такая архитектура предлагает подход, при котором приложение разбивается на небольшие независимые сервисы, специализирующиеся на определенных функциональных областях. Это позволяет достичь высокой гибкости, масштабируемости и легкости разработки и сопровождения системы.

Понять суть микросервиса проще всего на сравнении, или даже противопоставлении его крупному приложению – монолиту (Рисунок 2.1). Каждый отдельный сервис в такой архитектуре обладает следующими восьмью свойствами:

1) Он небольшой – микросервис отличается от монолитных приложений тем, что он имеет небольшой размер и ограниченные обязанности. Он фокусируется на решении конкретных задач в пределах своей функциональности.

2) Он независимый – микросервис является автономным и независимым компонентом, который может функционировать независимо от других сервисов. Он имеет свои собственные ресурсы, базу данных и логику обработки данных.

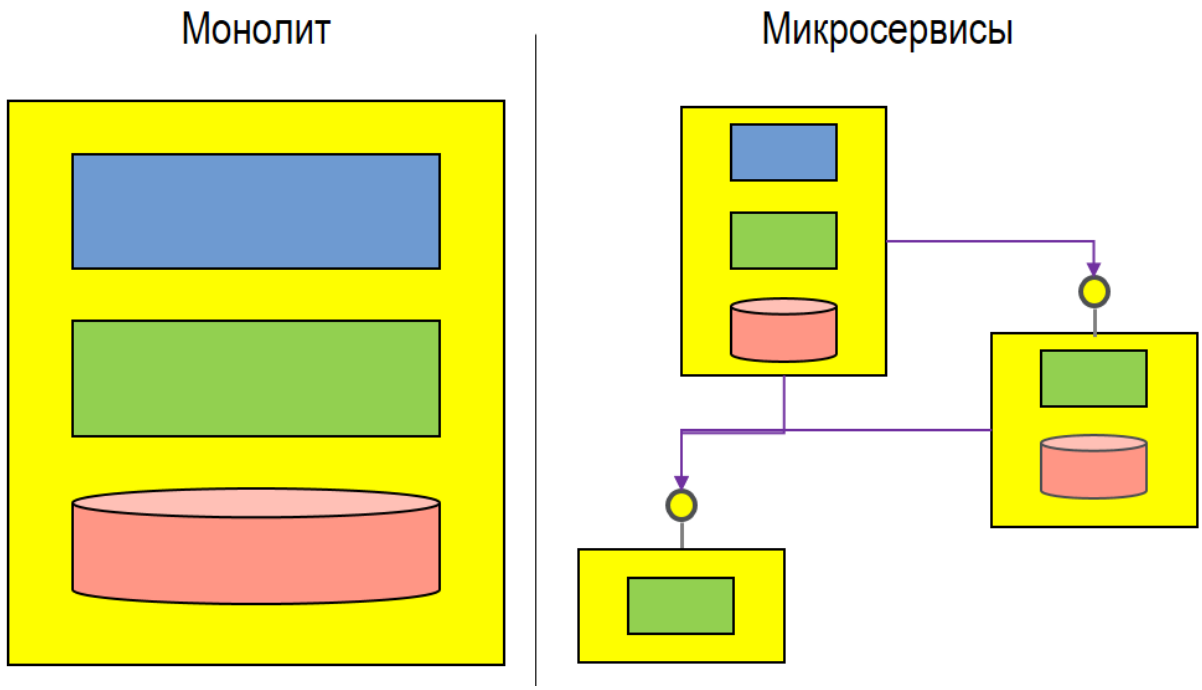

Рисунок 2.1 – Сравнение монолитной и микросервисной архитектур [2]

3) Он строится вокруг бизнес-потребности (или просто предметной потребности) и использует ограниченный контекст (Bounded Context). Bounded Context – это понятие в микросервисной архитектуре, которое описывает ограниченную область или границу в предметной области системы. В простых словах, это способ организации сложных систем, разбивая их на отдельные, логически связанные части, чтобы каждая из них отвечала за определенный участок функциональности. Каждый Bounded Context имеет свою локальную модель данных и границы, внутри которых существует ясное понимание того, как работает и взаимодействует каждый компонент. Это позволяет разрабатывать и масштабировать каждую часть независимо, обеспечивая гибкость и улучшенную поддержку изменений в развивающихся системах.

4) Он взаимодействует с другими микросервисами по сети на основе паттерна Smart endpoints and dumb pipes («умные эндпоинты и глупые каналы связи»): В микросервисной архитектуре взаимодействие между сервисами осуществляется по сети. При этом, подход «умные эндпоинты и глупые каналы связи» означает, что основная логика и функциональность находится внутри каждого микросервиса (умные эндпоинты), а каналы связи (глупые каналы) предоставляют простой способ передачи данных между сервисами.

5) Его распределенная суть обязывает использовать подход Design for failure («отказоустойчивое проектирование»): Микросервисы расположены на разных серверах и могут взаимодействовать друг с другом через сеть. Это означает, что система должна быть готова к возможным отказам сервисов, сетевым проблемам и другим ситуациям, которые могут возникнуть в распределенной среде. Подход «отказоустойчивое проектирование» подразумевает, что система разрабатывается таким образом, чтобы она могла продолжать работать даже при возникновении отказов в одном или нескольких сервисах.

6) Централизация ограничена сверху на минимуме – в микросервисной архитектуре стремятся минимизировать централизацию, особенно на уровне архитектуры и управления. Каждый сервис выполняет определенные функции и имеет свою собственную логику и хранение данных. Это позволяет улучшить гибкость и масштабируемость системы, так как изменения в одном сервисе не затрагивают остальные.

7) Процессы его разработки и поддержки требуют автоматизации – Разработка и поддержка микросервисных приложений включают в себя множество сервисов, которые работают вместе. Для облегчения и упрощения этого процесса требуется автоматизация. Это может быть автономная сборка, развертывание, мониторинг, тестирование и другие процессы, которые помогают обеспечить эффективное развитие и поддержку микросервисов.

8) Его развитие итерационное – В микросервисной архитектуре разработка и внедрение новых функций и изменений происходят итеративно. Это означает, что процесс разработки разбивается на небольшие итерации, в рамках которых выполняется планирование, разработка, тестирование и развертывание нового функционала. Такой подход позволяет быстрее реагировать на изменения и обратную связь от пользователей, обеспечивая постепенное развитие приложения.

Изучив только эти особенности микросервиса, можно провести ассоциативную параллель с классом в объектно-ориентированном программировании. Здесь мы видим принципы и соглашения схожие с основными принципами ООП, а также, вытекающими из них, принципами SOLID, только на один уровень абстракции выше. Особенно заметно влияние двух первых принципов SRP и SOP.

Single Responsibility Principle (SRP) или же Принцип единой ответственности – этот принцип гласит, что у каждого объекта есть своя ответственность и причина существования и эта ответственность у него только одна. Другими словами, через объект проходит только одна ось изменений.

Open-Closed Principle (OCP) или же Принцип открытости-закрытости – гласит, что любой модулю класс или функция должны быть открыты для расширения и закрыты для изменения.

а) Открыты для расширения. Это означает, что поведение модуля может быть расширено. То есть мы можем добавить модулю новое поведение в соответствии с изменившимися требованиями к приложению или для удовлетворения нужд новых приложений.

б) Закрыты для изменений. Исходный код такого модуля неприкасаем. Никто не вправе вносить в него изменения.

Принцип единой ответственности (SRP) аналогичен идее, что каждый микросервис должен иметь только одну четко определенную ответственность. Принцип открытости-закрытости (OCP) также перекликается с идеей микросервисной архитектуры. В контексте микросервисов, каждый сервис должен быть открыт для расширения, то есть, его функционал может быть легко расширен или изменен без влияния на другие сервисы. В то же время, каждый сервис должен быть закрыт для изменения, чтобы предотвратить нежелательное влияние на другие сервисы или систему в целом.

Благодаря соблюдению этих принципов в микросервисной архитектуре достигается высокая устойчивость, строгость и безопасность системы. Каждый сервис имеет четко определенную ответственность, что упрощает разработку и поддержку. Изменения в одном сервисе могут быть внесены без привлечения других сервисов, что улучшает масштабируемость и гибкость системы. Более того, такая структура позволяет лучше изолировать возможные ошибки и проблемы, благодаря чему отказ одного сервиса не приводит к сбою всей системы. Всё это способствует стабильности работы и безопасности микросервисной архитектуры.

Всё это и обусловило выбор данной архитектуры, ведь главные показатели, которые нужны системе, обеспечивающей уведомление пользователей о чрезвычайных ситуациях это: отказоустойчивость, масштабируемость, строгость, безопасность, скорость.

Так же при выборе не были закрыты глаза на минусы данного подхода к реализации информационных систем. Сложность взаимодействия, сложность тестирования, и цена разработки и поддержки – это то, чем необходимо пожертвовать, ради достижения безопасности.

2.2 Предметно-ориентированный подход

Domain-Driven Design (DDD) или же Предметно-ориентированное проектирование – это методология проектирования и разработки программного обеспечения, которая ставит предметную область бизнеса в центр внимания. На самом деле эта концепция рабочая для проектирования информационной системы любой области, где это необходимо, а не только для бизнеса. Основная идея DDD заключается в том, что разработчики и доменные эксперты должны активно взаимодействовать, чтобы понять предметные-требования и перенести их в модель предметной области, которая будет использоваться для реализации системы [3].

Принципы DDD помогают структурировать и организовать систему, чтобы она лучше соответствовала предметной области. Основные 5 принципов DDD включают в себя:

1) Базовые понятия предметной области (Building Blocks): это концепции, которые используются для представления и описания предметной области. Они включают в себя сущности (Entities), значения (Value Objects), агрегаты (Aggregates), репозитории (Repositories) и сервисы (Services), которые помогают моделировать и организовывать систему на основе предметной-логики [3].

2) Соответствие между моделью и предметной областью (Ubiquitous Language): разработчики и доменные эксперты должны использовать общий язык, который будет удобен и понятен для всех участников проекта. Общий язык помогает уточнить требования и согласовать понимание процессов той системы, которую необходимо цифровизировать [3].

3) Обособленные (изолированные) контексты (Bounded Contexts): этот принцип предлагает разбить систему на изолированные контексты, где каждый контекст имеет собственную модель, правила и ограничения. Это помогает управлять сложностью системы и делить ее на небольшие компоненты, которые легче понять и поддерживать [3].

4) Фокус на аспектах значимых для предметной области (Core Domain): принцип DDD настоятельно рекомендует обращать особое внимание на процессы и функционал, на которых сосредоточена основная ценность для предметной области. Функциональность, которая не является основной для системы, может быть вынесена в отдельные компоненты или модули [3].

5) Разработка на основе эволюции (Emergent Design): DDD призывает к гибкому и постепенному проектированию системы. У разработчиков должна быть возможность итеративно вносить изменения в модель предметной области и архитектуру системы на основе реальных требований и улучшений [3].

В общем случае получаем, что применение принципов DDD позволяет создавать более поддерживаемые, модульные и гибкие системы, которые лучше соответствуют реальным потребностям предметной области и обеспечивают успешное взаимодействие между разработчиками и бизнесом.

В частном случае, видим решение, идеально подходящее для декомпозиции предметной области, и к подготовке её к цифровизации на основе микросервисной архитектуры.

Предметно-ориентированный подход (Domain-Driven Design, DDD) и микросервисная архитектура (Microservices Architecture, MSA) дополняют друг друга и могут быть эффективно применены вместе. Оба подхода фокусируются на разделении системы на изолированные компоненты с уникальной ответственностью, что позволяет достичь лучшей модульности, масштабируемости и гибкости системы.

Одним из ключевых принципов как DDD, так и MSA является ограниченный контекст (Bounded Context):

а) Каждый ограниченный контекст определяет свою собственную модель предметной области и язык, позволяя разработчикам и бизнес-экспертам иметь ясное понимание того, как работает конкретная часть системы.

б) В контексте микросервисной архитектуры, каждый микросервис может быть реализацией одного или нескольких ограниченных контекстов. Это позволяет разработчикам разрабатывать и поддерживать каждый микросервис как отдельное приложение, с четко определенной областью ответственности и языком.

DDD и MSA также поддерживают децентрализованное управление данными. В DDD, каждый ограниченный контекст может иметь собственное хранилище данных, а MSA стремится к принципу независимых баз данных для каждого сервиса. Это позволяет разрабатывать и масштабировать каждый микросервис независимо от других, улучшая производительность системы и снижая возможность конфликтов данных.

В целом, предметно-ориентированный подход в проектировании, основанный на принципах DDD, идеально подходит для микросервисной архитектуры, так как оба подхода стремятся к изоляции функциональности и управлению сложностью через Ограниченные Контексты и независимую разработку и развертывание компонентов системы. Это позволяет эффективно управлять сложностью и изменениями, создавая более гибкие и модульные системы.

2.3 Язык программирования

В данном разделе представлен анализ факторов, определяющих выбор языка программирования для разработки приложения, предназначенного для предоставления пользовательских уведомлений о чрезвычайных ситуациях на основе микросервисной архитектуры. Критерии выбора включают эффективность, гибкость и безопасность. В результате проведённых

исследований и сравнительного анализа, был выбран язык программирования Java и сопутствующее семейство фреймворков Spring.

Java представляет собой одну из самых распространенных технологий программирования в области backend разработки и реализации сложных и масштабных приложений. В настоящее время Java находит широкое применение благодаря ряду преимуществ, семь из которых рассмотрим подробнее:

1) Кроссплатформенность – Java разработан с учетом платформенной независимости, что означает, что разработанные на нем приложения могут быть запущены на различных операционных системах без необходимости изменения кода. Такой подход обеспечивает высокую гибкость и возможность развертывания приложений на различных платформах [4].

2) Обратная совместимость – принцип, который Java не нарушала никогда. Многие считают Java слишком консервативным и медленно развивающимся языком, но те, кто разрабатывает на Java всегда уверенны, что перенос своих проектов на новые версии Java будет осуществлён легче и быстрее, чем на каком-либо языке. Это качество ставит Java на уровень выше по сравнению с C#, казалось бы, очень похожим по всем принципам на неё.

3) Безопасность – Одним из привлекательных аспектов Java является наличие встроенных механизмов безопасности. Запуск программ на Java осуществляется в виртуальной машине Java (JVM), которая обеспечивает изоляцию исполняемого кода от операционной системы и снижает уязвимость к вредоносным атакам [4].

4) Отказоустойчивость – Java обладает механизмами управления памятью и обработки исключений, сделавшими его надежным и стойким к сбоям. Высокоуровневые средства управления памятью, включая сборку мусора, способствуют предотвращению утечек памяти, а система обработки исключений облегчает идентификацию и обработку ошибок. Всё это облегчает и ускоряет разработку в отличие от тех же C и C++ [4].

5) Богатая экосистема – Java располагает обширной и динамичной экосистемой, включающей в себя множество библиотек, фреймворков и инструментов разработки. Этот фактор позволяет разработчикам использовать готовые решения для типовых задач и значительно сократить время и ресурсы, затрачиваемые на разработку [4].

6) Масштабируемость – Java обладает возможностями разработки масштабируемых приложений. Он позволяет эффективно обрабатывать большое количество данных и обеспечивать высокую производительность благодаря использованию многопоточности, распределенных вычислений и других техник [4].

7) Строгость – Java строго типизированный язык, а также строго структурированный, соответствует строго объектно-ориентированному подходу. Строгость является одним из важнейших качеств языка Java, что позволяет выявлять ошибки на ранних стадиях, а также способствует читаемости кода. Это положительно выделяет её на фоне таких языков как Python и PHP. Хотя второй в своих новых версиях уже начал стримиться к строгой типизации.

Хорошим аналогом мог быть быстрый современный функциональный язык программирования Go lang, но по причинам его «незрелости», за которой не прерывно следует меньшая степень интеграции с сторонними технологиями, менее богатая экосистема и менее развитое сообщество, а также по причине того, что код в Go соответствует функциональному стилю, ломающему всю строгость и концепцию, выстроенную ранее, выбор пал на язык Java.

Spring является одним из наиболее популярных фреймворков разработки приложений на Java. Он предоставляет обширные возможности и инструменты для создания и организации микросервисных архитектур. Преимущества Spring по сравнению с другими фреймворками охарактеризованы в следующих 5 пунктах:

1) Легкая интеграция – Spring обладает возможностью интеграции с различными технологиями и фреймворками. Он поддерживает использование различных модулей и библиотек, таких как Spring Data, Spring Security и Spring MVC для реализации разнообразных функциональных возможностей в приложениях [5].

2) Inversion of Control (IoC) в Spring – это принцип разработки, который позволяет управление созданием и жизненным циклом объектов быть вынесенным из кода самого приложения и делегированным на платформу Spring. Суть IoC заключается в том, что контроль над созданием и взаимодействием между объектами осуществляется фреймворком, а не самим приложением. В Spring Framework IoC реализуется с использованием контейнера внедрения зависимостей (Dependency Injection Container). Контейнер внедрения зависимостей отвечает за создание и связывание объектов в приложении. Он автоматически создает и управляет экземплярами объектов, а также устанавливает им необходимые зависимости. Благодаря этому, разработчики могут создавать, тестировать и заменять компоненты приложения без изменения других частей системы [5].

3) Аспектно-ориентированное программирование (АОП) – В Spring Framework AOP реализуется с использованием прокси-объектов и аспектов. Прокси – это объект, который используется для перехвата методов другого объекта, тогда как аспект представляет собой перехватчик, который выполняет дополнительные действия перед, после или вместо выполнения метода. Это позволяет вынести "сквозную" функциональность, например, журналирование, управление транзакциями и безопасность, в отдельные модули. Такой подход упрощает процесс разработки и обеспечивает масштабируемость разрабатываемым системам [5].

4) Тестирование – Spring обладает встроенными средствами для тестирования приложений. Он предоставляет мок-объекты, инструменты для модульного и интеграционного тестирования, что способствует созданию надежного и стабильного кода [5].

5) Поддержка сообщества – у Spring активное сообщество разработчиков, которое предоставляет обширную документацию, готовые решения и поддержку. Это позволяет решать возникающие проблемы более эффективно, а разработчикам обратиться за помощью в случае необходимости [5].

В результате анализа, был сделан обоснованный выбор Java и Spring для разработки микросервисных архитектур. Java обеспечивает безопасность, отказоустойчивость и масштабируемость, а Spring предоставляет удобные инструменты и функциональность для разработки и управления микросервисами.

2.4 Базы данных

Базы данных являются одними из основных технологий в серверной разработке. Если в общем смысле можно описать программирование как алгоритмы над данными, и при этом мы будем представлять данные, как некоторые значения, лежащие в оперативной памяти, а алгоритмы, как вычислительную логику, производимую над этими данными, то теперь на уровне энтерпрайз разработки за алгоритмы будем принимать предметную или бизнес-логику (Domain), а за часть, отвечающую за данные, мы будем принимать именно слой базы данных.

Поэтому очень важно правильно подобрать технологии, которые будут хорошо выполнять необходимые нам задачи, а также будут совместимы с нашими другими технологиями. Перед выбором существующих баз данных, необходимо понять, каким образом вообще Java будет осуществлять работу с ними. В контексте взаимодействия Java-приложений с базами данных, существует несколько ключевых технологий и понятий, которые стоит рассмотреть.

Начнем с JDBC (Java Database Connectivity) – интерфейса прикладного программирования для языка Java, который определяет, как клиент может

получить доступ к базе данных. JDBC является технологией доступа к данным на основе Java и предоставляет методы для запроса и обновления данных в реляционных базах данных. Раньше, для взаимодействия с базами данных Java-разработчики часто использовали JDBC напрямую. Он требует непосредственного написания SQL-запросов и работы с ними через Java-код.

Однако с развитием технологий были созданы ORM-фреймворки (Object-Relational Mapping), которые предоставляют более удобные и абстрактные способы взаимодействия с базами данных. Одним из самых популярных ORM-решений для Java является Hibernate. Hibernate предоставляет возможности автоматического отображения Java-классов и их свойств на таблицы и столбцы базы данных. Он генерирует SQL-запросы и управляет процессом сохранения, обновления и извлечения данных, что значительно упрощает разработку и уменьшает необходимость вручную писать SQL-код. Hibernate также является реализацией спецификации JPA (Java Persistence API), которая описывает систему управления сохранением Java-объектов в реляционных базах данных.

С использованием Hibernate и JPA, разработчики могут работать с объектами в своем коде, а фреймворк автоматически преобразует эти объекты в SQL-запросы и обеспечивает их выполнение в базе данных. Такой подход существенно упрощает разработку и делает код более портируемым между различными базами данных.

В современной разработке Java-приложений также широко используется Spring Framework, который предоставляет множество полезных инструментов и модулей для разработки приложений. Один из таких модулей - Spring Data, который упрощает работу с базами данных в контексте Spring [5].

Spring Data позволяет организовать доступ к данным через репозитории, где достаточно объявить интерфейс с некоторыми методами, а фреймворк автоматически генерирует реализацию этих методов, выполняя необходимые SQL-запросы или вызывая соответствующие методы ORM-библиотеки. Это

значительно сокращает объем кода, облегчает разработку и повышает читаемость [5].

Именно поэтому по мимо того, что базы данных, которые будут описаны далее, соответствуют предметным требованиям проекта, они так же отлично сочетаются с работой совместно с Hibernate и Spring Data.

2.4.1 PosgreSQL

PostgreSQL является одним из ведущих реляционных баз данных и обладает рядом преимуществ и уникальных особенностей, которые делают его предпочтительным выбором для множества разработчиков и организаций.

