# **4** **РЕАЛИЗАЦИЯ**

## **4.1 Выбор инструментальных средств, системного и дополнительного программного обеспечения**

Разработка программного продукта осуществляется при помощи следующих инструментов:

1. NodeJS;
2. Фреймворк Prisma для создания структуры базы данных;
3. Фреймворк NextJS;
4. TypeScript;
5. PyCharm – IDE, в котором велась разработка web-приложения;
6. Git – для контроля версий написанного кода;
7. База данных MS SQL.

Рассмотрим подробно особенности используемых технологий:

1. Node.js: Это среда выполнения JavaScript, основанная на движке Chrome V8. Она позволяет разрабатывать серверные приложения на JavaScript, а также обеспечивает доступ к множеству пакетов и модулей через свой менеджер пакетов npm.
2. Prisma: Это инструмент для работы с базами данных, который позволяет создавать и мигрировать структуру базы данных с помощью декларативного языка моделирования. Prisma поддерживает различные базы данных, такие как PostgreSQL, MySQL и SQLite.
3. Next.js: Это фреймворк для разработки React приложений. Он предоставляет множество функций, таких как серверный рендеринг, статическая генерация, предварительная загрузка и динамическая

маршрутизация. Next.js позволяет создавать быстрые и масштабируемые приложения.

1. TypeScript: Это язык программирования, который является надмножеством JavaScript. Он добавляет статическую типизацию, что помогает выявлять ошибки на этапе разработки и облегчает сопровождение кода. TypeScript компилируется в JavaScript, поэтому его можно использовать вместе с Node.js и React.js.
2. PyCharm: Это интегрированная среда разработки (IDE) для языка программирования Python. Она предоставляет удобные инструменты для разработки и отладки Python-приложений, а также интеграцию с системами управления версиями, такими как Git.
3. Git: Это распределенная система контроля версий, которая позволяет отслеживать изменения в коде и сотрудничать с другими разработчиками. Git обеспечивает сохранность истории изменений, позволяет создавать ветки для параллельной разработки и упрощает слияние изменений.
4. MS SQL (Microsoft SQL Server) - это реляционная база данных, разработанная компанией Microsoft. Она предоставляет надежное и масштабируемое хранилище данных для различных приложений и систем. MS SQL поддерживает язык запросов SQL для работы с данными и предоставляет широкий набор функций и возможностей для управления и администрирования баз данных.

## **4.2 Описание реализации вариантов использования**

Одним из способов реализации описание варианта использования программного продукта является построение диаграмм деятельности. С одной стороны, диаграмма деятельности ‒ это полноценная диаграмма UML, с другой стороны, диаграмма деятельности немногим отличается от блок-схемы (а тем самым и от псевдокода). Таким образом, реализация варианта использования диаграммой деятельности является компромиссным способом ведения разработки ‒ в сущности, это проектирование сверху вниз в терминах и обозначениях UML.

Диаграммы деятельности состоят из ограниченного количества фигур, соединённых стрелками.

Прямоугольники с закруглениями – действия (операция). Узел управления – это абстрактный узел действия, которое координирует потоки действий.

Ромбы – решения. Узел решения предназначен для определения правила ветвления и различных вариантов дальнейшего развития сценария. В точку ветвления входит ровно один переход, а выходит – два или более.

Широкие полосы – начало (разветвление) и окончание (схождение) ветвления действий. Узел объединения имеет два и более входящих узла и один исходящий.

Чёрный круг – начало процесса (начальный узел). Начальный узел деятельности (или начальное состояние деятельности) является узлом управления, в котором начинается поток (или потоки) при вызове данной деятельности извне.

Чёрный круг с обводкой – окончание процесса (финальный узел). Конечный узел деятельности (или конечное состояние деятельности) является узлом управления, который останавливает все потоки данной диаграммы деятельности. На диаграмме может быть более одного конечного узла.

Стрелки идут от начала к концу процесса и показывают потоки управления или потоки объектов (данных).

Построим диаграммы деятельности по нескольким алгоритмам системы.

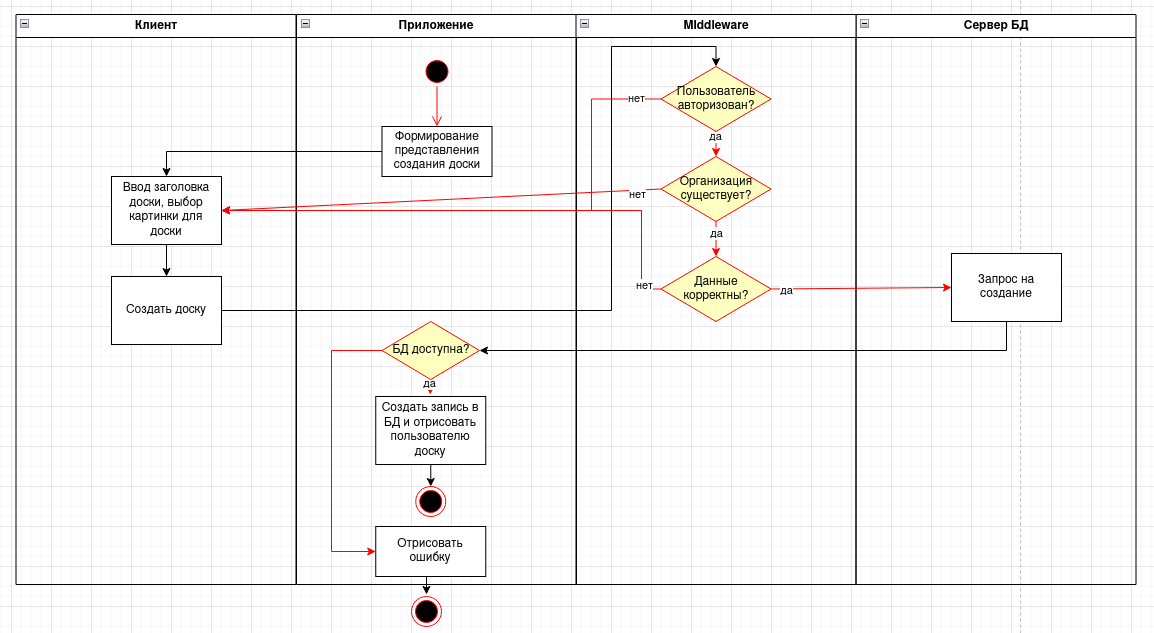
Алгоритмы авторизации, регистрации, управления организации не рассматриваются, так как данными алгоритмами управляет сторонний сервис - «Clerk».

