

Министерство науки и высшего образования Российской Федерации  
Федеральное государственное автономное  
Образовательное учреждение высшего образования  
МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ  
(МОСКОВСКИЙ ПОЛИТЕХ)

ДОПУЩЕН К ЗАЩИТЕ

Заведующий кафедрой

\_\_\_\_\_ / Пухова Е. А. /

Руководитель образовательной программы

\_\_\_\_\_ / Даньшина М. В. /

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА  
по теме:  
**АВТОМАТИЗИРОВАННАЯ ПЛАТФОРМА РАЗВЕРТЫВАНИЯ  
КОНТЕЙНЕРИЗОВАННЫХ ФУНКЦИЙ В СРЕДЕ KUBERNETES**

по направлению 09.03.01 Информатика и вычислительная техника  
Образовательная программа (профиль) «Веб-технологии»

Студент: \_\_\_\_\_ / Журавлев Давид Александрович, 211–321/  
*подпись* *ФИО, группа*

Руководитель ВКР: \_\_\_\_\_ / Гонтовой Сергей Викторович, доц., к.т.н. /  
*подпись* *ФИО, уч. звание и степень*

Москва 2025

Министерство науки и высшего образования Российской Федерации  
Федеральное государственное автономное  
Образовательное учреждение высшего образования  
МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ  
(МОСКОВСКИЙ ПОЛИТЕХ)

**ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ**  
по направлению 09.03.01 Информатика и вычислительная техника  
Образовательная программа (профиль) «Веб-технологии»

Тема ВКР	Автоматизированная платформа развертывания контейнеризованных функций в среде Kubernetes
<b>ПРАКТИЧЕСКИЙ РЕЗУЛЬТАТ</b>	
Назначение	Система предназначения для автоматизации процессов развертывания и управления бессерверными вычислениями в изолированной среде.
Основные функции	<ol style="list-style-type: none"><li>1. Регистрация и авторизация пользователей.</li><li>2. Предоставление функционала создания групп.</li><li>3. Предоставление функционала создания кодов приглашения в группу.</li><li>4. Предоставление управления правами членов групп и доступом к запущенным задачам.</li><li>5. Развертывание и управление задачами в контейнерах.</li><li>6. Настройка выполнения задач по расписанию, на webhook-событию, ручным запуском.</li><li>7. Мониторинг выполнения задач в реальном времени.</li><li>8. Просмотр логов выполнения задач.</li></ol>
Используемые технологии и платформы	Java 21, Kotlin, Spring, Keycloak, Postgres, Kubernetes, Docker, Vue3, TypeScript, Pinia, TailwindCSS, DaisyUI, Git

<b>ВЫПОЛНЕНИЕ РАБОТЫ</b>	
Решаемые задачи	<ol style="list-style-type: none"> <li>1. Провести анализ предметной области.</li> <li>2. Сравнить существующие аналогичные решения.</li> <li>3. Провести анализ целевой аудитории веб-приложения.</li> <li>4. Определить функциональные требования к веб-приложению.</li> <li>5. Разработать пользовательские сценарии.</li> <li>6. Спроектировать архитектуру веб-приложения.</li> <li>7. Разработать дизайн-макеты страниц и компонентов веб-приложения.</li> <li>8. Спроектировать схему базы данных.</li> <li>9. Разработать серверную часть веб-приложения.</li> <li>10. Разработать клиентскую часть веб-приложения.</li> <li>11. Провести различные виды тестирования веб-приложения.</li> </ol>
Состав технической документации	<ol style="list-style-type: none"> <li>1. Пояснительная записка.</li> <li>2. Техническое задание.</li> </ol>
Состав графической части	<ol style="list-style-type: none"> <li>1. Презентация.</li> <li>2. Диаграмма IDEF0 AS-IS: 2 экз.</li> <li>3. Диаграмма IDEF0 TO-BE: 2 экз.</li> <li>4. Диаграмма компонентов платформы: 1 экз .</li> <li>5. Схемы отношений объектов БД: 2 экз.</li> <li>6. Диаграмма развертывания: 1 экз.</li> <li>7. Экраны интерфейса: 13 экз.</li> </ol>

## ПЛАН РАБОТЫ НАД ВКР

Этапы	Недели семестра																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Провести анализ предметной области.																		
Сравнить существующие аналогичные решения.																		
Провести анализ целевой аудитории веб-приложения.																		
Определить функциональные требования к веб-приложению.																		
Разработать пользовательские сценарии.																		
Спроектировать архитектуру веб-приложения.																		
Разработать дизайн-макеты страниц и компонентов веб-приложения.																		
Спроектировать схему базы данных.																		
Разработать серверную часть веб-приложения.																		
Разработать клиентскую часть веб-приложения.																		
Провести различные виды тестирования веб-приложения.																		

РУКОВОДИТЕЛЬ ОП:

«\_\_»\_\_\_\_2025, \_\_\_\_\_ / Даньшина Марина Владимировна /  
*подпись* *ФИО, уч. звание и степень*

РУКОВОДИТЕЛЬ ВКР:

«\_\_»\_\_\_\_2025, \_\_\_\_\_ / Гонтовой Сергей Викторович, доц., к.т.н. /  
*подпись* *ФИО, уч. звание и степень*

СТУДЕНТ:

«\_\_»\_\_\_\_2025, \_\_\_\_\_ / Журавлев Давид Александрович, 211–321/ /  
*подпись* *ФИО, группа*

## АННОТАЦИЯ

Наименование работы: автоматизированная платформа для развертывания и управления лямбда-функциями «Lambda».

Цель работы: разработка платформы для автоматизированного развертывания и управления вычислительными задачами в контейнерах, обеспечивающей удобное взаимодействие пользователей с инфраструктурой и оптимизацию ресурсов.

Объект исследования: процессы автоматизированного развертывания, исполнения и мониторинга контейнеризированных вычислительных задач.

Предмет исследования: архитектурные, программные и алгоритмические решения, обеспечивающие эффективное управление контейнерами, балансировку нагрузки, обработку задач и анализ их выполнения в распределенной среде.

Объем работы составляет 87 страниц. Работа включает в себя 3 главы, 23 рисунка, 3 листинга исходного кода и приложение, состоящее из 28 страниц. Библиография включает 51 источник.

Во введении раскрываются актуальность темы и практическая значимость разработки.

Первая глава посвящена анализу предметной области, формулировке целей и задач исследования, рассмотрению существующих аналогов.

Во второй главе определяются функциональные возможности платформы и выбирается технологический стек, описываются архитектура системы, пользовательский интерфейс и технические аспекты реализации.

Третья глава включает процесс тестирования, развертывания платформы, анализ её масштабируемости и возможных путей развития.

В заключении подводятся итоги работы, оценивается её практическая ценность и формулируются основные выводы.

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ .....	8
1 ПРЕДМЕТНАЯ ОБЛАСТЬ .....	10
1.1 Предметная область разработки бессерверных приложений.....	10
1.2 Анализ аналогов и конкурентов.....	12
1.3 Анализ целевой аудитории .....	16
1.4 Проблема, цель и задачи разработки.....	17
1.5 Требования к разрабатываемой системе .....	18
1.6 Вывод.....	20
2 ТЕХНИЧЕСКАЯ РЕАЛИЗАЦИЯ .....	21
2.1 Описание бизнес-процессов.....	21
2.2 Описание архитектуры решения .....	24
2.3 Обоснование технологического стека.....	32
2.4 Разработка модели данных .....	34
2.5 Разработка ролевой модели.....	39
2.6 Разработка каркасных макетов пользовательского интерфейса .....	40
2.7 Разработка программного интерфейса серверной части .....	45
2.8 Разработка серверной части .....	50
2.9 Разработка клиентской части .....	61
2.10 Вывод.....	69
3 ВНЕДРЕНИЕ И ЭКСПЛУАТАЦИЯ.....	71
3.1 Развертывание платформы в среде Kubernetes .....	71
3.2 Реализация нефункциональных требований .....	75
3.3 Тестирование платформы.....	76
3.4 Вывод.....	77
ЗАКЛЮЧЕНИЕ.....	79
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....	81
ПРИЛОЖЕНИЕ А.....	88

## ВВЕДЕНИЕ

В современном мире информационных технологий все больше задач требует выполнения в автоматизированном и распределенном режиме. Малые и средние компании, и отдельные разработчики сталкиваются с необходимостью периодически запускать нерегулярные ресурсоемкие вычислительные процессы, такие как обработка данных, генерация отчетов, обучение моделей машинного обучения и другие. Традиционные серверные решения требуют сложной настройки, постоянного администрирования и финансовых вложений, что делает их использование не всегда целесообразным.

Существующие облачные платформы, большинство из которых не представлены на российском рынке, такие как AWS Lambda, Google Cloud Functions и Azure Functions, предлагают serverless-вычисления, позволяя запускать код по требованию без необходимости управления серверами. Эти решения ориентированы в первую очередь на крупный бизнес и имеют ограничения по конфигурации окружения, стоимости вызовов и способам взаимодействия с системой. Использование таких сервисов в частных или внутрикорпоративных инфраструктурах часто оказывается невозможным. В качестве альтернативы некоторые компании разворачивают собственные решения на базе Kubernetes.

В данной работе рассматривается разработка платформы Lambda, предназначенной для автоматизированного развертывания и управления контейнеризированными вычислениями. Платформа позволяет пользователям загружать и исполнять задачи в виде Docker-контейнеров с возможностью настройки параметров запуска, ограничений по ресурсам.

Предлагаемая платформа ориентирована на малые и средние компании, которым требуется удобное и экономически эффективное решение для выполнения вычислений без необходимости разворачивания и поддержки сложных серверных инфраструктур.



Таким образом, разработка платформы Lambda является актуальной задачей, способной упростить доступ к автоматизированным вычислениям снизить затраты на поддержку серверных мощностей.

# 1 ПРЕДМЕТНАЯ ОБЛАСТЬ

## 1.1 Предметная область разработки бессерверных приложений

Бессерверные архитектуры представляют собой облачную модель исполнения программного кода, при которой используется модель запуска серверных вычислений «Backend as a service» (BaaS) [1].

Бессерверность – это общая технология, позволяющая не связываться с базовой инфраструктурой. Двумя областями применения бессерверной технологии являются серверная часть как услуга (BaaS) и Функция как услуга (FaaS) или серверная часть как услуга и функция как услуга соответственно.

BaaS (Серверная часть как услуга) – это облачное решение, предоставляющее готовую инфраструктуру для серверных приложений. Оно включает в себя набор сервисов, таких как аутентификация пользователей, база данных, файловое хранилище, push-уведомления и другие стандартные серверные компоненты. BaaS позволяет разработчикам сосредоточиться на логике приложения, не беспокоясь о развертывании и обслуживании серверной части. Примеры BaaS: Firebase, Parse, Back4App.

FaaS (Функция как услуга) – это вычислительная модель, в которой разработчик записывает отдельные функции, которые выполняются по запросу (например, при запуске события). Эти функции выполняются в ответ на такие события, как HTTP-запросы или изменения в базе данных, и масштабируются автоматически. В отличие от BaaS, который предоставляет более широкий спектр серверных сервисов, FaaS фокусируется исключительно на выполнении кода. Примеры FaaS: AWS Lambda, облачные функции Google, функции Azure [2].

Применение бессерверных вычислений, в отличие от традиционных серверных, позволяет не заботиться о доступности и масштабировании инфраструктуры, а также снижать издержки возникающие из-за простоев оборудования в периоды малой нагрузки [3].

Несмотря явное указание отсутствия серверов при применении бессерверных вычислений, сервера при использовании данного подхода используются, но управление ими передается облачному провайдеру. Изначально термин «Serverless» использовался в контексте rich-client [4] приложений, одностраничные веб-приложения (SPA) [5] или мобильных приложений, отличительной чертой которых является использование обширной экосистемы доступных в облаке данных, таких как удаленные базы данных (например Firebase), службы аутентификации и так далее.

К serverless приложениям также относятся приложения, в которых серверная логика по-прежнему реализуется разработчиком приложения, но, в отличие от традиционных архитектур, она выполняется в вычислительных контейнерах без сохранения состояния, которые запускаются по событиям. Такие контейнеры недолговечны, полностью управляются третьей стороной, так же существуют только в рамках одного вызова.

Один из способов реализовать такой подход это - “Function as a service” или “FaaS”. AWS Lambda в настоящее время является одной из самых популярных реализаций платформы "Function as a service, но существует также множество других. Однако, несмотря на различия в реализации, фундаментальные принципы общие вызовы остаются схожими для большинства FaaS-решений.

Несмотря на все преимущества, которые дает применение бессерверных вычислений, необходимо учитывать особенности разработки под бессерверные системы и знать о налогах, которые накладывает применение бессерверных вычислений, поскольку именно осознанное управление этими аспектами позволяет в полной мере реализовать потенциал данной архитектуры.

Наиболее важной может оказаться проблема холодного запуска [6]. При масштабировании до 0 при получении запроса на выполнение функции значительное время может быть затрачено на первоначальный запуск контейнера с необходимым сервисом [7].

## 1.2 Анализ аналогов и конкурентов

Первое появление концепции бессерверных вычислений связано с AWS Lambda, запущенной в 2014 году Amazon Web Services. Это решение позволило разработчикам загружать небольшие функции, которые выполнялись в ответ на такие события, как HTTP-запросы, изменения в базе данных или сообщения в очереди.

После успеха AWS Lambda, другие крупные облачные провайдеры представили свои serverless-решения:

- 1) 2016г. – Google Cloud Functions;
- 2) 2016г. – Microsoft Azure Functions;
- 3) 2017г. – IBM Cloud Functions;
- 4) 2019г. – Google Cloud Run.

Ни один из крупнейших поставщиков Serverless решений на данный момент не доступен в России.

В России на данной момент доступны поставщики serverless инфраструктуры:

- 1) Cloud.ru;
- 2) Yandex Cloud.

С развитием контейнерных технологий появилась возможность объединять Serverless и Container-based подходы, что позволило избежать жестких ограничений FaaS.

Для возможности плавной смены поставщика serverless инфраструктуры существуют open-source фреймворки позволяющие разрабатывать решения, одинаково работающие на инфраструктуре различных поставщиков, реализующих поддержку данных решений. К таким решениям можно отнести:

- 1) Apache Openwhisk <https://openwhisk.apache.org/>;
- 2) Fission <https://fission.io/>;
- 3) Fn <https://fnproject.io/>;

- 4) Knative <https://knative.dev/docs/>;
- 5) Kubeless <https://github.com/vmware-archive/kubeless>;
- 6) Nuclio <https://nuclio.io/>;
- 7) OpenFaaS <https://www.OpenFaaS.com/>.

Рассмотрим основных игроков на рынке serverless решений.

AWS Lambda является флагманом подхода «Function as a Code», одним из первых и самых популярных решений для запуска облачных функций. Сервис работает не с контейнерами, а напрямую с программным кодом, что накладывает ограничения на стек технологий, которые могут быть использованы при интеграции с данным сервисом.

К плюсам системы AWS Lambda можно отнести:

- 1) автоматическое масштабирование;
- 2) полная интеграция с AWS-экосистемой (S3, DynamoDB);
- 3) поддержка множества языков (Python, Node.js, Java, Go и др.);
- 4) гибкая система триггеров (HTTP, очередь сообщений).

Минусами системы являются:

- 1) задержки из-за холодных стартов;
- 2) высокая стоимость при интенсивных нагрузках;
- 3) недоступность в России.

Google Cloud Run, продукт, разработанный компанией Google, является представителем класса BaaS систем, которые используют в качестве единицы развертывания контейнер. В отличие от систем, использующих в качестве единицы развертывания код, подход с контейнерами не накладывает никаких ограничений на стек используемых технологий и сложность разворачиваемого сервиса.

Единственным требованием к запускаемому юниту является соответствие спецификации Open Container Initiative (OCI) [8], благодаря чему разработчики получают возможность переиспользования существующих контейнеризованных решений без привязки к специфическим средам исполнения.

К плюсам системы Google Cloud Run можно отнести:

- 1) поддержка любых языков и сред (т.к. работает с контейнерами);
- 2) простота развертывания;
- 3) гибкость в выборе окружения. Минусами системы являются:
- 4) более высокая цена по сравнению с FaaS-решениями;
- 5) холодные старты при отсутствии постоянной нагрузки;
- 6) сильная интеграция с Google Cloud;
- 7) недоступность в России.

OpenFaaS, OpenWhisk, Knative являются популярными open-source [9] решениями для развертывания локальных FaaS сервисов. В отличие от проприетарных облачных FaaS-платформ такие решения, при использовании для управления локальной инфраструктурой позволяют избежать ограничений на время выполнения и более точно прогнозировать и управлять нагрузкой. Некоторые из представленных на рынке систем могут развернуты на любых мощностях: Kubernetes, Docker Swarm, локальные сервера. Все представленные системы работают с контейнерами как с единицей развертывания, а значит так же накладывают ограничение соответствия стандарту OCI.

К плюсам open-source систем разворачиваемых локально можно отнести:

- 1) полный контроль над инфраструктурой;
- 2) оптимизация и прогнозируемость затрат;
- 3) технологическая независимость.

Минусами систем являются:

- 1) необходимость самостоятельной настройки сервисов;
- 2) отсутствие поддержки вендора.

Аналогом же разрабатываемого решения будет являться использование классических решений размещения приложений на собственных или арендованных мощностях, таких как виртуальные машины, выделенные серверы, управляемые Kubernetes или Docker Swarm кластеры.

Использование традиционных решений для размещения приложений на собственных или арендованных мощностях, таких как виртуальные машины (VM), выделенные серверы, а также управляемые кластеры Kubernetes или Docker Swarm, имеет свои преимущества и недостатки.

Преимущества:

- 1) полный контроль над инфраструктурой;
- 2) гибкость в выборе технологий;
- 3) предсказуемость затрат.

Недостатки:

- 1) необходимость наличия команды поддержки и технических компетенций;
- 2) потребность в ресурсах;
- 3) необходимость самостоятельной настройки сервисов;
- 4) высокие капитальные затраты при простоях вычислительных мощностей;
- 5) сложности с масштабированием;
- 6) ограниченная гибкость.

В следствие наличия плюсов и минусов у каждого подхода к управлению инфраструктурой, большинству компаний подойдет гибридный подход, объединяющий локальные ресурсы с облачными сервисами, позволяет достичь баланса между контролем над инфраструктурой, гибкостью в масштабировании и эффективным управлением затратами.

При использовании гибридного подхода сервисы, испытывающие постоянную нагрузку, а также сервисы время отклика которых критично для операций, которые они выполняют не должны быть развернуты с использованием классических подходов управления доставкой и развертыванием. В это же время использование FaaS сервисов для функционала, не находящегося под постоянной нагрузкой и не имеющего строгих требований к времени ответа, может помочь оптимизировать расходы на серверную инфраструктуру и простои оборудования.

### 1.3 Анализ целевой аудитории

На начальных этапах разработки платформы целевой аудиторией разрабатываемой системы являются стартапы, разработчики, малые и средние компании, а также образовательные учреждения, у которых есть потребность в выполнении задач, лежащих в прикладной области платформы.

Важным аспектом целевой аудитории является отсутствие собственной серверной инфраструктуры или отсутствие возможности избежать ее простоев. Целевая аудитория системы не имеет возможности и компетенций что бы использовать существующие open-source [10] решения, такие как OpenFaaS, OpenWhisk, Knative на мощностях собственной инфраструктуры.

Для целевой аудитории целью использования платформы является автоматизация и оптимизация процессов компании, выполнение требовательных вычислений, лежащих в области деятельности компании.

Исходя из выявленных сегментов целевой аудитории, ключевые потребности можно условно разделить на несколько направлений:

- 1) компенсация отсутствия собственной инфраструктуры;
- 2) оптимизация затрат на развертывание и эксплуатацию;
- 3) выполнение ресурсоемких вычислений.

На основании анализа целевой аудитории можно сделать вывод, что представители выделенных сегментов заинтересованы в локализованном решении для запуска бессерверных вычислений. Основными требованиями для них являются простота управления и отказ от зависимости от зарубежных вендоров, что особенно актуально в текущих условиях.

Таким образом, реализация поддержки стандартов, обеспечивающих совместимость с другими платформами, может выступить значимым конкурентным преимуществом. Это позволит привлечь более широкую аудиторию и обеспечить плавную миграцию или интеграцию с уже используемыми решениями.



## 1.4 Проблема, цель и задачи разработки

Современные решения задач в области выполнения бессерверных вычислений в настоящее время сопряжены с рядом проблем и компромиссов, таких как недоступность крупных FaaS сервисов в России, ограничения по формату, времени выполнения, нагрузке выполняемых функций.

Разрабатываемая платформа призвана устранить часть этих барьеров, предлагая локализованное и удобное в управлении решение для запуска бессерверных вычислений.

Платформа ориентирована на выполнение задач различных типов, обеспечивая гибкость и универсальность в применении. Это делает её эффективным инструментом для широкой аудитории пользователей с различными требованиями к вычислительным процессам.

Целью работы является разработка платформы для автоматизированного развертывания и управления вычислительными задачами в контейнерах.

Для достижения целей разработки необходимо выполнить следующие задачи:

- 1) провести анализ предметной области;
- 2) сравнить существующие аналогичные решения;
- 3) провести анализ целевой аудитории веб-приложения;
- 4) определить функциональные требования к веб-приложению;
- 5) разработать пользовательские сценарии;
- 6) спроектировать архитектуру веб-приложения;
- 7) разработать дизайн-макеты страниц и компонентов веб-приложения;
- 8) спроектировать схему базы данных;
- 9) разработать серверную часть веб-приложения;
- 10) разработать клиентскую часть веб-приложения;
- 11) провести различные виды тестирования веб-приложения.

## 1.5 Требования к разрабатываемой системе

Данный раздел описывает ключевые требования к проектируемой системе. Эти требования определяют ожидаемую функциональность и качественные характеристики системы, служа основой для дальнейшего проектирования и реализации.

Платформа должна предоставлять пользователям необходимые инструменты для работы с учетными записями и вычислительными задачами. Пользователи должны иметь возможность самостоятельно регистрироваться в системе, входить под своей учетной записью и восстанавливать доступ при утере пароля. Также необходим просмотр базовой информации своего профиля и членства в группах.