Во-первых, PostgreSQL обладает высокой степенью расширяемости, что позволяет разработчикам настраивать базу данных в соответствии с их потребностями. Возможность использования хранимых процедур, триггеров, пользовательских типов данных и операторов расширяет функциональность PostgreSQL и позволяет создавать собственные расширения.

Во-вторых, PostgreSQL строго соответствует стандартам SQL, включая последние версии стандарта, такие как SQL:2008. Это обеспечивает переносимость приложений и упрощает разработку и поддержку кода, поскольку разработчики могут полагаться на единый набор стандартных SQL-запросов в различных базах данных.

В-третьих, PostgreSQL предоставляет богатый набор расширений для работы с географическими данными, включая популярную платформу PostGIS. Это позволяет хранить, обрабатывать и выполнять сложные геопространственные запросы, что особенно важно для приложений, связанных с картографией, геолокацией и аналитикой данных [6].

Дополнительно, PostgreSQL обладает продвинутой системой репликации данных, предлагая методы потоковой репликации и логической репликации. Это позволяет создавать отказоустойчивые системы с высокой доступностью данных.

Основная сила PostgreSQL также заключается в богатом наборе типов данных. Это включает различные специализированные типы для работы с JSON, XML, массивами, полнотекстовым поиском и многими другими, что упрощает моделирование данных и обработку различных типов информации [6].

В конечном итоге, PostgreSQL предлагает различные алгоритмы индексации для оптимизации производительности запросов и повышения быстродействия базы данных при поиске и фильтрации данных.

Конкурентом данной базы является Oracle MySQL, который имеет не меньшую популярность и является очень серьёзным инструментом для работы с данными.

Первое существенное различие между ними заключается в их функциональности. PostgreSQL предлагает более широкий набор возможностей и функций, включая более полное соответствие стандартам SQL, поддержку географических данных и расширяемость. С другой стороны, MySQL позиционируется как простая в использовании и быстрая база данных, особенно подходящая для простых веб-приложений, и имеет лучшую производительность на некоторых типах нагрузки, таких как простые CRUD-операции.

Также следует учесть экосистему и инструментальную поддержку. Обе базы данных имеют разнообразные экосистемы и инструменты, однако PostgreSQL, обладает более разнообразным сообществом пользователей и разработчиков, а также большим количеством готовых решений для различных задач.

Поскольку многие приложения на языке Java требуют эффективной и надежной работы с базами данных, PostgreSQL представляет собой идеальный вариант для использования в связке с Java и Hibernate.

PostgreSQL предоставляет хорошо развитый и популярный драйвер JDBC, обеспечивающий удобное взаимодействие с базой данных на языке

Java. Это создает удобную и мощную среду для разработки Java-ориентированных приложений.

Hibernate, также тесно интегрирован с PostgreSQL. Он обеспечивает удобное отображение объектов Java на таблицы в базе данных PostgreSQL, а также автоматическую генерацию SQL-запросов. Такая комбинация облегчает разработку приложений и упрощает работу с данными.

2.4.2 MongoDB

MongoDB является одним из лидеров среди документных баз данных по ряду причин. Во-первых, ее гибкая схема данных позволяет каждому документу в коллекции иметь собственную структуру, что облегчает изменение схемы данных без обновления всей базы данных. Во-вторых, MongoDB обладает уникальными возможностями горизонтального масштабирования, позволяющими распределить данные по нескольким серверам и обеспечить высокую производительность даже при работе с большими объемами данных. Это особенно важно для растущих проектов. MongoDB также поддерживает репликацию данных для обеспечения отказоустойчивости. Кроме того, MongoDB обеспечивает высокую производительность благодаря хранению данных в близкой к памяти форме и использованию индексов. Она также предоставляет возможность выполнения распределенных запросов, что позволяет эффективно распределять нагрузку на серверы. Наконец, MongoDB предлагает богатый набор функций, включая индексы, запросы на основе текста, агрегационные запросы, отслеживание изменений и географические запросы, делая ее удобной для различных типов приложений [7].

При сравнении MongoDB и Cassandra, обе базы данных являются популярными и масштабируемыми решениями, но они имеют несколько различий. Во-первых, модель данных в MongoDB основана на документах, где каждый документ представляет собой JSON-подобный объект (Рисунок 2.2).

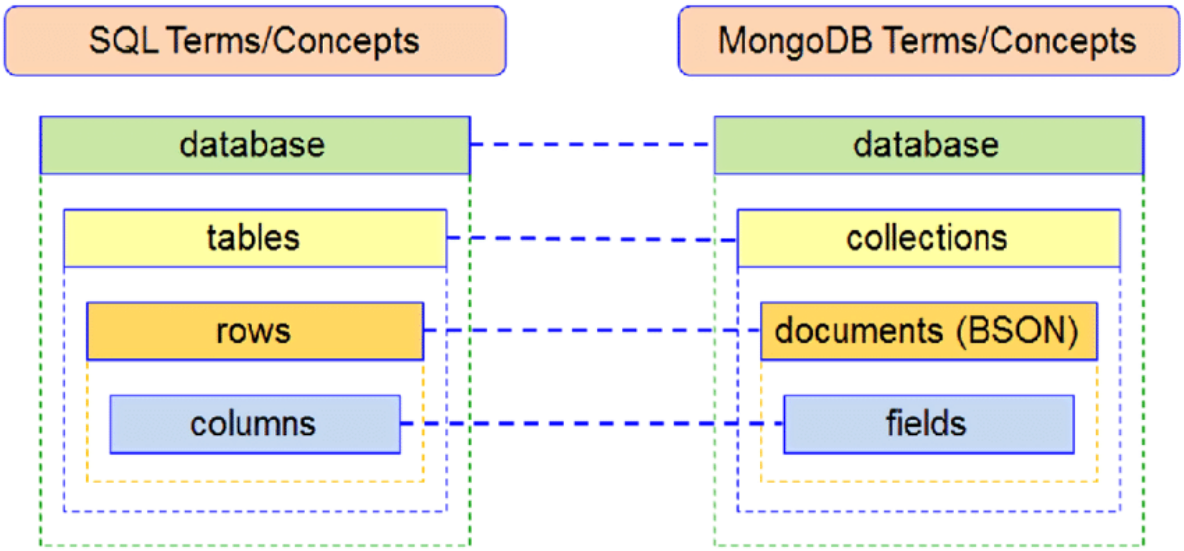

Рисунок 2.2 – Сравнение MongoDB с SQL базами [7]

В то время как Cassandra использует модель данных на основе колонок, где данные хранятся в виде широких рядов с неопределенным числом столбцов. Во-вторых, MongoDB обеспечивает согласованность данных по умолчанию (strong consistency), в то время как Cassandra обеспечивает связь между доступностью и разделением (tuneable consistency). Это означает, что Cassandra может обеспечить высокую доступность данных в условиях распределенных систем, но может потерять согласованность в некоторых случаях. Наконец, язык запросов MongoDB более гибок и похож на язык запросов для реляционных баз данных, в то время как CQL (Cassandra Query Language) было специально разработано для работы с моделью данных Cassandra [7].

Для использования MongoDB с Java и Hibernate существует несколько причин, почему эта комбинация может быть идеальной. Во-первых, MongoDB предоставляет официальный драйвер для Java, который обеспечивает нативную интеграцию между Java-приложениями и базой данных MongoDB. Этот драйвер предлагает различные функции, такие как поддержка аннотаций, маппинг объектов на документы MongoDB и простота в использовании. Во-вторых, Hibernate также предоставляет модуль под названием Hibernate OGM (Object/Grid Mapper), который позволяет работать с документными базами

данных, включая MongoDB. Это позволяет разработчикам использовать привычные средства Hibernate для работы с MongoDB.

2.4.3 Распределение задач

Так как разрабатываемая архитектура основана на микросервисах, базы данных которых обособленны друг от друга, мы без лишних сложностей можем использовать одновременно различные технологии для работы с базами данных.

Для чего понадобилось использование двух различных технологий, таких как PosgreSQL и MongoDB.

Реляционная база, такая как PosgreSQL, необходимо в разрабатываемом проекте по причине своей надёжности. В реляционных базах данных существуют принципы, обеспечивающие безопасность на уровне транзакций (ACID), эти принципы могут ассоциативно напоминать принципы SOLID для объектно-ориентированной разработки.

ACID – это аббревиатура, которая описывает набор принципов и свойств, обычно применяемых в контексте транзакций в базах данных [7]. Каждая из четырёх буква в слове ACID представляет отдельный принцип:

1) Атомарность (Atomicity) – Принцип атомарности гарантирует, что транзакция будет выполнена либо полностью, либо не выполнена совсем [7]. Это означает, что если в рамках транзакции возникает ошибка или проблема, то все изменения, сделанные до этой точки в транзакции, откатываются, и не происходит никаких изменений в базе данных. Таким образом, принцип атомарности обеспечивает целостность данных.

2) Согласованность (Consistency) – Принцип согласованности гарантирует, что транзакция приводит к изменению базы данных из одного согласованного состояния в другое согласованное состояние. Это означает, что транзакция должна соблюдать все ограничения целостности данных,

установленные в базе данных. Если транзакция не соблюдает эти ограничения, то она откатывается, и база данных остается в прежнем состоянии.

3) Изолированность (Isolation) – Принцип изолированности гарантирует, что каждая транзакция выполняется независимо от других транзакций. Изменения, сделанные в рамках одной транзакции, не будут видимы другим транзакциям до тех пор, пока первая транзакция не будет завершена. Это предотвращает нежелательное взаимодействие между транзакциями и обеспечивает консистентность данных.

4) Долговечность (Durability) – Принцип долговечности гарантирует, что результаты выполненных транзакций сохраняются даже в случае сбоев или перезапуска системы. Это достигается с помощью сохранения изменений в постоянное хранилище, такое как жесткий диск или несколько реплик базы данных. Таким образом, даже после сбоя данные остаются надежными и доступными.

ACID-принципы важны для обеспечения надежности и целостности данных в базах данных, особенно в контексте транзакций [7]. Их соблюдение гарантирует, что операции записи и чтения данных выполняются надежно и предсказуемо, даже в случае сбоев или параллельного выполнения множества транзакций.

Поэтому эта база будет использована как хранилище пользовательской информации, их статусов, паролей, логинов, и другой важной информации, такой как статус отправки уведомления.

MongoDB основанная на документах представляющих собой JSON-подобный объект удобна в тех сервисах, которые будут содержать шаблоны уведомлений создаваемых пользователями и сообществами.

2.5 Работа с кэшем

В современных микросервисных архитектурах широко применяется кэширование данных для повышения производительности, снижения

задержек и облегчения нагрузки на базы данных и другие службы. Выбор подходящей технологии кеширования играет важную роль в обеспечении эффективной работы микросервисов. В данной главе мы рассмотрим несколько популярных технологий кэширования, их плюсы и минусы.

In-Memory Data Grids (IMDG) – IMDG-системы предлагают хранение данных в оперативной памяти распределенных систем, что обеспечивает высокую производительность и быстрый доступ к данным. Примерами IMDG-систем являются Apache Ignite, Hazelcast, Oracle Coherence и GridGain. Они предоставляют удобные API для работы с кэшем и масштабируются горизонтально для управления большими объемами данных. Однако, в некоторых случаях, использование IMDG может потребовать дополнительной конфигурации и настройки. Обобщая, IMDG-системы обладают высокой производительностью и легко масштабируются, но требуют дополнительной конфигурации [8].

Distributed Caches – Распределенные кэши - предназначены для хранения данных на нескольких узлах в распределенной сети. Они имеют возможность обрабатывать большие объемы данных и обеспечивают отказоустойчивость. Некоторыми из популярных распределенных кэш-систем являются Redis, Memcached и Apache Cassandra. Redis отличается высокой скоростью и поддержкой широкого спектра функций (например, структуры данных, пайплайнинг, публикация/подписка), что делает его привлекательным вариантом для кэширования данных в микросервисных архитектурах. Резюмируя, распределенные кэши обладают высокой производительностью, поддерживают отказоустойчивость. Redis является мощным вариантом, предлагая широкий набор функций [8].

Библиотеки и фреймворки кэширования - существуют различные библиотеки и фреймворки, которые предоставляют специализированные средства для работы с кэшем. Примерами таких фреймворков в экосистеме Java являются Spring Cache, Ehcache и Caffeine. Они предлагают удобные абстракции для работы с кэшем, интеграцию с другими компонентами

приложений и легко настраиваются. Однако, они могут быть менее гибкими и масштабируемыми по сравнению с IMDG или распределенными кэш-системами [8].

БД с поддержкой кэширования - некоторые базы данных (например, Oracle Database или Microsoft SQL Server) имеют встроенные функции кэширования для оптимизации доступа к данным. Они позволяют предварительно загрузить данные в кэш или кэшировать часто запрашиваемые данные, что повышает производительность системы. Однако, эти функции могут быть лимитированы, и базой данных может быть сложно масштабироваться горизонтально для обработки растущей нагрузки [8].

HTTP-кэширование - для веб-приложений, использующих протокол HTTP, можно использовать механизмы кэширования, предоставляемые браузерами и прокси-серверами. Это позволяет сохранять результирующие данные запросов на клиентской стороне или промежуточных серверах для повторного использования. HTTP-кэширование прост в использовании и может снизить нагрузку на серверы, но оно имеет ограниченные возможности настройки и контроля над данными в кэше [8].

На основании представленной выше информации, напрашивается очевидный вывод: Redis является привлекательной технологией кэширования данных в микросервисной архитектуре, особенно для Java и Spring разработчиков. Redis обладает высокой производительностью, поддерживает широкий спектр функций и имеет хорошую совместимость с Java и Spring.

Redis (Remote Dictionary Service) – это опенсорсный сервер баз данных типа ключ-значение. Redis – это база данных, размещаемая в памяти, которая используется, в основном, в роли кеша, находящегося перед другой, «настоящей» базой данных, вроде MySQL или PostgreSQL (Рисунок 2.3) [9].

Redis представляет собой распределенную кэш-систему, которая может хранить данные в оперативной памяти, обеспечивая высокую скорость доступа. Он поддерживает различные структуры данных, такие как строки, списки, хэши, множества и сортированные множества, что делает его гибким

инструментом для кэширования различных типов данных. Redis также предлагает дополнительные функции, такие как публикация/подписка, транзакции и Lua-скрипты, что дает возможность реализации сложной логики кэширования.

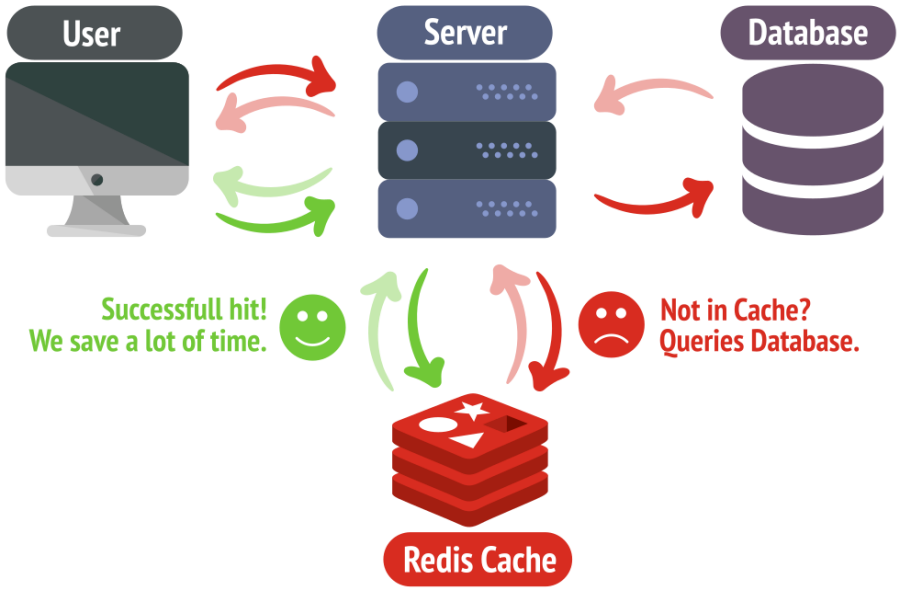

Рисунок 2.3 – Принцип работы с Redis кешированием [9]

Redis имеет открытый и активно развивающийся экосистему, которая предлагает различные клиентские библиотеки и инструменты для интеграции с Java и Spring. Например, существует официальная библиотека Jedis для взаимодействия с Redis на языке Java, а также Spring Data Redis, который предоставляет удобные абстракции и интеграцию с Spring-приложениями.

2.6 Брокеры данных

В развитии современных информационных систем становится все более актуальной задача обеспечения эффективного обмена данными между компонентами системы. Одним из ключевых инструментов, позволяющих решить эту задачу, являются брокеры данных. Брокеры данных – это программные компоненты, предназначенные для приема, хранения и передачи

сообщений между различными приложениями и сервисами. Они обеспечивают надежность, масштабируемость и гарантированную доставку данных, а также позволяют реализовать асинхронную обработку сообщений, что особенно важно в условиях высокой нагрузки и распределенной архитектуры системы.

Одним из известных и широко применяемых брокеров данных является Apache Kafka. Kafka предоставляет распределенную, высокопроизводительную и устойчивую платформу для потоковых данных. Он может обрабатывать и хранить огромные объемы данных в режиме реального времени, обеспечивая высокую производительность и низкую задержку при передаче сообщений. Особенностью Kafka является его устройство, основанное на паттерне Pub-sub (публикатор-подписчики), где производители (публикаторы) отправляют сообщения в определенные топики, а потребители (подписчики) получают сообщения из этих топиков. Kafka также предоставляет механизмы для репликации данных, обеспечивая надежность и отказоустойчивость системы [10].

Еще одним популярным брокером данных является RabbitMQ. RabbitMQ реализует протокол AMQP (Advanced Message Queuing Protocol) и обеспечивает надежную доставку сообщений между приложениями. Он основан на принципе очередей сообщений, где производители отправляют сообщения в очередь, а потребители извлекают их из очереди для обработки. RabbitMQ обладает гибкой системой маршрутизации сообщений и поддерживает различные схемы обмена сообщениями, что позволяет эффективно управлять потоком данных в системе [11].

Одно из отличий связано с основными паттернами, используемыми в брокерах: Kafka основан на концепции «Глупый брокер, умный потребитель», в то время как RabbitMQ использует модель «Умный брокер, глупый потребитель».

Принцип «Умный брокер, глупый потребитель» по отношению к RabbitMQ означает, что брокер берёт на себя много дополнительных действий.

Например, следит за прочитанными сообщениями и удаляет их из очереди. Или сам организует процесс распределения сообщений между подписчиками (Рисунок 2.4) [12].

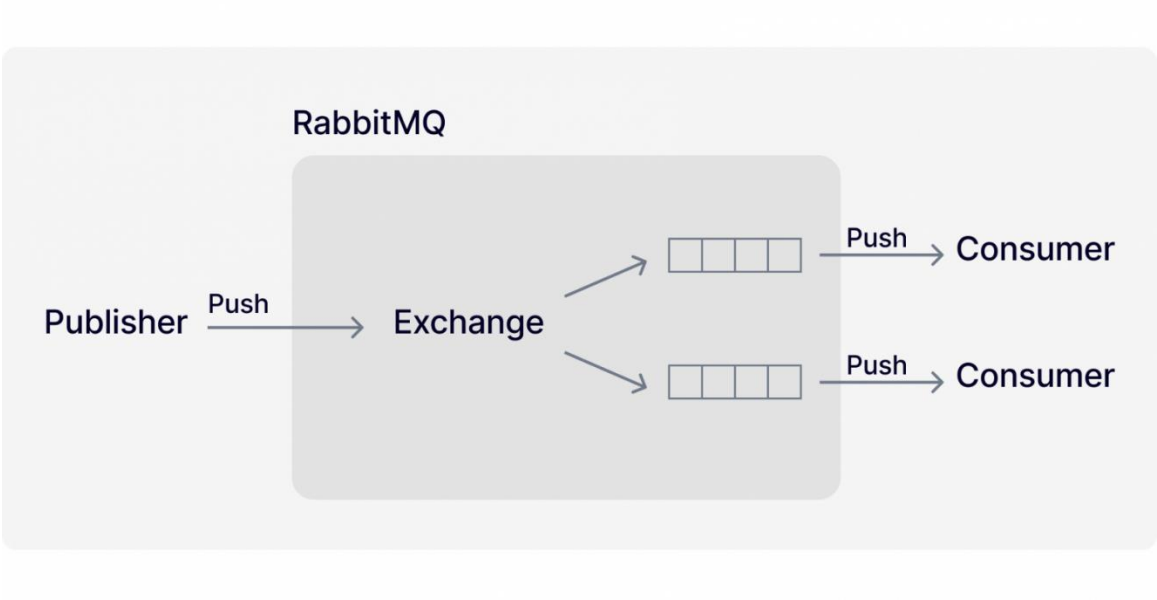

Рисунок 2.4 – Принцип работы RabbitMQ [12]

Для Kafka принцип «Глупый брокер, умный потребитель» означает, что, в отличие от RabbitMQ, он не занимается контролем и распределением сообщений. Потребители сами опрашивают брокер и решают, какие сообщения им читать, брокер только хранит данные (Рисунок 2.5). Приложения-подписчики (Consumer) читают, вытягивают (pull) сообщения из заданного топика. Для каждого подписчика Kafka запоминает указатель на последнее прочитанное им сообщение (offset). Если приложение падает, то восстановившись может продолжать чтение с прежнего места или перемотать (rewind) offset в прошлое и прочитать данные повторно. Количество партиций в топике зависит от количества его конкурирующих подписчиков. Одно приложение не может читать данные из одной партиции в несколько потоков. Параллелизм достигается за счёт увеличения количества партиций, для каждого потока своя [12].

