

**Министерство науки и высшего образования Российской Федерации**

Федеральное государственное бюджетное образовательное учреждение   
высшего профессионального образования

**«ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ УПРАВЛЕНИЯ»**

Институт \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Выпускная квалификационная (бакалаврская) работа**

обучающегося \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Фамилия, Имя, Отчество)

на тему \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Направление подготовки \_\_\_\_\_\_\_ – \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

(шифр) (название)

***Направленность (профиль) подготовки*** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(название)

**ОБУЧАЮЩИЙСЯ** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Фамилия, Инициалы)

**РУКОВОДИТЕЛЬ** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(ученая степень, звание, Фамилия и Инициалы)

**КОНСУЛЬТАНТ** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(ученая степень, звание, Фамилия и Инициалы)

Допустить к защите

**ЗАВЕДУЮЩИЙ КАФЕДРОЙ** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(ученая степень, звание, Фамилия и Инициалы)

" \_\_\_ " \_\_\_\_\_\_\_\_\_\_\_\_ 201\_\_г.

**Москва - 2022**

Оглавление

[СПИСОК СОКРАЩЕНИЙ 3](#_Toc10352184)

[ВВЕДЕНИЕ 4](#_Toc10352185)

[Глава 1. АНАЛИТИЧЕСКАЯ ГЛАВА 6](#_Toc10352186)

[1.1. Описание и анализ предметной области 6](#_Toc10352187)

[1.2. Спецификация GEDCOM 14](#_Toc10352188)

[1.3. Существующие примеры генеалогического ПО 17](#_Toc10352189)

[2. ИНСТРУМЕНТАРИЙ ДЛЯ РАЗРАБОТКИ ПО 22](#_Toc10352190)

[2.1. Подробное описание спецификации GEDCOM 22](#_Toc10352191)

[2.2. Язык программирования 35](#_Toc10352192)

[2.3. Среда разработки 38](#_Toc10352193)

[3. РАЗРАБОТКА ПО 40](#_Toc10352194)

[3.1. Подготовка к разработке ПО 40](#_Toc10352195)

[3.2. Описание первой рабочей сборки генеалогического ПО 43](#_Toc10352196)

[3.3. Описание первого релиза генеалогического ПО 47](#_Toc10352197)

[3.4. Дальнейшая разработка генеалогического ПО 50](#_Toc10352198)

[ЗАКЛЮЧЕНИЕ 53](#_Toc10352199)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ 54](#_Toc10352200)

[Приложение 1 56](#_Toc10352201)

[Приложение 2 57](#_Toc10352202)

[Приложение 3 59](#_Toc10352203)

[Приложение 4 61](#_Toc10352204)

[Приложение 5 66](#_Toc10352205)

# СПИСОК СОКРАЩЕНИЙ

Таблица 1. Список сокращений

|  |  |
| --- | --- |
| БД | База данных |
| ОС | Операционная система |
| ПО | Программное обеспечение |
| GUI | Графический пользовательский интерфейс |
| Ged-файл | Текстовый файл, содержащий генеалогическую информацию, структура которого соответствует спецификации GEDCOM |
|  |  |
|  |  |

Зеленым фоном выделяю предлагаемую мной редакцию. Исходный вариант — ~~сохраняю, но зачеркиваю~~.

Желтым фоном выделяю свои комментарии или мысли по поводу.

Красным шрифтом выделяю то, что надо переписать.

Мелкую редакторскую правку могу сделать, не отмечая в тексте, поэтому для дальнейшей работы лучше использовать данный файл.

А.Е.

# ВВЕДЕНИЕ

Современному бизнесу нужны инструменты для быстрой обработки, а главное - надежного хранения информации о взаимоотношении с их клиентами. В качестве таковых могут выступать многочисленные технологии - от таблиц Excel до таких платформ как 1С Предприятие или SAP CRM здесь и далее: обязательно в начале работы надо сделать таблицу всех аббревиатур с достаточно подробной расшифровкой (модуль платформы SAP S/4HANA). Но эти решения не универсальны и едва ли идеально будут отвечать потребностям организаций с определенными надо конкретнее масштабами, бизнес-сферами надо конкретнее, IT решениями надо конкретнее или политиками безопасности надо конкретнее. Для подобных компаний будет целесообразна разработка собственных CRM систем, спроектированных и разработанных для конкретных условий эксплуатации и/или бизнес сфер.

Примерами организаций, которые могут принять подобное решение, являются коммерческие банки. Такие банки представляют собой сложные многопрофильные организации, которые могут предоставлять разнообразные финансовые услуги, в частности:

* Различные виды кредитования, в том числе, потребительское кредитование, автокредитование и ипотечное кредитование;
* Валютные услуги;
* Инвестиционные услуги;
* Депозитные услуги;
* Трастовые услуги;
* Информационные услуги;
* Консалтинговые услуги.

По оформлению списков (здесь и далее):

- Если все пункты — не более одного предложения, их надо завершать ; , а последний — точкой. Если какие-то из пунктов более одного предножения, все надо завершать точкой.

- Вложенных списков лучше по возможности избегать.

- Лучше без И т.п.

Большинство коммерческих банков на данный момент относятся к финансово-технологической отрасли и имеют сложную и масштабную IT инфраструктуру, а также применяют серьезные меры информационной безопасности. Для принятия многих решений по отношению к клиентам используются СППР, а множество инструментов автоматизации процессов являются внутренней разработкой.

Из сказанного выше следует, что для подобных организаций не пойдут CRM для малого или среднего бизнеса. Чтобы полноценно применять CRM системы в подобных организациях, нужно интегрировать эти системы в IT инфраструктуру и дорабатывать множество других систем, как, например, уже вышеупомянутые СППР.

Проблема заключается в том, что банк – очень инертная в отношении внутренних перестроек и аккуратная организация, которая не стремится к резким изменениям. Любые изменения бизнес-процессов и интеграции в них проприетарных систем влекут за собой огромные финансовые риски и риски репутации банка. Также любые внутренние перестройки в банках – это значительные затраты. Поэтому разработка собственной CRM-системы, которая будет спроектирована под определенные оказываемые услуги, особенности инфраструктуры и бизнес-процессы, а также будет следовать всем внутренним технологическим нормам и нормам информационной безопасности, представляет собой актуальную проблему.

Цель данной работы – создание каркаса каркаса чего? и программного конструктора, которые упростят и ускорят разработку CRM-систем. Достижение данной цели предполагает решение следующих задач: исследование типовых CRM-систем, анализ технологий и архитектурных стилей и шаблонов, а также разработка прототипа CRM-системы.

На данный момент в России особо остро стоит проблема поиска отечественных IT решений, и CRM одно из них. В текущей ситуации, когда большинство IT гигантов уходит с российского рынка, данная работа позволит выявить основные этапы в разработке модульных CRM-систем, а также технологии, которые могут значительно упростить данный процесс.

В первой главе ВКР проводится анализ типовых CRM систем, выявлены основные компоненты ПО и требования к его разработке, приведены примеры существующих решений, а также произведен анализ потребностей пользователей и функциональных возможностей ПО, удовлетворяющих эти потребности.

Во второй главе произведен обзор наиболее популярных IT технологий для разработки CRM систем, их положительные и отрицательные стороны. Произведен анализ технологий и выбран технологический стэк для разработки CRM решения, описана структура, стандарты и архитектура готовой системы вместе с обоснованием принятия этих решений. Также представлены возможные инфраструктурные решения для готовой системы.

В третьей главе представлено детальное описание процесса разработки CRM решения с использованием выбранных инструментальных средств, методик, стандартов, выполненной в соответствии с результатами, полученными при реализации предыдущих этапов работы.

# Глава 1. АНАЛИТИЧЕСКАЯ ГЛАВА

## Описание и анализ предметной области

В соответствии с классическим определением, CRM система – это специальное программное обеспечение, которое предназначено для автоматизации бизнесс-процессов и оперативной работы в компании, сокращения длительности выполнения рутинных задач, персонального и коллективного планирования с целью достижения высокого уровня обслуживания клиентов [1]. Но на самом деле “что такое CRM система” – это спорный вопрос, который может иметь много ответов. Одни компании считают, что это ПО для автоматизации бизнес-процессов, вторые видят в ней систему телефонии, третьи – планировщик. Поэтому смысл CRM системы трактуется в каждой компании в свете ее бизнес-сфер и конкретных задачах.

При всей неоднозначности требований, предъявляемых к CRM-системам, они имеют ряд общих особенностей. Структура CRM систем представляет из себя набор таблиц, где хранится разнообразная информация, с которой работает пользователь. Пользователь через интерфейс системы может просматривать, изменять, удалять и анализировать эту информацию. При таком хранении информация не теряется, она легко доступна и повторно используема. На рисунке 1 представлен пример интерфейса CRM-системы — BasePlan. Впоследствии при разработке CRM решения будем отталкиваться от этой структуры, так как она выполняет главную функцию любой CRM-системы – хранение и управление данными о клиентах.

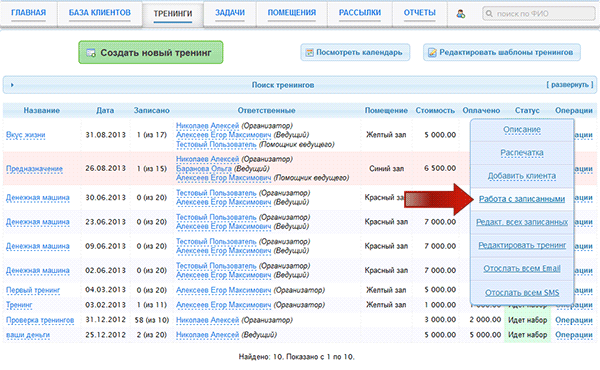


Рис. 1. Пример интерфейса CRM-системы BasePlan.

Система CRM в одной организации может применяться следующими лицами и отделами:

* Отдел продаж;
* Отдел маркетинга и PR;
* Служба поддержки;
* Руководитель.

1) Для отдела продаж CRM – это правая рука. Вся деятельность менеджеров абсолютно завязана на данной системе. В нее поступает информация о клиенте, с помощью нее совершаются звонки, проводится информирование клиента, ведется переписка. В этой системе сделка проходит все этапы и получает свой статус. Все обязанности менеджера продаж так или иначе исполняются в CRM или с ее помощью.

Для отдела продаж можно выделить основные выгоды от CRM систем:

* Централизованное хранение всей информации о клиенте: менеджерам доступна вся информация за считанные секунды;
* Информация унифицирована и легко приводима к виду отчета: так как информация в CRM системах хранится в максимально простом виде, ее легко использовать для создания отчетов;
* Процесс продаж упорядочен и строго зафиксирован: в CRM системе можно настроить нужный жизненный цикл сделки с клиентом, например: найти клиента – квалифицировать – первый звонок – письмо – презентация – сделка;
* CRM упрощает адаптацию новых сотрудников: в CRM системе можно легко выдать новому сотруднику доступ, а после освоения системы постепенно повышать его уровень доступа.

2) Для маркетинга CRM системы предоставляют доступ к профилю продаж, к записям о переговорах с клиентом, к вопросам и претензиям клиентов, к данным о цикле продаж и периоде до принятия решения. CRM помогает в следующих задачах:

* Рассылки клиентам: система помогает в создании выборок клиентов по какому-либо признаку и рассылке персонифицированных писем;
* Стимулирование отложенных сделок: благодаря вышеописанной рассылке, сотрудник может мотивировать клиентов к финализации сделки;
* Анализ клиентской базы: маркетолог воспользуется данными о продажах в CRM и сможет сделать выводы об удовлетворённости клиентов, их проблемах и сложностях. Исходя из этого можно будет создать продуктовое решение, которое закроет эти вопросы.

3) CRM в службе поддержки имеет спорный характер. Доступ для сотрудников службы поддержки является большой угрозой безопасности, так как в данном отделе наблюдается самая большая текучка кадров, а сами сотрудники при устройстве проверяются гораздо меньше. Поэтому могут происходить утечки информации, связанные с человеческих фактором. Масштабы данной проблемы можно сократить при правильной и строгой системе ролей и допусков, при которой каждый сотрудник будет иметь доступ только к тем модулям и функциям CRM, которые необходимы для осуществления их должностных обязанностей.

Для сотрудников службы поддержки CRM системы предоставляют следующие преимущества:

* Персонализированное обслуживание клиентов: CRM система предоставит сотруднику всю нужную информацию о клиенте, его прошлых обращениях и сделках, что значительно сократит время на выяснение подробностей;
* Выстраивается система разрешения спорных и конфликтных ситуаций: в системе хранятся все чаты и переговоры с клиентом, что помогает разобраться в подобных ситуациях и выявить причину проблемы. Из такой информации другие отделы могут выяснить основные причины недовольства клиентов и решать проблемы систематически.

4) Руководство само не продает, не занимается маркетингом и не занимается поддержкой. Поэтому для руководства CRM – это источник информации, подробный dashboard, в список сокращений и массив ~~цифр~~ данных, которые можно использовать для формирования отчетов и анализа KPI в список сокращений. Эта информация позволяет понять слабые и сильные стороны выстроенных бизнес-процессов и скорректировать стратегию в организации.

Таким образом, можно отметить основные характеристики CRM систем, которые присущи большинству организаций [2]. Система должна:

* Соответствовать требованиям вашего бизнеса;
* Покрывать большую часть задач автоматизации оперативной работы;
* Уменьшать, а не увеличивать время- и трудозатраты на рутинные действия;
* Быть безопасной;
* Предоставлять информацию для аналитики.

Описав выше способы эксплуатации и достоинства CRM-систем, надо уточнить очень значительный момент: далеко не все готовые CRM решения настолько универсальны, что будут иметь столь обширный функционал, чтобы покрыть все сферы деятельности внутри организации. И не всегда это нужно.

Также надо учесть, что любой развивающийся бизнес должен быстро адаптироваться к новым условиям современного рынка, развивать старые и принимать новые практики и расширять свою сферу деятельности. Для того, чтобы этот процесс был проще, быстрее и надежнее, надо адаптировать уже имеющиеся IT-системы в компании, а также разрабатывать новые. Поэтому модульность и скорость изменения CRM должна быть одним из приоритетов в разработке или покупке таких систем.

Как уже было сказано во введении, организации могут заниматься комплексными и нестандартными сферами деятельности, а их инфраструктура может состоять из глубоко интегрированных между собой систем со строгими политиками безопасности, где готовые CRM решения не будут являться идеальным или даже реальным вариантом для использования. В таких организациях обычно уже используется несколько IT-систем разного назначения и происхождения. Внедрение готового решения при таком устройстве потребует больших изменений в IT структуре организации. Поэтому разработка собственной CRM, которая будет отвечать всем условиям и процессам – это самый подходящий вариант. Такие внутренние системы будут отвечать за конкретные задачи в своей сфере. Также они будут интегрированы с остальными системами организации и передавать, получать и хранить только нужную информацию.

## Спецификация CRM систем

CRM-системы делятся на четыре основных типа [58]:

1. Стратегические CRM;
2. Операционные CRM;
3. Аналитические CRM;
4. Коллаборативные CRM.

1) Стратегические CRM – это ориентированные на клиента системы, нацеленные на привлечение и удержание прибыльных клиентов. Эти системы собирают и разделяют данные о клиентах, которые в свою очередь используются совместно с информацией о рыночных трендах, чтобы создавать более привлекательные предложения для клиентов.

Такой тип CRM использует бизнес, который считает мнение клиента важным для выживания. В противовес нацеленному на продукт бизнесу, клиентоориентированный бизнес постоянно изучает требования клиентов и адаптируется к ним. Подобный бизнес знает, что довольные клиенты будут совершать покупки куда чаще [3].

2) Операционные CRM основаны на таких функциях как продажа, маркетинг и обслуживание клиентов. Такие CRM занимаются автоматизацией этих процессов. Для них свойственен dashboard тут лучше написать по-русски, который дает общее представление о всех трех функциях на одной панели для каждого клиента (пример на рисунке 2).

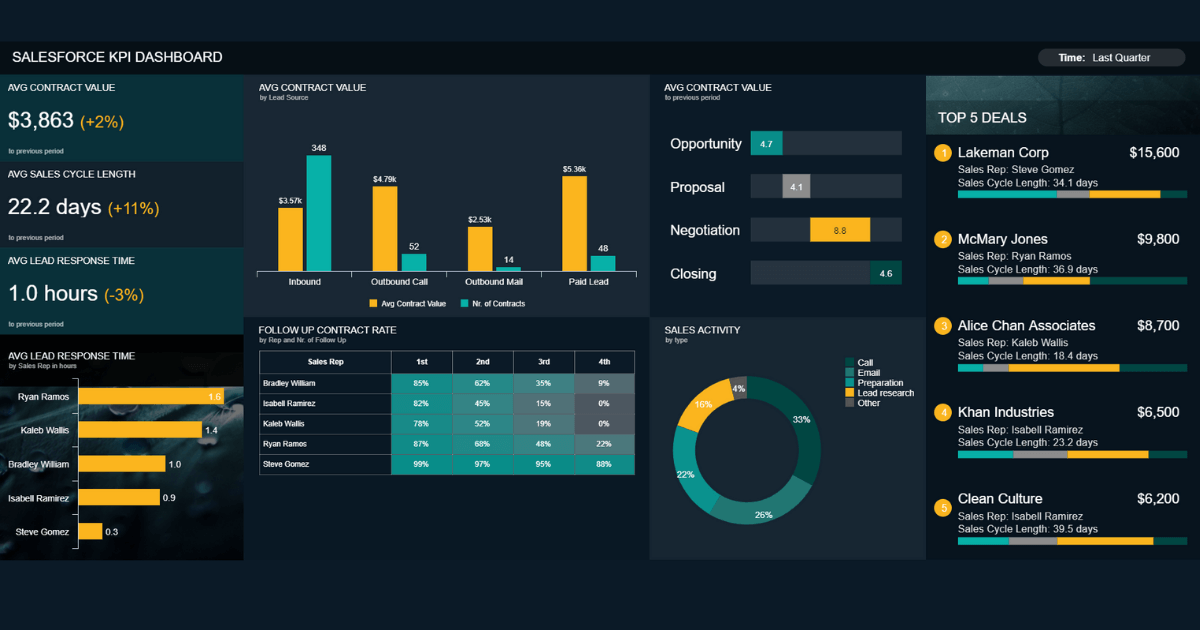


Рис. 2. Пример дэшборда CRM-системы [дай ссылку на источник инф].

Автоматизация продаж происходит на всех этапах, от первоначального ввода информации в систему, до превращения потенциального клиента в реального. Также автоматизируется отслеживание истории учетной записи клиента для повторных или будущих продаж, координация самих продаж, маркетинга, колл-центров и торговых точек. Это предотвращает дублирование действий между продавцом и покупателем (например: звонки с одним и тем же предложением или одинаковая рекламная рассылка), а также система автоматически отслеживает все контакты и действия между обеими сторонами.

Автоматизация маркетинга сфокусирована на упрощении процесса для достижения максимальной его эффективности. Определенные инструменты операционных CRM автоматизируют рутинные задачи по рассылке маркетинговых писем или по размещению маркетинговой информации в социальных сетях. CRM-системы сегодня также работают над привлечением клиентов через социальные сети.

Автоматизация поддержки фокусируется на технологиях прямой поддержки клиентов. С помощью данной системы поддержка клиентов происходит через несколько каналов, таких как: мобильные звонки, электронная почта, базы знаний, часто задаваемые вопросы и прочее [3, стр. 7].

3) Аналитические CRM основаны на сборе, интерпретации, разделении, хранении, изменении, обработке и представлении данных, связанных с клиентом. Также подобные CRM-системы хранят такие внутренние данные, как:

* Данные о продажах: продукты, объемы, история покупок;
* Финансовые данные: история покупок, кредитный рейтинг;
* Marketing Data: ответ на данные кампании, данные о схемах лояльности клиентов.

Организации бизнес-аналитики, которые предоставляют демографические данные и данные об образе жизни клиентов на большой территории, уделяют большое внимание внутренним данным, чтобы получить более подробную информацию, такую как “Какие клиенты является наиболее ценными?”, “Какие потребители положительно отреагировали на последнюю кампанию?” и прочее.

Аналитическая CRM может устанавливать разные подходы к продажам для разных сегментов клиентов. Кроме того, разным сегментам клиентов может быть предложен разный контент и стиль. Для клиентов аналитическая CRM дает индивидуальные и своевременные решения на основе анализа их отношения с компанией. Для бизнеса это дает больше возможностей для продаж, а также для привлечения и удержания клиентов [3].

4) Коллаборативные CRM занимаются созданием платформы для объединения внешних заинтересованных сторон, таких как: поставщики, продавцы и дистрибьюторы, и обменом информацией о клиентах между ними. Такая система используется в сценарии B2B сп. сокр., где множество бизнесов может совместно заниматься разработкой продуктов, исследованиями рынка и маркетингом. Некоторые коллаборативные CRM-системы могут иметь встроенные PRM-системы сп. сокр.для управления продвижением партнеров [4].

Также надо выделить отдельную группу ПО, смежную с CRM-системами – CDP (customer data platform). Это системы для агрегации и хранения данных о клиенте, собираемых из множества источников и доступных разным системам. Такие системы собирают данные и создают из них единый профиль клиента, который потом используется другими системами для разных действий [5]. CDP может собирать следующие типы данных:

* События клиентов: действия в браузере, действия на веб-сайте или в приложении, клики по баннеру и прочее;
* Транзакционные данные: данные, включая покупки, возвраты, данные с касс;
* Атрибуты клиентов: возраст, пол, день рождения, дата первой покупки, данные сегментации, прогнозы клиентов;
* Данные об оценке кампании: показы, клики, охват, вовлеченность и прочее;
* История отношений клиента и компания: данные взаимодействия со службой поддержки, показатели NPS (Индекс потребительской лояльности), данные чат-ботов, сообщения в социальных сетях, дословные опросы, записи колл-центра и прочее.