Центральной функцией системы является управление контейнерами с исполняемыми приложениями. Пользователи должны определять параметры задачи: имя, образ среды выполнения, команду запуска и переменные окружения. Система должна поддерживать разные типы запуска: однократный, периодический по расписанию, с возможностью настройки этого расписания или по внешнему сетевому вызову. Задачу можно создать, как в пространстве пользователя, так и при необходимости связать с группой для совместной работы.

Для контроля за задачами пользователи должны иметь возможность просматривать список доступных им задач и их текущее состояние, также получать детальную информацию о конфигурации и истории запусков конкретной задачи. Важно обеспечить доступ к результатам выполнения каждого запуска, включая статус, логи и код завершения. Платформа должна позволять перезапускать существующие задачи, отменять активные запуски и удалять задачи, прекращая их дальнейшее выполнение. Для webhook-задач требуется механизм их активации по уникальному URL.

Система должна поддерживать создание групп для совместного использования ресурсов и управления задачами. Пользователи должны иметь возможность присоединяться к группам и покидать их. Администраторы

групп должны иметь функционал исключения участников и управления правами доступа внутри группы (просмотр, редактирование, запуск, администрирование). Все участники могут просматривать состав группы и права доступа, а администраторы – управлять процессом приглашения.

Необходимо реализовать логирование ключевых событий и ошибок для диагностики и анализа, а также предоставлять пользователям доступ к логам их задач.

Для безопасного взаимодействия с системой, должна быть разработана система аутентификации пользователей и контроля доступа на основе ролей. Среды выполнения пользовательских задач должны быть надежно изолированы друг от друга и от системных компонентов. Необходимо обеспечить защиту данных, а также защиту от стандартных сетевых угроз.

Производительность системы должна обеспечивать быстрый отклик интерфейсов при ожидаемой нагрузке и эффективное использование вычислительных ресурсов. Масштабируемость архитектуры важна для обработки растущего числа пользователей и задач путем добавления ресурсов. Удобство использования системы достигается через интуитивно понятный пользовательский интерфейс и логичный, хорошо документированный API.

Поддерживаемость и расширяемость должны обеспечиваться качественной архитектурой и кодом, что упрощает внесение изменений и добавление функционала.

Система оперирует несколькими основными категориями данных, которые необходимо корректно хранить и обрабатывать. К ним относятся данные пользователей, данные групп, данные задач и данные результатов выполнения. Также используются справочные данные. Важно обеспечить логическую целостность и непротиворечивость этих данных в хранилище.

Основным интерфейсом для пользователей является графический пользовательский интерфейс, представляющий веб-приложение. Для автоматизации и интеграции предоставляется программный интерфейс (API).

Предоставляемые интерфейсы должны быть спроектированы с учетом удобства пользователя (UI/UX). Графический интерфейс должен быть интуитивно понятным, логически структурированным и визуально последовательным. Программный интерфейс (API) должен быть четко определен, логичен и удобен для интеграции, следуя современным практикам проектирования API.

## **1.6 Вывод**

В первой главе дипломной работы был проведен анализ предметной области управления бессерверными вычислениями. Были рассмотрены существующие подходы к управлению вычислениями, проанализированы существующие крупные решения, их преимущества и ограничения.

В ходе проведения анализа было выявлено что существующие решения в области FaaS и BaaS накладывают существенные ограничения на пользователей, делая невозможными выполнение большого спектра задач. Был проведен анализ существующих технических решений с открытым исходным кодом позволяющих добиться схожих возможностей управления бессерверными вычислениями на инфраструктуре пользователя. Было выявлено что такие решения требуют от пользователя повышенных затрат и компетенций для поддержания собственной инфраструктуры.

Целевая аудитория была разделена на несколько сегментов, в том числе малые и средние предприятия, образовательные учреждения. Анализ потребностей, рыночных трендов и конкурентного окружения позволил выделить ключевые требования целевой аудитории. На основе проведенного анализа сформулированы выводы и рекомендации к разработке.

В результате была сформирована цель разработки и сформулированы задачи необходимые для ее выполнения, а также описаны функциональные и нефункциональные требования к разрабатываемой системе.

## 2 ТЕХНИЧЕСКАЯ РЕАЛИЗАЦИЯ

### 2.1 Описание бизнес-процессов

Главные процессы связаны с выполнением вычислительных задач и развертыванием контейнеров, что требует ручной настройки серверной инфраструктуры и управления вычислительными ресурсами. Это обеспечивает выполнение клиентских запросов, но сопровождается значительными временными затратами и сложностями администрирования.

На рисунке 1 показан сам бизнес-процесс выполнения вычислительных задач и развертывания контейнеров до автоматизации, а на рисунке 2 представлена декомпозиция данного процесса.

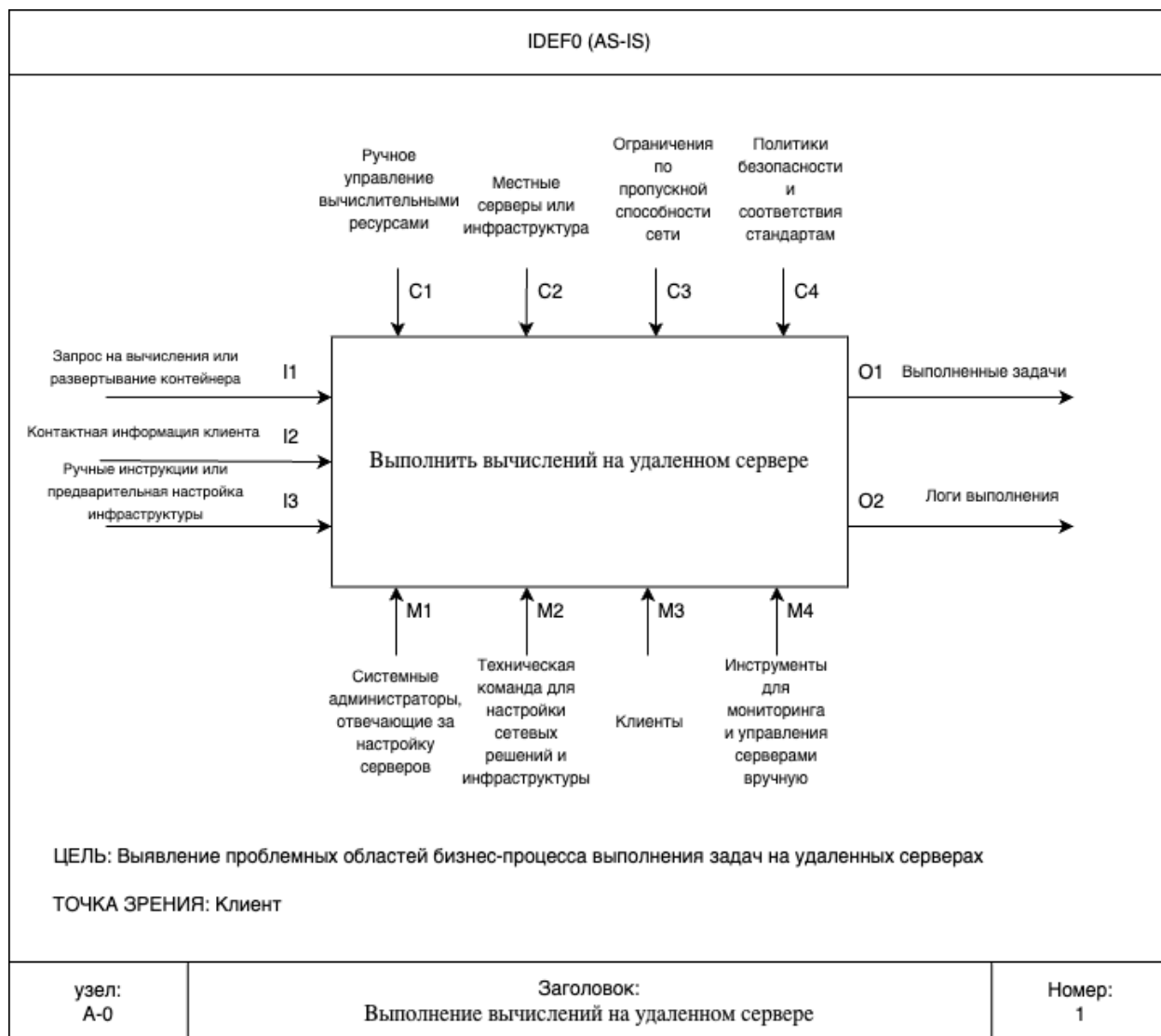


Рисунок 1 – Процесс выполнения вычислений на удаленном сервере

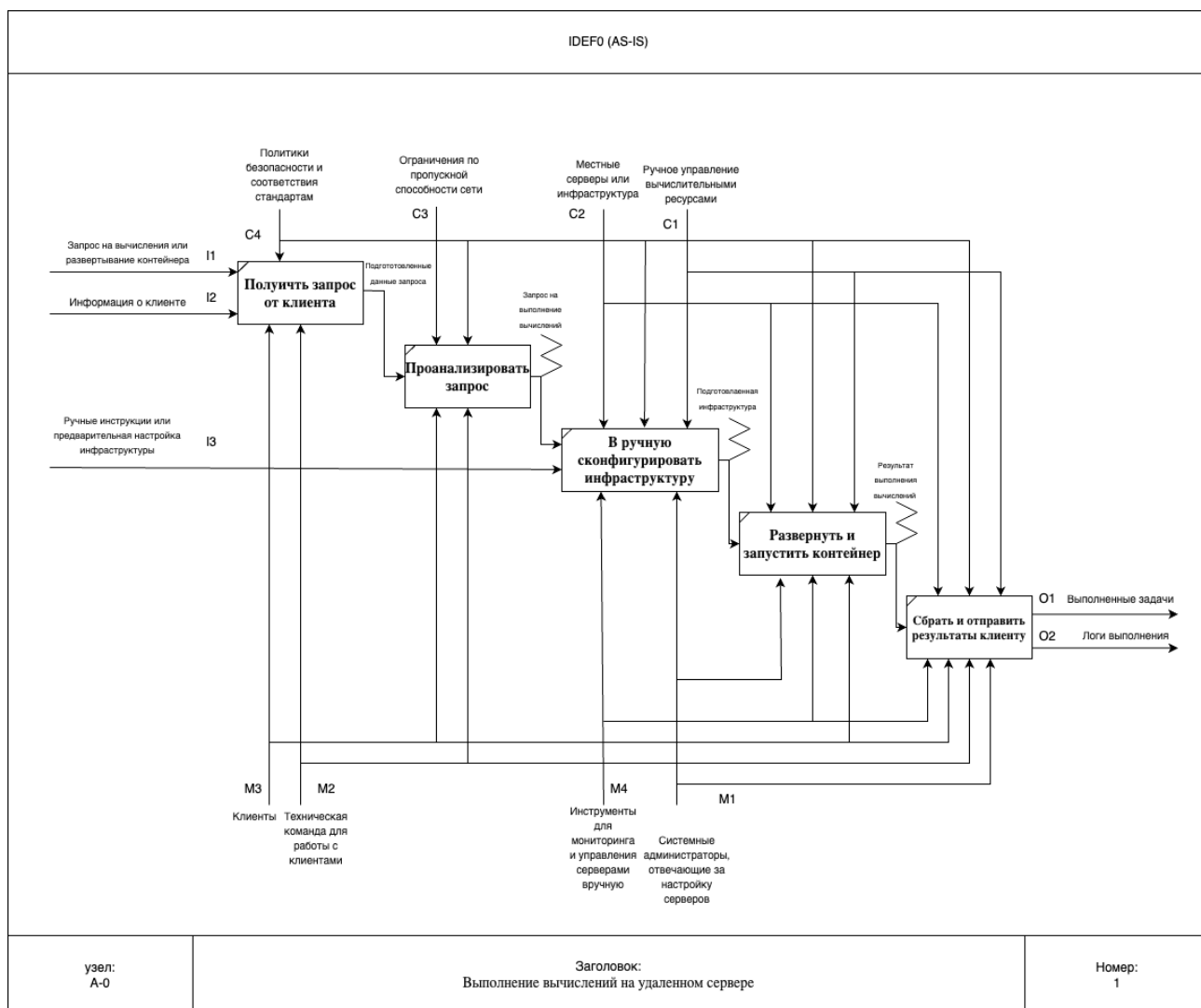


Рисунок 2 – Декомпозиция процесса выполнения удаленных вычислений

Среди основных проблем неавтоматизированного бизнес-процесса можно выделить:

- 1) получение запросов от клиентов через ручные каналы связи, что увеличивает время на их обработку и уточнение технических требований;
- 2) ручное конфигурирование серверной инфраструктуры под каждую задачу, что требует значительных временных и трудовых затрат от администраторов;
- 3) отсутствие централизованной системы мониторинга выполнения задач, что приводит к необходимости постоянного ручного контроля и увеличению вероятности ошибок;
- 4) передача результатов клиентам осуществляется вручную, что замедляет процесс и увеличивает вероятность задержек.

После автоматизации значительно минимизируется участие специалистов в ручной настройке серверной инфраструктуры и мониторинге выполнения задач, что упрощает и ускоряет процесс развертывания и выполнения контейнеров. На рисунке 3 показан бизнес-процесс выполнения лямбда-функций после автоматизации, а на рисунке 4 представлена декомпозиция данного процесса.

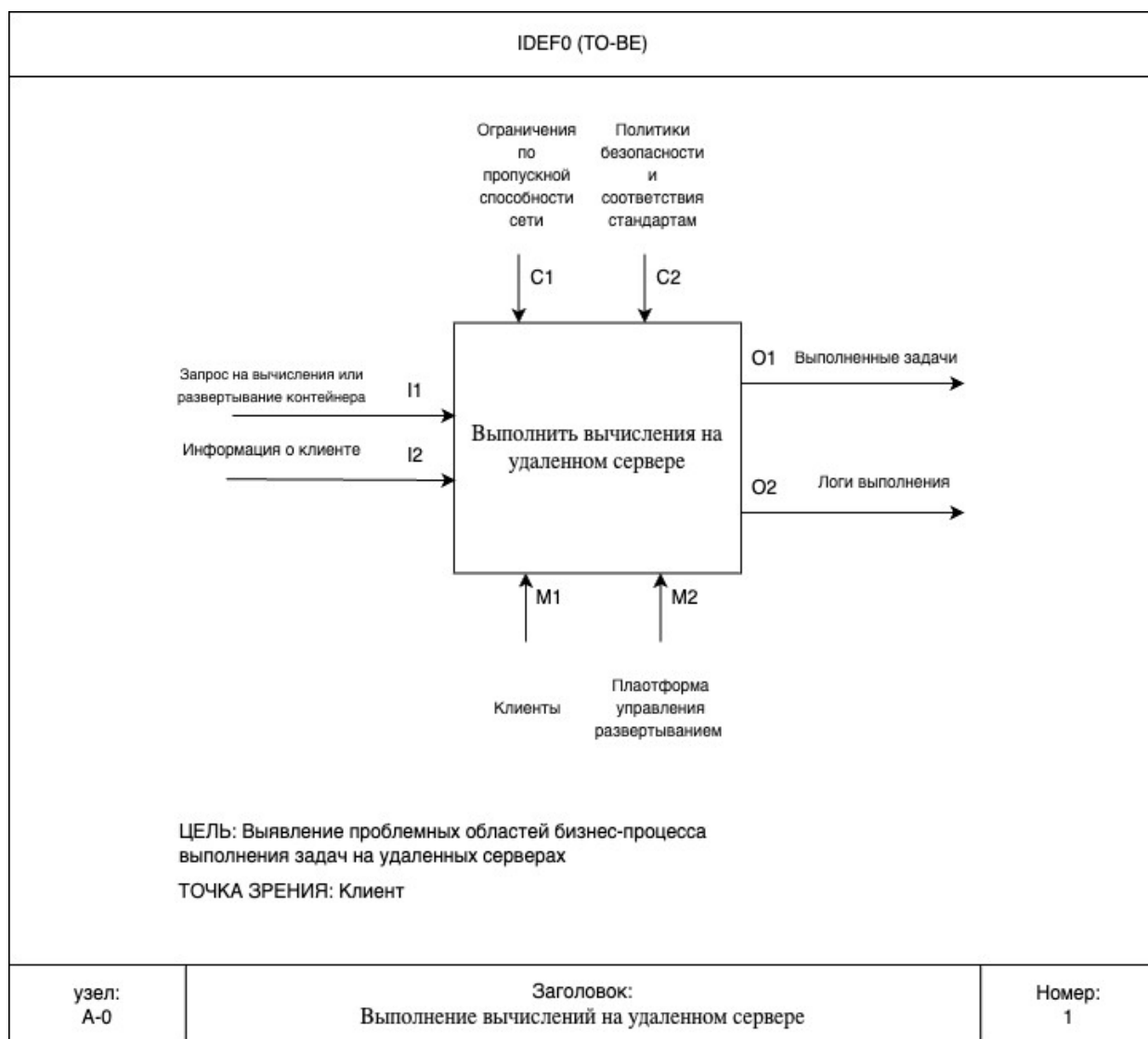


Рисунок 3 – Процесс выполнения вычислений в бессерверной среде

Основными преимуществами автоматизированного бизнес-процесса являются:

- 1) автоматическое развертывание контейнеров с минимальным участием специалистов;
- 2) гибкость настройки выполнения задач;

- 3) автоматическое формирование отчетов по выполнению задач и сбор статистики о загрузке ресурсов;
- 4) возможность одновременного выполнения множества задач с оптимизацией использования инфраструктуры.

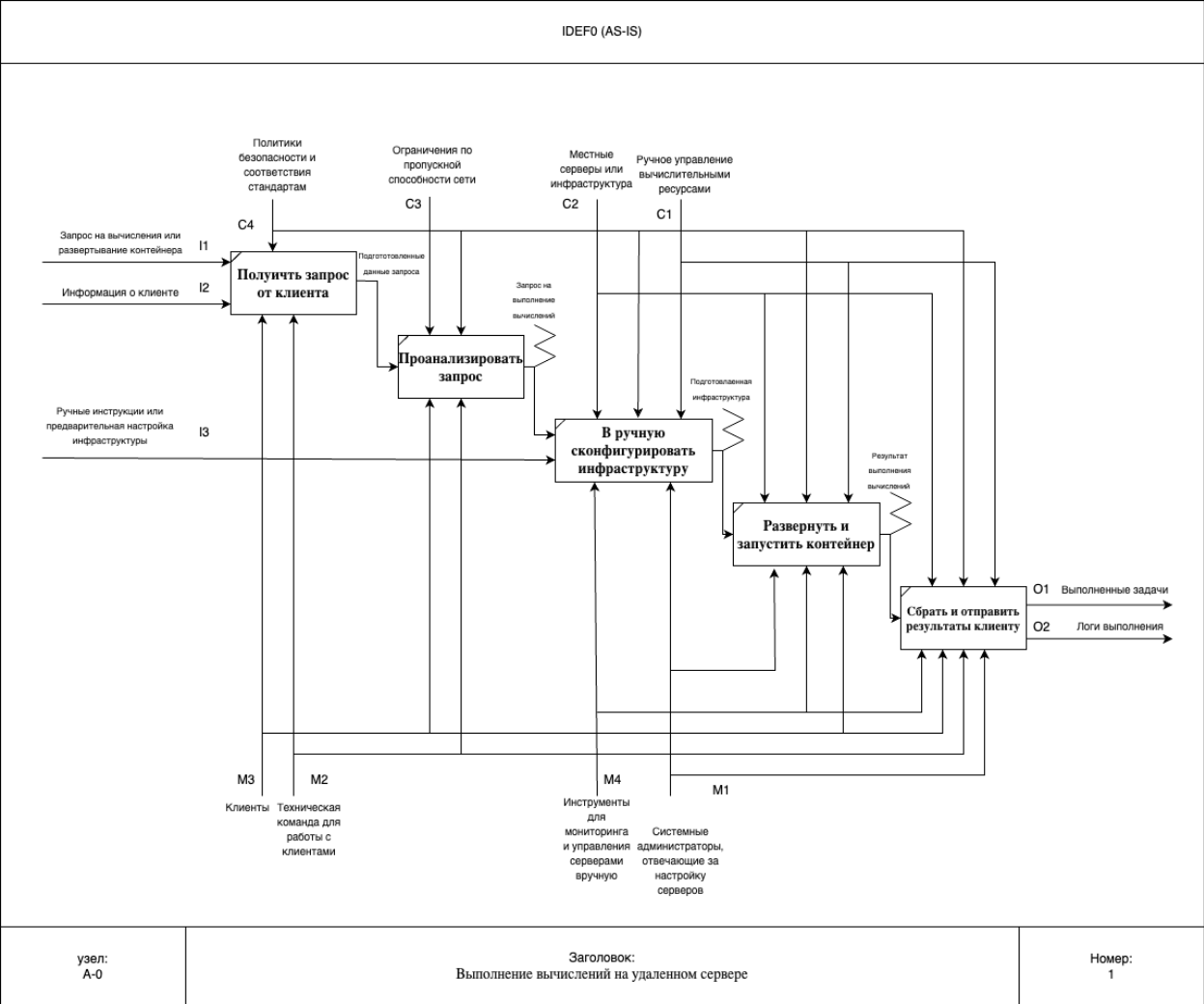


Рисунок 4 – Декомпозиция процесса выполнения вычислений  
в бессерверной среде

### 2.2 Описание архитектуры решения

Описание архитектуры приложения является ключевым этапом в проектировании программного решения. Описанная архитектура позволяет разделить систему на независимые компоненты, определить взаимосвязи между ними, правила их масштабирования, а также обеспечить согласованность в их разработке.



Разрабатываемое приложение представляет собой многокомпонентное клиент-серверное решение и состоит из независимых функциональных блоков:

- 1) серверная часть приложения;
- 2) клиентская часть приложения.

Серверная часть в свою очередь состоит из таких компонентов как:

- 1) Internal API Gateway;
- 2) External API Gateway;
- 3) сервер авторизации;
- 4) Reverse proxy [11];
- 5) брокер сообщений;
- 6) СУБД;
- 7) слой платформенных сервисов;
- 8) подчиненный кластер Kubernetes.

На рисунке 5 представлена схема взаимодействия и размещения компонентов системы.

