

**ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»**

Факультет компьютерных наук
Образовательная программа «Программная инженерия»

УДК 004.42

УТВЕРЖДАЮ
Академический руководитель
образовательной программы
«Программная инженерия»,
старший преподаватель департамента
программной инженерии

_____ Н.А. Павлочев
«__» _____ 2025 г.

Выпускная квалификационная работа

на тему: **Информационно-аналитическая система управления
инцидентами в области обеспечения безопасности. Серверная часть:
управление участниками системы, уведомлениями и дежурствами**

по направлению подготовки 09.03.04 «Программная инженерия»

Научный руководитель

_____ Доцент департамента
программной инженерии факультета
компьютерных наук _____
Должность, место работы

_____ Кандидат технических наук _____
ученая степень, ученое звание

_____ С.А. Виденин _____
И.О. Фамилия



_____ 27 апреля 2025 _____
Подпись, Дата

Выполнил

студент группы ____БПИ214____
4 курса бакалавриата
образовательной программы
«Программная инженерия»

_____ Е. К. Фортов _____
И.О. Фамилия



_____ 27 апреля 2025 _____
Подпись, Дата

Москва 2025

Реферат

На момент написания данной работы рынок компаний, требующих особые режимы безопасности, огромен. Каждый год подобные предприятия теряют миллионы рублей из-за недостатка контроля над своими объектами, несвоевременного реагирования на происходящие инциденты, халатности сотрудников. Особенно из-за этого страдают компании по добыче драгоценных металлов, производству легковоспламеняющихся материалов, первичной и вторичной обработке сырья.

Разрабатываемая система призвана решить проблему оперативного реагирования на такие инциденты и взять под контроль такие случаи. Она представляет единую платформу по удобному управлению нештатными ситуациями, позволяя руководству компании автоматизировать процесс их незамедлительного решения, делегировав каждую задачу отдельному сотруднику и контролируя ее выполнение.

Данная работа представляет из себя серверную часть системы, отвечающую за управление пользователями, уведомлениями и дежурствами. Большое внимание в работе уделяется проработке оптимальной архитектуры данной части, что позволит в дальнейшем масштабировать ее, а также обеспечит должную отказоустойчивость и производительность.

Работа содержит 91 страницу, 3 главы, 27 иллюстраций, 36 источников и 40 листингов.

Ключевые слова: инциденты, серверная часть, микросервисная архитектура.

Abstract

At the time of writing this paper, the market of companies requiring special security regimes is huge. Every year such companies lose millions of roubles due to lack of control over their facilities, untimely response to incidents, and negligence of employees. Companies involved in the extraction of precious metals, production of flammable materials, primary and secondary processing of raw materials suffer especially because of this.

The system being developed is designed to solve the problem of prompt response to such incidents and take control of such cases. It presents a single platform for convenient management of abnormal situations, allowing the company management to automate the process of their immediate solution by delegating each task to a separate employee and controlling its fulfilment.

This work represents the server side of the system, responsible for managing users, notifications and duty. A lot of attention is paid to working out the optimal architecture of this part, which will allow for future scalability, as well as ensure proper fault tolerance and performance.

The work contains 91 Pages, 3 chapters, 27 Illustrations, 36 Sources and 40 Listings.

Содержание	
Основные определения, термины и сокращения.....	6
Введение.....	10
Глава 1. Предметная область и обзор существующих решений.....	12
1.1 Описание предметной области и актуальность.....	12
1.2 Обзор рынка систем управления инцидентами в России и за рубежом.....	12
1.2.1 Сервис Jira[3], Atlassian[6].....	13
1.2.2 Сервис Yandex Tracker, Yandex.....	19
1.2.3 Сервис Security Incident Response (SIR[5]), ServiceNow[11].....	20
1.3 Сравнительный анализ рассмотренных аналогов.....	23
1.4 Выводы по главе.....	24
Глава 2. Проектирование системы и технологии разработки.....	25
2.1 Выбор архитектуры системы.....	25
2.1.1 Монолитная архитектура.....	25
2.1.2 Микросервисная архитектура.....	26
2.1.3 Сравнительный анализ серверных архитектур.....	27
2.2 Описание архитектуры системы.....	28
2.3 Пользовательские сценарии.....	30
2.4 Описание общей архитектуры сервисов.....	30
2.5 Выбор технологий реализации.....	33
2.5.1 Языки программирования.....	33
2.5.2 Хранилища данных.....	34
2.5.2.1 PostgreSQL[26].....	34
2.5.3 Межсервисное взаимодействие в системе.....	35
2.5.4 Другие используемые технологии.....	36
2.5.4.1 ORM Hibernate[29].....	36
2.5.4.2 Фреймворки для тестирования JUnit[27] и Mockito[28].....	37
2.5.4.3 Docker[18].....	37
2.5.4.4 Docker-compose[19].....	38
2.6 Выводы по главе.....	41
Глава 3. Программная реализация системы.....	43
3.1 Сервис профилей пользователей.....	43
3.1.1 Особенности реализации сервиса.....	43
3.1.2 Функционал сервиса.....	43
3.1.3 Бизнес-логика.....	48

3.1.4 Взаимодействие с базой данных.....	50
3.1.5 Тестирование.....	51
3.2 Сервис дежурств.....	52
3.2.1 Особенности реализации сервиса.....	52
3.2.2 Функционал сервиса.....	52
3.2.3 Бизнес-логика.....	55
3.2.4 Взаимодействие с базой данных.....	56
3.2.5 Тестирование.....	58
3.3 Сервис уведомлений.....	59
3.3.1 Особенности реализации сервиса.....	59
3.3.2 Функционал сервиса.....	64
3.3.3 Бизнес-логика.....	66
3.3.4 Взаимодействие с базой данных.....	69
3.3.5 Тестирование.....	71
3.4 Библиотека Common.....	72
3.5 Система сборки.....	72
3.6 Документация.....	75
3.7 Платформа разработки.....	75
3.8 Выводы по главе.....	75
Заключение.....	77
Список использованных источников.....	78
ПРИЛОЖЕНИЕ.....	81
REST API спецификация для разрабатываемых сервисов:.....	81

Основные определения, термины и сокращения

1. API (англ. Application Programming Interface) – язык, на котором приложения общаются между собой.
2. JSON (JavaScript Object Notation) – текстовый формат обмена данными, основанный на языке программирования JavaScript. Но при этом формат независим от JavaScript и может использоваться в любом языке программирования.
3. XML (eXtensible Markup Language) — это язык разметки, разработанный для хранения и передачи структурированных данных.
4. JWT (англ. Json Web Token) – это открытый стандарт для создания токенов доступа, основанный на формате JSON.
5. REST API (англ. Representational State Transfer API) – архитектурный подход, который устанавливает ограничения для API: как они должны быть устроены, и какие функции они должны поддерживать.
6. SMTP (англ. Simple Mail Transfer Protocol) – протокол связи, используемый для отправки и получения сообщений электронной почты через Интернет.
7. БД — база данных.
8. СУБД – система управления базами данных.
9. Микросервис — это небольшой атомарный самостоятельный сервис, который отвечает за одну функциональную роль в системе.
10. Тикет — задача в системе планирования задач
11. Трекер — система отслеживания
12. Кастомизация — это создание чего-то особенного, своего, в каком-то роде уникального.
13. Кастомизированный — подвергшийся процессу кастомизации.
14. Контейнер – это модуль, в котором запускается одно приложение. Контейнеры занимают меньше памяти, используют небольшое количество ресурсов и почти не зависят от операционной системы кластера.
15. Бекенд — часть системы, отвечающая за логику работы системы.

16. Прокси (прокси-слой) – промежуточный сервер между клиентом и сервером.
17. Развертывание приложения – процесс выполнения необходимых шагов, чтобы сделать приложение доступным для пользователей.
18. Токен авторизации – специальная сущность, разрешающая доступ к данным конкретного пользователя.
19. Спринт — это небольшой отрывок времени (обычно 2 недели), за который команда разработки должна выполнить определенные задачи.
20. PR (pull request) — это запрос на вливание сделанных изменений в общую ветку проекта. Эти изменения должны быть одобрены другими разработчиками.
21. Вендорозамещение — это процесс замещения импортного ПО российскими аналогами.
22. Agile — это гибкая методология разработки ПО.
23. Kanban-доска — это визуальный инструмент, используемый для управления рабочими процессами и задачами, особенно в сфере проектного управления и Agile-методологиях.
24. MVP (minimal viable product) — это «минимально жизнеспособный продукт» - продукт первой версии, содержащий только основные функции, на которые создатели делают ставку.
25. Экземпляр программы — это одна запущенная программа.
26. Инстанс программы — то же самое, что экземпляр.
27. Парсинг — процесс разбора данных из закодированного / зашифрованного состояния.
28. Эндпоинт API — это конкретный адрес (URL), по которому клиент (например, веб-приложение, мобильное приложение или другой сервис) может взаимодействовать с сервером, использующим API.
29. HTTP(s) — сетевой протокол взаимодействия между компьютерами в сети.
30. HTML (HyperText Markup Language) — это язык разметки, используемый для создания и структурирования контента в веб-документах.
31. Маппер — класс, преобразующий сущность одного типа в сущность другого типа.

- 32. UI — графический интерфейс, а также клиентская часть системы (то, что отображается в браузере).
- 33. RESTful API — то же, что REST API.
- 34. Логирование — (процесс) добавления/-е информационного сообщения о каком-либо событии / совершенном действии в сервисе.
- 35. Логи — файлы, в которые пишутся информационные сообщения о сервисе.
- 36. Инкапсулировать — объединять, скрывая детали (реализации).
- 37. Рутинный — повседневный и легко выполняемый.
- 38. Валидация — проверка на корректность по какому-либо критерию.
- 39. Процессинг — обработка.
- 40. Препроцессинг — предварительная обработка.
- 41. Невалидный — непрошедший успешно процесс валидации.
- 42. Имплементация — реализация.
- 43. Индексироваться — проходить процесс помещения того, что индексируется, под версионный контроль системы контроля версий (например, git).
- 44. Сервлеты — это компоненты, которые первично обрабатывают входящие HTTP-запросы, а также отвечают за непосредственный HTTP-ответ в веб-приложении.
- 45. Null — специальный тип данных, обозначающий отсутствие значения.
- 46. Конечный класс — это класс, содержащий реализацию (не абстрактный, не шаблонный).
- 47. Самописный — в данном документе то же, что и проприетарный — написанный самостоятельно.
- 48. Контроллер — класс, являющийся входной точкой HTTP-запросов в приложение. Представляет собой обертку над механизмом сервлетов (см. п. 44).
- 49. Транзитивно — неявно, через несколько узлов. Например, если А зависит от В, а В зависит от С, то С транзитивно зависит от А.
- 50. Транзитивная зависимость — это зависимость, от которой данный объект зависит транзитивно (см. пред. пункт).
- 51. Разворачивание сервиса — это запуск сервиса в какой-либо среде.
- 52. Скрипт — файл с кодом, как правило, настраивающим что-то. Например, сборочный скрипт — это файл с инструкциями по разворачиванию сервиса.

53. Бизнес-логика приложения — это основная логика работы приложения, включающая алгоритмы по обработке поступивших данных.
54. Native SQL — язык запросов SQL общего формата, неспециализированный для конкретной СУБД.

Введение

Прибыль — один из ключевых показателей каждого коммерческого предприятия. Ее размер влияет на дальнейшее развитие компании, финансовое положение ее сотрудников, степень заинтересованности инвесторов, а также многие другие важные вещи. Эту величину можно повышать различными способами. Например, можно создавать рекламные кампании своего товара для выхода на новые рынки сбыта, привлекать дополнительные инвестиции для открытия новых направлений производства, повышать уровень интеллектуального капитала фирмы с целью производства более качественной продукции и, как следствие, захвата бОльшей доли рынка. Но все перечисленные действия направлены на использование новых ресурсов, зачастую требующих существенных финансовых затрат, которые компания не может позволить себе на текущий момент.

Одно из возможных решений — оптимизация расходов. Замечено, что производственные предприятия теряют большую часть денег из-за недостатка контроля над своими объектами, несвоевременного реагирования на происходящие инциденты, неспособности сотрудников согласованно устранить произошедшее. Все эти потребности может закрыть разрабатываемая система управления инцидентами, которая будет предоставлять обширный функционал по поэтапному реагированию на возникшую внештатную ситуацию с назначением ответственных лиц за каждый из этапов. Также система будет уметь принимать сигналы о таких ситуациях — например, от датчиков возгорания или задымления.

Роль моей работы в данном проекте — привнести в приложение часть серверного функционала, связанного с управлением пользователями приложения, их профилями, управлением уведомлениями по назначаемым работам, а также управлением дежурствами в тех или иных циклах реагирования на инцидент.

Таким образом, цель данной работы - разработать независимую систему, предоставляющую описанный выше функционал.

Для достижения цели были поставлены следующие задачи:

- Спроектировать архитектуру описанной серверной части, а также архитектуру отдельно разрабатываемых сервисов, обеспечивающую их должную отказоустойчивость и масштабируемость.
- Проанализировать и выбрать наиболее подходящие с точки зрения требований архитектурные решения: сервисы, базы данных, взаимодействие между элементами инфраструктуры.

- Описать архитектуру системы.
- Спроектировать и разработать сервис пользователей системы, который будет предоставлять стандартные операции по управлению профилем участника системы.
- Спроектировать и разработать сервис уведомлений, который будет предоставлять возможность для создания и назначения нотификаций пользователям, ответственным за ту или иную задачу в процессе решения инцидента.
- Спроектировать и разработать сервис дежурств, который будет позволять назначать ответственных за выполнения работ по устранению возникшей внештатной ситуации и позволит контролировать выполнение обязанностей каждого участника дежурства.
- Предоставить API для взаимодействия другими частями системы с описанными выше сервисами.
- Обеспечить работоспособность системы на независимом контуре.

В главе 1 приведен обзор и анализ смежных решений и аналогов разрабатываемого продукта.

В главе 2 рассматривается архитектура разрабатываемой части системы и отдельных ее сервисов.

В главе 3 приводится описание и демонстрация результатов работы.

В заключении приведены выводы по результатам работы и описан план дальнейшего развития проекта.

Глава 1. Предметная область и обзор существующих решений

1.1 Описание предметной области и актуальность

Основная предметная область разрабатываемой системы – платформы для управления проектами, которые предоставляют функционал по созданию различного рода задач, назначению их пользователям управлению задачами в соответствии со статусом их выполнения. Подобные системы также предоставляют возможности отслеживания выполнения задач, интеграции с другими сервисами, создание отдельных пространств задач и многие другие функции.

Данное направление разработки очень перспективно и пользуется большим спросом в наши дни, потому что позволяет осуществлять контроль за выполняемыми сотрудниками задачами, успешное завершение которых напрямую сказывается на финансовых показателях компании. Особенно это актуально на производственных предприятиях, современные объекты инфраструктуры которых требуют быстрого и эффективного реагирования на чрезвычайные ситуации (пожар, задымление, затопление), так как задержки в подобного рода случаях могут привести к значительным убыткам, утрате имущества и даже угрозам безопасности жизни людей.

Традиционные системы мониторинга и реагирования часто зависят от человеческого фактора, что может привести к медленной обработке событий и ошибкам по устранению нештатных ситуаций. Автоматизация событий безопасности с интеграцией датчиков и бизнес-процессов является необходимым решением для повышения скорости реагирования на инциденты, что позволит минимизировать ущерб и значительно сократить риски.

1.2 Обзор рынка систем управления инцидентами в России и за рубежом

Активная разработка подобных систем началась еще со времени создания крупных IT-предприятий, когда по мере роста руководству компании сложно было делегировать задачи своим сотрудникам и отслеживать их выполнение в режиме реального времени. Сначала это были проприетарные решения для каждой отдельной компании, позже эти решения стали продаваться на внешний рынок (например, Microsoft Project[1], BugZilla[2]). Позже появились более популярные Jira[3] и Яндекс Трекер[4], однако они не предоставляют интеграцию с внешними датчиками реагирования на особые условия и не позволяют настроить гибкую систему уведомлений для более точного контроля исполнения сотрудником назначенной ему задачи в рамках произошедшего инцидента.

Другой класс систем позволяет получать уведомления о некотором событии от внешних устройств, однако не предоставляет возможности для создания процесса

реагирования на возникшую ситуацию и тем более процесса контроля над своевременной обработкой этого инцидента. В таком классе программ наиболее известен Security Incident Response[5], который стал популярен благодаря возможности работы с целым рядом устройств.

Далее будут приведен сравнительный анализ разрабатываемой системы по сравнению с наиболее к ней приближенными, описанными выше аналогами.

1.2.1 Сервис Jira[3], Atlassian[6]

Данная система управления проектами сегодня является одной из самых популярных в России и по всему миру. Она была разработана австралийской компанией Atlassian[6] в 2002 году и за время своего существования показала себя как одно из самых оптимальных решений в области управления рабочими задачами благодаря богатому функционалу, надежности и удобства пользовательского интерфейса.

Jira[3] представляет пользователю концепцию проектов задач. Проект — это пространство для задач, в состав которого могут входить несколько участников. Каждое такое пространство имеет своего владельца, который обладает административными правами на этот проект и может:

- добавлять / удалять пользователей из него
- назначать им новые роли, задачи
- отслеживать прогресс их выполнения
- писать комментарии к ним
- переназначать на других пользователей
- и многое другое.

Все свои задачи пользователь видит на центральной доске, которая отображает их в различных колонках в соответствии с их статусом.

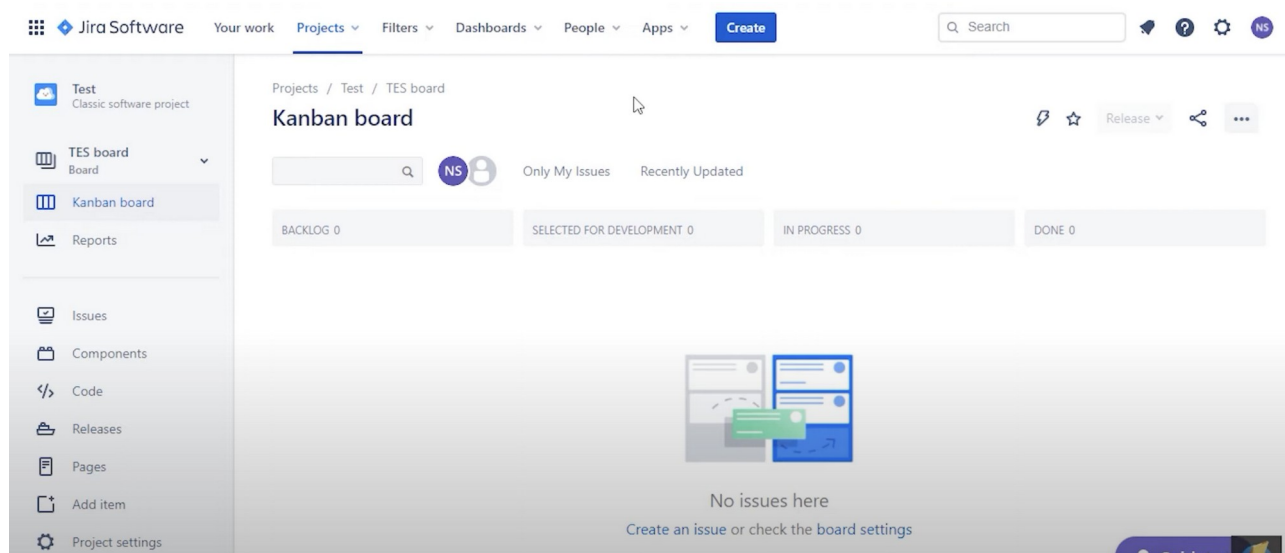


Рисунок 1. Главная страница сервиса Jira[3], доска задач.

Также Jira[3] представляет высокую степень кастомизации своих проектов. Например, сервис позволяет настраивать и управлять рабочими процессами (workflows) в соответствии с потребностями команды, включая различные статусы и этапы задач.

Пользователь имеет возможность:

- получать задачу
- изменять ее статус в соответствии с прогрессом
- писать комментарии к задаче
- привязывать задачу к определенному pull request-у в системе контроля версий
- переназначать задачу на другого пользователя
- создавать свои задачи
- и многое другое

Для некоторых из действий он должен получить соответствующие права от администратора проекта в сервисе.