При работе с приложением пользователь может создать доску(листинг 4.1). После создания доски, пользователь может создать лист для задач(листинг 4.2). Так же пользователь может создать задачу, внутри листа (листинг 4.3).

Для удаления задачи, пользователю необходимо открыть задачу, после чего нажать кнопку «Удалить» (листинг 4.4).

Диаграммы алгоритма создания доски, создания листа, создания задачи представлены на рисунках 4.1 - 4.3.

Диаграмма алгоритма удаления задачи представлена на рисунке 4.4.



**Рисунок 4.1 – Диаграмма активности выполнения кейса создания доски**

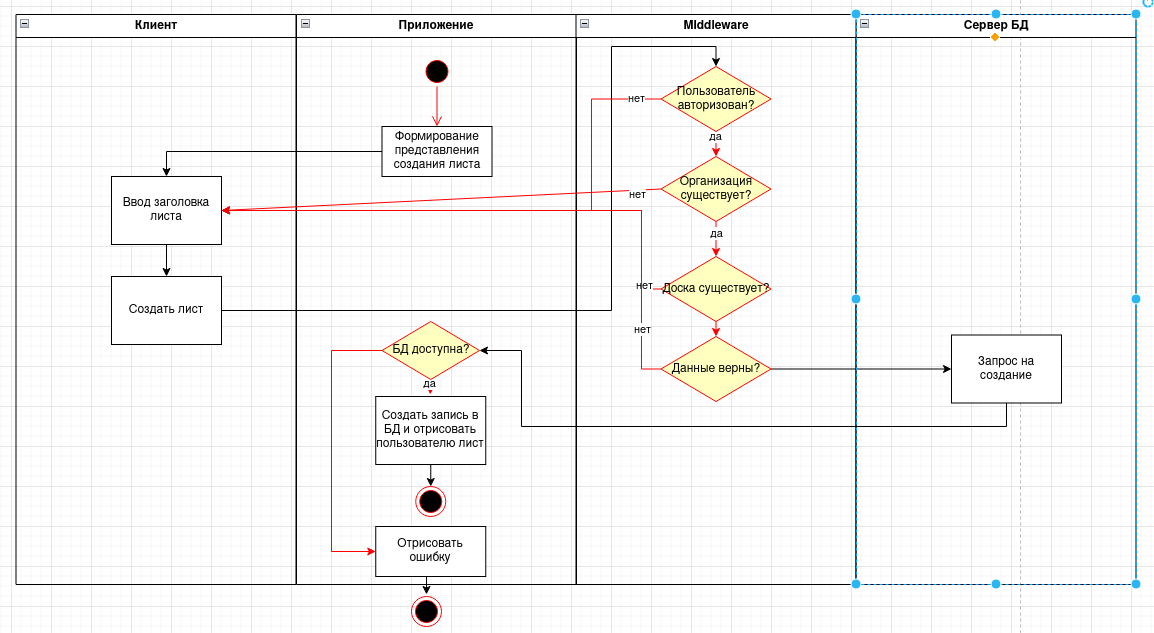
Листинг 4.1 – Выполнение кейса создания доски

const handler = async (data: InputType): Promise<ReturnType> => {  
 const { userId, orgId } = auth();  
  
 if(!userId || !orgId){  
 return {  
 error: "Пользователь неавторизован.",  
 }  
 }  
 const { title, image } = data;

const [  
 imageId,  
 imageThumbUrl,  
 imageLinkHTML,  
 imageFullUrl,

imageUserName  
 ] = image.split("|")

if (!imageId || !imageThumbUrl || !imageFullUrl || !imageUserName || !imageLinkHTML) {  
 return {  
 error: "Переданы не все поля! Невозможно создать доску."  
 };  
 }  
  
 let board;  
 try{  
 board = await db.board.create({  
 data: {  
 title,  
 orgId,  
 imageId,  
 imageThumbUrl,  
 imageFullUrl,  
 imageUserName,  
 imageLinkHTML  
 }  
 })  
 } catch (e) {  
 return {  
 error: "Невозможно создать доску."  
 }  
 }  
 await createAuditLog({  
 entityTitle: board.title,  
 entityId: board.id,  
 entityType: ENTITY\_TYPE.BOARD,  
 action: ACTION.CREATE,  
 })  
 revalidatePath(`/board/${board.id}`)  
 return {data: board}  
}