Разобрав основные типы CRM-систем и их предназначение, перейдем к основным компонентам таких систем. CRM-системы, как и любое современное ПО, состоит из множества компонентов, каждый из которых выполняет свою задачу. Ссылаясь на рассмотренную выше структуру CRM-систем, подразумевающую множество таблиц, с которыми взаимодействует пользователь для получения и обработки информации, можно выявить основные компоненты, такие как:

* Хранилище данных – это система для хранения данных о клиенте;
* Прослойка бизнес-логики – это система для обработки данных;
* Графический интерфейс – это основной компонент для взаимодействия пользователя с системой.

1) Хранилище данных представляет их себя базу данных, где хранятся все данные, используемые в CRM-системе. База данных может быть как реляционной, так и не-реляционной. Выбор полностью зависит от технических решений и задач CRM.

2) Прослойка бизнес-логики обычно представляет из себя промежуточное звено между хранилищем данных и интерфейсом. Это сервис, который получает данные из хранилища данных, обрабатывает их и отправляет в интерфейс, где пользователь уже взаимодействует с ними.

3) Графический интерфейс (также именуемый frontend) обычно представляется в двух видах: web-интерфейс и desktop-интерфейс, используемые тонким и толстым клиентами соответственно. Web-интерфейс отображается доступен из браузера с любого устройства, когда desktop-интерфейс требует установки и настройки на ПК. Как уже было сказано выше, интерфейс является компонентом взаимодействия пользователя и системы. Также интерфейс предоставляет дополнительную безопасность системе, например: ограничивает пользователю использование определенных функций или проверяет вводимые данные.

Условимся, что в практической части данной ВКР мы будет рассматривать только web-интерфейс - для использования через тонкий клиент.

4) Также надо упомянуть дополнительные компоненты, которые выполняют функции телефонии, рассылки, аналитики, отчетности и прочего. Данные компоненты также используются и настраиваются из интерфейса, и их реализация в основном зависит от требований бизнеса и технических решений.

Эти компоненты не будут рассматриваться в практической части данной ВКР, так как задача стоит не в разработке универсальной CRM-системы, а в подборе оптимальных технологий и архитектурных решений для разработки таких систем.

В следующей главе мы опишем спецификацию CRM-систем более подробно, проанализируем доступные нам средства разработки, возможное применение всевозможных концепций и сред разработки и постараемся выявить оптимальный в нашем случае инструментальный набор и финальную архитектуру, которые позволят создать набор компонентов и инструментов, упрощающих разработку и доработку ПО указанного типа, а также разработку рабочего прототипа CRM-системы с использованием данного набора средств.

## Существующие CRM решения

Как уже было сказано, CRM-системы бывают абсолютно разного типа и выполняют разные задачи в бизнесе. Пользователям предлагается множество готовых решений. Некоторое ПО предназначено для малых бизнесов, некоторое для больших международных организаций. Все эти решения по-своему уникальны и многофункциональны, и у каждого из них по отдельности есть свои достоинства и недостатки.

Перед разработкой собственной CRM-системы надо провести анализ нескольких самых известных решений: выявить в них главные недостатки, чтобы попытаться избежать их в нашем решении, а также плюсы — чтобы их не упустить в своем продукте.

Для анализа мы выбрали 3 самые популярные CRM-системы [6]:

* Salesforce;
* Freshworks;
* Pipedrive.

1) Самое популярное CRM решение на рынке – это Salesforce [7]. Данная система – это SaaS сп. сокр. платформа c доступом через Web и предоставляющая множество функциональных возможностей и модулей для пользователей. Это ведущее приложение для продаж, обслуживания и маркетинга. Пример интерфейса данной системы можно увидеть на рисунке 3.

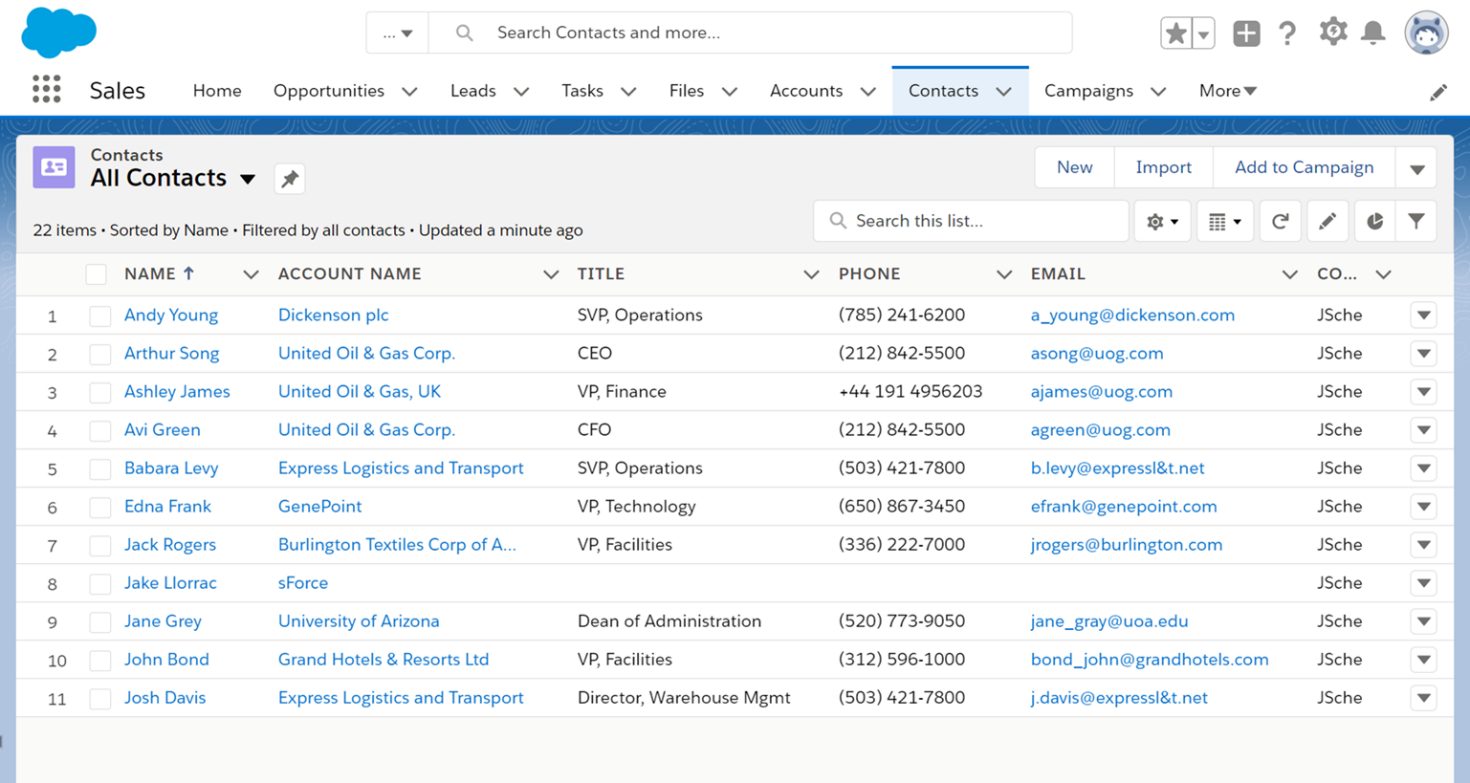


Рис. 3. Пример интерфейса Salesforce.

Эта система предоставляет следующие возможности:

* Управление данными клиентов / управление контактами;
* Территориальный менеджмент;
* Интеграция с почтовым клиентом (например, Outlook или Gmail);
* Управление котировками и заказами;
* Управление отношениями с каналами / партнерами;
* Управление рабочим процессом;
* Управление возможностями;
* Управление контрактами;
* Отслеживание взаимодействия;
* Рекламная рассылка;
* Управление задачами;
* Составление отчетов;
* Разрешения пользователей на основе ролей;
* Возможность единого входа в систему;
* Мобильный доступ.

Также данная система обладает следующими преимуществами над другими CRM системами:

* Работает в облачной системе, обеспечивая доступ из любого места;
* Обеспечивает множество бизнес-потребностей помимо CRM;
* Система, сильно настраиваемая в соответствии с потребностями бизнеса.

Но все же у системы Salesforce есть свои проблемы:

* Множеств функций системы требуют отдельной оплаты, поэтому ежемесячная стоимость подписки может существенно возрасти;
* Большое количество настроек может быть сложностью для сотрудников, что потребует их специального обучения, а это — затрата средств.

2) Вторым в нашем списке идет Freshworks CRM — это комплексное ПО для продаж с встроенными телефонией и электронной почтой, поддерживающее такие функции, как оценка потенциальных клиентов, визуальная воронка продаж, отчеты и информационные панели, автоматизация и мобильные приложения, гарантирующие, что пользователи не будут переключаться между несколькими инструментами, чтобы выполнить работу [8]. Пример интерфейса данной системы можно увидеть на рисунке 4.

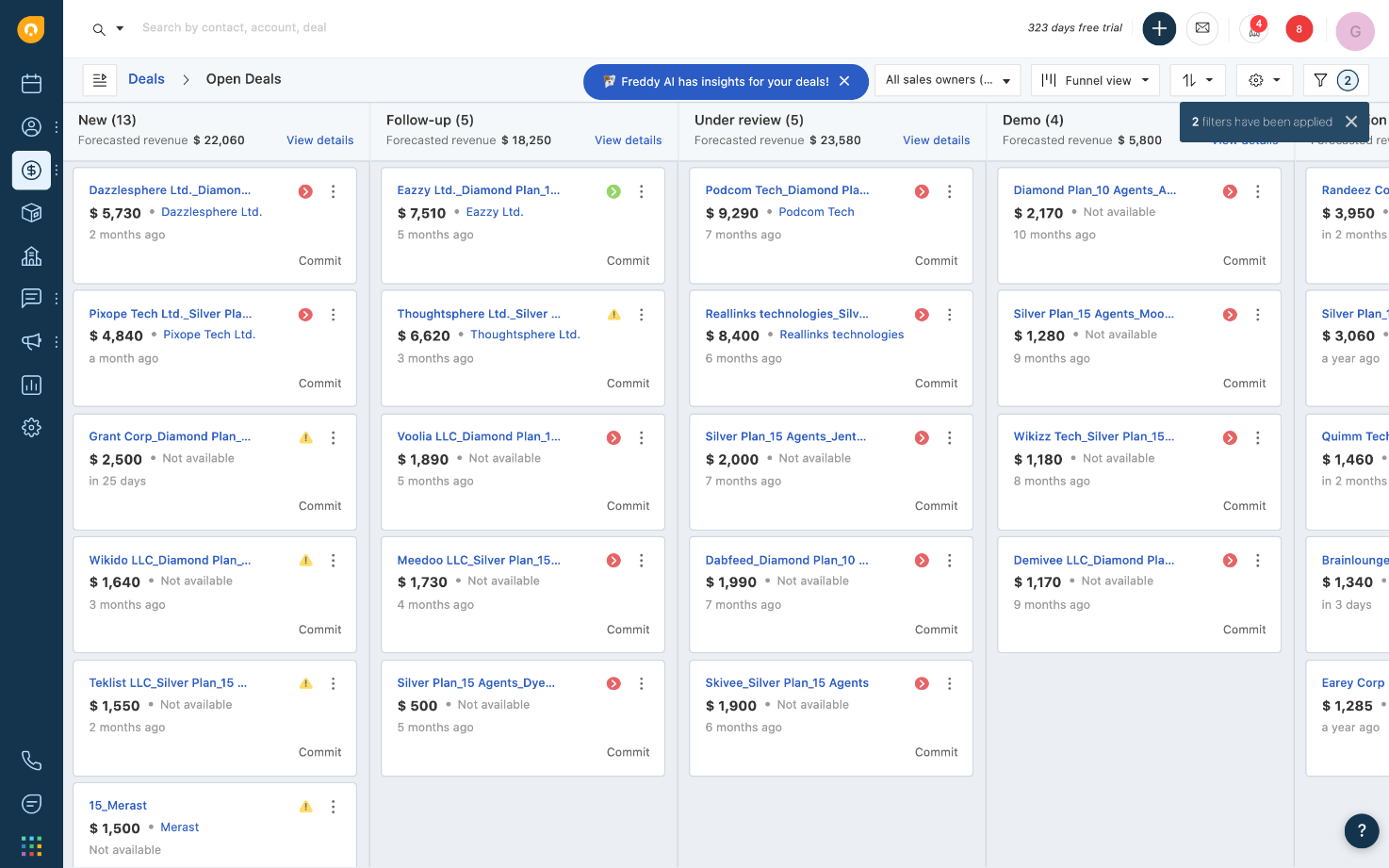


Рис. 4. Пример интерфейса Freshworks.

Freshworks CRM предоставляет следующие возможности:

* Управление данными клиентов / управление контактами;
* Территориальный менеджмент;
* Интеграция с почтовым клиентом;
* Управление отношениями с каналами / партнерами;
* Управление рабочим процессом;
* Управление возможностями;
* Отслеживание взаимодействия;
* Рекламная рассылка;
* Управление задачами;
* Составление отчетов;
* Разрешения пользователей на основе ролей;
* Возможность единого входа в систему;
* Мобильный доступ.

Данная система имеет следующие преимущества:

* Лиды и продажи организованы на основе интеллектуальной метрики данных;
* Система идеально подходит для начинающих компаний малого и среднего бизнеса;
* Настраивается в соответствии с потребностями конкретного бизнеса.

Данная система обладает следующими недостатками:

* Сторонние интеграции ограничены, что может увеличить затраты на поиск дополнительных платформ;
* Малая гибкость для настройки отчетов.

3) На третьем месте идет Pipedrive. Это портативная CRM система для бизнеса в начальной стадии развития [9]. Система фокусируется на контроле процесса продаж, позволяя масштабировать свой бизнес на полную мощность. Также Pipedrive очень легка в использовании. Пример интерфейса Pipedrive можно увидеть на рисунке 5.

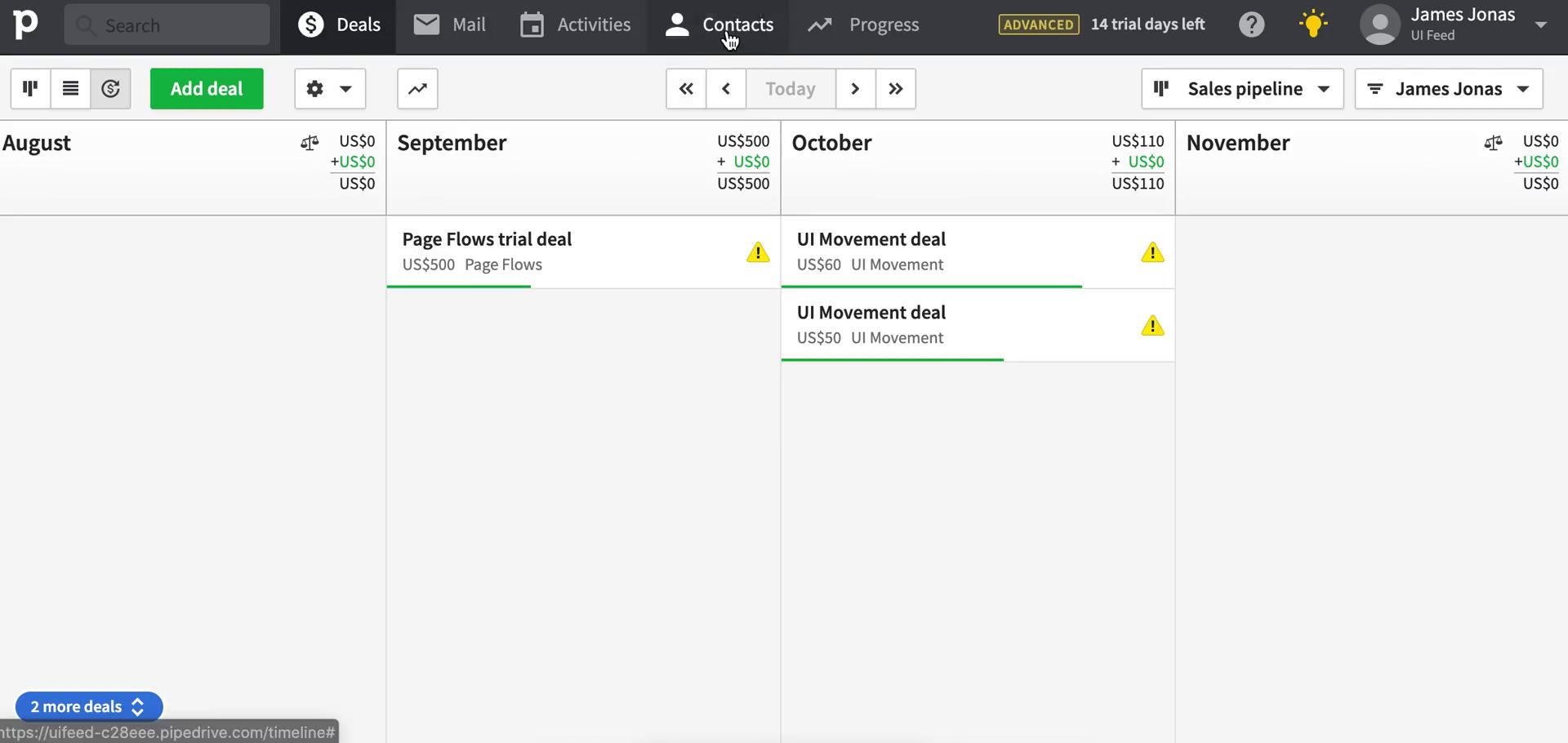


Рис. 5. Пример интерфейса Pipedrive.

Данная система имеет следующие возможности:

* Управление возможностями бизнеса;
* Интеграция с почтовым клиентом;
* Управление задачами;
* Составление отчетов;
* Визуализация конвейера продаж;
* Разрешения пользователей на основе ролей;
* Мобильный доступ.

У Pipedrive есть следующие преимущества:

* Отслеживание электронных писем в ПО, что облегчает поиск определенной информации;
* Простота в использовании и отслеживании лидов;
* Одна из лучших CRM для базового управления продажами.

А также следующие недостатки:

* Это CRM для бизнеса стартового уровня, так что это не лучший вариант для более крупного или более продвинутого бизнеса;
* Малая гибкость по настройке ПО;
* Отслеживание отчетов может быть утомительным и занимать много времени.

Также надо выделить один и самый важный момент относительно данных CRM-систем – это проприетарное ПО, поэтому даже самые гибкие готовые CRM решения не будут предоставлять достаточной гибкости в настройке и автоматизации бизнес-процессов в организации. Следовательно, если компания хочет идеальную CRM-систему, ей придется разрабатывать ее самой.

При создании своей CRM-системы мы постараемся заложить в нее максимальную гибкость для доработок и изменений, а также возможность интеграции с другими системами организации.

Проанализировав предметную область и существующие решения на рынке, мы можем переходить к выбору инструментария и архитектуры для разработки нашего ПО.

# ИНСТРУМЕНТАРИЙ ДЛЯ РАЗРАБОТКИ ПО

Цель данного этапа работы – подбор технологий и разработка с их помощью “каркаса” CRM-системы. Этот каркас будет основой для программного конструктора, который позволит легко и быстро дорабатывать существующие, а также разрабатывать новые модули CRM-системы. Само ПО при этом будет портативным и масштабируемым.

Также в основу каркаса будут вложены механизмы, обеспечивающие информационную безопасность: функции идентификации, авторизации, аутентификации, валидации.

Проведем технический анализ основных компонентов CRM-системы и определим основные требования к их разработке.

Начнем с хранилища данных. Основный объем данных в CRM-системах – это типизированные данные о взаимоотношении организации и клиентов. Задачу эффективного хранения таких данных идеально выполняют реляционные БД. Такие БД следуют основным принципам ACID (atomicity, consistency, isolation, durability) [10]:

1. Atomicity — атомарность: каждый оператор в транзакции (для чтения, записи, обновления или удаления данных) рассматривается как единое целое. Либо выполняется весь оператор, либо не выполняется ни один из них. Атомарная система должна гарантировать атомарность в любой ситуации, включая сбои питания, ошибки и падения системы.
2. Consistency — согласованность: транзакции вносят изменения в таблицы только предопределенными и предсказуемыми способами. Согласованность транзакций гарантирует, что повреждение или ошибки в данных не приведут к непредвиденным последствиям для целостности таблицы.
3. Isolation — изолированность: когда несколько пользователей работают с одной таблицей в режиме «чтение/запись» одновременно, изоляция их транзакций гарантирует, что параллельные транзакции не будут влиять друг на друга. Каждый запрос из группы может выполняться так, как если бы все они выполнялись один за другим, хотя на самом деле они выполняются одновременно.
4. Durability — прочность: изменения в данных, сделанные успешно выполненными транзакциями, будут сохранены даже в случае сбоя системы.

Далее рассмотрим серверный компонент системы, или сервер (далее backend).

Основные задачи этого сервиса:

1. Проведение транзакций с БД;
2. Предоставление API для общения с клиентской частью приложения;
3. Обработка данных при чтении, сохранении и изменении;
4. Реализация бизнес-логики;
5. Реализация основных механизмов информационной безопасности.

Определим основные требования, предъявляемые к данному сервису:

1) Backend должен быть легко дорабатываемым, что ускорит процесс модернизации и сделает дешевле изменения в CRM-системе.

2) Backend должен быть портативным и независимым от других компонентов CRM-системы. Данное свойство позволит легко масштабировать его, то есть запускать несколько экземпляров данного сервиса, и балансировать (балансировка заключается в распределении запросов на несколько экземпляров одного сервиса) эти экземпляры. Масштабирование позволит контролировать нагрузку на CRM-систему и, в зависимости от этой нагрузки, выделять больше или меньше вычислительных мощностей для системы.