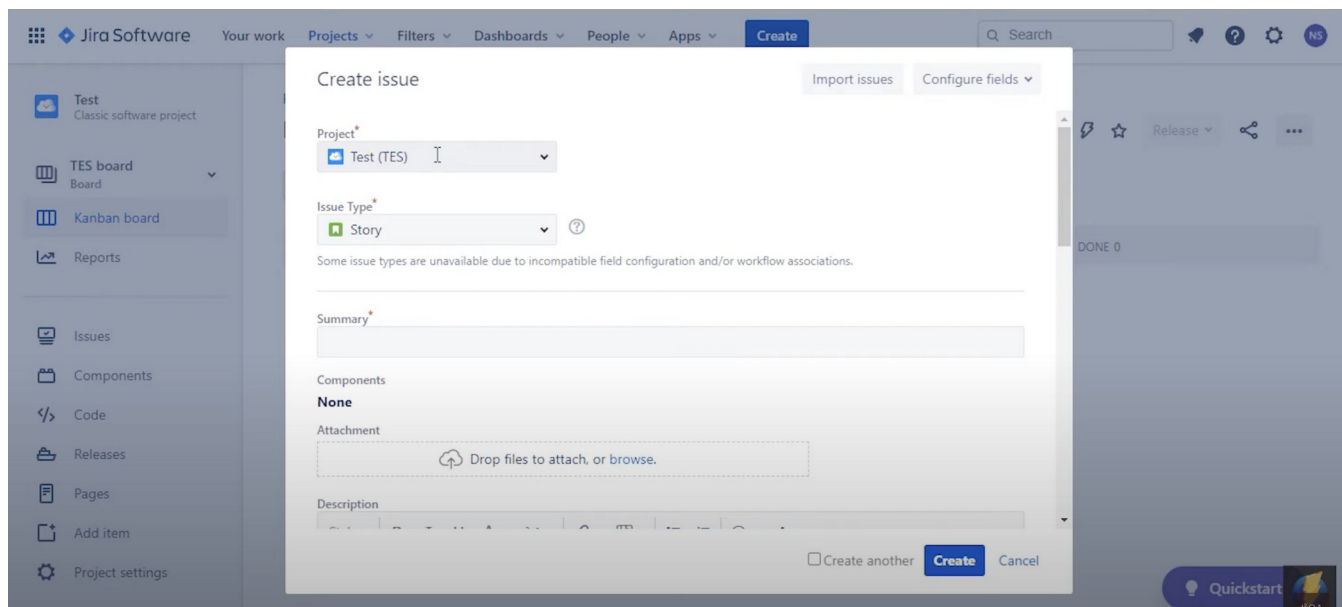


Рисунок 2. Создание задачи.

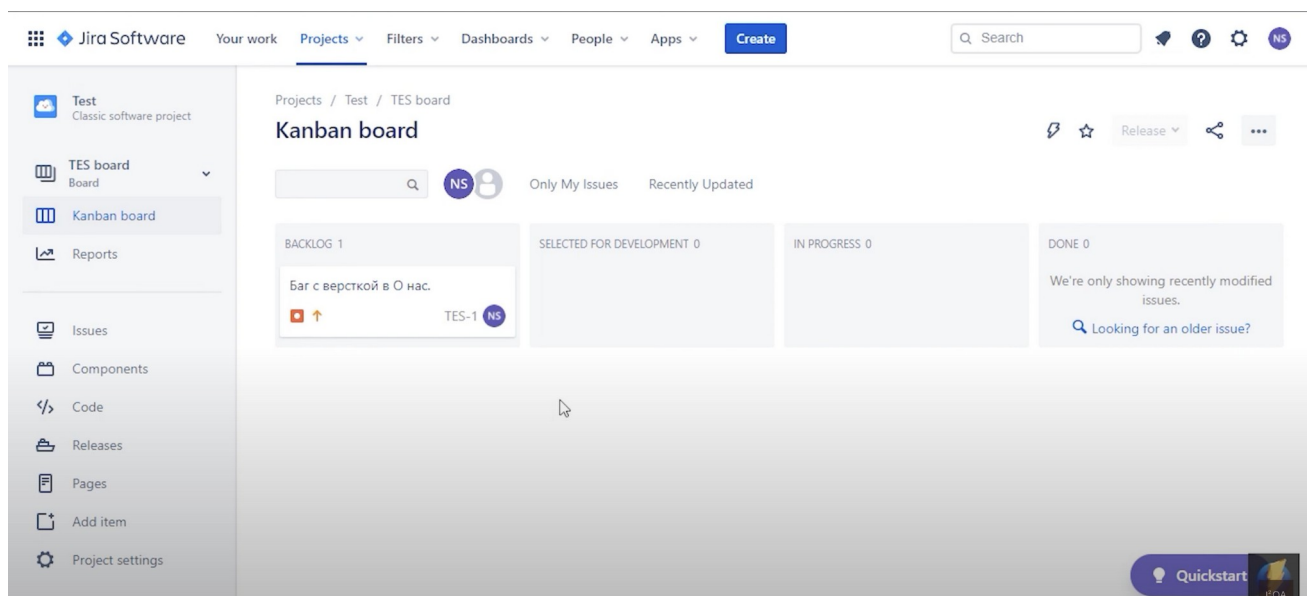


Рисунок 3. Доска с задачами пользователя. 1 созданная задача в статусе «Backlog».

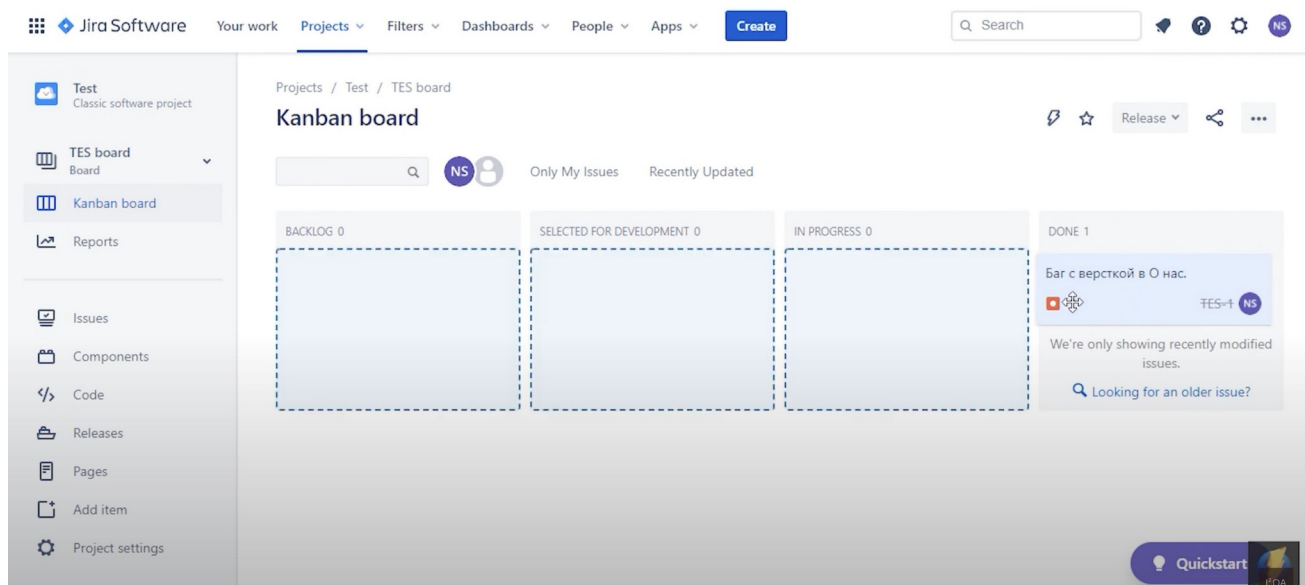


Рисунок 4. Перемещение задачи в статус «Done».

Jira[3] также ориентирована на команды, работающие по методологии Agile, которая включает в себя выполнение проекта короткими этапами, называемыми спринтами. Спринт в среднем идет 2 недели и нужен для выполнения какой-то более-менее весомой части всего проекта. Например, создание интеграции со смежным сервисом, работа по новому протоколу или миграция баз данных некоторых сервисов на новую СУБД.

В таких случаях на один спринт берутся некоторые задачи из пула задач, именуемого беклогом, которые распределяются между участниками системы. Владелец пространства, а также другие его администраторы имеют возможность запустить спринт, назначив каждому участнику задачи. Как только спринт запущен, его участники берут в работу данные им задачи, меняя их статус. По окончании спринта спринт также закрывается, невыполненные задачи возвращаются в пул задач или переносятся в следующий спринт.

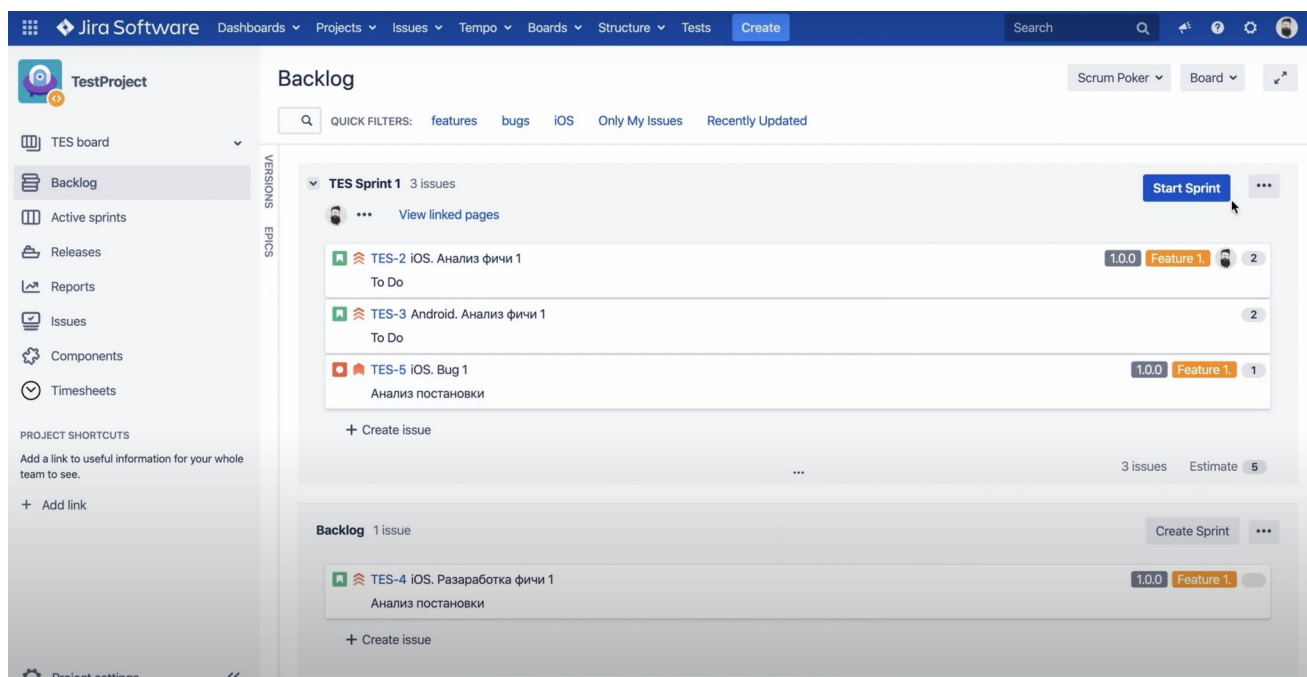


Рисунок 5. Наполнение спринта задачами из беклога, подготовка к запуску спринта.

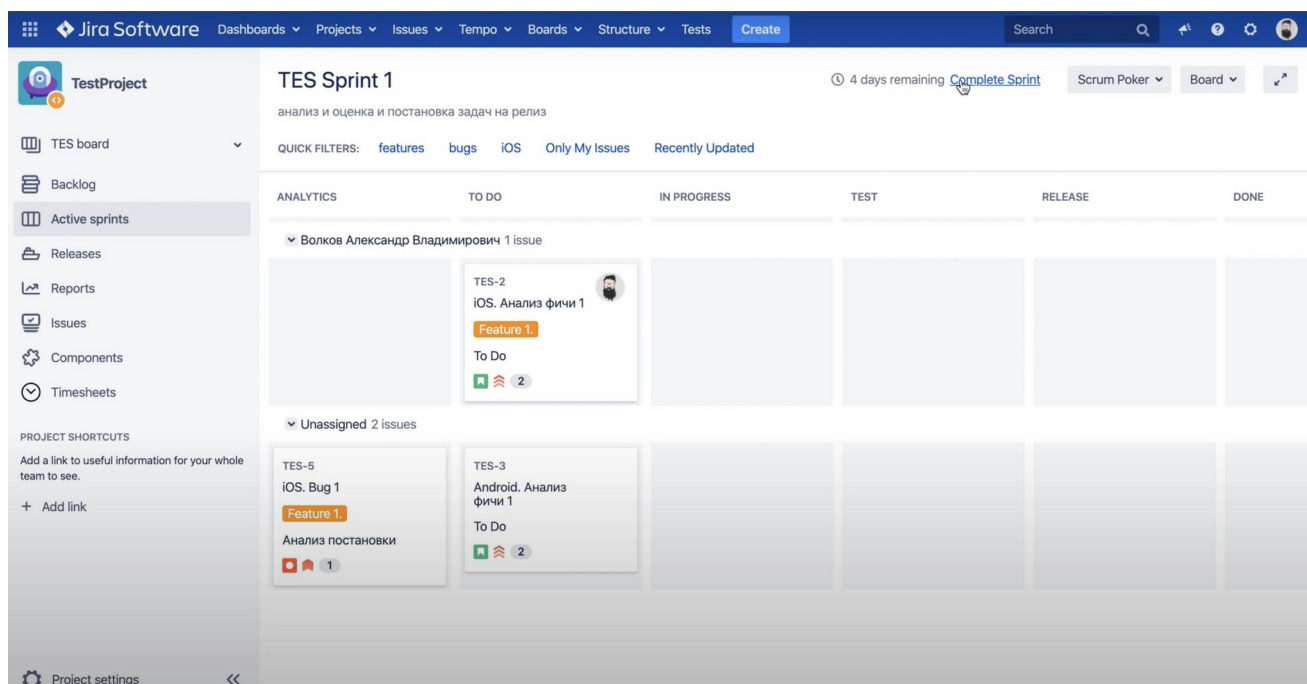


Рисунок 6. Запущенный спринт в Jira[3], задачи еще не выполнены.

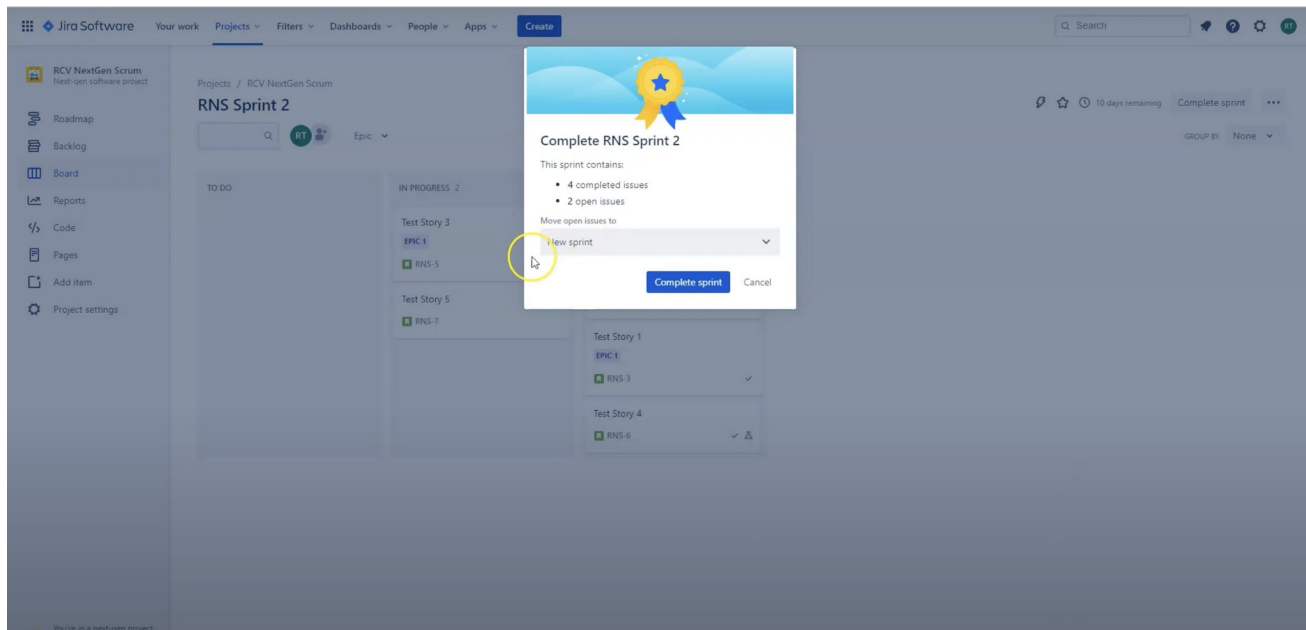


Рисунок 7. Завершение спринта, большинство задач выполнено. Незавершенные задачи будут перенесены в новый спринт.

Сервис предоставляет возможность интеграции со смежными системами, например, Confluence[7], Bitbucket[8] и Slack[9]. Также доступна отправка уведомлений для оповещения участников команды о том или ином событии.

1.2.2 Сервис Yandex Tracker, Yandex

В 2012 году вышел российский аналог Jira[3] сервис Yandex Treker[4], разработанный одноименной компанией. Сначала данный продукт был проприетарной разработкой и предназначался для внутреннего использования, однако в ноябре 2017 года он стал доступен внешним организациям. Особую популярность он приобрел в 2023 году, когда началась программа вендорозамещения, в рамках которой российским компаниям было рекомендовано отказаться от зарубежных IT-продуктов и использовать отечественные аналоги.

Сервис имеет схожие на Jira[3] функции и интерфейс и является на сегодняшний день полноценной заменой импортного продукта.

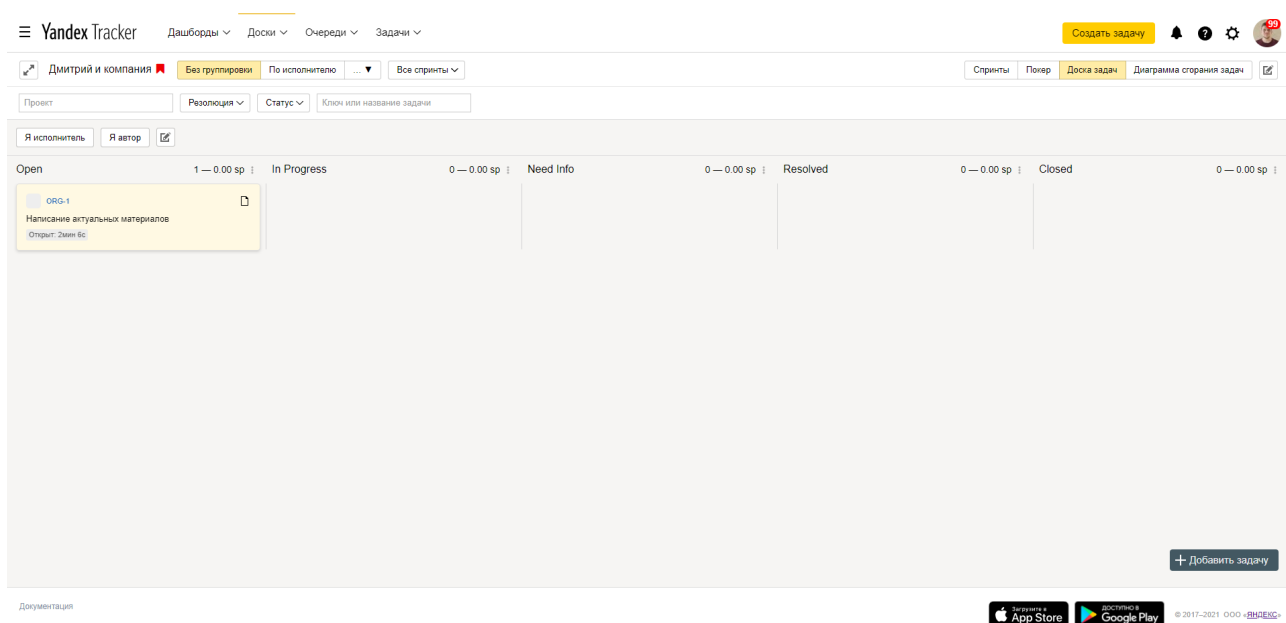
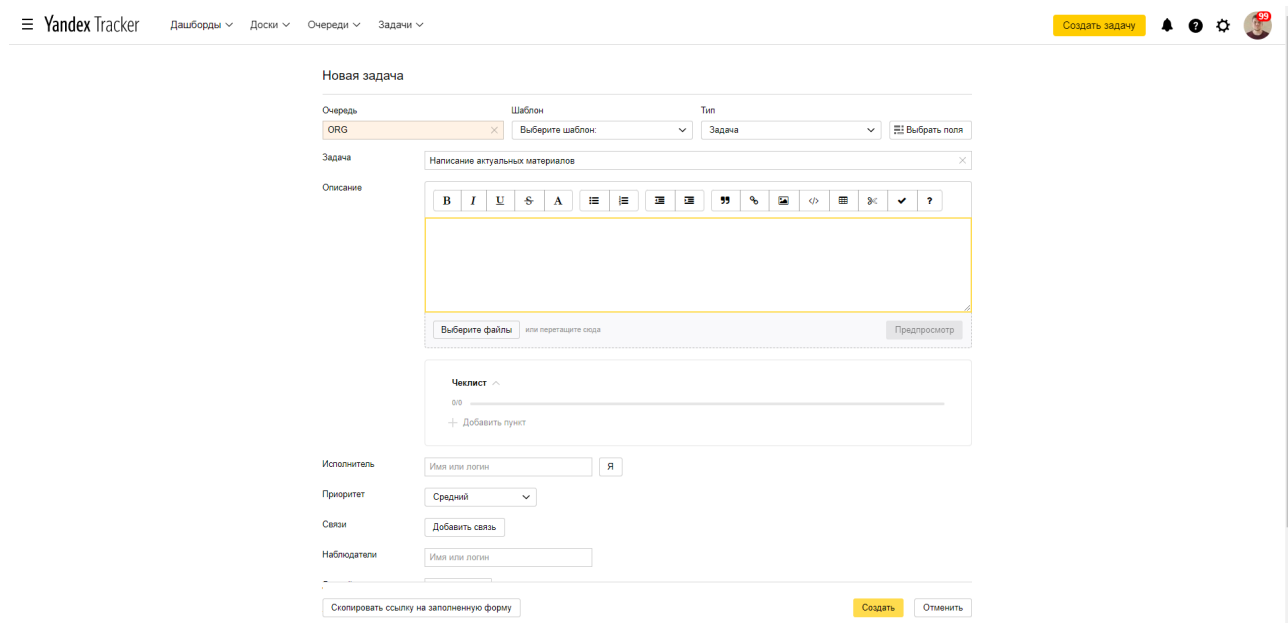


Рисунок 8. Центральная доска задач Yandex Treker[4].