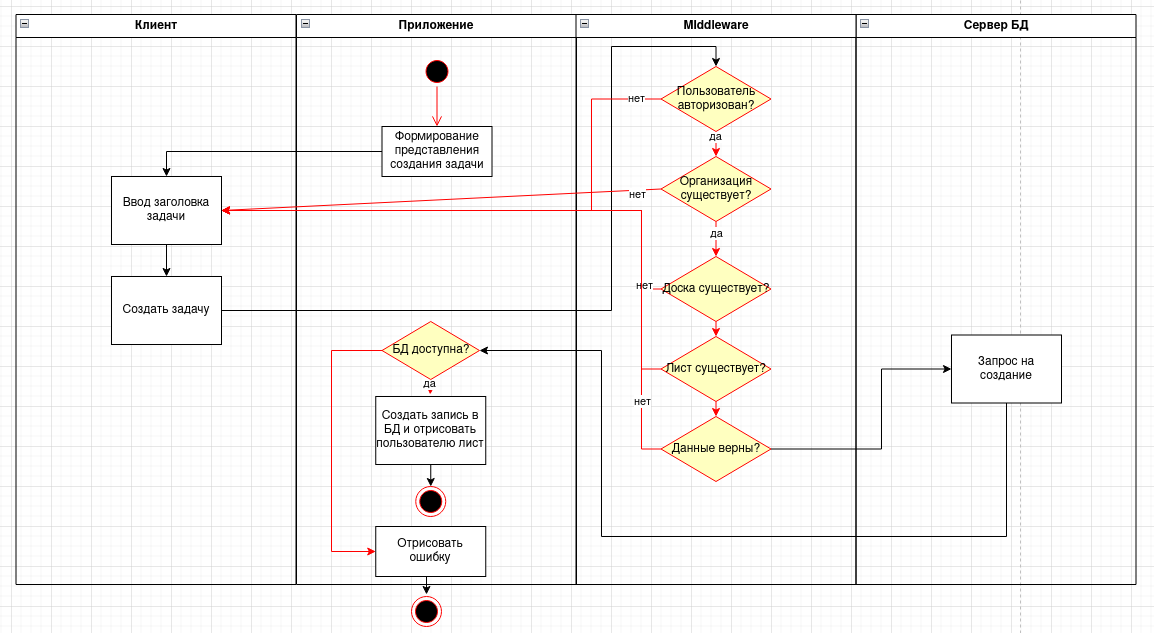


**Рисунок 4.2 – Диаграмма активности выполнения кейса создания листа**

Листинг 4.2 – Выполнение кейса создания листа

const handler = async (data: InputType): Promise<ReturnType> => {  
 const { userId, orgId } = auth();  
 if (!userId || !orgId) {  
 return {  
 error: "Пользователь не авторизован.",  
 };  
 }  
 const { title, boardId } = data;  
 let list;  
 try {  
 const board = await db.board.findUnique({  
 where: {  
 id: boardId,  
 orgId,  
 }  
 })  
 if (!board) {  
 return{  
 error: "Доска не найдена.",  
 }  
 }  
  
 const lastList = await db.list.findFirst({  
 where: { boardId: boardId},  
 orderBy: { order: "desc"},  
 select: {order: true},  
 })

const newOrder = lastList ? lastList.order + 1 : 1;  
 list = await db.list.create({  
 data: {  
 title,  
 boardId,  
 order: newOrder  
 },  
 });  
 await createAuditLog({  
 entityTitle: list.title,  
 entityId: list.id,  
 entityType: ENTITY\_TYPE.LIST,  
 action: ACTION.CREATE,  
 })  
 } catch (error) {  
 return {  
 error: "Невозможно создать лист задач."  
 }  
 }  
 revalidatePath(`/board/${boardId}`);  
 return { data: list };  
};



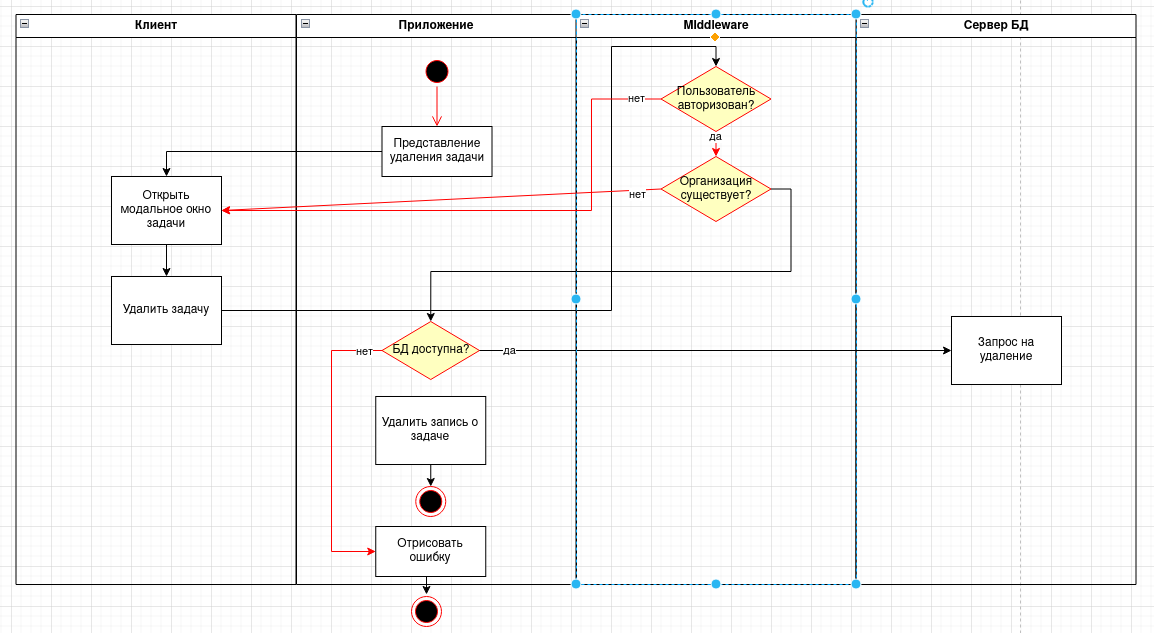
**Рисунок 4.3 – Диаграмма активности выполнения кейса создания задачи**