3) Backend должен иметь полный контроль над структурой таблиц БД. Все изменения в структуре данных и их отношениях должны быть сразу отражены в БД. Подробнее об этом свойстве будет сказано ниже при обсуждении архитектуры разрабатываемой системы.

Рассмотрим основные механизмы информационной безопасности, которые должен реализовывать сервис:

1. Идентификация — процедура, в результате выполнения которой для субъекта идентификации выявляется его идентификатор, однозначно определяющий этого субъекта в информационной системе;
2. Аутентификация — процедура проверки подлинности, например, проверка подлинности пользователя путем сравнения введенного им пароля с паролем, сохраненным в базе данных;
3. Авторизация — предоставление определенному лицу или группе лиц прав на выполнение определенных действий;
4. Cross-Origin Resource Sharing (CORS) — это механизм, основанный на анализе HTTP-заголовков, который позволяет указывать источники (домен, схему или порт), из которых будет разрешена загрузка ресурсов на сервер. CORS также реализует механизм, позволяющий браузеру сделать «предварительный» запрос на сервер, где размещен ресурс из стороннего источника, чтобы убедиться, что сервер разрешит фактический запрос. В ходе предварительной проверки браузер отправляет заголовки, указывающие метод HTTP, и заголовки, которые будут использоваться в фактическом запросе [11].

В заключение рассмотрим графический интерфейс, или клиентскую часть (далее frontend). Этот компонент будет отвечать за отрисовку интерфейса и взаимодействие с backend`ом системы для получения и обработки данных.

Определим требования к frontend`у:

1) Frontend должен быть легко дорабатываемым;

2) Все элементы интерфейса должны быть компонентами, которые подразумевают повторное использование;

3) Frontend должен быть удобным и интуитивно понятным пользователю;

4) Пользователь должен видеть уведомления при совершении определенных действий в интерфейсе системы, например, удаление записи или переход на страницу, к которой у пользователя нет доступа;

5) На frontend части системы должен быть реализован механизм валидации данных, которые вводит пользователь при создании или изменении записи;

6) Как и backend, frontend должен быть независимым от других компонентов системы, что позволит его масштабировать.

## Описание архитектуры CRM-системы

Для того чтобы определить необходимые для последующей работы языки программирования и технологии разработки, мы должны определиться с архитектурой самой системы.

Существует множество архитектур для информационных систем: монолит, сервис-ориентированная, микросервисный монолит и другие. При выборе архитектуры играет множество факторов, таких как цель и задачи системы, важные архитектурные требования, особенности обрабатываемых данных и так далее. Мы остановились на микросервисной архитектуре, исходя из основных требований к компонентам системы. Эти критерии таковы: компоненты системы должны быть максимально возможно независимы друг от друга, масштабируемы, портативны, а также позволять просто и быстро дорабатывать существующий и создавать новый функционал. Микросервисная архитектура позволяет соответствовать данным критериям по следующим причинам:

1) В микросервисной архитектуре компоненты системы могут дорабатываться и выноситься на продукционное окружение независимо друг от друга;

2) Компоненты в данной архитектуре являются простыми и портативными, что позволяет их легко и быстро дорабатывать и масштабировать.

Микросервисная архитектура – это архитектура ПО, направленная на взаимодействие небольших, слабо связанных и легко изменяемых модулей – микросервисов. Основные ее особенности [12]:

1. Модули системы легко заменить в любой момент времени: акцент делается на простоту и на независимость модулей;
2. Каждый модуль выполняет лишь одну функцию;
3. Разные модули одной системы могут быть реализованы на разных языках программирования, фреймворках, выполняться в различных средах контейнеризации, виртуализации, под управлением различных операционных систем на различных аппаратных платформах: приоритет отдается эффективности модуля, а стандартизация средств разработки и исполнения отходит на второй план;
4. Зависимости между модулями одноранговые: все модули находятся на одной иерархической ступени в системе, а значит все модули могут взаимодействовать между собой без модулей-посредников.

Микросервисная архитектура значительно упрощает backend компонент системы, благодаря распределению ряда задач системы между разными сервисами, что в свою очередь благоприятствует масштабированию и отказоустойчивости.

## Коммуникация между компонентами системы

В качестве способа взаимодействия компонентов системы мы решили использовать RESTful API по следующим причинам [13]:

1) RESTful API опирается только на основной протокол, используемый веб-приложениями, HTTP, и не требует дополнительных систем и протоколов для его использования. Это в свою очередь упрощает архитектуру отдельных компонентов и всей системы, а также ее стабильность.

2) RESTful API поддерживает множество форматов данных, в том числе JSON и файлы (например: PDF, DOCX, XML и другие), что позволит развивать функционал системы.

3) RESTful API использует минимум пропускной способности сети для передачи данных.

API (Application Programming Interface) — это прикладной программный интерфейс, который позволяет взаимодействовать двум и более системам через программный код. API определяет способ взаимодействия разных систем, принципы их общения, а также ограничения, накладываемые этими системами [14].

RESTful API (REpresentational State Transfer) — это архитектурный стиль прикладного программного интерфейса (API), который использует HTTP-запросы для доступа к данным и их использования. В RESTful API данные именуются ресурсами, они доступны по заранее определенным идентификаторам URI (Uniform Resource Identifier). К этим ресурсам можно обращаться с помощью HTTP-запросов GET, PUT, POST и DELETE, которые используются для чтения, обновления, создания и удаления ресурсов, соответственно [15].

Рассмотрим базовые принципы RESTful API.

1) Единый интерфейс. Комплекс архитектурных ограничений помогает получить единый интерфейс и управлять поведением компонентов. Следующие четыре ограничения обеспечивают единый RESTful интерфейс [15]:

1. Идентификация ресурсов. Интерфейс должен однозначно идентифицировать каждый ресурс, участвующий во взаимодействии между клиентом и сервером.
2. Манипулирование ресурсами через представления. Ресурсы должны иметь единообразные представления в ответе сервера. Потребители API должны использовать эти представления для изменения состояния ресурсов на сервере.
3. Сообщения с самоописанием. Каждое представление ресурса должно содержать достаточно информации для описания правил обработки сообщения. Оно также должно предоставлять информацию о дополнительных действиях, которые клиент может выполнять с ресурсом.
4. Гипермедиа, как механизм состояния приложения. У клиента должен быть только URI приложения. Клиентское приложение динамически управляет всеми другими ресурсами и функционалом систем с использованием гиперссылок.

2) Распределение функций между клиентской и серверной частями. Шаблон проектирования клиент-сервер принуждает к разделению обязанностей, что помогает компонентам frontend и backend развиваться независимо друг от друга. Благодаря разделению компонентов упрощается портативность на разные платформы, а также улучшается масштабируемость из-за упрощения backend компонента.

3) Многоуровневая система. В такой системе каждый компонент может общаться только со следующим по иерархической ступени. Надо уточнить, что это ограничение не противоречит четвертому свойству микросервисной архитектуры, так как речь идет об общении между frontend и backend компонентами системы. Например, frontend может взаимодействовать со всеми сервисами на серверной части системы, но не может напрямую взаимодействовать с БД.

4) Кэширование. Ответы от сервера должны явно или неявно указывать их кэшируемость. Если ответ кэшируемый, то клиент может пере-использовать данные из ответов на подобные запросы на определенный период. В CRM-системе кэширование предпочтительно не использовать чтобы пользователи не получали устаревшие и некорректные данные.

Перечисленные принципы делают RESTful API надежным и удобным инструментом.

Коммуникация посредством RESTful API в значительной степени упрощает взаимодействие компонентов системы и позволяет создавать пере-используемые сервисы, которые могут развиваться независимо от остальных компонентов.

## Описание frontend компонента

Обычно веб-приложения используют один и данных механизмов: Server-Side Rendering и Client-Side Rendering [16].

SSR (Server-Side Rendering) – это механизм генерации HTML-страниц, который позволяет заполнить HTML-шаблон данными перед тем как отправить его клиенту. Данный механизм ускоряет получение готовой страницы с данными пользователю [17].

CSR (Client-Side Rendering) – это механизм, позволяющий загруженному в браузер пользователя JavaScript коду взаимодействовать с сервером(-ами) для получения и отправки данных, с которыми работает пользователь. При этом, при получении или отправке данных, страница не перезагружается [17].

Веб приложения, использующие SSR, обычно разрабатываются с помощью backend фрэймворков, таких как: Spring, Django, Gin и другие. В таких приложениях страница генерируется и отправляется пользователю посредством фрэйворка. В свою очередь фрэймворк использует шаблонизатор для генерации страницы и добавления в нее данных [18]. Сегодня такой подход к разработке веб-приложений является устаревшим по причинам сильной связанности backend и frontend, а также потребности обновлять страницу для изменения данных. Также с таким подходом пере-использование компонентов интерфейса усложняется и требует дополнительных инструментов.

Современные веб-приложения в основном используют CSR и разрабатываются на frontend библиотеках и фрэймворках, таких как Vue.js, Angular, Next.js, Nuxt.js, React и другие. Ключевая особенность данных технологий заключается в том, что они позволяют создать независимые от серверной части веб-приложения. При использовании данных технологий frontend компонент системы становится независим от остальных компонентов, имеет свой жизненный цикл и может поддерживаться отдельной командой разработчиков [19]. Также данные технологии позволяют, при разработке веб-приложений, в реальном времени видеть все изменения без необходимости повторно собирать приложение.

Обычно CSR используется в SPA веб-приложениях. SPA (Single Page Application) – это одностраничное веб-приложение, которое загружается на одну HTML-страницу. Благодаря динамическому обновлению с помощью JavaScript, во время использования не нужно перезагружать или подгружать дополнительные страницы. На практике это означает, что пользователь видит в браузере весь основной контент, а при прокрутке или переходах на другие страницы, вместо полной перезагрузки нужные элементы просто подгружаются [20]. Такие приложения предоставляют более плавный и бесшовный опыт использования за счет более долгой первой загрузки сайта и большей нагрузки на систему пользователя.

Применение SPA, в свою очередь, позволяет создавать и использовать компоненты. Компоненты в контексте web-разработки – это независимые и пере-используемые части графического интерфейса. Компоненты позволяют разделить интерфейс на отдельные части, которые могут содержать в себе как стили, так и логику, а также думать о каждом компоненте в изоляции от других [21].

Также стоит упомянуть, что современные frontend технологии поддерживают использование CSR вместе с SSR. В таком случае пользователь получает сгенерированную страницу, а при дальнейших действиях JS подгружает нужные данные. Помимо всех вышеописанных преимуществ CSR и SSR, такой подход позволяет сократить первоначальную загрузку страницы, не потеряв плавности использования приложения.

Мы определили, что самый оптимальный вариант для создания frontend компонента системы – это создание SPA веб-приложения с использованием комбинации CSR и SSR. Данное решение принималось по следующим критериям:

* Возможность выбора оптимального механизма генерации страниц для каждой задачи;
* Использование компонентной системы, которая позволит создавать пере-используемые независимые элементы интерфейса;
* Ускоренная разработка благодаря мгновенном отображении изменений при изменении кода.

Такой подход к разработке интерфейса позволит нам создать логический скелет и набор компонентов для упрощения дальнейшей разработки новых модулей CRM-системы.

## Описание backend сервисов

Так как мы используем микросервисную архитектуру, очень важно определиться с арихтектурой каждого отдельного сервиса. В нашем случае будет создан сервис, который выполняет задачи по созданию и обработке нескольких структур данных. Для подобной задачи мы будем использовать шаблоны проектирования MVC так как это самое популярное решение для данной задачи.

MVC (Model-View-Controller) – это шаблон проектирования web-приложений, используемый для реализации пользовательских интерфейсов, который разделяет программную логику на три основных элемента [22]:

1. Model (Модель) – это центральный элемент, определяющий структуры данных, с которыми будет работать система.
2. View (Представление) – это элемент, который определяет, как данные будут представлены пользователю. Следует уточнить, что при совместном использовании с Repository Pattern, который описан ниже, View будет выполнять роль набора данных, с которым будет работать frontend нашей системы.
3. Controller (Контроллер) – это элемент, выполняющий функции посредника между Model и View. Данный элемент содержит в себе бизнес логику, а также создает API, о котором сказано выше.

Что с нумерацией пунктов? Начинается с 5-го. М.б., конечно, это из-за моего Linux'а, но проверь.

Вместе с MVC мы будем использовать популярный шаблон проектирования Repository Pattern. Repository Pattern (Шаблон Репозитория) – это шаблон проектирования, который позволяет абстрагироваться от взаимодействия с базами данных. При таком подходе таблицы абстрагируются до уровня коллекции данных, что позволяет добавлять, удалять, обновлять и получать данные с помощью простых методов [23]. Для реализации данного шаблона проектирования используют ORM (Object–Relational Mapping) технологии, которые позволяют создавать так называемые Repository.

Repository (Репозиторий) – это интерфейс, который абстрагирует таблицу в БД и позволяет выполнять все необходимые задачи по хранению и обработке данных с помощью привычного разработчику языка программирования [24].

Стоит уточнить, что с применением ORM структура таблицы в БД соответствует той, что описана в Model. Благодаря этому в программном коде все модели разделяются по доменной области и имеют полный набор данных, нужных для бизнес-логики в данном домене.

Все вышеописанные особенности и преимущества Repository Pattern соответствуют принципу Persistence Ignorance, который гласит, что классы, моделирующие бизнес-домен в системе, не должны зависеть от того, как они могут храниться. Таким образом, их дизайн должен максимально точно отражать идеальный дизайн, необходимый для решения текущей бизнес-задачи, и не должен быть изменен проблемами, связанными с тем, как состояние объектов сохраняется и впоследствии извлекается [25].

## Финальная архитектура CRM-системы

Определившись с архитектурой отдельных компонентов системы и способами их взаимодействия, мы построили схему, отражающую финальную архитектуру нашей системы, которую можно увидеть на рисунке 6.

Данная архитектура позволит разработчикам легко и надежно дорабатывать и расширять функциональность CRM-системы, масштабировать ее и интегрировать с внешними системами.



Рис. 6. Финальная схема CRM-системы.

## Языки программирования и технологии

Для реализации вышеописанной архитектуры мы подобрали подходящий инструментарий для разработки системы.

В процессе подбора технологий и языков программирования мы ориентировались не только на требования к системе и на особенности ее архитектуры, но и на их популярность и активность использования сообществом программистов. Эти факторы характеризуют уровень поддержки используемого инструментария, доступность информации и примеров реализации тех или иных компонентов и механизмов.

Одна из задач разработки нашей CRM-системы – это создание программного конструктора для упрощения и ускорения разработки. Поэтому предполагается использовать технологии для разработки каркаса, на котором будет создаваться данная система. Эти технологии позволят разбить все компоненты системы на отдельные части и легко их пере-использовать для дальнейшей разработки.

## Инструментарий разработки backend

Для разработки сервисов мы будем использовать язык программирования Java. Java — это высокоуровневый, основанный на классах, объектно-ориентированный язык программирования. Это язык программирования общего назначения, предназначенный для того, чтобы программисты могли, написав код один раз, запускать его на любом окружении. Cкомпилированный код Java может работать на всех платформах, поддерживающих Java, без необходимости повторной компиляции. Приложения Java обычно компилируются в байт-код, который может работать на любой виртуальной машине Java (JVM) независимо от базовой компьютерной архитектуры. Синтаксис Java подобен C и C++, но имеет меньше низкоуровневых средств, чем любой из них. Среда выполнения Java предоставляет динамические возможности (такие как рефлексия и модификация кода во время его исполнения), которые обычно недоступны в традиционных компилируемых языках [26]. По данным GitHub, по состоянию на 2021 год Java был одним из самых популярных языков программирования, который стабильно в период 2015-2021 годов занимал третье место по популярности [27].

Наш выбор пал на Java по следующим причинам:

1) Большое и активное сообщество программистов, благодаря чему легко найти информацию о решении многих проблем, с которыми может встретиться разработчик при реализации тех или иных компонентов системы. Также сообщество делится хорошими практиками и опытом, что помогает в написании чистого и читаемого кода, что, в свою очередь, упрощает дальнейшую разработку.

2) Широкий выбор технологий. На языке Java есть множество фрэймворков и библиотек, это предоставляет богатый выбор инструментов для решения разных задач.

3) Простота реализации многих механизмов. Благодаря широкому выбору технологий и мощной начальной библиотеке Java, большинство механизмов можно реализовать за минимальный период времени с минимальной кодовой базой.

4) Объектно-ориентированный язык. Для работы с бизнес логикой и большим объемом типизированных данных лучше всего подходят ООП языки. Определившись с выбором языка программирования, мы подобрали Java -технологии, которые будут использованы для реализации backend компонента системы.

За основу сервиса мы взяли фрэймворк Spring. Spring – это один из самых популярных фрэймворков для создания веб-приложений на Java. Spring выделяется среди других технологий благодаря особенностям, которые упрощают и ускоряют разработку [28]:

1) IOC (Inversion Of Control), инверсия управлния. В программной инженерии инверсия управления — это принцип программирования. IOC инвертирует поток управления по сравнению с традиционным потоком управления. В IOC специально написанные части компьютерной программы получают поток управления от фрэймворка. Архитектура программного обеспечения с таким дизайном инвертирует управление по сравнению с традиционным процедурным программированием: в традиционном программировании код обращается к повторно используемым библиотекам для решения общих задач, но с инверсией управления – это фрэймворк, который вызывает пользовательский или специфичный для задачи код.

2) DI (Dependency Injection), внедрение зависимостей. В разработке программного обеспечения внедрение зависимостей — это шаблон проектирования, используемый для получения объектом других объектов, от которых он зависит. Внедрение зависимостей – это форма инверсии управления, которая направлена на разделение задач по созданию объектов и их использованию, что приводит к слабосвязанному программному коду. Шаблон гарантирует, что объект-А, который хочет использовать объект-Б, не должен знать, как создавать объект-Б. Вместо этого принимающий объект-А (или «клиент») получает свои зависимости от внешнего кода («инжектора»), о котором он не знает. Внедрение зависимостей решает следующие проблемы:

* Независимость классов от создания объектов, от которых они зависит;
* Использование разных конфигураций для разных объектов в одном приложении;
* Возможность изменения поведения фрагмента кода, не редактируя его напрямую.

[IOC и DI это очень сложные механизмы, которые никак нельзя по-человечески объяснить. Я когда их изучал, перерыл пол интернета и в итоге просто на практике понял что и как. Так что тут я вообще адекватно это не объясню. **Ладно, пусть будет в том виде, как ты написал, но обязательно надо добавить ссылки на источники по каждому пункту.**]

3) Авто-конфигурация. Spring автоматически конфигурируется, основываясь на зависимостях. Например, если указать в зависимостях in-memory базу данных и не указать контейнер для запуска приложения, то Spring самостоятельно настроит соединение с базой данных и запустит приложение во встроенном контейнере. Авто-конфигурация в Spring не инвазивная, это значит, что в любой момент можно самостоятельно создать конфигурацию для любого механизма в фрэймворке.

4) Главная особенность данного фрэймворка заключается в том, что он также является платформой для других небольших фрэймворков (далее модулей), а его экосистема состоит из 21 проекта, некоторые из них мы обсудим ниже.

Для разработки нашей системы за основу мы взяли модуль Spring Web – это набор взаимозависимых компонентов, который включает [29]:

* spring-core – это ядро фрэймворка Spring;
* spring-boot-starter – это стартер ядра Spring, который включает в себя поддержку авто-конфигурации, логирования и создания конфигурационных файлов на языке YAML.
* spring-boot-starter-tomcat – это расширение стартера Spring, которое используется для запуска приложения на встроенном Tomcat сервере.
* spring-boot-starter-validation - это расширение стартера Spring, которое добавляет валидацию объектов.
* jackson – это библиотека, используемая Spring для маршализации и демаршализации данных в Json формат, пригодный для передачи через HTTP протокол.
* spring-web – это набор взаимозависимых компонентов, позволяющих создавать RESTful веб-приложения на основе MVC шаблона проектирования.

Применяя ORM в процессе разработке системы, мы будем использовать модуль Spring Data JPA. Это набор взаимозависимых компонентов, позволяющий использовать Hibernate – ORM, написанную на языке Java по спецификации JPA (Java Persistence API) [30]. Hibernate предоставляет весь функционал современных ORM технологий, а также поддерживает валидацию [31]. Благодаря этому модулю мы можем реализовать шаблон проектирования Repository Pattern.

Для обеспечения безопасности мы будем использовать модуль Spring Security. Это фрэймворк, который позволяет реализовать большинство механизмов информационной безопасности, таких как идентификация, аутентификация и авторизация. Одно из главных достоинств Spring Security заключается в том, что он позволяет интегрировать в систему множество механизмов без необходимости изменять остальной код приложения [32].

Для сокращения шаблонного кода на языке Java мы будем использовать библиотеку Lombok. Lombok – это плагин компилятора, который добавляет в Java новые «ключевые слова» и превращает аннотации в Java-код, уменьшая усилия на разработку и обеспечивая некоторую дополнительную функциональность. Данный плагин позволяет с помощью простых аннотаций создавать геттеры, сеттеры, пустые и полные конструкторы, а также перегрузку методов класса Object, от которого неявно наследуются все классы в Java [33]. Применение Lombok вместе со Spring дает ряд преимуществ, в частности, упрощение Dependency Injection и удобное логирование.

