**Лист с заданием**

**Резюме**

Тема дипломной работы «Программный комплекс учета и мониторинга выполнения заказов ремонтной компании».

В ходе выполнения дипломной работы был разработан программный комплекс, который помогает пользователям в формировании и мониторинге заказов для ремонтной компании, менеджерам помогает в обработки заказов, рабочим помогает в удобном отслеживании заказов.

Объектом разработки является программное средство, предназначенное для автоматизации работы с заказами как со стороны пользователей, так и со стороны сотрудников.

Поставленная задача была успешно выполнена в полном объеме.

**Рэзюмэ**

Тэма дыпломнай працы»праграмны комплекс ўліку і маніторынгу выпол-нення заказаў рамонтнай кампаніі".

У ходзе выканання дыпломнай працы быў распрацаваны праграмны комплекс, які дапамагае карыстальнікам ў фарміраванні і маніторынгу заказов для рамонтнай кампаніі, менеджэрам дапамагае ў апрацоўкі заказаў, рабочим дапамагае ў зручным удасканаленьні заказаў.

Аб'ектам распрацоўкі з'яўляецца праграмнае сродак, прызначанае для аўтаматызацыі працы з замовамі як з боку карыстальнікаў, так і з боку супрацоўнікаў.

Пастаўленая задача была паспяхова выканана ў поўным аб'ёме.

**Summary**

*The topic of the thesis is "A software package for accounting and monitoring the fulfillment of orders of a repair company".*

*During the completion of the thesis, a software package was developed that helps users in the formation and monitoring of orders for a repair company, helps managers in processing orders, and helps workers in conveniently tracking orders.*

*The object of development is a software tool designed to automate work with orders from both users and employees.*

*The task was successfully completed in full.*

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 6](#_Toc136259020)

[1 Анализ предметной области 7](#_Toc136259021)

[1.1 История развития ремонтных компаний 7](#_Toc136259022)

[1.2 Обзор существующих систем автоматизации 9](#_Toc136259023)

[1.3 Выбор языка программирования WEB-приложения 10](#_Toc136259024)

[1.4 Выбор языка программирования Android-приложения 11](#_Toc136259025)

[1.5 Технология Entity Framewokr 13](#_Toc136259026)

[1.6 Технология ASP.NET 14](#_Toc136259027)

[1.7 Выбор базы данных 18](#_Toc136259028)

[2 Архитектура приложения 20](#_Toc136259029)

[2.1 Функциональная схема 20](#_Toc136259030)

[2.2 База данных 23](#_Toc136259031)

[2.3 Общая структура приложения 30](#_Toc136259032)

[3 Разработка программного обеспечения 36](#_Toc136259033)

[3.1 Описание реализации программного комплекса 36](#_Toc136259034)

[3.2 Проект базы данных 36](#_Toc136259035)

[3.3 Серверная часть программного комплекса 37](#_Toc136259036)

[3.4 Клиентские приложения 39](#_Toc136259037)

[4 Тестирование, верефикация и валидация 53](#_Toc136259038)

[4.1 Модульное тестирование 53](#_Toc136259039)

[4.2 Ручное тестирование программного комплекса 55](#_Toc136259040)

# **ВВЕДЕНИЕ**

Программный комплекс, разработанный для учета и мониторинга заказов ремонтной компании, является важным инструментом для повышения эффективности работы компании. Он позволяет автоматизировать многие процессы, связанные с обработкой заказов, что значительно сокращает время на их выполнение и повышает качество обслуживания клиентов.

Одна из главных функций программного комплекса – это учет заказов. С его помощью можно быстро создавать новые заказы, указывать все необходимые данные о клиенте, виде работ, сроках выполнения и т.д. Кроме того, система позволяет отслеживать статус заказа на каждом этапе выполнения работ – от приемки заявки до окончания ремонта.

Важным элементом программного комплекса является мониторинг выполнения работ. С его помощью можно контролировать качество выполняемых работ, проверять соблюдение сроков и требований клиента. Также система позволяет вести учет затрат на каждый заказ, что помогает оптимизировать расходы компании и повысить ее прибыльность.

Еще одной важной функцией программного комплекса является управление персоналом. С его помощью можно назначать задачи сотрудникам, отслеживать их выполнение, контролировать рабочее время и оплату труда. Это позволяет повысить эффективность работы персонала и сократить время на выполнение заказов.

В целом, программный комплекс для учета и мониторинга заказов ремонтной компании является незаменимым инструментом для повышения эффективности работы компании. Он позволяет автоматизировать многие процессы, связанные с обработкой заказов, повысить качество обслуживания клиентов и увеличить прибыльность компании.

Целью данной дипломной работы является проектировка программного комплекса управления ремонтной компании, которая позволит пользователям пройти полный путь от создания заказа, до отзыва после его выполнения, сотрудникам легко обрабатывать заказы от пользователей и позволит организовать упорядочивание списков в базах данных, для составления отчетности.

Задачами дипломной работы являются:

* изучение методик разработки клиент-серверных приложений;
* классификация ролей пользователей и их ролевые политики;
* изучение методов реализации серверной части для приложения;
* проектирование структуры приложения, базы данных для хранения информации, формирование пользовательских правил для доступа к ресурсам и функциям приложения;
* разработка программных модулей: обеспечивающих авторизацию и аутентификацию пользователей; работу с данными с помощью графического интерфейса;
* верификация и опытная эксплуатация разработанного программного обеспечения.

# **1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ**

## **1.1 История развития ремонтных компаний**

Возникновение ремонта связано с появлением объекта производства. Производство металлических изделий вызвало необходимость устранения де­фектов производства этих изделий, а также восстановления деталей, пов­режденных и изношенных в процессе эксплуатации.

Одним из первых профессиональных ремонтников был кузнец, а кузница явилась прообразом ремонтных мастерских. В XV столетии от кузнечного ре­месла отделилось слесарное. Развитию ремонта в XVI-XVIII столетиях спо­собствовало применение литья, использование часов и мельниц, различных машин, приборов и инструмента.

Большой вклад в развитие ремонта в этот период внесли русские изоб­ретатели и ученые. Выдающийся русский изобретатель А.К.Нартов (1693 - 1756) предложил новые методы заделки раковин и трещин в отливках стволов пушек. Известный русский изобретатель Кулибин И.О. (1735 -1818) занимался ремонтом и изготовлением часов. В 1770 - 1785 г. он ру­ководил академическими мастерскими по ремонту приборов и инструмента [1].

Бурное развитие в XIX столетии железнодорожного транспорта, появле­ние автомобильного и воздушного транспорта потребовали дальнейшего развития ремонта, создания специальных ремонтных мастерских. При ремонте паровозов нашли применение изобретения Н.Н. Бенардоса - дуговая электросварка металлов (1886 г.) неплавящимся угольным электродом и Н. Г. Словянова - плавящимся электродом (1888 г.).

Остро встал вопрос о ремонте машин в России после первой мировой войны и революции 1917 г., когда производство автомобильной и другой техники было практически приостановлено. Дальнейшее развитие ремонтного производства осуществлялось вместе с развитием машиностроения и других отраслей народного хозяйства.

Особая роль ремонту техники и развитию ремонтного производства отводится в армии, где от своевременного восстановления вышедшей из строя техники зависит успех боевых действий. В Вооруженных Силах ремонт машин представляет собой систему, включающую эвакуацию поврежденной техники с поля боя, снабжение ремонтных подразделений и частей автомобильным и другими видами имущества, восстановление работоспособности поврежденных машин.

Первые мастерские по ремонту автомобилей в русской армии были созданы в 1911 г. в учебной автомобильной роте, а в начале первой мировой войны - тыловые авторемонтные мастерские фронтов.

Ремонт автомобилей выполнялся: ремонтными мастерскими автомобильных рот - текущий, средний, капитальный; тыловыми авторемонтными мастерскими - средний и капитальный. Кроме того, для капитального ремонта автомобилей привлекались заводы промышленности.

К концу войны русская армия имела тридцать одну мастерскую общей мощностью до 13450 капитальных ремонтов в год. В 1920 г. на каждые 400 автомобилей приходилась одна мастерская ар­мейского или фронтового подчинения. Однако и при этом к 1923 г. около 40 % автомобилей были неисправными [1].

С переходом на мирное положение в отдельных автогрузовых отрядах были мастерские для текущего и частично среднего ремонта, в военных ок­ругах - окружные авторемонтные мастерские, решавшие такие же задачи. Кроме того, функционировало несколько железнодорожных поездов-мастерских и военных авторемонтных заводов для капитального ремонта машин.

В 1936 г. в механизированных бригадах создаются ремонтно-восстано­вительные батальоны (*рвб*) двух ротного состава: ремонтной роты для те­кущего ремонта транспортных и боевых машин и тракторной - для эвакуации машин. В 1938 г. создаются автобронетанковые ремонтные базы для капи­тального ремонта автомобилей, тракторов, танков, двигателей, агрегатов и мотоциклов. В это время была разработана организационная структура от­дельного ремонтно-восстановительного батальона с размещением оборудова­ния на двадцати автомобилях ЗИС-6 и двух прицепах.

В конце 1939 г. были созданы унифицированные походные мастерские-летучки типа А на шасси ГАЗ-АА или ГАЗ-ААА (предназначенные для техни­ческого обслуживания машин, демонтажно-монтажных и регулировочных работ) и летучки типа Б на шасси ЗИС-5 или ЗИС-6 и прицепе 2-АП-З7 - для теку­щего и среднего ремонта машин всех типов.

В течение полтора года Великой Отечественной войны были сформированы подвижные ремонтные базы, подвижные авторемонтные заводы, ремонтные батальоны и т.д.

В 1943 г. было создано Главное автомобильное управление, которое к концу года имело 312 различных ремонтных частей, баз, мастерских изаво­дов, способных выполнять около 28 тыс. средних и 8 тыс. капитальных ремонтов автомобилей в месяц. За три года войны (1942 - 1944) более 1,5 млн. автомобилей были восстановлены путем проведения среднего икапи­тального ремонтов, что почти в 3 раза больше, чем поступило в армию за эти годы.

В послевоенный период ремонтные подразделения и части совершенство­вались путем создания новых ремонтных мастерских, модернизации их обору­дования, изменения организационно-штатной структуры и т.д.

Развитие ремонта осуществляется на научной основе. Ремонт как науч­ная дисциплина включает сумму знаний о возникновении в машинахнеисправ­ностей, методов их обнаружения и устранения, об организации производства ремонта машин во всех звеньях в мирное и военное время.

Огромный вклад в развитие ремонтного производства внесли такие ученые, как профессор Казарцев В.И., Ефремов В.В., Шадричев В.А. и др.

Таким образом, ремонтное производство – совокупность ремонтных подразделений и частей, баз, мастерских и заводов, имеющих целью восстановление утраченных свойств машин – их исправности или работоспособности.

Предметом труда ремонтного производства, являются неисправные машины - ремонтный фонд, готовой продукцией - восстановленные машины.

В настоящее время существуют различные фирмы, предоставляющие услуги по ремонту. Среди них наибольшей популярностью пользуются *Pridex*, *DWG Construction Company*, *Atitoka*, «Гинт-М», КМТ и другие.

## **1.2 Обзор существующих систем автоматизации**

Существует множество сервисов, в той, или иной мере реализующих требования к данному дипломному проекту. Они имеют схожий функционал и структуру.

Первый ресурс, располагающий подобным функционалом – *kabanchik.by. Kabanchik.by* – онлайн-сервис поиска исполнителей для бытовых и бизнес задач. Проект объединяет заказчиков, которым необходимо выполнить какую-либо работу, и компетентных исполнителей, ищущих подработку или дополнительный заработок.

Первая версия проекта была запущена 17 сентября 2012. Изначально проект начал работать только на территории Украины.

С 2015 года, география проекта расширилась на территорию Республики Беларусь, России и Казахстана.

Согласно [2], *Kabanchik* – это сервис поиска частных специалистов для решения бытовых, а также решения бизнес-задач. Данный сервис предоставляет возможность клиенту сделать заказ на услуги вида: домашний мастер, отделочные работы, клининговые услуги, курьерские услуги, строительные работы, ремонт техники, логистические и складские услуги, бытовые услуги, мебельные работы и прочее. Потенциальный клиент оставляет заявку на нужный ему вид услуг и ждет, пока сервис подберет ему команду или специалиста для ее выполнения. В сервисе существует отчетность для выбора лучшего варианта. Обобщая все выше написанное, можно заявить, что данный сервис располагает полным спектром возможностей необходимых для комфортной работы как клиентам, так и компаниям, предоставляющим услуги.