Листинг 4.3 – Выполнение кейса создания задачи

const handler = async (data: InputType): Promise<ReturnType> => {  
 const { userId, orgId } = auth();  
  
 if (!userId || !orgId) {  
 return {  
 error: "Пользователь не авторизован.",  
 };  
 }  
 const { title, boardId, listId } = data;  
 let card;  
  
 try {  
 const list = await db.list.findUnique({  
 where: {  
 id: listId,  
 board: {  
 orgId  
 }  
 }  
 })

if (!list) {  
 return{  
 error: "Список не найдена.",  
 }  
 }

const lastCard = await db.card.findFirst({  
 where:{listId},  
 orderBy: {order: "desc"},  
 select: {order: true}  
 })  
 const newOrder = lastCard ? lastCard.order + 1 : 1;  
 card = await db.card.create({  
 data:{  
 title,  
 listId,  
 order: newOrder  
 }  
 })  
 await createAuditLog({  
 entityId: card.id,  
 entityTitle: card.title,  
 entityType: ENTITY\_TYPE.CARD,  
 action: ACTION.CREATE  
 })  
  
 } catch (error) {  
 return {  
 error: "Невозможно создать задачу."  
 }  
 }  
 revalidatePath(`/board/${boardId}`);  
 return { data: card };};



**Рисунок 4.4 - Диаграмма активности выполнения кейса удаления задачи**

## Листинг 4.4 – Удаление задач

const handler = async (data: InputType): Promise<ReturnType> => {  
 const { userId, orgId } = auth();

if (!userId || !orgId) {  
 return {  
 error: "Пользователь не авторизован.",  
 };  
 }  
  
 const { id } = data;  
 let board;  
  
 try {  
 board = await db.board.delete({  
 where: {  
 id,  
 orgId,  
 },  
 });  
  
 await createAuditLog({  
 entityTitle: board.title,  
 entityId: board.id,  
 entityType: ENTITY\_TYPE.BOARD,  
 action: ACTION.DELETE,  
 })  
 } catch (error) {  
 return {  
 error: "Невозможно удалить данные."  
 }  
 }  
  
 revalidatePath(`/organization/${orgId}`);  
 redirect('/organization/$(orgId)')  
};

## **4.3 Функциональное тестирование**

Тестирование программного обеспечения – это метод проверки соответствия фактического программного продукта ожидаемым требованиям, который также необходим, чтобы убедиться, что продукт не содержит дефектов. Подразумевает выполнение предварительно определенных алгоритмов с использованием ручных или автоматизированных инструментов для оценки одного или нескольких интересующих свойств.

Целью тестирования ПО является выявление ошибок, пробелов или отсутствующих требований, заданных на этапе проектирования продукта.

Тестирование подразделяется на три категории:

1. Функциональное тестирование: модульное (компонентное), интеграционное, системное, регрессионное, приемочное.
2. Тестирование производительности: тестирование отказоустойчивости, нагрузочное, объемное, тестирование масштабируемости.
3. Обслуживание (регресс и обслуживание): регрессионное, тестирование технического обслуживания.

Некоторые предпочитают называть тестирование программного обеспечения:

* «белый ящик» – когда у нас есть доступ к коду, и мы его тестируем, читаем сам код (статическое тестирование);
* «черный ящик» – когда мы не знаем, как система устроена внутри, нет доступа к коду или мы не умеем его читать, и поэтому ориентируемся только на внешнее поведение или техническое задание;
* «серый ящик» – когда мы смотрим в код и понимаем, как он устроен, а потом открываем само приложение и проверяем, как этот код отображается уже в нем, но ориентируемся уже на техническое задание.

В качестве метода тестирования был использован метод «черного ящика», при котором не используется знание о внутреннем устройстве тестируемого объекта. Тестирование будет осуществляется только через пользовательский интерфейс в котором будет происходить сопоставление ожидаемых результатов с получаемыми.

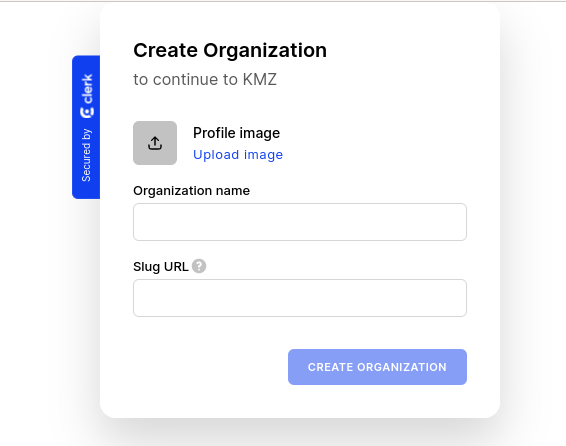
Для тестирования автоматизированной системы малых торговых предприятий были разработаны тест – кейсы. Тест – кейсы были составлены таким образом, чтобы проверить все основные функции системы.

Тест – кейс – это описанная последовательность определённых действия (шагов) и ожидаемого результата для проверки работы определённого функционала системы.