В качестве сборщика мы будем использовать Gradle. Gradle — это инструмент автоматизации сборки для разработки программного обеспечения. Он контролирует процесс разработки от задач компиляции и упаковки до тестирования, развертывания и публикации [34]. Поддерживает языки Java (а также Kotlin, Groovy, Scala), C/C++ и JavaScript. Gradle также работает как пакетный менеджер и использует глобальный репозиторий Maven (популярный сборщик для Java проектов). Gradle основывается на концепциях Apache Ant и Apache Maven и представляет предметно-ориентированный язык на основе Groovy и Kotlin, в отличие от конфигурации на основе XML, используемой Maven. Gradle использует направленный ациклический граф, чтобы определить порядок, в котором могут выполняться задачи, посредством обеспечения управления зависимостями. Он работает на виртуальной машине Java. Gradle был разработан для много-проектных сборок, которые могут перерасти в гигантские проекты. Он работает на основе серии задач сборки, выполняемых последовательно или параллельно. Инкрементные сборки поддерживаются путем определения частей дерева сборки, которые уже обновлены; любая задача, зависящая только от этих частей, не требует повторного выполнения. Он также поддерживает кэширование компонентов сборки в общей сети с использованием кэша сборки Gradle. Он создает веб-визуализацию сборки под названием Gradle Build Scans. Программное обеспечение расширяется для добавления новых функций и языков программирования с помощью подсистемы подключаемых модулей.

## 2.5.2. Механизм аутентификации

В качестве механизма аутентификации мы будем использовать JWT. JSON Web Token – это отрытый стандарт RFC 7519 [35] для создания токенов доступа в формате JSON. Как правило, используется для передачи данных для аутентификации в клиент-серверных приложениях. Токены создаются сервером, подписываются секретным ключом и передаются клиенту, который в дальнейшем использует данный токен для подтверждения своей личности [36]. Мы выбрали данный механизм по следующим причинам:

* Аутентификация на основе токенов более масштабируема и эффективна. Поскольку токены должны храниться на стороне пользователя, они предлагают масштабируемое решение. Более того, серверу просто нужно создать и проверить токены вместе с информацией, что означает, что одновременное обслуживание большего количества пользователей в системе не будет требовать много вычислительных ресурсов.
* Гибкость и производительность. Гибкость и повышенная общая производительность являются другими важными аспектами аутентификации на основе токенов, поскольку они могут использоваться на нескольких серверах и предлагать аутентификацию для различных веб-сайтов и приложений одновременно. Это помогает расширять возможности интеграции между системами и платформами для обеспечения бесшовного взаимодействия с ними.
* Токены обеспечивают надежную защиту. Поскольку токены, такие как JWT, не имеют состояния, только сервер, который их создал, может проверить их при получении с помощью секретного ключа. Следовательно, они считаются лучшим и наиболее безопасным способом аутентификации.

JWT токен состоит из трех частей: заголовок (header), полезные данные (payload) и подпись (signature). Header и payload – это JSON объекты с определенной структурой, а signature вычисляется на основании первых двух и зависит от выбранного алгоритма.

Заголовок содержит в себе информацию о том, как должен вычисляться JWT токен. В заголовке используются следующие поля:

* alg: алгоритм, используемый для подписи/шифрования (в случае неподписанного JWT используется значение «none»).
* typ: тип токена. Используется в случае, когда токены смешиваются с другими объектами, имеющими JOSE заголовки. Должно иметь место значение «JWT».
* cty: тип содержимого. Если в токене, помимо зарегистрированных служебных ключей, есть пользовательские, то данный ключ не должен присутствовать. В противном случае должно иметь место значение «JWT».

Полезные данные – это данные, которые хранятся в JWT. Эти данные также называют JWT-claims. Полезные данные могут включать в себе следующие поля:

* iss: чувствительная к регистру строка или URI, которая является уникальным идентификатором стороны, генерирующей токен.
* sub: чувствительная к регистру строка или URI, которая является уникальным идентификатором стороны, о которой содержится информация в данном токене. Значения с этим ключом должны быть уникальны в контексте стороны, генерирующей JWT.
* aud: массив чувствительных к регистру строк или URI, являющийся списком получателей данного токена. Когда принимающая сторона получает JWT с данным ключом, она должна проверить наличие себя в получателях — иначе проигнорировать токен.
* exp: время в формате Unix Time, определяющее момент, когда токен станет невалидным.
* nbf: в противоположность ключу exp, это время в формате Unix Time, определяющее момент, когда токен станет валидным.
* jti: строка, определяющая уникальный идентификатор данного токена.
* iat: время в формате Unix Time, определяющее момент, когда токен был создан. iat и nbf могут не совпадать, например, если токен был создан раньше, чем время, когда он должен стать валидным.

Подпись надежно валидирует JWT токен. Подпись вычисляется путем кодирования заголовка и полезных данных с помощью Base64url кодировки, описанной в RFC 4648. После этого мы пропускаем эту строку через криптографический алгоритм, который указали в заголовке.

После получения всех трех частей, мы объединяем их через точку и получаем строку, подобную нижеследующей: eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOiJiMDhmODZhZi0zNWRhLTQ4ZjItOGZhYi1jZWYzOTA0NjYwYmQifQ.-xN\_h82PHVTCMA9vdoHrcZxH-x5mb11y1537t3rGzcM.

Рассмотрим последовательность использования токена.

1) Пользователь заходит на сервер аутентификации с помощью какого-либо ключа. Это может быть комбинация логин/пароль или ключ от любого провайдера аутентификации (Google, VK и другие).

2) Сервер аутентификации создает JWT токен по вышеописанной схеме и отправляет его пользователю.

3) При любом запросе к API системы, этот токен добавляется к запросу.

4) Когда сервер получает запрос, он проверяет токен на валидность. При успешной проверке пользователь получает ответ данными.

Схематически данный процесс представлен на рисунке 7.

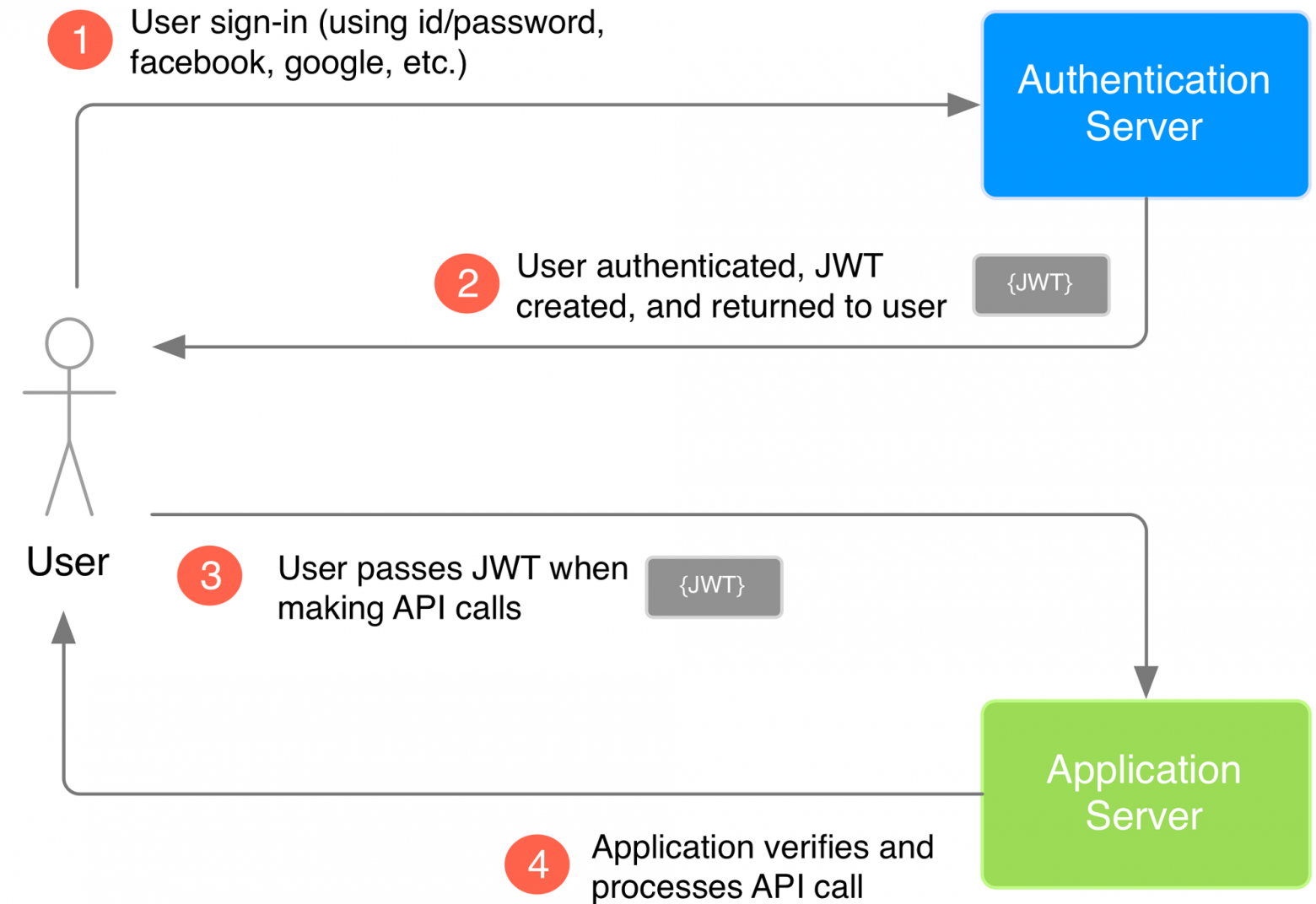


Рис. 7. Схема применения JWT токена.

Процесс проверки JWT токена при запросах проходит следующим образом. Так как сервер знает секретный ключ токена, то он может проверить токен, полученный из запроса пользователя, выполнив тот же алгоритм подписывания к токену. Сервер потом сравнивает полученную подпись свычисленной хэшированием. Если подписи не совпадают, то это может быть потенциальной атакой.

JWT токены при получении пользователем после аутентификации передаются в теле HTTP ответа. При запросах пользователя к защищенным ресурсам в заголовке Authorize HTTP запроса по следующей схеме. Первым в значении данного заголовка всегда стоит зарезервированное слово Bearer, потом через пробел записывается сам токен. В итоге получается: «Authorization: Bearer eyJhbGci...11y1537...yu5CSpyHI». Bearer в начале данного заголовка означает, что это носитель токена, и сервер должен дать доступ этому носителю данного токена.

Так как JWT токены имеют ограниченное время действия, то для механизма аутентификации нужен механизм обновления токенов.

Механизм обновления токенов состоит в получении пользователем двух токенов при аутентификации. Таким образом, JWT токены делятся на два типа:

1) Токены доступа (access token) – это токены, с помощью которых пользователь получает доступ к защищенным ресурсам. Данные токены короткоживущие и многоразовые.

2) Токены обновления (refresh token) – это токены, с помощью которых пользователь получает новую пару access token и refresh token, когда время действия access token подходит к концу. Данный токен долгоживущий, но одноразовый, так как при обновлении токена, обновляется и refresh token.

Основной сценарий обновления токенов заключается в следующем: при истечении срока действия access token или при приближении к этому времени, клиентская сторона системы, с которой работает пользователь, автоматически делает запрос к серверу с использованием refresh token в заголовке Authorize на специально выделенный ресурс, который выполняет обновление токена. После этого она получает оба токена и заменяет их в своем хранилище. Дальше система использует access token, пока схема не повторяется снова.

Для генерации JWT токенов в нашей системе мы будем использовать библиотеку для Java – Java JWT. Данная библиотека упрощает создание токенов благодаря удобному сборщику [37].

## 2.5.3. Инструментарий разработки frontend

Современные SPA веб-приложения создаются исключительно на языках JavaScript, HTML и CSS.

JavaScript (JS) — это высокоуровневый, портативный, интерпретируемый или JIT (Just-in-Time) компилируемый язык программирования с функциями как объектами первого класса. Хотя он наиболее известен как скриптовый язык для создания веб-страниц, многие не-браузерные среды исполнения также используют его, например Node.js, Apache CouchDB и Adobe Acrobat. JavaScript — основанный на прототипах, мультипарадигменный, однопоточный, динамический язык, поддерживающий объектно-ориентированный, императивный и декларативный (например, функциональное программирование) стили. JavaScript работает на клиентской стороне веб приложений, что позволяет использовать его для проектирования/программирования поведения веб-страниц [38]. JavaScript — это простой в освоении, а также мощный скриптовый язык, широко используемый для управления поведением веб-страницы.

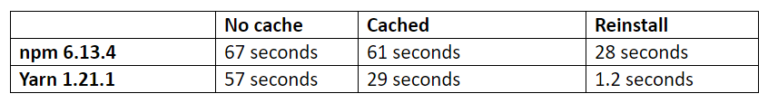
HyperText Markup Language (HTML) – это язык разметки для документов, предназначенных для отображения в веб-браузере. В табл. сокращений.

Cascading Style Sheets (CSS) – это язык таблицы стилей, используемый для описания представления документа, написанного на языке разметки, таком как HTML. В табл. сокращений.

Технологии, используемые для разработки SPA веб-приложений, работают на платформе Node.js. Node.js — это кроссплатформенная среда выполнения JavaScript с открытым исходным кодом, которая работает на движке V8 и выполняет код JavaScript вне веб-браузера [39]. Node.js позволяет разработчикам использовать JavaScript для написания инструментов командной строки и для программного кода на стороне сервера — запуск программного кода на стороне сервера для создания динамического содержимого веб-страницы до того, как страница будет отправлена в веб-браузер пользователя. Следовательно, Node.js представляет собой парадигму «JavaScript везде», объединяющую разработку веб-приложений вокруг одного языка программирования, а не разных языков для сценариев на стороне сервера и на стороне клиента [40]. Node.js имеет управляемую событиями архитектуру, поддерживающую асинхронный ввод-вывод. Это направлено на оптимизацию пропускной способности и масштабируемости в веб-приложениях с множеством операций ввода-вывода, а также в веб-приложениях реального времени (например, в программах связи в реальном времени и браузерных играх).

В качестве пакетного менеджера мы будем использовать Yarn. Yarn — это система упаковки программного обеспечения, разработанная Facebook в 2016 году для среды выполнения JavaScript Node.js [41]. Yarn, альтернатива менеджеру пакетов npm (Node Packet Manager – встроенный в Node.js пакетный менеджер), был создан в результате сотрудничества Facebook, Expo.dev, Google и Tilde для решения проблем согласованности, безопасности и производительности с большими кодовыми базами. Мы выбрали Yarn вместо npm по следующим критериям [42]:

* Производительность. Yarn работает быстрее npm в большинстве сценариев. Результаты тестов можно увидеть на рисунке 8.
* Кэширование пакетов. Yarn, в отличии от npm, кэширует пакеты, что позволяет мгновенно загружать пакеты.



ё

Лучше текстом в таблице.

SPA веб-приложения создаются с применением React или SPA фрэймворков, некоторые из которых используют React.

React – это библиотека c открытым программным кодом для JS, которая предназначена создания графического интерфейса веб станиц с помощью компонентов [43]. Данные компоненты представляют из себя независимые и пере-используемые части интерфейса, которые обычно написаны с помощью JSX [44]. JSX (JavaScript Syntax Extension) – это расширение для JavaScript, которое позволяет структурировать компоненты с использованием синтаксиса, привычного многим разработчикам [45]. JSX позволяет объединять в одном файле JS, HTML и CSS.

В нашем случае выбор пал на фрэймворк Next.js. Next.js — это фрэймворк для веб-разработки с открытым исходным кодом, созданный на основе Node.js и React и обеспечивающий такие функции веб-приложений, как SSR и создание статических веб-сайтов [46]. В документации React Next.js упоминается среди рекомендуемых инструментов для создания веб-сайта c SSR [47].

Next.js выделяется на фоне других технологий благодаря готовым и стандартизированным решениям, настройка которых в других фрэймворках может занимать много времени и быть проблематичной. Next.js предоставляет следующие готовые решения [48]:

1) Встроенный сборщик. В отличие от других фрэймворков, у Next.js есть встроенный сборщик, способный по умолчанию создавать тестовую и продуктовую сборку веб-приложения.

2) SSR. В Next.js есть готовые решения SSR, SSG (Static-site generation) и ISR (Incremental Static Regeneration).

3) Маршрутизация. В Next.js есть удобный и простой механизм маршрутизации между страницами, основанный на файловой системе, где каждый файл или директория в директории pages представляют из себя путь в веб приложении.

4) Встроенная поддержка CSS. Next.js позволяет импортировать CSS файлы из JS файлов.

5) Layouts. Next.js поддерживает концепцию макетов, позволяя создавать их как отдельные компоненты и пере-использовать на разных страницах.

6) Fast Refresh. Данных функционал позволяет увидеть результаты изменения программного кода при разработке, не теряя состояние страницы, то есть не теряя данные, которые там отображены.

7) API. Next.js является backend и frontend фрэймворком, поэтому поддерживает создание RESTful API, который можно использовать для получения данных и реализации механизмов информационной безопасности.

Для упрощения разработки определенных компонентов frontend составляющей нашей системы мы будем использовать следующие технологии:

* formik – это одна из самых популярных библиотек для React, которая предоставляет полный контроль над формами на веб-страницах. Данная библиотека позволяет определить полный жизненный цикл данных, который вводит пользователь в форму, до их отправки в backend [49]. Также formik предоставляет функционал валидации данных, вводимых пользователем.
* Yup – библиотека для построения схем данных. Данная библиотека предоставляет гибкий механизм парсинга и валидации данных. Yup часто используется с formik для создания надежного и легкого механизма валидации данных [50].
* axios – это веб HTTP-клиент, работающий в браузере пользователя. Данный клиент отличается от встроенного в JS клиента fetch большим функционалом и гибкостью [51].
* Next-Auth – это готовое решение с открытым программным кодом для аутентификации в Next.js. Данная библиотека позволяет использовать множество механизмов аутентификации и подключать существующие провайдеры аутентификации, такие как Auth0, Google, VK и десятки других. При использовании Next Auth, разработчик может получить информацию о сессии пользователя в клиентской и серверной части Next.js приложения [52].

Также для создания React компонентов мы будем использовать MUI. Material UI – это набор готовых React компонентов от разных полей до кнопок и всплывающих окон. Данная библиотека ляжет в основу некоторых компонентов, которые будут использоваться в интерфейсе нашей CRM-системы [53].

## 2.5.3. База данных

Перед выбором базы данных надо обозначить, что в микросерсвисной архитектуре можно и даже выгодно в некоторых сценариях использовать больше одной БД. Это объясняется тем, что для разных задач и сценариев всегда можно подобрать БД, которая будет наилучшим образом соответствовать их требованиям. Например, для хранения данных о пользователях, которые обычно строго соответствуют определенной структуре, лучше всего подходят реляционные БД, а для хранения документов или для аналитики – нереляционные.

Для реализации запланированного функционала мы будем использовать реляционную базу данных, так как данные, с которыми требуется работать, будут строго структурированы. При выборе реляционной базы данных сразу выделяются два претендента: MySQL и PostgreSQL [54]. Эти реляционные БД являются самыми мощными и популярными бесплатными решениями с открытым исходным кодом.

В таблице 1 представлено сравнение MySQL и PostgreSQL баз данных по основным характеристикам [55].

Таблица 1 – Сравнение MySQL и PostgreSQL

|  |  |  |
| --- | --- | --- |
|  | **PostgreSQL** | **MySQL** |
| Архитектура | Объектно-реляционная; мультипоточная | Реляционная; однопоточная |
| Поддерживаемые типы данных | Numeric, date/time, character, boolean, enumerated, geometric, network address, JSON, XML, HSTORE, arrays, ranges, composite | Numeric, date/time, character, spatial, JSON |
| Поддерживаемые индексы | B-tree, hash, GiST, SP-GiST, GIN и BRIN | Primarily B-tree; R-tree, hash и инвертированные индексы для определенных типов данных |
| Производительность | Подходит для систем с большим количеством запросов на чтение и запись | Подходит для систем с большим количеством запросов на чтение |
| Безопасность | Контроль доступа, несколько вариантов зашифрованного соединения | Контроль доступа, зашифрованные соединения |
| Поддержка | Поддержка сообщества. Компании, у которых есть собственная версия PostgreSQL, могут предлагать поддержку по ней | Поддержка сообщества, а также контракты на поддержку, предоставляемые поставщиком |

Для выбора подходящей базы данных мы будем использовать метод анализа иерархий (МАИ).

Первым делом мы построили матрицу попарных сравнений, которую можно увидеть в таблице 2, где сравним критерии баз данных по их предпочтительности, а также посчитаем сумму по столбцам.

Таблица 2 – Матрица попарных сравнений

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Архитектура | Поддерживаемые типы данных | Поддерживаемые индексы | Производительность | Безопасность | Поддержка |
| Архитектура | 1,00 | 1,00 | 1,00 | 0,20 | 0,20 | 0,50 |
| Поддерживаемые типы данных | 1,00 | 1,00 | 1,00 | 1,00 | 0,25 | 0,25 |
| Поддерживаемые индексы | 1,00 | 1,00 | 1,00 | 0,25 | 0,20 | 0,50 |
| Производительность | 5,00 | 1,00 | 4,00 | 1,00 | 0,17 | 4,00 |
| Безопасность | 5,00 | 4,00 | 5,00 | 6,00 | 1,00 | 6,00 |
| Поддержка | 2,00 | 3,00 | 2,00 | 0,25 | 0,17 | 1,00 |
| **СУММА** | **15,00** | **11,00** | **14,00** | **8,70** | **1,98** | **12,25** |

Далее мы построили матрицы попарных сравнений по каждому критерию, где определили, какая база данных предпочтительнее по определенному критерию, а также посчитали суммы по каждому столбцу. Результат можно увидеть в таблицах 3, 4, 5, 6, 7 и 8.