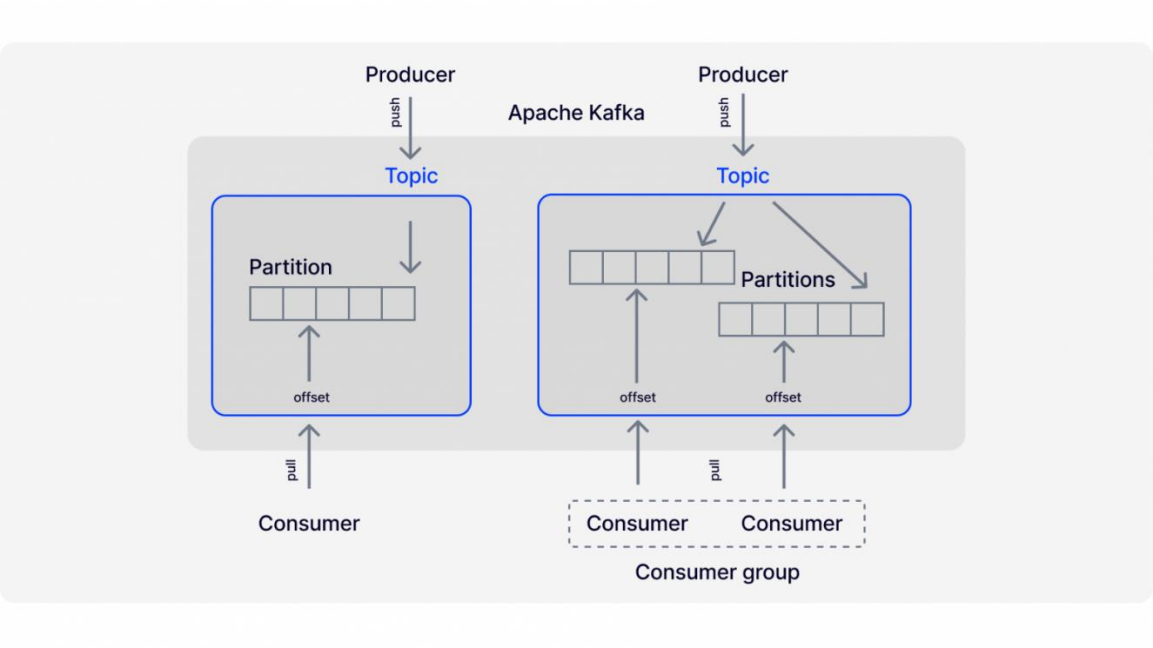

Рисунок 2.5 – Принцип работы Kafka [12]

В контексте проектирования системы оповещения пользователей о чрезвычайных ситуациях, выбор Kafka в качестве брокера данных является обоснованным решением. Его преимущества, такие как высокая производительность, низкая задержка и гибкость модели публикации-подписки, позволяют эффективно обмениваться данными между различными компонентами системы. Кроме того, Kafka обладает хорошей экосистемой инструментов особенно хорошо интегрируется с сервисами на Java, что делает его привлекательным выбором для разрабатываемой микросервисной архитектуры.

2.7 Связь между сервисами

Такое приложение, как платформа для обеспечения удобной нотификации о чрезвычайных ситуациях, где масштабируемость и гибкость становятся ключевыми требованиями к системе, эффективная связь между сервисами играет важную роль в его архитектуре. В этом разделе рассмотрены различные аспекты связи между сервисами, а также подобраны подходящие технологии для проектирования такой системы.

Связь между сервисами в микросервисной архитектуре позволяет сервисам взаимодействовать друг с другом и обмениваться информацией, достигая таким образом целостности и согласованности данных системы. Кроме того, правильно спроектированная связь позволяет добиться высокой отказоустойчивости и горизонтальной масштабируемости.

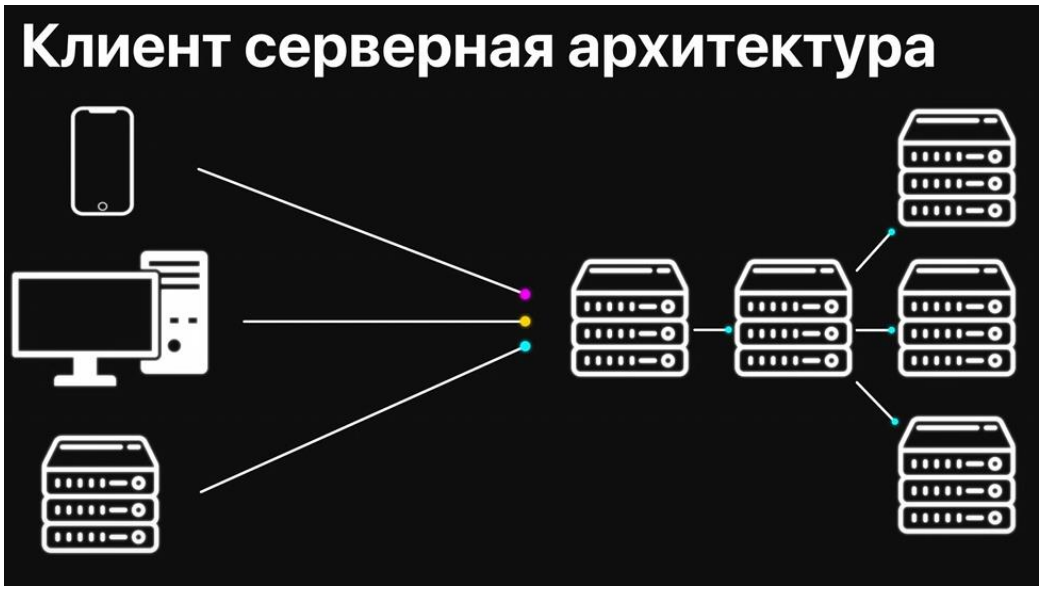

Рисунок 2.6 – Клиент-сервер в микросервисной архитектуре [13]

В микросервисной архитектуре распространены различные виды связи типа клиент-сервер (Рисунок 2.6), такие как синхронные и асинхронные вызовы. В синхронном вызове клиент отправляет запрос на сервер и ожидает получения ответа. Этот подход прост в использовании и понимании, поэтому часто применяется в микросервисных системах. В асинхронном вызове клиент отправляет запрос на сервер и продолжает свою работу, не ожидая ответа. Такой подход позволяет повысить отзывчивость системы, но требует дополнительной логики для обработки асинхронных сообщений.

Протоколы передачи информации играют важную роль в связи между сервисами. Рассмотрим 4 типа протоколов актуальных на сегодняшний день:

1) HTTP (Hypertext Transfer Protocol) – это протокол, который используется для передачи гипертекстовых документов в Интернете. Он основан на клиент-серверной модели и поддерживает различные методы

запроса, такие как GET, POST, PUT и DELETE. HTTP легко масштабируется, обеспечивает прозрачность и позволяет использовать различные форматы данных для обмена информацией [13].

2) SOAP (Simple Object Access Protocol) – это протокол, который позволяет обмениваться структурированной информацией между компьютерными системами. SOAP использует XML для представления данных и часто используется в распределенных системах, где требуется высокая надежность и безопасность [13].

3) gRPC (Google Remote Procedure Call) – это протокол, разработанный Google, который позволяет клиентским и серверным приложениям вызывать функции друг друга через сеть. gRPC основан на протоколе HTTP/2 и использует бинарный формат передачи данных, что обеспечивает высокую производительность и эффективность [13].

4) Thrift – это протокол, разработанный компанией Facebook, который позволяет сериализовывать структурированные данные и обмениваться ими между различными языками программирования. Thrift поддерживает различные типы данных, компактный формат передачи и генерацию кода для различных языков программирования [13].

Для разработки микросервисной архитектуры в контексте системы оповещения пользователей о чрезвычайных ситуациях на Java и с использованием Spring фреймворка, подходящим выбором будет протокол HTTP. HTTP обладает широкой поддержкой в Java и Spring, предоставляет простой и легковесный механизм передачи данных, а также отлично подходит для взаимодействия между микросервисами.

Конкретно в рамках микросервисного приложения будем использовать архитектурный стиль REST (Rest API), основанный на HTTP протоколе. Rest API позволяет организовать связь между сервисами через унифицированный набор методов, таких как GET, POST, PUT/PATCH и DELETE, что облегчает разработку, понимание и тестирование системы. Он позволяет передавать данные в формате JSON, XML и др., что упрощает разбор и обработку

информации. Далее подробнее о стиле и его правилах, обеспечивающих эффективность разрабатываемых API.

REST (Rest API) – архитектурный стиль, то есть набор правил, который описывает как наиболее эффективно использовать протокол передачи данных HTTP, и как правильно настраивать свои API (Application Programming Interface) – интерфейсы прокладного программирования, чтобы сервисами было удобно пользоваться, чтобы они выдерживали большие нагрузки и т.д.

Архитектурный стиль REST включает в себя следующие 10 концепций или же правил использования протокола HTTP:

1) Модель взаимодействия клиент-сервер.

2) Система может быть многоуровневой, то есть клиент отправив запрос на какой-то сервис, получает ответ, но при этом сколько ещё уровней было задействовано внутри для ответа он знать не должен (микросервисная архитектура).

3) Сервер не должен обладать никаким состоянием (Stateless), то есть сервер не обязан хранить никакого промежуточного состояния, и один и тот же клиент, при каждом обращении, для сервера как совершенно новый клиент.

4) Единообразный унифицированный интерфейс – правила для создания эндпоинтов, которые будут логичны и понятны всем.

5) Запрос должен содержать всю необходимую информацию для его выполнения.

6) Для каждого запроса необходимо использовать семантически правильный метод (Получить – GET, Отправить – POST), хотя чисто технически мы сможем использовать любой из методов для любой из операций.

7) Кэширование – причём кэширование может быть осуществлено, как и средствами HTTP, так и сторонними технологиями (GET и POST – кэшируются, PUT и DELETE – нет).

8) Формат обмена данными – в REST архитектуре это практически любой формат, но чаще всего используется JSON и так же иногда XML.

9) Версионирование – удобная возможность за счёт различных версий эндпоинтов вносить не обратно-совместимые правки.

10) Документирование – обязательный этап работы над API. Когда мы в удобном формате предоставляем всю необходимую информацию, необходимую для работы с API.

Таким образом, выбор протокола HTTP и архитектуры REST для связи между сервисами в проектируемом приложении на Java и Spring обусловлен их простотой использования, широкой поддержкой, гибкостью и соответствием требованиям MSA. Это позволит эффективно проектировать и разрабатывать систему.

2.8 Браузерный клиент

Современные браузерные приложения представляют собой мощные инструменты, которые оперируют с использованием различных языков и технологий. В основе функционирования браузерных приложений лежат браузерные интерпретаторы – программы, которые выполняют код и преобразуют его в последовательность действий, понятную для браузера.

Сегодня наиболее распространенными видами браузерных интерпретаторов являются интерпретаторы языка JavaScript и нового бинарного формата WebAssembly. JavaScript – это широко используемый язык программирования, который обрабатывается браузером и позволяет создавать динамические и интерактивные веб-страницы. Он является основным языком для разработки большинства сайтов и приложений. WebAssembly – это бинарный формат исполняемого кода, который работает в браузере и позволяет выполнять высокопроизводительные вычисления на различных языках программирования, таких как C++, C#, Rust, и других.

Для разработки современных больших веб-приложений разработчики часто прибегают к использованию фреймворков для JavaScript. Фреймворки представляют собой набор инструментов, библиотек и стандартов, которые

упрощают разработку, обеспечивают повторное использование кода и повышают производительность разработчика.

React, Angular и Vue являются тремя известными и широко используемыми фреймворками для JavaScript. Каждый из них имеет свои особенности и преимущества, которые делают их привлекательными для разработки разнообразных веб-приложений.

React – это библиотека для разработки пользовательского интерфейса, разработанная командой Facebook. Она отличается простотой, гибкостью и эффективностью, позволяя разработчикам создавать компоненты, которые могут быть легко переиспользованы. React использует виртуальную DOM (Document Object Model), что улучшает производительность и быстродействие приложения. Кроме того, огромное сообщество разработчиков поддерживает React, что обеспечивает доступ к множеству документации, инструментов и готовых решений, способствующих ускорению разработки [14].

Angular – это полноценный фреймворк, разработанный командой Google. Он предоставляет широкий набор инструментов и функциональных возможностей, включая удобную систему управления состоянием приложения и привязку данных. Angular основывается на компонентной архитектуре и предлагает разработчикам строгие правила и стандарты для создания надежных и масштабируемых приложений. Однако, для работы с Angular может потребоваться больше времени и усилий для овладения его особенностями и настройки [14].

Vue – это прогрессивный фреймворк с открытым исходным кодом, который обладает простым и интуитивно понятным API. Vue предлагает легкую и прагматичную альтернативу для разработки веб-приложений. Он отличается гибкостью и простотой в использовании, что позволяет быстро создавать привлекательные интерфейсы и масштабировать приложения. Vue также имеет активное сообщество разработчиков и богатый набор плагинов и компонентов, что делает его привлекательным выбором для проектирования системы оповещения пользователей о чрезвычайных ситуациях [14].

Исходя из анализа современных фреймворков для JavaScript, React представляет собой оптимальный выбор для проектирования системы оповещения пользователей о чрезвычайных ситуациях. Его простота, гибкость и эффективность позволяют быстро создавать компоненты, обеспечивать высокую производительность и ускорять разработку. Кроме того, широкая поддержка и активное сообщество разработчиков делают React надежным и доступным инструментом.

2.9 Мобильное приложение

В настоящее время мобильные приложения играют важную роль в обмене информацией и доставке оповещений пользователям. Это вызывает необходимость в выборе оптимального стека технологий для разработки мобильного приложения, обеспечивающего эффективное и быстрое оповещение пользователей в ситуациях чрезвычайного характера по средствам пуш-уведомлений.

Пуш-уведомления – это способ отправки сообщений с сервера на устройства пользователей в реальном времени, даже когда приложение не активно или закрыто. В современном мире являются одной из важнейших технологий доставки мобильных оповещений, поэтому так важно реализовать мобильный клиент используя подходящие технологии.

Среди нативных технологий разработки мобильных приложений следует отметить Java, Kotlin, Swift и Objective-C. Язык программирования Java в своё время являлся одним из наиболее распространенных для разработки мобильных приложений под платформу Android. Kotlin, разработанный компанией JetBrains на основе языка Java и запускаемый на той же виртуальной машине, становится все более популярным в сообществе Android-разработчиков благодаря своей простоте и удобству. Swift и Objective-C, в свою очередь, являются основными языками программирования для разработки мобильных приложений под iOS.

Кроссплатформенные технологии разработки мобильных приложений предоставляют возможность создавать приложения, которые могут работать на разных операционных системах, таких как Android и iOS. Среди таких технологий стоит выделить язык программирования Dart, а конкретнее его фреймворк Flutter и React Native. Dart – язык программирования, разработанный компанией Google для создания кроссплатформенных приложений с использованием Flutter, мощного фреймворка для разработки пользовательских интерфейсов. React Native, в свою очередь, JS-фреймворк для создания нативно отображаемых iOS- и Android-приложений. В его основе лежит разработанная в Facebook JS-библиотека React, предназначенная для создания пользовательских интерфейсов. Но вместо браузеров она ориентирована на мобильные платформы.

Каждая из этих технологий имеет свои преимущества и недостатки. Нативные технологии, такие как Java, Kotlin, Swift и Objective-C, предлагают полный доступ к функциональным возможностям платформы, но требуют разработки двух отдельных версий приложения для Android и iOS. Кроссплатформенные технологии, такие как Dart (Flutter) и React Native, позволяют разработчикам использовать общий код для создания приложений под разные платформы, что упрощает и ускоряет процесс разработки.

Однако, при выборе технологии для проектирования системы оповещения пользователей о чрезвычайных ситуациях, особую роль играет максимальная скорость разработки и возможность оперативно вносить изменения и обновления при необходимости. С учетом этого фактора лучшим выбором является React Native.

По мимо скорости разработки React Native удивляет тем, что он «действительно» нативный. Другие решения JavaScript для мобильных платформ просто оборачивают ваш JS-код в некоторое веб-представление. Они могут «перереализовать» какое-нибудь нативное поведение интерфейса, например, анимацию, но всё же это остаётся веб-приложение [15].

В React компонент описывает собственное отображение, а затем библиотека обрабатывает рендеринг. Эти две функции разделены ясным уровнем абстракции. Если нужно отрисовать компоненты для веб-страницы, то React использует стандартные HTML-тэги. Благодаря тому же уровню абстракции – «мосту» – для рендеринга в iOS и Android, React Native вызывает соответствующие API (Рисунок 2.7). В iOS компоненты отрисовываются в настоящие UI-виды, а в Android — в нативные виды [15].

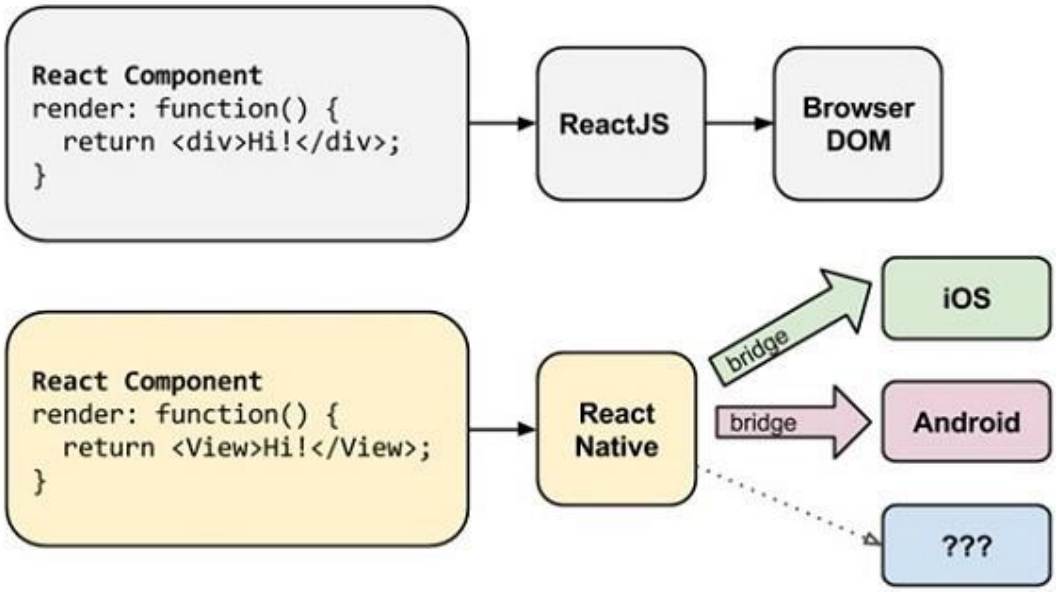

Рисунок 2.7 – Работа библиотек React и React Native [15]

Написав привычный код, очень похожий на стандартный JavaScript, CSS и HTML. Вместо его компиляции в нативный код, React Native берёт ваше приложение и запускает его с помощью JS-движка хост-платформы, без блокирования основного UI-потока. Таким образом получаем преимущества нативной производительности, анимации и поведения, без необходимости писать код на Objective-C или Java.

Другие методы разработки кроссплатформенных приложений, вроде Cordova или Titanium, не могут предоставить такой уровень нативной производительности и отображения.

Исходя из всех факторов, таких как скорость разработки, уровень нативной производительности, а также того факта, что для разработки веб-клиента мы так же выбрали библиотеку React.js, можно не сомневаться в правильности выбора React Native в качестве технологии для мобильной разработки приложения.

2.10 Экосистема

Развитие современных информационных технологий привело к возникновению сложных распределенных систем, включающих множество компонентов и сервисов. При проектировании системы оповещения пользователей о чрезвычайных ситуациях необходимо учесть множество факторов, связанных с ее масштабируемостью, надежностью и удобством управления. Для достижения этих целей необходимо использовать современные технологии и инструменты, способные обеспечить эффективное функционирование экосистемы данного приложения.

Современные экосистемы вокруг микросервисных архитектур строятся на основе распределенной инфраструктуры, где различные компоненты системы запускаются на физическом или виртуальном оборудовании. Для запуска больших распределенных систем на железе используются средства виртуализации, позволяющие эффективно управлять вычислительными ресурсами и обеспечивать высокую доступность сервисов.

Для построения экосистемы необходимо создать инфраструктуру, включающую в себя такие компоненты, как серверы, сетевые устройства, хранилища данных и инструменты для управления и мониторинга системы. Кроме того, важным элементом инфраструктуры является платформа для контейнеризации, такая как Docker, которая позволяет упаковывать приложения и их зависимости в контейнеры для более эффективного развертывания и управления.

2.10.1 Веб-сервер

Веб-сервер – это программное обеспечение, которое выполняет основную роль в обработке запросов от клиентских устройств (обычно веб-браузеров) и отправки ответов в виде веб-страниц или других ресурсов. Веб-серверы являются важной частью архитектуры веб-приложений и играют ключевую роль в обеспечении доступности веб-сайтов и веб-служб.

Серверы являются основным строительным блоком инфраструктуры микросервисной архитектуры. Две наиболее популярные технологии – Nginx и Apache – предоставляют функциональность веб-серверов и прокси-серверов, обеспечивая балансировку нагрузки, кэширование и повышение производительности.