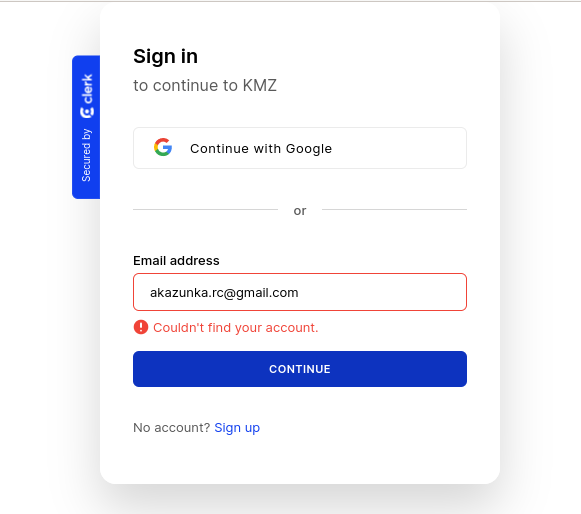
Для проверки правильности и точности работы основных функций приложения был разработан набор тест – кейсов, представленный в приложении А.

Тестирование было проведено на компьютере с установленной операционной системой Fedora Linux, средой разработки PyCharm 2023, используемый браузер Firefox. Результаты некоторых тест – кейсов показаны на рисунках 4.5 – 4.13.

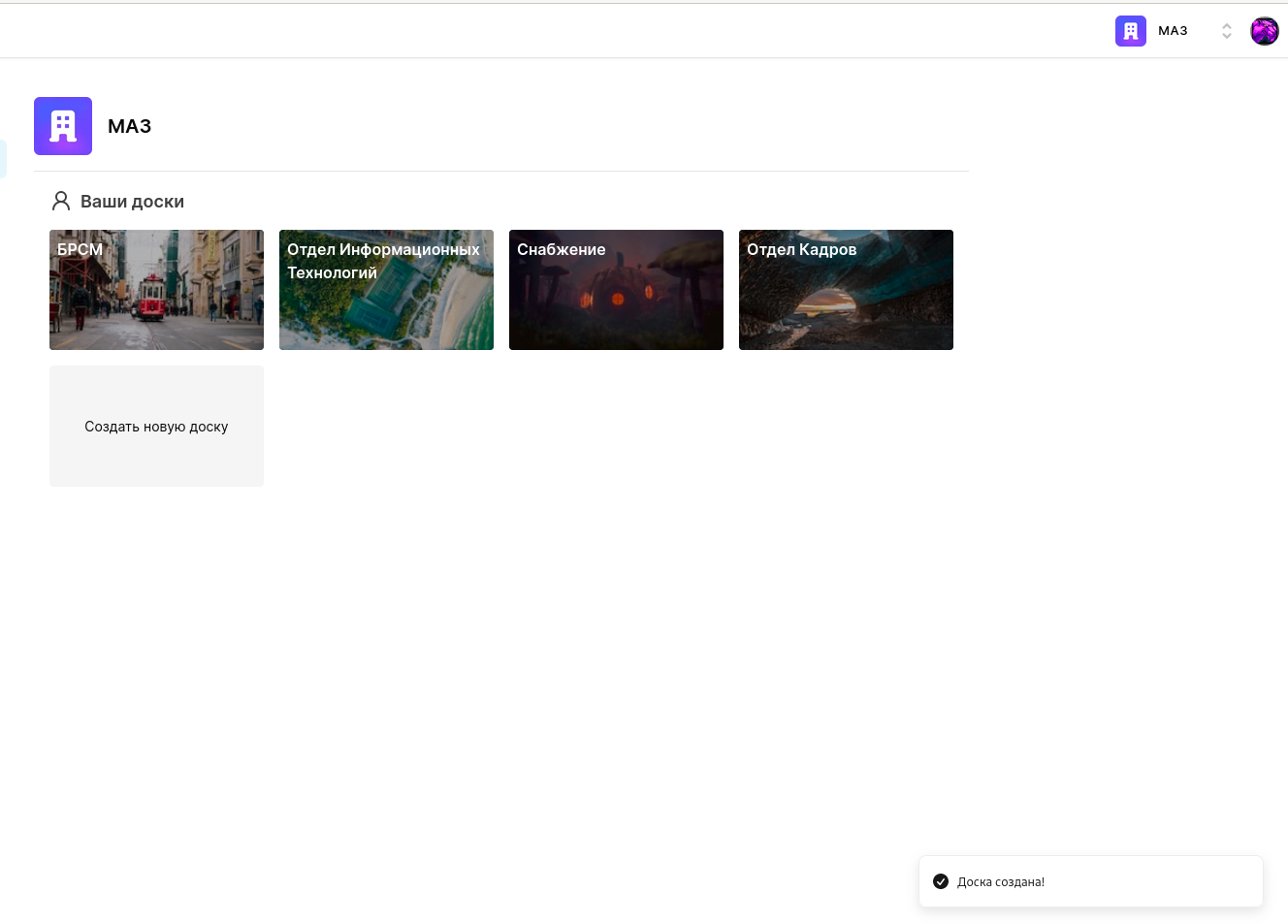
После успешной регистрации, пользователю будет предложено создать организацию.