Новая задача

Очередь: ORG Шаблон: Выберите шаблон Тип: Задача

Задача: Написание актуальных материалов

Описание: [Rich text editor with bold, italic, underline, link, list, and other formatting options]

Исполнитель: [Input field] Я

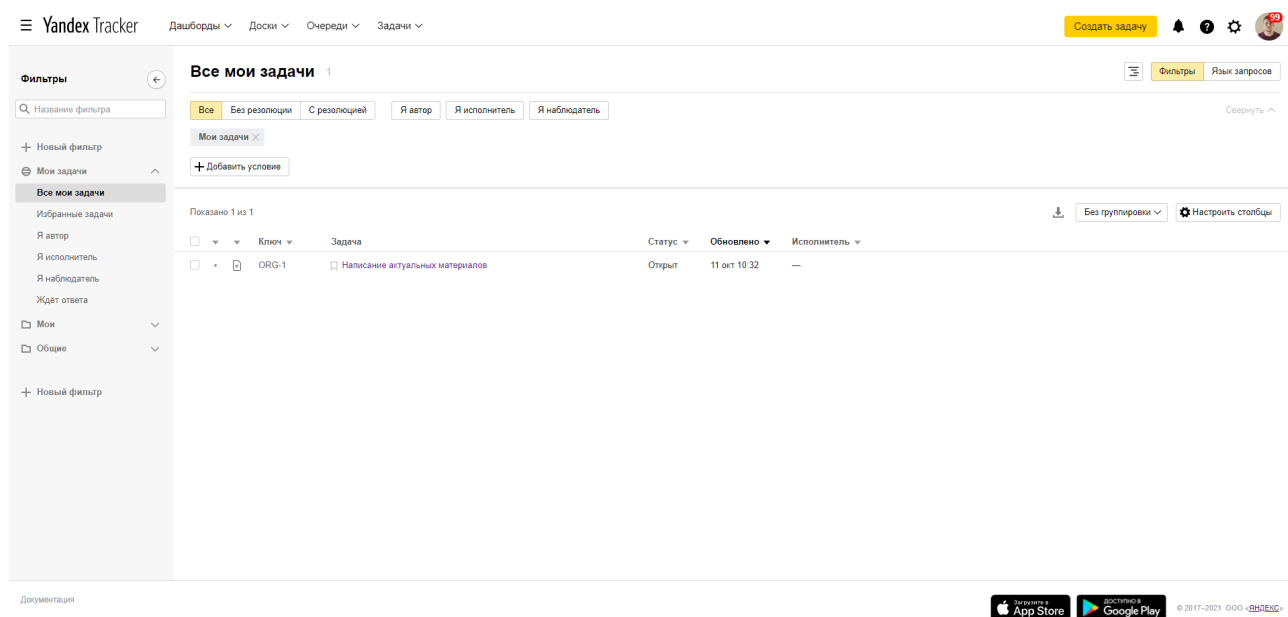
Приоритет: Средний

Связи: Добавить связь

Наблюдатели: [Input field]

Скопировать ссылку на заполненную форму Создать Отменить

Рисунок 9. Создание задачи в Yandex Treker[4].



Все мои задачи

Фильтры: [Search bar] + Новый фильтр

Мои задачи: [Dropdown menu]

Показано 1 из 1

	Ключ	Задача	Статус	Обновлено	Исполнитель
<input type="checkbox"/>	ORG-1	Написание актуальных материалов	Открыт	11 окт 10:32	—

Дokumentation

App Store Google Play © 2017–2021 ООО «Яндекс»

Рисунок 10. Возможность отображения всех задач вне kanban-доски.

Подытожив, можно сказать, что Яндекс Трекер[4] и Jira[3] имеют очень схожий функционал и их отличия — косметические. Например, у Трекера, субъективно, более минималистичный дизайн в тонах компании.

1.2.3 Сервис Security Incident Response (SIR[5]), ServiceNow[11]

Это облачный сервис, предоставляемый платформой ServiceNow[11], который предназначен для управления инцидентами безопасности в организациях. Сервис помогает автоматизировать процессы реагирования на инциденты безопасности.

Пользователю предоставляется широкий спектр функций, таких как:

- Автоматизация процессов управления инцидентами: сервис позволяет создавать инциденты вручную, а также автоматически с помощью датчиков. Инцидент эскалируется на ответственного лица, уведомляя его об этом.
- Интеграция с системами мониторинга: сервис позволяет выгружать обрабатываемые данные в такие SIEM[10]-системы для удобного мониторинга происходящих ситуаций на объекте.
- Единый интерфейс для управления инцидентами: система предлагает удобный UI для управления инцидентом разными лицами.
- Управляемое реагирование: система предлагает отслеживание прогресса реагирования на инцидент.

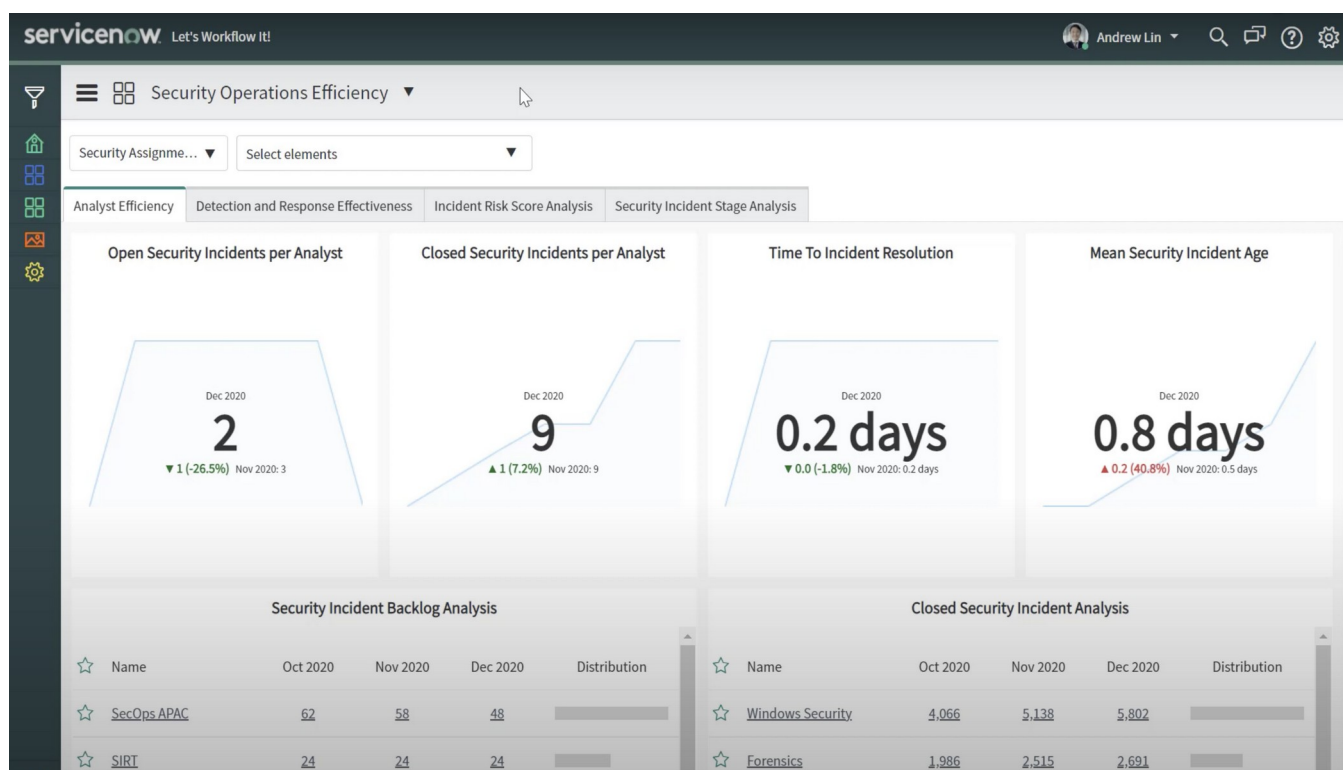


Рисунок 11. Центральная панель SIR[5].

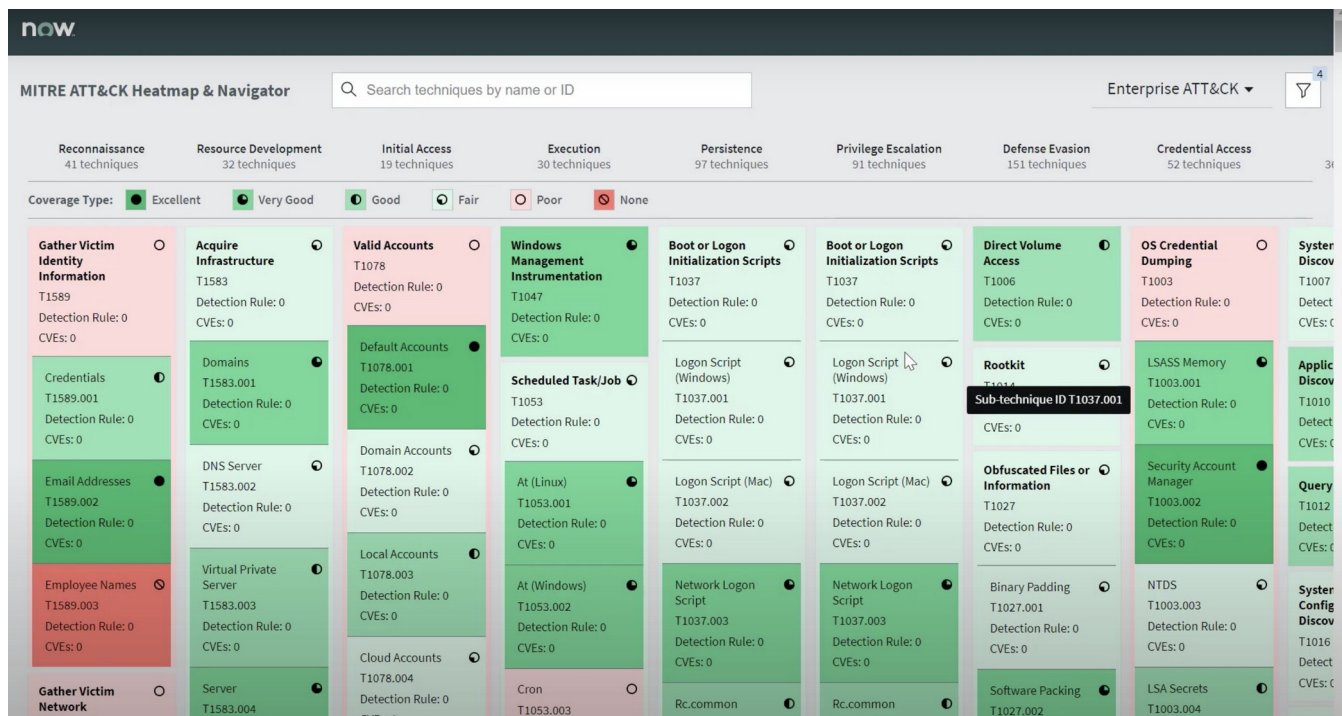


Рисунок 12. Доска с инцидентами SIR[5].

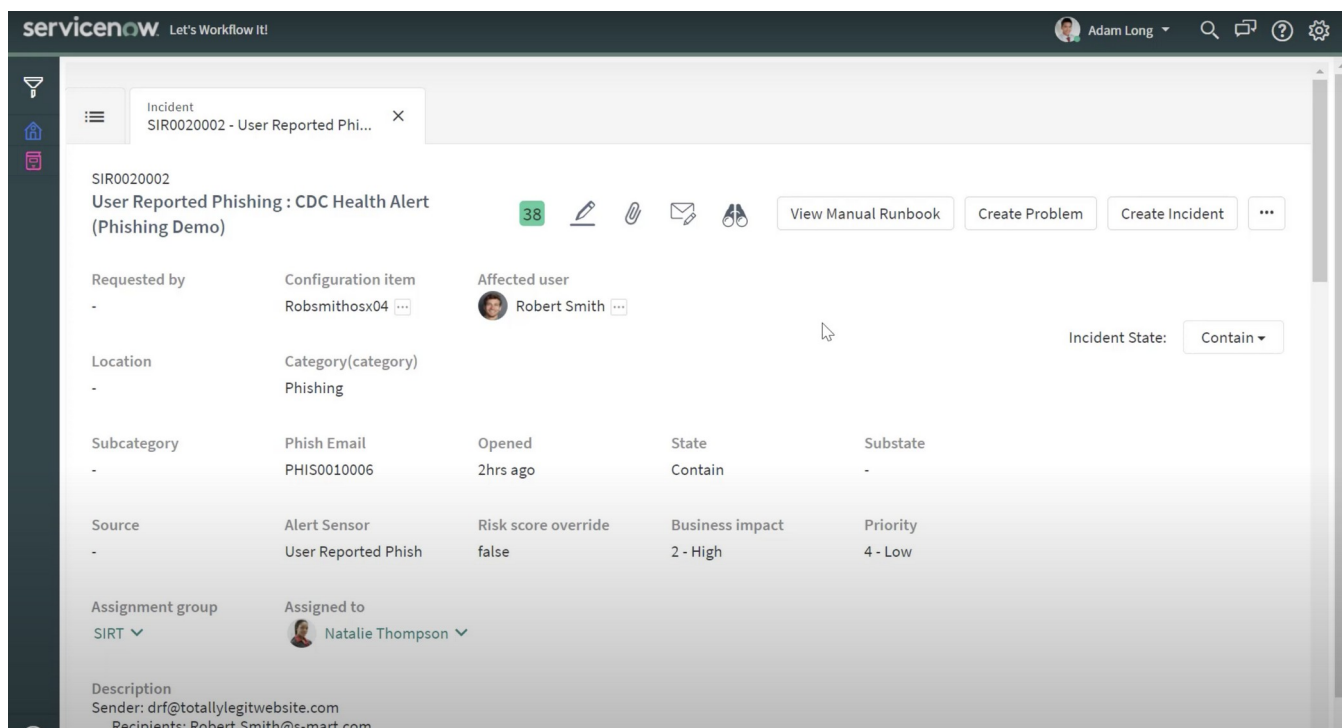


Рисунок 13. Корректировка инцидента в SIR[5].

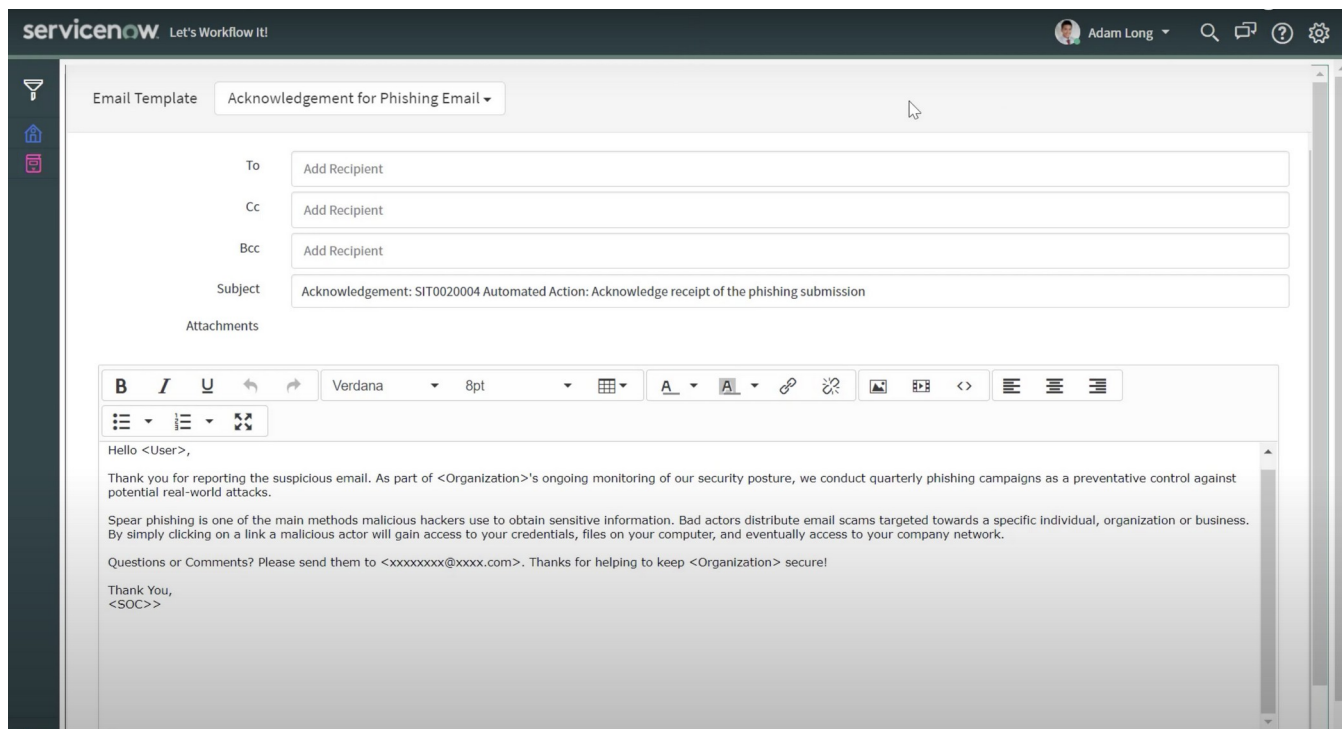


Рисунок 14. Создание кастомизированного уведомления на инцидент.

Таким образом, данная система является наиболее близким по функционалу конкурентов для разрабатываемой системы.

1.3 Сравнительный анализ рассмотренных аналогов

В Таблице 1 приводится сравнение разрабатываемой системы с конкурентами. Ключевые характеристики, подлежащие сравнению в данной таблице, выбраны релевантными функционалу серверной части, разрабатываемой в данной работе.

Необходимо заметить, что критерии сравнения в таблице были выбраны, исходя из технического задания и не отображают всего функционала рассматриваемых систем. Фокус в данном случае сделан на ключевых возможностях, которые должен предоставлять разрабатываемый продукт.

Критерий сравнения	Наша система	Jira[3]	Yandex Treker[4]	SIR[5]
Управление задачами	+	+	+	+
Доски с задачами	+	+	+	+
Доски с колонками-статусами	+	+	+	-
Интеграция датчиков	+	-	-	+
Гибкая настройка уведомлений	+	+-	+-	+
Динамическая система дежурств	+	-	-	-
Интуитивность	+	+-	+	+
Автосоздание тикетов	+	-	-	+

Таблица 1. Сравнение разрабатываемой системы с существующими аналогами.

В результате сравнительного анализа стало видно, что, во-первых, на рынке отсутствуют продукты с динамической системой дежурств, которая необходима для еще большей автоматизации процесса и гибкости его настройки. Во-вторых, нет решения, которое сочетало бы в себе как интеграцию датчиков, так и автосоздание тикетов в досках с колонками-статусами, что является упущением и снова лишением существующих систем гибкости. В разрабатываемой системе учтены оба недочета, делая ее более автоматизированной, не теряя при этом в интуитивности управления.

1.4 Выводы по главе

В данной главе представлено описание предметной области разрабатываемой системы, ее актуальность. После рассмотрения аналогов разрабатываемой системы были выявлены основные отличия и сходства существующих систем с текущим продуктом. В результате сравнительного анализа можно с уверенностью сказать, что он имеет перспективу к использованию в определенном сегменте предприятий, а именно там, где требуется end-to-end решение, начинающееся с интеграции с датчиками оповещения о нештатных ситуациях и заканчивающееся успешным завершением возникшего инцидента, поступившего от датчиков.

Глава 2. Проектирование системы и технологии разработки

2.1 Выбор архитектуры системы

На момент написания работы существует несколько архитектурных подходов к построению серверных приложений, каждый из которых имеет свои преимущества и недостатки. Среди наиболее распространенных архитектур можно выделить два основных типа: монолитная архитектура и микросервисная архитектура.

2.1.1 Монолитная архитектура

Данный подход подразумевает объединение всего функционала системы в одном запущенном приложении. Данный подход имеет преимущества, такие, как:

- Простота проектирования и быстрота разработки в начале: весь код сосредоточен в одном месте, нет необходимости думать о синхронизации отдельных частей системы.
- Простота тестирования: так как все компоненты находятся в одном месте, интеграционное тестирование системы сильно упрощается.
- Легкость в развертывании на сервере в начале: для функционирования системы, имеющей монолитную архитектуру, достаточно запустить один экземпляр приложения с файлами конфигурации.