Второй ресурс, реализующий нужды дипломного проекта – *redsale.by*. *Redsale.by* — платформа для размещения анкет специалистов во всех сферах оказания услуг и объявлений от заказчиков, которым эти услуги понадобились. Здесь есть репетиторы, мастера красоты; мастера по ремонту квартир, мебели (и всего, что с этим связано), техники; артисты; фото- и видеосъемка; спортивные тренеры. Площадка работает для всех областных центров Беларуси.

Согласно [3], это бесплатный сервис по подбору частных специалистов. В нем существует возможность найти специалистов в сферах ремонта жилых помещений, ремонта техники, строительства, грузоперевозки, ремонта автомобилей. Данный сервис проводит автоматизацию работы компаний, которые им пользуются, что непосредственно влияет на их продуктивность. Автоматизация облегчает жизнь как клиентам, так и исполнителям, что не может не сказываться на положительном впечатлении от использования данного сервиса. На *redsale.by* существует свой прейскурант, ведение статистики и, прочий функционал, соответствующий решению поставленной задачи.

Третий ресурс – строительная биржа *Remline.by*. *Remline.by* – это строительство и ремонт посредством создания тендера. Портал выводит уровень строительства и ремонта на новый уровень, позволяя заказчикам значительно экономить свои денежные средства, а также получать более качественные услуги за счет конкуренции.

Согласно [4], это уникальная биржа в сфере строительства, которая позволяет создать тендер частному лицу, что позволяет сильно сэкономить денежные средства. *Remline* сервис обладающий всем необходимым функционалом для решения задач курсового проекта, в нем можно создать заказ, сервис его обработает и подберет свободную команду для его выполнения, для услуг существует прейскурант, так же существуют отзывы клиентов и примеры работ различных команд. Сервис проводит автоматизацию работ ремонтно-строительных служб, что позволяет им выполнять свою работу с наибольшей эффективностью.

Проанализировав приведенные ранее сервисы, можно сделать вывод о том, какой основной функционал должен быть у приложения, а именно: оформление, подтверждение и выполнение заказов, хранения базы данных о клиентах, их заказах, различного рода отчеты для администраторов приложений.

## **1.3 Выбор языка программирования WEB-приложения**

*С#* – это объектно-ориентированный язык программирования. Он был создан в период с 1998 по 2002 год командой инженеров *Microsoft* под руководством Андерса Хейлсберга и Скотта Вильтаумота.

Язык входит в семью *С*-подобных языков. Синтаксис приближен к *Java* и *C++*. Его особенности:

* статистическая типизация,
* поддерживается полиморфизм,
* поддерживается перегрузка операторов,
* доступна делегация, атрибуты, события, обобщенные типы и анонимные функции.

Разработка *Microsoft* много особенностей унаследовала у *Delphi*, *Smalltalk* и *Java*. При этом создатели нового языка исключили из своего продукта многие практики и спецификации, считающиеся «проблемными».

Понятный синтаксис C# заметно упрощает не только разработку как таковую, но и другие важные аспекты совместной работы, например, чтение чужого кода. Это упрощает процесс рефакторинга и исправления ошибок при работе над приложениями в больших командах.

Также нельзя не упомянуть низкий порог вхождения. *С#* – популярная и достаточно простая в освоении технология.

Язык *C#* практически универсален. Можно использовать его для создания любого ПО: продвинутых бизнес-приложений, видеоигр, функциональных веб-приложений, приложений для *Windows*, *macOS*, мобильных программ для *iOS* и *Android*.

Объектно-ориентированный: в *С#* все является объектом. Дополнение может быть легко расширено, так как он основан на объектной модели

Кроссплатформенная разработка: С помощью *C#* и .*NET* можно создавать приложения, которые могут работать на нескольких платформах, включая *Windows*, *macOS*, *Linux* и даже на мобильных устройствах с *iOS* и *Android*. Можете охватить широкую аудиторию и создавать приложения, которые можно использовать на различных типах устройств.

Интеграция с продуктами *Microsoft*: Освоение *C#* и .*NET* упрощает разработку и развертывание приложений на платформе *Microsoft*, что идеально, для работы в компаниях, которые в значительной степени полагаются на такие технологии, как *Azure* и *Visual* *Studio*. Эти навыки также помогают работать над будущими моделями искусственного интеллекта, такими как *ChatGPT* и *Bing* *AI* от Microsoft, которые сегодня набирают все большую популярность во многих отраслях.

Масштабируемость: *C*# и .*NET* — это высокомасштабируемые технологии, подходящие для небольших и крупных приложений, включая настольные, веб-приложения, игры и мобильные приложения, что делает их идеальными для приложений корпоративного уровня. Эта универсальность также означает, что есть возможность использовать эти технологии для создания приложений любого типа, которые наилучшим образом соответствуют вашим требованиям и требованиям вашей организации [5].

**1.4 Выбор языка программирования Android-приложения**

*Android* *Studio* — интегрированная среда разработки (*IDE*) для работы с платформой *Android*, анонсированная 16 мая 2013 года на конференции *Google* *I/O*. В последней версии *Android* *Studio* поддерживается *Android* 4.1 и выше.

Данная *IDE* находилась в свободном доступе начиная с версии 0.1, опубликованной в мае 2013, а затем перешла в стадию бета-тестирования, начиная с версии 0.8, которая была выпущена в июне 2014 года. Первая стабильная версия 1.0 была выпущена в декабре 2014 года, тогда же прекратилась поддержка плагина *Android Development Tools (ADT)* для *Eclipse*.

*Android* *Studio*, основанная на программном обеспечении *IntelliJ* *IDEA* от компании *JetBrains*, — официальное средство разработки *Android* приложений. Данная среда разработки доступна для *Windows*, *macOS* и *GNU*/*Linux*. 17 мая 2017, на ежегодной конференции *Google I/O*, *Google* анонсировал поддержку языка *Kotlin*, используемого в *Android* Studio, как официального языка программирования для платформы *Android* в дополнение к *Java* и *C++.*

Язык *Java* разработан компанией *Sun* *Microsystems*, создателем которого был Джеймс Гослинг, и выпущен в 1995 году в качестве основных компонентов компании *Sun* *Microsystems* – *Java* платформ (*Java* 1.0 [*J2SE*]).

По состоянию на 2023 год последней версией *Java* *Standard* *Edition* является 16. С развитием *Java*, и её широкой популярностью, несколько конфигураций были построены для различных типов платформ. Например, *J2EE* – приложения для предприятий, *J2ME* – для мобильных приложений. Sun Microsystems переименовала прежнюю версию *J2* и ввела новые: *Java SE, Java EE* и *Java ME*. Введение в программирование *Java* различных версий подтверждало знаменитый слоган компании «Напиши один раз, запускай везде».

Объектно-ориентированный: в *Java* все является объектом. Дополнение может быть легко расширено, так как он основан на объектной модели.

Платформонезависимый: в отличие от многих других языков, включая *C* и *C++,* *Java*, когда был создан, он не компилировался в платформе конкретной машины, а в независимом от платформы байт-коде. Этот байт код 12 распространяется через интернет и интерпретируется в *Java Virtual Machine (JVM)*, на которой он в настоящее время работает.

Простой: процессы изучения и введение в язык программирования Java остаются простыми. Если Вы понимаете основные концепции объектно-ориентированного программирования, то он будет прост для Вас в освоении.

Безопасным: методы проверки подлинности основаны на шифровании с открытым ключом.

Архитектурно-нейтральным: компилятор генерирует архитектурно-нейтральные объекты формата файла, что делает скомпилированный код исполняемым на многих процессорах, с наличием системе *Java* *Runtime*.

Портативный: архитектурно-нейтральный и не имеющий зависимости от реализации аспектов спецификаций – все это делает *Java* портативным. Компилятор в *Java* написан на *ANSI* *C* с чистой переносимостью, который является подмножеством *POSIX*.

Прочный: выполняет усилия, чтобы устранить ошибки в различных ситуациях, делая упор в основном на время компиляции, проверку ошибок и проверку во время выполнения.

Многопоточный: функции многопоточности, можно писать программы, которые могут выполнять множество задач одновременно. Введение в язык *Java* этой конструктивной особенности позволяет разработчикам создавать отлаженные интерактивные приложения.

Интерпретированный: *Java* байт-код переводится на лету в машинные инструкции и нигде не сохраняется. Делая процесс более быстрым и аналитическим, поскольку связывание происходит как дополнительное с небольшим весом процесса.

Высокопроизводительный: введение *Just*-*In*-*Time* компилятора, позволило получить высокую производительность.

Распространенный: предназначен для распределенной среды интернета.

Динамический: программирование на *Java* считается более динамичным, чем на *C* или *C++,* так как он предназначен для адаптации к меняющимся условиям. Программы могут выполнять обширное количество во время обработки информации, которая может быть использована для проверки и разрешения доступа к объектам на время выполнения [6].

**1.5 Технология Entity Framewokr**

Для доступа к данным будет использована технология *Entity* *Framework* *Core*.

Entity *Framework* (*EF*) *Core* — это простая, кроссплатформенная и расширяемая версия популярной технологии доступа к данным *Entity* *Framework* с открытым исходным кодом.

*EF* *Core* может использоваться в качестве объектно-реляционного модуля сопоставления (O/RM), который:

* Позволяет разработчикам .*NET* работать с базой данных с помощью объектов .*NET*.
* Устраняет необходимость в большей части кода для доступа к данным, который обычно приходится писать.

В *EF* *Core* доступ к данным осуществляется с помощью модели. Модель состоит из классов сущностей и объекта контекста, который представляет сеанс взаимодействия с базой данных. Объект контекста позволяет выполнять запросы и сохранять данные.

*EF* поддерживает следующие подходы к разработке моделей:

* Создание модели на основе существующей базы данных.
* Написание кода модели вручную в соответствии с базой данных.
* После создания модели используйте Миграции *EF* для создания базы данных на основе модели. Миграции позволяют развивать базу данных по мере изменения модели.

Хотя *EF* *Core* хорошо подходит для абстрагирования многих деталей при программировании, существуют некоторые рекомендации, которые применяются ко всем технологиям *O*/*RM* и помогают избежать распространенных ошибок в рабочих приложениях.

* Знания базового сервера базы данных на промежуточном уровне или выше необходимы для проектирования, отладки, профилирования и переноса данных в рабочих приложениях с высокой производительностью. Например, знание первичных и внешних ключей, ограничений, индексов, нормализации, инструкций *DML* и *DDL*, типов данных, профилирования и т. д.
* Функциональное тестирование и тестирование интеграции — важно реплицировать рабочую среду как можно точнее, чтобы находите проблемы в приложениях, которые отображаются только при использовании определенных версий или выпуска сервера базы данных и перехватывать критические изменения при обновлении *EF* *Core* и других зависимостей. Например, при добавлении или обновлении платформ, таких как *ASP*.*NET* *Core*, *OData* или *AutoMapper*. Эти зависимости могут повлиять на *EF* *Core* неожиданным образом.
* Производительность и нагрузочное тестирование с репрезентативными нагрузками. Упрощенное использование некоторых функций затрудняет масштабирование. К таким случаям относятся: использование *Includes* в нескольких коллекциях, интенсивное применение отложенной загрузки, условные запросы к неиндексированным столбцам, массовые обновления и вставки с использованием созданных хранилищем значений, отсутствие обработки параллелизма, крупные модели и недостаточная политика кэширования.
* Проверки безопасности: например, обработка строк подключения и других секретов, разрешения базы данных для операций, не связанных с развертыванием, проверка ввода для необработанных запросов *SQL*, шифрование конфиденциальных данных.
* Восстановление после ошибок. Проанализируйте непредвиденные ситуации для распространенных сценариев сбоев, такие как откат версий, резервные серверы, горизонтальное масштабирование и балансировка нагрузки, устранение атак типа "отказ в обслуживании" и резервное копирование данных.
* Развертывание и миграция приложений. Спланируйте, как будет применяться миграция во время развертывания. Если миграция выполняется при запуске приложения, на нее могут негативно повлиять проблемы с параллелизмом, к тому же для нее потребуются более высокие разрешения по сравнению с обычной работой. Используйте промежуточное хранение, чтобы упростить восстановление после неустранимых ошибок во время миграции. Дополнительные сведения см. в разделе Применение миграций.
* Подробное изучение и тестирование созданных миграций. Необходимо тщательно протестировать миграции перед применением к рабочим данным. Форму схемы и типы столбцов невозможно легко изменить, если таблицы содержат рабочие данные. Например, в *SQL* *Server* *nvarchar*(*max*) и *decimal*(18, 2) редко являются наиболее подходящими типами для столбцов, сопоставленных со строковыми и десятичными свойствами. Однако в EF они используются по умолчанию, так как EF не знает о вашем конкретном сценарии [7].