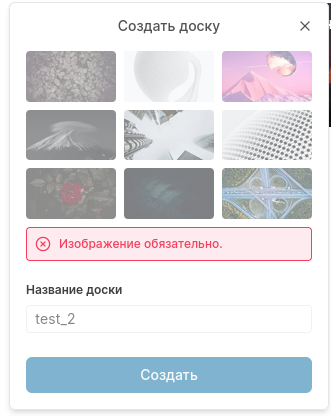
**Рисунок 4.5 – Успешная авторизация пользователя**



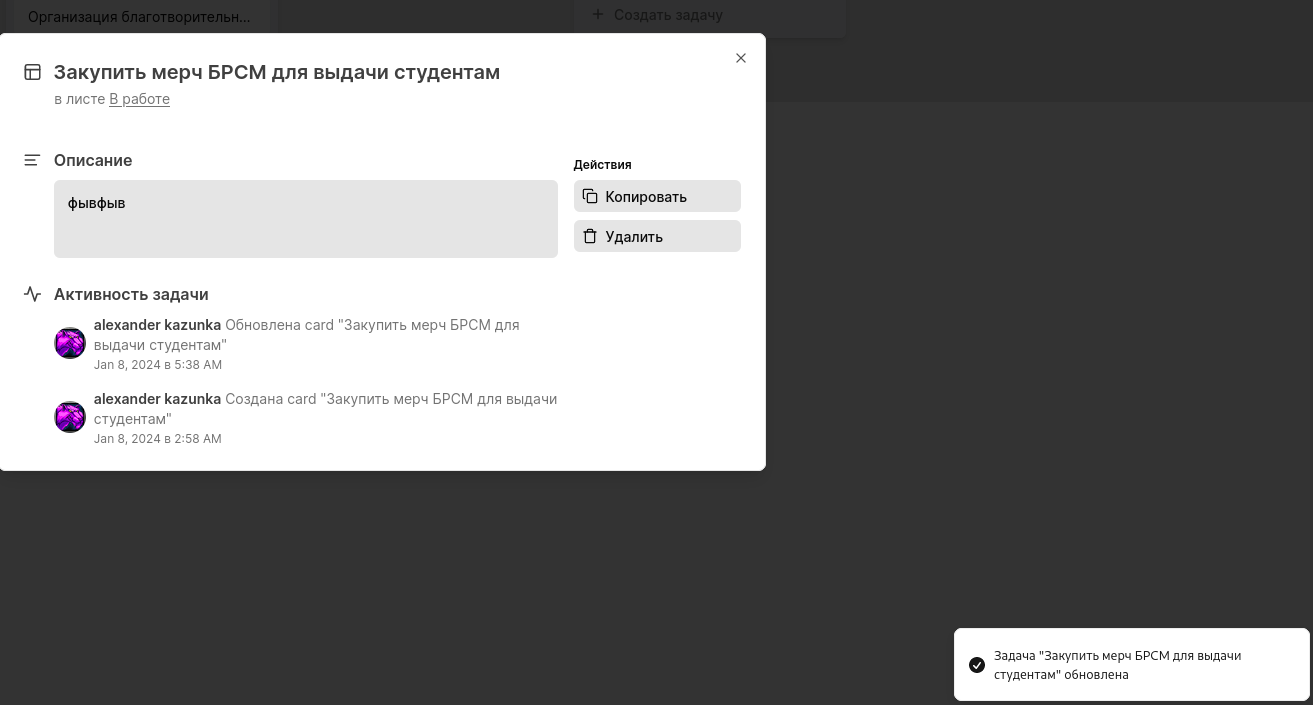
**Рисунок 4.6 – Ввод некорректных данных пользователя**