Однако монолит имеет ряд недостатков, в основном связанных с масштабированием приложения при кратном росте количества пользователей. Например:

- Сложность и высокая стоимость масштабирования: если нагрузка растет на определенную часть системы, мы вынуждены увеличивать количество экземпляров всего монолитного приложения, что будет очень дорого. Более того, очень сложно будет настроить связь между этими экземплярами.
- Сложность разработки: когда функционал приложения начинает расти, в единой кодовой базе проекта становится очень сложно разбираться. Более того, в монолите так или иначе присутствует высокая связанность модулей, что также будет затруднять процесс исправления дефектов и написания нового функционала.
- Сложность в развертывании: при даже самом маленьком изменении в коде на сервере придется перезапускать все приложение. Это может привести к простою и снижению доступности сервиса.

Таким образом, монолит — хорошее решение для MVP проекта, но при росте нагрузки система будет требовать миграции на микросервисную архитектуру.



Рисунок 15 – Монолитная архитектура.

2.1.2 Микросервисная архитектура

Ключевая особенность микросервисной архитектуры – это разделение системы на несколько самостоятельно работающих модулей, коммуницирующих друг с другом посредством сети. Данный подход имеет ряд преимуществ, который делает его де-факто стандартом архитектуры систем в наше время:

- **Масштабирование:** Возможность независимо масштабировать каждую службу в зависимости от требований нагрузки.
- **Отсутствие привязки к стеку разработки:** Возможность использования различных языков программирования и технологий для написания сервисов системы, что позволяет выбирать наиболее подходящие инструменты для конкретных задач и легче находить кадры
- **Устойчивость:** проблема в одной службе не приводит к сбою всей системы. Это повышает общую устойчивость приложения.
- **Упрощение тестирования:** Малые масштабы служб упрощают тестирование и отладку. Каждую службу можно тестировать отдельно.
- **Распределение нагрузки:** Возможность легко распределять нагрузку между различными службами и управлять этим процессом.
- И многое другое.

Ввиду перечисленного, можно сделать вывод, что микросервисная архитектура решает многие проблемы своего монолитного конкурента, позволяя системе выдерживать высокие нагрузки и широкие возможности для безопасного масштабирования.

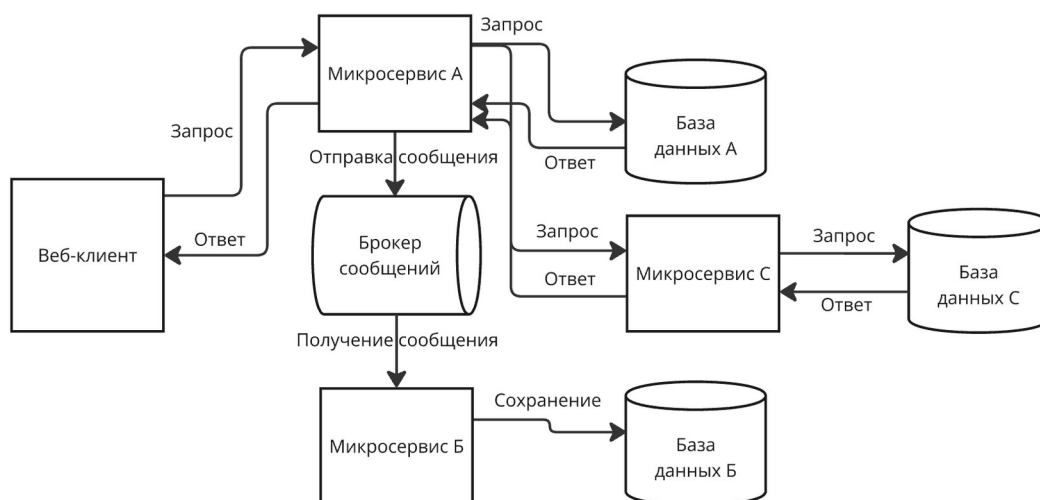


Рисунок 16 – Пример микросервисной архитектуры.

2.1.3 Сравнительный анализ серверных архитектур

В Таблице 1 приводится сравнение вышеописанных архитектур, используемых для разработки серверных приложений.

Критерий сравнения	Монолитная архитектура	Микросервисная архитектура
Сложность первичной разработки	Низкая	Средняя
Сложность дальнейшей разработки	Высокая	Средняя
Сложность тестирования	Низкая	Средняя
Сложность геораспределенности и масштабирования	Очень высокая	Средняя
Отказоустойчивость	Низкая	Высокая
Скорость первичного развертывания	Высокая	Средняя
Сложность дальнейшего развертывания	Средняя	Высокая
Затраты на первичное развертывание	Низкие	Средние
Затраты на дальнейшее развертывание	Высокие	Средние

Таблица 2. Сравнение архитектур серверных приложений.

2.2 Описание архитектуры системы

Исходя из требований к разрабатываемой системе в качестве основной архитектуры серверной части была выбрана микросервисная архитектура. Ключевыми аспектами при выборе были: высокая способность к масштабированию и отказоустойчивость, невысокий рост сложности разработки с течением времени и ростом проекта, а также средние затраты на развертывание системы.

В данной архитектуре каждый сервис представляет собой отдельную программу. Подход подразумевает, что сервисы не имеют состояний и используют собственные хранилища для их сохранения. Хранилища, как правило, не разделяются между различными сервисами, а доступ к данным осуществляется через API сервиса-владельца, что позволяет сервисам работать автономно и независимо.

На рисунке ниже приведена диаграмма всей системы. Сервисы, разрабатываемые автором данной работы, изображены зеленым цветом, а сервисы, разрабатываемые коллегами по проекту – серыми.

С системой взаимодействуют (то есть имеет возможность инициировать какое-то действие) пользователи и датчики. Пользователи используют клиент в виде браузера для доступа к веб-странице и имеют возможности по управлению системой. Датчики взаимодействуют с системой по сети и имеют возможности создавать тикеты по устранению инцидентов. Для обеспечения отказоустойчивости базы данных могут быть реплицированы, а количество экземпляров сервисов увеличено (не отображено на схеме с целью простоты ее понимания). Все сервисы также имеют систему логирования, которая позволит анализировать ошибки в работе системы, понимать их причины и устранять баги (также не отражено на схеме).

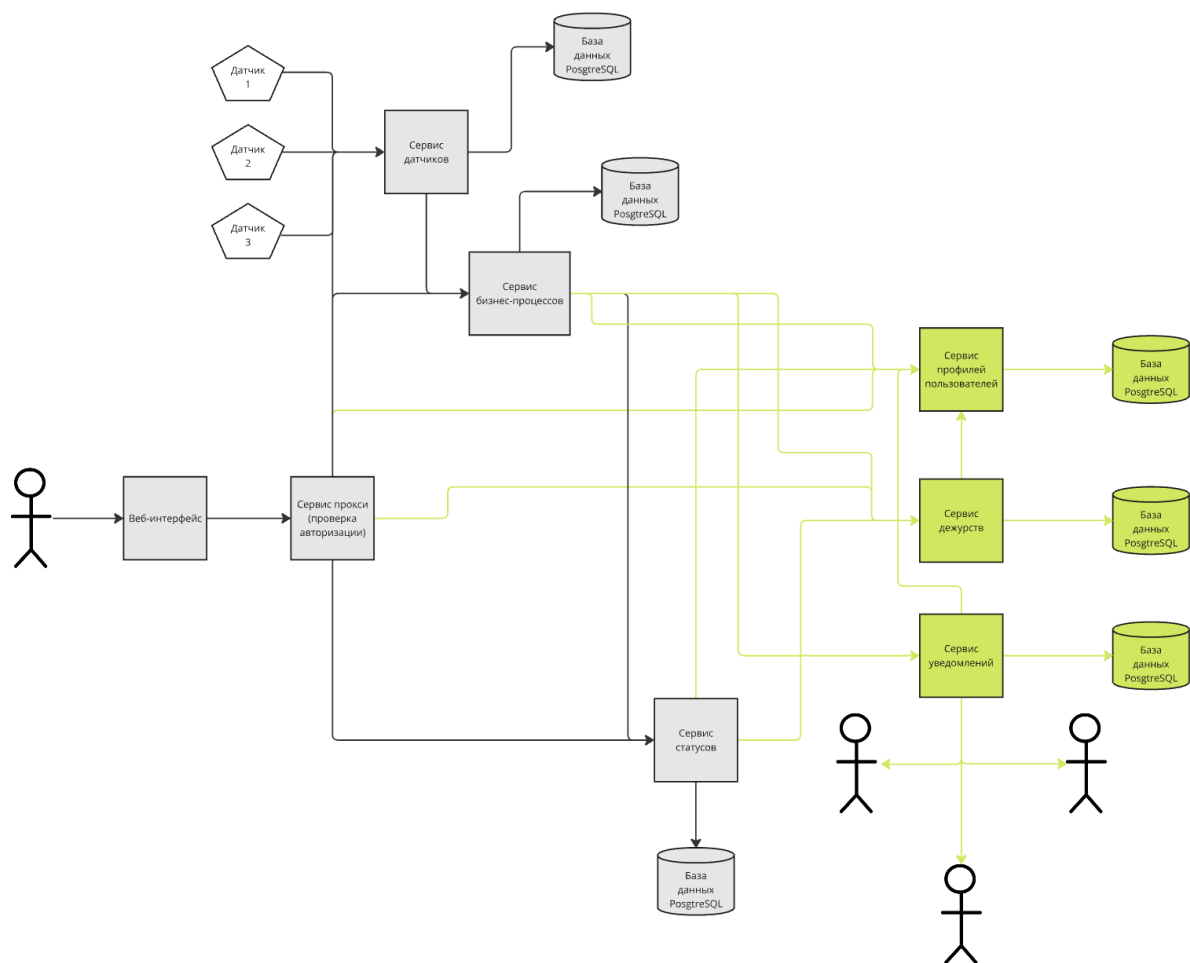


Рисунок 17 – Архитектура серверной части системы.

2.3 Пользовательские сценарии

На схеме приведены основные функции, которые пользователь может выполнять над указанной системой. Отдельно стоит отметить, что он совершает это через другие сервисы системы (например, любая операция идет через прокси-сервис, отображенный выше).



Рисунок 18. Пользовательские сценарии системы.

2.4 Описание общей архитектуры сервисов

Как уже было сказано выше, система состоит из множества сервисов, внутренняя реализация каждого из которых варьируется в соответствии с выполняемой им функцией.

Общий архитектурный подход, используемый в сервисах профилей пользователей, чата, дежурств и уведомлений, представляет из себя паттерн MVC, адаптированный под фреймворк Spring Boot[15].

- Model (модель) — слой, отвечающий за бизнес-логику (сервисы) и доступ к данным (репозитории и сущности).
- View — слой, отвечающий за визуализацию данных. При наличии тонкого клиента (отдельного веб-интерфейсного приложения) в стандартном сервисе Spring Boot[15] может отсутствовать.

- Controller — слой, обеспечивающий коммуникацию между слоями Model и View. Controller принимает запросы со стороны клиента и передает их Model, предоставляя определенный интерфейс для доступа к последней. Model обрабатывает запрос, формирует ответ и возвращает ответ View через Controller.

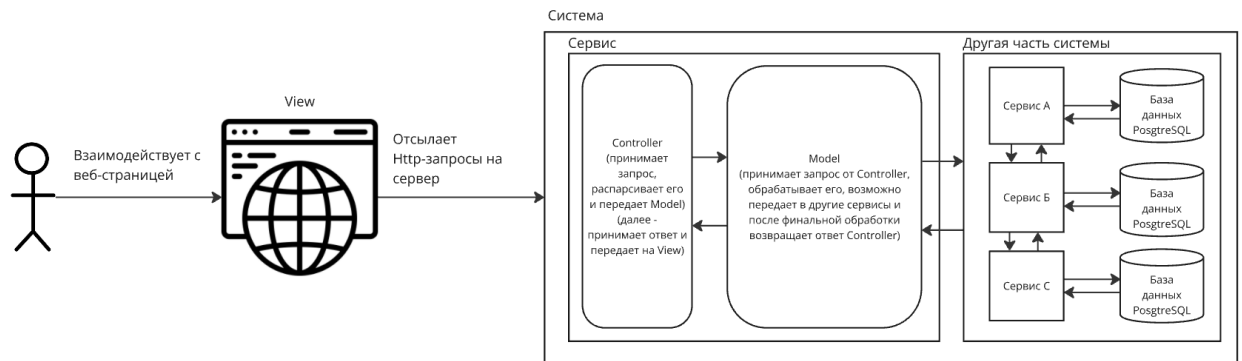


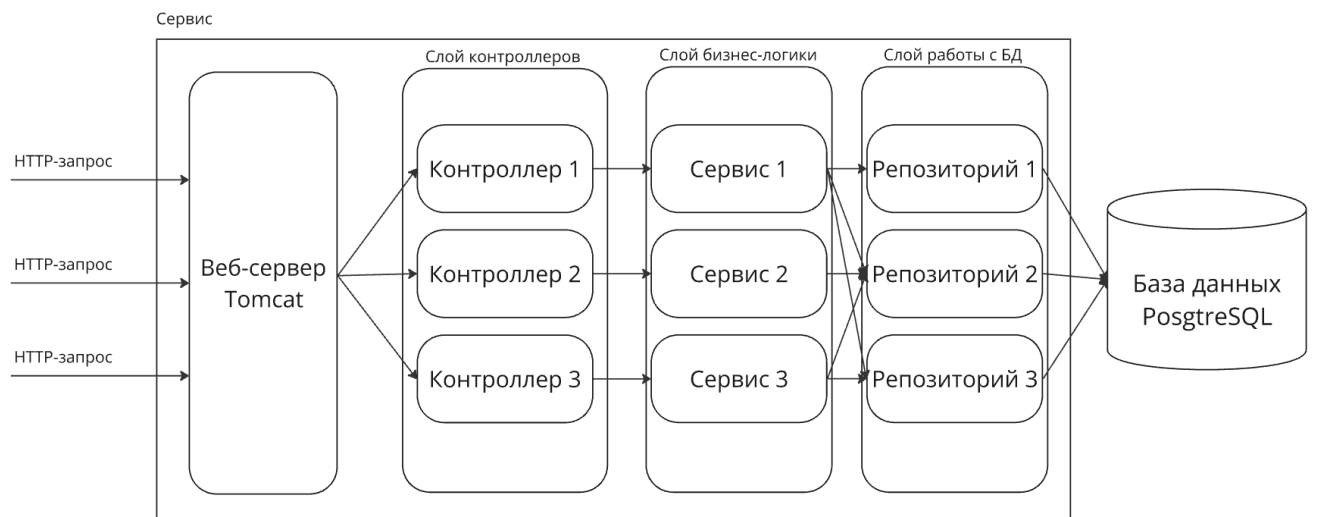
Рисунок 19. Модель MVC.

Каждый из слоев является слабо связанным слоем, что означает, что он спроектирован так, чтобы быть легко заменяемым: он не знает ничего о внутреннем устройстве других слоев, с которыми он взаимодействует, работая по некой общей спецификации. Такой подход был выбран для ускорения скорости разработки, отладки и исправления дефектов. Подобный подход называется Clean Architecture и широко используется в промышленных проектах.

Для конкретной реализации паттерна MVC была использована его вариация - Spring Boot MVC. В частности, каждый сервис разделен на следующие слои:

- Слой контроллеров
- Слой бизнес-логики (слой сервисов)
- Слой для взаимодействия с БД (слой репозитория)

Более детальная схема архитектуры каждого сервиса приведена на рисунке 20.



Примечание. На схеме показано направление обработки запроса от прихода запроса на веб-сервер до отправки запроса к БД. После выполнения запроса базой данных управление передается в обратном порядке. Обратные стрелки не отображены на схеме с целью сохранения читаемости.

Рисунок 20. Архитектура каждого сервиса системы, использующего Spring Boot[15].

2.5 Выбор технологий реализации

2.5.1 Языки программирования

В качестве основного языка программирования для реализации разрабатываемых в рамках данной работы сервисов был выбран язык Java[12].

Основные преимущества данного языка программирования:

- Универсальность: Java[12] следует принципу «пиши один раз, запускай где угодно». Это достигается благодаря тому, что скомпилированные программы на Java[12] исполняются в особом контейнере Java[12]-приложений, называемом виртуальной машиной Java[12] - JVM[13]. Она является платформенно-независимой программой и может запускаться на большинстве операционных систем.
- Многофункциональные фреймворки, имеющие обширную документацию и поддержку со стороны огромного сообщества Java[12]-программистов. Например, Spring[14] и его потомок Spring Boot[15] предоставляют инструменты для написания микросервисов, взаимодействия с базой данных, брокерами сообщений, системами кешей и многое другое. Все это имеет высокий уровень абстракции, что позволяет в очень сжатые сроки создавать и запускать новые сервисы, не теряя при этом в производительности.
- Производительность: несмотря на то, что Java[12] является по большей части интерпретируемым языком программирования, ее современные версии JVM[13] исполняют код очень быстро благодаря различного рода оптимизациям. Самой известной из них является JIT-компиляция — технология, позволяющая компилировать часто используемые куски байт-кода с целью их нативного выполнения (чистая компиляция всегда быстрее чистой интерпретации).
- Автоматическое управление памятью: в Java[12] отсутствует возможность прямого взаимодействия с памятью (в отличие, например, от C++[16] и Си[17]), что позволяет программисту думать о том, не «как написать», а «что написать», тем самым ускоряя его работу в разы. Это достигается за счет сборщика мусора — части JVM[13], отвечающей за своевременную очистку неиспользуемой памяти.
- Сообщество: на Java[12] написано очень большое число продуктов и их число растет с каждым днем. Это создает большое сообщество языка, что делает его популярным решением, которое следует использовать, зная, что решение той или иной проблемы можно найти намного быстрее, чем в других языках.
- Большое количество готовых библиотек: следствие из предыдущего пункта. Для большинства повседневных задач в программировании в Java[12] уже есть готовые решения. Например, создание функционала аутентификации и авторизации, парсинг

документов различных форматов, простые интеграции с другими популярными сервисами, — для всего этого в Java[12] есть множество библиотек, которые имеют достаточный уровень абстракции для «быстрого погружения в тему».

- Интеграция с DevOps: Java[12] отлично интегрируется с современными инструментами DevOps, такими как Docker[18] и Kubernetes[36], что позволяет автоматизировать процесс развертывания и управления микросервисами, обеспечивая надежность и гибкость.

Помимо всех вышеперечисленных преимуществ, данный язык является основным языком программирования в крупнейших российских банках, таких как: Сбер[20], Тинькофф[21], ВТБ[22] и др. Он также широко используется в ряде IT предприятий: Яндекс[23], МТС[24], Билайн[25] и др.

2.5.2 Хранилища данных

Для хранения обрабатываемой информации в системе используется реляционная СУБД PostgreSQL[26].

2.5.2.1 PostgreSQL[26]

База данных используется в качестве основного хранилища данных, которые хранятся долгосрочно, часто добавляются и обновляются, и именно с такого рода операциями PostgreSQL[26] работает лучше всего. В PostgreSQL[26] данные хранятся (в подавляющем большинстве случаев) нормализованно, в целях уменьшения использованного места и обеспечения согласованности данных в случаях их изменения. На рисунке ниже приведены схемы трех баз данных, каждая из которых принадлежит одному из трех микросервисов:

- сервис профилей пользователей
- сервис дежурств
- сервис отправки уведомлений

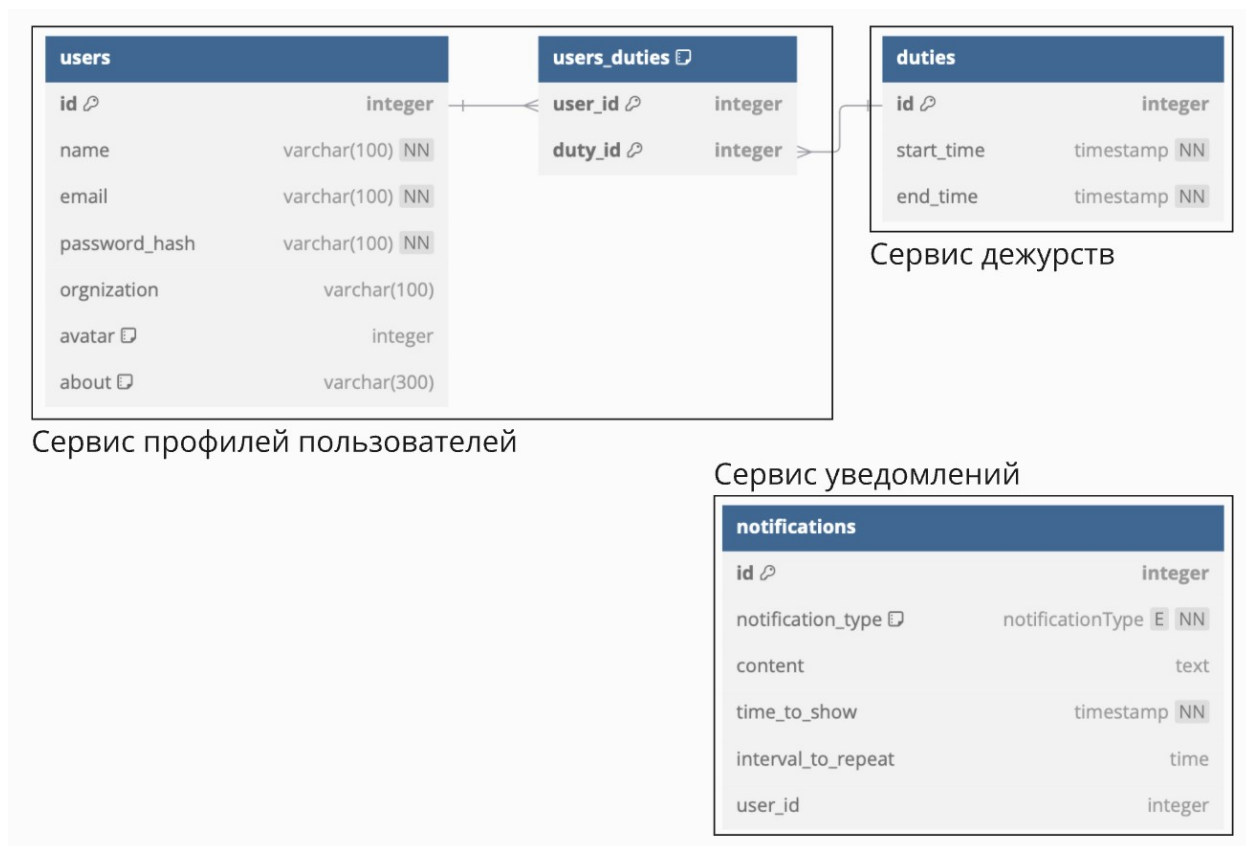


Рисунок 21 – Пример схемы данных PostgreSQL[26].