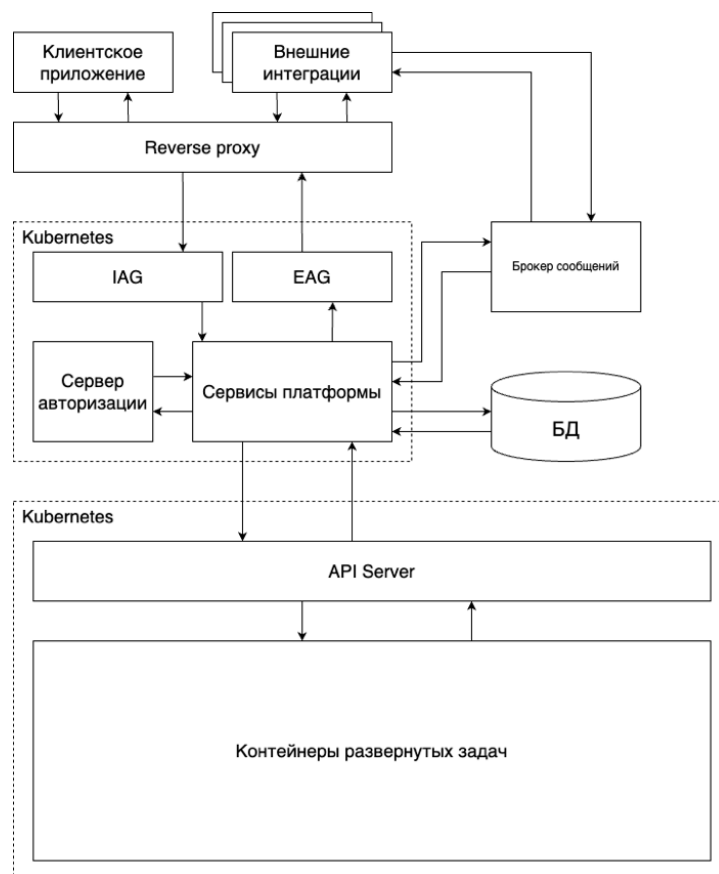


Рисунок 5 – Схема компонентов платформы

Сейчас на рынке существует несколько подходов к развертыванию сервисов. Каждая из представленных сред имеет свои плюсы и минусы.

В настоящее время развертывание веб-сервисов на физических серверах является устаревшим подходом. Такой подход имеет большое число минусов, таких как необходимость управления зависимостями разворачиваемых сервисов, сложности с масштабированием.

Виртуальные машины позволяют запускать сервисы в изолированных виртуальных средах, хорошо масштабируются, но требуют ручного управления ресурсами, сильно уступают контейнерным решениям по эффективности использования ресурсов и чаще всего используются для узкоспециализированных сервисов [12].

Развертывание в контейнерах без оркестрации хорошо подходит для разворачивания небольших сервисов, а также для локального запуска, но при использовании в промышленных средах требование ручного управления контейнерами делает такие системы неприменимыми [13].

Контейнерные оркестраторы предоставляют возможность объединения нескольких физических серверов в один кластер с едиными ресурсами, таким образом осуществляется единое эффективное управление всеми ресурсами системы, достигается ее масштабируемость и отказоустойчивость [14].

Для разворачивания разрабатываемой платформы оптимальным выбором является Kubernetes, так как он обеспечивает гибкое и эффективное управление ресурсами кластера при помощи yaml манифестов и является стандартом в индустрии для выполнения оркестрации контейнеров [15]. Его использование способствует унификации процессов развертывания и эксплуатации и открывает доступ к обширной экосистеме инструментов, готовых решений и накопленному сообществом опыту.

Размещение веб-сервера или сервера приложений непосредственно в Интернете дает злоумышленникам прямой доступ к любым уязвимостям базовой платформы. Для того что бы избежать появления такого рода уязвимостей широко распространен паттерн Reverse-proxy [11].

Reverse-proxy представляет собой промежуточный слой между внутренними сервисами приложения и внешними клиентами. Так же веб-сервер, выступающий в роли reverse-proxy может выполнять следующие функции:

- 1) маршрутизация запросов;
- 2) балансировка нагрузки;
- 3) SSL-терминация;
- 4) фильтрация запросов;
- 5) кеширование;
- 6) сжатие контента;
- 7) управление статическими ресурсами.

В качестве reverse-proxy для реализации платформы был выбран веб-сервер nginx. Этот компонент будет отвечать за SSL-терминацию и раздачу статического контента.

В качестве реализацией для Internal Api Gateway (IAG) и External Api Gateway (EAG) выбраны соответствующие реализации Istio Ingress и Egress.

Главная задача, которую выполняет IAG – распределение входящего трафика по репликам сервисов для обеспечения равномерного и эффективного использования ресурсов.

В то же время EAG отвечает за маршрутизацию исходящего трафика. Он служит точкой выхода трафика, направленного на получение внешних ресурсов.

Сервер авторизации в платформе отвечает за авторизацию и аутентификацию пользователей, а также запросов от входящих интеграций, выполняет централизованное управление доступом к сервисам платформы.

Среди большого количества реализаций сервера авторизации для платформы выбран сервис Keycloak. Keycloak по сравнению с конкурентами обладает следующими преимуществами:

- 1) решение с открытым исходным кодом;
- 2) поддержка большого количества протоколов авторизации и SSO;

- 3) наличие системы управления пользователями и ролями;
- 4) нативная интеграция с Kubernetes и масштабируемость;
- 5) возможность локального развертывания;
- 6) поддержка СУБД PostgreSQL;
- 7) наличие Java клиента;
- 8) наличие адаптера Spring Security;
- 9) наличие административной панели.

Брокер сообщений – это промежуточное звено в системе. Он обеспечивает асинхронное взаимодействие между компонентами системы. Основная задача, которую выполняет брокер сообщений - позволяет не дожидаться выполнения ресурсоемких операций и получить результат их выполнения асинхронно позже.

Брокеры сообщений играют ключевую роль в реализации таких системных паттернов как очередь сообщений [16], Saga [17], а так же целого семейства Enterprise Integration Pattern (EIP) [18].

На данный момент основными реализациями брокера сообщений представленными на рынке являются:

- 1) Apache Kafka;
- 2) RabbitMQ;
- 3) IBM MQ;
- 4) ActiveMQ;
- 5) Redis Streams.

В качестве основного брокера сообщений на платформе выбран брокер сообщений Apache Kafka. К основным плюсам по сравнению с другими реализациями брокеров можно отнести концептуальное отличие данного брокера от других представленных на рынке.

Kafka поддерживает многократное потребление сообщений, делая возможными сценарии использования, такие как использование Kafka в качестве Enterprise Service Bus [19] и позволяет не терять сообщения при ошибках и прерываниях обработки.

Так же в преимущества выбора брокера сообщений Kafka можно отнести:

- 1) масштабируемость и производительность;
- 2) высокая доступность и отказоустойчивость;
- 3) поддержка потоковой обработки;
- 4) гибкость интеграции;
- 5) интеграция с Kubernetes;
- 6) обширная экосистема и поддержка сообщества.

В проектируемой системе база данных необходима для хранения и управления пользовательскими данными. Система управления базой данных (СУБД) обеспечивает эффективную организацию данных, в отдельных случаях транзакционность, целостность и надежность. СУБД так же предоставляет возможности масштабирования и интеграций с другими системами.

Все СУБД можно разделить на две группы по способу организации данных: реляционные и нереляционные. Главное отличие реляционных баз данных от нереляционных заключается в использовании реляционными базами данных таблиц с фиксированной схемой для хранения данных и поддержка строгих отношений между таблицами с использованием внешних ключей, в то время как нереляционные базы данных являются более гибкими и хранят данные в формате ключ - значение и не предъявляя каких либо требований к схеме хранимых данных.

Данные, хранимые платформой сильно структурированы и для выполнения бизнес логики приложения критична их целостность, поэтому единственным верным выбором будет реляционная СУБД. На данный момент на рынке представлены следующие сервисы:

- 1) PostgreSQL;
- 2) MySQL;
- 3) Oracle Database;
- 4) MariaDB.

Для использования в реализации платформы выбрана СУБД PostgreSQL по следующим причинам:

- 1) поддержка сложных типов данных, возможность расширения встроенных типов данных;
- 2) поддержка транзакций;
- 3) открытый исходный код и активное сообщество;
- 4) масштабируемость и отказоустойчивость.

Слой платформенных сервисов играет ключевую роль в системе, так как именно он отвечает за функциональные возможности системы, управление процессами выполнения вычислений.

На данный момент существует два конкурирующих подхода к организации сервисов приложений - монолит и микросервисы [20].

Монолитный сервис, представленный на рисунке 6, представляет собой единую единицу развертывания, которая отвечает за все функции системы.

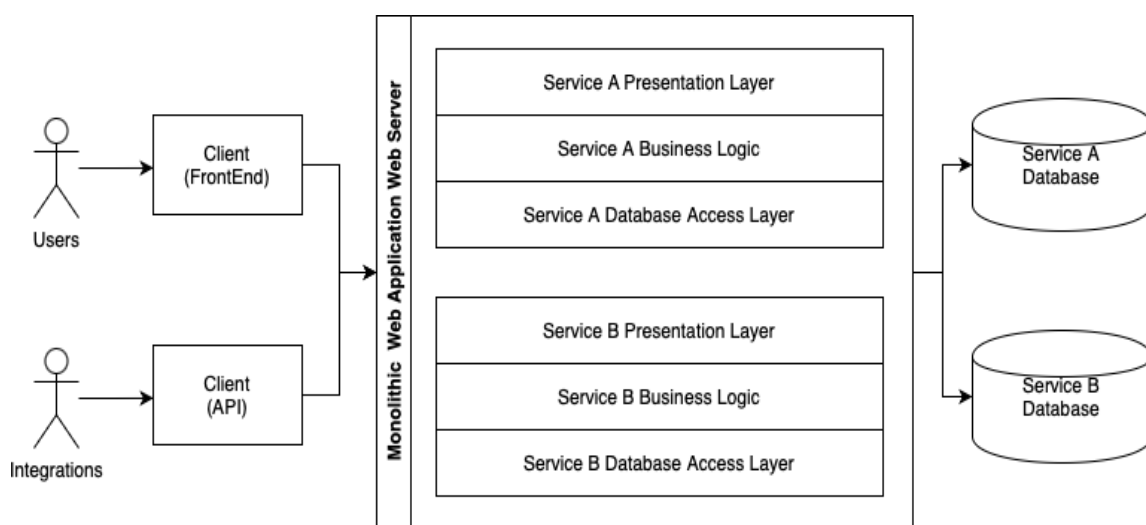


Рисунок 6 – Приложение с монолитной архитектурой

При использовании такого подхода на начальных этапах можно добиться быстрого вывода на рынок основного функционала, но при длительной разработке и с использованием монолитной архитектуры можно столкнуться с проблемами со сложностью внесения изменений, увеличении времени холодного старта, увеличением времени прохождения конвейера доставки и развертывания, а также с увеличенным расходом ресурсов при

масштабировании и возможными периодами недоступности при развёртывании или неожиданных отказах серверного оборудования.

Главная идея микросервисной архитектуры изображена на рисунке 7 и заключается в разделении системы на отдельные независимые подсистемы - микросервисы. Каждый микросервис должен отвечать за свою небольшую часть функционала.

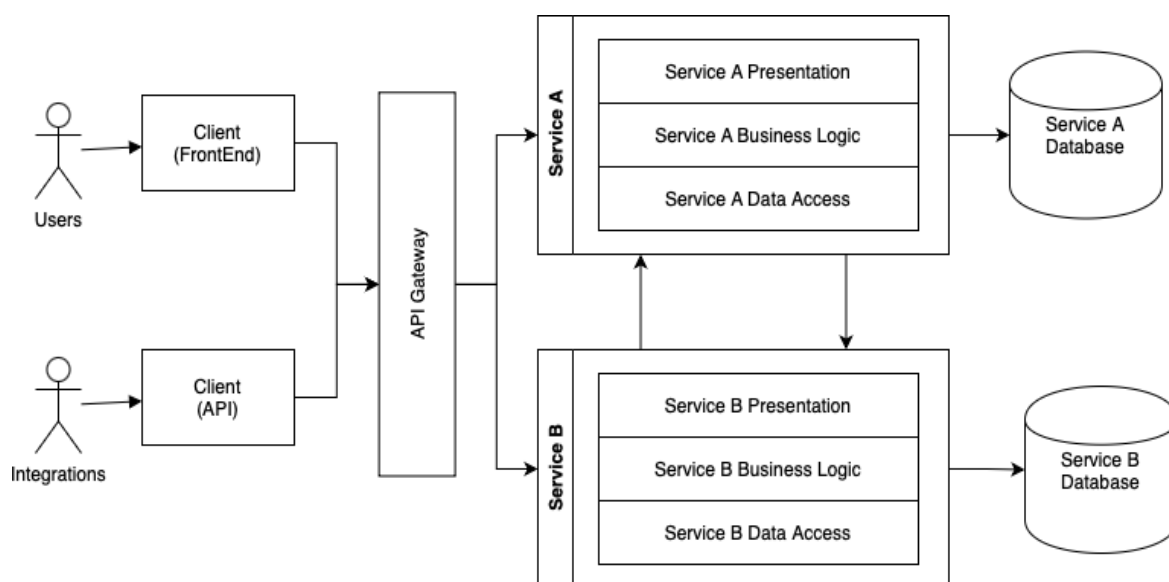


Рисунок 7 – Приложение с микросервисной архитектурой

Основным преимуществом использования микросервисов является возможность независимого развёртывания и прохождения конвейера доставки и развёртывания. Микросервисы могут масштабироваться независимо, позволяя добавлять реплики нагруженным сервисам во время пиковых периодов нагрузки, а малонагруженные сервисы разворачивать с малым количеством реплик.

Использование микросервисной архитектуры усложняет внесение изменений в существующую систему, а также требует более сложного управления инфраструктурой.

При использовании микросервисной архитектуры появляется необходимость в сохранении обратной совместимости между версиями сервисов, так как при установке поставки на стенде может находиться одновременно старая и новая версия сервиса для поддержки плавной незаметной для пользователя раскатки.

Использование микросервисов так же увеличивает время отклика сервисов, так как возникают дополнительные накладные расходы при выполнении межсервисных вызовов, так же при отказе отдельных сервисов могут возникать трудности с локализацией проблемы, что приводит к необходимости внедрения распределенной трассировки вызовов.

Для сервисного слоя платформы выбрана монолитная архитектура, при этом необходимо осуществлять проектирование функционала так, чтобы в последствие он мог быть отделен в отдельные сервисы. При разработке важно соблюдать слабую связанность [21] между сервисами, используя такие подходы как Domain Driven Design(DDD) [22] или Vertical Slice [23].

Подчиненный кластер Kubernetes в системе предназначен для разворачивания в нем задач, создаваемых пользователями. Он является выделенной изолированной средой, которая обеспечивает гибкость, масштабируемость и отказоустойчивость системы.

### **2.3 Обоснование технологического стека**

При разработке клиентской части приложения использован frontend фреймворк Vue3. Для управления состоянием клиентского приложения использована библиотека Pinia. Для работы со стилизацией компонентов интерфейса использованы библиотеки TailwindCSS и DaisyUI.

Такой стек технологий позволяет эффективно выполнять задачи, поставленные перед разработкой клиентской части. Фреймворк Vue3 обладает более высокой производительностью по сравнению с предыдущей версией Vue2 и аналогами React и Angular.

Так же Vue3 более удобен для разработки и имеет более низкий порог вхождения. Поддержка TypeScript позволят разрабатывать более надежные и предсказуемые компоненты.

Менеджер управления состоянием приложения Pinia выбран, так как является официальной заменой Vuex в Vue 3. Он предлагает более современный и легкий подход к работе с состоянием.



В отличие от предшественника, Pinia не использует мутации, что делает код чище и проще для понимания. Кроме того, он изначально поддерживает реактивность, основанную на новой системе Composition API, что упрощает интеграцию с остальной частью приложения и повышает его гибкость.

TailwindCSS это библиотека позволяющая избежать работы со стилями компонентов напрямую. Благодаря гибкой системе заранее предопределенных классов библиотека позволяет сократить дублирование кода и ускорить разработку.

DaisyUI является самой большой и популярной библиотекой UI компонентов построенной на TailwindCSS. Она предлагает большой набор компонентов и тем для приложения которые гибко интегрируются с остальной системой по средствам применения классов к HTML-тегам.

Такая архитектура обеспечивает высокую степень кастомизации и контроля над визуальным представлением каждого элемента и способствует быстрой итеративной разработке с минимальным объемом CSS-кода.

Для разработки серверной части платформы выбран язык программирования Kotlin, этот язык относится к семейству языков компилирующийся в байткод поддерживаемый виртуальной машиной Java.

Для разработки так же выбрана версия Java 21 так как она является самой последней версией языка, для которой на момент написания работы заявлена длительная поддержка (LTS).

По сравнению с Java, Kotlin имеет более выразительный синтаксис и является более безопасным так как частью языка являются механизмы защиты от разыменования null указателей [24].

Фреймворком для написания серверной части приложения выбран Spring и его надстройка Spring Boot. Это решение широко признано в индустрии и считается стандартом при разработке современных backend-систем. Spring Boot обеспечивает удобную конфигурацию, ускоряет процесс разработки и упрощает развертывание приложений.

Экосистема Spring обладает большим количеством библиотек позволяющими расширить функционал фреймворка и интегрироваться с большинством сторонних систем. Для разработки серверной части платформы с состав дистрибутива были включены следующие библиотеки:

- 1) keycloak-admin-client – для интеграции системы с сервером авторизации Keycloak через REST API;
- 2) spring-data-jdbc – для интеграции с СУБД PostgreSQL;
- 3) kubernetes-client-java – для интеграции системы с подчиненным кластером Kubernetes;
- 4) spring-security – для обработки авторизационных токенов и создание политик доступа к предоставляемым сервисам.

Для сборки серверной части приложения выбрана система автоматической сборки Gradle, так как она является современной заменой устаревший системе сборки Maven, дает более гибкий и удобный инструментарий, а также на момент написания работы является стандартом в отрасли.

## **2.4 Разработка модели данных**

На платформе используется две базы данных: одна предназначена для хранения данных проектируемого сервиса, вторая – для хранения информации, связанной с авторизацией и аутентификацией пользователей. Такое разделение обеспечивает безопасность и удобство управления различными аспектами системы.

В рамках предметной области основными сущностями являются пользователь, группа пользователей, задача и результат выполнения задачи. Эти элементы формируют основу логической структуры данных и отражают ключевые процессы взаимодействия пользователей с платформой.

Каждая задача связывается с конкретным пользователем или группой пользователей. Результаты выполнения задач сохраняются с возможностью последующего анализа и получения обратной связи.

На рисунке 8 ER диаграмма, на которой изображены сущности базы данных и взаимосвязи между ними.

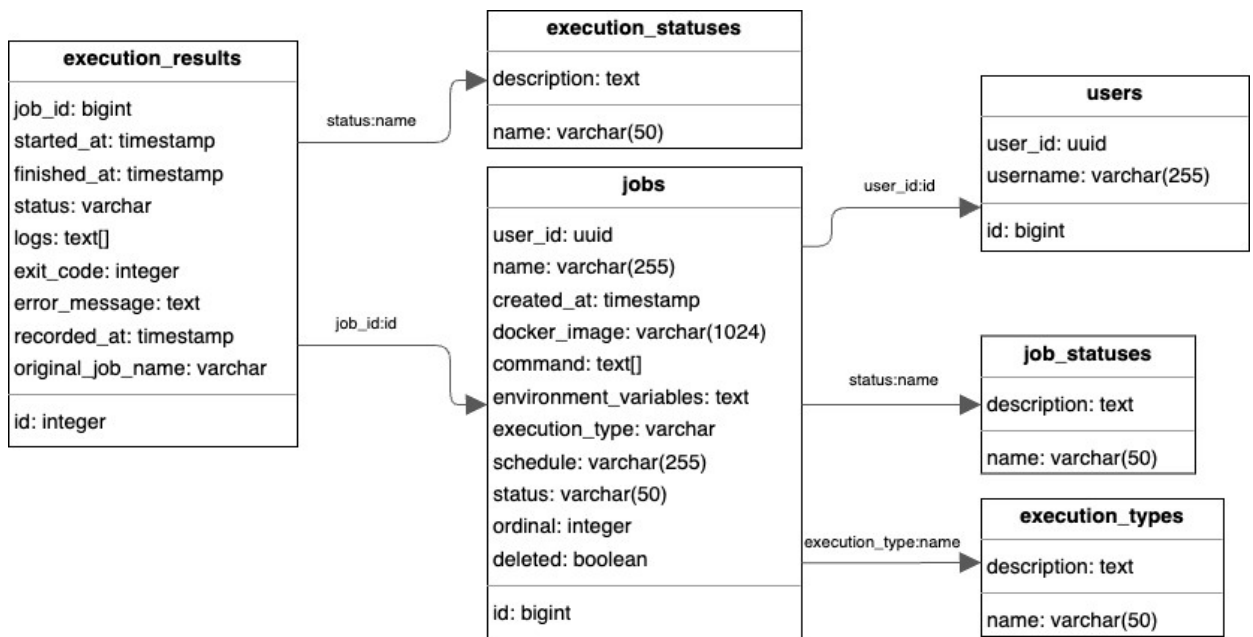


Рисунок 8 – Диаграмма отношений таблиц базы данных

В модели базы данных используется подход по реализации перечислений (enumerations) с использованием референсных таблиц, в которых первичным ключом является колонка name, на которую ссылаются таблицы использующие значения из данной таблицы в качестве перечислений. Колонка description при этом является справочной и служит только для хранения человекочитаемого описания значения перечисления. К референсным относятся таблицы job\_statuses, execution\_statuses, execution types которые хранят статусы задач и запусков.

Таблица jobs является центральной таблицей для хранения информации о запущенных задачах. Статусом задачи является одно из значений поля name референсной таблицы job\_statuses. Время создания записи в таблицу фиксируются автоматически в поле created\_at.

Таблица jobs имеет следующие поля:

- 1) id – bigint – Уникальный идентификатор задачи;
- 2) user\_id – uuid — Идентификатор пользователя, создавшего задачу;
- 3) name – varchar(255) – Имя задачи;

- 4) `created_at` – timestamp – Время создания задачи;
- 5) `dockerimage` – varchar(1024) – Docker-образ, который используется для выполнения задачи;
- 6) `command` – text – Команда или скрипт, который выполняется в рамках задачи;
- 7) `environment_varibales` – text – Переменные окружения, используемые для выполнения задачи;
- 8) `execution_type` – varchar – Тип выполнения задачи;
- 9) `schedule` – varchar(255) – Расписание выполнения задачи (cron-выражение);
- 10) `status` – varchar(50) – Статус задачи;
- 11) `ordinal` – integer – Порядковый номер задачи;
- 12) `deleted` – boolean – Флаг, показывающий, удалена ли задача.