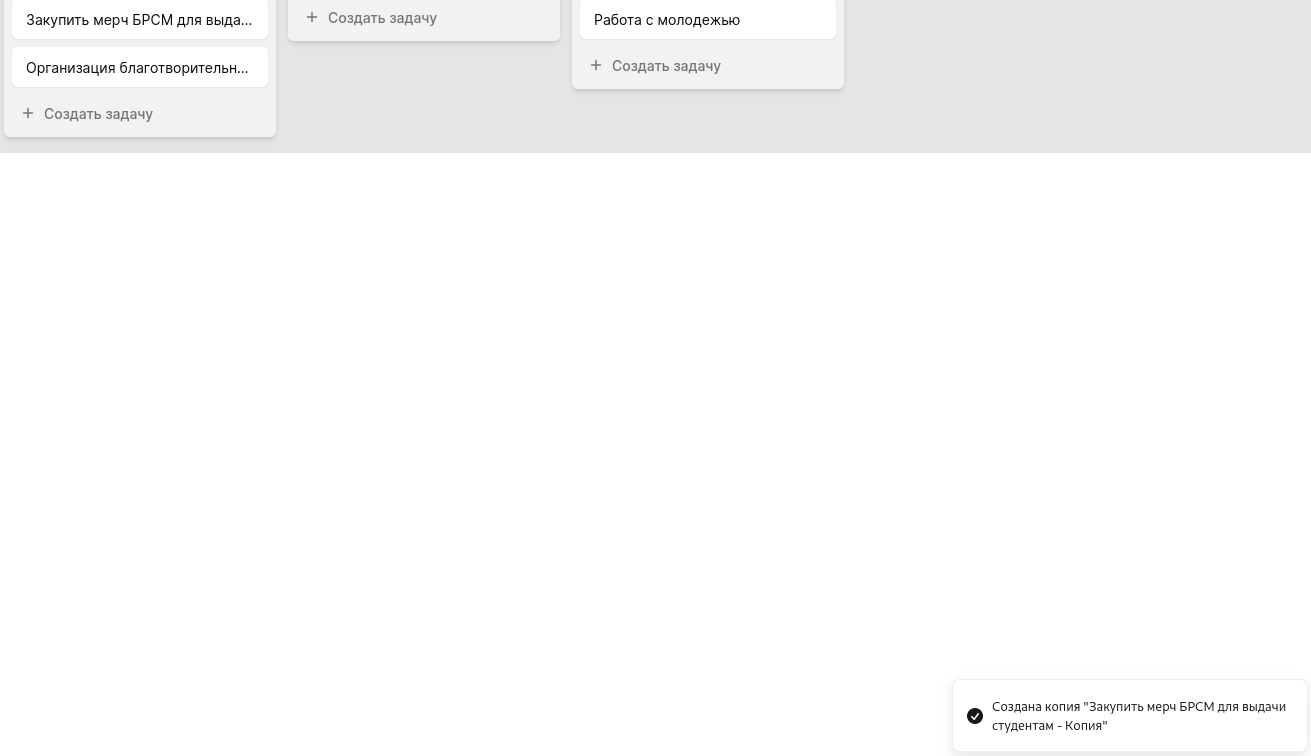
**Рисунок .4.7 – Создание доски**



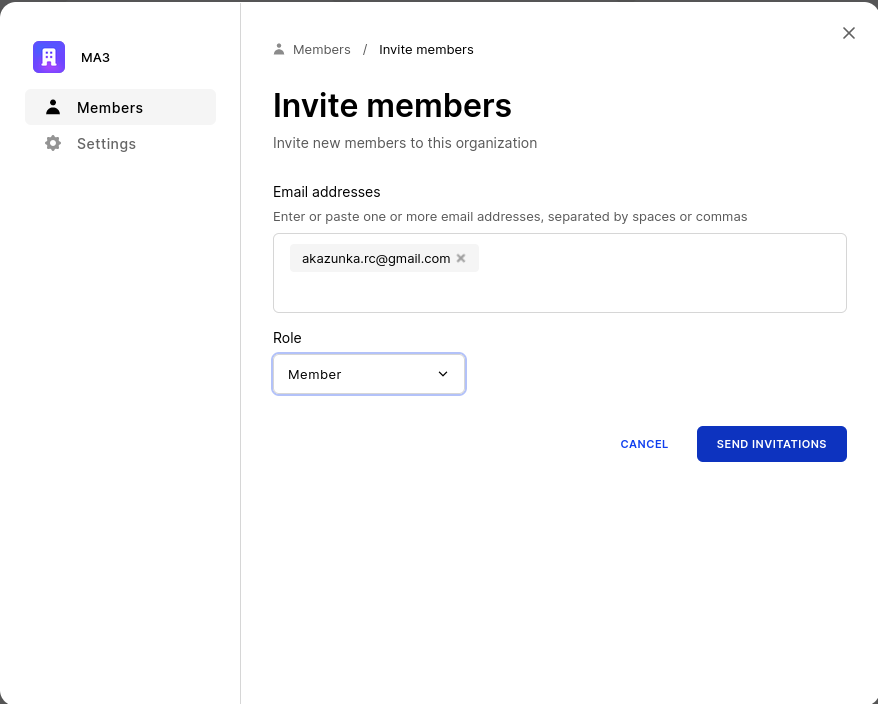
**Рисунок 4.8 – Ошибка заполнения обязательных полей**