## **1.6 Технология ASP.NET**

Платформой для создания *Web*-приложения была выбрана *ASP*.*NET* *MVC*.

*ASP*.*NET* *MVC* является многофункциональной платформой для создания веб-приложений и *API*-интерфейсов с помощью структуры проектирования *Model*-*View*-*Controller*. Структура архитектуры *MVC* разделяет приложение на три основных группы компонентов: модели, представлении и контроллеры. Это позволяет реализовать принципы разделения задач. Согласно этой структуре, запросы пользователей направляются в контроллер, который отвечает за работу с моделью для выполнения действий пользователей и (или) получение результатов запросов. Контроллер выбирает представление для отображения пользователю со всеми необходимыми данными модели.

На следующей схеме показаны три основных компонента и существующие между ними связи.



Рисунок 1 – Схема компонентов и взаимодействия паттерна *MVC*

Такое распределение обязанностей позволяет масштабировать приложение в контексте сложности, так как проще писать код, выполнять отладку и тестирование компонента (модели, представления или контроллера) с одним заданием. Гораздо труднее обновлять, тестировать и отлаживать код, зависимости которого находятся в двух или трех этих областях. Например, логика пользовательского интерфейса, как правило, подвергается изменениям чаще, чем бизнес-логика. Если код представления и бизнес-логика объединены в один объект, содержащий бизнес-логику, объект необходимо изменять при каждом обновлении пользовательского интерфейса. Это часто приводит к возникновению ошибок и необходимости повторно тестировать бизнес-логику после каждого незначительного изменения пользовательского интерфейса.

Представление и контроллер зависят от модели. Однако сама модель не зависит ни от контроллера, ни от представления. Это является одним из ключевых преимуществ разделения. Такое разделение позволяет создавать и тестировать модели независимо от их визуального представления.

Функции модели: Модель в приложении *MVC* представляет состояние приложения и бизнес-логику или операций, которые должны в нем выполняться. Бизнес-логика должна быть включена в состав модели вместе с логикой реализации для сохранения состояния приложения. Как правило, строго типизированные представления используют типы *ViewModel*, предназначенные для хранения данных, отображаемых в этом представлении. Контроллер создает и заполняет эти экземпляры *ViewModel* из модели.

Функции представления: Представления отвечают за представление содержимого через пользовательский интерфейс. Они используют обработчик представлений *Razor* для внедрения .*NET* кода в разметку *HTML*. Представления должны иметь минимальную логику, которая должна быть связана с представлением содержимого. Если есть необходимость выполнять большую часть логики в представлении для отображения данных из сложной модели, рекомендуется воспользоваться компонентом представления, *ViewModel* или шаблоном представления, позволяющими упростить представление.

Функции контроллера: Контроллеры — это компоненты для управления взаимодействием с пользователем, работы с моделью и выбора представления для отображения. В приложении *MVC* представление служит только для отображения информации. Обработку введенных данных, формирование ответа и взаимодействие с пользователем обеспечивает контроллер. В структуре *MVC* контроллер является начальной отправной точкой и отвечает за выбор рабочих типов моделей и отображаемых представлений (именно этим объясняется его название — он контролирует, каким образом приложение отвечает на конкретный запрос).

*ASP*.*NET* *Core* *MVC* представляет собой упрощенную, эффективно тестируемую платформу с открытым исходным кодом, оптимизированную для использования с *ASP*.*NET* *Core*.

*ASP*.*NET* *Core* *MVC* предоставляет основанный на шаблонах способ создания динамических веб-сайтов с четким разделением задач. Она обеспечивает полный контроль разметки, поддерживает согласованную с *TDD* разработку и использует новейшие веб-стандарты.

Платформа *ASP.NET* *Core* *MVC* создана на основе маршрутизации *ASP*.*NET* *Core* — мощного компонента сопоставления *URL*-адресов, который позволяет создавать приложения с понятными и поддерживающими поиск *URL*-адресами. Вы можете определять шаблоны именования *URL*-адресов приложения, эффективно работающие для оптимизации для поисковых систем (*SEO*) и для создания ссылок, независимо от способа организации файлов на веб-сервере. Вы можете определять маршруты с помощью понятного синтаксиса шаблонов маршрутов, который поддерживает ограничения значений маршрутов, значения по умолчанию и необязательные значения.

Маршрутизация на основе соглашений позволяет глобально определять форматы *URL*-адресов, которые принимает приложение, и как каждый из этих форматов сопоставляется с определенным методом действия на заданном контроллере. При поступлении входящего запроса модуль маршрутизации выполняет синтаксический анализ *URL*-адреса и соотносит его с одним из определенных форматов *URL*-адресов, а затем вызывает метод действия связанного контроллера.

Маршрутизация атрибутов используется для указания сведений о маршрутизации путем добавления атрибутов, определяющих маршруты приложения, к контроллерам и действиям. Это означает, что определения маршрутов помещаются рядом с контроллером и действием, с которым они связаны.

Внедрение зависимостей: *ASP.NET Core* имеет встроенную поддержку внедрения зависимостей (*DI*). В *ASP.NET MVC Core* контроллеры могут запрашивать необходимые служб через свои конструкторы, предоставляя им возможность следовать принципу явных зависимостей.

Фильтры: Фильтры помогают разработчикам решать общие задачи, такие как обработка исключений или авторизация. Фильтры активируют пользовательскую логику предварительной и завершающей обработки для методов действий и могут быть настроены для запуска в определенные моменты в конвейерном выполнении определенного запроса. Фильтры могут применяться к контроллерам или действиям в виде атрибутов (или могут выполняться глобально). В состав платформы входит несколько фильтров (например, *Authorize*). [*Authorize*] является атрибутом, который используется для создания фильтров авторизации *MVC*.

Области: Области позволяют секционировать большое *ASP.NET Core* веб-приложение *MVC* на небольшие функциональные группы. Область является структурой *MVC* внутри приложения. В проекте *MVC* логические компоненты, такие как модель, контроллер и представление, находятся в разных папках, и для создания связи между этими компонентами *MVC* использует соглашения об именовании. Крупное приложение может быть целесообразно разделить на отдельные высокоуровневые области функциональности. Например, приложение электронной коммерции с несколькими бизнес-подразделениями, такими как извлечение, выставление счетов и поиск и т. д. Каждый из этих единиц имеет собственные представления логических компонентов, контроллеры и модели.

Веб-*API*: Помимо того, что *ASP.NET Core MVC* прекрасно подходит для создания веб-сайтов, эта платформа располагает мощной поддержкой для построения веб-*API*. Создавайте службы, доступные для широкого круга клиентов, включая браузеры и мобильные устройства. Платформа включает поддержку согласования содержимого *HTTP* со встроенной поддержкой форматирования данных в формате *JSON* или *XML*. Пишите пользовательские модули форматирования для добавления поддержки собственных форматов.

Тестирование: Благодаря используемым интерфейсам и внедрению зависимостей платформа хорошо подходит для модульного тестирования. Кроме того, с помощью таких компонентов, как *TestHost* и поставщик *InMemory* для *Entity Framework*, можно быстро и просто выполнять интеграционные тесты.

*Razor* обработчик представлений: *ASP.NET Core* представления *MVC* используют обработчик представлений *Razor* для отрисовки представлений. *Razor*— это компактный, экспрессивный и гибкий язык разметки шаблона для определения представлений с помощью внедренного кода *C#*. *Razor* используется для динамического создания веб-содержимого на сервере. Серверный код можно полностью комбинировать с содержимым и кодом на стороне клиента. С помощью обработчика представлений *Razor* можно определить макеты, частичные представления и заменяемые разделы.

Строго типизированные представления: *Razor* представления в *MVC* могут быть строго типизированы на основе модели. Контроллеры передают строго типизированную модель в представления для поддержки в них *IntelliSense* и проверки типов.

Вспомогательные функции тегов: Вспомогательные функции тегов позволяют коду на стороне сервера участвовать в создании и отрисовке *HTML*-элементов в *Razor* файлах. Вспомогательные функции тегов используются для определения настраиваемых тегов (например, *<environment>*) или для изменения поведения существующих тегов (например, <*label*>). Вспомогательные функции тегов привязываются к определенным элементам на основе имени элемента и его атрибутов. Они предоставляют преимущества отрисовки на стороне сервера, сохраняя при этом возможности редактирования *HTML*. Существует множество встроенных вспомогательных функций тегов для общих задач — например, для создания форм, ссылок, загрузки ресурсов и так далее. Кроме того, огромное количество функций доступно в общедоступных репозиториях *GitHub* и в качестве пакетов *NuGet*. Вспомогательные функции тегов разрабатываются на *C#* и предназначены для *HTML*-элементов на основе имени элемента, имени атрибута или родительского тега[8].

## **1.7 Выбор базы данных**

В качестве хранилища данных была выбрана *MS* *SQL* *Server* 2019*. SQL* Server – это система управления реляционными базами данных, или RDBMS, разработанная и продаваемая *Microsoft*.

Подобно другому программному обеспечению СУБД, *SQL* *Server* построен на основе *SQL*, стандартного языка программирования для взаимодействия с реляционными базами данных. Сервер *SQL* связан с *Transact*-*SQL* или *T*-*SQL*, реализацией *SQL* от *Microsoft*, которая добавляет набор проприетарных программных конструкций.

*SQL* *Server* работает исключительно в среде *Windows* более 20 лет. В 2016 году *Microsoft* сделала его доступным для *Linux*. *SQL Server 2017* стал общедоступным в октябре 2016 года и работал как в Windows, так и в Linux. 2019 версия также поддерживает *Linux* и *Docker*.

*SQL Server* состоит из двух основных компонентов:

* *Database Engine*
* *SQLOS*

*Database Engine*: Основным компонентом *SQL Server* является компонент *Database Engine*. Компонент *Database Engine* состоит из реляционного механизма, который обрабатывает запросы, и механизма хранения, который управляет файлами базы данных, страницами, индексами и т. д. Объекты базы данных, такие как хранимые процедуры, представления и триггеры, также создаются и выполняются компонентом *Database Engine*.

Реляционный движок (*Relational Engine*): Реляционный движок содержит компоненты, определяющие наилучший способ выполнения запроса. *Relational* *Engine* также известен как обработчик запросов. Реляционный механизм запрашивает данные из механизма хранения на основе входного запроса и обрабатывает результаты. Некоторые задачи реляционного механизма включают обработку запросов, управление памятью, управление потоками и задачами, управление буфером и распределенную обработку запросов.

Движок хранения (*Storage Engine*): Механизм хранения отвечает за хранение и извлечение данных из систем хранения, таких как диски и *SAN*.

*SQLOS*: Под реляционным механизмом и хранилищем понимается операционная система *SQL Server* или *SQLOS*. *SQLOS* предоставляет множество сервисов операционной системы, таких как управление памятью и вводом-выводом. Другие услуги включают в себя обработку исключений и услуги синхронизации [9].

# **2 АРХИТЕКТУРА ПРИЛОЖЕНИЯ**

## **2.1 Функциональная схема**

Функциональная схема необходима, чтобы показать взаимодействия между пользователями. На рисунке 2.1 представлена функциональная схема, составленная для данной системы.

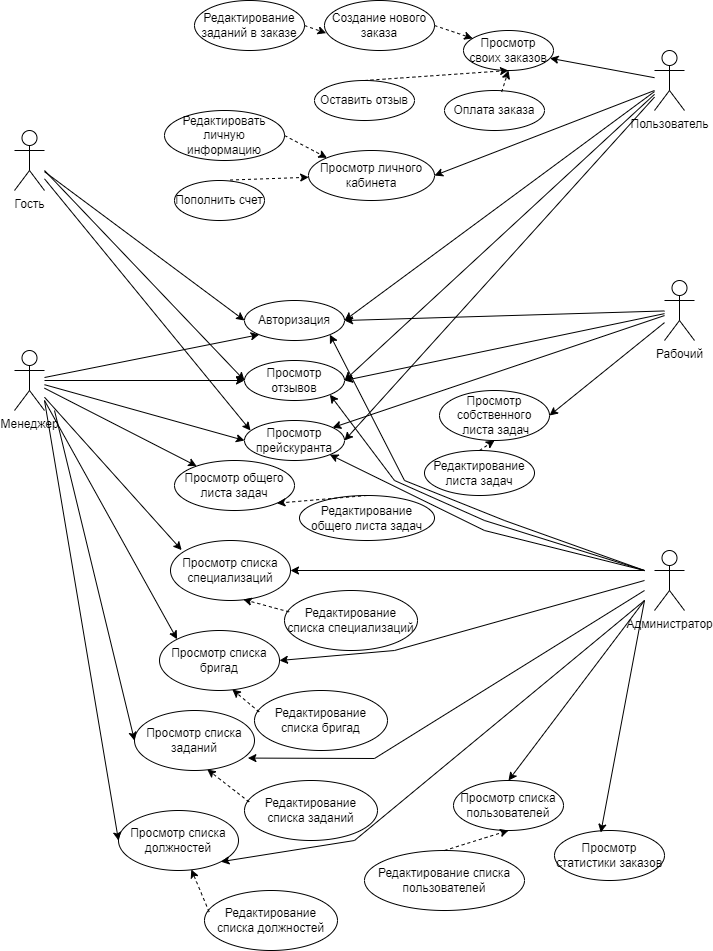


Рисунок 2.1 – Функциональная схема

В системе есть 5 ролей. Все роли между собой объединяют прецеденты «Просмотр отзывов», «Просмотр прейскуранта» и «Авторизация». Менеджер, рабочий и пользователь будут связаны между собой через заказ, роль администратора отвечает за нормальную и полноценную работу приложения, так она имеет возможность редактировать специализации, бригады, должности, имеет возможность менять пользователям роль, также обладает возможностью запросить статистику. Роль гостя обладает самыми скудными возможностями, а именно просмотром отзывов и прейскуранта.