В таблице `execution_result` располагаются результаты выполнения задач. Для создания отношения один ко многим с таблицей `job_statuses`, внешним ключом является поле `job_id`.

Таблица `execution_result` имеет следующие поля:

- 1) `id` – integer – Уникальный идентификатор результата выполнения;
- 2) `lob_id` – bigint – Идентификатор задачи;
- 3) `started_at` – timestamp – Время начала выполнения задачи;
- 4) `finished_at` – timestamp – Время завершения задачи;
- 5) `status` – varchar – Статус выполнения задачи;
- 6) `logs` – text – Массив логов, связанных с выполнением задачи;
- 7) `exit_code` – integer – Код завершения выполнения задачи;
- 8) `error_message` – text – Сообщение об ошибке;
- 9) `recordedat` – varchar – Время записи результата выполнения.

Таблица `users` хранит данные, дублирующие информацию о зарегистрированных пользователях, хранящаяся на сервере авторизации. Она необходима для соединения внешних ключей, в которых значением является идентификатор пользователя в системе авторизации.

Модель базы данных сервера авторизации играет важную роль в работе системы, так как информация о пользователях, а также их положение в ролевой модели хранится на сервере авторизации.

Ключевыми сущностями хранения информации о пользователях являются пользователь, атрибут пользователя, группа, роль. ER диаграмма части сервера авторизации по работе с данными пользователей и ролевой моделью представлена на рисунке 9.

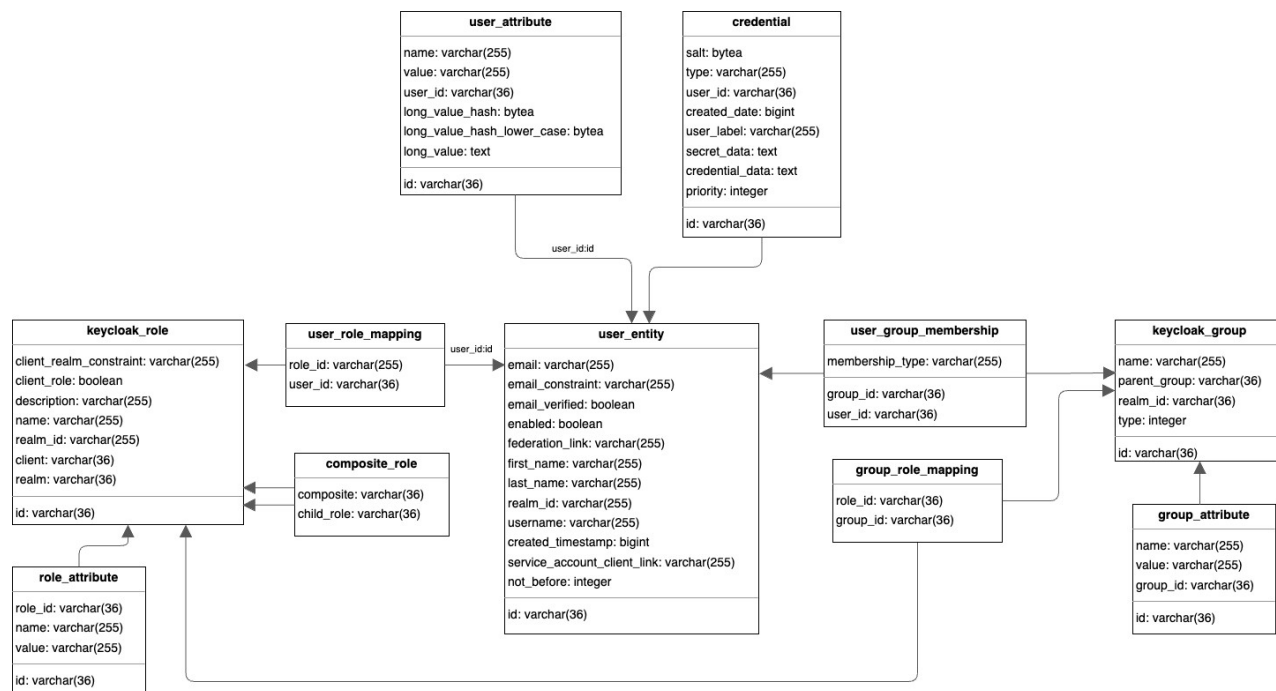


Рисунок 9 – Модель хранения данных пользователей на сервере авторизации

Таблица `user_entity` отвечает за хранение базовой информации об учетной записи. Первичным идентификатором таблицы является идентификатор пользователя, через который происходит связывание со всеми остальными таблицами, отвечающими за хранение пользовательских данных. Она включает в себя такие поля как:

- 1) `id` – Идентификатор пользователя;
- 2) `email` – Электронная почта;
- 3) `email_verified` – Признак подтверждения почты;
- 4) `enabled` – Признак активности учетной записи;
- 5) `first_name` – Имя;

- 6) last\_name – Фамилия;
- 7) realm\_id – Идентификатор тенанта пользователя;
- 8) user\_name – Имя пользователя;
- 9) created\_timestamp – Время создания учетной записи.

Пользователям можно добавлять произвольные атрибуты, которые хранятся в таблице user\_attribute.

Пароли пользователей хранятся в таблице credential в хешированном виде. Такой подход к хранению паролей пользователей является безопасным, так как хеш пароля невозможно однозначно преобразовать в пароль.

В таблице keycloak\_role хранится информация о ролях пользователей, существующих в системе. К роли можно добавлять произвольные атрибуты, которые хранятся в таблице role\_attribute.

Сервер авторизации поддерживает создание композитных ролей, путем связывания нескольких ролей с одной, родительской ролью. Связывание происходит через таблицу composite\_role.

Роли могут быть связаны с пользователем через отношение многие ко многим, которое реализовано через третью таблицу user\_role\_mapping.

Более предпочтительным способ связывания пользователей с ролями является использование механизма групп пользователей, который позволяет создать статический маппинг наборов ролей на пользователей системы, на основании их принадлежности той или иной группе.

В таблице user\_group хранится информация о группах пользователей. К каждой группе можно добавлять произвольные атрибуты, которые сохраняются в таблице group\_attribute. Это позволяет гибко настраивать свойства групп в зависимости от требований проекта и контекста использования.

К группе может быть присвоено несколько ролей, при помощи связывания многие ко многим через таблицу group\_role\_mapping. При связывании групп в иерархическую структуру пользователь наследует все роли родительских групп, в которых находится.

Пользователи и группы пользователей связаны отношением многие ко многим – пользователи могут состоять в нескольких группах и в одной группе может состоять несколько пользователей. Для реализации такого подхода используется связывание через третью таблицу `user_group_membership`.

## **2.5 Разработка ролевой модели**

Ролевая модель платформы реализована на механизмах управления ролями и группами Keycloak. Объединение пользователей в группы позволяет организовать совместное управление выполнением задач.

Группа пользователей может быть создана пользователем на странице профиля пользователя. При создании группы, пользователь становится ее администратором.

Так как группы не определены заранее и создаются пользователями в процессе использования платформы в названии группы должна присутствовать генерируемая компонента, гарантирующая уникальность идентификатора группы, так же необходимо предусмотреть возможность ввода имени группы пользователем.

Для того что бы соблюсти все заявленные к имени группы требованиям идентификатором группы является случайный UUID идентификатор. Для того что бы сохранить пользовательское название группы, создаются атрибут группы с ее названием.

Для каждой группы создаются роли, соответствующие необходимым уровням доступа к различным частям функционала платформы. Набор ролей является динамичным набором строковых констант и может быть расширен при появлении нового функционала.

Идентификатором роли также будет являться UUID, что гарантирует её уникальность в рамках системы. Имя роли сохраняется в виде строкового значения в атрибуте `name`, обеспечивая читаемость и однозначную идентификацию роли при взаимодействии с пользователями и сервисами.

В системе в рамках группы существуют следующие уровни доступа:

- 1) VIEW — просмотр задач, созданных в группе;
- 2) EDIT – изменение параметров запущенных задач, их остановка;
- 3) RUN – создание новых задач;
- 4) ADMIN – управление группой (добавление и удаление участников, назначение ролей, генерация токенов доступа).

Динамический характер набора ролей обеспечивает необходимую адаптивность системы к будущим расширениям функционала, позволяя добавлять новые уровни доступа.

## 2.6 Разработка каркасных макетов пользовательского интерфейса

Страница входа и регистрации позволяет пользователям авторизоваться или создать новый аккаунт. Форма входа представлена на каркасном макете на рисунке 10.

The wireframe shows a login page for 'Lambda'. At the top is a header bar with the 'Lambda' logo. The main content area is divided into two sections. On the left is a login form with a light gray background. It has the text 'Welcome back' and 'Login into your account'. Below this are two input fields: 'UserName' with the value 'foo\_bar@mail.ru' and 'Password' with masked characters '\*\*\*\*\*'. There is a 'Reset password' link below the password field. At the bottom of the form is a dark 'Login' button and a link 'Need account? Sign up'. On the right is the 'Lambda' logo, a large black cylinder, and the text 'Lambda' and 'Deploy. Execute. Optimize.'

Рисунок 10 – Каркасный макет страницы входа



Для пользователей, не имеющих учетной записи в системе, предусмотрена форма регистрации, изображенная на рисунке 11, которая доступна при переходе по ссылке, находящейся в нижней части формы входа.

The wireframe shows a web page for the 'Lambda' platform. At the top left is the 'Lambda' logo. The main content area is divided into two sections. On the left is a 'Create account' form with the subtext 'Sign up for an account to get started'. The form contains four input fields: 'UserName:' with the placeholder 'foo\_bar@mail.ru', 'Password:' with a masked password '\*\*\*\*\*', 'Organisation token:' with the placeholder 'as8f73kufds', and 'Full name:' with the placeholder 'Jhon Doe'. Below these fields is a dark 'Login' button and a link 'Need account? Sign up'. On the right is a large black cylinder icon representing a database. Below the icon is the word 'Lambda' in a bold sans-serif font, followed by the tagline 'Deploy. Execute. Optimize.'.

Рисунок 11 – Каркасный макет страницы регистрации пользователя

Данный функционал обеспечивает возможность самостоятельного создания учетной записи для новых пользователей, желающих получить доступ к ресурсам и возможностям платформы. Это упрощает процесс регистрации и снижает нагрузку на администраторов системы, позволяя автоматизировать первичное взаимодействие с пользователем.

Размещение ссылки для перехода к регистрации непосредственно на форме входа является интуитивно понятным решением, соответствующим стандартным практикам проектирования пользовательских интерфейсов. Такой подход повышает удобство использования платформы и способствует более быстрому вовлечению новых пользователей.

Процесс регистрации предусматривает обязательную проверку введенных данных, что обеспечивает корректность и безопасность создаваемых учетных записей.

На рисунке 12 показана страница задачи, на ней располагается вся информация о задаче.

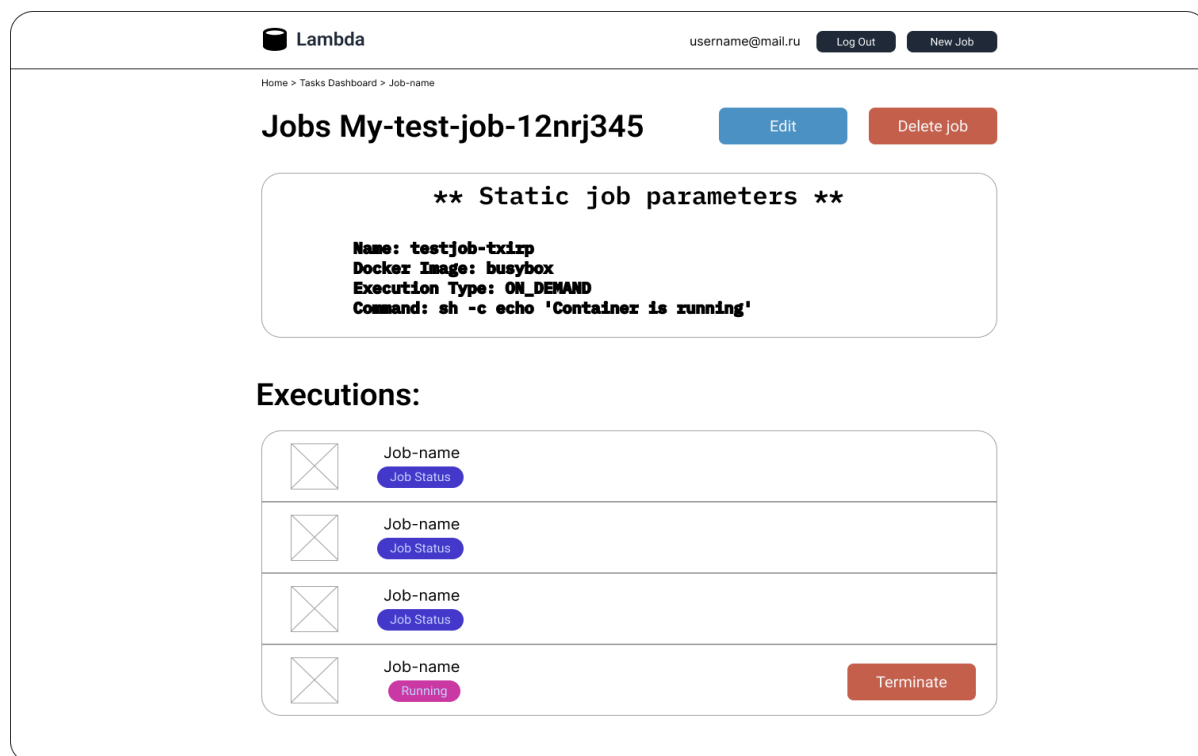


Рисунок 12 – Страница просмотра задачи

В верхней части страницы расположены две ключевые кнопки — редактирования и удаления задачи. Кнопка редактирования предоставляет пользователю возможность изменить параметры запуска задачи, которые будут применяться при всех последующих выполнениях. Это позволяет гибко настраивать работу задачи в зависимости от изменяющихся требований. Кнопка удаления задачи служит для прекращения всех будущих запусков и скрытия удалённой задачи из списка доступных пользователям, что помогает поддерживать актуальность и чистоту интерфейса.

В нижней части страницы размещён подробный список запусков задачи, отражающий историю и текущее состояние выполнений. Если задача запущена в данный момент, появляется кнопка для её завершения, позволяющая оперативно остановить выполнение, если это необходимо. Такой функционал обеспечивает пользователю полный контроль над процессом выполнения задач.

При взаимодействии с любым из элементов списка запусков пользователь автоматически переходит на отдельную страницу, где представлена детальная информация о конкретном запуске задачи. Там можно ознакомиться со статусом, логами и результатами выполнения, что облегчает мониторинг и анализ работы платформы на уровне отдельных задач.

На странице запущенных задач, изображенной на рисунке 13, располагается информация обо всех запущенных пользователем задачах.

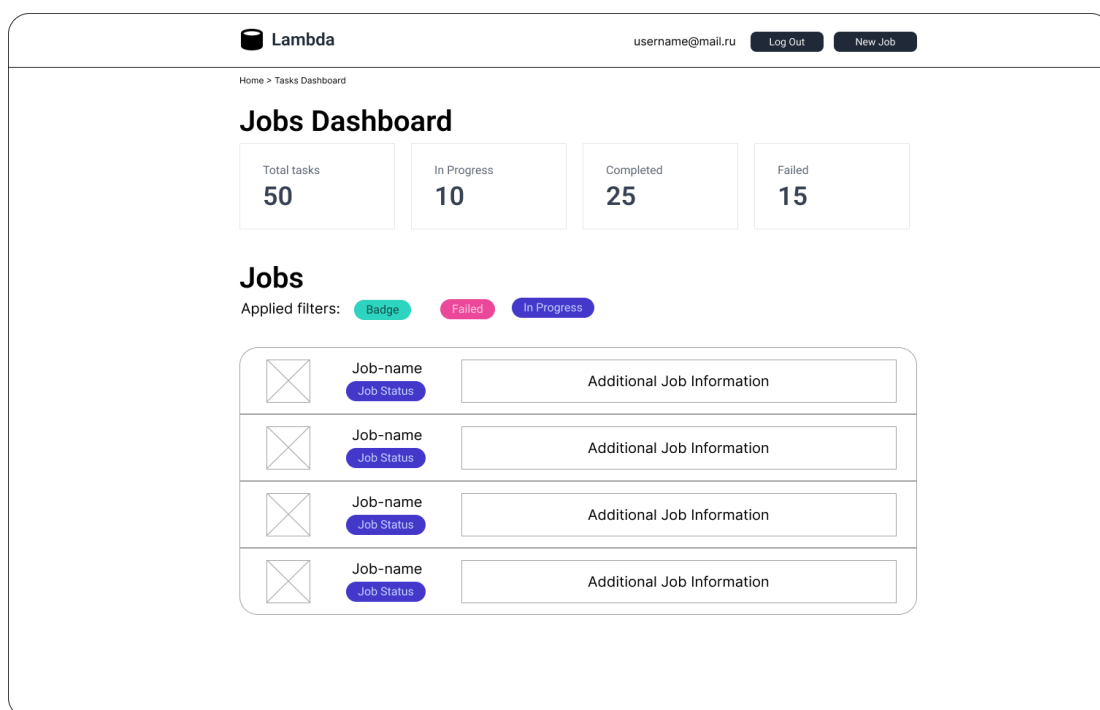


Рисунок 13 – Страница запущенных задач

В верхней части страницы располагается счетчик, отображающий общее количество задач, а также количество выполненных, запущенных и запущенных с ошибками задач. Каждый из этих счетчиков служит интерактивным элементом: при нажатии на него к списку запусков задач, расположенному ниже, автоматически применяется соответствующий фильтр.

Список задач, расположенный под счетчиками, содержит подробную информацию о каждом запуске и служит основным инструментом для мониторинга работы платформы. При взаимодействии с любым элементом из

этого списка пользователь переходит на страницу конкретной задачи, где доступна более подробная информация и возможность управления.

На рисунке 14 продемонстрирована страница профиля пользователя, на которой отображается информация о пользователе и о группах пользователей в которых он состоит.

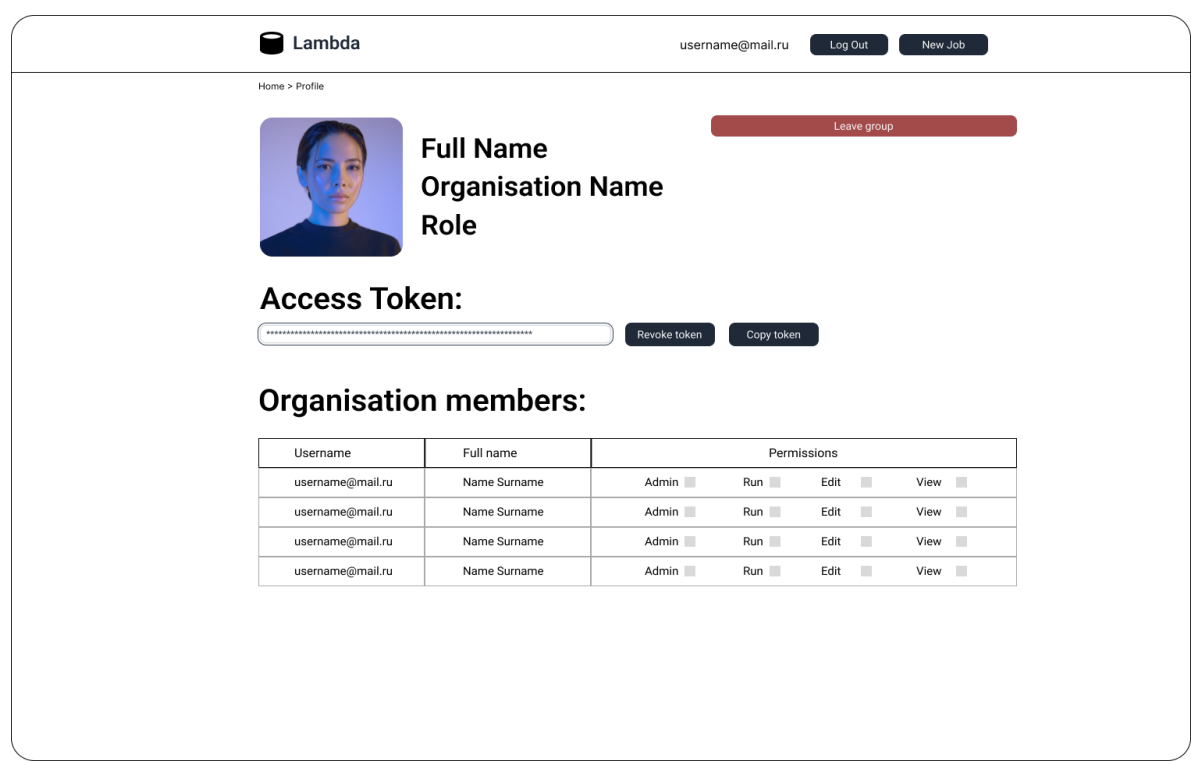


Рисунок 14 – Макет страницы профиля пользователя

Если пользователь состоит из какой-либо группе ему отображается кнопка выхода из группы.

Администратору группы доступна форма для генерации и копирования токена доступа в рабочее пространство группы. Этот функционал упрощает управление доступом и позволяет быстро предоставлять необходимые права для интеграции и взаимодействия с внешними сервисами.

В нижней части страницы расположен список участников группы, который динамически обновляется в зависимости от выбранной активной группы. Здесь отображаются все члены группы вместе с перечнем их текущих прав и уровней доступа, что обеспечивает прозрачность и удобство контроля. Если пользователь является членом группы администраторов, ему доступно управление правами доступа других членов группы.

## 2.7 Разработка программного интерфейса серверной части

Прикладной программный интерфейс (API) [25] серверной части системы выступает ключевым элементом архитектуры, обеспечивая взаимодействие между клиентами и сервисами разрабатываемой системы.