Таблица 3 – Матрица попарных сравнений по критерию архитектуры

|  |  |  |
| --- | --- | --- |
| Архитектура | MySQL | PostgreSQL |
| MySQL | 1,00 | 0,33 |
| PostgreSQL | 3,00 | 1,00 |
| **СУММА** | **4,00** | **1,33** |

Таблица 4 – Матрица попарных сравнений по критерию поддерживаемых типов данных

|  |  |  |
| --- | --- | --- |
| Поддерживаемые типы данных | MySQL | PostgreSQL |
| MySQL | 1,00 | 0,50 |
| PostgreSQL | 2,00 | 1,00 |
| **СУММА** | **3,00** | **1,50** |

Таблица 5 – Матрица попарных сравнений по критерию поддерживаемых индексов

|  |  |  |
| --- | --- | --- |
| Поддерживаемые индексы | MySQL | PostgreSQL |
| MySQL | 1,00 | 1,00 |
| PostgreSQL | 1,00 | 1,00 |
| **СУММА** | **2,00** | **2,00** |

Таблица 6 – Матрица попарных сравнений по критерию производительности

|  |  |  |
| --- | --- | --- |
| Производительность | MySQL | PostgreSQL |
| MySQL | 1,00 | 0,14 |
| PostgreSQL | 7,00 | 1,00 |
| **СУММА** | **8,00** | **1,14** |

Таблица 7 – Матрица попарных сравнений по критерию безопасности

|  |  |  |
| --- | --- | --- |
| Безопасность | MySQL | PostgreSQL |
| MySQL | 1,00 | 0,13 |
| PostgreSQL | 8,00 | 1,00 |
| **СУММА** | **9,00** | **1,13** |

Таблица 8 – Матрица попарных сравнений по критерию поддержки

|  |  |  |
| --- | --- | --- |
| Поддержка | MySQL | PostgreSQL |
| MySQL | 1,00 | 1,00 |
| PostgreSQL | 1,00 | 1,00 |
| **СУММА** | **2,00** | **2,00** |

После того, как мы сравнили все важные критерии баз данный, мы подсчитали веса всех альтернатив. Для этого поделили каждый элемент таблиц на сумму, соответствующую его столбцу, и посчитали среднее по всем строкам полученных таблиц. Результат можно увидеть в таблицах 9, 10, 11, 12, 13, 14 и 15.

Таблица 9 – Веса альтернатив

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Архитектура | Поддерживаемые типы данных | Поддерживаемые индексы | Производительность | Безопасность | Поддержка | **СРЗНАЧ** | **СРЗНАЧ в %** |
| Архитектура | 0,0667 | 0,0909 | 0,071 | 0,023 | 0,101 | 0,041 | **0,0656** | **6,6%** |
| Поддерживаемые типы данных | 0,0667 | 0,0909 | 0,071 | 0,1149 | 0,126 | 0,02 | **0,0817** | **8,2%** |
| Поддерживаемые индексы | 0,0667 | 0,0909 | 0,071 | 0,0287 | 0,101 | 0,041 | **0,0666** | **6,7%** |
| Производительность | 0,3333 | 0,0909 | 0,286 | 0,1149 | 0,084 | 0,327 | **0,2059** | **20,6%** |
| Безопасность | 0,3333 | 0,3636 | 0,357 | 0,6897 | 0,504 | 0,49 | **0,4563** | **45,6%** |
| Поддержка | 0,1333 | 0,2727 | 0,143 | 0,0287 | 0,084 | 0,082 | **0,1239** | **12,4%** |

Таблица 10 – Веса альтернатив по критерию архитектуры

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Архитектура | MySQL | PostgreSQL | **СРЗНАЧ** | **СРЗНАЧ в %** |
| MySQL | 0,25 | 0,25 | **0,25** | **25,0%** |
| PostgreSQL | 0,75 | 0,75 | **0,75** | **75,0%** |

Таблица 11 – Веса альтернатив по критерию поддерживаемых типов данных

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Поддерживаемые типы данных | MySQL | PostgreSQL | **СРЗНАЧ** | **СРЗНАЧ в %** |
| MySQL | 0,33 | 0,33 | **0,33** | **33,3%** |
| PostgreSQL | 0,67 | 0,67 | **0,67** | **66,7%** |

Таблица 12 – Веса альтернатив по критерию поддерживаемых индексов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Поддерживаемые индексы | MySQL | PostgreSQL | **СРЗНАЧ** | **СРЗНАЧ в %** |
| MySQL | 0,50 | 0,50 | **0,50** | **50,0%** |
| PostgreSQL | 0,50 | 0,50 | **0,50** | **50,0%** |

Таблица 13 – Веса альтернатив по критерию производительности

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Производительность | MySQL | PostgreSQL | **СРЗНАЧ** | **СРЗНАЧ в %** |
| MySQL | 0,13 | 0,13 | **0,13** | **12,5%** |
| PostgreSQL | 0,88 | 0,88 | **0,88** | **87,5%** |

Таблица 14 – Веса альтернатив по критерию безопасности

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Безопасность | MySQL | PostgreSQL | **СРЗНАЧ** | **СРЗНАЧ в %** |
| MySQL | 0,11 | 0,11 | **0,11** | **11,1%** |
| PostgreSQL | 0,89 | 0,89 | **0,89** | **88,9%** |

Таблица 15 – Веса альтернатив по критерию поддержки

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Поддержка | MySQL | PostgreSQL | **СРЗНАЧ** | **СРЗНАЧ в %** |
| MySQL | 0,50 | 0,50 | **0,50** | **50,0%** |
| PostgreSQL | 0,50 | 0,50 | **0,50** | **50,0%** |

Последним шагом идет определение оптимальной альтернативы. Для этого построили матрицу альтернатив, которую мы собрали из весов альтернатив всех критериев, и умножили эту матрицу на столбец весов альтернатив их общей таблицы критериев по правилу строка на столбец. Финальные значения находятся в таблице 16.

Таблица 16 – Финальные веса альтернатив по всем критериям

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Архитектура | Поддерживаемые типы данных | Поддерживаемые индексы | Производительность | Безопасность | Поддержка | **Веса альтернатив** |
| MySQL | 0,25 | 0,33 | 0,50 | 0,13 | 0,11 | 0,50 | 0,215311345 |
| PostgreSQL | 0,75 | 0,67 | 0,50 | 0,88 | 0,89 | 0,50 | 0,784688655 |

Таблицы надо по форме привести в порядок, чтоб красивые были. См. Стандарт. Если можно уменьшить шрифт внутри таблиц, это непременно надо сделать. **Я бы еще их сделала покомпактнее за счет уменьшения отступов и интервалов .**

В результате мы получили веса альтернатив с точки зрения достижения поставленной цели. Таким образом, в нашем случае PostgreSQL является наиболее предпочтительным вариантом.

## Среда разработки

В качестве среди разработки мы будем использовать IDE IntelliJ IDEA.

IDE (Integrated development environment) – это набор программных средств, используемых программистами для разработки программного обеспечения. Обычно IDE содержат в себе следующие компоненты:

* Специализированный текстовый редактор;
* Компилятор или интерпретатор для определенного языка программирования;
* Средство автоматизации сборки;
* Отладчик.

IntelliJ IDEA – это IDE, написанная на Java, используемая преимущественно для разработки ПО на Java [57].

Особенности IntelliJ IDEA [57]:

1) Поддержка Gradle, Spring, Hibernate, Lombok и других широко используемых технологий на Java.

2) Интегрированная поддержка инструмента для работы с БД.

3) Полноценная поддержка разработки веб приложений. IntelliJ IDEA поддерживает JavaScript, HTML, CSS, Node.js, Yarn, React и многие фрэймвоки для разработки веб приложений.

4) Расширение функционала. IntelliJ IDEA позволяет устанавливать множество плагинов для расширения функционала IDE.

6) Поддержка Git.

7) Скорость и простота использования.

Данная IDE позволит нам разрабатывать, запускать и отлаживать все компоненты системы из одного приложения.

Главы — с новой страницы, наверное, надо. См. стандарт.

## Реализация системы

В данной главе рассматривается ~~мы рассмотрим~~ процесс разработки каркаса и программного конструктора для создания CRM-систем, а также создание с его помощью готового решения такой системы. При обсуждении процесса разработки детально описана реализация ~~В процессе разработки мы углубимся в реализацию~~ выбранных нами шаблонов проектирования и механизмов информационной безопасности

## Техническое задание

Перед разработкой системы следует ~~мы должны~~ определиться с начальным функционалом и структурами данных. Мы разделим ~~будем разделять~~ систему на модули, каждый из которых будет отвечать за свою сущность данных.

В качестве примера мы решили реализовать базовый функционал CRM-системы для сферы автокредитования, который будет включать следующие модули:

1) Clients (Клиенты): данный модуль отвечает ~~будет отвечать~~ за основные данные о клиенте. Структура данных изображена в таблице 17.

Таблица 17 – Структура данных модуля Clients

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Название поля** | **Тип данных** | **Отношение** | **Тип поля в интерфейсе** | **Ограничения** | **Обязательность** |
| Id | Long | - | - |  | + |
| name | String | - | TextField | Минимальная длина 5 символов | + |
| dateOfBirth | Date | - | DateField |  | + |
| passportNumber | String | - | TextField | Длина строки 10 символов | + |
| nationality | String | - | ComboBoxField |  | + |

2) Loans (Кредиты): данный модуль отвечает ~~будет отвечать~~ за представление кредита.

Структура данных изображена в таблице 18.

Таблица 18 – Структура данных модуля Loans

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Название поля** | **Тип данных** | **Отношение** | **Тип поля в интерфейсе** | **Ограничения** | **Обязательность** |
| Id | Long | - | - |  | + |
| clientId | Long | Многие к одному к таблице Clients | - |  | + |
| creditNumber | String | - | TextField | Длина 5 символов | + |
| startDate | Date | - | DateField |  | + |
| totalSum | BigDecimal | - | MoneyField |  | + |
| carId | Long | Один к одному к таблице Cars | ComboBoxField |  | + |

3) Cars (Автомобили): данный модуль отвечает ~~будет отвечать~~ за автомобили клиента, информация о которых хранится ~~которые хранятся~~ в системе. Структура данных изображена в таблице 19.

Таблица 19 – Структура данных модуля Cars

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Название поля** | **Тип данных** | **Отношение** | **Тип поля в интерфейсе** | **Ограничения** | **Обязательность** |
| Id | Long | - | - |  | + |
| clientId | Long | Многие к одному к таблице Clients | - |  | + |
| carId | Long | - | - |  | + |
| brand | String | - | TextField | Минимальная длина 5 символов | + |
| model | Date | - | TextField |  | + |
| number | String | - | TextField | Длина строки 10 символов | + |
| price | BigDecimal | - | MoneyField |  | + |
| dateOfPurchase | Date | - | DateField |  |  |

~~Также~~ в каждой структуре данных будет два аудиторских поля: createdAt и updatedAt. Данные поля позволят отслеживать время создания и последнего изменения каждой из записей.

Финальную структуру таблиц можно увидеть на рисунке 9.

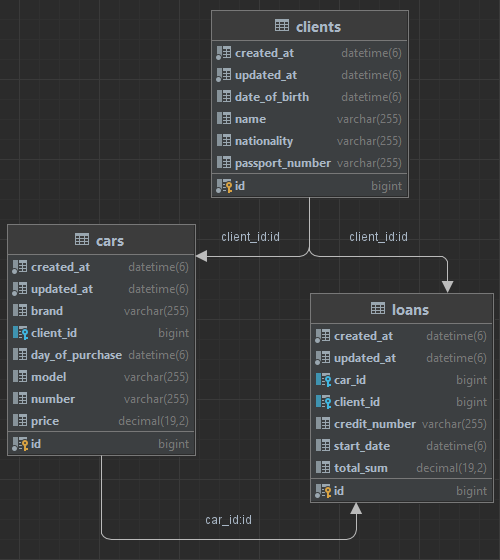


Рис. 9. Финальная структура базы данных.

Как было сказано в первой главе, основные задачи CRM-систем – это создание, хранение, изменение и просмотр данных о клиенте. Исходя из этого, каждый из вышеописанных модулей будет иметь следующий функционал:

1. Создание записи;
2. Просмотр записи;
3. Просмотр списка записей;
4. Изменение записи;
5. Удаление записи;
6. Валидация данных при создании и изменении записи.

## Разработка компонентов

Для реализации системы по вышеописанному техническому заданию первым делом нужно разработать каркас системы, а также компоненты, которые будут использоваться для разработки модулей.

## Разработка backend

Мы разделили реализацию backend компонента на основные составляющие шаблонов проектирования MVC и Repository Pattern, а также выделили систему, обеспечивающую информационную безопасность. В итоге получилось пять компонентов, из которых состоит backend: Model, View, Controller, Repository, Security.

В первую очередь следует реализовать компонент ~~Первым делом идет реализация~~ Model, так как он ~~данный компонент~~ отвечает за структуру данных, от которой будет зависеть реализация других компонентов. При использовании шаблона проектирования Repository Pattern, который **требует использование** требует использован**ие** или использован**ия**? Если первое, надо переформулировать, если второе, - поправить; вообще это предложение лучше разбить на составляющие, а то от двух «который» сложно уловить мысль ORM, данный компонент будет задавать как структуру данных, с которой будет работать программный код Java, так и структуру в таблицы в базе данных. В таком случае Model будет представлять из себя обычный Java класс с дополнительными аннотациями, которые будут задавать определенную конфигурацию и поведение записей в базе данных. Так как мы будем добавлять аудиторские поля, мы также будем наследовать каждый Model класс от абстрактного супер-класса, в котором объявлены нужные нам поля. Реализация данного класса представлена в листинге 1.

Листинг 1 – аудиторский супер-класс

@MappedSuperclass

@EntityListeners(AuditingEntityListener.class)

@Getter

@Setter

@JsonIgnoreProperties({"createdAt", "updatedAt"})

public abstract class AuditModel implements Serializable {

@Temporal(TemporalType.TIMESTAMP)

@Column(name = "created\_at", nullable = false, updatable = false)

@CreatedDate

private Date createdAt;

@Temporal(TemporalType.TIMESTAMP)

@Column(name = "updated\_at", nullable = false)

@LastModifiedDate

private Date updatedAt;

}

Реализации Model классов всех трех модулей нашей системы можно увидеть в листингах 2, 3 и 4.

Листинг 2 – Client Model

@Entity

@Table(name = "clients")

@Getter

@Setter

@NoArgsConstructor

public class Client extends AuditModel {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column

private String name;

@Temporal(TemporalType.TIMESTAMP)

@Column(name = "date\_of\_birth")

private Date dateOfBirth;

@Column(name = "passport\_number")

private String passportNumber;

private String nationality;

@Override

public String toString() {

return "Client{" +

"id=" + id +

", name='" + name + '\'' +

", dateOfBirth=" + dateOfBirth +

", passportNumber='" + passportNumber + '\'' +

", nationality=" + nationality +

'}';

}

}

Листинг 3 – Loan Model

@Entity

@Table(name = "loans")

@Getter

@Setter

@ToString

@NoArgsConstructor

public class Loan extends AuditModel {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToOne(fetch = FetchType.LAZY, optional = false)

@JoinColumn(name = "client\_id", nullable = false)

@OnDelete(action = OnDeleteAction.CASCADE)

@JsonIgnore

@ToString.Exclude

private Client client;

@Column(name = "client\_id", updatable = false, insertable = false)

private Long clientId;

@Column(name = "credit\_number")

private String creditNumber;

@Column(name = "start\_date")

@Temporal(TemporalType.TIMESTAMP)

private Date startDate;

@OneToOne(cascade = CascadeType.ALL)

@JoinColumn(name = "car\_id", referencedColumnName = "id")

@JsonIgnore

private Car car;

@Column(name = "car\_id", updatable = false, insertable = false)

private Long carId;

@Column(name = "total\_sum")

@Digits(integer = 10, fraction = 2)

private BigDecimal totalSum;

}

Листинг 4 – Car Model

@Entity

@Table(name = "cars")

@Getter

@Setter

@NoArgsConstructor

public class Car extends AuditModel {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToOne(fetch = FetchType.LAZY, optional = false)

@JoinColumn(name = "client\_id", nullable = false)

@OnDelete(action = OnDeleteAction.CASCADE)

@JsonIgnore

private Client client;

@Column(name = "client\_id", updatable = false, insertable = false)

private Long clientId;

@OneToOne(mappedBy = "car")

@JsonIgnore

private Loan loan;

private String brand;

private String model;

private String number;

@Temporal(TemporalType.TIMESTAMP)

@Column(name = "day\_of\_purchase")

private Date dateOfPurchase;

@Column(name = "price")

@Digits(integer = 10, fraction = 2)

private BigDecimal price;

}

Так как мы разрабатываем отдельный backend сервис, который не будет заниматься задачей отрисовки интерфейса, то компонент View будет представлен с помощью DTO. DTO (Data Transfer Object) – это экземпляр Model класса, с которым будет работать backend компонент системы.

Определившись со структурой данных, перейдем к разработке ~~следующим шагом идет разработка~~ репозиториев, которые необходимы для реализации Repository Pattern.

Репозиторий – это интерфейс, методы которого реализуются двумя способами: автоматический посредством механизма рефлексии и ручной, в котором разработчик сам задает SQL запрос и параметры. Мы использовали оба способа. Простые методы создания, получения, изменения и удаления записей мы реализовали первым способом, а метод поиска разработали вручную. Также благодаря использованию ORM, нам не нужно реализовывать маппинг возвращаемых БД данных.

Реализация репозиториев представлена в листингах 5, 6 и 7.

Листинг 5 – Client Repository

@Repository

public interface ClientRepo extends JpaRepository<Client, Long> {

@Query("select c from Client c where (:name is null or c.name like concat('%', :name, '%')) and (:passportNumber is null or c.passportNumber like concat('%', :passportNumber, '%')) and (:nationality is null or c.nationality = :nationality)")

List<Client> find(

@Param("name") String name,

@Param("passportNumber") String passportNumber,

@Param("nationality") String nationality

);

}

Листинг 6 – Loan Repository

@Repository

public interface LoanRepo extends JpaRepository<Loan, Long> {

@Query("select l from Loan l where (:creditNumber is null or l.creditNumber like concat('%', :creditNumber, '%')) and (:carId is null or l.carId = :carId) and :clientId = l.clientId")

List<Loan> find(@Param("creditNumber") String creditNumber, @Param("carId") Long carId, @Param("clientId") Long clientId);

}

Листинг 7 – Car Repository

public interface CarRepo extends JpaRepository<Car, Long> {

@Query("select c from Car where (:brand is null or c.brand like concat('%', :brand, '%')) and (:model is null or c.model like concat('%', :model, '%')) and (:number is null or c.number like concat('%', :number, '%')) and :clientId = c.clientId")

List<Car> find(

@Param("brand") String brand,

@Param("model") String model,

@Param("number") String number,

@Param("clientId") Long clientId

);

@Query(select c from Car c left join Loan l on c.id = l.carId where l.carId is null and c.clientId = :clientId or c.id = :carId")

List<Car> findUnused(@Param("clientId") Long clientId, @Param("carId") Long carId);

Optional<List<Car>> findByClientId(Long clientId);

}

Следующий компонент системы – это Controller, компонент, позволяющий другим системам взаимодействовать с текущей. С помощью контроллера создаются endpoints – точки канала связи. Endpoint в RESTful API – это какой-либо конкретный URL, позволяющий взаимодействовать с ресурсом посредством протокола HTTP.

Для создания трех основных модулей мы разработали RESTful контроллеры под каждый из них. RESTful Контроллеры в Spring представляют из себя класс с аннотацией @RestController (пример контроллера для модуля Clients в листинге 8). Данный класс содержит как простые утилитарные методы, так и методы, помеченные аннотациями @GetMapping, @PostMapping, @PutMapping, @DeleteMapping. Данные аннотации определяют, что метод – это endpoint, а также показывают, какой HTTP запрос он будет обрабатывать.

В каждом контроллере мы реализовали следующие основные endpoint`ы:

* find: возвращает список найденных записей на GET запросу;
* getById: возвращает запись, найденную по id, на GET запросу;
* create: получает запись, сохраняет ее в систему и возвращает ее с параметром id на POST запросу;
* update: получает запись с измененными данными, сохраняет ее в системе и возвращает ее на PUT запросу;
* delete: удаляет запись по полученному id на DELETE запросу.

Стоит заметить, что мы нигде не создаем экземпляр репозитория модели Clients, так как он создается самим фрэймворком Spring посредством механизма Dependency Injection. Также Dependency Injection упрощается посредством библиотеки Lombok: в данном случае аннотация @RequiredArgsConstructor при сборке проекта указывает компилятору, что следует создать конструктор для всех полей (в данном случае оно одно) с аннотацией @Autowired, благодаря которой Spring при запуске приложения создаст и внедрит экземпляр класса в поле.

Листинг 8 – Clients Controller

@RestController

@RequestMapping("clients")

@CrossOrigin(origins = "\*", allowedHeaders = "\*")

@RequiredArgsConstructor

public class ClientController {

private final ClientRepo clientRepo;

@GetMapping

public List<Client> find(Client client) {

return clientRepo.find(

client.getName(),

client.getPassportNumber(),

client.getNationality()

);

}

@GetMapping("{id}")

public Optional<Client> getById(@PathVariable("id") Long id) {

return Optional.ofNullable(clientRepo.findById(id)

.orElseThrow(() -> new NotFoundException("ClientId: " + id + " not found")));

}

@PostMapping

public Client create(@Valid @RequestBody Client client) {

return clientRepo.save(client);

}

@PutMapping("{id}")

public Client update(

@PathVariable Long id,

@Valid @RequestBody Client clientRequest

) {

return clientRepo.findById(id).map(client -> {

client.setName(clientRequest.getName());

client.setDateOfBirth(clientRequest.getDateOfBirth());

client.setPassportNumber(clientRequest.getPassportNumber());

client.setNationality(clientRequest.getNationality());

return clientRepo.save(client);

}).orElseThrow(() -> new NotFoundException("ClientId: " + id + " not found"));

}

@DeleteMapping("{id}")

public ResponseEntity<?> delete(@PathVariable Long id) {

return clientRepo.findById(id).map(client -> {

clientRepo.delete(client);

return ResponseEntity.ok().build();

}).orElseThrow(() -> new NotFoundException("ClientId: " + id + " not found"));

}

}

Последним и самым комплексным компонентом backend является Security, который реализован с помощью Spring Security. Данный компонент содержит реализацию следующих механизмов:

1. Идентификация;
2. Аутентификация;
3. Авторизация;
4. CORS.