В соответствии с функциональной схемой, можно сделать описание прецедентов и актеров.

* актёр «Гость» – актер с минимальными возможностями, может просматривать прейскурант и отзывы;
* актёр «Пользователь» – актер с возможностью создавать заказ и с возможностью его оплаты, так же может оставить отзыв на уже выполненный заказ, как и «гость» имеет возможность просматривать прейскурант;
* актёр «Менеджер» – актер, основная функция которого – подтверждать или отклонять поступивший от пользователя заказ, так же может редактировать списки бригад, заданий, специализаций, должностей;
* актёр «Администратор» – актер, который обладает следующими полномочиями: управление списком пользователей, а именно изменением ролей, редактированием списков бригад, заданий, специализаций, должностей, так же получение статистики как за год, так и за выбранный месяц;
* актер «Рабочий» – актер, обладающий функцией просмотра своего рабочего листа, в котором может отмечать выполненные задания, из количества выполненных заданий и их стоимости рассчитывается заработная плата для данного актера.

Описание прецедентов*:*

* прецедент «просмотр отзывов» – прецедент для просмотра всех отзывов, оставленных пользователями, отображается по умолчанию на главной странице изображения;
* прецедент «авторизация» – прецедент позволяет неавторизированным пользователям авторизироваться или зарегистрироваться;
* прецедент «просмотр прейскуранта» – просмотр всех заданий, которые может выполнить данная компания;
* прецедент «просмотр своих заказов» – просмотр уже существующих заказов пользователем;
* прецедент «создание нового заказа» – создание нового заказа пользователем;
* прецедент «редактирование заданий в заказе» – прецедент для добавления, изменения и удаления заданий в заказе;
* прецедент «просмотр списка пользователей» – прецедент для просмотра списка всех пользователей;
* прецедент «редактирования списка пользователей» – прецедент для просмотра учетных записей всех пользователей и изменения им ролей;
* прецедент «просмотр личного кабинета» – позволяет пользователю просмотреть свой личный кабинет;
* прецедент «редактирование личной информации» – изменение учётной записи, осуществляется самим пользователем этой учетной записи;
* прецедент «пополнить счет» – позволяет пользователю пополнить баланс на сайте
* прецедент «просмотр общего листа задач» – прецедент позволяет менеджеру просмотреть все заказы;
* прецедент «редактирование общего листа задач» – позволяет менеджеру подтвердить или отклонить заказ;
* прецедент «просмотр собственного листа задач» – позволяет рабочему просматривать собственный лист задач;
* прецедент «редактирование собственного листа задач» – прецедент, вызываемый работником в случае, если он выполнил задание, заказ считается закрытым, когда все задания из него выполнены;
* прецедент «просмотр статистики заказов» – прецедент, предназначенный для администратора, в котором он может просмотреть отчетность бригад по количеству выполненных заданий;
* прецедент «оставить отзыв» – после выполнения всех заданий из заказа пользователь может оставить отзыв, который будет виден на главной странице приложения;
* прецедент «просмотр списка специализаций» – позволяет просмотреть список специализаций;
* прецедент «редактирование списка специализаций» – создание, редактирование, удаление специализаций;
* прецедент «просмотр списка бригад» – позволяет просмотреть список бригад;
* прецедент «редактирование списка бригад» – создание, редактирование, удаление бригад;
* прецедент «просмотр списка заданий» – позволяет просмотреть список заданий;
* прецедент «редактирование списка заданий» – создание, редактирование, удаление заданий;
* прецедент «просмотр списка должностей» – позволяет просмотреть список должностей;
* прецедент «редактирование списка должностей» – создание, редактирование, удаление должностей;

## **2.2 База данных**

Базы данных – неотъемлемая часть работы подавляющего большинства программного обеспечения, существующего на сегодняшний день. Использование баз данных является достаточно эффективным способом хранить данные. Как уже упоминалось ранее, на этапе проектирования схемы данных было установлено, что проектируемые доменные сущности имеют множественные отношения друг между другом. Таким образом предпочтение в выборе СУБД было отдано реляционной базе данных *MSSQL*.

Структура базы данных представлена на рисунке 2.2.



Рисунок 2.2 – Схема базы данных

В процессе проектирования были выделены основные доменные классы Entity, объекты, которые соответствуют сущностям, находящимся в базе данных. Сущности представлены в таблице 2.1.

Таблица 2.1 – Описание таблиц базы данных.

|  |  |
| --- | --- |
| Название сущности | Назначение |
| *Order* | Хранение информации о заказе |
| *Feedback* | Хранение информации об отзывах |
| *Customer* | Хранение информации о пользователе |
| *AspNetUsers* | Хранение информации менеджерах, пользователях, работниках |
| *AspNetUserLogins* | Хранение возможностей авторизации пользователя |
| *AspNetRoles* | Хранение информации о возможных ролях |
| *AspNetUserRoles* | Хранение информации роли пользователя |
| *AspNetClaims* | Хранение требований к пользователю |
| *OrderTask* | Ассоциативная таблица между таблицами *Order* и *Task* |
| *Task* | Хранение информации о заказах |
| *Brigade* | Хранение информации о бригадах |
| *Specialization* | Хранение информации о специализациях |
| *Employee* | Хранение информации о работниках |
| *JobPosition* | Хранение информации о рабочих позициях |
| *Manager* | Хранение основной информации о менеджерах |

Описание атрибутов таблицы *Order* представлено в таблице 2.2.

Таблица 2.2 – Описание атрибутов таблицы «*Order*»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение | Первичный ключ | Внешний ключ |
| 1 | 2 | 3 | 4 | 5 |
| *Id* | *int* | Хранение номера записи в таблице | Да | Да |

Продолжение таблицы 2.2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| *Tilte* | *nvarchar(50)* | Хранение название заказа | Нет | Нет |
| *IdCustomer* | *Int* | Хранение номера пользователя | Нет | Да |
| *OrderStatus* | *nvarchar(50)* | Хранение состояние заказа на текущий момент | Нет | Нет |
| *Requirements* | *nvarchar(150)* | Хранение описание заказа | Нет | Нет |

Описание атрибутов таблицы *Feedback* представлено в таблице 2.3.

Таблица 2.3 – Описание атрибутов таблицы «*Feedback*»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение | Первичный ключ | Внешний ключ |
| *Id* | *int* | Хранение номера записи в таблице | Да | Да |
| *IdOrder* | *int* | Хранение название номера записи в таблице | Нет | Да |
| *Review* | *nvarchar(150)* | Хранение текста отзыва | Нет | Нет |

Описание атрибутов таблицы *Customer* представлено в таблице 2.4.

Таблица 2.4 – Описание атрибутов таблицы «*Customer*»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение | Первичный ключ | Внешний ключ |
| *Id* | *int* | Хранение номера записи в таблице | Да | Да |
| *IdentityUser* | *nvarchar(128)* | Хранение номер*a* записи о пользователе в таблице «*AspNetUsers*» | Нет | Да |

Описание атрибутов таблицы *AspNetUsers* представлено в таблице 2.5.

Таблица 2.5 – Описание атрибутов таблицы «*AspNetUsers*»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение | Первичный ключ | Внешний ключ |
| *Id* | *nvarchar(128)* | Хранение номера записи в таблице | Да | Да |
| *Surname* | *nvarchar(50)* | Хранение фамилии пользователя | Нет | Нет |
| *Balance* | *decimal* | Хранение баланса пользователя | Нет | Нет |
| *Email* | *nvarchar(50)* | Хранение электронной почты пользователя | Нет | Нет |
| *EmailConfirmed* | *bit* | Хранение подтверждения электронной почты | Нет | Нет |
| *PasswordHash* | *nvarchar(MAX)* | Хэш-образ пароля | Нет | Нет |
| *TwoFactorEnabled* | *bit* | Двух-факторная аунтификация | Нет | Нет |
| *PhoneNumber* | *nvarchar(50)* | Хранение телефонного номера пользователя | Нет | Нет |
| *PhoneNumberConfirmed* | *nvarchar(50)* | Хранение подтвержденности телефонного номера | Нет | Нет |
| *UserName* | *nvarchar(50)* | Хранение имени пользователя в его аккаунте | Нет | Да |

Описание атрибутов таблицы *AspNetLogins* представлено в таблице 2.6.

Таблица 2.6 – Описание атрибутов таблицы «*AspNetLogins*»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение | Первичный ключ | Внешний ключ |
| *LoginProvider* | *nvarchar(*128*)* | Хранение логина входящего пользователя | Да | Да |
| *ProviderKey* | *nvarchar(*128*)* | Хранение ключа входящего пользователя | Нет | Да |
| *UserId* | *nvarchar(*128*)* | Хранение номера пользователя | Нет | Да |

Описание атрибутов таблицы *AspNetRoles* представлено в таблице 2.7

Таблица 2.7 – Описание атрибутов таблицы «*AspNetRoles*»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение | Первичный ключ | Внешний ключ |
| *Id* | *nvarchar(*128*)* | Хранение уникальный номер роли | Да | Да |
| *Name* | *nvarchar(*128*)* | Хранение название роли | Нет | Да |

Описание атрибутов таблицы *AspNetUserRoles* представлено в таблице 2.8.

Таблица 2.8 – Описание атрибутов таблицы «*AspNetUserRoles*»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение | Первичный ключ | Внешний ключ |
| *UserId* | *nvarchar(*128*)* | Хранение номер пользователя | Да | Да |
| *RoleId* | *nvarchar(*128*)* | Хранение номер | Нет | Да |

Описание атрибутов таблицы *AspNetClaims* представлено в таблице 2.9.

Таблица 2.9 – Описание атрибутов таблицы «*AspNetClaims*»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение | Первичный ключ | Внешний ключ |
| 1 | 2 | 3 | 4 | 5 |
| *Id* | *int* | Хранение записи | Да | Да |
| *UserId* | *nvarchar(*128*)* | Хранение номер пользователя | Нет | Да |

Продолжение таблицы 2.9

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| *ClaimType* | *nvarchar(max)* | Хранение типа требования | Нет | Нет |
| *ClaimValue* | *nvarchar(max)* | Хранение трбования | Нет | Нет |

Описание атрибутов таблицы *OrderTask* представлено в таблице 2.10.

Таблица 2.10 – Описание атрибутов таблицы «*OrderTask*»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение | Первичный ключ | Внешний ключ |
| *Id* | *int* | Хранение номера записи | Да | Да |
| *OrderId* | *int* | Хранение номер заказа | Нет | Да |
| *TaskId* | *int* | Хранение номера задания | Нет | Да |
| *TaskCompletionDate* | *DateTime* | Хранение даты выполнения заказа | Нет | Нет |
| *Status* | *nvarchar(50)* | Хранение статуса задания | Нет | Нет |
| *Description* | *nvarchar(150)* | Хранение описания задания | Нет | Нет |

Описание атрибутов таблицы *Task* представлена в таблице 2.11.

Таблица 2.11 – Описание атрибутов таблицы «*Task*»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение | Первичный ключ | Внешний ключ |
| 1 | 2 | 3 | 4 | 5 |
| *Id* | *int* | Хранение номера записи | Да | Да |
| *Title* | *nvarchar(50)* | Хранение названия задания | Нет | Нет |
| *IdSpecialization* | *int* | Хранение номер специальности для данного задания | Нет | Да |
| *Price* | *decimal* | Хранение стоимости задания | Нет | Нет |
| *Description* | *nvarchar(150)* | Хранение описания задания | Нет | Нет |

Продолжение таблицы 2.11

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| *IdBrigade* | *int* | Хранение номера бригады | Нет | Да |

Описание атрибутов таблицы *Brigade* представлено в таблице 2.12.