API в системе выполняет такие функции как:

- 1) служит единой точкой входа для всех клиентов;
- 2) предоставляет единый способ взаимодействия с системой.

Основными потребителями программного интерфейса разрабатываемого серверного решения является веб-клиент платформы, а также потребители webhook [26] сервисов, предоставляемых платформой.

При проектировании серверной части критически важен выбор подхода и архитектуры проектирования API. Этот выбор определяет подходы взаимодействия между клиентом и сервером, а также влияет на производительность, масштабируемость и простоту интеграции с разрабатываемым сервисом. В настоящее время в проектировании API наиболее распространены 3 подхода:

- 1) REST [27];
- 2) SOAP [28];
- 3) различные RPC протоколы [29].

В качестве архитектурного подхода к проектированию программного интерфейса выбран Representational State Transfer (REST).

Ключевыми аспектами в выборе REST как архитектурного подхода к проектированию программного интерфейса стали:

- 1) широкое распространение и поддержка: rest является отраслевым стандартом при проектировании веб-API;
- 2) использование стандартных протоколов HTTP/HTTPS;
- 3) простота интеграции;
- 4) использование текстового кодирования данных: упрощает и ускоряет отладку.

Прикладной программный интерфейс платформы предусматривает набор операций, позволяющих клиентам управлять ресурсами платформы. К основным ресурсам платформы можно отнести:

- 1) JwtToken – ресурс, представляющий набор данных, содержащих криптографические токены, необходимые для взаимодействия с конечными точками, защищенными механизмами авторизации;
- 2) User – ресурс, представляющий набор данных о пользователе, в том числе информацию о пользователе, о его ролях и группах, в которых он состоит;
- 3) GroupDescription – ресурс, представляющий группу;
- 4) Permission – доступ к той или иной части функционала платформы в рамках группы пользователей;
- 5) Job – ресурс, представляющий информацию о задаче, включает все параметры выполнения задачи, а также информацию о ее запусках;
- 6) ExecutionResult – ресурс, представляющий информацию о запуске задачи.

Логически все конечные точки программного интерфейса системы можно разделить по различным наборам признаков: по доступности из неавторизованной зоны, по ресурсу, к которому конечная точка позволяет получить доступ, по наличию влияния на состояние системы.

Из неавторизованной зоны доступны только конечные точки, позволяющие пользователю зарегистрироваться, пройти авторизацию или сбросить пароль. К этой группе конечных точек относятся:

- 1) POST /public/api/v1/authentication/reset-password;
- 2) POST /public/api/v1/authentication/log-in;
- 3) POST /public/api/v1/registration.

Все остальные конечные точки доступны из авторизованной зоны.

По ресурсу, к которому контрольная точка предоставляет доступ, конечные точки можно разделить на три группы: управление пользователями, управление групповым доступом и управление

запущенными задачами. К группе конечных точек, позволяющих управлять пользователями, относятся конечные точки доступные из неавторизованной зоны.

К группе конечных точек, позволяющих управлять групповым доступом пользователей относятся конечные точки:

- 1) POST /api/v1/role-model/refresh-join-group-token/groupId;
- 2) POST /api/v1/role-model/leave-group/groupId;
- 3) POST /api/v1/role-model/join-group;
- 4) POST /api/v1/role-model/exclude-member-from-group;
- 5) POST /api/v1/role-model/create-group;
- 6) POST /api/v1/role-model/change-permission;
- 7) GET /api/v1/role-model/group/groupId;
- 8) GET /api/v1/role-model/get-join-group-token/groupId.

К группе конечных точек, позволяющих управлять запущенными задачами, относятся:

- 1) GET /api/v1/job;
- 2) POST /api/v1/job;
- 3) POST /api/v1/job/webhook/run/id;
- 4) POST /api/v1/job/rerun/id;
- 5) POST /api/v1/job/delete/id;
- 6) POST /api/v1/job/cancel/id;
- 7) GET /api/v1/job/id.

К группе конечных точек, позволяющих менять состояние ресурсов системы относятся все конечные точки, использующие метод пост при выполнении http запроса.

В системе предусмотрены различные сценарии использования API для выполнения задач управления удаленными вычислениями через платформу.

К основным сценариям можно отнести:

- 1) авторизация пользователя;
- 2) создание новой группы пользователей;

- 3) добавление пользователя в группу;
- 4) исключение пользователя из группы;
- 5) добавление, отнимание доступов пользователя в рамках группы;
- 6) просмотр списка участников группы и из доступов;
- 7) получение списка запущенных задач и просмотр результатов выполнения;
- 8) создание новой задачи;
- 9) перезапуск, остановка, удаление задачи.

Пользователь может пройти авторизацию вызвав конечную точку POST /public/api/v1/authentication/log-in.

Если пользователь не имеет учетной записи в системе он может ее создать, вызвав конечную точку POST /public/api/v1/registration.

Ответом сервера при входе и регистрации будет пара JWT токенов [30], которые пользователь должен использовать для доступа к конечным точкам, защищённым механизмами авторизации. Токен доступа предъявляется клиентским приложением при каждом запросе к защищенным ресурсам API для подтверждения прав пользователя, обеспечивая безсеансовый (stateless) характер взаимодействия, так как серверу не требуется хранить информацию о сессии.

Если пользователь уже имеет учетную запись, но забыл от нее пароль, он может его сбросить, при вызове сервиса reset-password, который на этапе реализации прототипа не требует подтверждения учетных данных.

Групповая модель доступа к ресурсам разрабатываемой системы предполагает гибкую систему групп и доступов, позволяющую разграничивать доступ к ресурсам платформы у различных пользователей.

По умолчанию пользователь не состоит ни в какой группе и может создавать и просматривать задачи только от своего имени.

Для того что бы создать группу пользователей необходимо вызвать конечную точку POST /api/v1/role-model/create-group. Пользователь, создавший группу, будет назначен ее администратором.



Для того что бы добавить пользователя в группу предусмотрен механизм токенов доступа в группу. Администратор группы может запросить токен доступа вызвав конечную точку GET `/api/v1/role-model/get-join-group-token`. Дальше токен любым удобным способ отправляется пользователю, который должен вступить в группу.

Что бы присоединиться к группе необходимо вызвать конечную точку POST `/api/v1/role-model/join-group`. Пользователь будет добавлен в группу с доступом на просмотр по умолчанию. Для того что бы расширить перечень доступов пользователя в рамках группы, администратор группы должен произвести соответствующие операции.

Для добавления или удаления доступа у пользователя, являющегося членом группы, администратор должен воспользоваться конечной точкой POST `/api/v1/role-model/change-permission`.

Пользователь может покинуть группу используя конечную точку POST `/api/v1/role-model/leave-group/{groupId}`.

Администратор так же может исключить пользователя из группы используя конечную точку POST `/api/v1/role-model/exclude-member-from-group`.

Применение политик группового доступа происходит в реальном времени и не требует повторной авторизации пользователей.

Пользователь может создать задачу доступную только ему или членам группы по его выбору вызвав эндпоинт POST `/api/v1/job`.

Запросить список текущих задач можно вызвав эндпоинт GET `/api/v1/job`. В зависимости от членства в различных группах и наличия соответствующих доступов у пользователя, при запросе текущих задач он получит список задач доступных ему или членам группы, в которую он входит и имеет доступ на просмотр.

При наличии доступа пользователь может перезапустить задачи при помощи сервиса POST `/api/v1/job/rerun/id`, отменить выполнение задачи POST `/api/v1/job/cancel/id` или удалить задачу POST `/api/v1/job/delete/id`.

Если задача является веб-хуком, то пользователь может его вызвать через интерфейс или API платформы воспользовавшись конечной точкой POST /api/v1/job/webhook/run/id.

## **2.8 Разработка серверной части**

Серверная часть система является распределенным приложением, в котором монолитное веб приложение служит связующим компонентом между другими частями системы. Основу системы составляет веб-приложение, написанное на языке Kotlin и фреймворке для написания веб-приложений Spring Boot. В состав приложения входит веб-сервер, отвечающий за маршрутизацию запросов к конечным точкам.

Помимо разрабатываемого приложения ключевыми компонентами системы являются кластер Kubernetes, в котором разворачиваются сервисы платформы и пользовательские задачи, база данных PostgreSQL, выполняющая функции хранения данных веб-приложения и сервера авторизации, а также сервер авторизации Keycloak, выполняющий задачи управления токенами доступа пользователей и хранения информации о ролях и доступах пользователей.

REST API платформы спроектирован в соответствии с подходами ресурс-ориентированной архитектуры (ROA) [31] и использует стандарт OpenAPI 3.0 для описания конечных точек.

По ресурсу, к которому контрольная точка предоставляет доступ, конечные точки можно разделить на несколько функциональных групп. К ним относятся управление пользователями, которое охватывает регистрацию, авторизацию и настройку профилей, управление групповым доступом, включающее назначение прав и ролей внутри групп.

Версионирование прикладного интерфейса платформы осуществляется путем добавления префикса v1 перед названием группы конечных точек, таким образом при дальнейшем развитии платформы, при отсутствии обратной совместимости могут безопасно эволюционировать.

Система авторизации разрабатываемой системы использует протокол OAuth 2.0 [32] для авторизации пользователей. За генерацию и валидацию JWT-токенов [30] отвечает сервер авторизации Keycloak.

Механизмы авторизации, основанные на использовании JSON Web Token, передают данные о пользователе в незашифрованном, но недоступном для модификации виде.

Каждый JWT токен состоит из трех частей:

- 1) заголовок – содержит название алгоритма подписи токена;
- 2) полезная нагрузка (payload) – набор утверждений (claims) о пользователе;
- 3) подпись токена – HMAC-хэш, вычисленный сервером авторизации с использованием приватного ключа.

Так как JWT-токен передает полезную нагрузку в незашифрованном виде, утверждения о пользователе не должны содержать какой-либо чувствительной информации. Подмена утверждений содержащихся в токене невозможна так как при изменении набора утверждений, неизбежно изменится их хэш, вычисляемый с использованием приватного ключа, который известен только серверу авторизации. Таким образом гарантируется безопасность доступа к ресурсу платформы.

При проектировании авторизации платформы, критическим требованием была реализация платформы без редиректов, из-за этого было принято решение отказаться от использования форм авторизации Keycloak и реализовать собственный API авторизации, делегировав серверу авторизации лишь задачи генерации и валидации токенов.

Спроектированный веб-сервис изображен на рисунке 15, данный веб-сервис построен с использованием распространенного паттерна шестигранной архитектуры [33] и соблюдает принципы второго уровня модели зрелости Ричардсона [34].

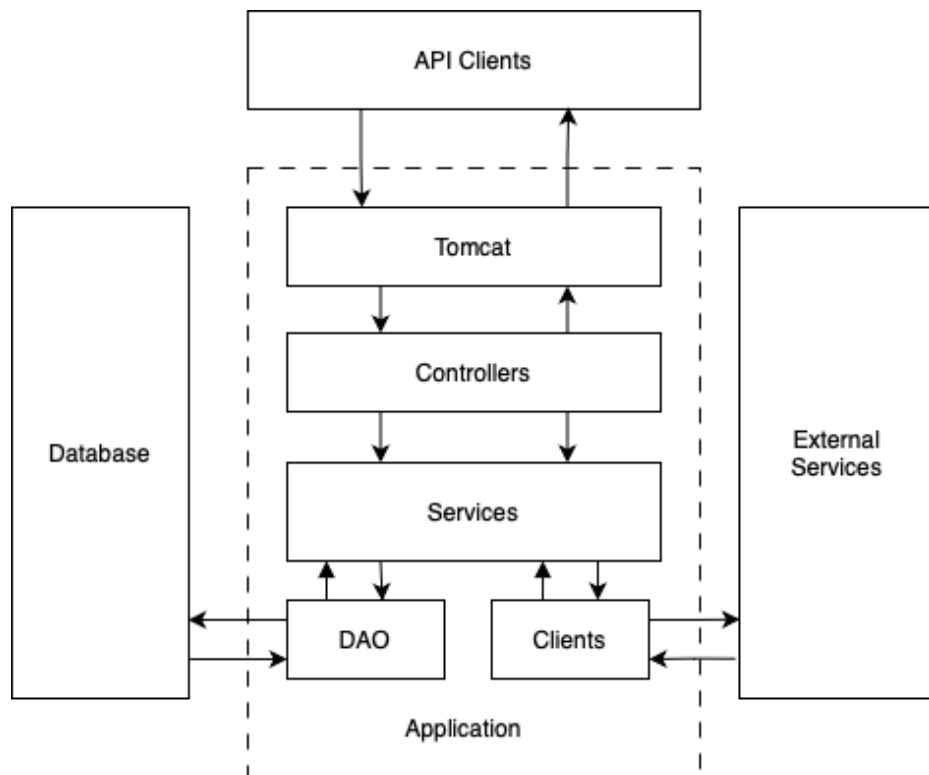


Рисунок 15 – Архитектура серверного приложения

На уровне приложения находятся сервисы платформы, отвечающие за реализацию бизнес-логики работы с доменными сущностями. К таким сервисам относятся:

- 1) AuthenticationService;
- 2) ExecutionService;
- 3) JobService;
- 4) RegisterUserService;
- 5) UserService.

В этих сервисах содержится бизнес-логика, отвечающая за согласованную работу системы и выполнение ей, возложенных на нее задач.

С Application слоем взаимодействует слой контроллеров, к которому относятся следующие классы:

- 1) AuthenticationController;
- 2) JobController;
- 3) RegistrationController;
- 4) RoleModelController;
- 5) UsersController.

Слой контроллеров отвечает за обработку входящих запросов и преобразование ответов сервисного слоя, в сущности, ResponseEntity.

Маршрутизацией трафика на конечные точки, объявленные в контроллерах, занимается веб-сервер Tomcat.

Для взаимодействия с базой данных используется слой DAO (Data Access Object).

Для работы с информацией, хранящейся в базе данных используется подход ORM (Object Relational Mapping) [35]. Данный подход реализован с использованием стандарта JPA (Java Persistence API) [36], который реализуется библиотекой Spring Data JDBC, предоставляющей лаконичный API для взаимодействия с базой данных, а также легко интегрируемой с основным фреймворком приложения Spring Boot.

На слое DAO представлены два типа сущностей – Entity и Repository.

Entity классы представляют собой отражение доменных сущностей хранящихся в базе данных, к ним относятся пользователи, задачи и результаты выполнения задач.

Repository в свою очередь является классом, позволяющим получить и менять состояние Entity которой он управляет. На слое DAO представлены следующие репозитории:

- 1) ExecutionResultRepository;
- 2) GroupAccessTokenRepository;
- 3) JobRepository;
- 4) UserRepository.

На интеграционном слое приложения располагаются сервисы, обеспечивающие взаимодействие с другими сервисами платформы. К таким сервисам относятся KeycloakService, реализующий интеграцию с сервером авторизации Keycloak и KubernetesService.

Для взаимодействия с Keycloak адаптер KeycloakService использует клиентскую библиотеку org.keycloak:keycloak-admin-client, которая является тонким клиентом для сервера авторизации, предоставляя удобный DSL

(Domain Specific Language) [37] интерфейс для взаимодействия с сервером авторизации. Библиотека берет на себя функции управления токенами доступа, кэшированными локально на стороне клиента, а также за преобразование сущностей DSL в вызовы REST API сервера авторизации.

Адаптер `KubernetesService` использует библиотеку `io.kubernetes:client-java` для взаимодействия с подчиненным кластером Kubernetes. Подключение к кластеру устанавливается при помощи файла конфигурации `kubeconfig`, путь к которому задан в переменной окружения `KUBECONFIG`.

Клиентская библиотека так же является тонким клиентом для Kubernetes и предоставляет удобный DSL API для генерации манифестов Kubernetes [38], а также предоставляет удобные абстракции для вызова методов `Kubernetes APIServer`.

Наличие DSL API для программного формирования манифестов Kubernetes значительно упрощает процесс создания и модификации объектов кластера. В свою очередь, предоставляемые абстракции для взаимодействия с `Kubernetes APIServer` инкапсулируют сложности низкоуровневых вызовов позволяя разработчикам адаптера сосредоточиться на бизнес-логике интеграции, а не на деталях протокола Kubernetes.

Для обеспечения изоляции пользовательских задач от сервисов платформы, реализован подход разделения пространств имен. Используются два основных пространства:

- 1) пространство `default` – служит для развертывания системных сервисов, таких как разработанный веб-сервис, сервер авторизации, база данных;
- 2) пространство `sandbox` – предназначено для развертывания пользовательских задач.

Такое разделение позволяет минимизировать риск несанкционированного доступа к ресурсам платформы и предотвратить риск возможных конфликтов именования ресурсов платформы, а также дает возможность независимого выделения ресурсов.

Для реализации возможности создания задач используются встроенные сущности Kubernetes Job и CronJob. KubernetesService управляет жизненным циклом объектов выполняя следующие операции:

- 1) создание ресурсов;
- 2) получение статуса выполнения;
- 3) получение логов выполнения;
- 4) остановка и удаление.

За генерацию манифестов отвечает фабрика [39] V1JobFactory. В этом классе инкапсулирована логика преобразования параметров создания задачи JobParameters в манифест Kubernetes.

Ключевые аспекты конфигурации включают:

- 1) установку metadata.name для идентификации Job и CronJob;
- 2) определение спецификации контейнера (V1Container), включая образ (image), команду (command) и переменные окружения (env);
- 3) установку политики перезапуска restartPolicy в значение Never для Pod, создаваемых в рамках Job, что соответствует семантике выполнения задачи до завершения;
- 4) конфигурацию backoffLimit для JobSpec, определяющую количество попыток перезапуска в случае сбоя;
- 5) для CronJob дополнительно указывается поле schedule и шаблон jobTemplate.

После запуска задачи сервис ExecutionService периодически опрашивает KubernetesService для актуализации статуса выполнения задачи. Собранный агрегированный информация о статусе выполнения задачи используется для обновления информации о задаче в базе данных, откуда в последствии данная информация будет доставлена пользователям.

Основное взаимодействие разработанного веб-сервиса с сервером авторизации Keycloak сокрыто в классе KeycloakService.

KeycloakService API предоставляет достаточный функционал для управления жизненным циклом учетной записи пользователя.

Интеграция с Keycloak осуществляется через экземпляр DSL-фабрики Keycloak. В процессе формирования обращений к API Keycloak ключевыми сущностями, в которые преобразуются доменные сущности платформы, являются:

- 1) UserRepresentation – DTO (Data Transfer Object) пользователя Keycloak;
- 2) CredentialRepresentation – DTO учетных данных пользователя;
- 3) GroupRepresentation – DTO группы пользователей Keycloak;
- 4) RoleRepresentation – DTO роли уровня realm Keycloak.

Для поддержания функциональности управления ролевым доступом реализованы механизмы генерации ролей уровня realm на основании идентификатора группы и доступа, который присваивается пользователю в рамках этой группы.

Такой подход к организации доступов пользователя позволят выдать пользователю разный набор доступов в разных группах, исключая возможность пересечения доступов между группами.

В последствии в токене доступа, в утверждении realm\_access.roles созданные роли принимают вид представленный в листинге 2.1. Такое представление доступов позволяет всем модулям приложения однозначно интерпретироваться доступы пользователя к различным частям предоставляемого функционала, а также закладывает возможности расширения набора доступов при возникновении такой потребности.

Листинг 2.1 – Пример представления ролей пользователя в токене доступа

```
1  "realm_access": {  
2    "roles": [  
3      "RUN/1b1d286b-1a33-4c17-8dda-d9a1b4037253",  
4      "ADMIN/1b1d286b-1a33-4c17-8dda-d9a1b4037253",  
5      "EDIT/216ef3cf-aa7c-4612-85f0-fad77f3a3637",  
6      "VIEW/1b1d286b-1a33-4c17-8dda-d9a1b4037253",  
7      "EDIT/1b1d286b-1a33-4c17-8dda-d9a1b4037253"  
8    ]  
9  }
```



Контроль за доступом к ресурсам платформы является критически важным аспектом разработки системы. Для обеспечения безопасности реализованы механизмы валидации токенов доступа пользователей, которые проверяют подлинность и актуальность учетных данных. Предусмотрено разделение конечных точек на авторизованную и неавторизованную зоны, что позволяет точно регулировать доступ к различным функциям и ресурсам платформы в зависимости от уровня прав пользователя.

Процесс валидации токенов доступа включает несколько ключевых этапов: проверку цифровой подписи, сроков действия токена и его эмитента. Такая проверка гарантирует аутентичность и актуальность учетных данных, предоставляемых пользователем. Только после успешного прохождения этих проверок система разрешает выполнение операций, требующих авторизации, обеспечивая безопасность.

Интеграция с сервером авторизации Keycloak, для валидации токенов доступа пользователей, осуществляется с использованием стандартных механизмов Spring Security для конфигурации авторизации через OAuth 2.0 Resource Server [40].

Ключевым аспектом конфигурации является указание параметра `issuer-uri`, указанного в листинге 2.2, определяющего адрес сервера, выпустившего JWT-токены доступа, которым доверяет веб-сервис.

Листинг 2.2 – Конфигурация сервера ресурсов

```
1  spring:
2    security:
3      oauth2:
4        resourceserver:
5          jwt:
6            issuer-uri: http://keycloak:8080/realms/users
```

Сервер авторизации, при обращении к конечной точке, `issuer-uri`, предоставляет набор публичных криптографических ключей, необходимых для процедур, основанных на асимметричной криптографии, таких как проверка цифровых подписей.

Веб-сервис использует эти публичные ключи для валидации токена в каждом запросе, что обеспечивает безопасность и подтверждает, что токен действительно был выдан доверенным сервером авторизации. Такой механизм позволяет надежно контролировать доступ к ресурсам платформы.

Пример ответа сервера авторизации на запрос публичных ключей представлен в листинге 2.3.

Листинг 2.3 – Ответ на запрос публичного ключа.

```
1  {  
2    "realm": "users",  
3    "public_key": "MIIBIj...QAB",  
4    "token-service": ".../realms/protocol/openid-connect",  
5    "account-service": ".../realms/account",  
6    "tokens-not-before": 0  
7  }
```

При получении ответа сервера авторизации с публичным ключом, Security middleware проверяют токен доступа по следующим параметрам:

- 1) подпись токена: проверяется с использованием публичного ключа, полученного от Keycloak по issuer-uri;
  - 2) сервер авторизации, выпустивший токен;
  - 3) срок действия: Проверяется, не истек ли срок жизни токена.
- Издатель (Issuer): Проверяется соответствие значения iss claim в токене сконфигурированному issuer-uri.