Из них Nginx обладает рядом преимуществ, таких как меньшее потребление ресурсов, высокая производительность и гибкость конфигурации, что делает его предпочтительным выбором для построения инфраструктуры. Nginx – это веб-сервер с открытым исходным кодом, изначально разработанный для обработки высоких нагрузок и обеспечения максимальной производительности. Он стал широко используемым и популярным инструментом в пространстве DevOps благодаря своей способности балансировать нагрузку и достигать высокой производительности при работе с множеством копий приложения.

Nginx использует асинхронную модель обработки запросов. Он может эффективно обрабатывать множество подключений с помощью асинхронной, событийно-ориентированной модели, не создавая отдельных потоков для каждого запроса (Рисунок 2.8). Nginx также может действовать в качестве прокси-сервера, распределяя трафик между копиями приложений для балансировки нагрузки [16].

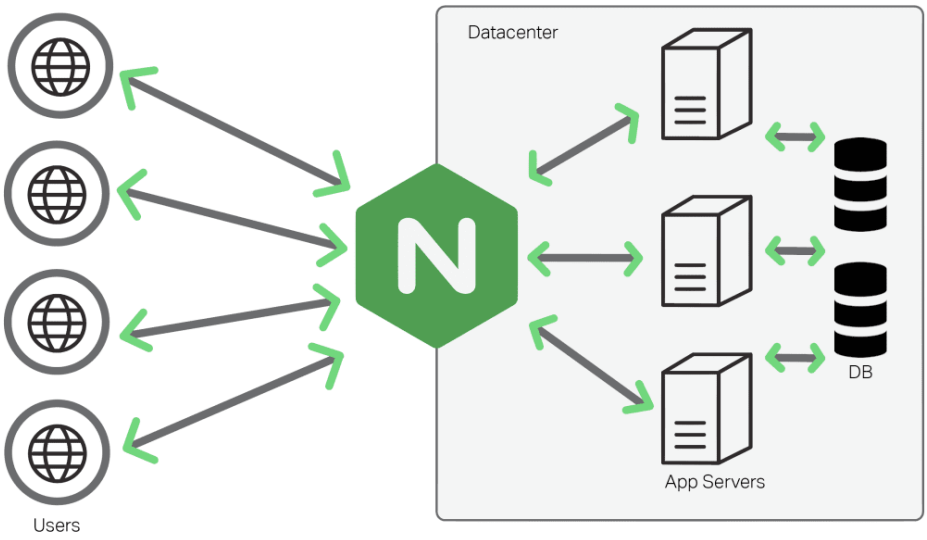

Рисунок 2.8 – Работа Nginx с множество подключений [16]

Кроме того, Nginx поддерживает HTTP-кэширование статических ресурсов, что позволяет снизить нагрузку на сервер и ускорить доставку контента клиентам. Он также обладает встроенными возможностями отказоустойчивости, позволяющими перенаправлять запросы к другим серверам в случае отказа одного сервера [16].

Гибкая конфигурация Nginx позволяет оптимизировать его производительность, выбирать алгоритмы балансировки нагрузки, устанавливать ограничения на ресурсы и настраивать кластеры серверов в соответствии с требованиями приложения [16].

2.10.2 Docker

Docker – это среда для разработки, доставки и выполнения приложений в контейнерах [17]. Он позволяет упаковывать приложения со всеми их зависимостями в небольшие, изолированные контейнеры, которые могут быть запущены практически на любой операционной системе. Docker обеспечивает унифицированную платформу для развертывания микросервисов, упрощает масштабирование системы и облегчает процесс управления зависимостями и обновлениями.

Стандартом в индустрии долгое время было (для кого-то и сейчас) – использовать виртуальные машины для запуска приложений. Виртуальные машины запускают приложения внутри гостевой операционной системы, которая работает на виртуальном железе основной операционной системы сервера (Рисунок 2.9).

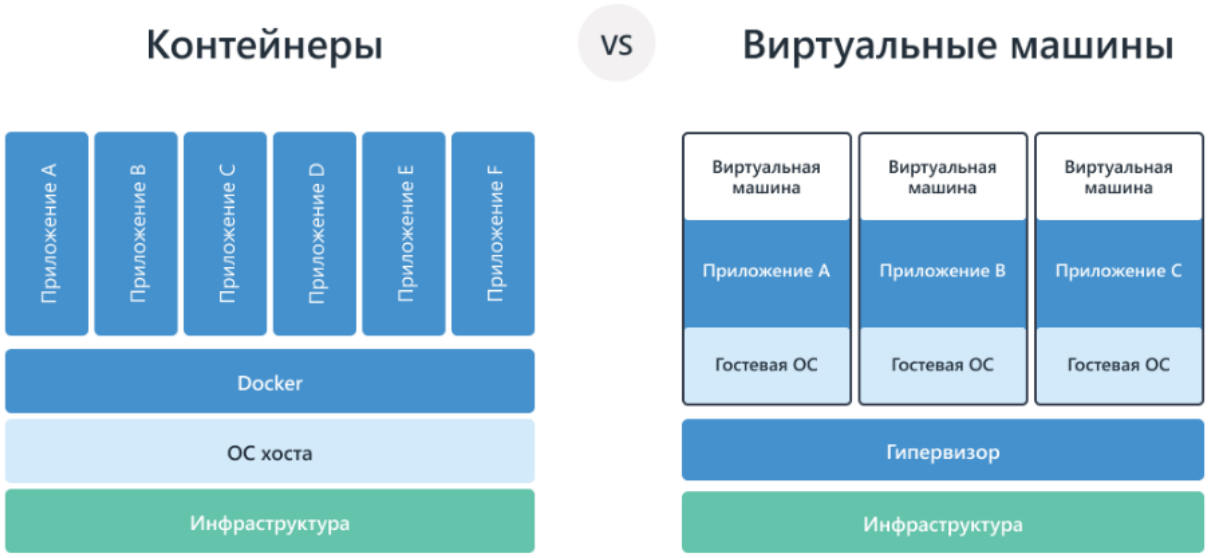

Рисунок 2.9 – Сравнение Docker контейнеров с виртуальными машинами [17]

Виртуальные машины отлично подходят для полной изоляции процесса для приложения: почти никакие проблемы основной операционной системы не могут повлиять на софт гостевой операционной системы, и наоборот. Но за такую изоляцию приходится платить. Существует значительная вычислительная нагрузка, необходимая для виртуализации железа гостевой операционной системы.

Контейнеры используют другой подход: они предоставляют схожий с виртуальными машинами уровень изоляции, но благодаря правильному задействованию низкоуровневых механизмов основной операционной системы делают это с в разы меньшей нагрузкой.

Использование Docker в микросервисной архитектуре предоставляет ряд преимуществ. Он обеспечивает изоляцию между микросервисами, предотвращая конфликты и обеспечивая стабильность. Контейнеры также

обладают портативностью, что позволяет их запускать на различных платформах и легко масштабировать (Рисунок 2.10).

Docker упрощает развертывание микросервисов, обеспечивая повторяемость и простоту в настройке. Это также способствует ускорению цикла разработки и улучшает сотрудничество, позволяя разработчикам создавать и запускать контейнеры на разных средах. Общение и взаимодействие между разработчиками и DevOps также упрощаются благодаря использованию единого типа контейнеров [17].

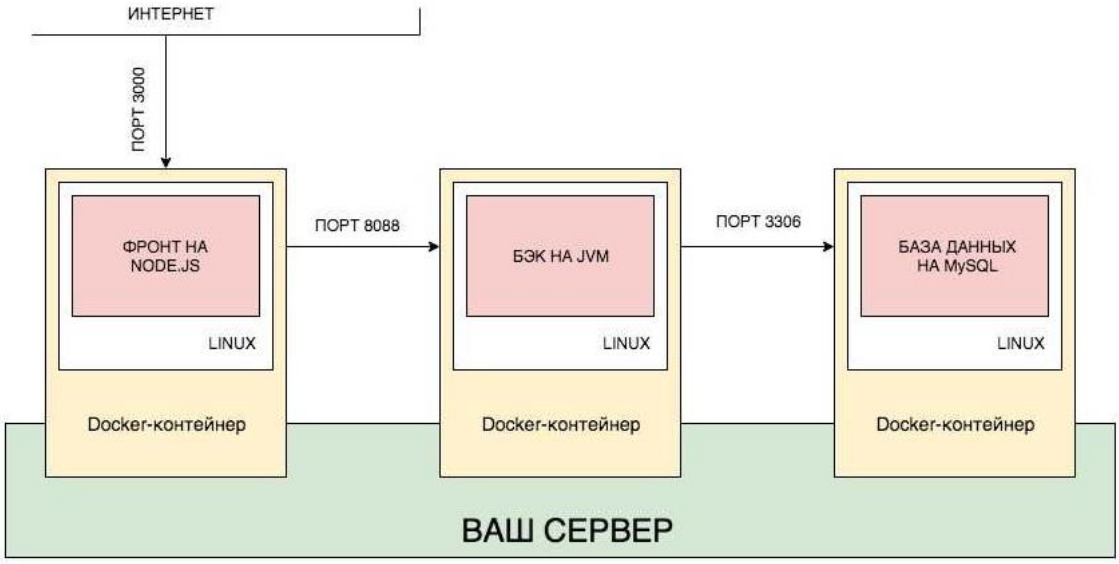

Рисунок 2.10 – Контейнеризация отдельных компонентов системы с помощью Docker [17]

2.10.3 Kubernetes

Kubernetes (также известный как K8s) – это система для автоматизации развертывания, масштабирования и управления контейнеризированными приложениями. Он является одним из наиболее популярных инструментов в области контейнеризации и оркестрации приложений. Предоставляет набор инструментов для управления кластером серверов, на которых работают контейнеры, и позволяет эффективно управлять ресурсами, обеспечивать

высокую доступность и автоматически масштабировать приложения в зависимости от нагрузки [18].

Kubernetes позволяет упаковывать приложения и их зависимости в контейнеры, которые затем можно развернуть на любой подходящей инфраструктуре – физических серверах, виртуальных машинах или в облаке. Контейнеры в Kubernetes группируются в наборы подов (pods), которые являются базовыми единицами развертывания и масштабирования [18].

Kubernetes поднимает кластер, который может состоять как из нескольких физически отдельных серверов, так и на одном мощном сервере, на котором запущено некоторое количество виртуальных машин, между которыми грамотно распределена мощность «железа» (Рисунок 2.11). K8s является невероятно мощнейшим и очень сложным инфраструктурным инструментом, и достоен огромного количества отдельных научных работ. Поэтому точно не будет в полной мере раскрыт в этой курсовой, но здесь будут освещены основные его возможности, понимания которых достаточно, для начала взаимодействия с ним.

Kubernetes поддерживает горизонтальное масштабирование, автоматическое восстановление после сбоев, самообслуживание и динамическое управление ресурсами, то есть распределение мощностей железа между узлами, а также распределение нагрузок. Он обеспечивает устойчивость к отказам, обнаружение и автоматическую замену неработающих подов, управление сетевыми связями между различными подами и обеспечение надежности работы приложений.

Kubernetes важен для DevOps инженерии, так как позволяет упростить и автоматизировать процессы развертывания и управления микросервисными приложениями. В микросервисной архитектуре приложение разделено на небольшие, независимо разворачиваемые и масштабируемые сервисы. Kubernetes предоставляет средства для оркестрации и балансировки нагрузки между этими сервисами, обеспечивая удобное масштабирование каждого сервиса отдельно.

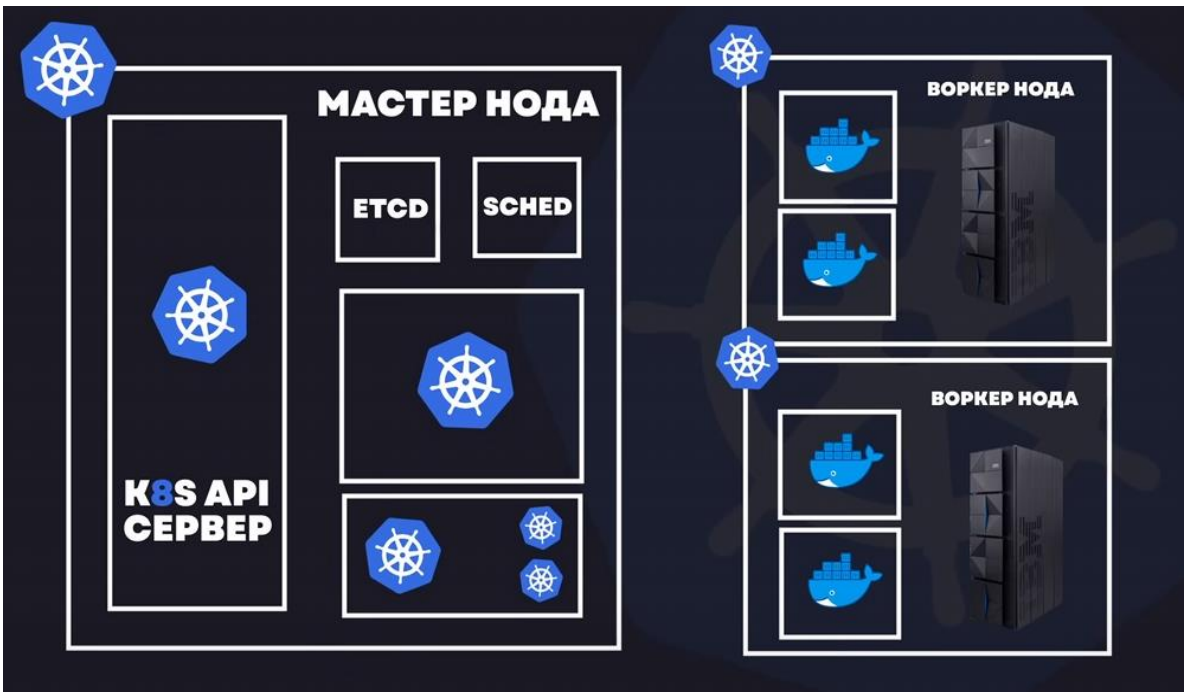

Рисунок 2.11 – Kubernetes упрощённая визуализация развёртывания на физических серверах [18]

Kubernetes интегрируется с Docker, который является одной из наиболее популярных платформ для контейнеризации приложений (Рисунок 2.11). Docker позволяет упаковывать приложение и все его зависимости в изолированный контейнер, в котором поднята упрощённая версия операционной системы (LXC - технология на базе ядра Linux). Kubernetes же управляет и оркестрирует работу этих контейнеров на кластере серверов. Docker позволяет создавать контейнеры, в то время как Kubernetes берет на себя ответственность за их развертывание, масштабирование, мониторинг и управление.

2.10.4 CI/CD

CI/CD (Continuous Integration/Continuous Delivery) – это подход к разработке программного обеспечения, включающий непрерывное интегрирование изменений в код и автоматическую поставку приложения в производственную среду. Этот процесс позволяет ускорить разработку и

улучшить качество программного обеспечения, а также обеспечивает быстрое внедрение изменений в систему оповещения [19].

Основные этапы CI/CD включают:

а) Continuous Integration (непрерывную интеграцию): на этом этапе разработчики регулярно интегрируют свой код в общий репозиторий, используя систему управления версиями, такую как Git. Это позволяет обнаруживать и устранять конфликты между различными фрагментами кода и своевременно выявлять ошибки [19].

б) Build and Test Automation (автоматизация сборки и тестирования): после интеграции кода CI-система выполняет автоматическую сборку приложения и запускает набор автоматических тестов для проверки его работоспособности. Это помогает выявлять ошибки и проблемы в коде на ранних стадиях разработки [19].

в) Continuous Delivery (непрерывную доставку): при успешном завершении сборки и тестирования, CI/CD-система выпускает приложение в промежуточную среду или стейджинг. Здесь оно проходит дополнительное тестирование и проверку перед окончательной доставкой в продакшн-среду [19].

г) Continuous Deployment (непрерывное развертывание): на этом этапе автоматизированно доставляется приложение в продакшн-среду после успешного прохождения всех тестов и проверок. Это позволяет обновлять приложение в продакшн-среде практически мгновенно и безопасно [19].

Технологии, которые могут быть использованы для реализации CI/CD, включают следующие 6 направлений:

1) Системы управления версиями: например, Git, позволяют эффективно управлять кодом и его версиями.

2) CI/CD-серверы: например, Jenkins, Travis CI, CircleCI, GitLab CI/CD, TeamCity - предоставляют инструменты для автоматической сборки, тестирования и доставки приложений.

3) Контейнеризация: Технологии, такие как Docker, позволяют упаковывать приложение и его зависимости в контейнеры для облегчения развертывания и управления конфигурациями.

4) Оркестрация контейнеров: например, Kubernetes, позволяет управлять и масштабировать контейнеризованным приложением в распределенной среде.

5) Инструменты автоматизации тестирования: например, Selenium, JUnit, NUnit, PHPUnit - позволяют автоматизировать тестирование приложения на различных уровнях (функциональном, интеграционном и т.д.).

6) Инструменты для инфраструктуры как кода: например, Terraform, Ansible, AWS CloudFormation - позволяют описывать инфраструктуру и ее конфигурацию в виде кода, что упрощает ее развертывание и управление.

2.10.5 Итоги по инфраструктуре

Современные экосистемы вокруг микросервисных архитектур строятся на базе распределенной инфраструктуры, которая включает в себя серверы, сетевые устройства, хранилища данных и инструменты для управления и мониторинга системы. Большие распределенные системы запускаются на физическом или виртуальном оборудовании с использованием средств виртуализации. Это позволяет эффективно управлять вычислительными ресурсами, обеспечивать высокую доступность сервисов и обеспечивать масштабируемость системы.

Docker, Kubernetes и Nginx – три популярных инструмента, используемых в разработке и развертывании микросервисных архитектур. Docker предоставляет платформу для создания и управления контейнерами, что позволяет упаковывать приложения и их зависимости в изолированные среды для запуска на любой платформе. Kubernetes обеспечивает управление контейнерами и автоматическую оркестрацию, позволяя запускать, масштабировать и управлять приложениями с высокой доступностью и

отказоустойчивостью. Nginx работает в качестве балансировщика нагрузки, распределяя трафик на контейнеры внутри кластера Kubernetes и обеспечивая равномерное распределение нагрузки и повышение производительности системы (Рисунок 2.12). Вместе эти инструменты обеспечивают эффективное управление и развертывание микросервисных архитектур [20].

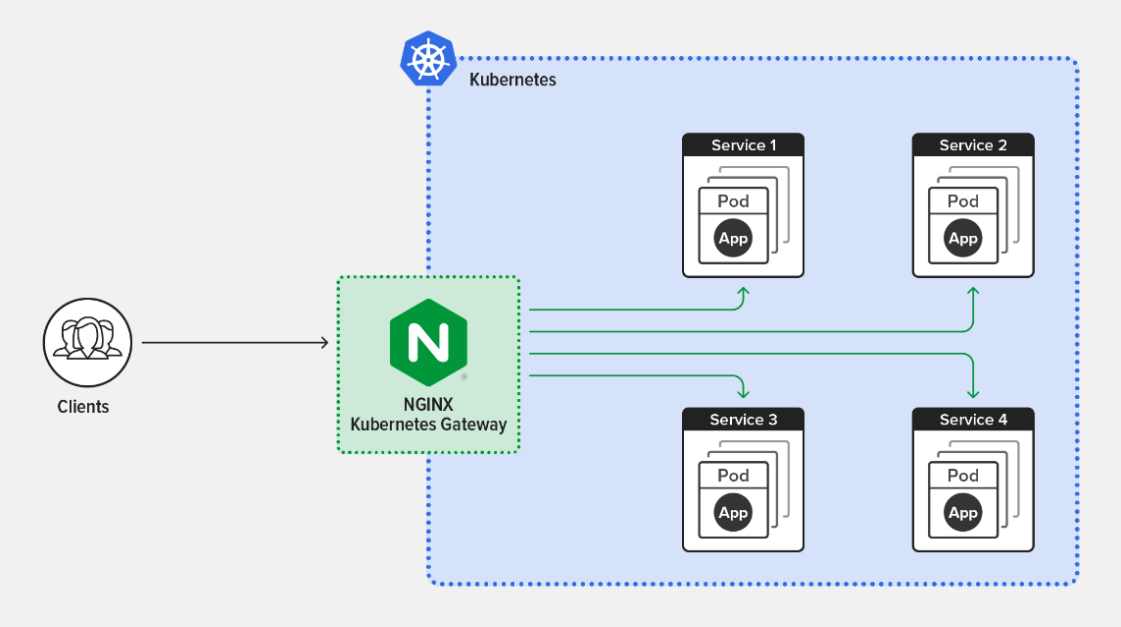

Рисунок 2.12 – Nginx распределяет трафик на контейнеры внутри кластера Kubernetes [20]

В процессе разработки микросервисных архитектур широко применяется подход CI/CD (Continuous Integration/Continuous Delivery), который позволяет автоматизировать процесс интеграции изменений в код и непрерывной поставки приложений в производственную среду. Что ускорит разработку, и обеспечить быстрое внедрение изменений в систему оповещения пользователей о чрезвычайных ситуациях.

3 Проектирование системы

В третьей главе акцент смещён на проектирование системы в контексте микросервисной архитектуры. Основная цель по-прежнему сосредоточена на разработке платформы для уведомления о чрезвычайных ситуациях, при этом особое внимание было уделено визуализации, как системы целиком, так и разных её аспектов в частности, в виде различных макетов и UML-диаграмм. Таким образом главная цель третьей главы заключается в аналитике и проектировании разрабатываемой системы, что в свою очередь закрывает основную цель курсовой работы.