Для реализации первых трех механизмов мы создали следующие компоненты:

1) Модели AppUser и Role (листинги 8 и 9), которые определяют структуру данных для хранения пользователей и ролей;

2) Репозитории AppUserRepo и RoleRepo (листинги 10 и 11) для хранения, получения и создания в БД сущностей AppUser и Role;

3) Контроллер AppUserController (листинг 12) для создания пользователей и ролей, добавления пользователю ролей и обновления токенов;

4) Сервис AppUserServiceImpl (листинг 13), который используется контроллером для выполнения его функций.

Листинг 8 – AppUser Model

@Entity

@Data

public class AppUser {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String username;

private String password;

@ManyToMany(fetch = FetchType.EAGER)

private Collection<Role> roles = new ArrayList<>();

}

Листинг 9 – Role Model

@Entity

@Data

public class Role {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

}

Листинг 10 – AppUserRepo Repository

@Repository

public interface AppUserRepo extends JpaRepository<AppUser, Long> {

AppUser findByUsername(String username);

}

Листинг 11 – RoleRepo Repository

@Repository

public interface RoleRepo extends JpaRepository<Role, Long> {

Role findByName(String name);

}

Листинг 12 – AppUserController Controller

@RestController

@RequestMapping("/auth")

@CrossOrigin(origins = "\*", allowedHeaders = "\*")

@RequiredArgsConstructor

@Slf4j

public class AppUserController {

private final AppUserService appUserService;

@GetMapping("/users")

public ResponseEntity<List<AppUser>> getAllUsers() {

return ResponseEntity.ok().body(appUserService.getAppUsers());

}

@PostMapping("/users")

public ResponseEntity<AppUser> createUser(@RequestBody AppUser appUser) {

URI uri = URI.create(ServletUriComponentsBuilder.fromCurrentContextPath().path("/api/users").toUriString());

return ResponseEntity.created(uri).body(appUserService.saveAppUser(appUser));

}

@PostMapping("/roles")

public ResponseEntity<Role> createRole(@RequestBody Role role) {

URI uri = URI.create(ServletUriComponentsBuilder.fromCurrentContextPath().path("/api/roles").toUriString());

return ResponseEntity.created(uri).body(appUserService.saveRole(role));

}

@PostMapping("/users/addrole")

public ResponseEntity<?> addRoleToUser(

@RequestBody RoleToUserForm form

) {

appUserService.addRoleToAppUser(

form.getUsername(), form.getRoleName()

);

return ResponseEntity.ok().build();

}

@GetMapping("/token/refresh")

public void refreshToken(

HttpServletRequest request,

HttpServletResponse response

) throws IOException {

String authorizationHeader = request.getHeader(AUTHORIZATION);

if (authorizationHeader != null && authorizationHeader.startsWith(TOKEN\_PREFIX)) {

try {

String refreshToken =

authorizationHeader.substring(TOKEN\_PREFIX.length());

Algorithm algorithm = AlgorithmConstructor.getAlgorithm();

JWTVerifier verifier = JWT.require(algorithm).build();

DecodedJWT decodedJWT = verifier.verify(refreshToken);

String username = decodedJWT.getSubject();

AppUser appUser = appUserService.getAppUser(username);

Date expiresIn = new Date(

System.currentTimeMillis() + 30 \* 60 \* 1000

);

List<String> roles = appUser.getRoles().stream()

.map(Role::getName).collect(Collectors.toList());

String accessToken = JWT.create()

.withSubject(appUser.getUsername())

.withIssuedAt(

new Date(System.currentTimeMillis())

)

.withExpiresAt(expiresIn)

.withIssuer(request.getRequestURI())

.withClaim("user", appUser.getUsername())

.withClaim("roles", roles)

.sign(algorithm);

writeSuccessfulAuthResponse(

response,

expiresIn,

accessToken,

refreshToken,

appUser.getUsername(),

roles

);

log.info(

"Successful token refresh for User: {}",

appUser.getUsername()

);

} catch (Exception e) {

log.error("Error logging in: {}", e.getMessage());

writeUnsuccessfulAuthResponse(response, e);

}

} else {

throw new RuntimeException("Refresh token is missing");

}

}

}

@Data

class RoleToUserForm {

private String username;

private String roleName;

}

Листинг 13 – AppUserServiceImpl Service

@Service

@RequiredArgsConstructor

@Transactional

@Slf4j

public class AppUserServiceImpl implements AppUserService, UserDetailsService {

private final AppUserRepo appUserRepo;

private final RoleRepo roleRepo;

private final PasswordEncoder passwordEncoder;

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

AppUser appUser = appUserRepo.findByUsername(username);

if (appUser == null) {

log.error("User: {} not found in DB", username);

throw new UsernameNotFoundException("User: " + username + " not found in DB");

} else {

log.info("User: {} found in DB", username);

}

Collection<SimpleGrantedAuthority> authorities = new ArrayList<>();

appUser.getRoles().forEach(role -> {

authorities.add(new SimpleGrantedAuthority(role.getName()));

});

return new User(appUser.getUsername(), appUser.getPassword(), authorities);

}

@Override

public AppUser saveAppUser(AppUser appUser) {

log.info("Saving new User: {} to DB", appUser.getUsername());

appUser.setPassword(passwordEncoder.encode(appUser.getPassword()));

return appUserRepo.save(appUser);

}

@Override

public Role saveRole(Role role) {

log.info("Saving new Role: {} to DB", role.getName());

return roleRepo.save(role);

}

@Override

public void addRoleToAppUser(String username, String roleName) {

log.info("Adding Role: {} to User: {}", roleName, username);

AppUser appUser = appUserRepo.findByUsername(username);

Role role = roleRepo.findByName(roleName);

appUser.getRoles().add(role);

}

@Override

public AppUser getAppUser(String username) {

log.info("Fetching User: {} from DB", username);

return appUserRepo.findByUsername(username);

}

@Override

public List<AppUser> getAppUsers() {

log.info("Fetching all users from DB");

return appUserRepo.findAll();

}

}

Создав все необходимые компоненты, мы реализовали механизм аутентификации. В Spring Security логика аутентификации реализуется посредством фильтра аутентификации. Для этого мы создали класс CustomAuthenticationFilter, который наследуется от UsernamePasswordAuthenticationFilter (это предопределенный ~~реализованный~~ в Spring Security класс) и перегрузили следующие три метода:

1) attemptAuthentication (листинг 14). Данный метод инициирует процесс аутентификации с помощью AuthenticationManager и проверяет подлинность введенных пользователем данных.

2) successfulAuthentication (листинг 15). Этот метод вызывается при успешной аутентификации. Его задача – это генерация JWT токенов (токена доступа и токена обновления) и создание HTTP-ответа который будет содержать токены и роли пользователя.

3) unsuccessfulAuthentication (листинг 16). Данный метод вызывается, когда пользователь ввел неверные данные. Его задача – создание HTTP ответа, который содержит сообщение об ошибке.

Упомянутые листинги 14-16 представлены ниже.

Листинг 14 – attemptAuthentication

public Authentication attemptAuthentication(

HttpServletRequest request,

HttpServletResponse response

) throws AuthenticationException {

String username = request.getParameter("username");

String password = request.getParameter("password");

UsernamePasswordAuthenticationToken authenticationToken =

new UsernamePasswordAuthenticationToken(username, password);

return authenticationManager.authenticate(authenticationToken);

}

Листинг 15 – successfulAuthentication

protected void successfulAuthentication(

HttpServletRequest request,

HttpServletResponse response,

FilterChain chain,

Authentication authentication

) throws IOException {

User user = (User) authentication.getPrincipal();

Algorithm algorithm = AlgorithmConstructor.getAlgorithm();

Date expiresIn =

new Date(System.currentTimeMillis() + 30 \* 60 \* 1000);

List<String> roles = user.getAuthorities().stream().map(

GrantedAuthority::getAuthority

).collect(Collectors.toList());

String accessToken = JWT.create()

.withSubject(user.getUsername())

.withIssuedAt(new Date(System.currentTimeMillis()))

.withExpiresAt(expiresIn)

.withIssuer(request.getRequestURI())

.withClaim("user", user.getUsername())

.withClaim("roles", roles)

.sign(algorithm);

String refreshToken = JWT.create()

.withSubject(user.getUsername())

.withIssuedAt(new Date(System.currentTimeMillis()))

.withExpiresAt(

new Date(System.currentTimeMillis() + 120 \* 60 \* 1000))

.withIssuer(request.getRequestURI())

.withClaim("user", user.getUsername())

.withClaim("roles", roles)

.sign(algorithm);

writeSuccessfulAuthResponse(

response,

expiresIn,

accessToken,

refreshToken,

user.getUsername(),

roles

);

}

Листинг 16 – unsuccessfulAuthentication

protected void unsuccessfulAuthentication(

HttpServletRequest request,

HttpServletResponse response,

AuthenticationException failed

) throws IOException {

String username = request.getParameter("username");

log.info("Failed to authenticate User: {}", username);

Map<String, String> error = new HashMap<>();

error.put("error", "failed to authenticate user: " + username);

error.put("message", "invalid username and password combination");

response.setContentType(APPLICATION\_JSON\_VALUE);

response.setStatus(SC\_FORBIDDEN);

new ObjectMapper().writeValue(response.getOutputStream(), error);

}

Мы оставили стандартный в Spring Security endpoint для аутентификации: /login.

Финальная схема аутентификации представлена на рисунке 10.

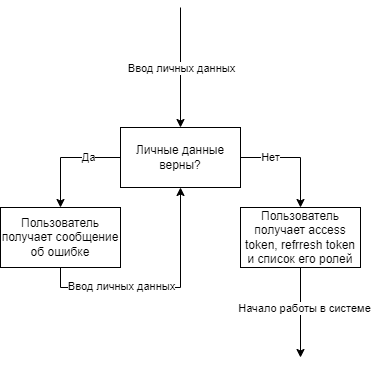


Рис. 10. Финальная схема аутентификации.

Авторизация также требует создания фильтра. Для этого Мы создали класс CustomAuthorizationFilter, унаследовав его ~~который унаследовали~~ от класса OncePerRequestFilter, логика которого срабатывает при каждом запросе пользователя. В нашем классе мы перегрузили метод doFilterInternal (листинг 17).

Листинг 17 – doFilterInternal

protected void doFilterInternal(

HttpServletRequest request,

HttpServletResponse response,

FilterChain filterChain

) throws ServletException, IOException {

if (

request.getServletPath().equals("/login") || request.getServletPath().equals("/auth/token/refresh/\*\*")

) {

filterChain.doFilter(request, response);

} else {

String authorizationHeader = request.getHeader(AUTHORIZATION);

if (

authorizationHeader != null && authorizationHeader.startsWith(TOKEN\_PREFIX)

) {

try {

String token =

authorizationHeader.substring(TOKEN\_PREFIX.length());

Algorithm algorithm = AlgorithmConstructor.getAlgorithm();

JWTVerifier verifier = JWT.require(algorithm).build();

DecodedJWT decodedJWT = verifier.verify(token);

String username = decodedJWT.getSubject();

String[] roles =

decodedJWT.getClaim("roles").asArray(String.class);

Collection<SimpleGrantedAuthority> authorities =

new ArrayList<>();

stream(roles).forEach(

role -> authorities.add(

new SimpleGrantedAuthority(role)

)

);

UsernamePasswordAuthenticationToken authenticationToken =

new UsernamePasswordAuthenticationToken(

username,

null,

authorities

);

SecurityContextHolder.getContext()

.setAuthentication(authenticationToken);

filterChain.doFilter(request, response);

} catch (Exception e) {

log.error(

"Error logging in: {}, token: {}",

e.getMessage(),

authorizationHeader

);

writeUnsuccessfulAuthResponse(response, e);

}

} else {

filterChain.doFilter(request, response);

}

}

}

Финальная схема авторизации представлена на рисунке 11.

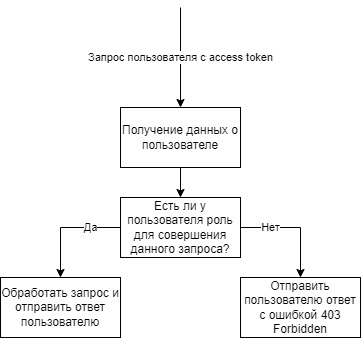


Рис. 11. Финальная схема авторизации.

После реализации основных механизмов информационной безопасности, мы подключили их к конфигурационному классу Spring Security. Для этого мы создали класс SecurityConfig, унаследовав его ~~который унаследовали~~ от класса WebSecurityConfigurerAdapter, и указали аннотации @Configuration и @EnableWebSecurity, которые указывают фрэймворку Spring, что это конфигурационный класс и класс, отвечающий за веб безопасность. В данном классе мы перегрузили следующие методы:

1) configure. Данный метод мы перегрузили дважды с разными сигнатурами. Первая для конфигурации кодировщика паролей пользователей (листинг 18). В качестве кодировщика мы использовали класс BCryptPasswordEncoder, который хэширует пароль со случайно сгенерированной солью. Данный класса рекомендуется разработчиками Spring Security и является более безопасным, чем MD5 и SHA.

Листинг 18 – конфигурация кодировщика паролей

private final UserDetailsService userDetailsService;

private final PasswordEncoder passwordEncoder;

@Override

protected void configure(AuthenticationManagerBuilder auth)

throws Exception {

auth.userDetailsService(userDetailsService)

.passwordEncoder(passwordEncoder);

}

2) Вторая сигнатура создана для общей конфигурации безопасности (листинг 19). В данной сигнатуре мы включили CORS, настроили авторизацию и подключили фильтры аутентификации и авторизации.

Листинг 19 – конфигурация безопасности

protected void configure(HttpSecurity http) throws Exception {

http.cors();

http.csrf().disable();

http.sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.STATELESS);

http.authorizeRequests().antMatchers(

"/login/\*\*",

"/token/refresh/\*\*"

).permitAll();

http.authorizeRequests().antMatchers(

HttpMethod.GET,

"/auth/\*\*"

).hasAnyAuthority("ROLE\_ADMIN");

http.authorizeRequests().antMatchers(

HttpMethod.POST,

"/auth/\*\*"

).hasAnyAuthority("ROLE\_ADMIN");

http.authorizeRequests().antMatchers(

HttpMethod.PUT,

"/auth/\*\*"

).hasAnyAuthority("ROLE\_ADMIN");

http.authorizeRequests().antMatchers(

HttpMethod.DELETE,

"/auth/\*\*"

).hasAnyAuthority("ROLE\_ADMIN");

http.authorizeRequests().anyRequest().authenticated();

http.addFilter(

new CustomAuthenticationFilter(authenticationManagerBean())

);

http.addFilterBefore(

new CustomAuthorizationFilter(),

UsernamePasswordAuthenticationFilter.class

);

}

Также мы создали метод corsConfigurationSource (листинг 20), в котором настроили CORS. В данном методе мы указали, какие HTTP-запросы следует серверу принимать, а какие отклонять с ошибкой 403 Forbidden. В настройках CORS мы указали следующие параметры:

1. Разрешенные хосты. В данном параметре мы указали адрес frontend сервиса.
2. Разрешенные заголовки запросов. В данном параметре мы указали допустимые HTTP заголовки, с которыми внешний сервис может делать запросы к backend.
3. Разрешенные заголовки ответов. В данном параметре мы указали допустимые HTTP заголовки, с которыми backend может отвечать на запросы внешних сервисов.
4. Разрешенные методы. В данном параметре мы указали разрешенные методы HTTP запросов, которые могут отправлять внешние сервисы.
5. Маппинг настроек. Последним параметром было указанно, к какому URL применять вышеописанные настройки CORS. Мы указали “/\*\*”, чтобы применить ко всем endpoint`ам.

Листинг 20 – настройка CORS

public CorsConfigurationSource corsConfigurationSource() {

CorsConfiguration corsConfiguration = new CorsConfiguration();

corsConfiguration.setAllowCredentials(true);

corsConfiguration.setAllowedOrigins(

Collections.singletonList("http://localhost:3000")

);

corsConfiguration.setAllowedHeaders(Arrays.asList(

"Origin",

"Access-Control-Allow-Origin",

"Content-Type",

"Accept",

"Authorization",

"Origin, Accept",

"X-Requested-With",

"Access-Control-Request-Method",

"Access-Control-Request-Headers"

));

corsConfiguration.setExposedHeaders(Arrays.asList(

"Origin",

"Content-Type",

"Accept",

"Authorization",

"Access-Control-Allow-Origin",

"Access-Control-Allow-Credentials"

));

corsConfiguration.setAllowedMethods(Arrays.asList(

"GET", "POST", "PUT", "DELETE", "OPTIONS")

);

UrlBasedCorsConfigurationSource urlBasedCorsConfigurationSource =

new UrlBasedCorsConfigurationSource();

urlBasedCorsConfigurationSource.registerCorsConfiguration(

"/\*\*", corsConfiguration);

final UrlBasedCorsConfigurationSource source =

new UrlBasedCorsConfigurationSource();

source.registerCorsConfiguration("/\*\*", corsConfiguration);

return source;

}

Данный подход к разработке упростит и ускорит дальнейшую разработку backend компонента по следующим причинам:

1. Использование RESTful API упростит интеграцию новых сервисов и систем;
2. Реализация Repository Pattern позволит разработчикам не тратить время на реализацию промежуточного слоя между БД и backend, а также обеспечит получение полной доменной модели данных;
3. Использование Spring Security предоставляет возможность тонкой настройки механизмов информационной безопасности, при этом реализация данных механизмов никак не влияет на остальные компоненты backend;
4. Микросервисная архитектура позволит масштабировать и балансировать систему автоматически в зависимости от ее нагрузки.

## Разработка frontend

Разработку frontend мы разделили на несколько стадий:

1. Разработка React компонентов для интерфейса;
2. Имплементация механизмов информационной безопасности;
3. Создание форм для модулей.

Приведи к единому стилю межстрочные интервалы (по стандарту).

Для создания программного конструктора мы разработали набор React компонентов, которые будут использованы в формах модулей. Список основных компонентов включает в себя:

1. Кнопки;
2. Поля;
3. Компоненты формы;
4. Компоненты таблицы;
5. Всплывающие окна;
6. Панель инструментов;
7. Панель навигации по системе;
8. Панель навигации по модулям.

React компоненты бывают двух типов: классовые и функциональные. Для создания классовых типов создается JS класс c методом render, который занимается отрисовкой компонента. Функциональные компоненты представляют из себя функцию, которая возвращает сам компонент. В классовых компонентах их жизненный цикл управляется с помощью методов класса, ~~когда~~ а в функциональных – хуками. Никакой принципиальной разницы в этих методах создания компонентов нет, но функциональный метод требует меньше кода и выглядит более читабельным ~~разработчику~~. Поэтому мы создавали функциональные компоненты.

В качестве примера разработки компонентов, мы рассмотрим ~~возьмем~~ поле ComboBox. ComboBox – это традиционно поле, которое состоит из строки ввода и всплывающего списка значений. Данное поле используется для выбора значения из заранее заданного списка. В нашем случае это поле также имеет функцию поиска, где пользователь вводит первые символы значения и список обновляется, показывая схожие значения. Нашу реализацию React компонента ComboBox можно увидеть в листинге 21, а пример в отрисованном интерфейсе на рисунке 12. В качестве основы ~~За основу~~ мы использовали MUI компонент Autocomplete, который является основой для полей, подобных ComboBox. К данному компоненту мы добавили следующие элементы:

1. Ярлык, который используется для задания названия поля;
2. Плейсхолдер, который отмечается, когда в поле не выбрано значение;
3. Сообщение об ошибке, которое выводится при не пройденной валидации.

Данное поле получает два основных параметра: список значений и конкретное значение. Значение представляет из себя объект с двумя полями: key и label. Поиск значений происходит по полю label, а система сохраняет в записи значение поля key. Если в поле подается и список значений, и само значение, то при отрисовке интерфейса поле уже будет иметь выбранное значение и будет отображаться label.

Также у компонента есть следующие параметры:

1. Ширина самого поля;
2. Ширина заголовка;
3. Флаг отключения поля чтобы пользователь не мог с ним взаимодействовать;
4. Поля для интеграции с менеджером форм formik.

Для стилизации данного поля мы использовали модульный CSS, который импортировали в JS файл. Также нам пришлось перегрузить стандарные стили компонента Autocomplete. Для этого мы создали объект sx и передали его в компонент Autocomplete. Он перегружает встроенные CSS класса данного компонента на inherit, что заставляет их наследовать стили, которые мы задали компоненту Combobox.