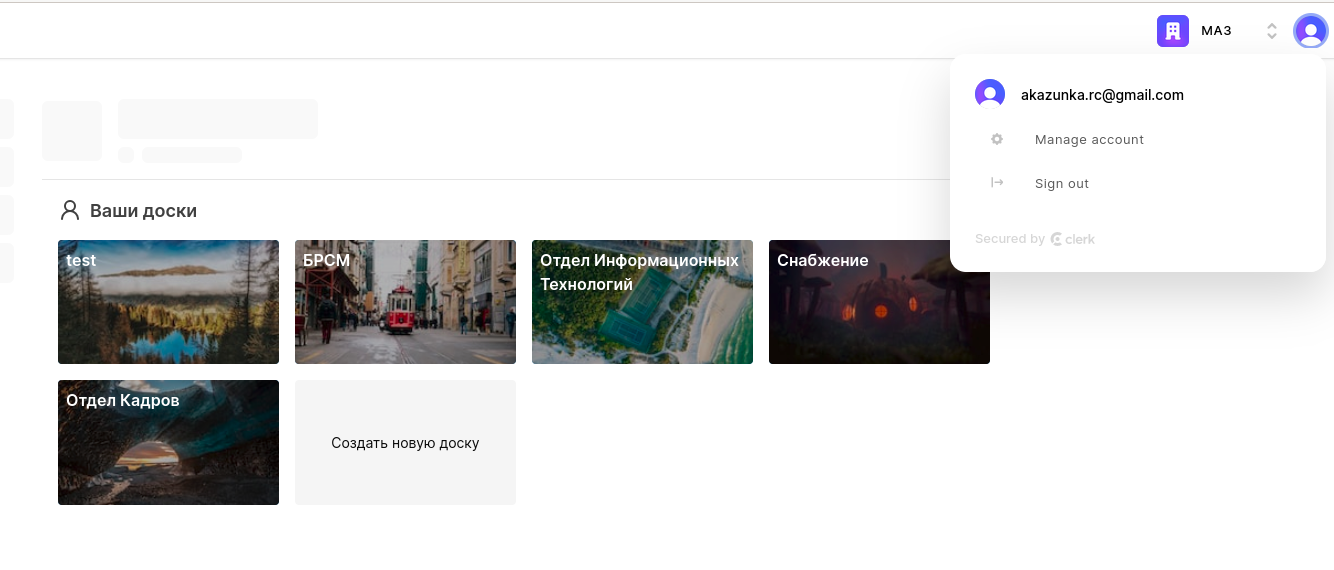
**Рисунок 4.9 – Обновление задачи**



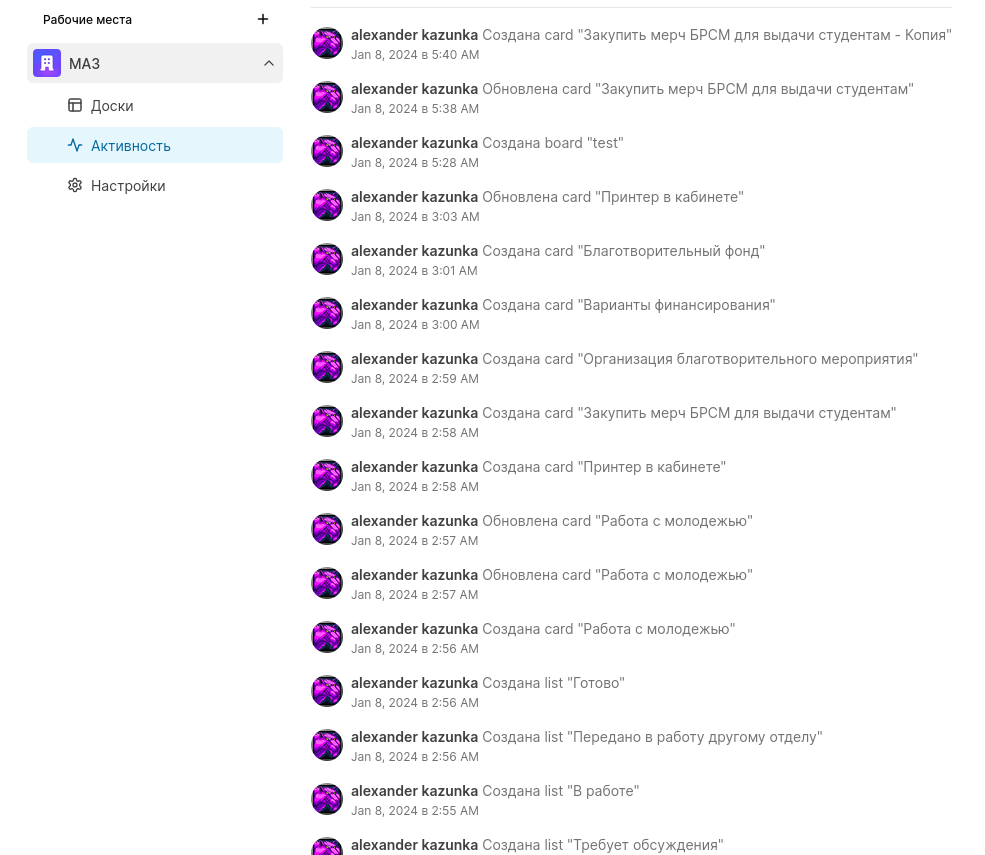
**Рисунок 4.10 – Копирование задачи**



**Рисунок 4.11 – Отправка приглашения пользователю**



**Рисунок 4.12 – Проверка доступа пользователя**



**Рисунок 4.13 – Активность по доске**

## **4.4 Прочие виды тестирования**

Тестирование производительности в инженерии программного обеспечения – тестирование, которое проводится с целью определения, как быстро работает вычислительная система или её часть под определённой нагрузкой. Также может служить для проверки и подтверждения других атрибутов качества системы, таких как масштабируемость, надёжность и потребление ресурсов .

В тестировании производительности различают следующие направления:

1. Стресс – тестирование (Stress Test). Этот тест проводится первым. Нагрузка постепенно увеличивается до тех пор, пока приложение не перестанет работать корректно. В конце теста фиксируется количество пользователей, которое приложение выдерживало, соответствуя требованиям производительности, и сколько выдержать не смогло. Первое значение и будет пределом производительности вашего приложения. Часто этот вид

тестирования проводится, если заказчик предвидит [резкое увеличение нагрузки на систему](https://www.a1qa.ru/blog/sajt-gotov-k-chernoj-pyatnitse-kiberponedelniku/).

1. Нагрузочный тест (Load Test). Нагрузка на систему подается на протяжении 4-8 часов. В это время собираются метрики производительности: количество запросов в секунду, транзакций в секунду, время отклика от сервера, процент ошибок в ответах, утилизация аппаратных ресурсов и т д. Собранные метрики проходят проверку на соответствие заданным требованиям. Также на выходе имеем локализацию узких мест в производительности приложения и дефектов, подробное профилирование всех компонентов системы и утилизацию аппаратных ресурсов под целевой нагрузкой.
2. Тестирование на больших объемах данных (Volume Test). Данный вид тестирования помогает сделать прогноз относительно работоспособности приложения. Форма подаваемой нагрузки та же, что и при нагрузочном тестировании. Задача теста – узнать, какое влияние окажет увеличение объема данных на систему. Чем меньше продолжительность теста, тем больший уровень нагрузки требуется. Чем больше продолжительность теста, тем меньшая нагрузка должна быть установлена.
3. Тестирование отказоустойчивости (Stability Test). Продолжительность нагрузки может варьироваться в зависимости от целей и возможностей проекта, доходя до семи дней и более. В результате получаем представление о том, как изменится производительность системы в течение длительного периода времени под нагрузкой, например, в течение недели.

Тестирование программного обеспечения было проведено посредством инструмента «Системный монитор» операционной системы Fedora Linux. Показатели нагружености можно увидеть на рисунке 4.14.



**Рисунок 4.14 –Диспетчер задач**