2.5.3 Межсервисное взаимодействие в системе

Все микросервисы в системе имеют RESTful взаимодействие, используя протокол HTTP и передачу тела запроса/ответа в json формате.

RESTful-взаимодействие имеет ряд преимуществ, таких как:

- Простота использования и реализации: как правило, RESTful-сервисы содержат стандартные HTTP(s)-методы: GET, POST, PUT, PATCH, DELETE. Этим методов зачастую достаточно, чтобы реализовать любую логику. Более того, Spring Boot[15] имеет поддержку этим методов через аннотации, что делает проектирование RESTful API в Controller-ах простым и удобным.
- Статус без состояния: один из принципов REST API гласит о том, что сервис не должен хранить состояния запросов, то есть каждый запрос должен быть самодостаточным — данных внутри его должно хватать, чтобы обработать запрос. Это упрощает масштабирование, избавляет сервер от дополнительных расходов на память, а также делает систему более устойчивой.
- Гибкость: REST позволяет клиенту и серверу взаимодействовать независимо друг от друга. Клиенты могут запрашивать данные в различных форматах (JSON, XML, HTML и

т.д.), что обеспечивает совместимость с различными клиентскими приложениями.

- Кеширование на стороне клиента: RESTful API могут использовать кеширование на стороне клиента или сети, что значительно улучшает производительность и уменьшает количество обращений к серверу.
- Удобство тестирования: RESTful API легко тестировать с помощью инструментов, таких как Postman или cURL. Это упрощает процесс отладки и проверки работы сервисов.
- Легкость в документировании: RESTful API обычно более легки для документирования напрямую из кода с использованием спецификации, таких как: Swagger[30] или OpenAPI[31].
- Широкая совместимость: RESTful API могут работать на любом устройстве, способном выполнять HTTP-запросы, включая веб-браузеры, мобильные приложения и другие серверные приложения.

Протокол HTTP был выбран как самый популярный для реализации RESTful API.

2.5.4 Другие используемые технологии

Среди нерассмотренных ранее технологий, но использованных при разработке системы, надо отметить: фреймворки юнит-тестирования JUnit[27] и модульного тестирования Mockito[28], ORM Hibernate[29], систему контейнеризации Docker[18] и оркестрации Docker-compose[19].

2.5.4.1 ORM Hibernate[29]

Hibernate[29] является одним из самых популярных объектно-реляционного отображения (ORM) в языке Java[12], который упрощает взаимодействие между объектами Java[12] и реляционными базами данных. Он позволяет разработчикам работать с базой данных, используя объекты, а не SQL-запросы. Hibernate[29] автоматически преобразует объекты Java[12] в записи в базе данных и обратно, что значительно упрощает обработку и управление данными.

Основными преимуществами, из-за которых данный фреймворк был выбран для использования в разрабатываемой системе, помимо указанного выше, являются:

- Портруемость: Hibernate[29] работает с различными реляционными базами данных, такими как MySQL[32], PostgreSQL[26], Oracle[33] и другими, что позволяет проектам

быть более универсальными и переносимыми. Это избавит разработчика от необходимости переписывать слой взаимодействия с БД при миграции на другую СУБД.

- Управление транзакциями: Hibernate[29] предоставляет механизмы для управления транзакциями, что обеспечивает надежность и консистентность данных.
- Кэширование: одна из ключевых вещей, которая делает использование ORM-фреймворков актуальным. Hibernate[29] поддерживает встроенное кэширование первого и второго уровня, что позволяет повысить производительность приложений, минимизируя количество запросов к базе данных.
- Запросы с использованием HQL: Hibernate Query Language (HQL) позволяет выполнять запросы к объектам Java[12], а не к таблицам в базе данных, что делает запросы более удобочитаемыми и объектно-ориентированными.
- Поддержка отношений: Hibernate[29] предоставляет удобные механизмы для управления отношениями между объектами (например, один к одному, один ко многим и многие ко многим).
- Интеграция со Spring Boot[15]: Hibernate[29] интегрируется с фреймворком Spring Boot[15], что позволяет использовать его вместе с другими компонентами Spring Boot[15] для создания мощных и масштабируемых приложений.

2.5.4.2 Фреймворки для тестирования JUnit[27] и Mockito[28]

Данные фреймворки являются де-фактом стандартом для тестирования функционала приложений, написанных на Java[12]. Mockito[28] также имеет продвинутые возможности для работы с компонентами Spring[14] и Spring Boot[15].

JUnit[27] предоставляет функционал для юнит-тестирования, то есть тестирования корректности работы отдельных методов.

Mockito[28] предоставляет возможности для модульного и в некотором роде интеграционного тестирования Spring[14] приложений, позволяя настроить заглушки при взаимодействии между различными слоями Spring[14]-приложения.

2.5.4.3 Docker[18]

Инструмент контейнеризации Docker [18] используется для запуска какого-либо сервиса в изолированной среде, называемой контейнером. Последний представляет из себя урезанную версию ОС без графического интерфейса и предоставляет взаимодействие с внешней средой по сети, используя порты.

Чаще всего этот инструмент используется для локального тестирования, значительно упрощая этот процесс благодаря тонкой настройке окружения, в котором запускается сервис.

Например, для тестирования написанного приложения с базой данных, проще всего написать Dockerfile, поднимающий инстанс СУБД в контейнере и предоставляющий доступ к нему. Преимущество на лицо — один раз написав скрипт, можно запускать его одной командой и быть уверенным, что никакие побочные эффекты не повлияют на работу самой БД. Это обеспечивает детерминированность работы БД.

Также Docker [18] необходим для последующего развертывания сервиса в системах оркестрации (о них ниже). Пример такого файла приведен ниже.

Листинг 2. Dockerfile, использующийся для сборки каждого сервиса.

```
FROM eclipse-temurin:21-jre-jammy
VOLUME /tmp
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["sh", "-c", "java ${JAVA_OPTS} -jar /app.jar"]
```

2.5.4.4 Docker-compose[19]

Docker-compose[19] представляет из себя легковесную среду оркестрации сервисов. Это значит, что такая программа управляет контейнерами с запущенными приложениями в них, настраивает их взаимодействие, отслеживает работоспособность и контролирует нагрузку на каждый из них.

Для запуска Docker-compose[19] достаточно пары команд.

Листинг 3. Docker-compose[19] файл, который используется для разворачивания разрабатываемой системы в облаке:

```
services:
  ##### business services #####
  duties-service:
    build: ./duties-service
    container_name: duties-service
    environment:
      SPRING_DATASOURCE_URL: jdbc:postgresql://duties-db/duty_db
      SPRING_DATASOURCE_USERNAME: duty_user
      SPRING_DATASOURCE_PASSWORD: duty_pass
```

```

healthcheck:
  test: curl -f http://duties-service/actuator/health || exit 1
  timeout: 10s
  interval: 7s
  retries: 10
ports:
  - "7070:7070"
depends_on:
  - duties-db
restart: on-failure

notifications-service:
  build: ./notifications-service
  container_name: notifications-service
  environment:
    SPRING_DATASOURCE_URL:
jdbc:postgresql://notifications-db/notification_db
    SPRING_DATASOURCE_USERNAME: notification_user
    SPRING_DATASOURCE_PASSWORD: notification_pass
  healthcheck:
    test: curl -f http://notifications-service/actuator/health ||
exit 1
    timeout: 10s
    interval: 7s
    retries: 10
  ports:
    - "8080:8080"
  depends_on:
    - notifications-db
    - profiles-service
  restart: on-failure

profiles-service:
  build: ./profiles-service
  container_name: profiles-service
  environment:
    SPRING_DATASOURCE_URL: jdbc:postgresql://profiles-db/profile_db

```



```

    SPRING_DATASOURCE_USERNAME: profile_user
    SPRING_DATASOURCE_PASSWORD: profile_pass
healthcheck:
    test: curl -f http://profiles-service/actuator/health || exit 1
    timeout: 10s
    interval: 7s
    retries: 10
ports:
    - "9090:9090"
depends_on:
    - profiles-db
restart: on-failure

##### DB services #####
duties-db:
    image: postgres:16.1
    container_name: duties-db
    ports:
        - "6541:5432"
    environment:
        - POSTGRES_PASSWORD=duty_pass
        - POSTGRES_USER=duty_user
        - POSTGRES_DB=duty_db
    healthcheck:
        test: pg_isready -q -d $$POSTGRES_DB -U $$POSTGRES_USER
        timeout: 10s
        interval: 7s
        retries: 10
    restart: on-failure

notifications-db:
    image: postgres:16.1
    container_name: notifications-db
    ports:
        - "6542:5432"
    environment:
        - POSTGRES_PASSWORD=notification_pass

```

```

    - POSTGRES_USER=notification_user
    - POSTGRES_DB=notification_db
healthcheck:
    test: pg_isready -q -d $$POSTGRES_DB -U $$POSTGRES_USER
    timeout: 10s
    interval: 7s
    retries: 10
restart: on-failure

profiles-db:
    image: postgres:16.1
    container_name: profiles-db
    ports:
        - "6543:5432"
    environment:
        - POSTGRES_PASSWORD=profile_pass
        - POSTGRES_USER=profile_user
        - POSTGRES_DB=profile_db
    healthcheck:
        test: pg_isready -q -d $$POSTGRES_DB -U $$POSTGRES_USER
        timeout: 10s
        interval: 7s
        retries: 10
    restart: on-failure

```

2.6 Выводы по главе

В данной главе представлено подробное описание архитектуры всей системы и общей архитектуры отдельных подсистем. Важной частью работы является анализ и рассмотрение различных подходов при проектировании приложения с уточнением преимуществ каждого из них. Как итог, после проведенного анализа, было принято решение использовать микросервисную архитектуру при разработке всей системы, а также RESTful API с протоколом HTTP для взаимодействия между сервисами системы. Спецификация RESTful API находится в приложении.

В подробностях были описаны технологии, использованные в работе, и объяснена причина выбора приведенных инструментов, начиная с языков программирования и хранилищ данных, заканчивая дополнительными инструментами для организации

взаимодействия с базой данных и развертывания приложений в изолированной среде.

Так, для разработки серверной части системы в основном были выбраны язык Java[12], СУБД PostgreSQL[26], ORM Hibernate[29], JUnit[27] и Mockito[28], а также Docker[18] и Docker-compose[19].

Глава 3. Программная реализация системы

В данной главе приведен функционал серверной части системы для каждого из разрабатываемых сервисов и для использующейся самописной библиотеки Common. Также в главе рассмотрена бизнес-логика каждого сервиса, его взаимодействие с базой данных и тестирование. Завершать главу будут разделы, посвященные системе сборки и документации REST API.

3.1 Сервис профилей пользователей

Ответственность данного сервиса — хранение информации о каждом пользователе и управление своим аккаунтом.

3.1.1 Особенности реализации сервиса

Сервис является внутренним сервисом, что означает, что он используется исключительно другими сервисами системы, не взаимодействуя напрямую с UI. Коммуникация с сервисом происходит посредством RESTful API, использующийся при этом протокол — HTTP.

Для обработки HTTP-запросов используется веб-сервер Tomcat (поставляется в составе Spring Boot[15]), низкоуровневый механизм сервлетов (аналогично — в составе Spring Boot[15]), а также написанный слой контроллера, каждый метод которого отвечает за определенный эндпоинт. В свою очередь каждый эндпоинт имеет контракт, специфицирующий, какие данные будут присланы для парсинга вместе с запросом и какие будут предоставлены в качестве результата.

В программу было добавлено логирование различных уровней, которое будет полезно для разбора нештатных ситуаций работы сервиса, мониторинга система, а также для разработки нового функционала.

3.1.2 Функционал сервиса

Сервис предоставляет следующий функционал:

Создать пользователя:

Листинг 4. Контракт создания пользователя.

```
@PostMapping
```

```
@ResponseStatus(HttpStatus.CREATED)
```

```

public UserDto createUser(@Valid @RequestBody NewUserRequest request)
{
    User user = userService.createUser(request);
    return userMapper.toDto(user);
}

```

@Data

```

public class NewUserRequest {
    @NotBlank
    @Size(min = 2, max = 250, message = "Имя пользователя должно
содержать от 2-х до 250-ти символов")
    private String name;

    @NotNull
    @Email
    @Size(min = 6, max = 100, message = "Почта пользователя должна
содержать от 6 до 100 символов")
    private String email;

    @NotNull
    @Phone
    private String phone;

    @NotBlank
    @Size(min=8, max = 40, message = "Пароль должен иметь длину от 8
до 40 символов")
    private String password;
}

```

@Data

```

public class UserDto {
    private Long id;

    private String name;

    private String email;
}

```

```

        private String phone;
    }

```

Изменить информацию о пользователе:

Листинг 5. Контракт изменения пользователя.

@PutMapping

```

public Long put(@RequestBody User user) {
    return userService.updateUser(user).getId();
}

```

@Data

@Entity

@Builder

@ToString

@Table(name = "users")

@NoArgsConstructor

@AllArgsConstructor

@EqualsAndHashCode(of = {"id"})

```

public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, length = 250)
    private String name;

    @Column(nullable = false, unique = true, length = 254)
    private String email;

    @Column
    private String phone;

    @Column
    private String password;

    @Column

```

```

        private String organization;

        @Column
        private String avatar;

        @Column
        private String about;
    }

```

Получить информацию о пользователе:

Листинг 6. Контракт получения информации о пользователе.

```

@GetMapping("/{id}")
public UserFullInfoDto getUserById(@PathVariable Long id) {
    User user = userService.getUserById(id);
    return userMapper.toFullInfo(user);
}

```

```

@Data
@NoArgsConstructor
@AllArgsConstructor
public class UserFullInfoDto {
    private Long id;

    private String name;

    private String email;

    private String phone;

    private String organization;

    private String avatar;

    private String about;
}

```

Получить идентификаторы пользователей по имени:

Листинг 7. Контракт для получения идентификаторов всех пользователей по имени.

@GetMapping

```
public List<Long> getUserIdsByName(@RequestParam(name = "name")
String name) {
    List<User> users = userService getUsersByName(name);
    List<Long> userIds = users.stream()
        .map(User::getId)
        .collect(Collectors.toList());
    return userIds;
}
```

Определить по имеющимся id, существуют ли пользователи с такими id:

Листинг 8. Контракт для получения идентификаторов существующих пользователей по имеющимся идентификаторам.

@GetMapping("/notexisting")

```
public List<Long> getNotExistingUsersByIds(@RequestParam List<Long>
ids) {
    return userService.getNotExistingUserIds(ids);
}
```

Получить идентификатор пользователя по его email и паролю:

Листинг 9. Контракт для получения идентификатора пользователя по его email и паролю.

@GetMapping("/userId")

```
public Long getUserIdByEmailAndPassword(@RequestParam String email,
                                         @RequestParam String
password) {
    return userService.getUserIdByEmailAndPassword(email, password);
}
```

Получить всю информацию о пользователях (кроме пароля) по их идентификаторам:

Листинг 10. Контракт для получения всей информации о пользователях (кроме пароля) по их идентификаторам.

@GetMapping("/info")

```
public List<UserDtoPartial> getUsersByIds(@RequestParam List<Long>
ids) {
    return userMapper.toPartial(userService.getId(ids));
}
```


3.1.3 Бизнес-логика

Основная бизнес-логика сервиса сосредоточена в классе `UserService`. Он инкапсулирует в себе логику по обработке запросов — от принятия запроса в том виде, котором он пришел в предыдущий слой (слой контроллеров) до возврата результата.

Каждый соответствующий определенному типу запросу (и эндпоинту) метод содержит в себе логирование, препроцессинг запроса с возможным обращением к базе данных (например, если нужна требуется обновить данные пользователя, необходимо сперва проверить существование такого пользователя и вернуть HTTP-код 404 клиенту). Далее следует главная обработка запроса с обращением к базе данных. После того, как база данных вернула результат, запускается процесс его обработки (например, если вернулась пустая сущность — `null`, то клиенту аналогично нужно вернуть HTTP-код 404). После этого происходит маппинг сущности, вернувшейся из базы данных, к сущности, которая будет отдана клиенту — то есть происходит преобразование объекта одного класса к объекту другого класса. Для этого в сервисе используются мапперы из библиотеки `mapstruct`, которая генерирует необходимые методы, избавляя программиста от написания рутинного кода.

Пример маппера для сервиса пользователей:

Листинг 11. Mapper `UserMapper`.

```
@Mapper(componentModel = "spring", injectionStrategy =
InjectionStrategy.CONSTRUCTOR)
public interface UserMapper {
    User toUser(NewUserRequest request);

    UserDto toDto(User user);

    UserFullInfoDto toFullInfo(User user);

    UserDtoPartial toDtoPartial(User user);

    List<UserDtoPartial> toPartial(List<User> users);
}
```

При неуспешной валидации запроса бросается исключение. Например, в запросе на обновление пользователя при ненайденном пользователе бросается исключение `NotFoundException`, а при некорректном `id` — `NotValidIdException`.

Листинг 12. Метод обновления данных о пользователе и исключения, которые он бросает.

```
@Transactional
public User updateUser(User user) {
    if (user.getId() == null || user.getId() < 0) {
        throw new NotValidIdException();
    }
    log.info("updating user with id = {}");
    User currentUserData = repo.findUserById(user.getId());
    if (currentUserData == null) {
        throw new NotFoundException("No user with such id!");
    }
    if (user.getName() == null) {
        user.setName(currentUserData.getName());
    }
    if (user.getEmail() == null) {
        user.setEmail(currentUserData.getEmail());
    }
    if (user.getPhone() == null) {
        user.setPhone(currentUserData.getPhone());
    }
    if (user.getPassword() == null) {
        user.setPassword(currentUserData.getPassword());
    }
    if (user.getOrganization() == null) {
        user.setOrganization(currentUserData.getOrganization());
    }
    if (user.getAvatar() == null) {
        user.setAvatar(currentUserData.getAvatar());
    }
    if (user.getAbout() == null) {
        user.setAbout(currentUserData.getAbout());
    }
    return repo.save(user);
}

public class NotFoundException extends RuntimeException {
```

```

        private static final String message = "%s отсутствует в памяти
программы.";

        public NotFoundException(Object obj) {
            super(String.format(message, obj));
        }

        public NotFoundException(String s) {
            super(s);
        }
    }

    public class NotValidIdException extends RuntimeException {
        public NotValidIdException() {
            super("id должен быть > нуля.");
        }
    }
}

```

Как упоминалось ранее, реализация методов сервиса предусматривает препроцессинг — предварительную обработку запроса — например, значение `id` на положительность. Это сделано с целью экономии ресурсов системы — нет смысла перегружать базу данных запросом с невалидными данными, если они могут быть проверены еще до обращения к базе.

3.1.4 Взаимодействие с базой данных

В качестве СУБД для сервиса был выбран PostgreSQL[26]. Взаимодействие с ним происходит через одноименный драйвер, инкапсулирующий взаимодействие с базой по бинарному протоколу.

Для взаимодействия с базой данных на бизнес-уровне используется ORM Hibernate[29], который генерирует SQL-запросы. Для выполнения сложных запросов используется native SQL.

За логику работы с базой данных отвечает класс `UserRepository`, унаследованный от `JpaRepository` — класса, который предоставляет ORM Hibernate[29] и который отвечает за генерацию SQL-запросов. Каждый метод `UserRepository` соответствует определенному SQL-запросу. Например, метод `getUsersById` отвечает за получение пользователей из базы данных по введенным `ids`.