Листинг 20 – ComboBox компонент

const sx = {

"& .MuiOutlinedInput-root": {

color: "inherit",

fontFamily: "inherit",

"& fieldset": {

border: "inherit",

},

"&.Mui-focused fieldset": {

border: "inherit",

},

"&.MuiOutlinedInput-root": {

border: "inherit",

color: "inherit",

},

},

"& .MuiInputBase-input": {

color: "inherit",

fontFamily: "inherit",

"&.Mui-disabled": {

color: "inherit",

fontFamily: "inherit",

WebkitTextFillColor: "inherit",

}

},

}

const ComboBoxField = ({

label,

name,

placeholder,

value,

handleChange,

handleBlur,

error,

options,

width = 20,

disabled = false,

onChangeEvent,

labelWidth = `15rem`

}) => {

const [initOption, setInitOption] = useState({key: ``, label: ``})

useEffect(() => {

const getInitOption = () => {

const result = options.find(option => option.key === value)

setInitOption(result ? result : {key: ``, label: ``})

}

if (options) {

getInitOption()

}

}, [options, value])

return (

<div className={classes.comboBoxField}>

<div

className={classes.label}

style={{width: labelWidth}}>

{label}

</div>

<div

style={{width: width + `rem`}}

className={classes.fieldWrapper}

>

<Autocomplete

value={initOption}

options={options}

getOptionLabel={(option) => option.label ?? option}

name={name}

isOptionEqualToValue={(option, value) =>

option.key === value.key

}

onChange={(\_, newValue) => {

handleChange(name, newValue ? newValue.key : ``)

if (onChangeEvent) {

onChangeEvent(newValue ? newValue : ``)

}

}}

onBlur={handleBlur}

sx={{width: width + `rem`}}

disablePortal

autoComplete

autoHighlight

disabled={disabled}

renderInput={(params) =>

<TextField

{...params}

placeholder={disabled ? `` : placeholder}

className={classes.comboBoxInput}

sx={sx}

/>

}

/>

</div>

{error && (

<div className={classes.error}>

{error}

</div>

)}

</div>

)

}

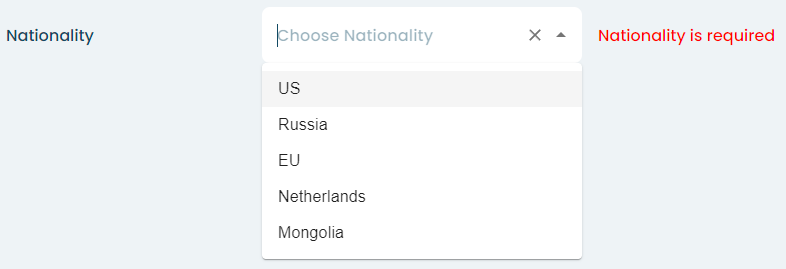


Рис. 12. Отрисованный ComboBox с ярлыком, списком значений и сообщением об ошибке.

После разработки всех необходимых компонентов, мы имплементировали аутентификацию и авторизацию. Чтобы имплементировать данные механизмы, мы ~~сделали следующее:~~ создали компонент RouteGuard, переопределили компонент App, настроили серверную сторону Next.js для использования Next Auth. Рассмотрим каждый из этих этапов.

Создание компонента RouteGuard (листинг 21).

Данный компонент отвечает за слежение за состоянием сессии пользователя. При каждом переходе со страницы на страницу приложения данный компонент проверяет наличие сессии через метод getSession и настраивает HTTP-клиент axios, передавая в заголовок Authorization токен доступа, полученный из сессии. Если у пользователя нет установленной сессии, то он перенаправляет его на страницу аутентификации. Логическую схему данного компонента можно увидеть на рисунке ннн.

Листинг 21 – компонент RouteGuard

const RouteGuard = ({children}) => {

const router = useRouter()

useEffect(() => {

getSession().then(async session => {

if (!session) {

axios.defaults.headers.common['Authorization'] = ``

if (

router.pathname !== `/signin` &&

router.pathname !== `/autherror`

) {

signIn()

}

} else if (

router.pathname !== `/signin` &&

session.error === `RefreshAccessTokenError`

) {

axios.defaults.headers.common['Authorization'] = ``

signIn()

} else {

axios.defaults.headers.common['Authorization'] =

`Bearer ${session.accessToken}`

}

})

})

return (children);

}

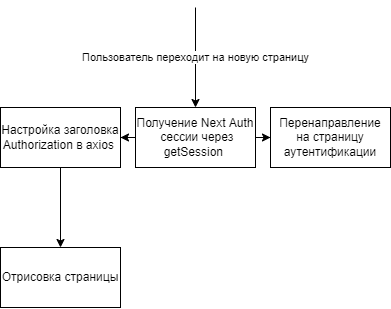


Рис. 12. Отрисованный ComboBox с ярлыком, списком значений и сообщением об ошибке.

Переопределение компонента App (листинг 22).

Мы переопределили компонент App, который используется Next.js для отрисовки всех страниц в приложении. Данный компонент мы обернули в два слоя. Первый – это RouteGuard. Второй – это компонент SessionProvider, который идет с Next Auth. Данный подход позволяет получить полный контроль над сессией пользователя на всех страницах приложения.

Листинг 22 – переопределенный компонент App

const App = ({

Component,

pageProps: {session, ...pageProps}

}) => {

return (

<SessionProvider session={session} refetchInterval={5 \* 60}>

<RouteGuard>

<div id={`rootElement`}>

<Component {...pageProps} />

</div>

</RouteGuard>

</SessionProvider>

)

}

Настройка серверной стороны Next.js для использования Next Auth (листинг 23).

В данной конфигурации мы настроили создание сессии. Для этого на серверной стороне Next.js мы указали способ аутентификации пользователей через комбинацию логина и пароля и реализовали логику обращения к backend для получения токенов. Также мы реализовали алгоритм обновления токенов, если срок годности токена доступа заканчивается, чтобы пользователю не приходилось аутентифицироваться каждый раз, когда токен заканчивается.

Листинг 23 – настройка Next Auth

async function refreshAccessToken(token) {

try {

const response = await

axios.get(`${process.env.SERVER\_PATH}/auth/token/refresh`, {

headers: {

"Authorization": `Bearer ${token.refreshToken}`

}

}).catch((error) => {

throw error

})

const refreshedTokens = response.data

return {

accessToken: refreshedTokens.accessToken,

refreshToken: refreshedTokens.refreshToken,

expiresIn: refreshedTokens.expiresIn,

error: null

}

} catch (error) {

return {

...token,

error: "RefreshAccessTokenError",

}

}

}

const options = {

providers: [

CredentialsProvider({

id: `credentials`,

name: `Credentials`,

credentials: {

username: {label: `Username`, type: `text`, placeholder: `username`},

password: {label: `Password`, type: `password`}

},

async authorize(credentials) {

const params = new URLSearchParams({

username: credentials.username,

password: credentials.password,

})

const response = await axios.post(`${process.env.SERVER\_PATH}/login`, params)

if (response) {

return response.data;

} else {

return null

}

}

}),

],

secret: process.env.SECRET,

pages: {

signIn: '/signin',

},

session: {

strategy: `jwt`,

},

callbacks: {

async jwt({token, account, user}) {

if (account) {

token.accessToken = user.accessToken

token.refreshToken = user.refreshToken

token.expiresIn = user.expiresIn

token.roles = user.roles

}

if (Date.now() >= token.expiresIn - 60 \* 10) { // refresh token if it`s lifetime is less than 10min

return refreshAccessToken(token);

}

return token

},

async session({session, token}) {

session.user = token.user

session.accessToken = token.accessToken

session.expires = token.expiresIn

session.error = token.error

session.roles = token.roles

axios.defaults.headers.common['Authorization'] = `Bearer ${session.accessToken}`

return session

},

async redirect({url, baseUrl}) {

if (url === baseUrl + `/signin`) {

axios.defaults.headers.common['Authorization'] = ``

}

if (url.startsWith(baseUrl)) {

return url

} else if (url.startsWith("/")) {

return new URL(url, baseUrl).toString()

}

return baseUrl

}

},

debug: false

}

export default (req, res) => NextAuth(req, res, options)

Последним шагом идет создание форм, из которых состоят модули. Формы представляют из себя React компонент, содержащий ~~который содержит~~ набор полей или таблицу. При создании форм мы использовали созданные нами React компоненты. В качестве примера создания формы мы опишем форму создания клиента в системе – CreateClient (листинг 24). Процесс создания формы можно разделить на следующие этапы:

1. Создание страницы. В Next.js создание страниц заключается в формировании ~~создании~~ файловой структуры, где адрес каждой страницы это - путь до файла от папки pages, в котором находится React компонент, содержащий элементы страницы. Мы создали страницу /clients/create, ~~создав~~ определив следующую структуру: папка pages содержит папку clients, которая содержит файл create.js. Файловую структуру нашего проекта можно увидеть на рисунке 13.

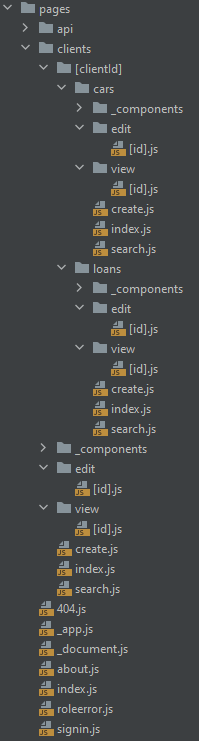


Рис. 13. Файловая структура страниц.

1. Создание менеджера форм и валидация. Мы создали объект менеджера форм formik (листинг 24) и схему валидации Yup (листинг 25). В объекте formik мы указали начальные значения полей (в данном случае это пустые строки), схему валидации Yup и обработчик формы onSubmit. Схема валидации Yup представляет из себя объект с правилами и ограничениями полей. Обработчик формы вызывается, когда пользователь сохраняет запись. Данный обработчик обращается к backend endpoint`у, отвечающему за сохранение записи клиента, и перенаправляет пользователя на форму просмотра записи, которую он создал.

Листинг 24 – объект formik

const formik = useFormik({

initialValues: {

name: ``,

dateOfBirth: ``,

passportNumber: ``,

nationality: ``,

},

validationSchema: formValidation,

onSubmit: () => {

create(formik.values).then(r =>

router.push({

pathname: `/clients/view/${r.id}`,

})

)

}

})

Листинг 25 – схема валидации Yup

const formValidation = Yup.object().shape({

name: Yup.string().trim()

.min(5, `Too Short`).required(`Name is required`),

dateOfBirth: Yup.string().nullable()

.required(`Date of Birth is required`),

passportNumber: Yup.string()

.length(10, `Passport number must be 10 symbols`).required(`Passport Number is required`),

nationality: Yup.string()

.required(`Nationality is required`),

})

1. Создание полей. В React компоненте формы мы создаем поля, используя уже созданные нами компоненты полей (листинг 26). В параметры полей мы передаем обработчики formik для управления данными и валидации. Также мы используем другие React компоненты для отображения панели инструментов и панели навигации.

Листинг 26 – поля формы создания клиентов

<MainLayout title={`Search Posts`}>

<ClientsNavBar childPanelsEnabled={false}/>

<FormBody

searchEnabled

submitEnabled

onCreate={() => router.push({

pathname: `/clients/create`

})}

onSearch={() => router.push({

pathname: `/clients/search`

})}

onSubmit={formik.handleSubmit}

>

<TextField

width={20}

label={`Name`}

name={`name`}

placeholder={'Input Name'}

value={formik.values.name}

handleChange={formik.handleChange}

handleBlur={formik.handleBlur}

error={formik.errors.name}

/>

<DateField

label={`Date Of Birth`}

name={`dateOfBirth`}

inputFormat={`dd/MM/yyyy`}

value={formik.values.dateOfBirth}

handleChange={formik.setFieldValue}

handleBlur={formik.handleBlur}

error={formik.errors.dateOfBirth}

/>

<TextField

width={20}

label={`Passport Number`}

name={`passportNumber`}

placeholder={'Input Passport Number'}

value={formik.values.passportNumber}

handleChange={formik.handleChange}

handleBlur={formik.handleBlur}

error={formik.errors.passportNumber}

/>

<ComboBoxField

label={`Nationality`}

name={`nationality`}

placeholder={`Choose Nationality`}

value={formik.values.nationality}

options={SSOptions}

handleChange={formik.setFieldValue}

handleBlur={formik.handleBlur}

error={formik.errors.nationality}

/>

</FormBody>

</MainLayout>

1. SSR. Для получения записей ComboBox, мы использовали SSR, создав функцию getServerSideProps (листинг 27). Данная функция выполняет две задачи: проверка ролей пользователя и получение данных для генерации страницы на стороне сервера. Для проверки ролей мы получаем список ролей пользователя из сессии и проверяем наличие необходимой. Если у пользователя нет необходимой роли, то он перенаправляется на страницу с сообщением об ошибке. Для получения данных мы вызываем backend endpoint и возвращаем их в объекте props в поле SSOptions. Для того чтобы React компонент получил эти данные, мы объявляем в его сигнатуре параметр с таким же названием.

Листинг 27 – схема валидации Yup

export const getServerSideProps = async ({query, req}) => {

const {roles} = await getSession({req})

const options = await getNationality()

if (!roles.includes(`ROLE\_ADMIN`)) {

return {

redirect: {

destination: '/roleerror',

permanent: false,

},

}

}

return {

props: {

SSOptions: options,

}

}

}

Такой подход ускоряет и упрощает разработку системы, превращая сам процесс разработки в сборку конструктора из готовых элементов интерфейса и логики. Также с таким подходом данные о сессии пользователя доступны в любой точке интерфейса, что позволяет имплементировать механизм авторизации во всех React компонентах.

## Этапы использования системы

Для того чтобы пользователи смогли пользоваться системой, надо создать для них учетные записи и распределить роли. После этого пользователи смогут зайти в систему с помощью своих учетных данных и пользоваться теми модулями, для которых у них есть необходимые роли.

При заходе в систему, пользователи попадают на страницу аутентификации (рисунок 14). На данной странице пользователи вводят свои учетные данные и получают доступ к системе.

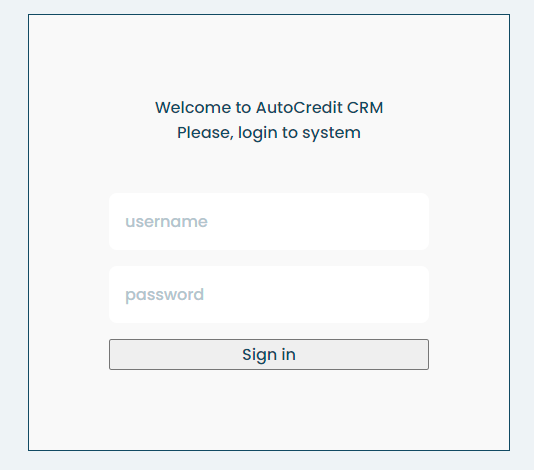


Рис. 14. Страница аутентификации.

После аутентификации пользователи попадают на домашнюю страницу, где находится панель навигации по системе. В нашем случае данная панель имеет следующие вкладки: Home, Clients, Sign out. При переходе на вкладку Clients, пользователи попадают на списочную форму модуля Clients, где отображены клиенты, сохраненные в системе (рисунок 15). С данной формы можно перейти на формы создания (рисунок 16), изменения (рисунок 17) и просмотра (рисунок 18) записей клиентов, а также можно удалить любую запись.

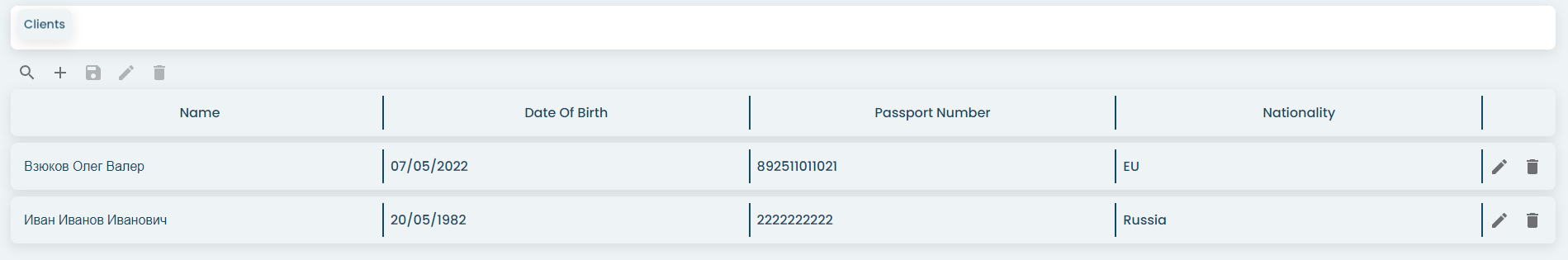


Рис. 15. Списочная форма модуля Clients. Содержит уже созданные записи о клиентах.

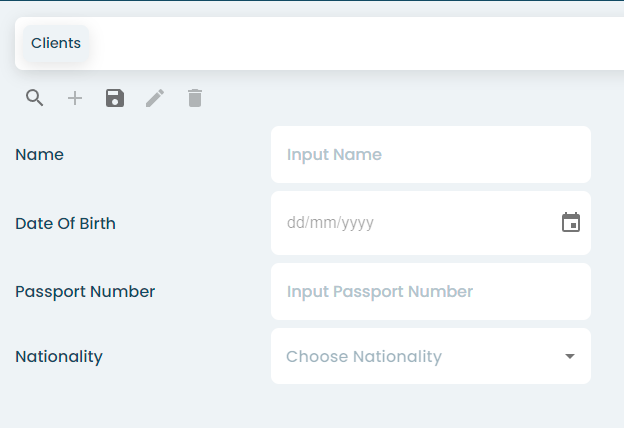


Рис. 16. Форма создания модуля Clients. Поля пусты.

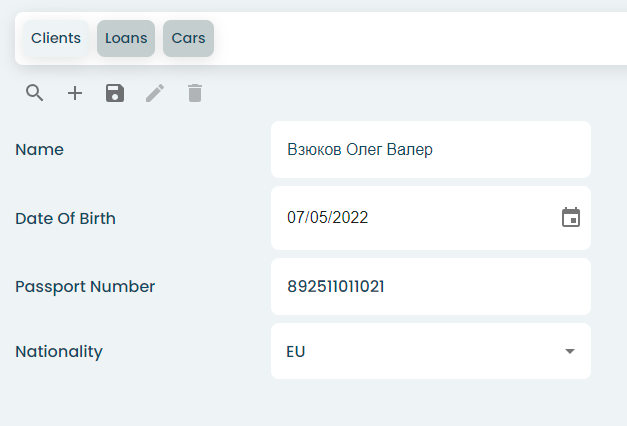


Рис. 17. Форма изменения модуля Clients. Поля содержат текущие значения из записи клиента.

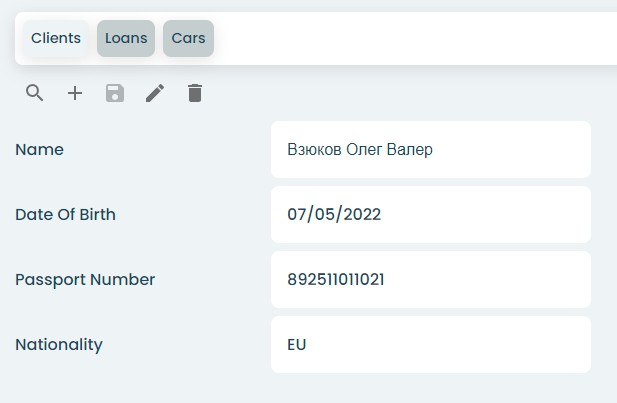


Рис. 18. Форма просмотра модуля Clients. Взаимодействовать с полями нельзя.

Если пользователь нажмет на кнопку удаления напротив записи, то высветится одно из следующих всплывающих окон:

1) Подтверждение удалении записи (рисунок 19);

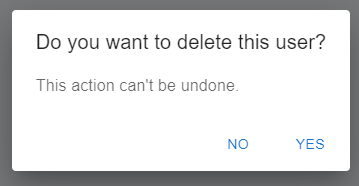


Рис. 19. Всплывающее окно при нажатии на кнопу удаления.

2) Уведомление об отсутствии необходимой роли чтобы совершить данное действие (рисунок 20).

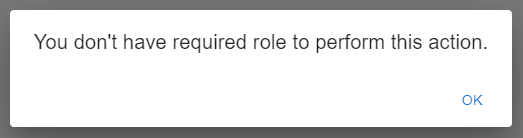


Рис. 20. Всплывающее окно при отсутствии необходимой роли.

Также при переходе в дочерние модули Clients (Loans и Cars), обновляется панель навигации по модулям, выделяя модуль, в котором находится пользователь (рисунок 21).

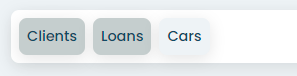


Рис. 21. Панель навигации по модулям. Пользователь находится в модуле Cars.

## Публикация проекта с открытым доступом

Мы выложили исходный код своего проекта в открытый доступ в системе GitHub [57]. Для этого нам необходимо было создать репозиторий – специальное хранилище для отслеживания изменения исходного кода и доступа к нему с любого устройства.

Директорию удаленного репозитория можно увидеть на рисунке 22.

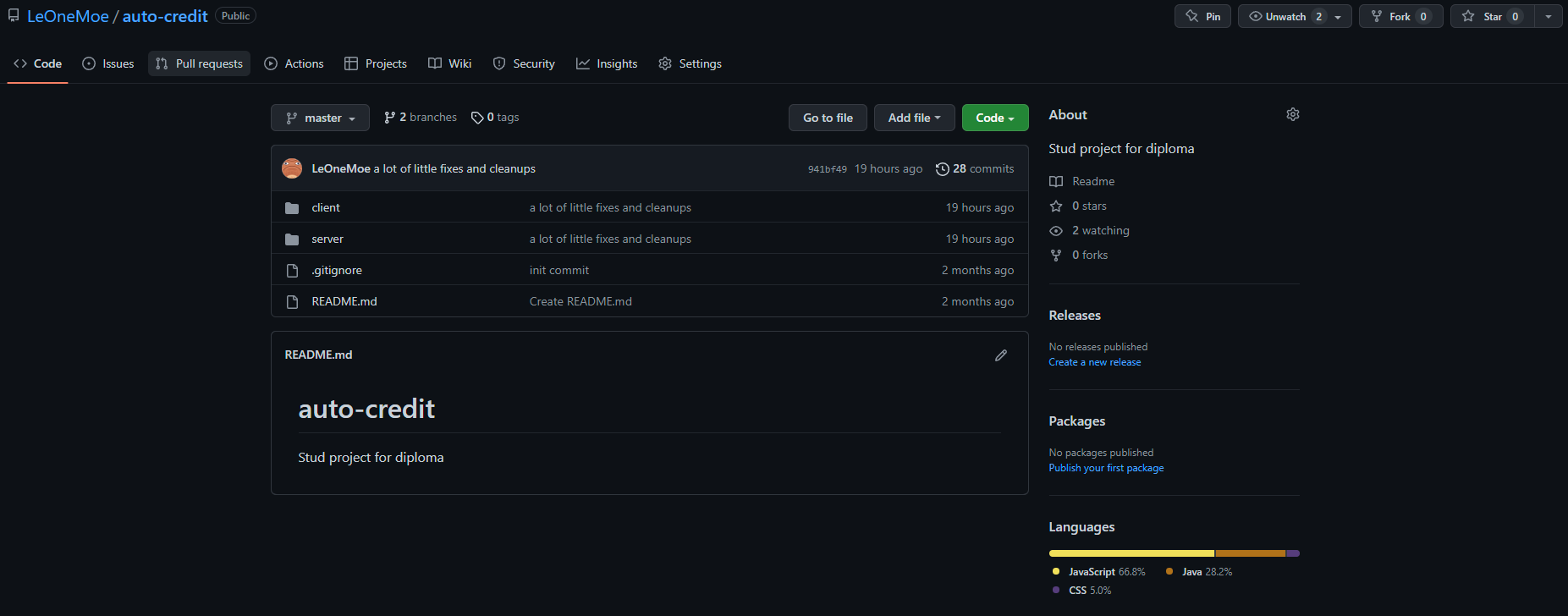


Рис. 22. Репозиторий проекта на GitHub.