Таблица 2.12 – Описание атрибутов таблицы «*Brigade*»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение | Первичный ключ | Внешний ключ |
| *Id* | *int* | Хранение номера записи | Да | Да |
| *Title* | *nvarchar(50)* | Хранение названия бригады | Нет | Нет |
| *IdSpecialization* | *int* | Хранение номер специальности для этой бригады | Нет | Да |

Описания атрибутов таблицы *Specialization* представлено в таблице 2.13.

Таблица 2.13 – Описание атрибутов таблицы «*Specialization*»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение | Первичный ключ | Внешний ключ |
| *Id* | *int* | Хранение номера записи | Да | Да |
| *Name* | *nvarchar(50)* | Хранение названия специализации | Нет | Нет |
| *Description* | *nvarchar(*1*50)* | Хранение описания специальности | Нет | Нет |

Описание атрибутов таблицы *Employee* представлено в таблице 2.14.

Таблица 2.14 – Описание атрибутов таблицы «*Employee*»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение | Первичный ключ | Внешний ключ |
| 1 | 2 | 3 | 4 | 5 |
| *Id* | *int* | Хранение номера записи | Да | Да |
| *IdBrigade* | *int* | Хранение номер бригады | Нет | Да |

Продолжение таблицы 2.14

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| *Selary* | *decimal* | Хранение коэффициента заработной платы | Нет | Нет |
| *IdJobPosition* | *int* | Хранение номера должности | Нет | Да |
| *IdentityUserId* | *nvarchar(128)* | Хранение номера пользователя в таблице «*AspNetUsers*» | Нет | Да |

Описание атрибутов таблицы *JobPosition* представлено в таблице 2.15.

Таблица 2.15 – Описание атрибутов таблицы «*JobPosition*»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение | Первичный ключ | Внешний ключ |
| *Id* | *int* | Хранение номера записи | Да | Да |
| *Title* | *nvarchar(50)* | Хранение названия должности | Нет | Нет |
| *Purpose* | *nvarchar(*15*0)* | Хранение описания должности | Нет | Нет |

Описание атрибутов таблицы *Manager* представлено в таблице 2.16.

Таблица 2.16 – Описание атрибутов таблицы «*Manager*»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя атрибута | Тип атрибута | Назначение | Первичный ключ | Внешний ключ |
| *Id* | *int* | Хранение номера записи | Да | Да |
| *Selary* | *decimal* | Хранение коэффициента заработной платы | Нет | Нет |
| *IdentityUserId* | *nvarchar(128)* | Хранение номера пользователя в таблице «*AspNetUsers*» | Нет | Да |

## **2.3 Общая структура приложения**

***2.3.1*** По поставленной задаче будет создано *WEB*-приложение. Приложение будет создано на клиент-серверном составляющем. Данное приложение будет имеет СУБД на основе *MSSQL*. Сервер будет написан на *C#* с использованием фреймворка *Entity Framework Core*. Клиент используемый в данном приложение, будет использовать *ASP.NET* фреймворк. Также будет разработано *android* приложение, которое будет позволять рабочим получать быстрый доступ к списку заказов на выполнение. *Android* общается с сервером, посылая ему запросы и обрабатывая ответы. Доступ к *android* приложению имеют только пользователи с ролью «рабочий».

Общая схема данного приложения представлена на рисунке 2.3.

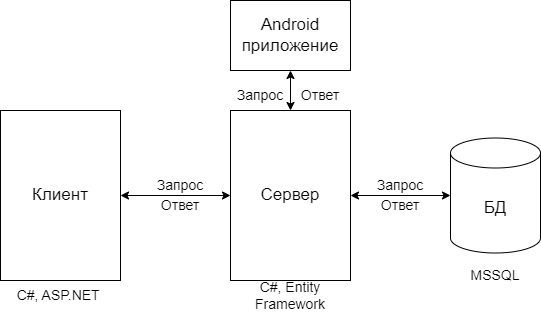
****

Рисунок 2.3 – Общая схема приложения

Примечание к рисунку 2.3: для практически всех операций, исключая операцию просмотра прейскуранта, требуется авторизация пользователя в системе. Если пользователь не авторизован, ему будет предложено произвести регистрацию.

***2.3.2*** В понятие структуры программы включается состав и описание связей всех модулей, которые реализуют самостоятельные функции программы и описание носителей вводимых и выводимых данных, а также данных, участвующих в обмене между отдельными подпрограммами.

Для разработки больших и сложных программ программисту необходимо овладеть специальными приемами получения рациональной структуры программы, которая обеспечивает почти двукратное сокращение объема программирования. Подчиненность модулей программы отражается в схеме иерархии. Однако последняя не отражает порядок их вызова или функционирование программы. Помимо этого, выделение структурной схемы позволяет снизить порог опыта, необходимого разработчику для реализации продукта, так как данная операция способна помочь разработчику с меньшим опытом совершать меньше ошибок при разработке, чем если бы он выполнял планирование непосредственно в процессе написания программного кода.

Структурная схема разрабатываемого приложения представлена на рисунке 2.4.

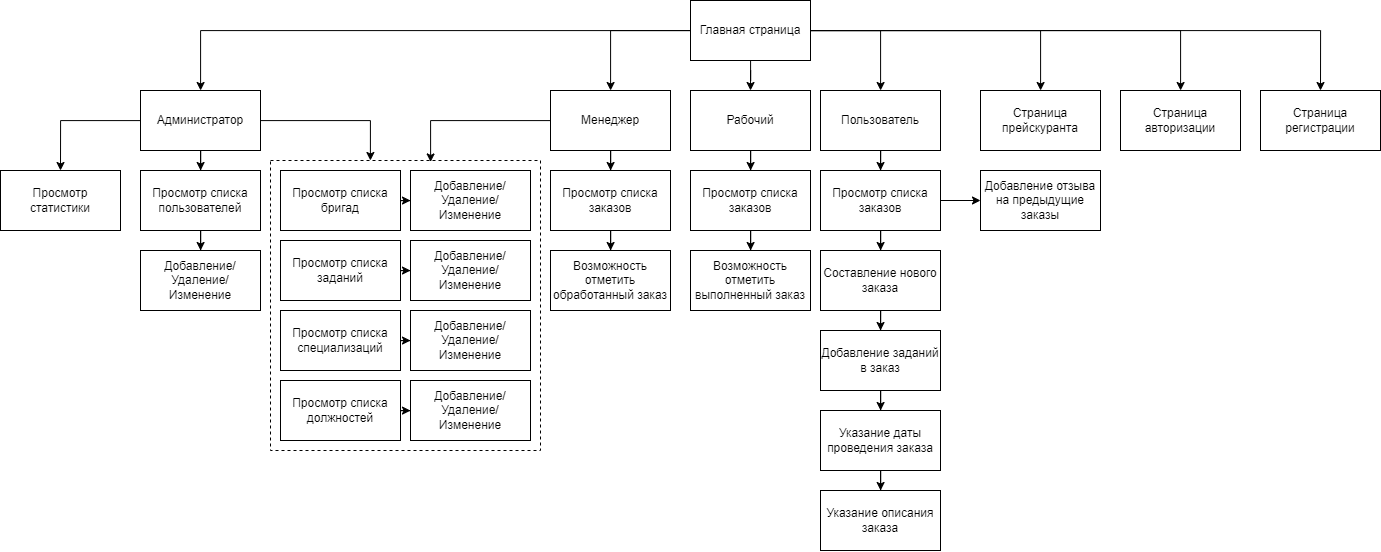
****

Рисунок 2.4 – Структурная схема приложения

***2.3.3*** Для обеспечения безопасности было использовано решение, которое предоставляет фреймворк *ASP.NET Core*, а именно *ASP.NET Core Identity*. *ASP.NET Core Identity* **–** это *API*, поддерживающий функцию входа в пользовательский интерфейс, позволяет управлять пользователями, паролями, данными профиля, ролями, утверждениями, маркерами, подтверждением электронной почты и другим. Пользователи могут создать учетную запись с данными для входа, хранящимися в *Identity* или использовать внешний поставщик входа. Поддерживаемые внешние поставщики входа включают *Facebook*, *Google*, учетную запись Майкрософт и *Twitter* [10]. Выделим основные элементы контекста:

* *IdentityDbContext* - для работы с базой данных *ASP.NET Identity* использует контекст данных, который наследуется от класса *IdentityDbContext* из пространства имен *Microsoft.AspNetCore.Identity.EntityFrameworkCore*;
* *IdentityUser* – В *ASP.NET Core Identity* пользователь представлен классом *IdentityUser* из пространства имен *Microsoft.AspNetCore.Identity.EntityFrameworkCore*, этот класс предоставляет базовую информацию о пользователе с помощью следующих свойств: *Id* – уникальный идентификатор пользователя, *UserName* – ник пользователя, *Email* – электронный адрес пользователя, *Logins* – коллекция логинов, которые использовались пользователем для входа через сторонние сервисы (*Google*, *Facebook* и так далее), *Claims* – коллекция клеймов или дополнительных объектов, которые используются для авторизации пользователя, *PasswordHash* – хеш пароля. В базе данных напрямую не хранится пароль, а только его хеш, *Roles* – набор ролей, к которым принадлежит пользователь, *PhoneNumber* – номер телефона, *SecurityStamp* – некоторое специальное значение, которое меняется при смене аутентификационных данных, например, пароля, *AccessFailedCount* – количество неудачных входов пользователя в систему, *EmailConfirmed* – подтвержден ли адрес электронной почты, *PhoneNumberConfirmed* – подтвержден ли номер телефона;
* *UserManager* – как правило, для управления пользователями используется не контекст данных, а специальный класс - *UserManager*<*T*> из пространства имен *Microsoft.AspNetCore.Identity*, основные из его методов и свойств: *ChangePasswordAsync*(*user*, *old*, *new*) – изменяет пароль пользователя, *CreateAsync*(*user*) – создает нового пользователя, *DeleteAsync*(*user*) – удаляет пользователя, *FindByIdAsync*(*id*) – ищет пользователя по *id*, *FindByEmailAsync*(*email*) – ищет пользователя по *email*, *FindByNameAsync*(*name*) – ищет пользователя по нику, *UpdateAsync*(user) – обновляет пользователя, *Users* – возвращает всех пользователей, *AddToRoleAsync*(*user*, *role*) – добавляет для пользователя *user* роль *role*, *GetRolesAsync* (user) – возвращает список ролей, к которым принадлежит пользователь user, *IsInRoleAsync*(*user*, *name*) – возвращает true, если пользователь *user* принадлежит роли *name*, *RemoveFromRoleAsync*(*user*, *name*) – удаляет роль *name* у пользователя *user*;
* *IdentityRole* – по умолчанию роль в *ASP.NET Core Identity* представлена классом *IdentityRole*, который определяет три свойства: *Id* – идентификатор роли, *Name* – название роли, *Users* – коллекция объектов *IdentityUserRole*, через которые пользователи ассоциированы с данной ролью;

2.3.4 Для создания *Android*-приложений наиболее востребовано и оправдано использование библиотек, фреймворков и обращение к готовым *API* для уменьшения срока разработки приложения.

В основе архитектуры активностей и фрагментов приложения лежит использование шаблона проектирования *MVVM* (от английского – *Model* – *View* – *ViewModel*). Использование данного архитектурного шаблона позволяет данным активности и её операциям быть менее подверженными потерям в результате изменения состояния жизненного цикла активности. Пример схемы работы архитектуры *MVVM* в *Android* приложении представлен на рис. 2.5.