Для разграничения доступа к конечным точкам, требующим авторизованный доступ, используется библиотека Spring Security. Класс SecurityConfig является единым местом настройки политик доступа. Для сервисов платформы применяются следующие политики доступа:

- 1) пути, начинающиеся с /public/\*\* доступны всем пользователям;
- 2) HTTP-метод OPTIONS разрешен для всех путей для корректной работы механизма Cross-Origin Resource Sharing (CORS) [41], который используется при взаимодействии frontend-приложения с backend API из разных источников (доменов);

3) все остальные запросы требуют наличия валидного JWT-токена в заголовке `Authorization: Bearer <token> (authenticated())`;

4) отключение CSRF [42]: Защита от Cross-Site Request Forgery отключается, так как JWT-аутентификация является stateless и не подвержена данному типу атак в той же мере, что и сессионная аутентификация;

5) настройка обработки JWT: Секция `oauth2ResourceServer().jwt()` конфигурирует Spring Security для работы в режиме ресурсного сервера.

Здесь же подключается преобразователь токенов `KeycloakJwtTokenConverter`, преобразующий утверждения токена доступа в `Principal` объект Spring Security.

По завершении процедуры успешной валидации токена доступа, фреймворк Spring Security инициирует создание экземпляра объекта `Authentication`. Данный объект инкапсулирует информацию об аутентифицированном субъекте и предоставленных ему полномочиях.

Впоследствии, сформированный объект `Authentication` размещается в `SecurityContextHolder`. `SecurityContextHolder` функционирует как централизованное хранилище контекста безопасности, обеспечивая глобальный доступ к аутентификационным данным из любой точки кодовой базы приложения в рамках обработки текущего HTTP-запроса.

Для удобного доступа к данным пользователя и его ролям, извлеченным из токена, используется абстракция `getUserAuthentication` над хранилищем `SecurityContextHolder`, позволяющая преобразовать его в удобный вспомогательный DTO класс `UserAuthentication`, хранящий информацию об идентификаторе и ролях пользователя.

В рамках платформы реализован функционал глобального логгирования вызовов методов и обработки ошибок. Современные подходы к разработке для решение задач реализации сквозного функционала, затрагивающего большое количество различных мест программы, но при этом реализующего схожий функционал предлагают внедрения AOP (Aspect Oriented Programming) [43] для решения такого рода задач.

Реализация единой обработки ошибок выполнена с использованием класса `ExceptionHandler`. Аннотация `@ControllerAdvice` в Spring Boot позволяет классу перехватывать исключения, выброшенные из любого контроллера.

Класс `ExceptionHandler` перехватывает все исключения, возникающие в системе и производит над ними следующие действия:

- 1) в лог записывается информация об ошибке, включая HTTP-метод, URI запроса, тип исключения и его сообщение;
- 2) создается объект `RestError`, содержащий стандартизированную структуру данных об ошибке: HTTP-статус, общее сообщение об ошибке и детальное сообщение;
- 3) сформированный `RestError` оборачивается в класс `ResponseEntity` с соответствующим HTTP-статусом.

Это гарантирует, что клиент всегда получит ответ в предсказуемом JSON-формате, даже в случае непредвиденных сбоев на сервере.

Применение такого централизованного механизма обработки исключений унифицирует формат ответов об ошибках и существенно повышает качество сопровождения и разработки API. Консолидация логики обработки ошибок избавляет разработчиков от необходимости имплементации однотипного кода для перехвата исключений.

Логирование является ключевым инструментом для отладки, мониторинга и анализа поведения системы. Однако, внедрение логирующих вызовов в бизнес-логику методов может приводить к зашумлению кода и нарушению принципа единственной ответственности. Для решения этой проблемы реализован логирующий аспект `RestLoggingAspect`.

Преимуществом данного подхода является декларативность. Достаточно пометить требующий логирования метод аннотацией `@LogExecution`. На основе данной аннотации аспектно-ориентированный механизм автоматически внедряет необходимую логику для протоколирования выполнения метода, не требуя при этом модификации его исходного кода.

## 2.9 Разработка клиентской части

Для платформы разработано клиентское веб-приложение. Оно позволяет обеспечить взаимодействие пользователя с системой.

Основной целью разработки клиентского приложения является создание интуитивного, простого и понятного пользовательского интерфейса (UI), позволяющего в полной мере использовать доступный функционал платформы. Для разработки пользовательского интерфейса выбран фреймворк Vue3. К плюсам выбранного фреймворка можно отнести поддержку реактивности в интерфейсе, богатую экосистему библиотек, производительность и размер бандла скомпилированного приложения.

Для управления состояниями клиентского приложения выбран стэйт-менеджер Pinia. Pinia для фреймворка Vue3 поставляется как стейт-менеджер по умолчанию, что обеспечивает легкую интеграцию библиотеки с фреймворком, в то же время является самой удобной и гибкой реализацией менеджера состояний приложения среди аналогов.

К основным преимуществам Pinia можно отнести:

- 1) официальная рекомендация для Vue 3;
- 2) простота API, модульность;
- 3) интеграция с Vue DevTools;
- 4) поддержка TypeScript.

Для управления стилизацией HTML-элементов выбрана комбинация библиотек TailwindCSS и DaisyUI.

TailwindCSS реализует подход "utility-first". Вместо предопределенных классов компонентов, он предоставляет обширный набор низкоуровневых служебных классов, каждый из которых отвечает за одно конкретное CSS-свойство. Это позволяет разработчикам создавать полностью управляемые дизайны непосредственно в HTML-разметке. Использование TailwindCSS сокращает объем написанного CSS-кода, что упрощает поддержку проекта и повышает его масштабируемость.

DaisyUI функционирует как плагин для TailwindCSS, предоставляя набор готовых, стилизованных компонентов пользовательского интерфейса (кнопки, формы, карточки, модальные окна, оповещения и т.д.). Важно отметить, что эти компоненты построены с использованием утилит TailwindCSS. Это позволяет значительно ускорить разработку стандартных элементов UI, так как не требуется собирать их с нуля из отдельных утилит.

Использование DaisyUI даёт неоспоримое преимущество – консистентность дизайна интерфейса приложения, достигается минимальными усилиями разработчика. Таким образом выбор DaisyUI многократно ускоряет прототипирование и делает результат намного более качественным.

В совокупности, выбранный стек для разработки клиентской части значительно повышает удобство разработки и скорость прототипирования.

Структурно компоненты клиентского приложения разделены на несколько групп, объединенных общей функциональностью. К основным группам компонентов можно отнести следующие:

- 1) View – компоненты отвечающие за отображение страниц приложения;
- 2) Component – компоненты интерфейса, располагающиеся на страницах приложения;
- 3) Api – сервисы, позволяющие взаимодействовать с серверной частью приложения;
- 4) Store – модули хранилища состояния приложения, позволяющие различным компонентам приложения обмениваться информацией;
- 5) Utils – утилитарные компоненты, позволяющие решать специфические задачи, такие как управление и настройка сетевых соединений, работа с сессионными хранилищами, управление поллингом запросов и маршрутизацией.

Архитектура клиентского приложения представлена на рисунке 16.

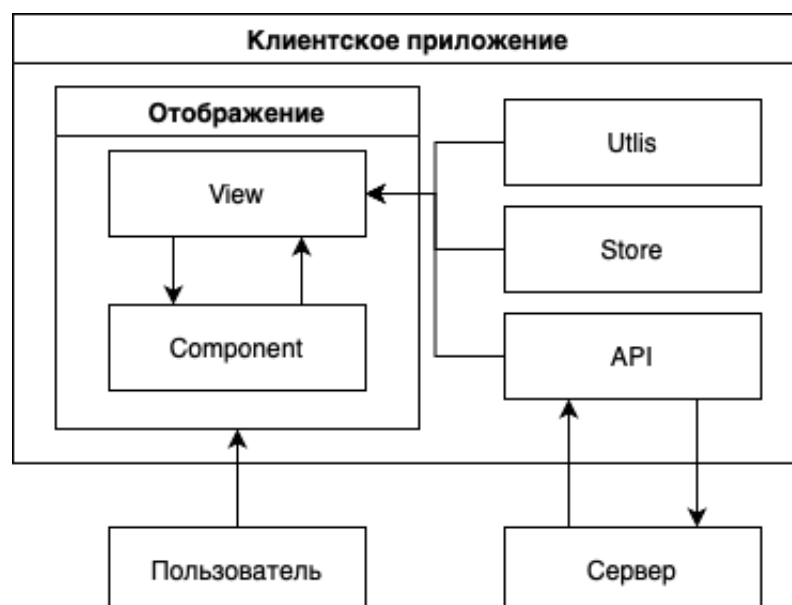


Рисунок 16 – Архитектура клиентского приложения

Клиентское приложение разработано с использованием компонентной архитектуры, предоставляемой фреймворком Vue.js. В рамках данной архитектуры, реализация каждого пользовательского сценария осуществляется через скоординированное взаимодействие одного или нескольких компонентов, которые отвечают за отображение и логику пользовательского интерфейса. Ключевую роль в управлении состоянием играют хранилища Pinia (обозначенные как "Store"), а взаимодействие с бэкендом обеспечивается через специализированные сервисы для работы с API.

Аутентификация и регистрация обеспечивают безопасный доступ к функциям платформы. В клиентской части эти процессы реализуются через набор компонентов: представление LoginView содержит формы LoginForm предназначена для сбора и первичной клиентской валидации учетных данных, таких как имя пользователя (или email) и пароль, необходимых для входа в систему, RegisterForm для создания и валидации новой учетной записи, и ResetPasswordForm для инициирования процедуры восстановления пароля. Компонент AuthSwitch обеспечивает навигацию между формами входа и регистрации внутри LoginView, позволяя пользователю переключаться между ними.

Экран входа представлен на рисунке 17.

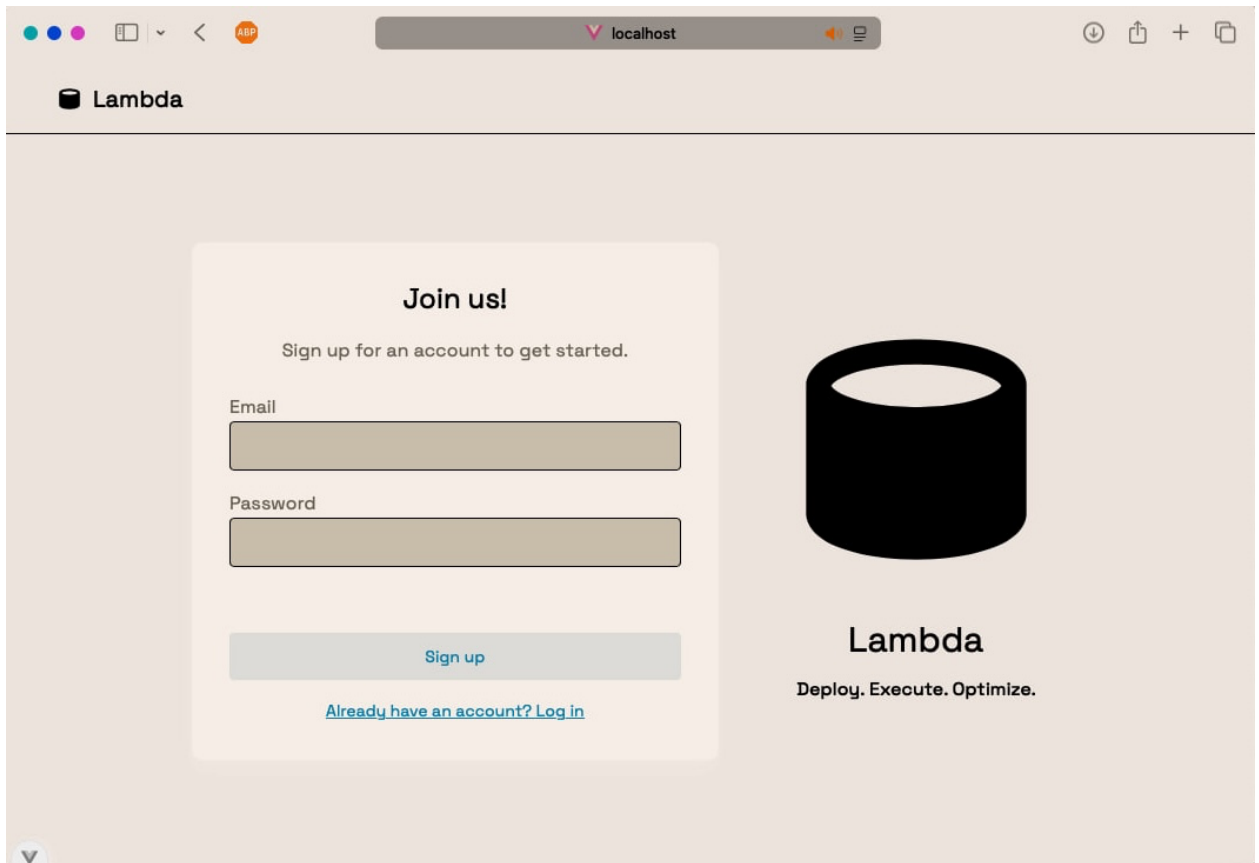


Рисунок 17 – Форма входа

При нажатии кнопки войти или зарегистрироваться, введенные пользователем данные отправляются на соответствующую конечную точку.

В случае успешной авторизации полученная пара JWT-токенов сохраняется в сессионное хранилище браузера `sessionStorage` посредством `StorageService`. Пользователь перенаправляется на страницу `tasks`.

В последствии токен доступа пользователя используется при выполнении любых запросов к серверной части платформы. Если время действия токена доступа истекает, сетевой клиент `Axios` выполняют `fallback` обработку ошибки 401 полученной от бекенда сервиса и выпускает новый токен доступа, обращаясь на соответствующий `url` с токеном `refresh_token`, который так же получает из сессионного хранилища. Время действия токена обновления значительно больше, чем токена доступа.

В случае ошибки, информация об ошибке, через систему уведомлений выводится на экран пользователя.



Экран просмотра задач представлен на рисунке 18.

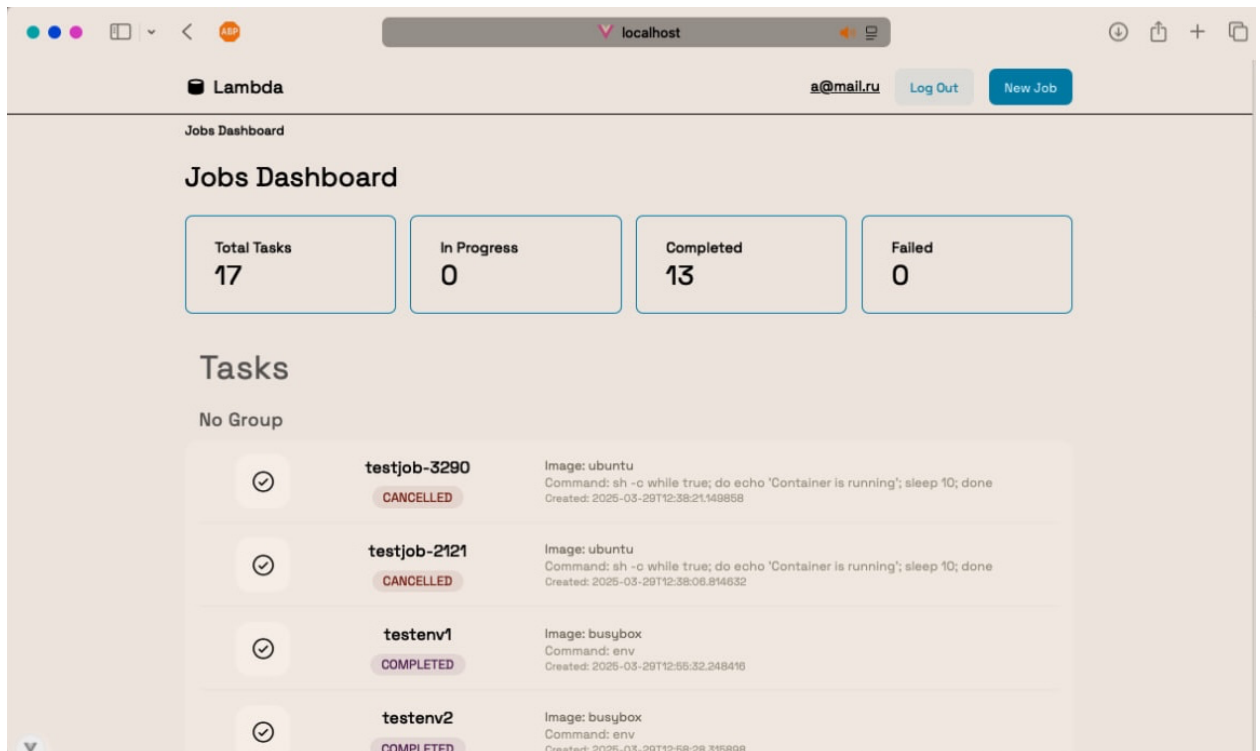


Рисунок 18 – Экран просмотра задач.

Полученные данные сохраняются в модуле состояния `tasksStore`. Компонент `TasksView` передает этот список задач в дочерний компонент `TasksList`, который отвечает за их отображение. `TasksList` итерируется по списку, для каждой задачи создает экземпляр компонента `TaskItem`. Компонент `TaskItem` отображает ключевую информацию о задаче и использует вложенный компонент `StatusBadge` для визуализации ее текущего статуса.

Представленная архитектура потока данных основана на использовании `tasksStore` как централизованного хранилища состояния для задач. Компонент `TasksView` инициирует передачу этих данных в нисходящем направлении по иерархии компонентов.

Такая организация обеспечивает упрощение процессов отслеживания изменений состояния и отладки. Это обусловлено последовательным прохождением данных через четко определенные интерфейсы компонентов. Результатом является минимизация вероятности возникновения побочных эффектов и облегчение анализа логики функционирования.

Форма создания новой задачи представлена на рисунке 19.

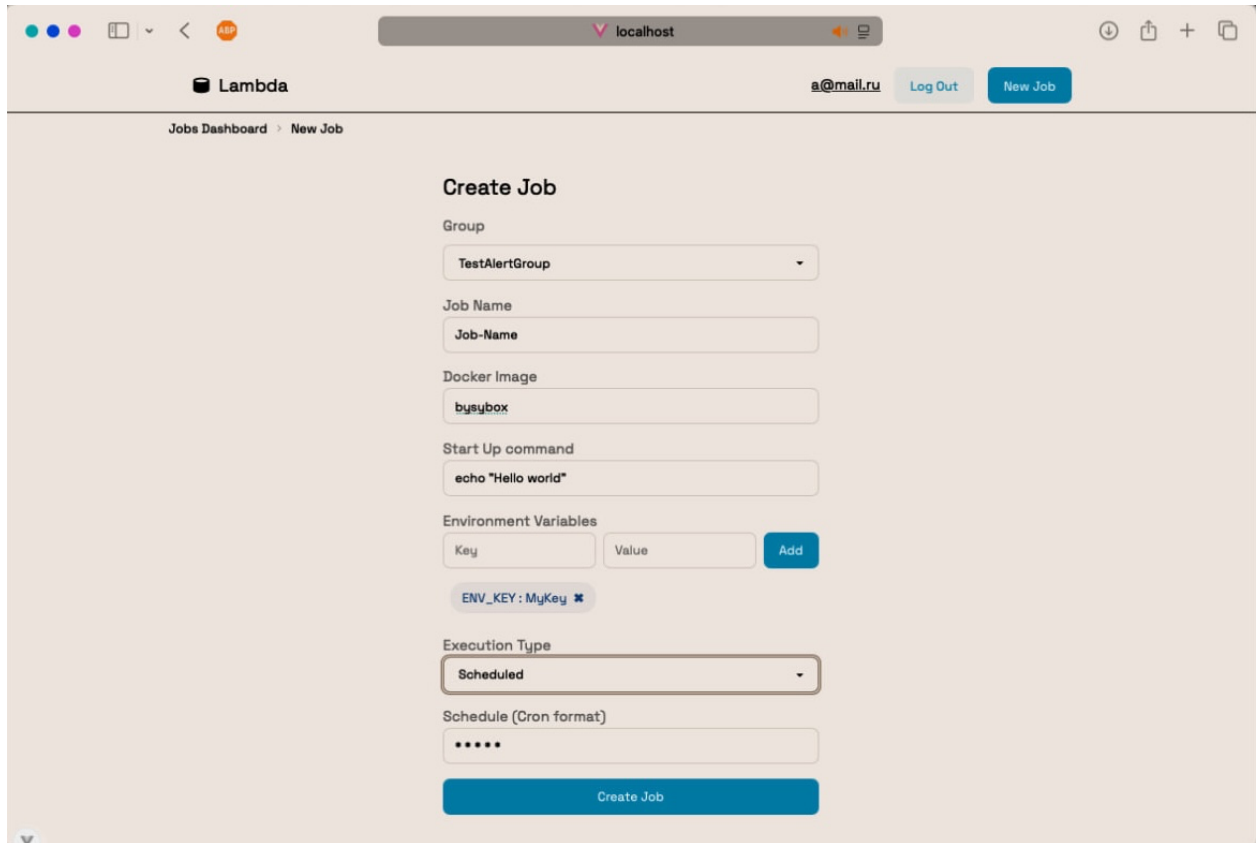
The screenshot shows a web browser window with the URL 'localhost'. The page title is 'Lambda'. In the top right corner, there is a user profile 'a@mail.ru', a 'Log Out' button, and a 'New Job' button. The main content area is titled 'Jobs Dashboard > New Job'. The 'Create Job' form includes the following fields: 'Group' (a dropdown menu with 'TestAlertGroup' selected), 'Job Name' (a text input with 'Job-Name'), 'Docker Image' (a text input with 'busybox'), 'Start Up command' (a text input with 'echo "Hello world"'), 'Environment Variables' (a section with 'Key' and 'Value' inputs, an 'Add' button, and a list item 'ENV\_KEY: MyKey'), 'Execution Type' (a dropdown menu with 'Scheduled' selected), and 'Schedule (Cron format)' (a text input with '\*\*\*\*\*'). At the bottom of the form is a large blue 'Create Job' button.