Листинг 13. Метод получения пользователей из базы данных по их идентификаторам.

```

@Query(value = "SELECT * FROM users WHERE id IN :ids", nativeQuery =
true)

```

```
List<User> getUsersById(List<Long> ids);
```

Для создания таблицы при подключении к базе данных используется скрипт `schema.sql`, который удаляет старую таблицу и создает новую.

Листинг 14. Скрипт для инициализации базы данных сервиса пользователей.

```
DROP TABLE IF EXISTS users;
```

```
CREATE TABLE users (  
    id BIGSERIAL PRIMARY KEY,  
    name VARCHAR(100) NOT NULL,  
    email VARCHAR(100) NOT NULL UNIQUE,  
    phone VARCHAR(20) NOT NULL UNIQUE,  
    password VARCHAR(100) NOT NULL, -- length = 32? (in case of  
usage md5 for hasing)  
    organization VARCHAR(100),  
    avatar TEXT DEFAULT 'https://img.icons8.com/?size=100&id=z-  
JBA_KtSkxG&format=png&color=000000',  
    about VARCHAR(300) DEFAULT 'hey I''m using Argus'  
);
```

3.1.5 Тестирование

Для данного сервиса были написаны unit-тесты с использованием библиотек JUnit5[27] и Mockito[28]. Они покрывают базовый функционал сервиса и по большей части выполняют роль регрессионных тестов, которые помогут не повредить имеющийся функционал сервиса при реализации новой функциональности.

Тесты декомпозированы на группы и вложенные классы. Например, тест на создание нового пользователя, проверяющий корректную обработку невалидного запроса с неверном формате номера телефона, располагается в файле `UserControllerTest.java` в одноименном классе и вложенном классе `CreateUser`.

Листинг 15. Метод для тестирования ответа системы на невалидный запрос создания нового пользователя.

```
@Test  
void createUserWithIncorrectPhone_gotException() throws Exception {  
    NewUserRequest request = new NewUserRequest(  

```

```

        "username", "useremail@mail.ru", "NOT_A_PHONE",
        "password");

        mvc.perform(post("/users")
            .content(mapper.writeValueAsString(request))
            .contentType(MediaType.APPLICATION_JSON)
            .accept(MediaType.APPLICATION_JSON)
            .andExpect(status().is5xxServerError()));
    }
}

```

3.2 Сервис дежурств

Ответственность данного сервиса — хранение и управление дежурствами пользователей.

3.2.1 Особенности реализации сервиса

Сервис является внутренним сервисом, что означает, что он используется другими сервисами системы, не взаимодействуя напрямую с UI. Коммуникация с сервисом происходит посредством RESTful API, использующийся при этом протокол — HTTP.

Для обработки HTTP запросов используется веб-сервер Tomcat (поставляется в составе Spring Boot[15]), низкоуровневый механизм сервлетов (аналогично — в составе Spring Boot[15]), а также написанный слой контроллера, каждый метод которого отвечает за определенный эндпоинт. В свою очередь каждый эндпоинт имеет контракт, специфицирующий, какие данные будут присланы для парсинга вместе с запросом и какие будут предоставлены в качестве результата.

В программу было добавлено логирование различных уровней, которое будет полезно для разбора нештатных ситуаций работы сервиса, мониторинга система, а также для разработки нового функционала.

3.2.2 Функционал сервиса

Сервис предоставляет следующий функционал:

Создать дежурство:

Листинг 16. Контракт создания дежурства.

```

@PostMapping
public Duty createDuty(@Valid @RequestBody CreateDutyRequest request)
{

```

```

        return dutyService.createDuty(request);
    }
}
@Data
public class CreateDutyRequest {
    @NotNull(message = "Необходимо специфицировать время начала дежурства")
    LocalDateTime start_time;

    @NotNull(message = "Интервал необходим для определения времени работы одного дежурного")
    Duration interval;

    @NotNull(message = "Должен быть назначен список дежурных")
    Long[] ids;
}
@Data
@Builder
public class Duty {
    private Long id;

    private LocalDateTime start_time;

    private Duration interval;

    private Long[] ids;
}

```

Изменить дежурство:

Листинг 17. Контракт на изменения дежурства.

```

@PutMapping
public Duty updateDuty(@Valid @RequestBody UpdateDutyRequest request)
{
    return dutyService.updateDuty(request);
}
@Data
public class UpdateDutyRequest {
    @NotNull

```

```

        Long id;

        LocalDateTime start_time;

        Duration interval;

        Long[] ids;
    }

```

Получить информацию о дежурстве по id:

Листинг 18. Контракт на получение информации о дежурстве по его идентификатору.

```

@GetMapping("/{dutyId}")
public DutyDto getDuty(@PathVariable Long dutyId) {
    return dutyService.getDuty(dutyId);
}

@Data
@Builder
public class DutyDto {
    Long id;

    LocalDateTime start_time;

    Duration interval;

    Long[] ids;

    Long currentDutyUserId;
}

```

Получить следующее дежурство пользователя по его id:

Листинг 19. Контракт на получение информации о следующем дежурстве пользователя идентификатору пользователя.

```

@GetMapping("/user/{userId}")
public UserDutyDto getNextUserDuty(@PathVariable Long userId) {
    return dutyService.getNextUserDuty(userId);
}

@Data

```

```
@Builder
public class UserDutyDto {
    LocalDateTime nextDutyDate;
    Duration duration;
    Duration intervalBetweenDuties;
}
```

Удалить дежурство по его id:

Листинг 20. Контракт удаление дежурства по его идентификатору.

```
@DeleteMapping("/{dutyId}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public Void deleteDuty(@PathVariable Long dutyId) {
    dutyService.deleteDuty(dutyId);
    return null;
}
```

3.2.3 Бизнес-логика

Основная бизнес-логика сервиса сосредоточена в классе `DutyService`. Он инкапсулирует в себе логику по обработке запросов — от принятия запроса в том виде, котором он пришел в предыдущий слой (слой контроллеров) до возврата результата.

Каждый соответствующий определенному типу запросу (и эндпоинту) метод содержит в себе логирование, препроцессинг запроса с возможным обращением к базе данных (например, если нужна требуется обновить данные пользователя, необходимо сперва проверить существование такого пользователя и вернуть клиенту HTTP-код 404). Далее следует главная обработка запроса с обращением к базе данных. После того, как база данных вернула результат, происходит его обработка (например, если вернулась пустая сущность — `null`, то клиенту аналогично нужно вернуть 404 HTTP-код). После этого необходимо сделать маппинг сущности, вернувшейся из базы данных, к сущности, которая будет отдана клиенту — то есть сделать преобразование объекта одного класса к объекту другого класса. Для этого в сервисе используются мапперы из библиотеки `mapstruct`, которая генерирует необходимые преобразования, избавляя программиста от написания рутинного кода.

Пример маппера для сервиса дежурств:

Листинг 21. Mapper DutyMapper.

```
@Mapper(componentModel = "spring", injectionStrategy =
InjectionStrategy.CONSTRUCTOR)
```



```
public interface DutyMapper {

    Duty toDuty(CreateDutyRequest request);
    Duty toDuty(UpdateDutyRequest request);

}
```

При неуспешной валидации запроса бросается исключение. Например, в запросе на обновление дежурства при ненайденном дежурстве бросается исключение `NotFoundException`.

Листинг 22. Метод обновления данных о дежурстве.

```
@Transactional
public Duty updateDuty(UpdateDutyRequest request) {
    log.info("updating duty: {}", request);
    Long dutyId = request.getId();
    Duty duty = dutyStorage.getDutyById(dutyId);
    if (duty == null) {
        throw new NotFoundException("Failed to find duty with id = "
+ dutyId);
    }

    if (request.getStart_time() == null) {
        request.setStart_time(duty.getStart_time());
    }
    if (request.getInterval() == null) {
        request.setInterval(duty.getInterval());
    }
    if (request.getIds() == null) {
        request.setIds(duty.getIds());
    }

    Duty dutyResult = mapper.toDuty(request);
    return dutyStorage.update(dutyResult);
}
```

3.2.4 Взаимодействие с базой данных

В качестве СУБД для сервиса был выбран PostgreSQL[26]. Взаимодействие с ним происходит через одноименный драйвер, инкапсулирующий взаимодействие с базой по

бинарному протоколу.

Для взаимодействия с базой данных на бизнес-уровне используется JDBC, использующий native SQL запросы.

За логику работы с базой данных отвечает класс DutyStorageDb, реализующий интерфейс DutyStorage.

Листинг 23. Интерфейс, предоставляющий методы с декларативным названием для обращения к БД.

```
public interface DutyStorage {  
    Duty save(Duty duty);  
  
    Duty update(Duty duty);  
  
    void delete(Long dutyId);  
  
    List<Duty> getUserDuties(Long userId);  
  
    List<Duty> getDutiesByIds(List<Long> dutiesIds);  
  
    Duty getDutyById(Long dutyId);  
}
```

Каждый метод выполняет один запрос к базе данных, выполняет парсинг результата и возвращает объект соответствующего контракту типа. Например, метод save(Duty duty) отвечает за сохранение дежурства в базу данных.

Листинг 24. Метод сохранения дежурства в базу данных.

```
@Override  
public Duty save(Duty duty) {  
    String sql = "INSERT INTO duties (start_time, interval, ids)  
VALUES (?, ?, ?)";  
  
    KeyHolder keyHolder = new GeneratedKeyHolder();  
    jdbcTemplate.update(  
        connection -> {  
            PreparedStatement ps =  
connection.prepareStatement(sql, new String[]{"id"});  
            ps.setObject(1, duty.getStart_time());  
            ps.setLong(2, duty.getInterval().toSeconds());  
            Array idsArray = connection.createArrayOf("bigint",  
duty.getIds());  
            ps.setArray(3, idsArray);  

```

```

        return ps;
    }, keyHolder);
    duty.setId(keyHolder.getKey().longValue());
    return duty;
}

```

Для создания таблицы при подключении к базе данных используется скрипт `schema.sql`, который удаляет старую таблицу и создает новую.

Листинг 25. Скрипт для инициализации базы данных сервиса дежурств.

```

DROP TABLE IF EXISTS duties;

CREATE TABLE duties (
    id BIGSERIAL PRIMARY KEY,
    start_time TIMESTAMP NOT NULL,
    interval BIGINT NOT NULL, -- seconds
    ids BIGINT[]
);

COMMENT ON TABLE duties IS 'Таблица дежурств';
-- Схема дежурств: Round Robin: при окончании смены одного
пользователя наступает смена следующего из ids;
--
-- При окончании смены последнего дежурного из ids
наступает смена первого из ids; И т.д.
COMMENT ON COLUMN duties.id IS 'Идентификатор дежурства';
COMMENT ON COLUMN duties.start_time IS 'Начало дежурства';
COMMENT ON COLUMN duties.interval IS 'Сколько длится дежурство для
одного пользователя';

```

3.2.5 Тестирование

Для данного сервиса были написаны unit-тесты с использованием библиотек JUnit5[27] и Mockito[28]. Они покрывают базовый функционал сервиса и по большей части выполняют роль регрессионных тестов, которые помогут не повредить имеющийся функционал сервиса при реализации новой функциональности.

Тесты декомпозированы на группы и вложенные классы. Например, тест на создание

нового дежурства, проверяющий корректную обработку валидного запроса, располагается в файле `DutiesControllerTest.java` в одноименном классе и вложенном классе `CreateDuty`.

Листинг 26. Метод для тестирования ответа системы на валидный запрос удаления дежурства.

```
@Test
void deleteDuty() throws Exception{

    doNothing().when(dutyService).deleteDuty(anyLong());

    mvc.perform(delete("/duties/" + duty.getId())
        .contentType(MediaType.APPLICATION_JSON)
        .accept(MediaType.APPLICATION_JSON))
        .andExpect(status().is2xxSuccessful());
}
```

3.3 Сервис уведомлений

Ответственность данного сервиса – управление уведомлениями, а также отправкам их по различным каналам связи, включающим смс и почту.

3.3.1 Особенности реализации сервиса

Сервис является внутренним сервисом, что означает, что он используется другими сервисами системы, не взаимодействуя напрямую с UI. Коммуникация с сервисом происходит посредством RESTful API, использующийся при этом протокол — HTTP.

Для обработки HTTP запросов используется веб-сервер Tomcat (поставляется в составе Spring Boot[15]), низкоуровневый механизм сервлетов (аналогично — в составе Spring Boot[15]), а также написанный слой контроллера, каждый метод которого отвечает за определенный эндпоинт. В свою очередь каждый эндпоинт имеет контракт, специфицирующий, какие данные будут присланы для парсинга вместе с запросом и какие будут предоставлены в качестве результата.

В программу было добавлено логирование различных уровней, которое будет полезно для разбора нештатных ситуаций работы сервиса, мониторинга система, а также для разработки нового функционала.

Ключевой же особенностью реализации данного сервиса является своя реализация логики по отправке уведомлений по различным каналам связи. При реализации был применен паттерн «Декоратор», который позволил создать гибкую систему классов,

абстрактных классов и интерфейсов для возможности добавления дополнительных каналов уведомлений.

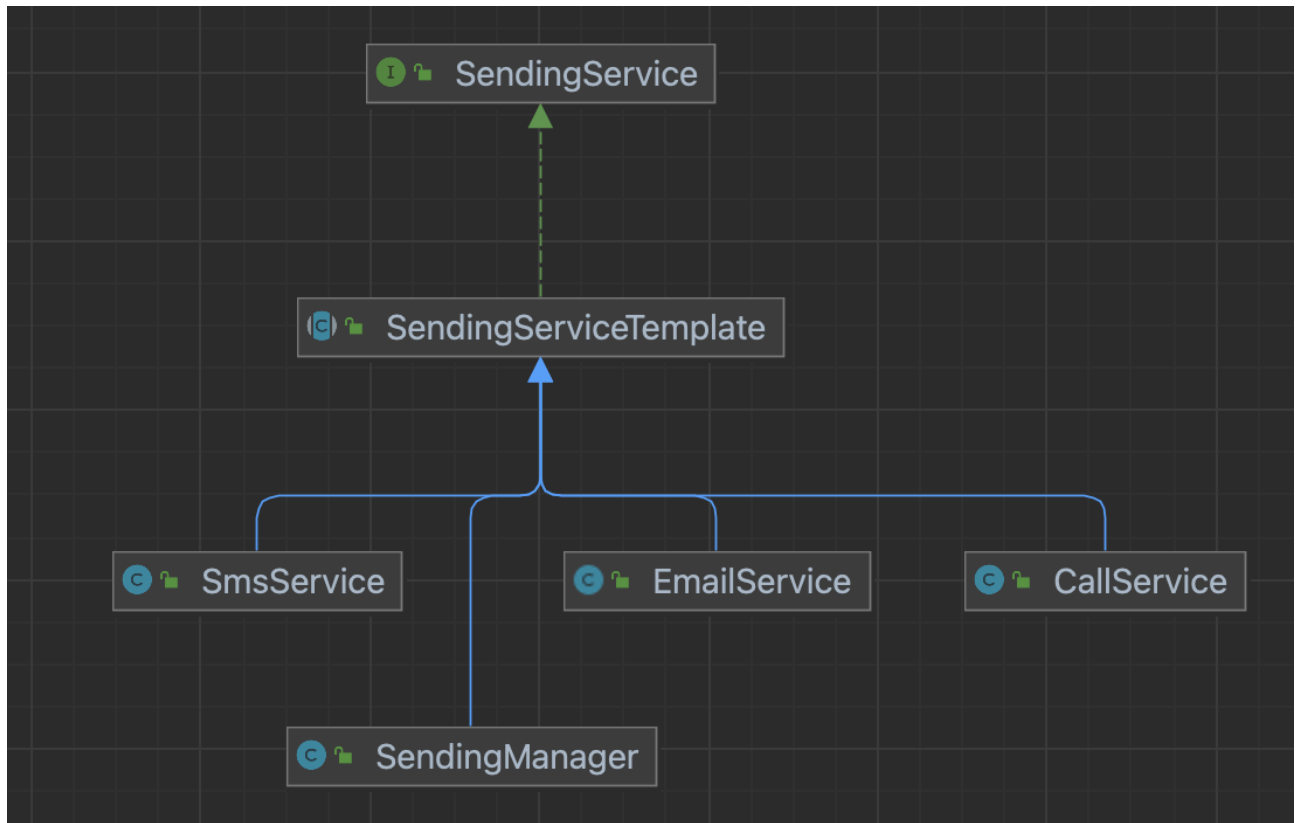


Рисунок 22. Структура классов, отвечающих за механизм отправки уведомлений.

Заметим, что каждый класс на схеме реализует интерфейс SendingService (реализация может быть как прямая, так и транзитивная), имеющий следующий контракт:

Листинг 27. Контракт интерфейса SendingService.

```
public interface SendingService {  
    void send(UserFullInfoDto receiver, String content);  
  
    String getType();  
}  
  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
public class UserFullInfoDto {  
    private Long id;  
  
    private String name;
```

```

private String email;

private String phone;

private String organization;

private String avatar;

private String about;
}

```

Также для упрощения конструирования «конечных» классов для непосредственной отправки уведомлений был создан шаблонный класс `SendingServiceTemplate`, имеющий свою реализацию метода `send(UserFullInfoDto receiver, String content)`.

Листинг 28. Контракт абстрактного класса `SendingServiceTemplate`.

```

@Slf4j
@AllArgsConstructor
@NoArgsConstructor
public abstract class SendingServiceTemplate implements
SendingService {
    @Getter
    private String type = "UNKNOWN";

    @Override
    public void send(UserFullInfoDto receiver, String content) {
        log.info("sending notification with content {} to {} via
sender of type = {}",
                content, receiver, this.getType());
    }
}

```

На основе данного шаблона были созданы «конечные реализации» сервисов-отправителей (`EmailService`, `SmsService`, `CallService`), а также класс-менеджер `ServiceManager`, отвечающий за управлений этими сервисами и позволяющий отправлять уведомление по нескольким каналам сразу. Классы-отправители довольно объемные, поэтому здесь будет приведена реализация только класса-менеджера.

Листинг 29. Класс-менеджер, отвечающий за отправку уведомлений по различным каналам связи.

```
@Slf4j
@Service
public class SendingManager extends SendingServiceTemplate {
    List<SendingService> sendingServices;

    @Autowired
    SendingManager(List<SendingService> sendingServices) {
        this.sendingServices = sendingServices;
    }

    @Override
    public void send(UserFullInfoDto receiver, String context) {
        log.info("sending notification to {}, content: {} via {}
senders", receiver, context, sendingServices.size());
        for (SendingService service : sendingServices) {
            service.send(receiver, context);
        }
    }
}
```

Для отправки уведомлений по почте используется пакет `org.springframework.mail` из экосистемы Spring[14]. Он использует более низкоуровневое API для отправки почтовых сообщений по протоколу SMTP (Simple Transfer Mail Protocol). Конфигурация для почтового взаимодействия описана в файлах `application-prod.yml` и `application-test.yml`. Первый используется в «боевом» окружении и содержит чувствительные данные, такие как пароли от используемых сторонних сервисов. Этот файл не индексируется системой контроля версий и может быть настроен исключительно вручную, используя соответствующий ему файл тестового окружения `application-test.yml`.

Для отправки уведомлений по смс используется фреймворк SMS Aero. Он предоставляет удобный API со всем необходимым функционалом.

SMS Aero отправляет SMS-сообщения по протоколу SMPP (Short Message Peer-to-Peer). Этот протокол используется для обмена текстовыми сообщениями между SMS-центрами и другими системами, такими как приложения и веб-сервисы.

Конфигурация, необходимая для настройки и использования фреймворка SMS Aero, также прописывается в application-prod.yml и application-test.yml файлах.

Листинг 30. Файл с конфигурацией для сервиса уведомлений для запуска приложения в тестовой среде.

```
spring:
  application:
    name: notifications-service
  datasource:
    url: jdbc:postgresql://localhost:6542/notification_db
    username: notification_user
    password: notification_pass
    driver-class-name: org.postgresql.Driver
  jpa:
    hibernate:
      ddl-auto: validate
    properties:
      hibernate:
        format_sql: true
    show-sql: true
  sql:
    init:
      mode: always
  mail:
    host: smtp.gmail.com
    port: 587
    username: argus.treker@gmail.com
    password: INSERT_PASSWORD_HERE
    properties:
      mail.smtp.auth: true
      mail.smtp.starttls.enable: true

logging:
  level:
    org.springframework.jdbc.core: DEBUG
    org.springframework.jdbc.datasource.DataSourceUtils: DEBUG

users:
  url: http://profiles-service:9090

server:
  port: 8080
  error:
    include-stacktrace: never

aero-sms:
  email: argus.treker@gmail.com
  key: INSERT_KEY_HERE
```



```
management:
  endpoints:
    web:
      exposure:
        include: health, info
```

3.3.2 Функционал сервиса

Сервис предоставляет следующий функционал:

Создать и зарегистрировать уведомление:

Листинг 31. Контракт создания и регистрации уведомления.