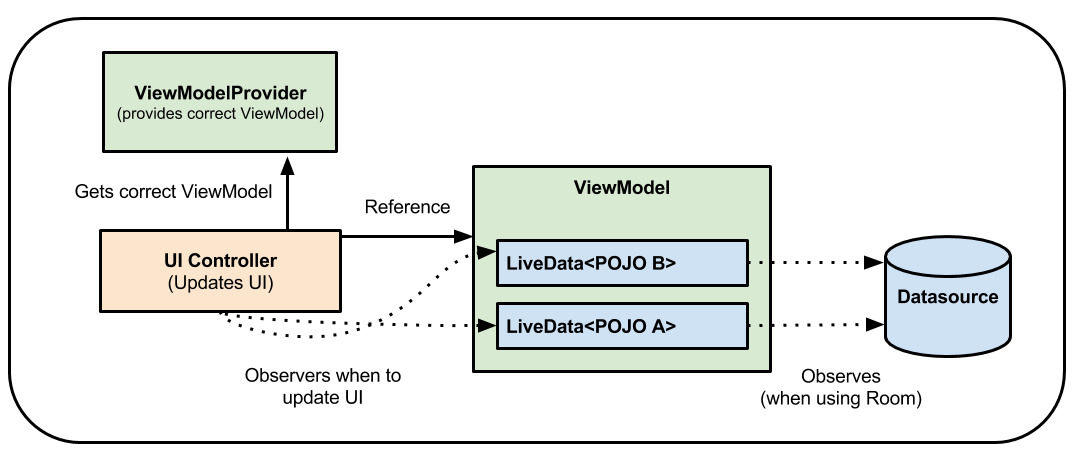


Рисунок 2.5 – Пример схемы работы архитектуры *MVVM* в *Android* приложении

*ViewModel* (в контексте используемого набора библиотек *Android* *Architecture* *Components*) – это объекты, которые предоставляют данные для UI компонентов, таких как активность или фрагмент. Они независимы от жизненного цикла *LifecycleOwner’a* (владельца жизненного цикла). Реализация класса такого объекта осуществляется путём наследования от класса *ViewModel*. Компонент (активность или фрагмент) в процессе работы запрашивает соответствующий ему *ViewModel*-объект у *ViewModelProvider* и, если объект не существует в памяти, создаёт объект, иначе берёт объект из памяти. Жизненный цикл объекта *ViewModel* (и, как следствие, его наследников) приведён на рис. 2.6.

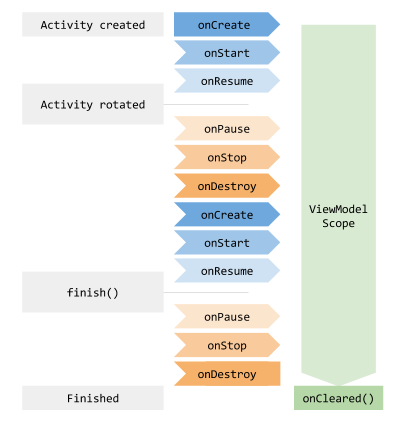


Рисунок 2.6 – Жизненный цикл объекта *ViewModel* из библиотеки *Android* *Architecture* *Components* в *Android* приложении

Пояснение к рисунку 2.6: слева направо расположены следующие логические части: состояние компонента, жизненный цикл *LifecycleOwner*-компонента и жизненный цикл объекта *ViewModel*. Как можно видеть из данного рисунка, объект *ViewModel* уничтожается только после уничтожения *LifecycleOwner*-объекта.

Все экраны приложения (или их составляющие) содержат в себе функционал привязки данных (*Data* *Binding*). Данный функционал позволяет обращаться к элементам экрана из кода более лаконичным и удобным образом, устанавливать значения переменных с использованием кода макета. Данный подход позволяет избавиться от *boilerplate*-кода и устанавливать значения переменных наиболее лёгким способом для программиста. Такой подход используется в местах, где нужно показывать данные.

Передача данных между активностями осуществляется с помощью механизма передачи аргументов при инициализации *Intent*-а (в русской локализации – «Намерение») на запуск некоторой активности. *Intent* – это абстрактное определение намерения на запуск активности. При инициализации объекта *Intent* есть возможность установить аргументы в формате «Ключ-значение». С помощью данного механизма и осуществляется передача аргументов.

Взаимодействие с сервером осуществляется при помощи класс *HttpClient*, который находится в библиотеке *Apache*. Для выполнения запроса необходимо в дополнение создать класс *HttpGet* для отправки *GET* запроса или *HttpPost* для

отправки *POST* запроса. Конструктор классов *HttpGet* и *HttpPost* принимают *URL*. После создания одного из объектов, вызвав метод *execute* у класса *HttpClient* и передав в параметр *HttpGet* или *HttpPost*, мы получаем объект класса *HttpResponse*, где мы можем узнать ответ от сервера. Для выполнения запросов на сервер реализованы специальные классы, которые наследуются от *AsyncTask*. Сделано это с той целью, чтобы остальные функции *android* не ждали ответа от сервера и функционировали независимо от этого.

# **3 РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

## **3.1 Описание реализации программного комплекса**

Данное программное обеспечение представляет собой комплекс серверной и клиентских частей. Взаимодействие клиентов и сервера происходит по протоколу *HTTPS*. В отличие от простого *HTTP*, имеет защиту *SSL*-сертификатом (рисунок 3.1). Это повышает безопасность личных данных, а также на уровень доверия к сайту как со стороны поисковиков (сайты, поддерживающие *HTTPS*, ранжируются выше в результатах поиска), так и со стороны пользователей [11].

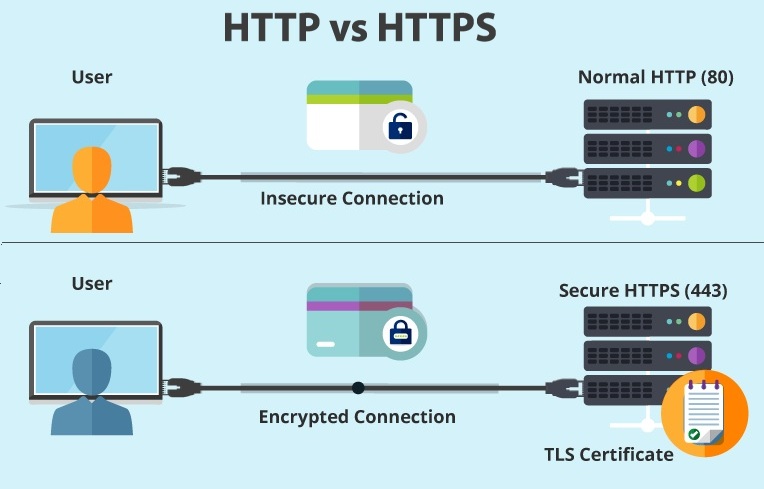


Рисунок 3.1 – *HTTP* и *HTTPS* протоколы

Сервер доступен посредством запросов через сеть Интернет, поэтому для корректного пользования клиентскими приложениями необходимо наличие стабильного соединения с Интернетом.

## **3.2 Проект базы данных**

Помимо основных проектов (серверного и клиентского), в решении присутствует проект базы данных, состоящий из *SQL* скриптов для создания таблиц, связей между ними посредством внешних ключей, а также заполнения базы данных стартовыми данными (роли, данные аккаунтов по каждой из ролей и, непосредственно, связь их с ролями, создание начальных бригад, специализаций и рабочих позиций). Структура проекта приведена на рисунке 3.2.

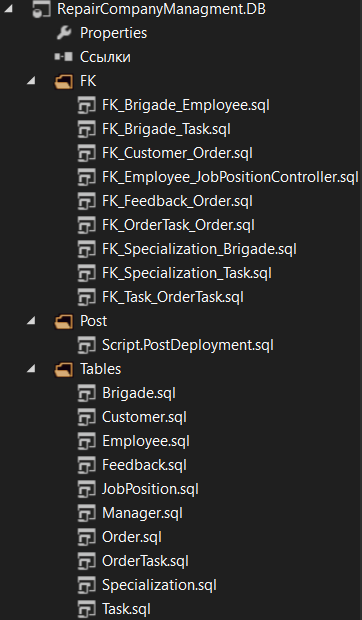


Рисунок 3.2 – Структура проекта базы данных

Перед использованием проекта следует развернуть данную базу данных. В разработанном проекте база данных была успешно локально развернута.

## **3.3 Серверная часть программного комплекса**

***3.3.1*** Серверная часть программного комплекса представляет собой трехслойное *RESTfull* *Web* *API* приложение, разработанное с использованием технологии *ASP*.*NET* *Core*. Приложение разделяется на три логические составляющие, каждая из которых имеет свою ответственность. Таким образом, серверная часть имеет слой доступа к данным, бизнес-логики и слой представления. Преимуществом данного подхода обеспечивает приложению расширяемость и заменяемость, под заменяемостью подразумевается то, что при необходимости, можно заменить слой доступа данных, на другой, использующий иной провайдер данных. Общение между слоями происходит посредством специальных объектов обмена данными – *DTO* [12].

Для проецирования (*маппинга*) объектов одного слоя в *DTO* и обратно, в разработанной серверной части была использована библиотека *AutoMapper*. Библиотека позволяет переносить данные между одноименными полями или по правилам переноса, которые задаются вручную.

Схематичная схема трехслойного приложения представлена на рисунке 3.3.

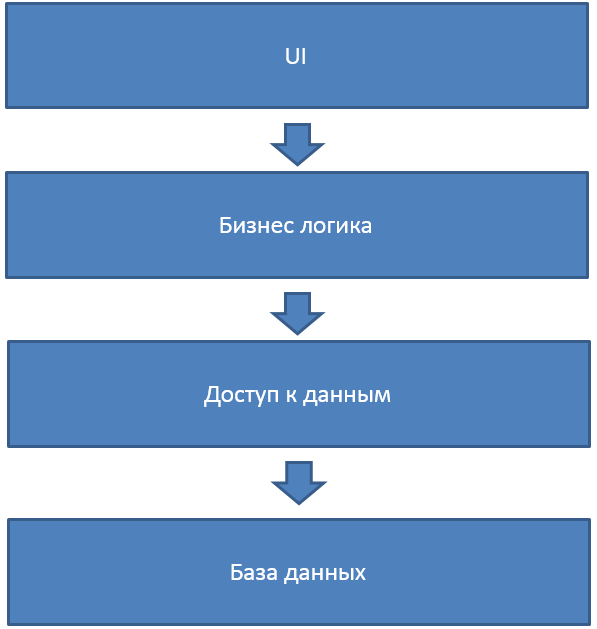


Рисунок 3.3 – Трехслойная архитектура

Для управления доступом к данным используется паттерн "Репозиторий". Он позволяет выполнять основные операции над данными (создание, чтение, изменение и удаление) без зависимости от провайдера данных. Для работы с базой данных *SQL* используется технология *ORM Entity Framework Core*. Она позволяет работать с таблицами, индексами и ключами как с обычными объектами в рамках объектно-ориентированного программирования.

В слое бизнес-логики имеется аналогия с сервис-ориентированной архитектурой, так как он включает в себя набор классов-сервисов, которые управляют репозиториями доступа к данным и выполняют различные дополнительные операции, такие как фильтрация и объединение.

Слой представления является набором конечных точек в контроллерах, которые обеспечивают корректную обработку *HTTP-*запросов различных видов (*GET*, *PUT*, *POST*, *DELETE* и т.д.).

***3.3.2***\_Согласно [10], Внедрение зависимостей (*DI*) – это механизм, который позволяет создавать слабосвязанные объекты в приложении, что делает их более гибкими и расширяемыми. Контейнеры зависимостей используются для настройки зависимостей, и в *ASP*.*NET* *Core* встроен контейнер *Microsoft*.*Extensions*.*DependencyInjection*. Он поддерживает три типа жизненного цикла: *Transient*, *Scoped* и *Singleton*. *Transient* создает новый объект сервиса при каждом обращении к нему, *Scoped* создает один объект сервиса для каждого запроса, а *Singleton* создает один объект сервиса и использует его для всех последующих запросов. Для создания каждого типа сервиса используются соответствующие методы *AddTransient*(), *AddScoped*() и *AddSingleton*().

***3.3.3*** [*ASP*.*NET*](https://asp.net/) *Core* имеет модульную структуру веб-приложений, которая позволяет разработчикам настраивать конвейер обработки запросов и использовать различные компоненты *middleware*. Один из таких компонентов – *Authentication*, который в сочетании с *API*-интерфейсом *Identity* предоставляет широкие возможности для управления доступом пользователей к различным частям веб-приложения по ролям. В данном приложении *Identity* используется для защиты конечных точек с функционалом администратора. Для авторизации используется стандарт *JWT*, который отправляется клиентскими приложениями на серверный компонент для проверки. При успешной авторизации сервер присылает *JWT* токен, который используется для доступа к функциям и данным, которые требуют авторизации пользователя. Клиентское приложение хранит токен как *cookie* или *SharedPreferences* в случае андроид-приложения.

Структура разработанного серверного приложения приведена на рисунке 3.4.