Рисунок 19 – Экран создания новой задачи

Для создания новых задач предназначена страница `CreateJobView`, состоящая из формы ввода параметров: имени задачи, типа выполнения (однократный запуск, webhook, cron), Docker-образа, команды и переменных окружения.

Для задач типа `CronJob` предусмотрено поле ввода Cron-выражения, корректность которого проверяется утилитой `CronValidator.ts`.

После заполнения и клиентской валидации данных формы, `CreateJobView` инициирует действие выполняет запрос к серверной части приложения с параметрами задачи, заданными пользователем. При успешном выполнении запроса сервер возвращает подтверждение об успешном создании задачи, пользователю отображается уведомление об успехе через `alertStore.showSuccess` с последующим перенаправлением на страницу созданной задачи. В случае возникновения ошибок при создании задачи, через компонент `alertStore.showError`, список возникших ошибок выводится на экран.

Для отображения подробной информации о конкретной задаче используется страница TaskView, показанная на рисунке 20.

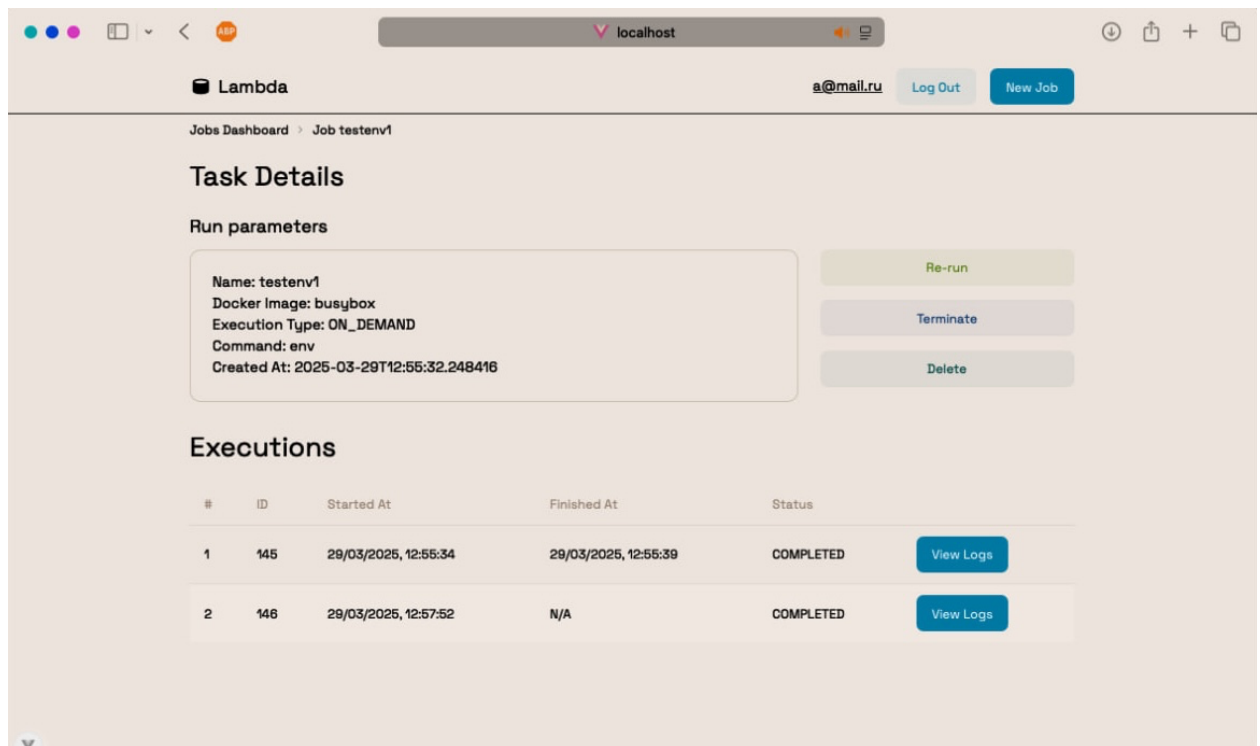


Рисунок 20 – Экран просмотра задачи

При навигации пользователя по маршруту `/task/:id`, соответствующий компонент, ответственный за отображение деталей задачи ("Task Details"), извлекает уникальный идентификатор задачи из параметров URL.

После получения идентификатора задачи, инициируется сервис поллинга. Данный сервис с заданной периодичностью выполняет запросы к серверной части платформы. Целью этих запросов является получение актуализированной информации о состоянии и параметрах текущей задачи (например, "Run parameters" и статус выполнения в секции "Executions"), обеспечивая динамическое обновление отображаемых данных без необходимости ручного обновления страницы пользователем. Полученные данные сохраняются в `tasksStore` и передаются дочерним компонентам: `TaskDetails` для отображения основной информации о задаче и `TaskExecutions` для вывода списка ее запусков со статусами и временными метками. При выборе конкретного запуска отображаются его логи в компоненте `TaskLogs`, изображенная на рисунке 21.

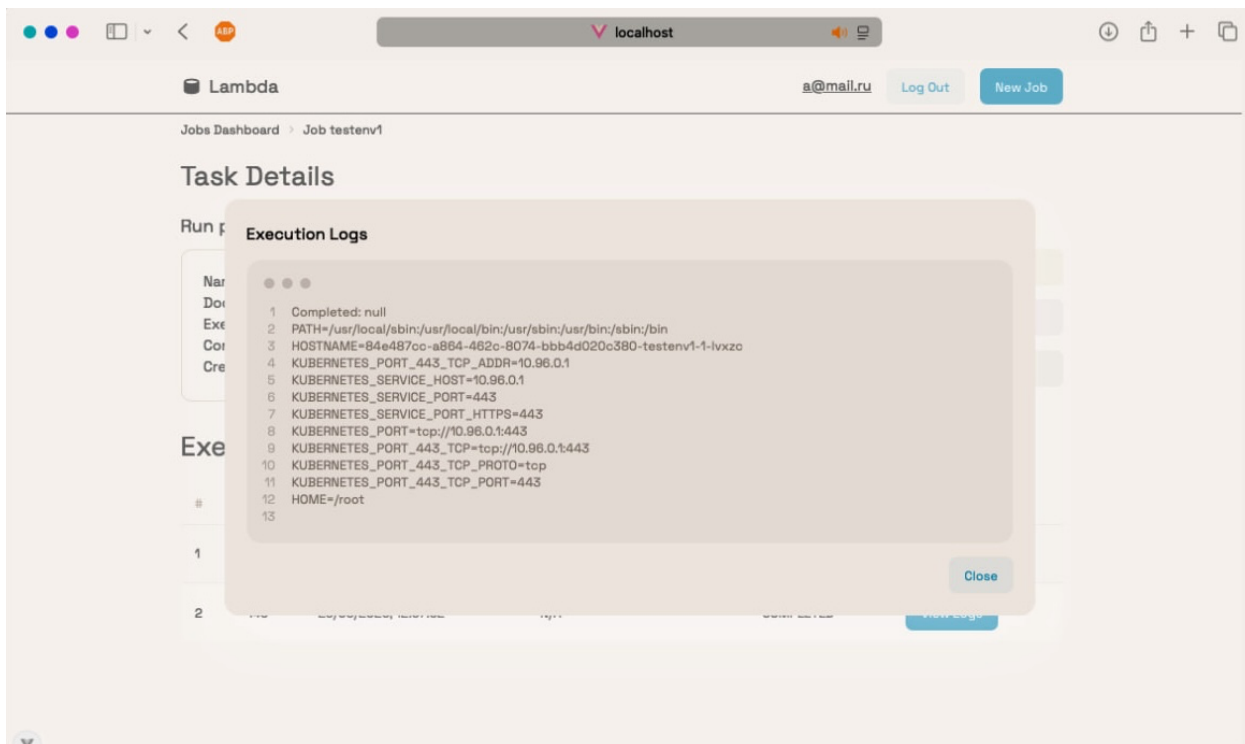


Рисунок 21 – Экран просмотра результатов выполнения задачи

Страница TaskView также предоставляет пользователю возможность выполнять действия над задачей, такие как перезапуск, отмена выполнения, удаление задачи, инициируя соответствующие вызовы API через JobsApi.

Раздел ProfileView предоставляет пользователю инструменты для управления персональными данными и участием в рабочих группах. При инициализации страницы происходит загрузка информации о текущем пользователе через UserApi.ts с последующим поллингом информации о текущем пользователе и списка групп, в которых он состоит.

Так же на данной странице отображается информация обо всех группах, в которых состоит пользователь. Пользователю предоставляется возможность переключения между активными группами, просмотра членов групп, а также редактирование полномочий.

Данные пользователя отображаются в компоненте ProfileCard. Список доступных групп представлен в GroupSelection. Выбор конкретной группы инициирует загрузку списка ее участников и их доступов в рамках группы, которые отображаются в MembersTable. Пользователям с соответствующими правами доступны функции администрирования группы, такие как управление доступа членов группы и исключение ее участников.

Реализованы также механизмы для создания новых групп и присоединения к существующим по токenu доступа. Управление токенами приглашений осуществляется через компонент AccessToken.

Страница управления профилем и групповым доступом представлена на рисунке 22.

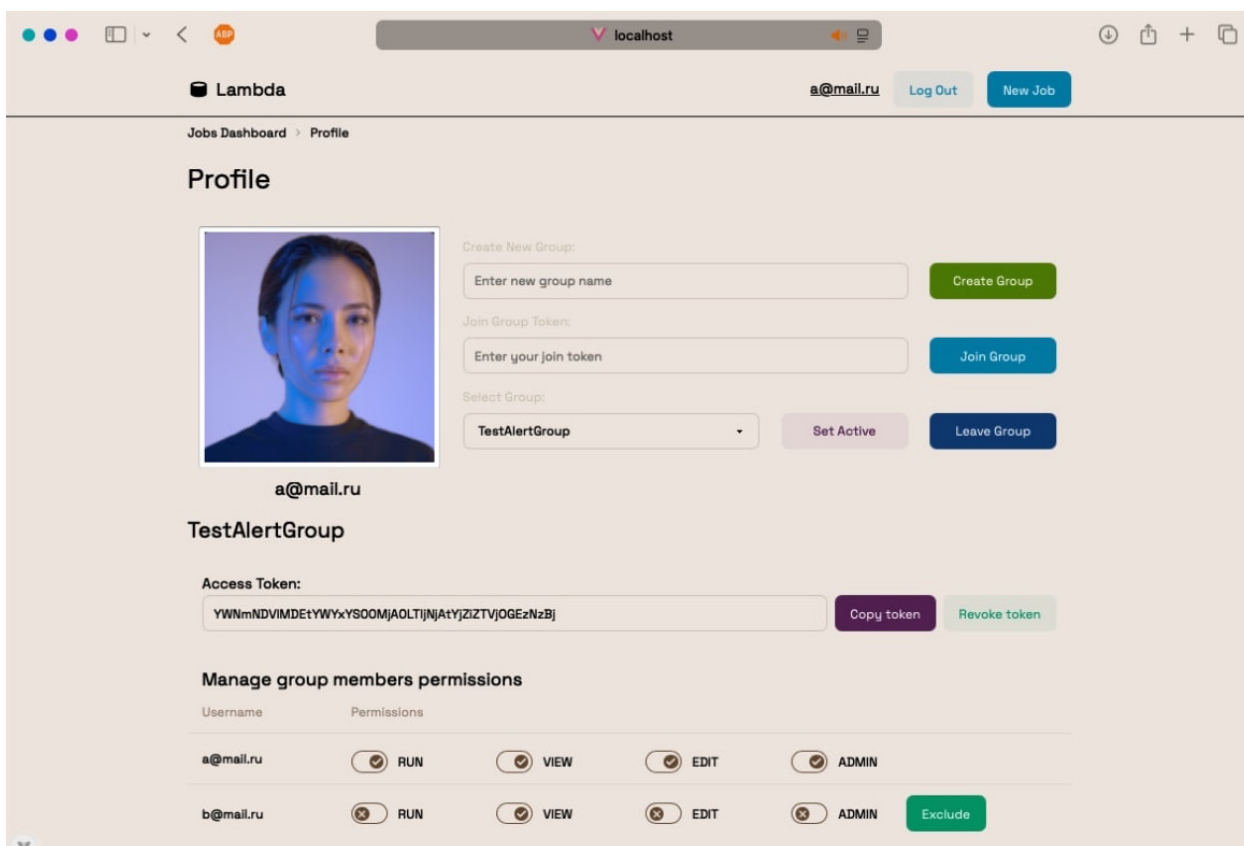


Рисунок 22 – Страница управления групповым доступом

## 2.10 Вывод

В ходе технической реализации автоматизированной платформа развертывания контейнеризованных функций в среде Kubernetes были применены современные технологии и методы разработки, обеспечивающие эффективность, удобство использования и безопасность системы.

При разработке клиентской части веб-сервиса был использован язык программирования JavaScript с фреймворком Vue.js. Благодаря использованию современных подходов к реализации пользовательского интерфейса обеспечено удобное взаимодействие пользователя с интерфейсом приложения, не требующее перезагрузок страницы.

Серверная часть платформы разработана с использованием языка программирования Java и фреймворка Spring Boot для обработки запросов от пользователей, управления данными и взаимодействия с базой данных, сервером авторизации и оркестратором контейнеров.

База данных PostgreSQL использована для хранения информации о пользователях и запущенных задачах. Для работы с учетными записями пользователей использован сервер авторизации Keycloak. Он позволил реализовать авторизацию пользователей с использованием спецификации JWT, а также предоставил механизмы для реализации гибкой ролевой модели с динамическим набором групп и доступов.

Для развертывания приложения и обеспечения его доступности и масштабируемости был использован Kubernetes, зарекомендовавший себя как надежное и удобное решение для управления контейнеризованными приложениями.

В результате разработки было создано современное решение, предоставляющее мощный и удобный пользовательский и программный интерфейс для управления распределенными вычислениями.

Применение передовых технологий, удобный интерфейс делают этот сервис удобным инструментом для использования в большом количестве прикладных задач.

## 3 ВНЕДРЕНИЕ И ЭКСПЛУАТАЦИЯ

### 3.1 Развертывание платформы в среде Kubernetes

Важным аспектом разработки информационной системы является размещение готового решения на целевой инфраструктуре. При выборе конфигурации серверного оборудования важно учитывать архитектурные особенности разработанной системы. В текущем разделе работы представлены сведения об архитектуре развертывания приложения, требования к низлежащей инфраструктуре кластера Kubernetes, описание процесс установки основных компонентов платформы.

Вся разработанная система развертывается в Kubernetes. Это обеспечивает отказоустойчивость, масштабируемость, а также возможность использования большого количества серверов для развертывания пользовательских задач.

Высокоуровневая диаграмма развертывания компонентов системы представлена на рисунке 23.

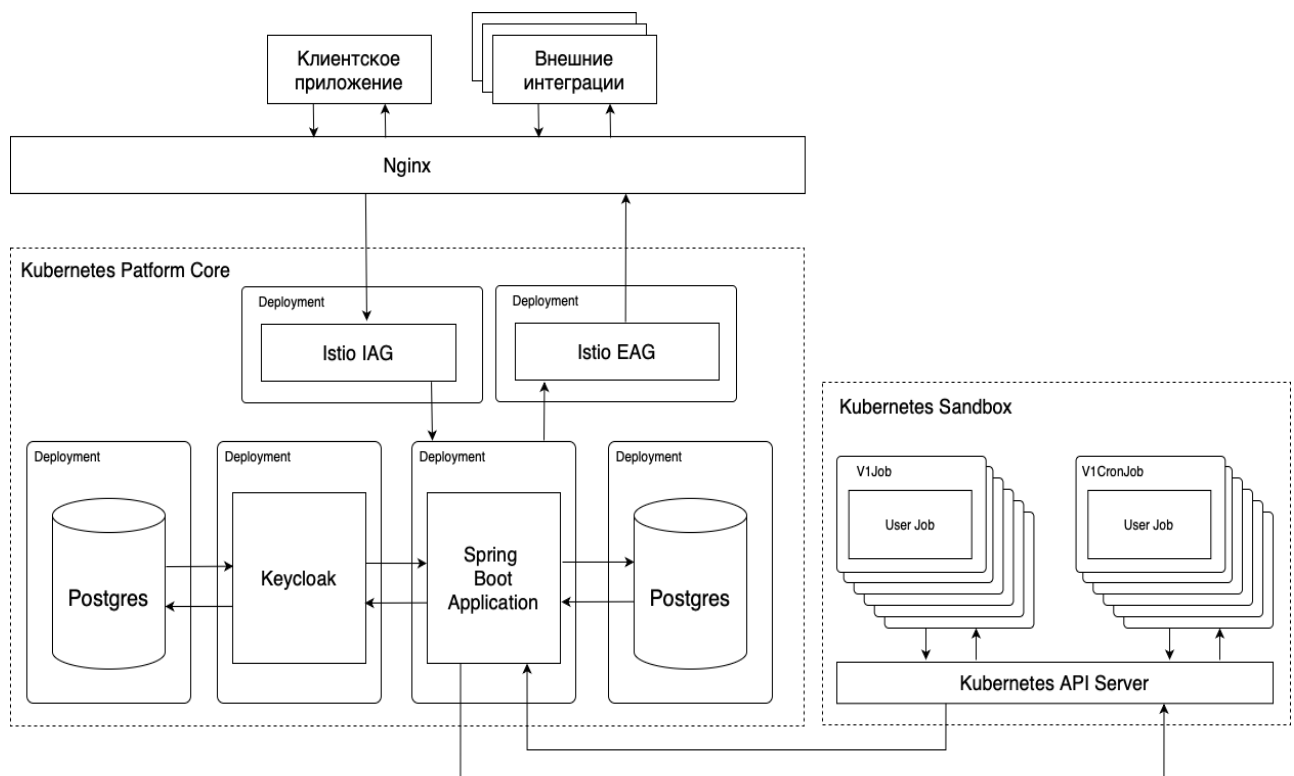


Рисунок 23 – Диаграмма развертывания

Все ключевые компоненты системы разворачиваются в platform-core пространстве имен и каждый представляет собой отдельный Deployment компонент Kubernetes. В то же время пользовательские компоненты разворачиваются в изолированной среде sandbox и являются объектами V1Job и V1CronJob в зависимости от конфигурации компонента.

Внешний пользовательский трафик поступает в систему через обратный прокси Nginx, который в тоже же время является SSL-терминатором [45] и корнем раздачи статических файлов.

Для успешного развертывания и функционирования автоматизированной платформы развертывания контейнеризованных функций необходимо обеспечить соответствие инфраструктуры следующим требованиям:

Требуется наличие функционирующего кластера Kubernetes. Для использования всех возможностей и обеспечения совместимости с актуальными API рекомендуется версия 1.25 или выше.

Кластер должен иметь стандартную конфигурацию с работающими компонентами Control Plane (API Server, etcd, Scheduler, Controller Manager) и Worker Nodes (Kubelet, Kube-proxy, Container Runtime) [46].

Определение необходимого объема вычислительных ресурсов для обеспечения функционирования системы является многофакторной задачей. Требования к аппаратной и программной инфраструктуре не являются статичными и напрямую коррелируют с прогнозируемой эксплуатационной нагрузкой. Минимальные требования к рабочему узлу для базового развертывания без значительной нагрузки:

- 1) Backend: ~1 CPU, ~1-2 GiB RAM;
- 2) Keycloak: ~1 CPU, ~1-2 GiB RAM (может требовать больше при большой базе пользователей/сессий);
- 3) PostgreSQL: ~1 CPU, ~1-2 GiB RAM (+ дисковое пространство);
- 4) пользовательские задачи (sandbox): Ресурсы зависят от самих задач.



Необходимо проводить нагрузочное тестирование [48] для определения оптимальных лимитов и запросов ресурсов (requests/limits) для подов платформы и планировать ресурсы узлов с запасом.

Узлы кластера Kubernetes должны иметь сетевой доступ к реестру контейнеров Docker Hub, где хранятся образы Backend, Frontend и базовые образы для пользовательских задач.

Развертывание платформы осуществляется в среде Kubernetes и базируется на применении манифестов, поставляемых в составе релизного дистрибутива, описывающих требуемое состояние системы.

Процесс включает последовательное создание и настройку всех необходимых ресурсов Kubernetes для запуска основных компонентов платформы, таких как серверное приложение, система аутентификации, база данных, а также настройку сетевого взаимодействия и подготовку изолированного окружения (sandbox) для выполнения пользовательских задач.

Процесс развертывания серверной части включает подготовку исполняемого артефакта, его контейнеризацию и последующее развертывание в кластере Kubernetes.

Первым шагом является сборка приложения с использованием системы сборки Gradle [49]. Команда сборки `./gradlew build` создает исполняемый JAR-файл в директории `build/libs`.

После создания образ загружается в реестр контейнеров [50], доступный для кластера Kubernetes. Этот процесс может быть выполнен вручную, так и автоматизирован в CI/CD пайплайне.

Для развертывания Backend в Kubernetes используются следующие основные типы манифестов:

- 1) Secret – используется для хранения чувствительных данных, таких как пароли и приватные ключи;
- 2) ConfigMap – используется для хранения нечувствительной конфигурации в виде пар ключ-значение;

3) Deployment – основной манифест, описывающий желаемое состояние для подов Backend приложения;

4) Service – определяет способ доступа к подам Backend внутри кластера.

После подготовки всех YAML-файлов манифестов они применяются к кластеру Kubernetes.

Kubernetes создаст необходимые объекты, скачает Docker-образ и запустит поды Backend приложения в соответствии с описанием в Deployment. После успешного запуска подов и прохождения readiness-проб, Service начнет направлять трафик на них. На этом этапе серверная часть готова к приему запросов от Frontend или других компонентов системы.

Клиентская часть платформы, разработанная на Vue3, представляет собой одностраничное приложение (SPA) [51], которое взаимодействует с пользователем и отправляет запросы к API серверной части.

Развертывание Frontend включает сборку статических артефактов и их размещение на веб-сервере, доступном для пользователей. Исходный код Vue3 приложения компилируется в набор статических файлов (HTML, CSS, JavaScript) с помощью инструментов сборки, таких как Vite или Vue CLI. Сборка запускается командой `npm run build`.

Результат сборки (директория `dist`) содержит все необходимые файлы для работы приложения в браузере.