Вначале были проведены анализ и декомпозиция предметной области, результаты которых представлены далее в виде различных UML-диаграмм. Этот этап был нужен, чтобы увидеть взаимосвязи между компонентами системы и определить основные функции каждого из них.

Далее представлена общая структура системы. Описан процесс работы над системным дизайном и представлены макеты – его результат. Это необходимо для лучшего понимания о размещении компонентов в архитектуре системы и их взаимодействие друг с другом.

Затем рассмотрено проектирование каждого сервиса в отдельности. Определены основные функции и задачи, которые будут выполняться каждым сервисом, а также описано их взаимодействие между собой и с другими компонентами системы.

Следующим этапом было проектирование баз данных. Поэтому будут представлены разработанные ER-диаграммы баз данных, а также описание взаимодействия между системой и базами данных, которые позволили разработать эффективную схему хранения и обработки данных, облегчающую работу с информацией в рамках проектируемой системы.

Заключительным шагом, было проектирование дизайна пользовательского интерфейса следую принципам UX/UI. Это помогло

создать удобный и интуитивно понятный пользовательский интерфейс для платформы.

В итоге, составлен обобщенный план для разработки проекта, основанный на проведенном анализе и проектировании системы, и задокументирован в третьей главе данной работы.

3.1 Анализ и декомпозиция предметной области

При анализе рынка подобных систем были выявлены основные требования, такие как высокая доступность, масштабируемость, надежность, отказоустойчивость и удобство использования. Для достижения этих целей было необходимо правильно разобраться с той областью знаний, для которой проектируется информационная система. Поэтому была проведена декомпозиция предметной области, которая включает в себя следующие 4 этапа:

1) Идентификация сущностей: определение основных сущностей, которые будут использоваться в системе, таких как пользователи, сообщества, шаблоны уведомлений и т.д.

2) Определение взаимодействия между сущностями: создание диаграмм взаимодействия между сущностями, которые позволят нам определить, как данные сущности взаимодействуют друг с другом.

3) Создание диаграмм классов, которые позволяют нам представить классы, используемые в системе, и их взаимодействие.

4) Создание диаграмм последовательностей, которые позволяют нам представить последовательность действий, выполняемых системой.

Декомпозиция – это процесс разбиения сложной системы на более мелкие компоненты. Это позволяет упростить проектирование и улучшить понимание структуры системы. А визуализация этого процесса позволяет более наглядно работать с архитектурой приложения, и ещё на этапе

проектирования выявлять различные сложности и исключительные ситуации, и придумывать методы их обработки.

Для визуализации существует UML (Unified Modeling Language) — это универсальный язык моделирования, используемый для визуализации, проектирования и документирования различных аспектов программных систем. Другими словами, UML это набор графических элементов и нотаций, которые обладают своей семантикой. Благодаря чему визуализация становитсяа очень информативной и понятной всем, кто знает язык UML.

Результатом декомпозиции рассматриваемой предметной области стали следующие 5 диаграмм:

1) Диаграмма вариантов использования или Use Case диаграмма (Приложение А, Рисунок А.1) - данная диаграмма показывает кем и как будет использоваться информационная система, это необходимо для понимания, а чего вообще будут ожидать от системы, тем самым начинаем проектирование с постановки вопроса, а что вообще необходимо продумать?

2) Классовые диаграммы (Приложение А, Рисунок А.2 – А.3) – на них изображены основные сущности проектируемой системы, и схематически показано их взаимодействие. Эти диаграммы помогут в дальнейшем при разработке блока Model в контексте паттерна MVC.

3) Диаграмма BPMN (Business Process Model and Notation) диаграмма бизнес-процесса (Приложение А, Рисунок А.4) – модель, в которой мы опишем весь процесс работы приложения, от создания шаблона, до его получения другими пользователями.

4) Диаграмма последовательности или Sequence Diagram (Приложение А, Рисунок А.5) – в заключительной визуализации представлена последовательность взаимодействий различных сервисов нашей информационной системы.

Таким образом, процесс декомпозиции предметной области и визуализация его результатов при помощи UML позволили более глубоко и точно проработать архитектуру информационной системы. В результате

проведенной декомпозиции предметной области, были получены основные диаграммы, которые помогут лучше понять структуру и взаимодействие компонентов системы, что позволит ускорить процесс дальнейшей её разработки, и минимизирует количество ошибок.

3.2 Общая структура

В больших распределенных системах, таких как платформа для оповещения населения о чрезвычайных ситуациях, системный дизайн играет важную роль в обеспечении масштабируемости, отказоустойчивости и эффективности работы системы.

System Design – это процесс проектирования архитектуры программной системы. Он включает в себя выбор технологий и инструментов, определение стратегии управления проектом, разработку общей структуры системы и проектирование сервисов, баз данных и пользовательского интерфейса.

System Design является важным этапом разработки любой программной системы. Он позволяет определить архитектуру системы, выбрать технологии и инструменты, а также разработать стратегию управления проектом.

Дизайн системы, это то, что является продуктом данной курсовой работы. Уже были описаны этапы: анализа рынка, подбора подходящих технологий и архитектурных паттернов, а также была проведена декомпозиция предметной области, и проработаны основные процессы, протекающие через проектируемую систему. Теперь проектирование переходит на более высокий уровень абстракции, и визуализировать мы будем всю систему целиком, включая сторонние технологии.

Для начала обобщим используемые паттерны системного дизайна в следующих 7 пунктах:

1) Клиент-сервер: разделяет функциональные возможности между клиентом и сервером, обеспечивая масштабируемость удобство в разработке.

2) Многоуровневая архитектура. Организует компоненты системы по слоям, обеспечивая возможность повторного использования и удобства обслуживания.

3) Отказоустойчивость: этот паттерн предназначен для обеспечения непрерывной работы системы даже в случае сбоев или отказов в отдельных компонентах. Распределение нагрузки, репликация данных, автоматическое восстановление и контроль за состоянием компонентов - ключевые аспекты отказоустойчивости.

4) Микросервисная архитектура: концепция предполагает разделение приложения на небольшие и независимые сервисы, каждый из которых сфокусирован на конкретной функциональности. Это обеспечивает гибкость, масштабируемость и возможность быстрой разработки и развертывания.

5) Паттерн трехуровневой архитектуры: помогает разделить систему на три основных слоя - представление, бизнес-логику и доступ к данным.

а) Представление (Presentation Layer) – этот слой отвечает за представление данных пользователю и взаимодействие с пользователем.

б) Бизнес-логика (Business Logic Layer или Domain Layer) - этот слой содержит бизнес-логику и правила обработки данных.

в) Доступ к данным (Data Access Layer) – этот слой отвечает за доступ к данным и работу с хранилищами данных (например, базы данных).

6) Pub/Sub (Публикатор-подписчики): использует систему обмена сообщениями, в которой издатели отправляют сообщения без ведома подписчиков, что обеспечивает слабую связь и масштабируемость.

7) MVC (Model-View-Controller) Модель-Представление-Контроллер – это подход, который разделяет приложение на три основных компонента: модель, представление и контроллер. Модель представляет данные и бизнес-логику приложения. Представление отображает данные модели на экране. Контроллер обрабатывает пользовательский ввод и координирует взаимодействие между моделью и представлением [21].

В соответствии со всеми паттернами и концепциями, а также опираясь на модели, построенные при декомпозиции, мы получаем следующий обобщённый системный дизайн (Приложение Б, Рисунок Б.1).

На первой диаграмме системного дизайна показано общее взаимодействие системы, и 5 её основных составляющих:

1) Веб-клиент на React – это компонент, который отвечает за отображение пользовательского интерфейса на веб-странице, а также через него пользователь взаимодействует с сервером по средствам клиент-серверной архитектуры, отправляя HTTP запросы, составленные в правилах концепции REST API. Он состоит из различных экранов, подробнее о нём в части 3.5.

2) Мобильный клиент на React Native – это клиентское приложение, которое отвечает за отображение пользовательского интерфейса на мобильных устройствах, и осуществляет точно такое же взаимодействие с сервером, как и веб-интерфейс.

3) Серверная часть – в нашем случае это большое многослойное серверное приложение, который отвечает за обработку запросов от веб-клиента и мобильного клиента, а также за доставку пуш-уведомлений. Он состоит из различных сервисов, таких как сервис обработки запросов, балансировщики нагрузки, брокеры данных, базы данных, системы кэширования данных, сервисы отправки уведомлений (Telegram, почта, пуш-уведомления).

4) Внешние сервисы – это сервисы сторонних разработчиков, которые могут быть использованы в системе для выполнения определенных функций (например, сервисы отправки пуш-уведомлений).

5) Так же помимо серверной части, можно выделить ещё одну часть, которая не прерывна связана с серверной разработкой – инфраструктура. Основной часть инфраструктуры проектируемого приложения стали: Docker, Kubernetes, Nginx (Приложение Б, Рисунок Б.2).

Таким образом имеем готовый макет дизайна общей структуры будущей системы. Теперь же перейдём к каждому сервису, в частности.

3.3 Сервисы

Любой сервис в системе выполняет свою функцию и отвечает за определенную область функциональности. Например, есть сервисы, отвечающие за отправку уведомлений пользователям, при этом в разрабатываемой системе их три вида. Так же есть сервис обработки запросов отвечает за обработку запросов от веб-клиента и мобильного клиента.

В следующих 6 пунктах описаны как задачи каждого сервиса или компонента системы в отдельности, так и общая работа системы. Описание построено на основе диаграмм (Приложение А, Приложение Б), реализованных в предыдущих частях, при этом пункты стоят в хронологической последовательности обработки транзакции внутри информационной системы:

1) Первым шагом, компоненты пользовательского интерфейса, а именно веб-клиент или мобильный клиент, в ответ на действие пользователя, отправляют POST запрос на серверную часть нашего приложения. В этом запросе содержится JSON файл, в котором заключена вся информация о новосозданном шаблоне уведомления (в данном примере моментально реализуемого уведомления).

2) Запрос принимается сервером, в нашем случае веб-сервером является Nginx, балансировщик которого крутиться в нашей инфраструктуре, управляемой оркестратором Kubernetes. Запрос попадает на одни из копий нашего приложения, сервисы которого контейнерезированы системой Docker.

3) Первым сервисом на пути запроса становиться API сервис, который принимает все запросы, и реализует дальнейшую логику, над пришедшими данными. Записывает шаблон в MongoDB, и так как он моментальный, сразу начинает его реализацию:

– считывает всю информацию о том, кому нужно доставить уведомления по данному шаблону, и какими способами эти пользователи готовы принимать оповещение;

– формирует очереди сообщений в Kafka, разделённые на топики для трёх разных вариантов получения (Telegram, почта, пуш-уведомление), с учётом того, как выбрали для себя пользователи.

4) Сервисы, которые мы будем именовать как Workers (три вида сервисов, занимающихся непосредственной реализацией оповещения), имеют по несколько копий в системе, каждая из которых читает свою партицию из определённого топика в Kafka, и реализует уведомление по заданной для каждого из трёх типов Workers логике, пока не осуществит всю очередь.

5) Если какие-то из уведомлений небыли доставлены по причине сбоев, оно повторно перенаправляются на Kafka, для реализации через другие копии Workers нужного типа, которые функционируют в штатном режиме.

6) Заключительный пункт актуален только для осуществления оповещения посредствам пуш-уведомлений, так как их реализацией занимается мобильный клиент. Он принимает информацию с сервера, и реализует нативными средствами уведомление. Остальные же сервисы на прямую доставляют оповещения через боты на почту или в Telegram.

Помимо всех тех сервисов, что визуально отображены на диаграммах, на которые были ссылки в предыдущих частях, существуют внутренние сервисы, которые можно рассматривать и как модули основного сервиса обработки запросов, так и отдельные сервисы. Примерами таких сервисов являются:

– сервис, который отвечает за мониторинг работоспособности системы, и сигнализирует о её неисправностях.

– сервис, который занимается внутренней балансировкой пуш-уведомлений, перенаправляя их на Kafka, а также регулирует статусы в БД, чтобы убедиться в повторной отправке уведомления, при предыдущей неудаче.

3.4 Проектирование баз данных

Для проектирования структуры хранилищ данных в системе была использована ещё одна вариация UML-диаграммы, а именно ER-диаграмма или по-другому – диаграмма сущность-связь. Она позволила определить структуру базы данных и связи между сущностями, что в последствии поможет при создании архитектуры, процедур и триггеров внутри самой базы, используя диаграмму (Приложение В), в качестве руководства к реализации.

ER-диаграммы как и классовые являются самыми информативными из всех UML-диаграмм, ведь они не ограниченны в детализации, и при правильном использовании и хорошем знании языка UML, способны в полной мере перенести предметную область на код.

Качественно спроектированная ER-диаграмма может быть полностью перенесена на язык SQL без дополнительных правок, что позволяет в разы ускорить разработку базы данных. Кроме того, тщательный подход к проектированию помогает выявить все недопонимания, сложности, и исключительные ситуации до процесса реализации, что в свою очередь способствует минимизации багов, и дополнительной работы над кодом базы.

3.5 Проектирование пользовательского интерфейса

Для создания удобного и функционального пользовательского интерфейса были использованы принципы современного UX/UI дизайна. Были созданы макеты пользовательского интерфейса, которые позволяют легко находить нужную информацию и выполнять необходимые действия. Так же учтены принципы доступности и удобства использования для лиц любых возрастных групп.

UI/UX расшифровывается как User interface / User experience, что в переводе на русский язык «пользовательский интерфейс / пользовательский опыт». Эти термины относятся к дисциплинам дизайна, которые направлены

на повышение удовлетворенности пользователей за счет улучшения удобства использования, доступности и удовольствия, получаемого от взаимодействия между пользователем и продуктом [22].

UI (User Interface) – это оформление и визуальный аспект интерфейса, который взаимодействует с пользователем. Он включает в себя элементы дизайна, такие как цвета, шрифты, изображения и макеты, а также взаимодействие с пользователем через кнопки, поля ввода, переключатели и другие элементы интерфейса. UI является важным аспектом веб-приложений, так как он отвечает за передачу информации от пользователя к системе и обратно [22].

UX (User Experience) – это опыт, который пользователь получает при взаимодействии с веб-приложением. Он включает в себя эмоциональные, практические и ценностные аспекты, связанные с использованием приложения. Хороший UX обеспечивает удобство, удовлетворенность и эффективность взаимодействия пользователя с приложением. Он учитывает потребности и ожидания пользователей, предоставляет интуитивно понятные функции и шаги взаимодействия, а также обеспечивает эстетическое удовлетворение [23].

Простыми словами UI конкретно касается визуальных элементов продукта, в то время как UX охватывает более широкую область, включая общее впечатление пользователя от продукта, удобство, интуитивность и логичность в использовании.

Основные этапы создания современного и качественного дизайна веб-приложений включают в себя исследование, проектирование, прототипирование [23]:

– На первом этапе исследования производится анализ целевой аудитории и их потребностей. Исследование включает изучение рынка, конкурентов, анализ трендов.

– На втором этапе проектирования определяются основные концепции, идеи и цели приложения. Создается информационная архитектура, структура

и навигация, а также определяются основные функциональные и эстетические требования.

– На третьем этапе прототипирования создается прототип интерфейса, который демонстрирует взаимодействие пользователя с приложением. Прототип позволяет проверить удобство использования, предоставить обратную связь и внести корректировки в дизайн.

Готовый макет, реализованный в приложении Figma, можно использовать при реализации клиентской части. Figma – это популярное онлайн-инструмент для проектирования пользовательского интерфейса и разработки дизайна, которое предоставляет множество функций, позволяющих разработчикам создавать интерактивные прототипы и представлять дизайн веб-приложений. При чем некоторые CSS свойства можно импортировать на прямую из проекта с макетом нашего приложения в Figma в программный код.

В разработке дизайна и создании макетов для платформы, обеспечивающей возможность оповещения пользователей о чрезвычайных ситуациях, были применены принципы современного UX/UI дизайна для создания удобного и функционального пользовательского интерфейса веб-приложения. Разработаны макеты, которые обеспечивают легкий доступ к нужной информации и выполнение необходимых действий.

(Приложение Г) содержит макеты лэндинга (Рисунок Г.1), который является визитной карточкой приложения. Он предоставляет необходимую информацию о приложении. Так же в приложении находятся окна авторизации и регистрации (Рисунок Г.2 – Г.5), которые были разработаны в соответствии с основными принципами, перечисленными ранее, чтобы обеспечить простоту и удобство использования.

(Приложение Д) включает в себя макеты пользовательского интерфейса, которые обеспечивают удобство навигации и доступа к основным функциям приложения, так как сделаны на основе знакомого всем UX социальных-сетей

включая элементы управления, размещение информации и общую организацию пользовательского интерфейса.

(Приложение Е) содержит макеты интерфейса, для сообществ. Макеты реализованы по тому же принципу, что и интерфейс обычных пользователей, но с некоторыми нюансами, связанными со спецификой деятельности сообществ.

В целом, разработанные макеты представляют собой удобный и привлекательный дизайн пользовательского интерфейса и обеспечивают гармоничный пользовательский опыт при работе с платформой для оповещения о чрезвычайных ситуациях.

ЗАКЛЮЧЕНИЕ

В рамках данной курсовой работы было проведено проектирование микросервисной архитектуры на Java с использованием фреймворка Spring. Целью работы – разработка платформы, предоставляющей возможность оповещения пользователей о чрезвычайных ситуациях, при этом обеспечивающая отказоустойчивость, и масштабируемость, обладая доступным, интуитивно понятным, современным пользовательским интерфейсом.

В первой главе был проведен анализ рынка аналогичных решений, и выявлена необходимость разработки нового современного решения, которое решает все проблемы аналогов.

Во второй главе был проведен анализ технологий и выбран стек, наиболее подходящий для реализации проекта. Были выбраны следующие технологии: Java, Spring, PostgreSQL, MongoDB, Kafka, Redis, Docker, Kubernetes, Nginx, и сопутствующие им паттерны и концепции.

В третьей главе был задокументирован процесс проектирования на основе предметной области и представлены визуализация проекта через UML-диаграммы. Были разработаны диаграммы классов, диаграмма бизнес-процесса, диаграмма последовательности, диаграмма сущность-связь, а также диаграммы системного дизайна и архитектуры системы.

В результате работы были получены UML-диаграммы и макеты пользовательского интерфейса, разработанные в Figma. Эти результаты успешно соответствуют поставленным задачам и целям работы. Оценивая полноту решения, можно сказать, что все необходимые аспекты и компоненты системы были учтены и проработаны.

Планируется использовать результаты данного исследования в следующих курсовой и дипломной работах при реализации данной информационной системы. Разработанные диаграммы и проектная

документация могут быть использованы в качестве основы для разработки и внедрения системы оповещения пользователей о чрезвычайных ситуациях.

Все вышеперечисленные аспекты являются гарантом того, что работа актуальна и станет ценным ресурсом для практического применения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ньюмен, С. Создание микросервисов / С. Ньюмен. – СПб.: Питер, 2023. – 624 с. – ISBN 978-5-4461-1145-9.

2. Просто о микросервисах // Хабр: [сайт]. – 2018. – URL: https://habr.com/ru/companies/raiffeisenbank/articles/346380/ (дата обращения: 07.11.2023).

3. Эванс, Э. Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем / Э. Эванс – ООО "И.Д. Вильямс", 2011 – 448 с. – ISBN 978-5-8459-1597-9.

4. Шилдт, Г. Java. Полное руководство / Г. Шилдт – СПб. "Диалектика", 2023 – 1344 с. – ISBN 978-5-907458-86-4.

5. Уоллс, К. Spring в действии / К. Уоллс – М.: ДМКПресс, 2022 – 544 с. – ISBN 978-5-93700-112-2.

6. Mihalcea, V. High-Performance Java Persistence: Get the most out of your persistence / V. Mihalcea – Leanpub, 2020 – 444 с. – ISBN 978-5-93951-352-8.

7. Редмонд, Э. Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQL / Э. Редмонд, Д. Р. Уилсон – М.: ДМК Пресс, 2013 – 384 с. – ISBN 978-5-94074-866-3.

8. Шаблоны интеграции корпоративных приложений / Г. Хоп, Б. Вульф, К. Брауна, К. Ф. Д'Круза [и др.]. – ООО ''И.Д. Вильямс'', 2007. – 672 с. – ISBN 978-5-8459-1146-9.

9. Carlson, J. L. Redis in Action / J. L. Carlson – Manning Publications Co., 2013 – 294 с. – ISBN 978-1-9351-8205-4.

10. Apache Kafka. Потоковая обработка и анализ данных. / Г. Шапира, Т. Палино, Р. Сиварам, К. Петти – СПб.: Питер, 2023. – 512 с. – ISBN 978-5-4461-2288-2.

11. Videla, A. RabbitMQ in Action / A. Videla, J.W. Williams – Manning Publications Co., 2012 – 287 с. – ISBN 978-1-9351-8297-9.

12. Чем различаются Kafka и RabbitMQ простыми словами // Хабр: [сайт]. – 2022. – URL: https://habr.com/ru/companies/innotech/articles/698838/ (дата обращения: 18.11.2023).

13. Ричардсон, К. Микросервисы. Паттерны разработки и рефакторинга / К. Ричардсон – СПб.: Питер, 2019. – 544 с.: – ISBN 978-5-4461-0996-8.