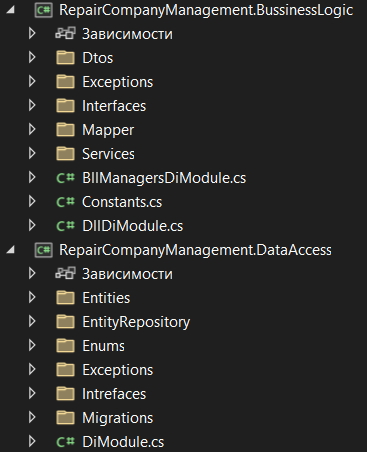
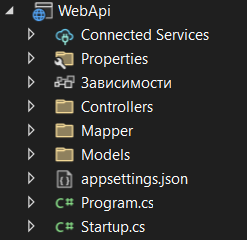


Рисунок 3.4 – Структура серверного приложения

## **3.4 Клиентские приложения**

***3.4.1*** Веб-приложение на клиентской стороне создано с использованием [*ASP.NET*](https://asp.net/) *Core* на платформе .*NET* и основано на паттерне *MVC*, включающем классы моделей, представлений и контроллеров. Структура проекта приведена на рисунке 3.5.

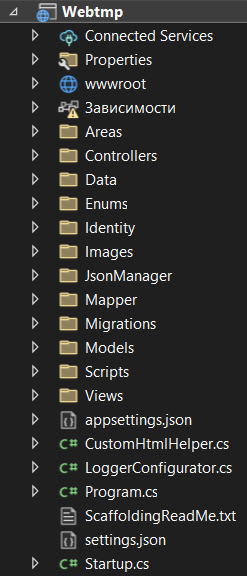


Рисунок 3.5 – Структура клиентского веб-приложения

Так для незарегистрированного пользователя доступна лишь главная страница *c* регистрацией и авторизацией и страница с прейскурантом. Главная страница для незарегистрированного пользователя представлена на рисунке 3.6.

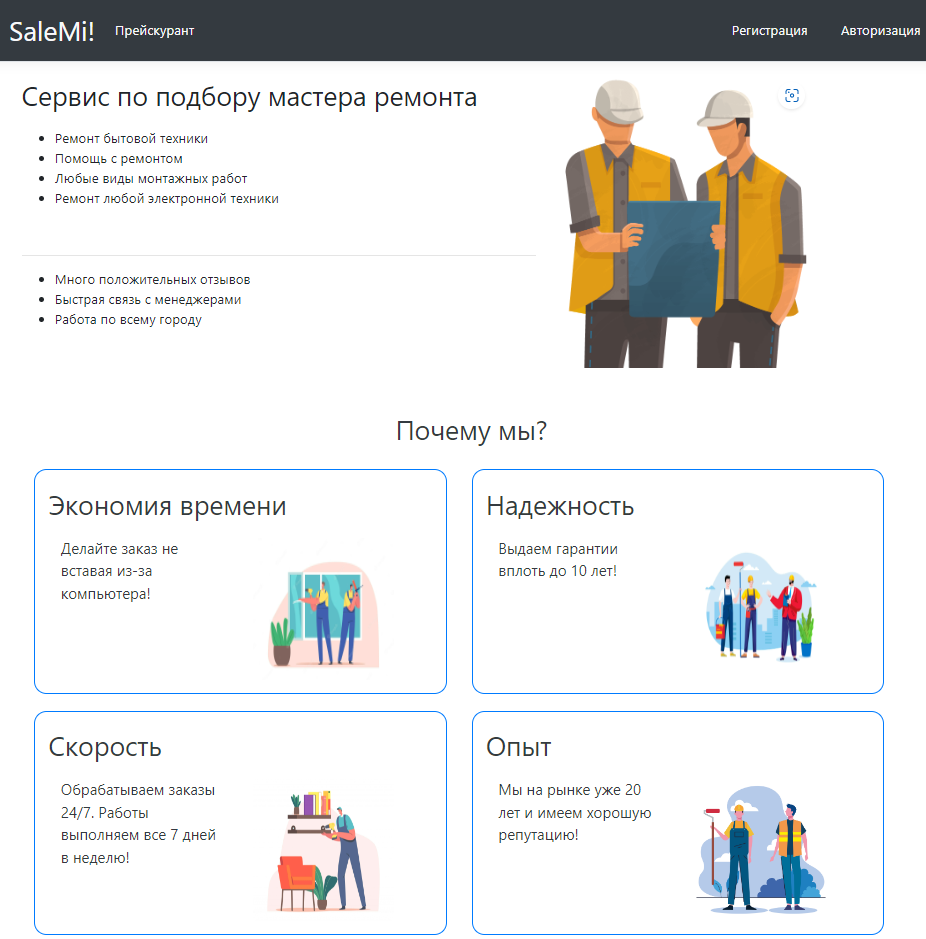


Рисунок 3.6 – Главная страница для незарегистрированного пользователя

Прейскурант представляет собой список задач, который может выполнять компания, список состоит из названий задач и диапазона цен, если услугу предоставляет одна бригада, то цена в диапазоне не пишется. Страница с прейскурантом представлена на рисунке 3.7.

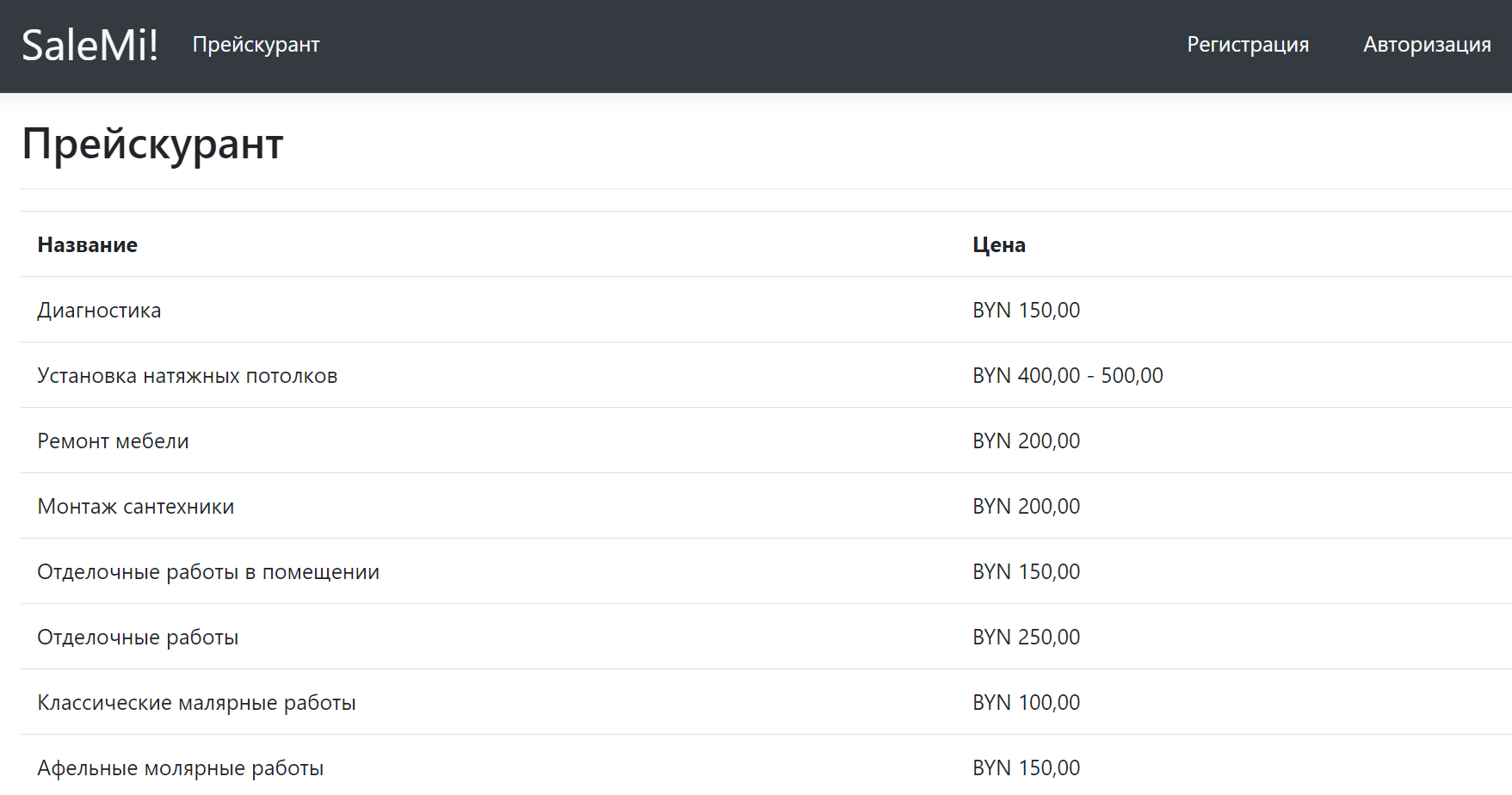


Рисунок 3.7 – Страница с прейскурантом

На главной странице у гостя есть возможность зарегистрироваться, если он новый пользователь или же зайти в свой аккаунт, если он регистрировался ранее. Страница регистрации представлена на рисунке 3.8.

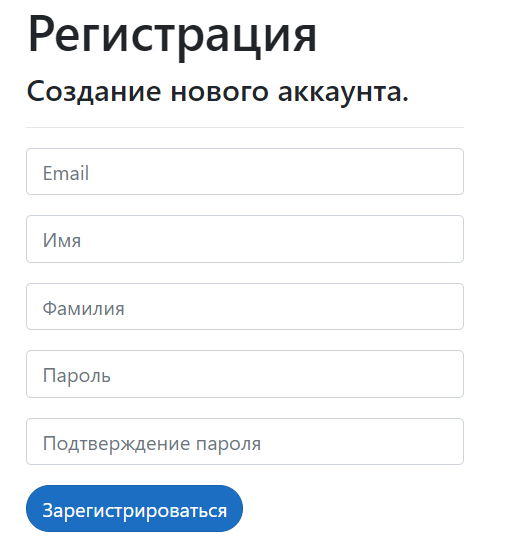


Рисунок 3.8 – Страница регистрации

Страница авторизации представлена на рисунке 3.9.

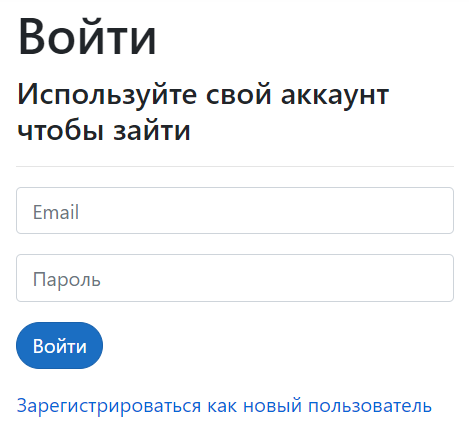


Рисунок 3.9 – Страница авторизации

После авторизации пользователю доступны вкладки «Заказы», «Прейскурант», «Мой аккаунт», «Выйти» и текущий баланс с возможностью его пополнить. На странице «Заказы» пользователь может создать и составить необходимый заказ. Страница «Заказы» представлена на рисунке 3.10.

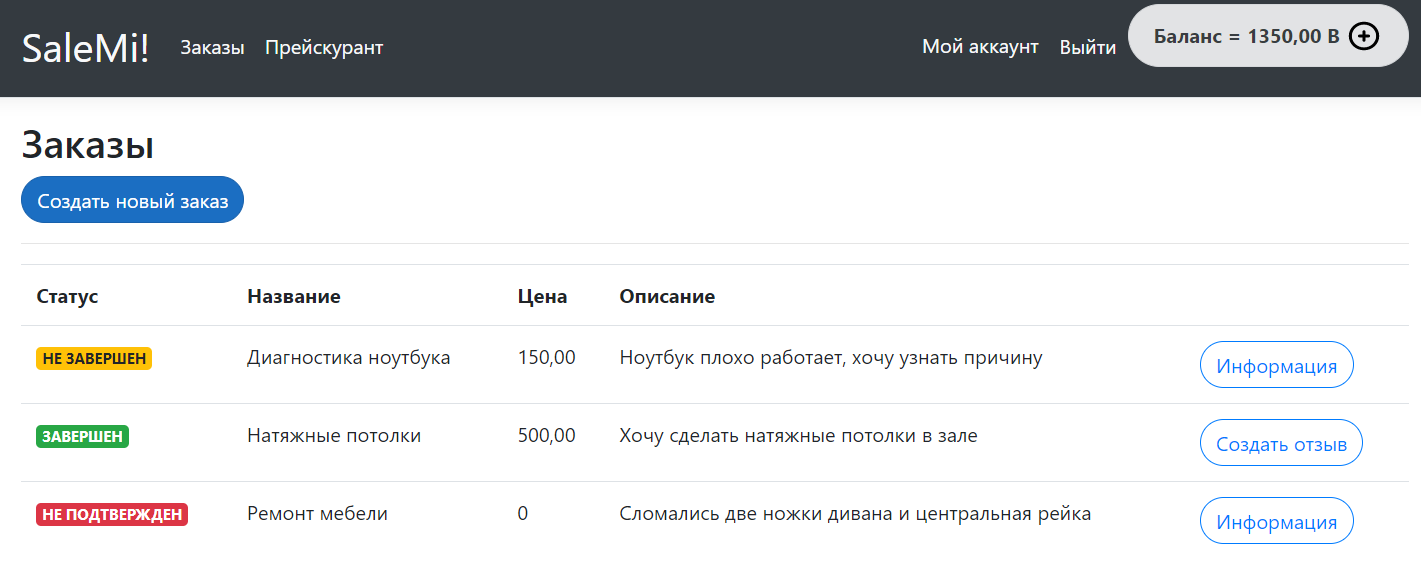


Рисунок 3.10 – Страница «Заказы»

После создания заказа пользователь может его открыть и добавить в заказ необходимые задания, подходящие ему по дате и специализации. Страница подробной информации о заказе представлена на рисунке 3.11.