Для обслуживания статических файлов Frontend в Kubernetes использован веб-сервер Nginx, который раздает статические файлы.

После развертывания экземпляра PostgreSQL и перед полноценным запуском серверной части, необходимо инициализировать схему базы данных. Это включает создание таблиц, индексов, внешних ключей.

В разработанной системе для управления схемой базы данных используется инструмент миграций Liquibase. Этот инструмент интегрирован в Spring Boot приложение и позволяет версионировать изменения схемы БД и автоматически применять их при старте приложения.

Сервер авторизации Keycloak так же использует Liquibase для организации миграций базы данных.

Миграции Liquibase поставляются вместе с дистрибутивом приложения и применяются при его запуске.

При последующих запусках пода Backend или при развертывании новых версий приложения с новыми скриптами миграций Liquibase будет применять только те миграции, которые еще не были выполнены, обеспечивая консистентность схемы БД.

### **3.2 Реализация нефункциональных требований**

Помимо реализации заявленной функциональности, описанной в предыдущих разделах, для создания надежной и эффективной системы критически важно уделить внимание нефункциональным требованиям (НФТ). В рамках разработки особое внимание было уделено трем нефункциональным требованиям: безопасности, масштабируемости и отказоустойчивости. Выбор архитектурных решений, стека технологий и конкретных механизмов реализации был во многом продиктован необходимостью обеспечения этих качеств.

Для реализации механизмов авторизации используется внешний сервер идентификации Keycloak, интегрированный с серверным приложением посредством стандартов OAuth 2.0 и OpenID Connect.

Backend, выступая в роли Resource Server, отвечает за валидацию JWT-токенов доступа, получаемых от Keycloak, извлечение информации о пользователе и его ролях, а также за принудительное применение правил авторизации на уровне API сервисов и отдельных операций с использованием возможностей фреймворка Spring Security.

Поскольку платформа предназначена для запуска произвольного пользовательского кода, критически важно предотвратить любое нежелательное взаимодействие этого кода с компонентами ядра системы или с кодом других пользователей.

Для достижения этой цели реализован механизм строгой изоляции на уровне оркестратора Kubernetes – обособленное пространство имен.

Все пользовательские задания, создаваемые платформой в виде объектов Job или CronJob, выполняются исключительно в пределах этого изолированного окружения, что фундаментально ограничивает их область видимости и потенциальное воздействие на остальную инфраструктуру кластера.

Использование Kubernetes в качестве среды развертывания предоставляет возможности для горизонтального масштабирования как компонентов самой платформы, так и ресурсов, выделяемых для выполнения пользовательских задач, позволяя динамически адаптироваться к изменениям нагрузки и обеспечивать стабильную работу системы.

Серверная часть является stateless приложением и развертывается с использованием объекта Deployment. Этот объект позволяет декларативно управлять количеством работающих экземпляров каждого компонента.

При возрастании нагрузки администратор кластера может вручную увеличить количество реплик для Backend и других компонентов платформы, и Kubernetes автоматически запустит дополнительные поды, а Service (или Ingress) обеспечит распределение входящих запросов между всеми доступными экземплярами, повышая таким образом пропускную способность и отказоустойчивость сервиса.

### **3.3 Тестирование платформы**

Тестирование позволяет выявить ошибки, узкие места и несоответствия на ранних стадиях, что снижает затраты на их исправление и повышает общее качество конечного продукта.

На текущем этапе проект находится в стадии активного прототипирования и разработки. В связи с этим, API платформы и внутренняя структура компонентов подвержены частым изменениям. По этой причине было принято решение временно отказаться от реализации

покрытия кода модульными и интеграционными тестами. В условиях нестабильного API и быстрого развития функциональности, затраты на постоянную поддержку тестовой базы превысили бы получаемую пользу, замедляя процесс разработки прототипа.

Было проведено ручное функциональное тестирование для проверки корректности работы прототипа. Это тестирование охватывало как прямое взаимодействие с программным интерфейсом (API Testing), так и проверку пользовательских сценариев через веб-интерфейс, что позволило убедиться в работоспособности основного функционала. Для тестирования API использовались инструменты, такие как Postman, позволяющие формировать HTTP-запросы к эндпоинтам и анализировать ответы, в то время как E2E-тестирование выполнялось непосредственно в веб-браузере.

В результате проведенного тестирования, была подтверждена корректная работа основного функционала прототипа и соответствие его поведения ожидаемому на текущем этапе разработки. Проверка ключевых пользовательских сценариев через веб-интерфейс, а также тестирование взаимодействия с API посредством специализированных инструментов, не выявили значительных дефектов, которые могли бы препятствовать дальнейшему развитию или ставить под сомнение жизнеспособность реализуемых подходов. Это позволяет заключить, что прототип успешно демонстрирует заложенную в него базовую функциональность.

### **3.4 Вывод**

В данной главе было представлено описание процессов внедрения и эксплуатации разработанной системы. Были подробно рассмотрено влияние принятых архитектурных решений на потребительские характеристики и процессы эксплуатации продукта.

Архитектура платформы построена на основе многоуровневой модели с четким разделением ответственности. Использование Kubernetes в качестве центрального элемента для оркестрации задач и развертывания компонентов,

Nginx в качестве точки входа и обратного прокси, Keycloak для централизованной аутентификации и авторизации, а также PostgreSQL для персистентного хранения данных, формирует надежную и современную основу для создания легко масштабируемой распределённой системы.

Логическое разделение на пространство имен ядра платформы (platform-core) и изолированное окружение для пользовательских задач (sandbox) является ключевым архитектурным решением, повышающим безопасность и управляемость системы.

Выбранный технологический стек, включающий Kotlin и Spring Boot для Backend, Vue3 для Frontend, соответствует требованиям к производительности, удобству разработки и интеграции с экосистемой Kubernetes.

Данная глава продемонстрировала, что разработанная платформа обладает продуманной архитектурой и реализована с использованием современных технологий, обеспечивающих необходимую функциональность, безопасность и масштабируемость. Были детально описаны механизмы взаимодействия с Kubernetes, Keycloak и базой данных. Рассмотрены подходы к обеспечению нефункциональных требований, таких как горизонтальное масштабирование, а также изоляция пользовательских задач. Проведенное на этапе прототипирования функциональное тестирование подтвердило работоспособность основных сценариев использования. Представленная техническая реализация создает прочную основу для дальнейшего развития и внедрения платформы.

## ЗАКЛЮЧЕНИЕ

В ходе выполнения данной выпускной квалификационной работы ожидаемые результаты были достигнуты. Был разработан и реализован действующий прототип автоматизированной платформы для развертывания контейнеризованных функций в среде Kubernetes. Созданная платформа способна упростить процессы разработки, тестирования и эксплуатации приложений, предоставляя разработчикам управляемую, безопасную и масштабируемую среду для выполнения их вычислительных задач.

В процессе работы над платформой были реализованы ключевые функциональные возможности, такие как управление жизненным циклом задач, централизованная аутентификация и авторизация пользователей с использованием Keuscloak и гранулярной ролевой модели на основе групп.

Реализованный функционал позволяет достичь поставленных целей по автоматизации развертывания, обеспечению изоляции выполнения пользовательского кода и контролю доступа к ресурсам.

В процессе работы были решены следующие основные задачи:

- 1) проведено исследование предметной области, включая технологии контейнерной оркестрации, а также существующие подходы к реализации платформ FaaS и автоматизации развертывания;
- 2) осуществлен анализ требований к разрабатываемой платформе, определены ключевые сущности, и основные сценарии использования;
- 3) спроектирована архитектура системы, выбран технологический стек, определены компоненты системы и их взаимодействие, разработана спецификация API и схема базы данных;
- 4) разработана серверная часть и клиентская части платформы;
- 5) настроена среда развертывания в Kubernetes, подготовлены необходимые манифесты и конфигурационные файлы;
- 6) проведено функциональное тестирование разработанного прототипа для проверки работоспособности основных функций.

На следующих этапах разработки необходимо расширение функциональности, внедрение модульного и интеграционного тестирования, реализацию более гибких механизмов управления задачами, интеграция с системами непрерывной интеграции и доставки (CI/CD), внедрение механизмов квотирования ресурсов и детального мониторинга.

Исходя из вышеизложенного, можно заключить, что цель данной выпускной квалификационной работы – разработка прототипа автоматизированной платформы развертывания контейнеризованных функций достигнута, а поставленные задачи успешно решены на текущем этапе, создав основу для дальнейшего развития продукта.



## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Roberts, M. What is Serverless? / M. Roberts, J. Chapin. – Sebastopol, CA : O'Reilly Media, Incorporated, 2017. – 60 с. – Текст : непосредственный.
2. Elsten, J. M. Exploring the potential use of FaaS within an iPaaS infrastructure : Master's thesis / Julian M Elsten ; University of Twente. – Enschede, 2023. – 67 с. – Текст : непосредственный.
3. Pu, Q. Shuffling, fast and slow: Scalable analytics on serverless infrastructure / Q. Pu, S. Venkataraman, I. Stoica. – Текст : непосредственный // 16th USENIX symposium on networked systems design and implementation (NSDI 19), Boston, MA, February 26–28, 2019 : proceedings. – Boston : USENIX Association, 2019. – С. 193–206.
4. Lowber, C. Thin-client vs. fat-client tco : research note TCO-13-6585 / C. Lowber ; Gartner. – Stamford, CT : Gartner, Inc., 2001. – 12 с. – Текст : непосредственный.
5. Kokkonen, J. Single-page application frameworks in enterprise software development : Master's Thesis / Juuso Kokkonen ; Aalto University, School of Science, Degree Programme in Computer Science and Engineering. – Espoo, 2015. – 79 с. – Текст : непосредственный.
6. Li, Q. [et al.] Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing / Q. Li, J. Shen, Y. Vigfusson, T. Xu. – Текст : непосредственный // ACM Transactions on Software Engineering and Methodology. – 2023. – Т. 32, № 5. – Art. № 133. – С. 1–29.
7. Silva, C. Prebaking functions to warm the serverless cold start / C. Silva, D. Fireman, T. E. Pereira. – Текст : непосредственный // Proceedings of the 21st International Middleware Conference, Delft, December 7–11, 2020. – New York : ACM, 2020. – С. 1–13.
8. Open Container Initiative. Open container initiative image format specification [Электронный ресурс] : Version 1.1.0 / Open Container Initiative. –

[San Francisco, CA : Open Container Initiative], 2023. – URL: <https://github.com/opencontainers/image-spec/blob/v1.1.0/spec.md> (дата обращения: 30.05.2025). – Текст : электронный.

9. Bretthauer, D. Open source software: A history : Technical Report UCTL-TR-2001-05 / D. Bretthauer ; University of Connecticut Libraries. – Storrs, CT, 2001. – 15 с. – Текст : непосредственный.

10. Fitzgerald, B. The transformation of open source software / B. Fitzgerald. – Текст : непосредственный // MIS Quarterly. – 2006. – Т. 30, № 3. – С. 587–598.

11. Sommerlad, C. Reverse proxy patterns / C. Sommerlad. – Текст : непосредственный // 8th European Conference on Pattern Languages of Programs (EuroPLoP 2003), Irsee, Germany, July 2–6, 2003 : proceedings / eds. C. Schwanninger, L. Hvatum. – Konstanz : Universitätsverlag Konstanz, 2003. – С. 431–458.

12. Jain, N. Overview of virtualization in cloud computing / N. Jain, S. Choudhary. – Текст : непосредственный // 2016 Symposium on Colossal Data Analysis and Networking (CDAN), Indore, India, March 18–19, 2016 : proceedings. – Piscataway, NJ : IEEE, 2016. – С. 1–4.

13. Bentaleb, O. [et al.] Containerization technologies: Taxonomies, applications and challenges / O. Bentaleb, A. S. Z. Belloum, A. Sebaa, A. El-Maouhab. – Текст : непосредственный // The Journal of Supercomputing. – 2022. – Т. 78, № 1. – С. 1144–1181.

14. Casalicchio, E. The state-of-the-art in container technologies: Application, orchestration and security / E. Casalicchio, S. Iannucci. – Текст : непосредственный // Concurrency and Computation: Practice and Experience. – 2020. – Т. 32, № 17. – Art. № e5668.

15. Pan, Y. [et al.] A performance comparison of cloud-based container orchestration tools / Y. Pan, I. Y. Chen, F. Brasileiro, G. Niu. – Текст : непосредственный // 2019 IEEE International Conference on Big Knowledge (ICBK), Beijing, China, November 10–11, 2019 : proceedings. – Piscataway, NJ :

IEEE, 2019. – С. 191–198.

16. Raje, S. N. Performance comparison of message queue methods : Master's thesis / Sanika N Raje ; University of Nevada, Las Vegas, Howard R. Hughes College of Engineering, Department of Computer Science. – Las Vegas, NV, 2019. – 50 с. – Текст : непосредственный.

17. Dürr, K. An evaluation of saga pattern implementation technologies / K. Dürr, R. Lichtenthäler, G. Wirtz. – Текст : непосредственный // 13th ZEUS Workshop (ZEUS 2021), Virtual Event, Germany, February 25–26, 2021 : proceedings / eds. O. Kopp, M. Picht, C. Diefenthal. – Aachen : CEUR-WS.org, 2021. – (CEUR Workshop Proceedings ; Т. 2821). – С. 74–82.

18. Hohpe, G. Enterprise integration patterns: Designing, building, and deploying messaging solutions / G. Hohpe, B. Woolf. – Boston, MA : Addison-Wesley Professional, 2004. – 736 с. – Текст : непосредственный.

19. Menge, F. Enterprise service bus / F. Menge. – Текст : непосредственный // 2nd Free and Open Source Software Conference (FrOSCon 2007), Sankt Augustin, Germany, August 25–26, 2007 : proceedings. – Sankt Augustin : Hochschule Bonn-Rhein-Sieg, 2007. – С. 1–6.

20. Newman, S. Monolith to microservices: evolutionary patterns to transform your monolith / S. Newman. – Sebastopol, CA : O'Reilly Media, Incorporated, 2019. – 296 с. – Текст : непосредственный.

21. Valipour, M. H. [et al.] A brief survey of software architecture concepts and service oriented architecture / M. H. Valipour, B. AmirZafari, K. N. Maleki, N. Daneshpour. – Текст : непосредственный // 2009 2nd IEEE International Conference on Computer Science and Information Technology, Beijing, China, August 8–11, 2009 : proceedings. – Piscataway, NJ : IEEE, 2009. – С. 34–38.

22. Evans, E. Domain-driven design: tackling complexity in the heart of software / E. Evans. – Boston, MA : Addison-Wesley Professional, 2004. – 560 с. – Текст : непосредственный.

23. Ratner, I. M. Vertical slicing: Smaller is better / I. M. Ratner, J.

Harvey. – Текст : непосредственный // 2011 Agile Conference, Salt Lake City, UT, USA, August 8–12, 2011 : proceedings. – Piscataway, NJ : IEEE, 2011. – С. 240–245.

24. Samuel, S. Programming kotlin / S. Samuel, S. Bocutiu. – Birmingham : Packt Publishing Ltd, 2017. – 422 с. – Текст : непосредственный.

25. Ong, S. C. [et al.] The materials application programming interface (api): A simple, flexible and efficient api for materials data based on representational state transfer (rest) principles / S. C. Ong, S. Cholia, A. Jain [et al.]. – Текст : непосредственный // Computational Materials Science. – 2015. – Т. 97. – С. 209–215.

26. Biehl, M. Webhooks–Events for RESTful APIs / M. Biehl. – Berlin : API-University Press, 2017. – 68 с. – (API-University Series ; Т. 4). – Текст : непосредственный.

27. Wilde, E. REST: from research to practice / E. Wilde, C. Pautasso. – New York : Springer Science & Business Media, 2011. – 587 с. – Текст : непосредственный.

28. Box, D. [et al.] Simple object access protocol (soap) 1.1 [Электронный ресурс] : W3C Note 08 May 2000 / D. Box, D. Ehnebuske, G. Kakivaya [et al.] ; World Wide Web Consortium. – [Б. м.] : W3C, 2000. – URL: <https://www.w3.org/TR/2000/NOTE-SOAP-20000508/> (дата обращения: 30.05.2025). – Текст : электронный.

29. Srinivasan, R. Rpc: Remote procedure call protocol specification version 2 [Электронный ресурс] : RFC 1831 / R. Srinivasan ; Network Working Group. – [S.l.] : IETF, August 1995. – URL: <https://www.rfc-editor.org/rfc/rfc1831.html> (дата обращения: 30.05.2025). – Текст : электронный.

30. Ahmed, S. An authentication based scheme for applications using json web token / S. Ahmed, Q. Mahmood. – Текст : непосредственный // 2019 22nd International Multitopic Conference (INMIC), Islamabad, Pakistan, November 29–30, 2019 : proceedings. – Piscataway, NJ : IEEE, 2019. – С. 1–6.

31. Guinard, D. [et al.] From the internet of things to the web of things: Resource-oriented architecture and best practices / D. Guinard, V. Trifa, F. Mattern, E. Wilde. – Текст : непосредственный // Architecting the Internet of things / ed. by D. Uckelmann, M. Harrison, F. Michahelles. – Berlin ; Heidelberg : Springer, 2011. – С. 97–129.
32. Boyd, R. Getting started with OAuth 2.0 / R. Boyd. – Sebastopol, CA : O'Reilly Media, Inc., 2012. – 62 с. – Текст : непосредственный.
33. Richardson, K. Microservices. Patterns of development and refactoring / K. Richardson ; [перевод с английского А. А. Слинкин]. – Санкт-Петербург : Питер, 2023. – 544 с. – Текст : непосредственный.
34. Salvadori, I. A maturity model for semantic restful web apis / I. Salvadori, F. Siqueira. – Текст : непосредственный // 2015 IEEE International Conference on Web Services (ICWS), New York, NY, USA, June 27 – July 2, 2015 : proceedings. – Piscataway, NJ : IEEE, 2015. – С. 703–710.
35. O'Neil, E. J. Object/relational mapping 2008: hibernate and the entity data model (edm) / E. J. O'Neil. – Текст : непосредственный // 2008 ACM SIGMOD International Conference on Management of Data, Vancouver, BC, Canada, June 9–12, 2008 : proceedings. – New York, NY : ACM, 2008. – С. 1351–1356.
36. Yang, D. Java persistence with JPA / D. Yang. – Parker, CO : Outskirts Press, 2010. – 316 с. – Текст : непосредственный.
37. Mernik, M. When and how to develop domain-specific languages / M. Mernik, J. Heering, A. M. Sloane. – Текст : непосредственный // ACM Computing Surveys (CSUR). – 2005. – Т. 37, № 4. – С. 316–344.
38. Poulton, N. The kubernetes book / N. Poulton. – Bolton, UK : NIGEL POULTON LTD, 2023. – 344 с. – Текст : непосредственный.
39. Cooper, J. W. Java design patterns: a tutorial / J. W. Cooper. – Boston, MA : Addison-Wesley, 2000. – 368 с. – Текст : непосредственный.
40. Ferry, E. Security evaluation of the oauth 2.0 framework / E. Ferry, J. O Raw, K. Curran. – Текст : непосредственный // Information & Computer

Security. – 2015. – Т. 23, № 1. – С. 73–101.

41. Arai, K. Method resource sharing in on-premises environment based on cross-origin resource sharing and its application for safety-first constructions / K. Arai, K. Norikoshi, M. Oda. – Текст : непосредственный // International Journal of Advanced Computer Science & Applications. – 2024. – Т. 15, № 5. – С. 504–512.

42. Barth, A. Robust defenses for cross-site request forgery / A. Barth, C. Jackson, J. C. Mitchell. – Текст : непосредственный // Proceedings of the 15th ACM conference on Computer and communications security, Alexandria, VA, USA, October 27–31, 2008. – New York : ACM, 2008. – С. 75–88.

43. Kiczales, G. [et al.] Aspect-oriented programming / G. Kiczales, J. Lamping, A. Mendhekar [et al.]. – Текст : непосредственный // ECOOP'97–Object-Oriented Programming : 11th European Conference, Jyväskylä, Finland, June 9–13, 1997 : Proceedings / eds. M. Aksit, S. Matsuoka. – Berlin ; Heidelberg : Springer, 1997. – (Lecture Notes in Computer Science ; Т. 1241). – С. 220–242.

44. Tikhonova, A. Design and Implementation of Reusable Component for Vue.js : Bachelor's Thesis / Anna Tikhonova ; Metropolia University of Applied Sciences, Degree Programme in Information Technology. – Helsinki, 2021. – 44 с. – Текст : непосредственный.

45. Boistrond, C. To terminate or not to terminate secure sockets layer (ssl) traffic at the load balancer [Электронный ресурс] : arXiv:2011.09621v1 [cs.CR] / Corentin Boistrond. – 2020. – 19 Nov. – URL: <https://arxiv.org/abs/2011.09621> (дата обращения: 30.05.2025). – Текст : электронный.

46. Poniszewska-Marańda, A. Kubernetes cluster for automating software production environment / A. Poniszewska-Marańda, E. Czechowska. – Текст : непосредственный // Sensors. – 2021. – Т. 21, № 5. – Art. № 1910. – С. 1–23.

47. Ibryam, B. Kubernetes patterns / B. Ibryam, R. Huß. – Sebastopol, CA : O'Reilly Media, Inc., 2022. – 320 с. – Текст : непосредственный.

48. Jiang, Z. M. A survey on load testing of large-scale software systems /

Z. M. Jiang, A. E. Hassan. – Текст : непосредственный // IEEE Transactions on Software Engineering. – 2015. – Т. 41, № 11. – С. 1091–1118.

49. McIntosh, S. The evolution of Java build systems / S. McIntosh, B. Adams, A. E. Hassan. – Текст : непосредственный // Empirical Software Engineering. – 2012. – Т. 17, № 4–5. – С. 578–608.

50. Buchanan, S. Container registries / S. Buchanan, B. Coleman, K. Ots. – Текст : непосредственный // Introducing Azure Kubernetes Service: A Practical Guide to Container Orchestration / S. Buchanan, B. Coleman, K. Ots. – Berkeley, CA : Apress, 2020. – С. 17–34.

51. Scott, E. A., Jr. SPA Design and Architecture: Understanding single-page web applications / E. A. Scott Jr. – New York : Simon and Schuster, 2015. – 300 с. – Текст : непосредственный.