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Notification createNotification(@Valid @RequestBody
CreateNotificationRequest request) {
    return service.create(request);
}

@Data
public class CreateNotificationRequest {

    @NotNull
    private String type;

    @NotNull
    private String content;

    private String time_to_show;

    private Duration interval_to_repeat;

    @NotNull
    private Long userId;

    private Boolean immediately;
}

@Data
@Builder
public class Notification {
```

```

private Long id;
private String type;

@NotNull
@Max(254)
private String content;

private LocalDateTime time_to_show;

private Duration interval_to_repeat;

@NotNull
private Long userId;

private Boolean immediately;
}

```

Изменить уведомление и перерегистрировать его:

Листинг 32. Контракт изменения и перерегистрации уведомления.

```

@PutMapping
public Notification updateNotification(@Valid @RequestBody
UpdateNotificationRequest request) {
    return service.update(request);
}

@Data
public class UpdateNotificationRequest {
    private Long id;

    private String type;

    private String content;

    private LocalDateTime time_to_show;

    private Duration interval_to_repeat;

    private Long userId;
}

```

```

        private Boolean immediately;
    }

```

Удалить уведомление по его идентификатору:

Листинг 33. Контракт удаления уведомления.

```

@DeleteMapping("/{notificationId}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public Void deleteNotification(@PathVariable Long notificationId) {
    service.delete(notificationId);
    return null;
}

```

Удалить несколько уведомлений по их идентификаторам:

Листинг 34. Контракт удаления нескольких уведомлений.

```

@DeleteMapping
@ResponseStatus(HttpStatus.NO_CONTENT)
public Void deleteNotifications(@RequestBody List<Long>
notificationIds) {
    service.delete(notificationIds);
    return null;
}

```

Получить полную информацию об уведомлениях по их идентификаторам:

Листинг 35. Контракт получения всей информации об уведомлениях по их идентификаторам.

```

@GetMapping("/{notificationIds}")
public List<Notification> getNotificationsFullData(@PathVariable
List<Long> notificationIds) {
    return service.get(notificationIds);
}

```

3.3.3 Бизнес-логика

Основная бизнес-логика сервиса сосредоточена в классах `NotificationsService`, `NotificationsManager`, а также в классах подпакета `sendingServices`. `NotificationsService` инкапсулирует в себе логику по обработке запросов — от принятия запроса в том виде, котором он пришел в предыдущий слой (слой контроллеров) до возврата результата.

Каждый соответствующий определенному типу запросу (и эндпоинту) метод содержит в

себе логирование, препроцессинг запроса с возможным обращением к базе данных (например, если нужна требуется обновить данные пользователя, необходимо сперва проверить существование такого пользователя и вернуть HTTP-код 404 клиенту). Далее следует главная обработка запроса с обращением к базе данных. После того, как база данных вернула результат, происходит его обработка (например, если вернулась пустая сущность — null, то клиенту аналогично нужно вернуть 404 HTTP-код). После этого происходит маппинг сущности, вернувшейся из базы данных, к сущности, которая будет отдана клиенту — то есть сделать преобразование объекта одного класса к объекту другого класса. Для этого в сервисе используются мапперы из библиотеки mapstruct, которая генерирует необходимые преобразования, избавляя программиста от написания рутинного кода.

Пример маппера для сервиса уведомлений:

Листинг 36. Mapper DutyMapper.

```
@Mapper(componentModel = "spring", injectionStrategy =
InjectionStrategy.CONSTRUCTOR)
public interface NotificationsMapper {
    Notification toNotification(CreateNotificationRequest request);

    Notification toNotification(UpdateNotificationRequest request);
}
```

При неуспешной валидации запроса бросается исключение. Например, в запросе на удаление уведомления при ненайденном уведомлении бросается исключение NotFoundException.

Листинг 37. Метод удаления дежурства.

```
@Transactional
public void deleteDuty(Long dutyId) {
    log.info("deleting duty with id = {}", dutyId);
    Duty duty = dutyStorage.getDutyById(dutyId);
    if (duty == null) {
        throw new NotFoundException("Failed to find duty with id = "
+ dutyId);
    }
    dutyStorage.delete(dutyId);
}
```

Класс `NotificationsManager` отвечает за управление уведомлениями в контексте сервиса (не базы данных). Он позволяет:

- зарегистрировать уведомление (это действие будет переадресовано методу обновления уведомления с целью предотвращения утечки ресурсов — в частности, процессорного времени,— например, при уже имеющемся таком уведомлении новое будет просто создано и запущено, а старое будет удалено только из хранилища, но поток, обрабатывающий его, будет продолжать жить и будет тратить ресурсы ЦП + также это может повлиять на максимально допустимое число уведомлений — так как каждое уведомление обрабатывается в отдельном потоке, а у ОС, как правило, есть ограничение на максимальное число потоков одного процесса — то есть одной запущенной программы.
- обновить уведомление (после чего начнется его обработка — см. ниже)
- обработка уведомления (создается отдельный поток-обработчик, который будет отправлять уведомление в определенное время, которое определяется параметрами уведомления, такими как, время первой отправки уведомления, интервал отправки и флаг мгновенной отправки уведомления) (этот метод является приватным и не может быть вызван никем извне класса)
- удаление уведомления (происходит в 2 этапа: сначала прерывается поток, обрабатывающий данное уведомление, далее информация об уведомлении стирается из локального хранилища — словаря)
- удаление нескольких уведомлений (для каждого уведомления вызывается предыдущий метод)
- получение пользователя по `id` (этот метод является приватным и не может быть вызван никем извне класса)

Один из самых трудных алгоритмов, реализованных в данном классе — это алгоритм процессинга (обработки) уведомления. Он принимает на вход объект класса `Notification` и отвечает за отправку уведомления. Запускается этот метод в отдельном потоке. Одно уведомление — один поток. Сама обработка уведомления происходит в бесконечном цикле `while`. Прервать работу потока можно, пошлав ему сигнал `interrupted` (что делает метод удаления уведомления) или любой другой сигнал завершения потока (например, извне). Его работа состоит из следующих этапов:

1. На основе типа уведомления определяем сервис, который будет использоваться для отправки.
2. Получаем необходимую информацию о получателе уведомления по его `id`, передаем ее для отправки. На этом этапе делается HTTP-запрос в сервис профилей пользователей.

Если нужный пользователь не найден, завершаем обработку уведомления и логируем ошибку.

3. Далее проверяем, первая ли итерация обработки уведомления. Это особенность реализации самого механизма отправки.
 1. Если это первая итерация, то проверяем флаг мгновенной отправки уведомления.
 1. Если он выставлен, отправляем уведомление и «засыпаем» на интервал между отправками.
 2. Если нет, то ждем, пока наступит момент первой отправки уведомления. Если такой момент уже наступил, то отправляем уведомление и «засыпаем» на интервал между отправками.
 2. Если не первая, то отправляем уведомление и «засыпаем» на интервал между отправками.

После первой итерации цикла фиксируем флаг первой итерации как false. Также логируем отправку.

3.3.4 Взаимодействие с базой данных

В качестве СУБД для сервиса был выбран PostgreSQL[26]. Взаимодействие с ним происходит через одноименный драйвер, инкапсулирующий взаимодействие с базой по бинарному протоколу.

Для взаимодействия с базой данных на бизнес-уровне используется JDBC, использующий native SQL запросы. За логику работы с базой данных отвечает класс NotificationStorageDb, реализующий интерфейс NotificationsStorage. Каждый метод выполняет один запрос к базе данных, выполняет парсинг результата и возвращает объект соответствующего контракту типа. Например, метод findById отвечает за получение дежурства из базу данных по его идентификатору.

Листинг 38. Метод получения дежурства из базы данных по идентификатору дежурства.

```
@Override
public Notification findById(Long notificationId) {
    String sql = "SELECT id, type, content, time_to_show,
interval_to_repeat, user_id, immediately " +
        "FROM notifications WHERE id = ? LIMIT 1";
    List<Notification> notifications = jdbcTemplate.query(sql,
        (ResultSet rs, int rowNum) ->
Notification.builder()
```

```

        .id(rs.getLong("id"))
        .type(rs.getString("type"))
        .content(rs.getString("content"))
        .time_to_show(rs.getTimestamp("time_t
o_show").toLocalDateTime())

        .interval_to_repeat(Duration
.ofSeconds(rs.getLong("interval_to_repeat")))
        .userId(rs.getLong("user_id"))
        .immediately(rs.getBoolean("immediate
ly"))

        .build(), notificationId);

    if (notifications.isEmpty()) return null;
    return notifications.getFirst();
}

```

Для создания таблицы при подключении к базе данных используется скрипт `schema.sql`, который удаляет старую таблицу и создает новую.

Листинг 39. Скрипт для инициализации базы данных сервиса уведомлений.

```

DROP TABLE IF EXISTS notifications;

CREATE TABLE notifications (
    id BIGSERIAL PRIMARY KEY,
    type VARCHAR(100) NOT NULL,
    content TEXT NOT NULL,
    time_to_show TIMESTAMP,
    interval_to_repeat BIGINT,  -- seconds
    user_id BIGINT NOT NULL,
    immediately BOOLEAN DEFAULT false

    CONSTRAINT type_values CHECK (type IN ('sms', 'email', 'call')),
    CONSTRAINT check_immediately_timeToShow CHECK (
        NOT (immediately = false AND time_to_show IS NULL)  -- иначе
уведомление никогда не будет отправлено
    )
);

```

```

COMMENT ON TABLE notifications IS 'Таблица уведомлений';
COMMENT ON COLUMN notifications.id IS 'Идентификатор уведомления';
COMMENT ON COLUMN notifications.type IS 'Тип уведомления';
COMMENT ON COLUMN notifications.content IS 'Содержание уведомления';
COMMENT ON COLUMN notifications.time_to_show IS 'Дата и время показа
первого уведомления';
COMMENT ON COLUMN notifications.interval_to_repeat IS 'Интервал,
через который уведомление будет повторено';
COMMENT ON COLUMN notifications.user_id IS 'Идентификатор
пользователя, которому будет отправлено уведомление';
COMMENT ON COLUMN notifications.immediately IS 'Если true, то
уведомление должно быть отправлено сейчас же';

```

3.3.5 Тестирование

Для данного сервиса были написаны unit-тесты с использованием библиотек JUnit5[27] и Mockito[28]. Они покрывают базовый функционал сервиса и по большей части выполняют роль регрессионных тестов, которые помогут не повредить имеющийся функционал сервиса при реализации новой функциональности.

Тесты декомпозированы на группы и вложенные классы. Например, тест на создание нового уведомления, проверяющий корректную обработку валидного запроса, располагается в файле NotificationsControllerTest.java в одноименном классе и вложенном классе CreateNotification.

Листинг 40. Метод для тестирования ответа системы на валидный запрос создания нового уведомления.

```

@Test
void createNotificationSuccessful() throws Exception {
    CreateNotificationRequest request = new
CreateNotificationRequest(
        "sms",
        "This is a notification content",
        LocalDateTime.now().toString(),
        Duration.ofMinutes(5),
        1L,
        false);

    Mockito
        .when(notificationsService.create(request))
        .thenReturn(notification);

    mvc.perform(post("/notifications")
        .content(mapper.writeValueAsString(request))

```



```

        .contentType(MediaType.APPLICATION_JSON)
        .accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isCreated())
        .andExpect(jsonPath("$.id",
equalTo(notification.getId()), Long.class))
        .andExpect(jsonPath("$.type",
equalTo(notification.getType()))))
        .andExpect(jsonPath("$.content",
equalTo(notification.getContent()))))
        .andExpect(jsonPath("$.time_to_show",
equalTo(notification.getTime_to_show().toString()))))
        .andExpect(jsonPath("$.interval_to_repeat",
equalTo(notification.getInterval_to_repeat().toString()))))
        .andExpect(jsonPath("$.userId",
equalTo(notification.getUserId()), Long.class))
        .andExpect(jsonPath("$.immediately",
equalTo(notification.getImmediately())));
    }

```

3.4 Библиотека Common

Разработанные сервисы имеют перекрестные зависимости. Чтобы избежать дублирования кода и при этом обеспечить корректную упаковку сервисов в jar-файлы, было принято решение разработать общий модуль с общими зависимостями, который будет также собираться в jar файл и подключаться в сервисах.

Данный модуль содержит 2 класса:

- `NotFoundException` — проприетарный класс исключений, которые должны выбрасываться, если та или иная сущность, необходимая для дальнейшего процессинга, не была найдена.
- `UserFullInfoDto` — класс, использующийся в модулях `profiles` и `notifications`.

3.5 Система сборки

В качестве системы сборки была выбрана декларативная система Maven. Она отличается относительной простотой настройки, использует xml-файлы конфигурации.

Данный проект является многомодульным проектом и имеет следующую структуру.

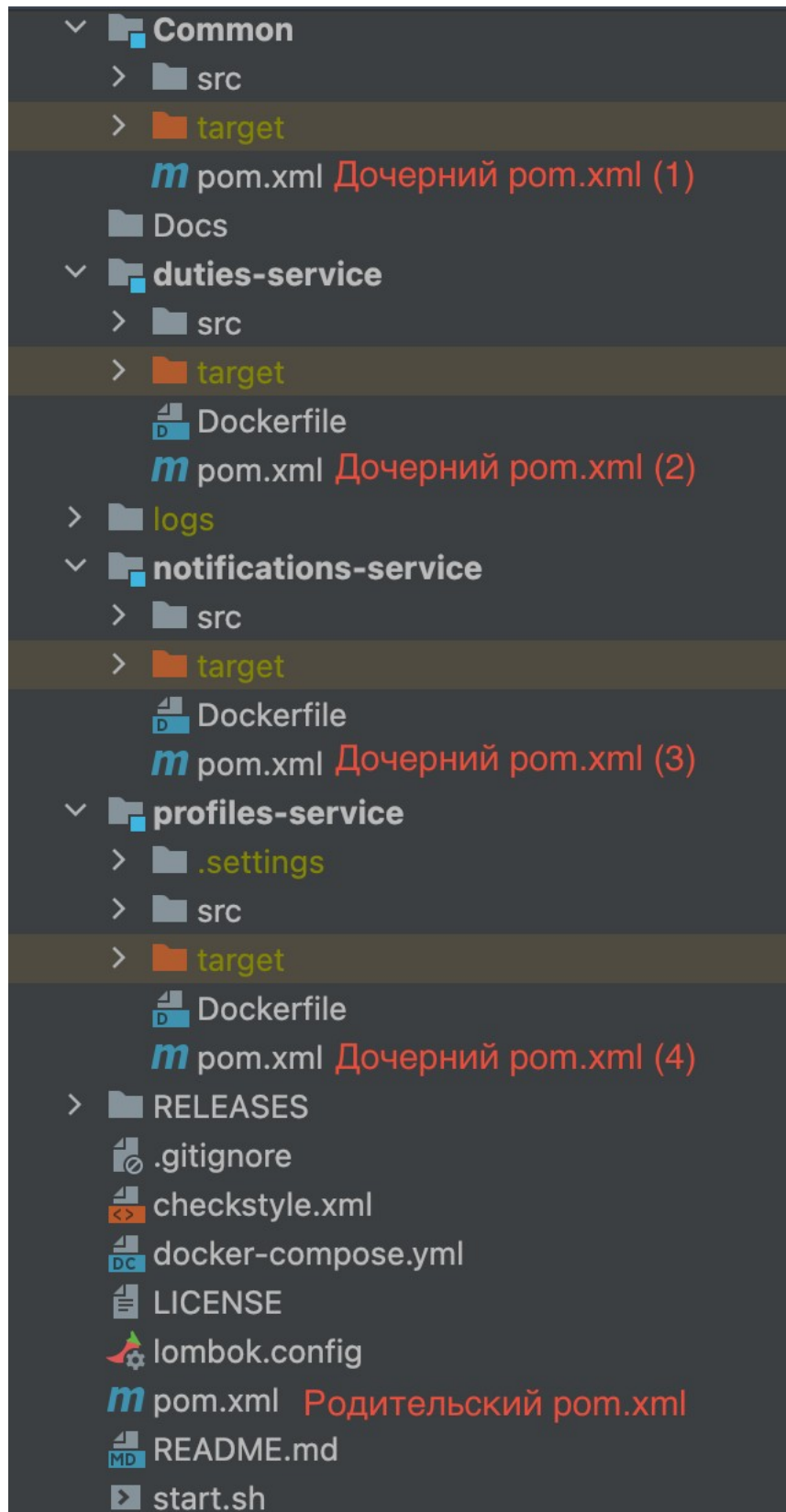


Рисунок 23. Структура проекта.

```

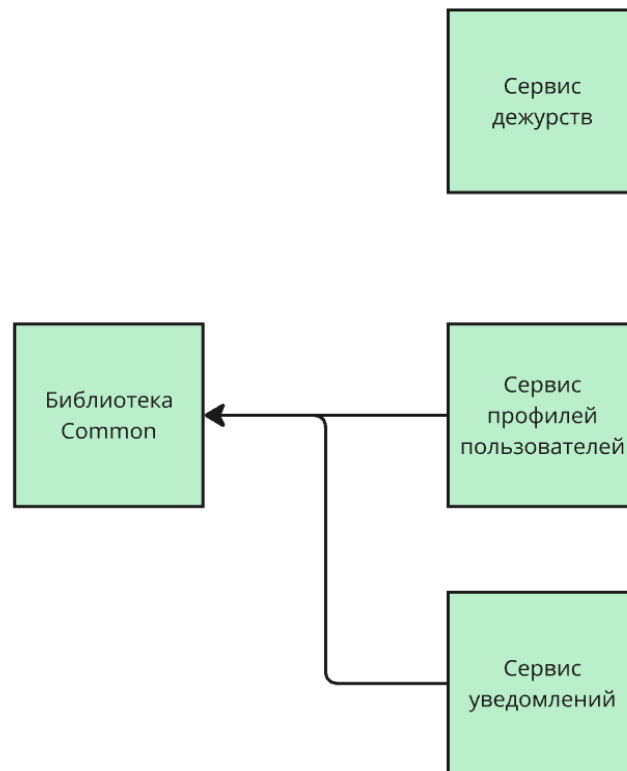
<modules>
  <module>Common</module>
  <module>profiles-service</module>
  <module>duties-service</module>
  <module>notifications-service</module>
</modules>

```

Рисунок 24.

Спецификация дочерних модулей в родительском pom.xml.

Каждый сервис собирается в отдельный jar-файл и может далее быть запущен независимо от остальных.



Зависимости между модулями проекта.

Сплошная стрелка означает прямую зависимость (влинкованный jar-файл).
Прерываемая стрелка - транзитивную.

Рисунок 25. Зависимости между модулями проекта. Сплошная стрелка означает прямую зависимость (влинкованный jar-файл), прерываемая стрелка - транзитивную.

В проекте также присутствует сборочный скрипт start.sh, который:

1. Удаляет предыдущие сборки проекта
2. Компилирует модули проекта в jar-файлы
3. Запускает скрипт docker-compose.yml для разворачивания сервисов в докер-контейнерах и настройки их взаимодействия.