14. Что нужно знать о популярных JS-фреймворках // Хабр: [сайт]. – 2020. – URL: https://habr.com/ru/companies/yandex_praktikum/articles/533702/ (дата обращения: 18.11.2023).

15. Создание кроссплатформенных приложений с помощью React Native // Хабр: [сайт]. – 2020. – URL: https://habr.com/ru/companies/nix/articles/324562/ (дата обращения: 19.11.2023).

16. NGINX изнутри: рожден для производительности и масштабирования // Хабр: [сайт]. – 2015. – URL: https://habr.com/ru/articles/260065/ (дата обращения: 23.11.2023).

17. Полное практическое руководство по Doker: с нуля до кластера на AWS // Хабр: [сайт]. – 2016. – URL: https://habr.com/ru/articles/310460/ (дата обращения: 23.11.2023).

18. Основы Kubernetes // Хабр: [сайт]. – 2015. – URL: https://habr.com/ru/articles/258443/ (дата обращения: 23.11.2023).

19. Что такое CI/CD? Разбираемся с непрерывной интеграцией и непрерывной поставкой // Хабр: [сайт]. – 2020. – URL: https://habr.com/ru/companies/otus/articles/515078/ (дата обращения: 23.11.2023).

20. Разворачиваем веб-приложение в Kubernetes с нуля // Хабр: [сайт]. – 2023. – URL: https://habr.com/ru/articles/752586/ (дата обращения: 23.11.2023).

21. Карнелл, Д. Микросервисы Spring в действии / Д. Карнелл, И. У. Санчес – М.: ДМК Пресс, 2022. – 490 с. – ISBN 978-5-97011-971-2.

22. Унгер, Р. UX-дизайн. Практическое руководство по проектированию опыта взаимодействия / Р. Унгер, К. Чендлер – СПб.: Символ-Плюс, 2011. – 336 с. – ISBN 978-5-93286-184-4.

23. Эяль, Н. На крючке. Как создавать продукты, формирующие привычки / Н. Эяль, Р. Хувер – М.: Манн, Иванов и Фербер, 2017. – 272 с. – ISBN 978-5-00100-554-4.

ПРИЛОЖЕНИЕ А

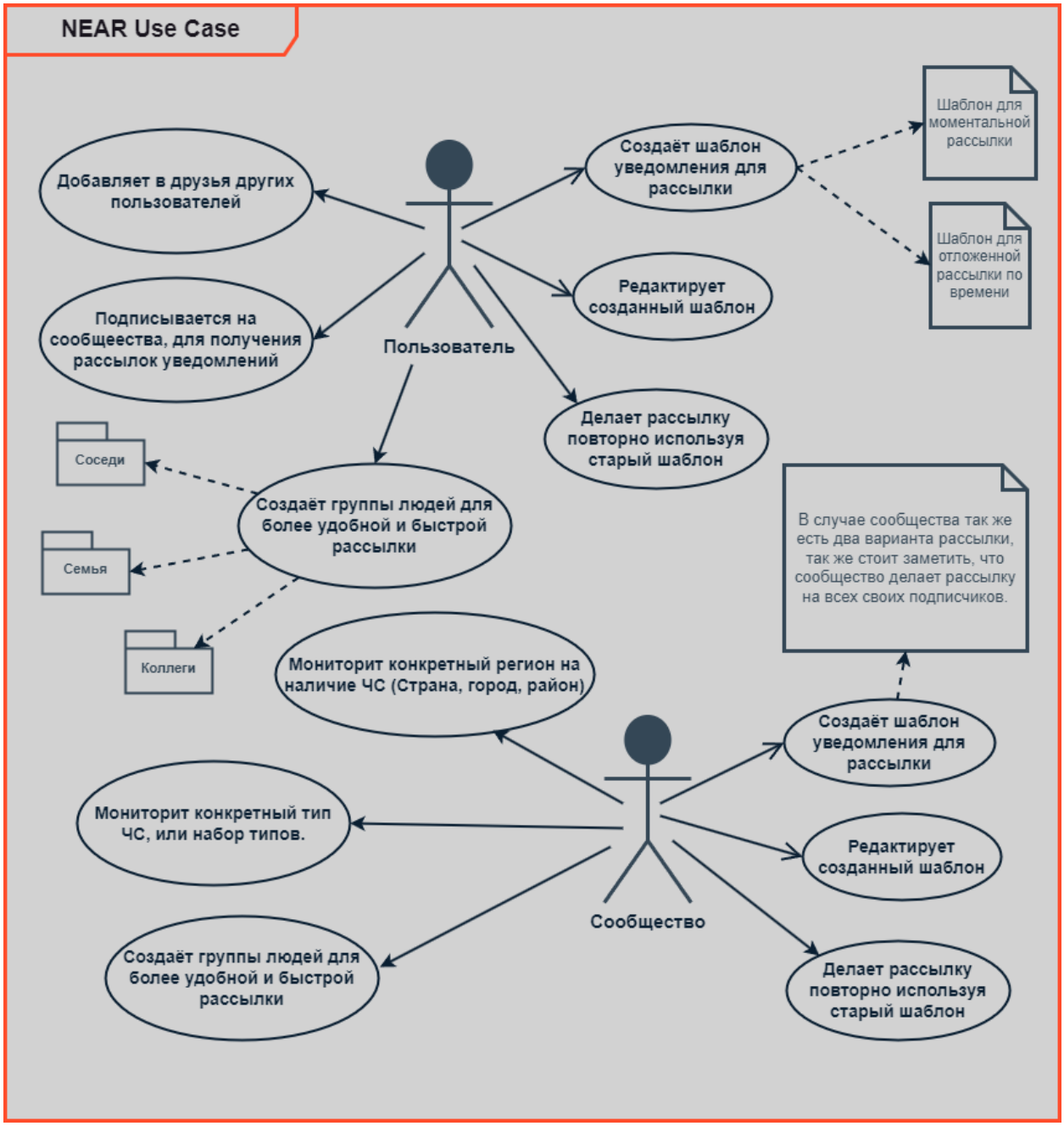

Рисунок А.1 – Варианты использования (Use Case) разрабатываемого использования

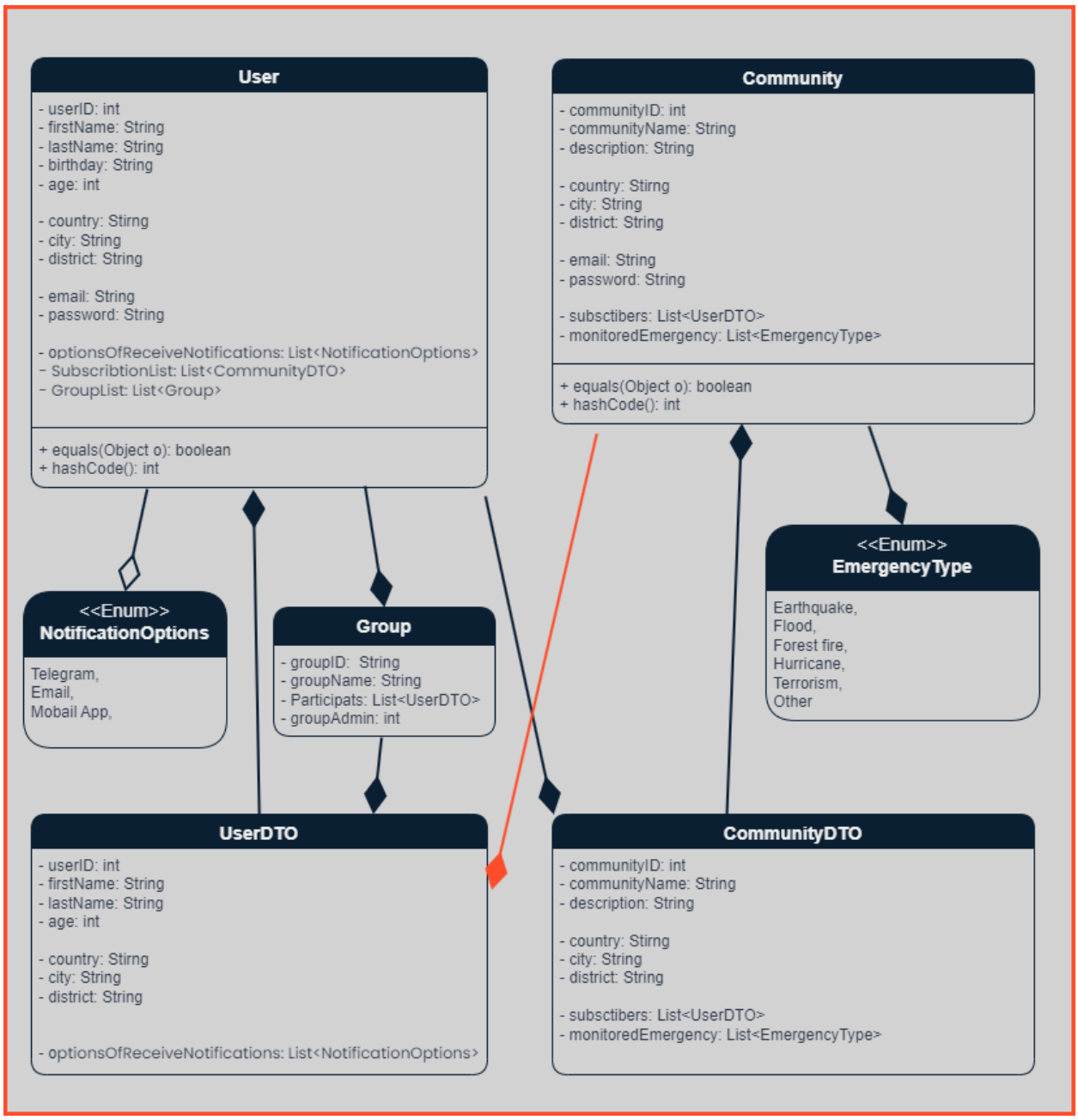

Рисунок А.2 – Основные сущности разрабатываемого приложения Часть 1

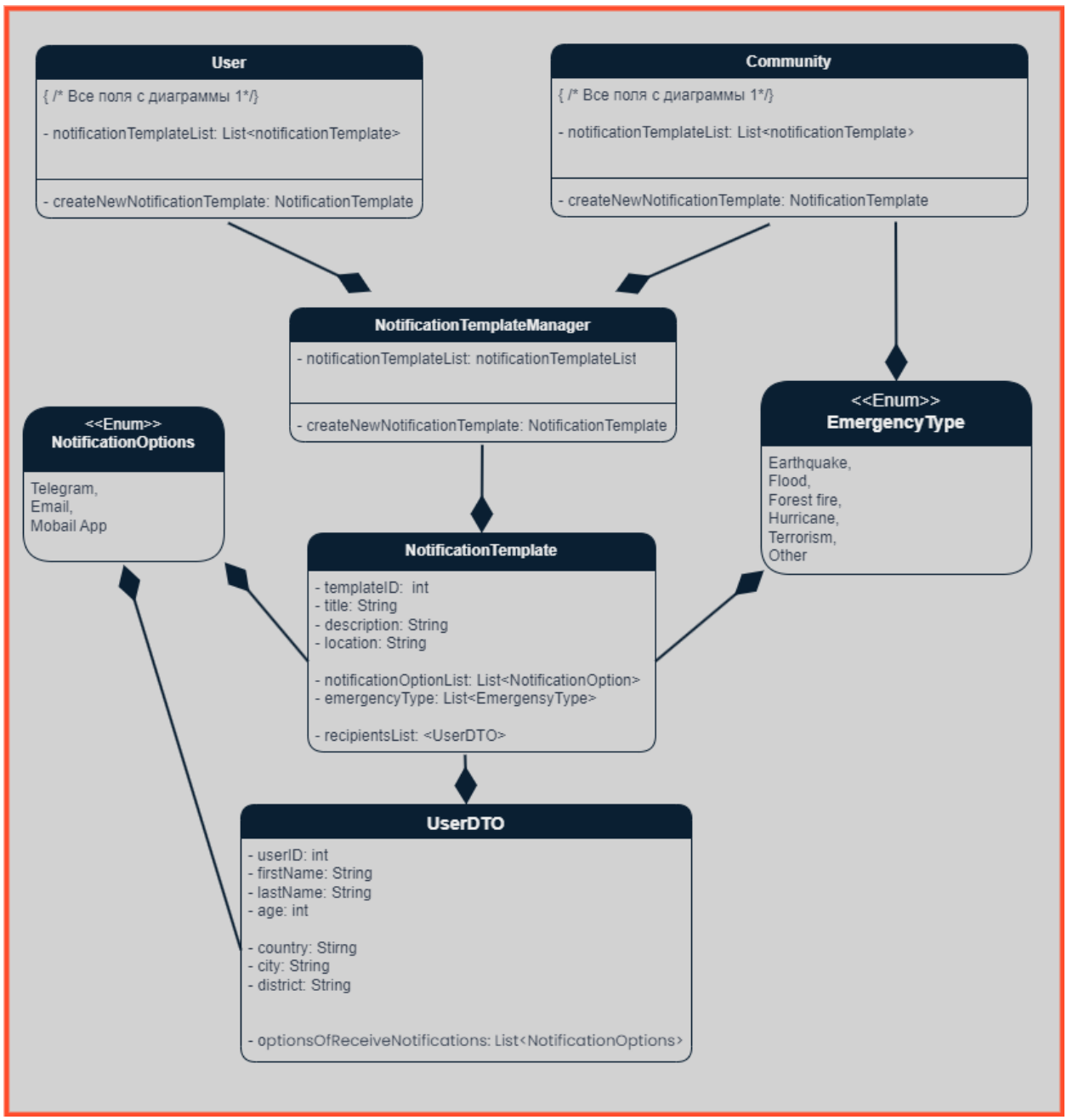

Рисунок А.3 – Основные сущности разрабатываемого приложения Часть 2

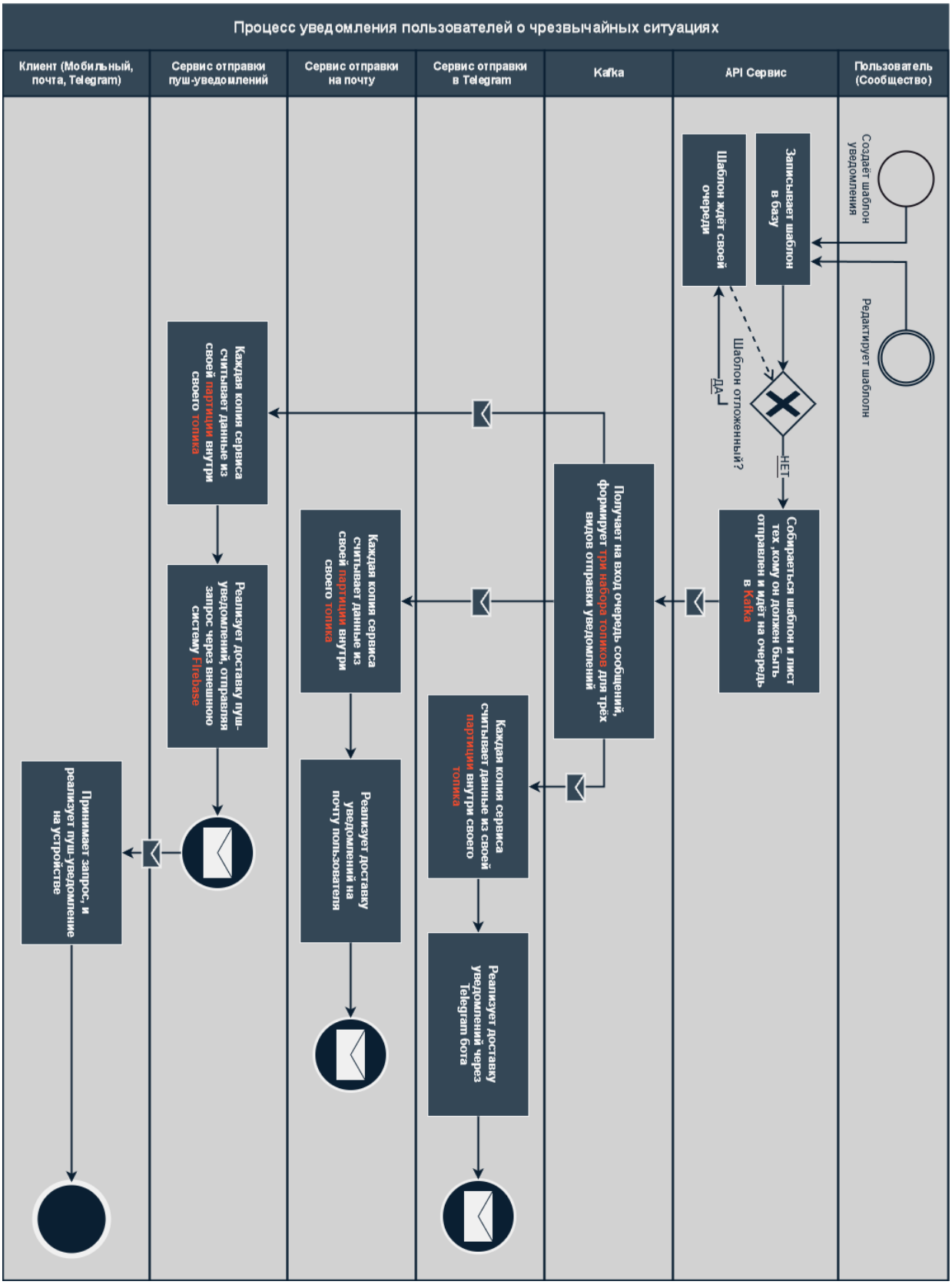

Рисунок А.4 – BPMN-диаграмма или диаграмма процесса

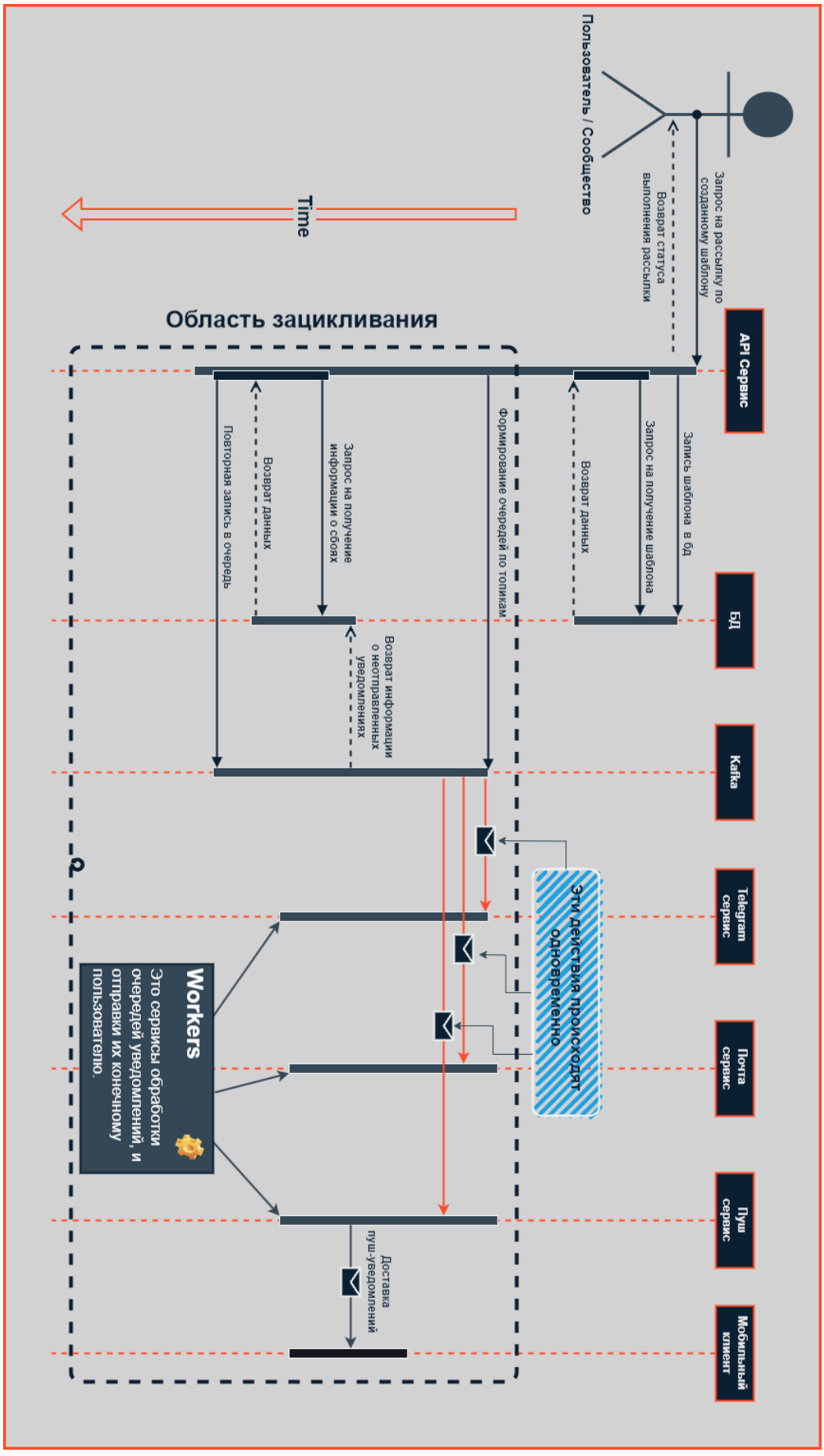

Рисунок А.5 – Диаграмма последовательности (Sequence diagram)