На странице надо увеличить шрифт Ctrl+ и перескринить, а то слишком мелко.

# ЗАКЛЮЧЕНИЕ

Проанализировав материалы о CRM системах, мы пришли к выводу, что CRM-системы сегодня – это краеугольный камень автоматизации бизнеса. Такие системы обеспечивают организации аналитикой, сокращают время на рутинные задачи, повышают производительность сотрудников, помогают в вовлечении новых клиентов и повышают их удовлетворенность.

Без CRM-систем в наше время способны функционировать только бизнесы, сделки которых исчисляются единицами. У любого бизнеса от малого до большого должна быть подобная система, настроенная и отточенная для максимального соответствия установленным практикам и бизнес-процессам.

Были проанализированы несколько существующих CRM-систем и выявлены их сильные и слабые стороны. Определены основные компоненты CRM-систем.

Была спроектирована наша система и сформирован набор требований к каждому ее компоненту. Отталкиваясь от выдвинутых требований, мы сформировали архитектуру системы и подобрали языки программирования и изученли технологии для разработки.

В процессе реализации системы был создан каркас и программный конструктор на языках программирования Java и JavaScript с базой данных PostgeSql и с использованием таких технологий, как Spring, Spring Security, Hibernate, Next.js, React, formik и другие. Разработка системы велась в среде разработки IntelliJ IDEA. Также в процессе реализации соблюдались современные практики и методы разработки программного обеспечения и механизмов информационной безопасности.

На основе каркаса и программного конструктора был создан прототип CRM-системы для автокредитования, который отвечал всем выдвинутым требованиям.

Исходный код нашего проекта выложен в открытый доступ.

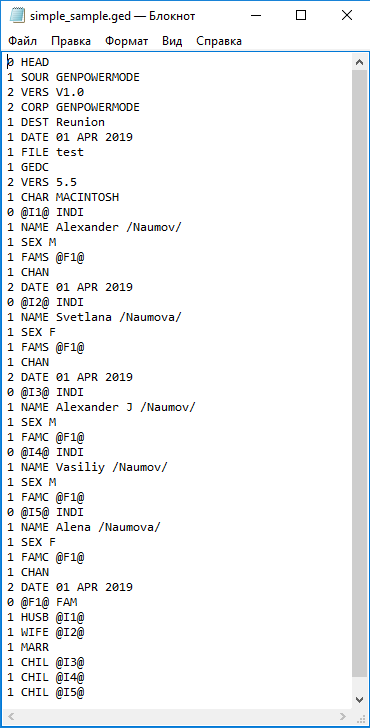
Добавить пару предложений о том, что работа позволяет... расписать положительный эффект от применения на практике ее результатов.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. CRM — для кого?. — Текст : электронный // Хабр : [сайт]. — URL: https://habr.com/ru/company/regionsoft/blog/500006/ (дата обращения: 29.03.2022).
2. 22 вопроса о CRM, которые искали в поиске, а ответим мы. — Текст : электронный // Хабр : [сайт]. — URL: https://habr.com/ru/company/regionsoft/blog/594519/ (дата обращения: 29.03.2022).
3. Francis, B. Customer Relationship Management: Concepts and Technologies / B. Francis, M. Stan. — 3-е изд. — Нью-Йорк : Routledge, 2015. — 426 c. — Текст : непосредственный.
4. Anastasiia, D. Collaborative CRM: Key Features and Benefits / D. Anastasiia. — Текст : электронный // trujay : [сайт]. — URL: https://www.trujay.com/blog/collaborative-crm-key-features-and-benefits (дата обращения: 31.03.2022).
5. Allie, D. What’s a Customer Data Platform? The Ultimate Guide to CDPs / D. Allie. — Текст : электронный // hubspo : [сайт]. — URL: https://blog.hubspot.com/service/customer-data-platform-guide (дата обращения: 31.03.2022).
6. Brian, C. Best CRM Software / C. Brian. — Текст : электронный // Investopedia : [сайт]. — URL: https://www.investopedia.com/best-crm-software-5116883#toc-best-overall-salesforce (дата обращения: 04.04.2022).
7. Salesforce. — Текст : электронный // Salesforce : [сайт]. — URL: https://www.salesforce.com/ (дата обращения: 04.04.2022).
8. Freshsales Suite. — Текст : электронный // Freshworks : [сайт]. — URL: https://www.freshworks.com/crm/suite/ (дата обращения: 04.04.2022).
9. Pipedrive. — Текст : электронный // Pipedrive : [сайт]. — URL: https://www.pipedrive.com/ (дата обращения: 04.04.2022).
10. Jim, G. Transaction Processing: Concepts and Techniques / G. Jim. — Сан-Франциско : San Mateo, CA. : Morgan Kaufmann Publishers, 1993. — 1122 c. — Текст : непосредственный.
11. Cross-Origin Resource Sharing (CORS). — Текст : электронный // MDN Web Docs : [сайт]. — URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS (дата обращения: 14.05.2022).
12. Eberhard, W. Microservices: Flexible Software Architecture / W. Eberhard. — Крофордсвилл : Pearson Education, 2016. — 368 c. — Текст : непосредственный.
13. Alyssa, W. SOAP Vs. REST: Difference between Web API Services / W. Alyssa. — Текст : электронный // guru99 : [сайт]. — URL: https://www.guru99.com/comparison-between-web-services.html (дата обращения: 14.05.2022).
14. Martin, R. API Design for C++ / R. Martin. — Берлингтон : Elsevier Science, 2011. — 472 c. — Текст : непосредственный.
15. Lokesh, G. What is REST / G. Lokesh. — Текст : электронный // restfulapi : [сайт]. — URL: https://restfulapi.net/ (дата обращения: 14.05.2022).
16. Tabassum, K. Server-Side Rendering (SSR) Vs Client-Side Rendering (CSR) / K. Tabassum. — Текст : электронный // dev : [сайт]. — URL: https://dev.to/codewithtee/server-side-rendering-ssr-vs-client-side-rendering-csr-3m24 (дата обращения: 14.05.2022).
17. Claudia, S. SR vs CSR- which is the right choice for your Progressive Web App (PWA)? / S. Claudia. — Текст : электронный // kruschecompany : [сайт]. — URL: https://kruschecompany.com/ssr-or-csr-for-progressive-web-app/ (дата обращения: 14.05.2022).
18. Server-side web frameworks. — Текст : электронный // MDN Web Docs : [сайт]. — URL: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First\_steps/Web\_frameworks (дата обращения: 14.05.2022).
19. Introduction to client-side frameworks. — Текст : электронный // MDN Web Docs : [сайт]. — URL: https://developer.mozilla.org/en-US/docs/Learn/Tools\_and\_testing/Client-side\_JavaScript\_frameworks/Introduction (дата обращения: 14.05.2022).
20. SPA (Single-page application). — Текст : электронный // MDN Web Docs : [сайт]. — URL: https://developer.mozilla.org/en-US/docs/Glossary/SPA (дата обращения: 14.05.2022).
21. Piotr, M. Component-Based Design. What Is It? What Is It All About? What Are Its Benefits? / M. Piotr. — Текст : электронный // Droptica : [сайт]. — URL: https://www.droptica.com/blog/component-based-design/ (дата обращения: 14.05.2022).
22. MVC. — Текст : электронный // MDN Web Docs : [сайт]. — URL: https://developer.mozilla.org/en-US/docs/Glossary/MVC (дата обращения: 14.05.2022).
23. Anshul, B. DAO vs Repository Patterns / B. Anshul. — Текст : электронный // Baeldung : [сайт]. — URL: https://www.baeldung.com/java-dao-vs-repository#repositoryPattern (дата обращения: 14.05.2022).
24. Repository Pattern. — Текст : электронный // deviq : [сайт]. — URL: https://deviq.com/design-patterns/repository-pattern (дата обращения: 14.05.2022).
25. Persistence Ignorance. — Текст : электронный // deviq : [сайт]. — URL: https://deviq.com/principles/persistence-ignorance (дата обращения: 14.05.2022).
26. Kathy, S. Effective Java / S. Kathy, B. Bert. — 3-е изд. — : Addison-Wesley Professional, 2017. — 412 c. — Текст : непосредственный.
27. A SMALL PLACE TO DISCOVER LANGUAGES IN GITHUB. — Текст : электронный // GitHut 2.0 : [сайт]. — URL: https://madnight.github.io/githut/#/pull\_requests/2021/4 (дата обращения: 14.05.2022).
28. Craig, W. Effective Java / W. Craig. — 5-е изд. — Шелтер-Айленд, Нью-Йорк : Manning, 2018. — 520 c. — Текст : непосредственный.
29. Spring Boot Starter Web. — Текст : электронный // javatpoint : [сайт]. — URL: https://www.javatpoint.com/spring-boot-starter-web (дата обращения: 14.05.2022).
30. Spring Data JPA. — Текст : электронный // Хабр : [сайт]. — URL: https://habr.com/ru/post/435114/ (дата обращения: 14.05.2022).
31. Christian, B. Java Persistence with Hibernate / B. Christian, K. Gavin, G. Gary. — 2-е изд. — Шелтер-Айленд, Нью-Йорк : Manning, 2015. — 608 c. — Текст : непосредственный.
32. Security with Spring. — Текст : электронный // Baeldung : [сайт]. — URL: https://www.baeldung.com/security-spring (дата обращения: 14.05.2022).
33. Project Lombok. — Текст : электронный // Project Lombok : [сайт]. — URL: https://projectlombok.org/ (дата обращения: 14.05.2022).
34. Benjamin, M. Gradle in Action / M. Benjamin. — 1-е изд. — Шелтер-Айленд : Manning, 2014. — 480 c. — Текст : непосредственный.
35. Jones, M. JSON Web Token (JWT) / M. Jones. — Текст : электронный // Internet Engineering Task Force (IETF) : [сайт]. — URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата обращения: 14.05.2022).
36. Sebastián, E. JWT Handbook / E. Sebastián. — Текст : электронный // auth0 : [сайт]. — URL: https://assets.ctfassets.net/2ntc334xpx65/o5J4X472PQUI4ai6cAcqg/13a2611de03b2c8edbd09c3ca14ae86b/jwt-handbook-v0\_14\_1.pdf (дата обращения: 14.05.2022).
37. Java JWT. — Текст : электронный // github : [сайт]. — URL: https://github.com/auth0/java-jwt (дата обращения: 14.05.2022).
38. Marijn, H. Eloquent JavaScript: A Modern Introduction to Programming / H. Marijn. — 3-е изд. — Сан-Франциско : No Starch Press, 2018. — 472 c. — Текст : непосредственный.
39. Azat, M. Practical Node.js: Building Real-World Scalable Web Apps / M. Azat. — 1-е изд. — Нью-Йорк : Apress, 2018. — 505 c. — Текст : непосредственный.
40. Run JavaScript Everywhere. — Текст : электронный // nodejs : [сайт]. — URL: https://nodejs.dev/ (дата обращения: 17.05.2022).
41. yarn. — Текст : электронный // yarnpkg : [сайт]. — URL: https://yarnpkg.com/ (дата обращения: 17.05.2022).
42. Alfrick, O. NPM vs. Yarn: Which Package Manager Should You Choose? / O. Alfrick. — Текст : электронный // whitesourcesoftware : [сайт]. — URL: https://www.whitesourcesoftware.com/free-developer-tools/blog/npm-vs-yarn-which-should-you-choose/ (дата обращения: 17.05.2022).
43. Tutorial: Intro to React. — Текст : электронный // reactjs : [сайт]. — URL: https://reactjs.org/tutorial/tutorial.html (дата обращения: 17.05.2022).
44. React.Component. — Текст : электронный // reactjs : [сайт]. — URL: https://reactjs.org/docs/react-component.html#gatsby-focus-wrapper (дата обращения: 17.05.2022).
45. Introducing JSX. — Текст : электронный // reactjs : [сайт]. — URL: https://reactjs.org/docs/introducing-jsx.html#gatsby-focus-wrapper (дата обращения: 17.05.2022).
46. Introduction. — Текст : электронный // nextjs : [сайт]. — URL: https://nextjs.org/learn/foundations/about-nextjs?utm\_source=next-site&utm\_medium=homepage-cta&utm\_campaign=next-website (дата обращения: 17.05.2022).
47. Recommended Toolchains. — Текст : электронный // reactjs : [сайт]. — URL: https://reactjs.org/docs/create-a-new-react-app.html#recommended-toolchains (дата обращения: 17.05.2022).
48. Getting Started. — Текст : электронный // nextjs : [сайт]. — URL: https://nextjs.org/docs/getting-started (дата обращения: 17.05.2022).
49. Overview. — Текст : электронный // formik : [сайт]. — URL: https://formik.org/docs/overview (дата обращения: 17.05.2022).
50. Validation. — Текст : электронный // formik : [сайт]. — URL: https://formik.org/docs/guides/validation#validationschema (дата обращения: 17.05.2022).
51. Getting Started. — Текст : электронный // axios : [сайт]. — URL: https://axios-http.com/ (дата обращения: 17.05.2022).
52. Getting Started. — Текст : электронный // next-auth : [сайт]. — URL: https://next-auth.js.org/getting-started/example (дата обращения: 17.05.2022).
53. Move faster with intuitive React UI tools. — Текст : электронный // MUI : [сайт]. — URL: https://mui.com/ (дата обращения: 17.05.2022).
54. Open-Source Database Software. — Текст : электронный // trustradius : [сайт]. — URL: https://www.trustradius.com/open-source-database (дата обращения: 17.05.2022).
55. Brandon, C. PostgreSQL vs. MySQL / C. Brandon. — Текст : электронный // fivetran : [сайт]. — URL: https://www.fivetran.com/blog/postgresql-vs-mysql (дата обращения: 17.05.2022).
56. What is IntelliJ IDEA?. — Текст : электронный // IntelliJ IDEA : [сайт]. — URL: https://www.jetbrains.com/idea/features/ (дата обращения: 17.05.2022).
57. Взюков, О. В. auto-credit / О. В. Взюков. — Текст : электронный // github : [сайт]. — URL: https://en.wikipedia.org/wiki/GitHub (дата обращения: 20.05.2022).
58. May, O. 4 Types of CRM: In-Depth Guide / O. May. — Текст : электронный // HuskyJam : [сайт]. — URL: https://huskyjam.com/blog/types-of-crm-software/ (дата обращения: 20.05.2022).

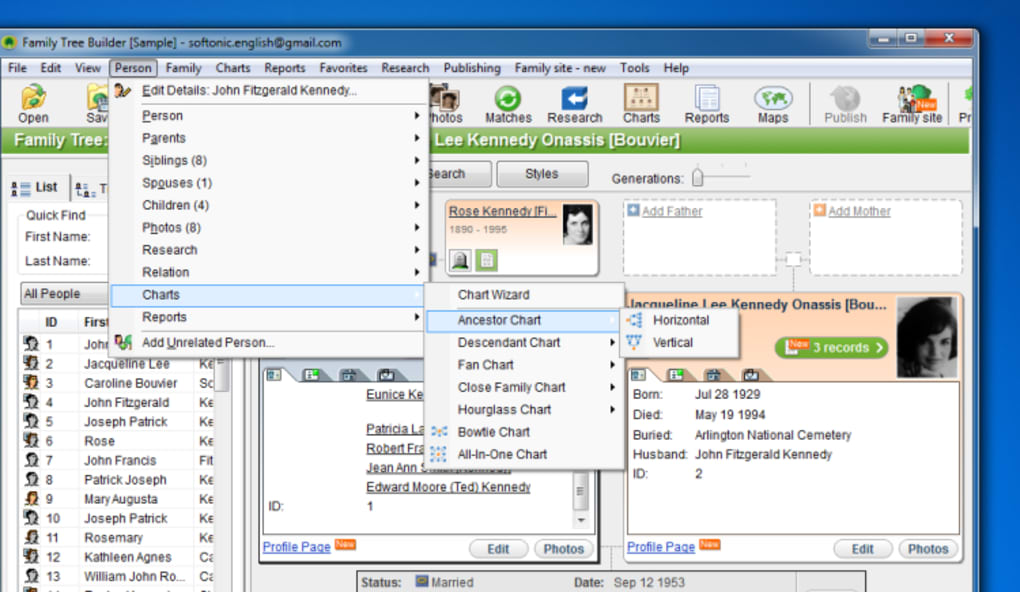
# Приложение 1

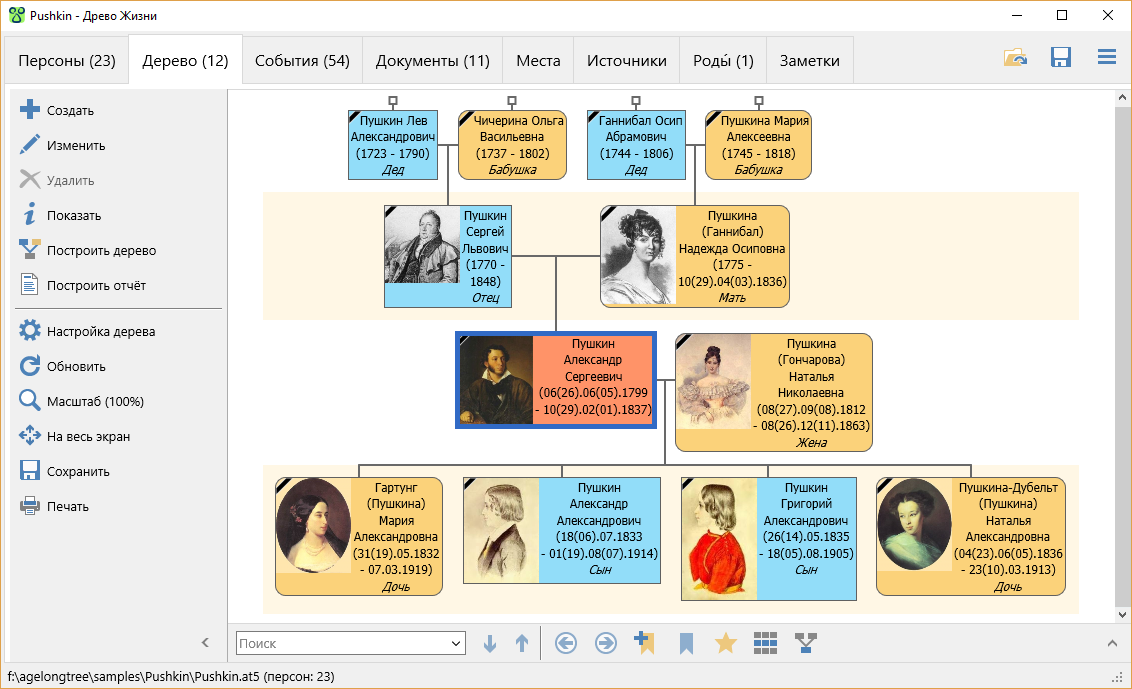
Пример отображения простого примера файла спецификации GEDCOM

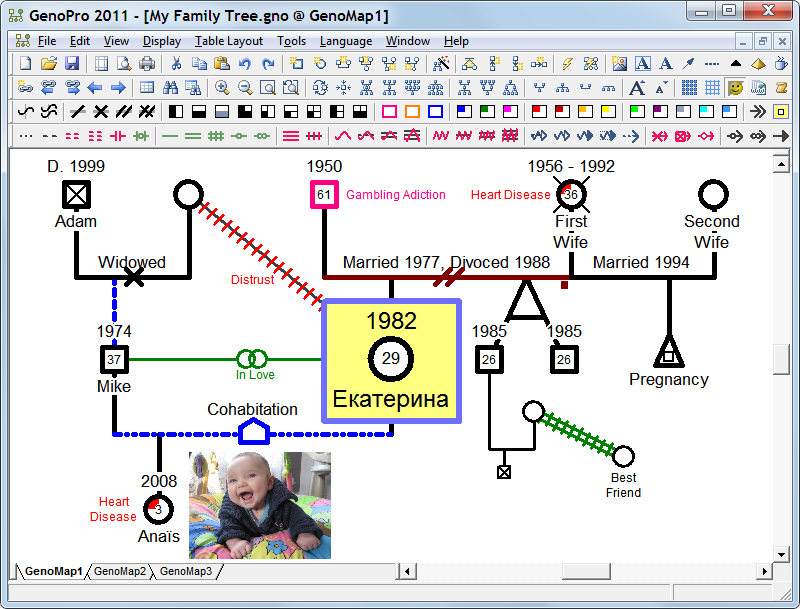
  
Рис. 1.1. Файл simple\_sample.ged

# Приложение 2

Скриншоты рабочих окон генеалогических программ

  
Рис 2.1. Family Tree Builder

  
Рис 2.2. Древо жизни

  
Рис. 2.3. Genopro