```
▶ # script builds jars from source & deploys them into virtual containers
  mvn clean package
  docker compose up --build
```

Рисунок 26. Скрипт shart.sh

3.6 Документация

Документация для проекта представляет из себя спецификацию REST API в виде .yaml файлов, которые могут быть импортированы в Swagger[30] для наглядного отображения. Файлы спецификации были автоматически сгенерированы с помощью зависимости springdoc-openapi-starter-webmvc-ui и были помещены в директорию API_DOCS корня репозитория.

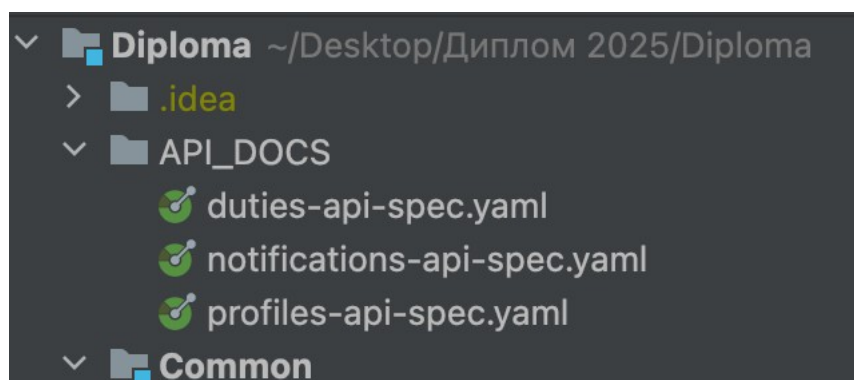


Рисунок 27. REST API документация для каждого сервиса.

3.7 Платформа разработки

Серверная часть «Argus» разрабатывается несколькими разработчиками совместно, однако благодаря слабой связанности компонентов системы разработчики вольны выбирать свой язык программирования и стек технологий, если их конечный продукт отвечает требованиям технического задания.

3.8 Выводы по главе

В данной главе описано функциональное назначение каждого из разрабатываемых сервисов, приведены подробные контракты API разрабатываемых методов и способы взаимодействия с базами данных сервисов, которые удовлетворяют функциональным требованиям системы, описаны особенности реализации, тестирования, сборки проектов и их документации.

Также было уделено внимание описанию библиотек, используемых в данной системе: было приведено описание их назначения и настройки конфигурации для их подключения и использования.

Заключение

Система управления инцидентами в области обеспечения безопасности обладает высоким спросом и неограниченным потенциалом: благодаря слабой связанности компонентов, а также применению паттернов проектирования, есть возможность добавлять в систему новые интеграции. Например, датчик шума или датчик проникновения в помещение.

Результатом данной работы является часть общей системы, состоящей из сервисов профилей пользователей, дежурств и уведомлений, которые представляют важные аспекты функционала.

В рамках исследования были проведены анализы предметной области и аналогов, на основании которых были выбраны тот функционал и те решения, которые были включены в итоговую систему. В начале проекта была тщательно проработана архитектура всей системы, отвечающая требованиям разрабатываемого приложения, а также спроектирована каждая из подсистем в отдельности, что позволило не вносить критических правок во время процесса разработки.

Отдельное внимание было уделено выбору технологий и инструментов для реализации: языков программирования, хранилищ данных, фреймворков и библиотек. Это позволило выбрать оптимальный технологический стек для реализации данного проекта, подходящий под условия разработки промышленного приложения с высокими требованиями к надежности и производительности.

По окончании работы система отвечает всем функциональным требованиям, зафиксированным в начале проекта. Дальнейшая поддержка и развитие проекта будут продолжаться (например, подключить к системе сервис логирования и мониторинга Grafana[34] и Prometheus[35], заменить Docker-compose[19] на более продвинутый аналог Kubernetes[36]), есть планы по выведению продукта на коммерческие рельсы.

Список использованных источников

1. Microsoft Project. [Электронный ресурс] // URL: <https://www.workzone.com/blog/microsoft-project-history/> (дата обращения: 02.02.2025, режим доступа: свободный)
2. BugZilla. [Электронный ресурс] //URL: <https://www.bugzilla.org/about/> (дата обращения: 02.02.2025, режим доступа: свободный).
3. Jira. [Электронный ресурс] //URL: <https://community.atlassian.com/t5/Jira-articles/Evolution-of-Jira-Design-Part-1/ba-p/1304571> (дата обращения: 02.02.2025, режим доступа: свободный).
4. Yandex Treker. [Электронный ресурс] //URL: https://www.tadviser.ru/index.php/Продукт:Яндекс.Трекер_%28Yandex_Tracker%29 (дата обращения: 02.02.2025, режим доступа: свободный).
5. Security Incident Response (SIR) [Электронный ресурс] //URL: <https://www.servicenow.com/products/security-incident-response.html#benefits> (дата обращения: 02.02.2025, режим доступа: свободный).
6. Atlassian [Электронный ресурс] //URL: <https://www.atlassian.com> (дата обращения: 02.02.2025, режим доступа: свободный).
7. Confluence [Электронный ресурс] //URL: <https://www.atlassian.com/software/confluence> (дата обращения: 02.02.2025, режим доступа: свободный).
8. BitBucket [Электронный ресурс] //URL: <https://bitbucket.org/product/> (дата обращения: 02.02.2025, режим доступа: свободный).
9. Slack [Электронный ресурс] //URL: <https://slack.com> (дата обращения: 02.02.2025, режим доступа: свободный).
10. SIEM (Security Information and Event Management) [Электронный ресурс] //URL: <https://www.securityvision.ru/blog/siem-cto-eto-i-zachem-nuzhno/> (дата обращения: 02.02.2025, режим доступа: свободный).
11. ServiceNow [Электронный ресурс] //URL: <https://www.servicenow.com> (дата обращения: 02.02.2025, режим доступа: свободный).
12. Java [Электронный ресурс] //URL: <https://www.java.com/ru/> (дата обращения: 02.02.2025, режим доступа: свободный).

13. JVM [Электронный ресурс] //URL: https://www.nic.ru/help/jvm-chto-eto-kak-ustroena-virtualnaya-mashina-java_11250.html (дата обращения: 02.02.2025, режим доступа: свободный).
14. Spring [Электронный ресурс] //URL: <https://spring.io> (дата обращения: 02.02.2025, режим доступа: свободный).
15. Spring Boot [Электронный ресурс] //URL: <https://spring.io/projects/spring-boot> (дата обращения: 02.02.2025, режим доступа: свободный).
16. C++ [Электронный ресурс] //URL: <https://blog.skillfactory.ru/c-plus-komu-i-dlya-chegonuzhen-yazik/> (дата обращения: 02.02.2025, режим доступа: свободный).
17. Си [Электронный ресурс] //URL: [https://znanierussia.ru/articles/Си_\(язык_программирования\)](https://znanierussia.ru/articles/Си_(язык_программирования)) (дата обращения: 02.02.2025, режим доступа: свободный).
18. Docker [Электронный ресурс] //URL: <https://www.docker.com> (дата обращения: 02.02.2025, режим доступа: свободный).
19. Docker-compose [Электронный ресурс] //URL: <https://docs.docker.com/compose/> (дата обращения: 02.02.2025, режим доступа: свободный).
20. Сбер [Электронный ресурс] //URL: <https://rabota.sber.ru/search/?query=java> (дата обращения: 02.02.2025, режим доступа: свободный).
21. Тинькофф [Электронный ресурс] //URL: <https://education.tbank.ru/start/java/> (дата обращения: 02.02.2025, режим доступа: свободный).
22. ВТБ [Электронный ресурс] //URL: <https://rabota-vtb.ru/vacancy/610ad63f3f4eb50001c5c507?city=moskva> (дата обращения: 02.02.2025, режим доступа: свободный).
23. Яндекс [Электронный ресурс] //URL: <https://yandex.ru/jobs/vacancies/javarazrabotchik-v-market-3762> (дата обращения: 02.02.2025, режим доступа: свободный).
24. МТС [Электронный ресурс] //URL: <https://job.mtsbank.ru/vacancies/moskva/razrabotchik-java-mikroservisy-i-spr--1007375> (дата обращения: 02.02.2025, режим доступа: свободный).
25. Билайн [Электронный ресурс] // URL: <https://job.beeline.ru/job-specialists/moskva/it-i-tehnicheskie-funktsii/> (дата обращения: 02.02.2025, режим доступа: свободный).
26. PostgreSQL [Электронный ресурс] // URL: <https://www.postgresql.org> (дата обращения: 02.02.2025, режим доступа: свободный).
27. JUnit [Электронный ресурс] // URL: <https://junit.org/junit5/> (дата обращения: 02.02.2025, режим доступа: свободный).

28. Mockito [Электронный ресурс] // URL: <https://site.mockito.org> (дата обращения: 02.02.2025, режим доступа: свободный).
29. ORM Hibernate [Электронный ресурс] // URL: <https://hibernate.org> (дата обращения: 02.02.2025, режим доступа: свободный).
30. Swagger [Электронный ресурс] // URL: <https://swagger.io> (дата обращения: 02.02.2025, режим доступа: свободный).
31. OpenAPI [Электронный ресурс] // URL: <https://www.openapis.org> (дата обращения: 02.02.2025, режим доступа: свободный).
32. MySQL [Электронный ресурс] // URL: <https://www.mysql.com> (дата обращения: 02.02.2025, режим доступа: свободный).
33. Oracle [Электронный ресурс] // URL: <https://www.oracle.com/database/> (дата обращения: 02.02.2025, режим доступа: свободный).
34. Grafana [Электронный ресурс] // URL: <https://grafana.com> (дата обращения: 02.02.2025, режим доступа: свободный).
35. Prometheus [Электронный ресурс] // URL: <https://prometheus.io> (дата обращения: 02.02.2025, режим доступа: свободный).
36. Kubernetes [Электронный ресурс] // URL: <https://kubernetes.io> (дата обращения: 02.02.2025, режим доступа: свободный).

ПРИЛОЖЕНИЕ

REST API спецификация для разрабатываемых сервисов:

Группа требований	Формулировка требования	Требования к конечной точке API
Сервис управления профилями пользователей	Создать нового пользователя	<p>Относительный url: /users</p> <p>Метод запроса: POST</p> <p>Параметры:</p> <p>-</p> <p>Тело запроса:</p> <ul style="list-style-type: none"> • <i>name</i> — имя пользователя • <i>email</i> — почта • <i>phone</i> — телефон • <i>password</i> — пароль <p>Ограничения на входные данные:</p> <ul style="list-style-type: none"> • name: <ul style="list-style-type: none"> ◦ не пусто ◦ длина: от 2 до 250 символов включительно • email: <ul style="list-style-type: none"> ◦ присутствует ◦ соответствие формату почты ◦ длина: от 6 до 100 символов включительно ◦ уникальность • phone: <ul style="list-style-type: none"> ◦ присутствует ◦ соответствует формату телефона ◦ уникальность • password: <ul style="list-style-type: none"> ◦ не пусто ◦ длина: от 8 до 40 символов <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> • Status code: 201 при успешном выполнении запроса, 400/409 при некорректных входных данных, 500 — любая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) • Response body: информация в формате JSON: <ol style="list-style-type: none"> 1. id

		2. name 3. email 4. phone
	Редактировать пользователя	<p>Относительный url: /users</p> <p>Метод запроса: PUT</p> <p>Параметры:</p> <p>-</p> <p>Тело запроса:</p> <ul style="list-style-type: none"> • <i>id</i> – идентификатор пользователя • <i>name</i> – новое имя • <i>email</i> — новая почта • <i>phone</i> — новый телефон • <i>password</i> — новый пароль пользователя • <i>organization</i> — новая компания, где работает пользователь • <i>avatar</i> — url-адрес новой аватарки пользователя • <i>about</i> — новое описание пользователя <p>При этом каждое поле, кроме <i>id</i>, может отсутствовать — в таком случае данное поле сохранит свое старое значение.</p> <p>Ограничения на входные данные:</p> <ul style="list-style-type: none"> • <i>id</i>: <ul style="list-style-type: none"> ◦ присутствует ◦ ≥ 0 • <i>email</i>: <ul style="list-style-type: none"> ◦ уникальность • <i>телефон</i>: <ul style="list-style-type: none"> ◦ уникальность <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> • Status code: 200 при успешном выполнении запроса, 404, если пользователь с текущим <i>id</i> не найден, 400/409 при некорректных входных данных, 500 — любая другая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) • Response body: информация в формате JSON: <ul style="list-style-type: none"> ◦ <i>id</i>
	Получить данные о пользователе	<p>Относительный url: /users/info</p> <p>Метод запроса: GET</p>

		<p>Параметры:</p> <p>-</p> <p>Переменные пути:</p> <p><i>id</i> — идентификатор пользователя</p> <p>Тело запроса:</p> <p>Нет</p> <p>Ограничения на входные данные:</p> <ul style="list-style-type: none"> • id: <ul style="list-style-type: none"> ◦ > 0 <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> • Status code: 200 при найденном пользователе, 400 при некорректных входных данных, 404 при отсутствии пользователя, 500 — любая другая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) • Response body: информация о пользователе в формате JSON: <ul style="list-style-type: none"> • id • name • email • phone • organization • avatar • about
	Найти пользователей по имени	<p>Относительный url: /users</p> <p>Метод запроса: GET</p> <p>Параметры:</p> <ul style="list-style-type: none"> • <i>name</i> — имя пользователя <p>Тело запроса:</p> <p>Нет</p> <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> • Status code: 200 при успешном запросе, 404 — если ни один пользователь не был найден, 500 — любая другая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) • Response body: информация в формате JSON:

		<ul style="list-style-type: none"> массив <code>ids</code> пользователей с таким именем
	По списку <code>id</code> вернуть <code>id</code> пользователей, которых НЕ существует	<p>Относительный url: <code>/users/notexisting</code> Метод запроса: GET</p> <p>Параметры:</p> <p>- <code>ids</code> – <code>id</code> пользователей</p> <p>Переменные пути:</p> <p>-</p> <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> Status code: 200 при успешном запросе, 404 — если ни один пользователь не был найден, 500 — любая другая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) Response body: информация в формате JSON: <ul style="list-style-type: none"> <code>id</code> пользователей, которых не существует (из тех <code>id</code>, которые поступили в запрос)
	Получить <code>id</code> пользователя по <code>email</code> и паролю	<p>Относительный url: <code>/users/userId</code> Метод запроса: GET</p> <p>Параметры:</p> <p><code>email</code> — имя пользователя <code>password</code> - пароль</p> <p>Тело запроса:</p> <p>Нет</p> <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> Status code: 200 при найденном пользователе, 404 при отсутствии пользователя с заданными параметрами, 500 — любая другая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) Response body: информация в формате JSON: <ul style="list-style-type: none"> <code>id</code> пользователя
	Получить	<p>Относительный url: <code>/users/info</code> Метод запроса: GET</p>

	<p>частичную информацию о пользователях</p>	<p>Параметры:</p> <p>- ids: массив id пользователей</p> <p>Тело запроса:</p> <p>-</p> <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> • Status code: 200 при успешном запросе, 404 — если ни один пользователь не был найден, 500 — любая другая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) • Response body: <ul style="list-style-type: none"> ◦ массив: <ul style="list-style-type: none"> ▪ id ▪ name ▪ avatar
	<p>Получить информацию обо всех существующих пользователях</p>	<p>Относительный url: /users/all Метод запроса: GET</p> <p>Параметры:</p> <p>-</p> <p>Тело запроса:</p> <p>-</p> <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> • Status code: 200 при успешном запросе, 404 — если ни один пользователь не был найден, 500 — любая другая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) • Response body: <ul style="list-style-type: none"> ◦ массив: <ul style="list-style-type: none"> ▪ id ▪ name ▪ email ▪ phone ▪ organization ▪ avatar ▪ about
Сервис отправки уведомлений	<p>Создать и зарегистрировать уведомление</p>	<p>Относительный url: /notifications Метод запроса: POST</p> <p>Параметры:</p>

		<p>-</p> <p>Тело запроса:</p> <ul style="list-style-type: none"> • <i>type</i> – тип уведомления (эл.письмо / смс) • <i>content</i> - содержание уведомления • <i>time_to_show</i> – когда уведомление должно показываться • <i>interval_to_repeat</i> — интервал повтора уведомления • <i>user_id</i> — на кого назначено уведомление • <i>immediately</i> — флаг мгновенной доставки уведомления <p>Ограничения на входные данные:</p> <ul style="list-style-type: none"> • <i>type</i>: <ul style="list-style-type: none"> ◦ присутствует ◦ допустимые значения: «sms» и «email» • <i>content</i>: <ul style="list-style-type: none"> ◦ присутствует • <i>userId</i>: <ul style="list-style-type: none"> ◦ присутствует • <i>time_to_show</i> (если присутствует): <ul style="list-style-type: none"> ◦ формат: "yyyy-MM-dd'T'HH:mm:ss" • недопустимо, чтобы <i>immediately</i> = false И <i>time_to_show</i> = null • <i>interval_to_repeat</i> (если присутствует): <ul style="list-style-type: none"> ◦ формат: PnYnMnDTnHnMnS <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> • Status code: 201 при успешном выполнении запроса, 400/409 для некорректных входных данных, 500 — любая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) • Response body: информация в формате JSON: <ul style="list-style-type: none"> ◦ id созданного уведомления ◦ все, что в теле запроса (см. выше)
	<p>Редактировать уведомление</p>	<p>Относительный url: /notifications</p> <p>Метод запроса: PUT</p> <p>Параметры:</p> <p>-</p> <p>Тело запроса:</p> <ul style="list-style-type: none"> • <i>type</i> – тип уведомления (эл.письмо / смс)

		<ul style="list-style-type: none"> • <i>content</i> - содержание уведомления • <i>time_to_show</i> – когда уведомление должно показываться • <i>interval_to_repeat</i> — интервал повтора уведомления • <i>user_id</i> — на кого назначено уведомление • <i>immediately</i> — нужно ли доставить уведомление мгновенно <p>Ограничения на входные данные:</p> <ul style="list-style-type: none"> • <i>type</i>: <ul style="list-style-type: none"> ◦ допустимые значения: «sms» и «email» (если присутствует) • <i>time_to_show</i> (если присутствует): <ul style="list-style-type: none"> ◦ формат: "уууу-ММ-дд"Т"НН:мм:сс" • недопустимо, чтобы <i>immediately</i> = false И <i>time_to_show</i> = null <p>При этом каждое поле, кроме <i>id</i>, может отсутствовать — в таком случае данное поле сохранит свое старое значение.</p> <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> • Status code: 200 при успешном выполнении запроса, 400/409 для некорректных входных данных, 500 — любая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) • Response body: информация в формате JSON: все поля обновленного уведомления
	Удалить уведомление	<p>Относительный url: /notifications</p> <p>Метод запроса: DELETE</p> <p>Переменная пути:</p> <ul style="list-style-type: none"> • <i>notificationId</i> – id уведомления <p>Тело запроса:</p> <p>Нет</p> <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> • Status code: 204 при успешном выполнении запроса, 404 — если уведомление не найдено, 500 — любая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) • Response body: пустое
	Удалить уведомления	<p>Относительный url: /notifications</p> <p>Метод запроса: DELETE</p>

		<p>Параметры:</p> <p>-</p> <p>Тело запроса:</p> <p>notificationIds — id уведомлений</p> <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> • Status code: 204 при успешном выполнении запроса, 404 — если ни одно из уведомлений не найдено, 500 — любая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) • Response body: пустое
	Получить информацию по уведомлениям по их id	<p>Относительный url: /notifications</p> <p>Метод запроса: GET</p> <p>Переменная пути:</p> <p>notificationIds — id уведомлений</p> <p>Тело запроса:</p> <p>-</p> <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> • Status code: 200 при успешном запросе, 404 — если ни одно уведомление не было найдено, 500 — любая другая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) • Response body: массив с полной информацией об уведомлениях

Сервис дежурств	Создать дежурство для пользователя	<p>Относительный url: /duties</p> <p>Метод запроса: POST</p> <p>Параметры:</p> <p>-</p> <p>Тело запроса:</p> <ul style="list-style-type: none"> • <i>start_time</i> — начало дежурства • <i>name</i> — название дежурства • <i>interval</i> — продолжительность дежурства (1-ой итерации дежурства) • <i>ids</i> — (упорядоченный) массив id дежурных (дежурят по очереди, по завершении дежурства)
-----------------	------------------------------------	---

		<p>последнего в списке наступает дежурство 1-ого)</p> <p>Ограничения на входные данные:</p> <ul style="list-style-type: none"> • start_time: <ul style="list-style-type: none"> ◦ должно присутствовать ◦ формат: уууу-ММ-дд"Т"НН:мм:ss • name: <ul style="list-style-type: none"> ◦ не пусто • interval: <ul style="list-style-type: none"> ◦ должен присутствовать ◦ формат: PnYnMnDTnHnMnS <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> • Status code: 201 при успешном запросе, 400/409 при некорректных входных данных, 500 — любая другая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) • Response body: <ul style="list-style-type: none"> ◦ id дежурства ◦ name ◦ start_time ◦ interval ◦ ids
	Получить следующее дежурство пользователя	<p>Относительный url: /duties/user</p> <p>Метод запроса: GET</p> <p>Переменная пути:</p> <p><i>userId</i> — идентификатор пользователя</p> <p>Тело запроса:</p> <p>Нет</p> <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> • Status code: 200 при успешном выполнении запроса, 404, если пользователь не найден, 500 — любая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) • Response body: информация о дежурстве пользователя в формате JSON: <ul style="list-style-type: none"> • дата его следующего дежурства • продолжительность дежурства • периодичность (через сколько он снова становится дежурным)

<p>Редактировать дежурство</p>		<p>Относительный url: /duties Метод запроса: PUT</p> <p>Параметры:</p> <p>-</p> <p>Тело запроса:</p> <ul style="list-style-type: none"> • <i>id</i> — идентификатор дежурства • <i>name</i> — название дежурства • <i>start_time</i> — новое начало дежурства • <i>interval</i> — новый конец дежурства • <i>ids</i> — новый массив id дежурных <p>При этом каждое поле, кроме id, может отсутствовать — в таком случае данное поле сохранит свое старое значение.</p> <p>Ограничения на входные данные:</p> <ul style="list-style-type: none"> • <i>id</i>: <ul style="list-style-type: none"> ◦ должен присутствовать • <i>start_time</i> (если присутствует): <ul style="list-style-type: none"> ◦ формат: уууу-ММ-дд'T'HH:mm:ss • <i>interval</i> (если присутствует): <ul style="list-style-type: none"> ◦ формат: PnYnMnDTnHnMnS <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> • Status code: 200 при успешном выполнении запроса, 404, если дежурство не нашлось, 500 — любая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) • Response body: информация в формате JSON: все поля обновленного уведомления
<p>Получить информацию о дежурстве</p>		<p>Относительный url: /duties Метод запроса: GET</p> <p>Переменная пути:</p> <p><i>id</i> — идентификатор пользователя</p> <p>Тело запроса:</p> <p>Нет</p> <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> • Status code: 200 при успешном выполнении запроса, 404 — дежурство не найдено, 500 — любая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) • Response body: информация о дежурстве в формате JSON:

		<ul style="list-style-type: none"> • id • name • start_time • interval • ids • currentlyUserId
	Получить информацию обо всех дежурствах	<p>Относительный url: /duties</p> <p>Метод запроса: GET</p> <p>Переменная пути: -</p> <p>Тело запроса: Нет</p> <p>Возвращаемые данные:</p> <ul style="list-style-type: none"> • Status code: 200 при успешном выполнении запроса, 500 — любая ошибка на стороне сервера (в любом из узлов цепочки запросов сервера) • Response body: информация о дежурствах в формате JSON: <ul style="list-style-type: none"> • id • name • start_time • interval • ids • currentlyUserId