ПРИЛОЖЕНИЕ Б

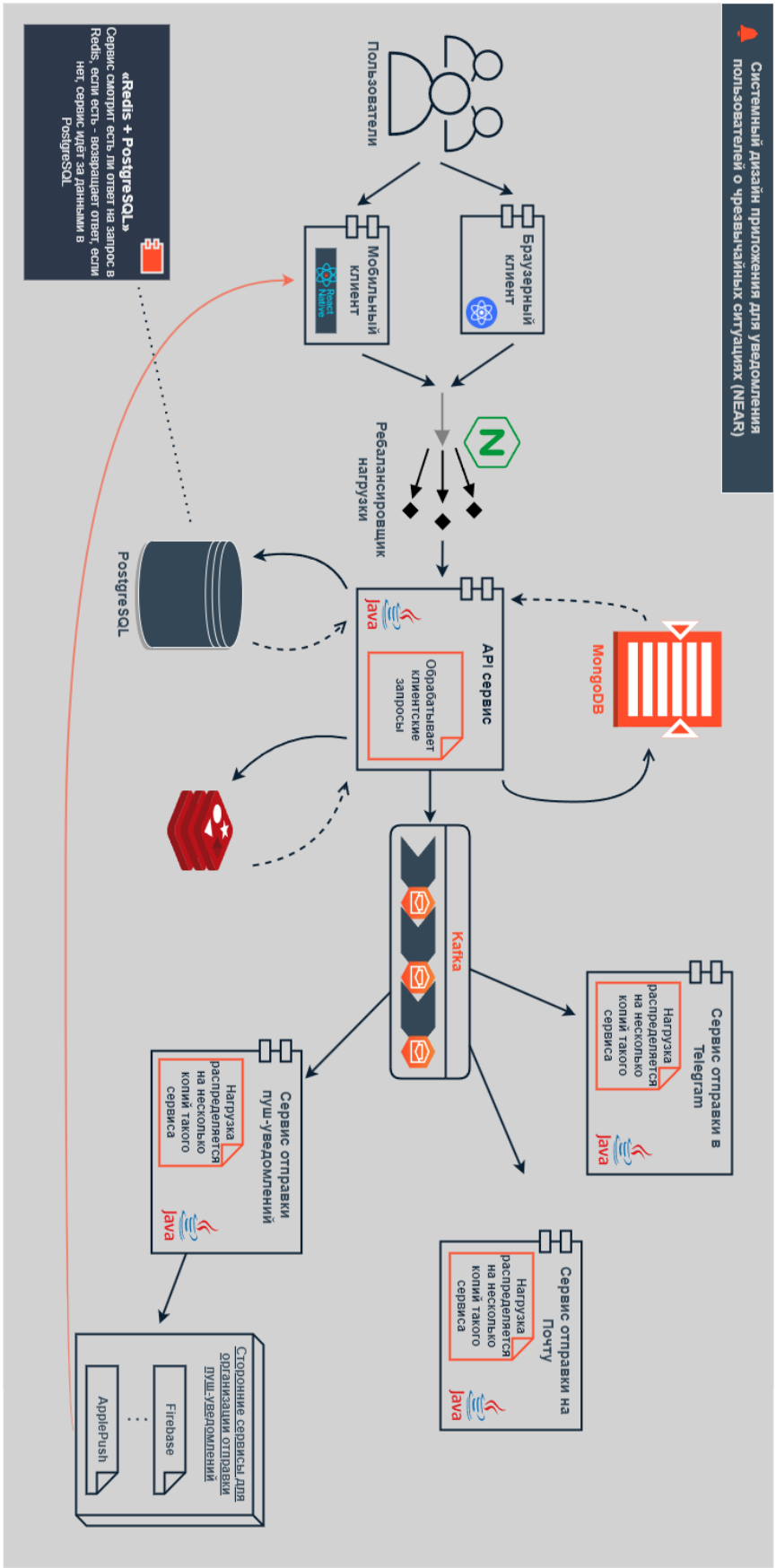

Рисунок Б.1 – Системный дизайн разрабатываемого приложения

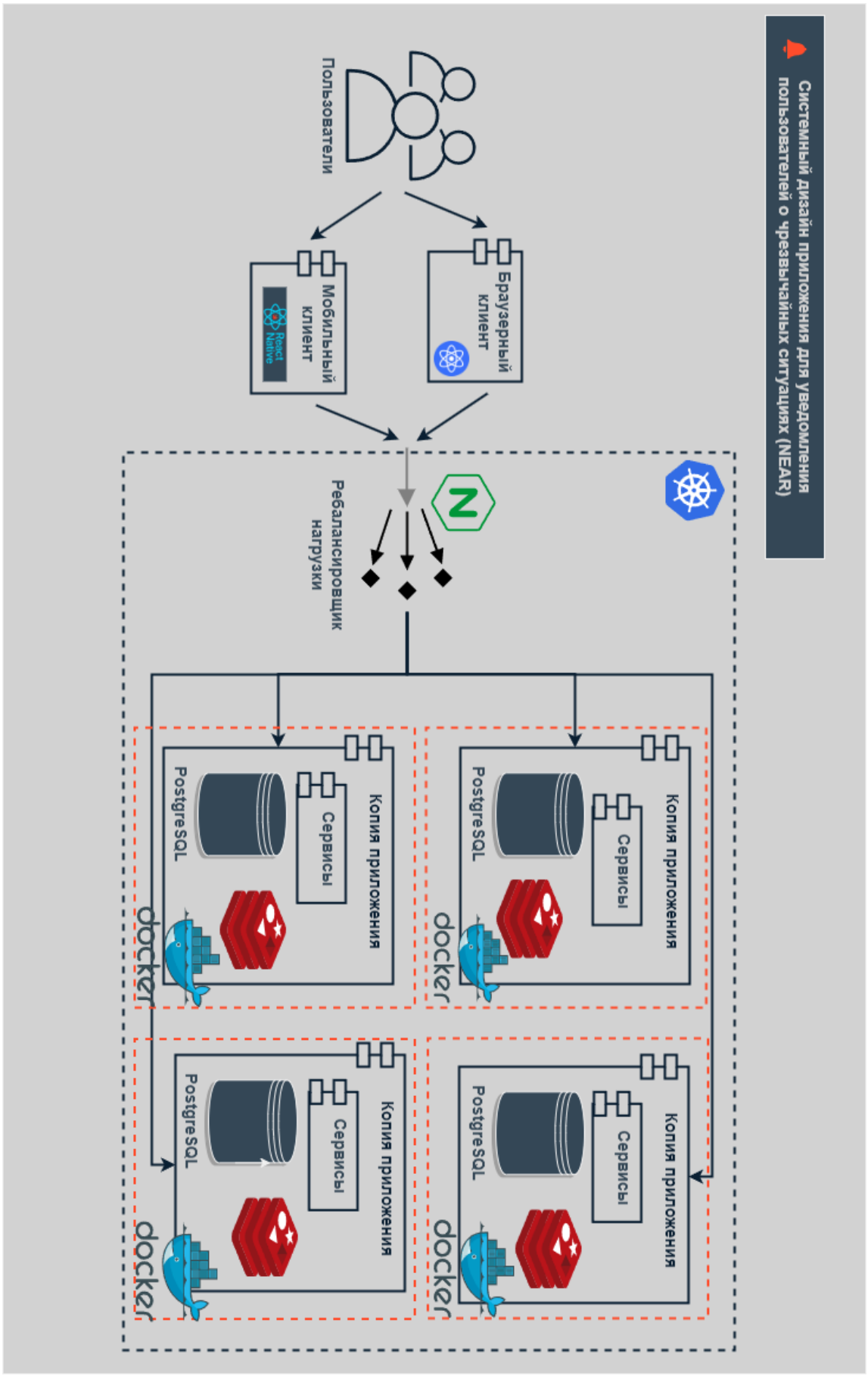

Рисунок Б.2 – Системный дизайн, обёрнутый в контейнеры

ПРИЛОЖЕНИЕ В

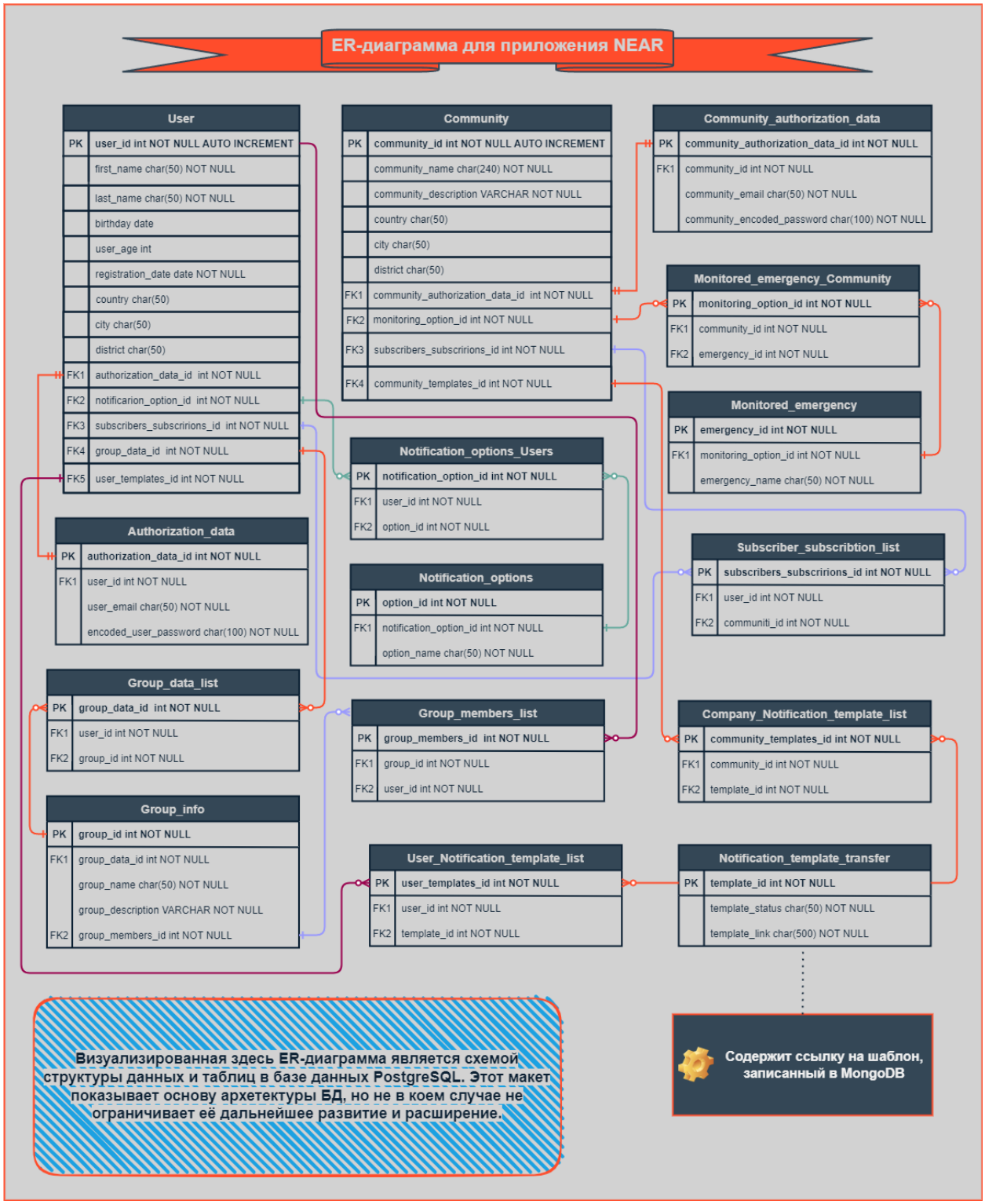

Рисунок В.1 – ER-диаграмма базы данных

ПРИЛОЖЕНИЕ Г

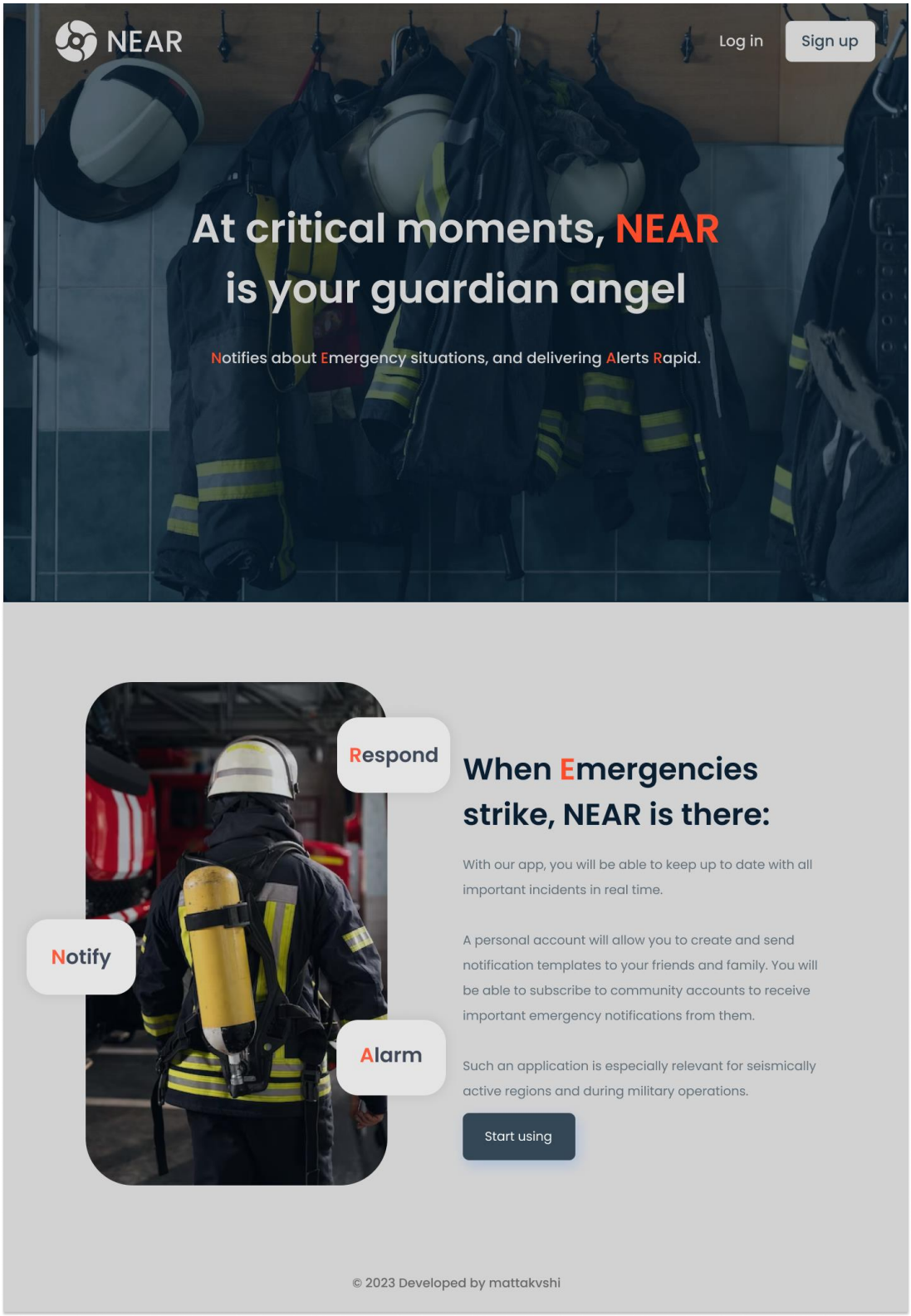

Рисунок Г.1 – Макет лэндинга разрабатываемого приложения

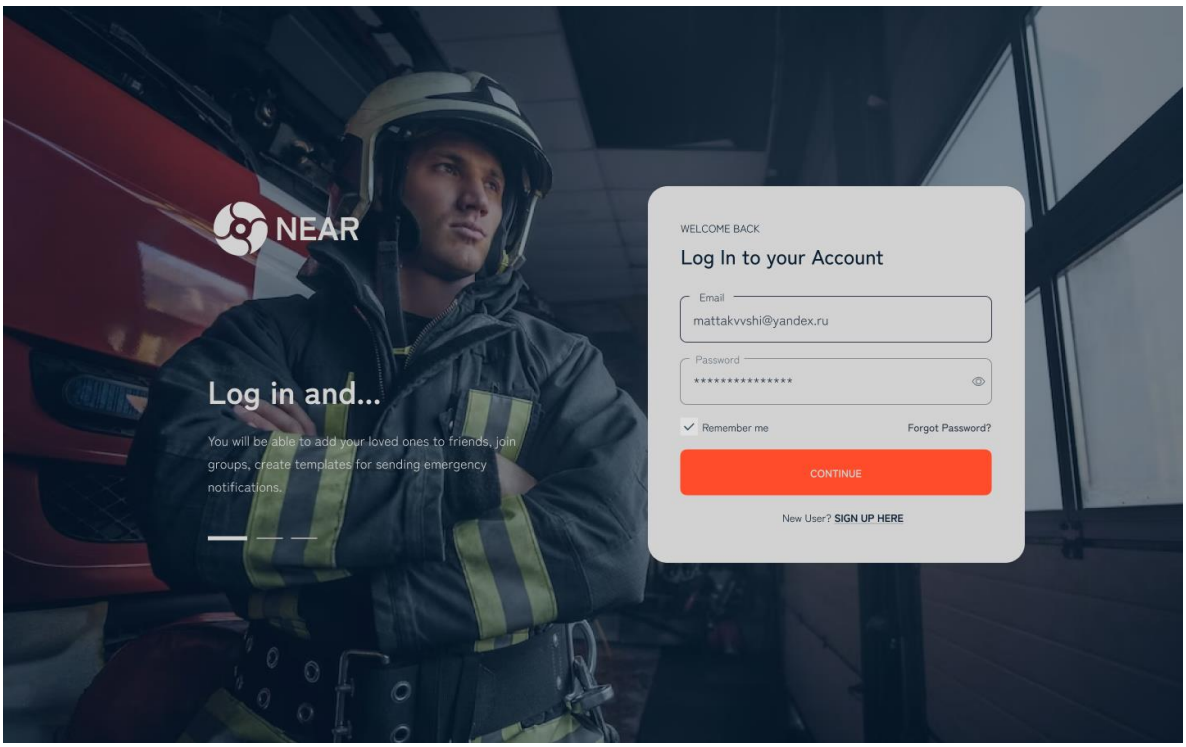

Рисунок Г.2 – Макет страницы авторизации пользователя разрабатываемого приложения

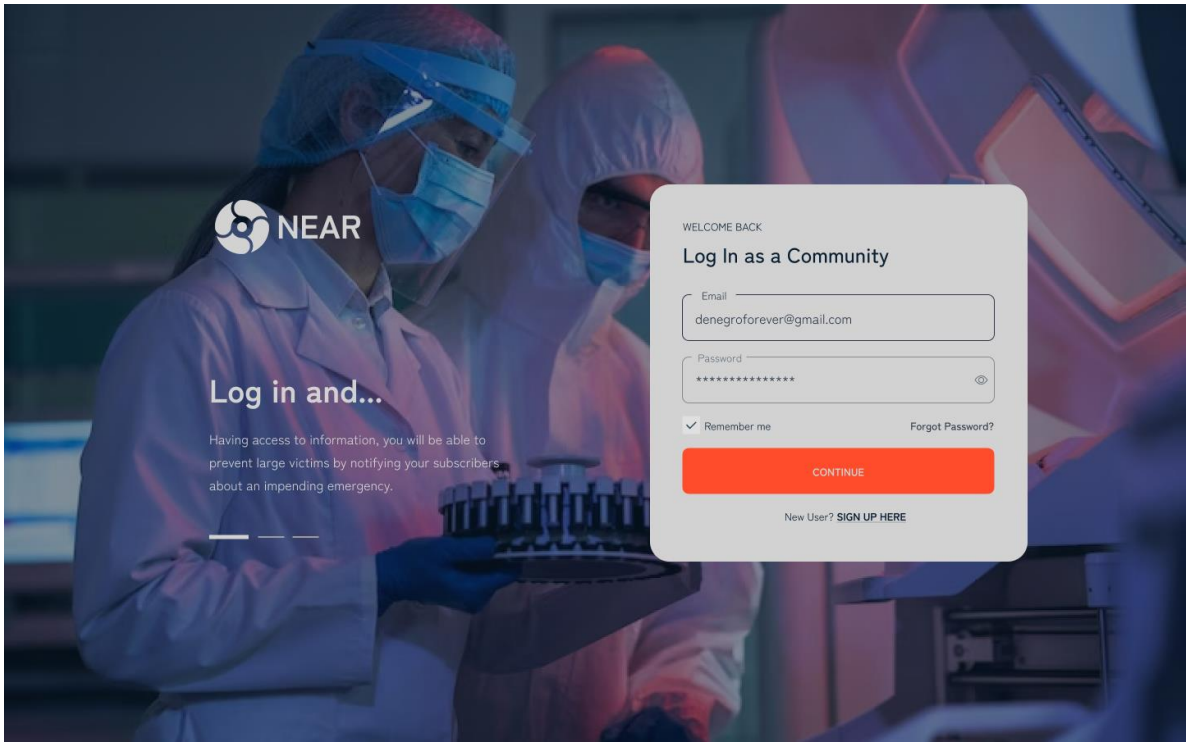

Рисунок Г.3 – Макет страницы авторизации сообщества разрабатываемого приложения

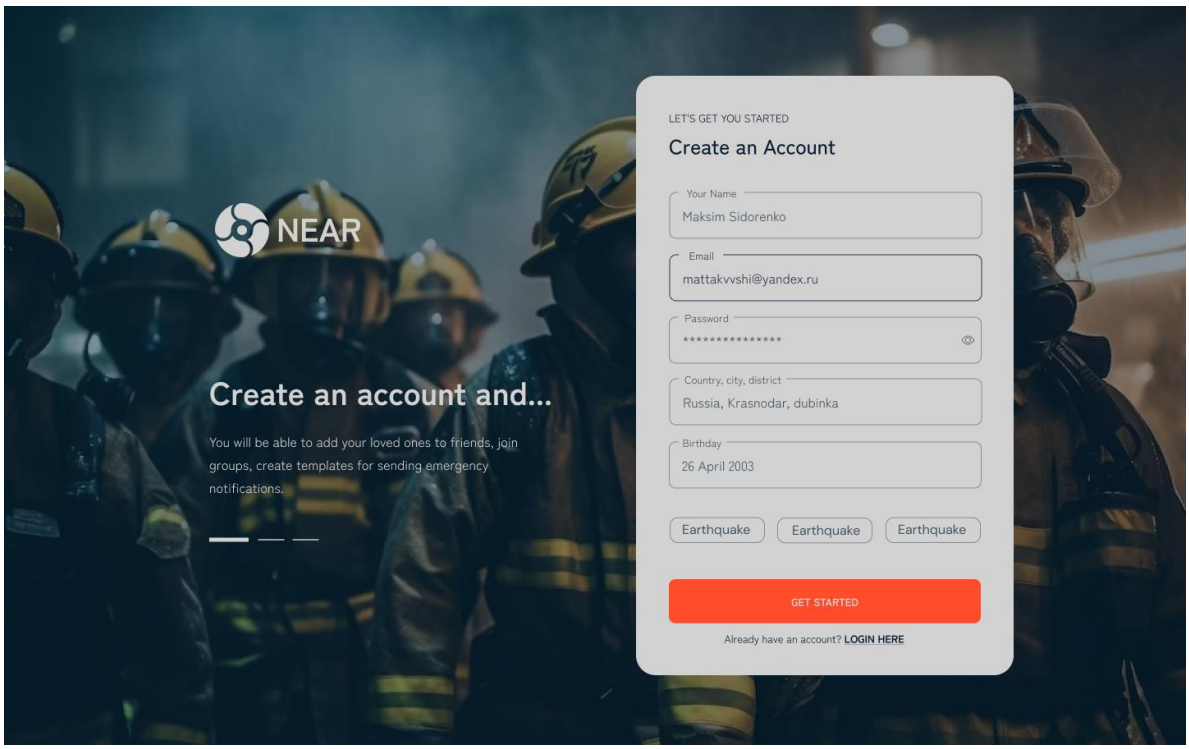

Рисунок Г.4 – Макет страницы регистрации пользователя разрабатываемого приложения

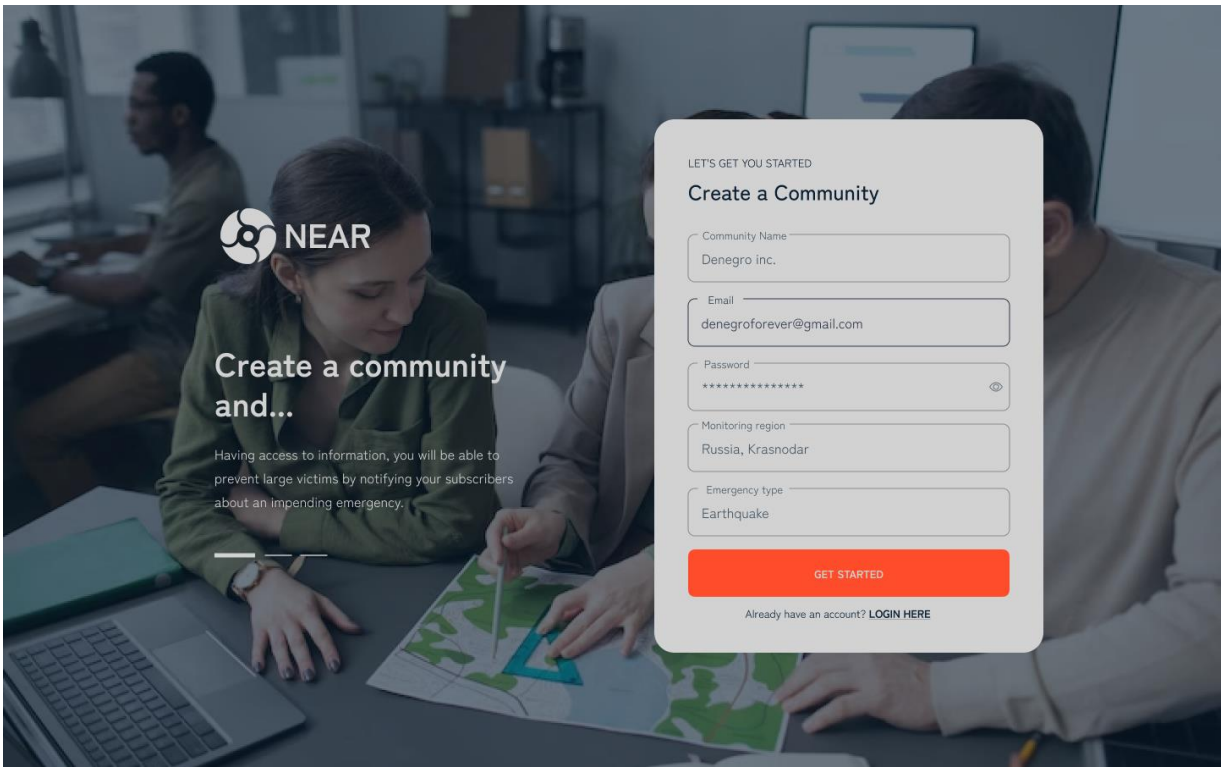

Рисунок Г.5 – Макет страницы регистрации сообщества разрабатываемого приложения

ПРИЛОЖЕНИЕ Д

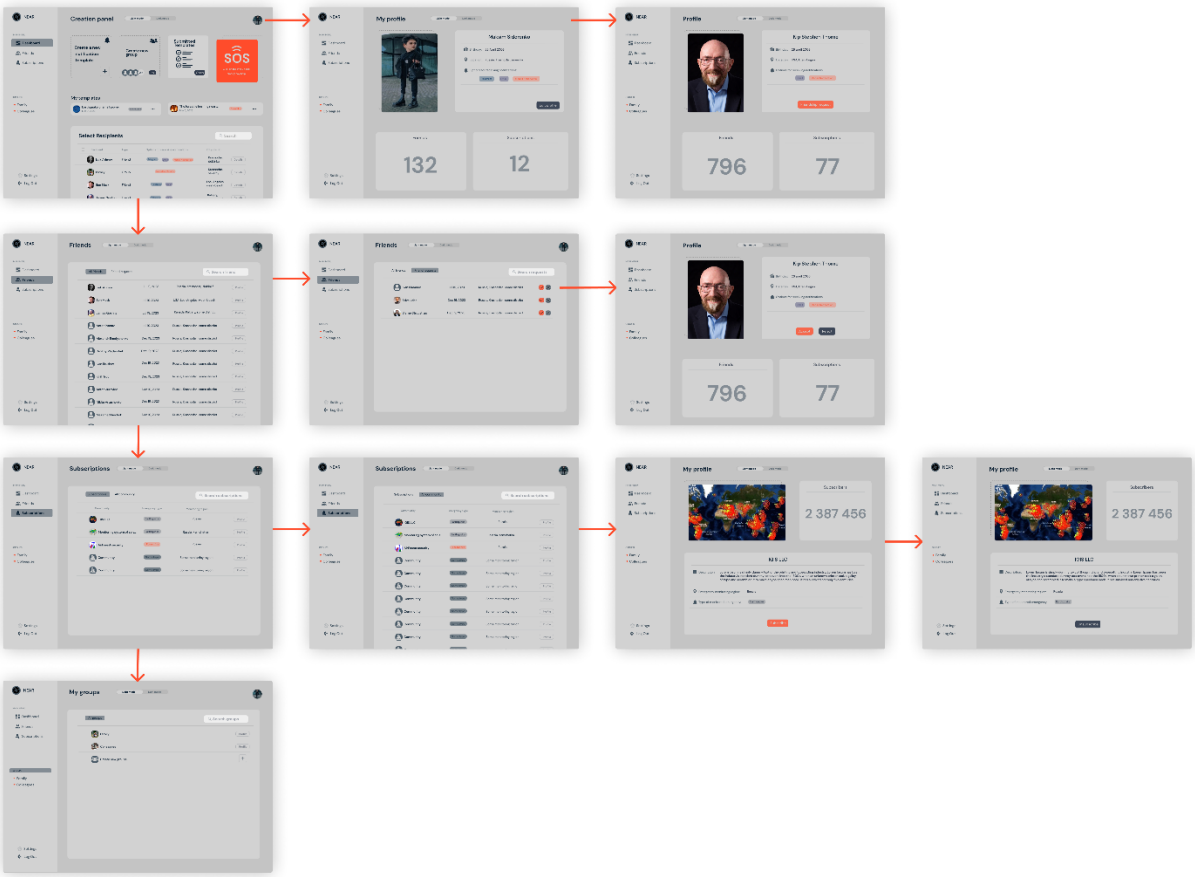

Рисунок Д.1 – Макет экранов пользовательского интерфейса (пользователь)

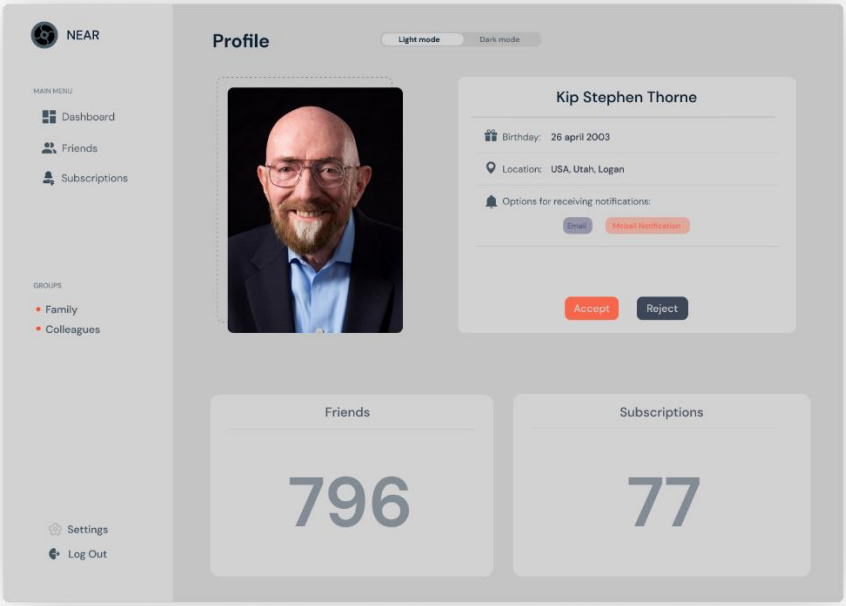

Рисунок Д.2 – Макет профиля странички

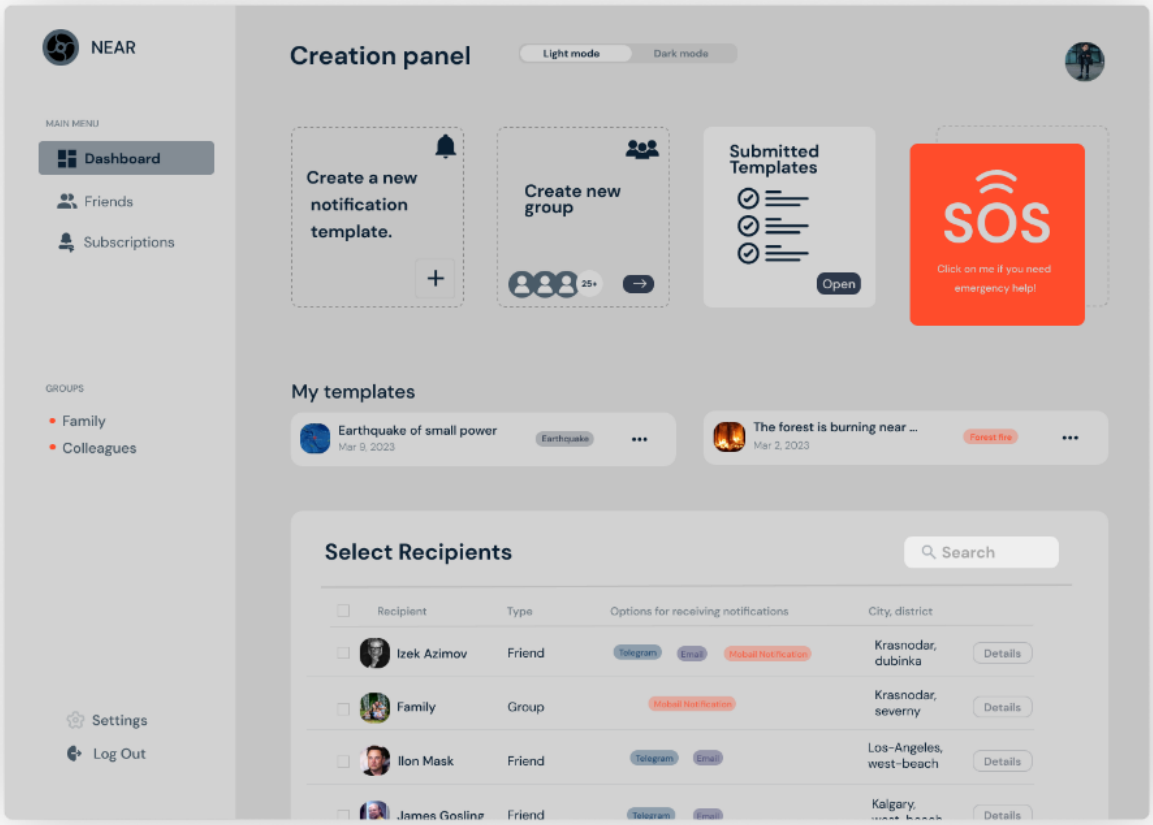

Рисунок Д.3 – Макет главного экрана интерфейса (пользователь)

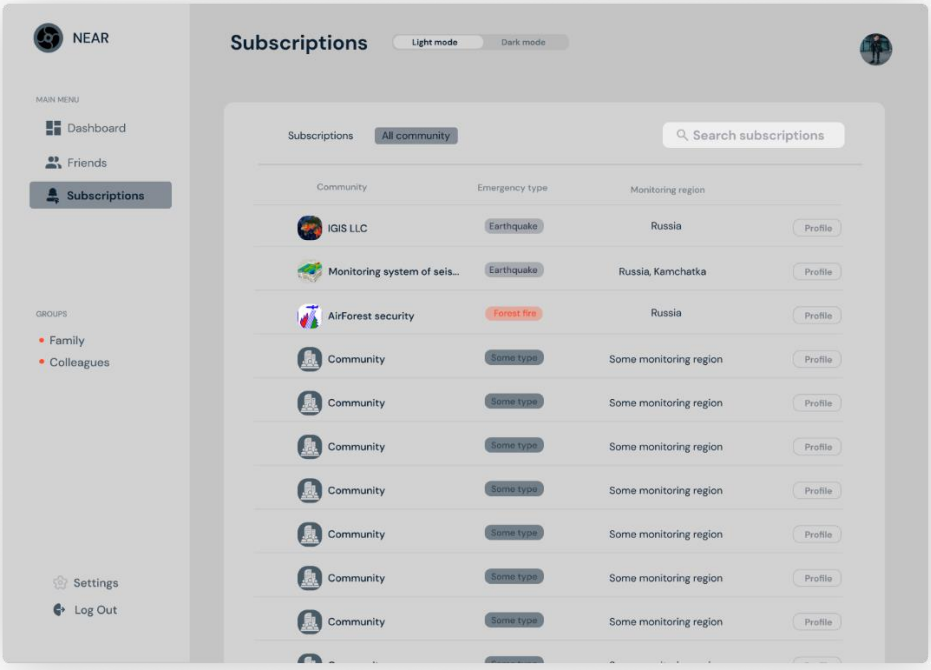

Рисунок Д.4 – Макет экрана со списком сообществ (пользователь)

ПРИЛОЖЕНИЕ Е

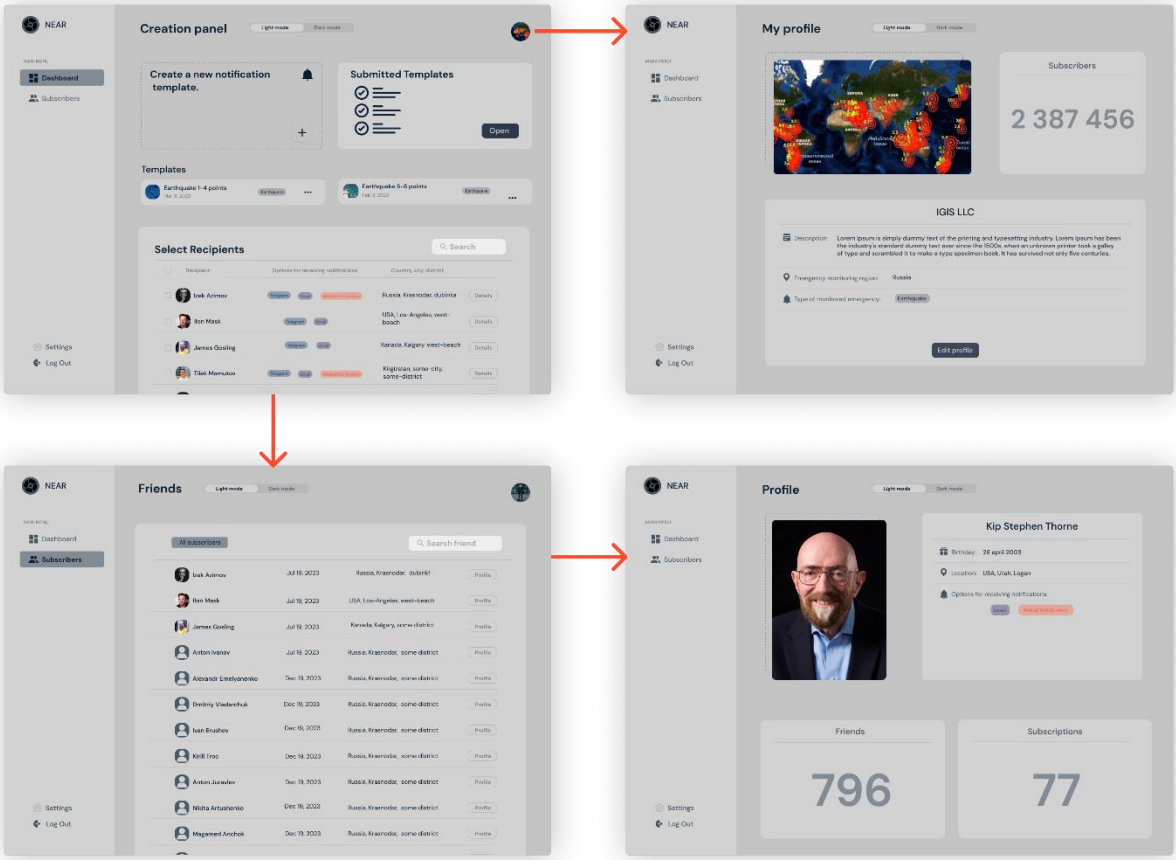

Рисунок Е.1 – Макет экранов пользовательского интерфейса (сообщество)

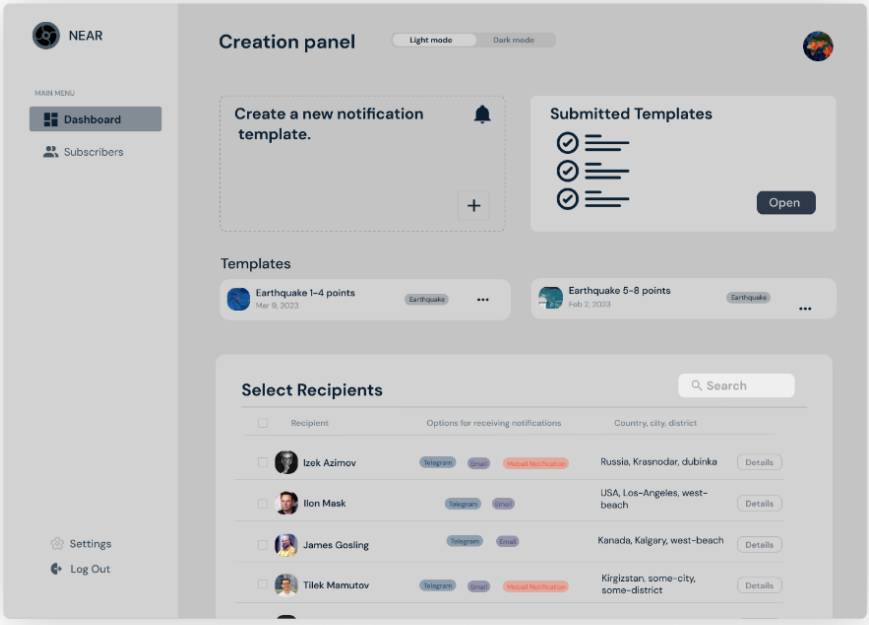

Рисунок Е.2 – Макет главного экрана интерфейса (сообщество)