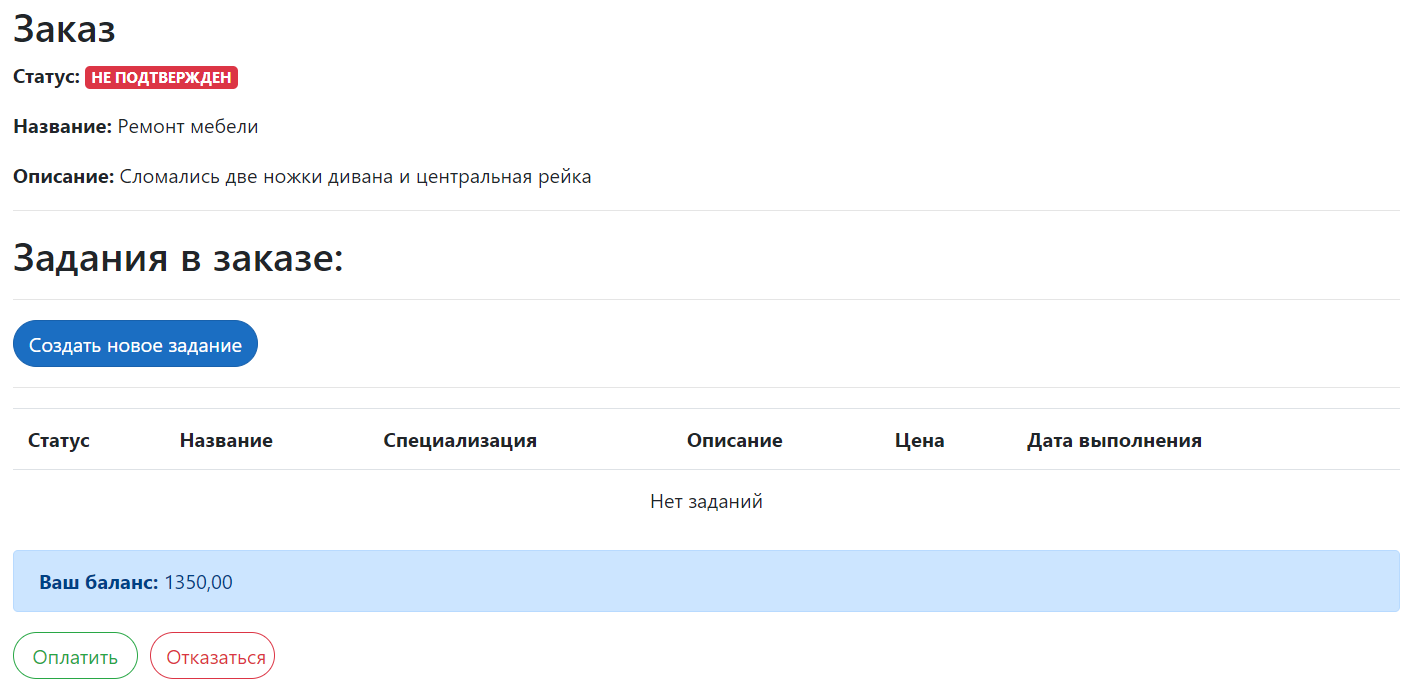


Рисунок 3.11 – Подробная информация о заказе

Пользователь может добавить задания в заказ нажав на кнопку «Создать новое задание», после чего откроется форма для выбора специализации, задания и даты будущего задания. Страницы создания заказа представлены на рисунках 3.12 - 3.14.

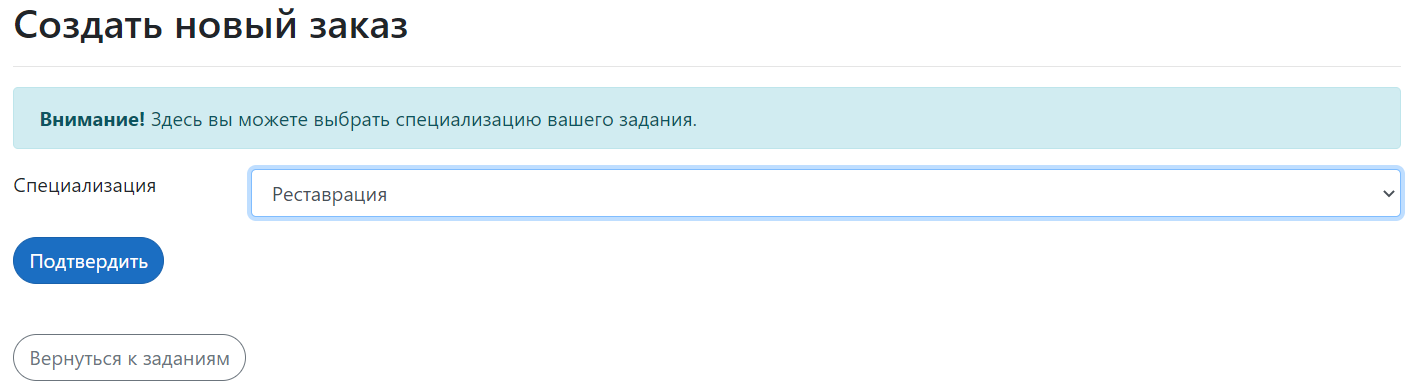


Рисунок 3.12 – Страница с выбором специализации

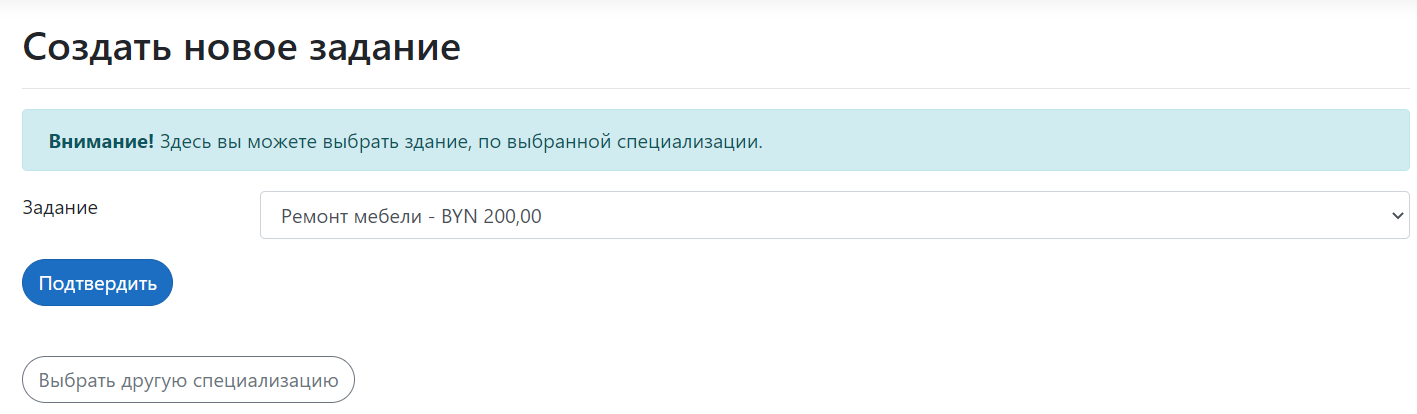


Рисунок 3.13 – Страница с выбором задания по специализации

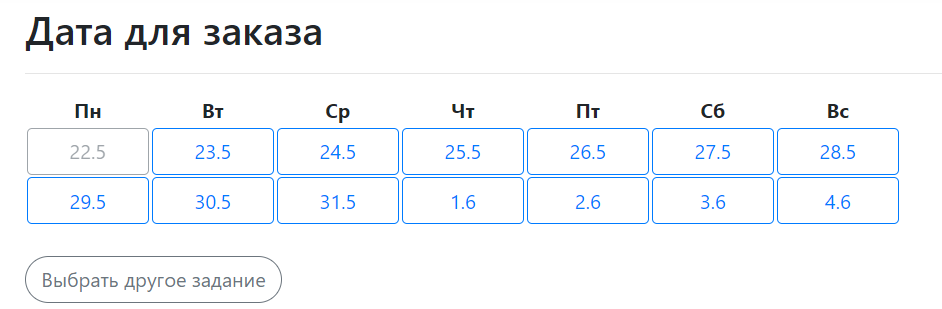


Рисунок 3.14 – Страница с выбором даты для задания

После выбора специализации, задания и подходящей даты, пользователю необходимо подтвердить корректность составленного задания. Страница подтверждения представлена на рисунке 3.15.

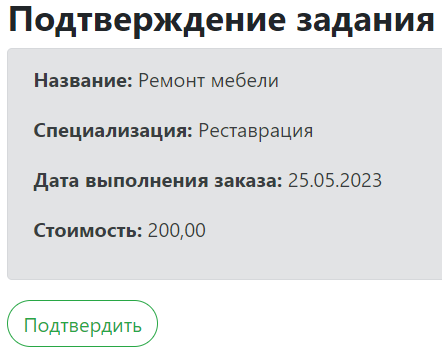


Рисунок 3.15 – Страница подтверждения задания

После подтверждения задания, оно появится в созданном ранее заказе. Далее пользователю необходимо оплатить заказ, чтобы его заказ попал на рассмотрение менеджеру. После оплаты, состояние заказа изменится на«Оплачен».

Баланс пользователя так же расположен в навигационной панели сверху. При переходе по кнопке «+» в элементе «Баланс»,пользователю откроется страница для пополнения баланса. Страница для пополнения баланса представлена на рисунке 3.16.

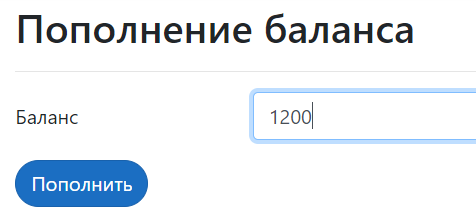


Рисунок 3.16 – Страница для пополнения баланса

Пользователю так же доступен его личный кабинет, в котором он может:

* добавить мобильный номер*;*
* изменить свой электронный адрес;
* изменить свой пароль*;*
* добавить двухфакторную авторизацию.

Личный кабинет пользователя представлен на рисунке 3.17.

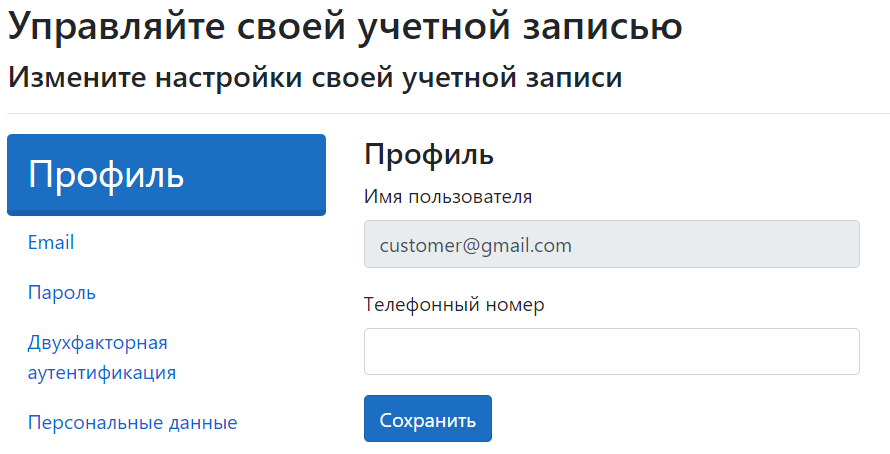


Рисунок 3.17 – Личный кабинет пользователя

После того как оплаченный заказ поступил в систему, он будет отображаться у менеджера во вкладке «Заказы». Там он сможет посмотреть информацию о пользователе, который сделал заказ и о заданиях, которые присутствуют в заказе. На основе этих данных менеджер может как принять заказ, так и отклонить его. Страница менеджера «Заказы» представлена на рисунке 3.18.

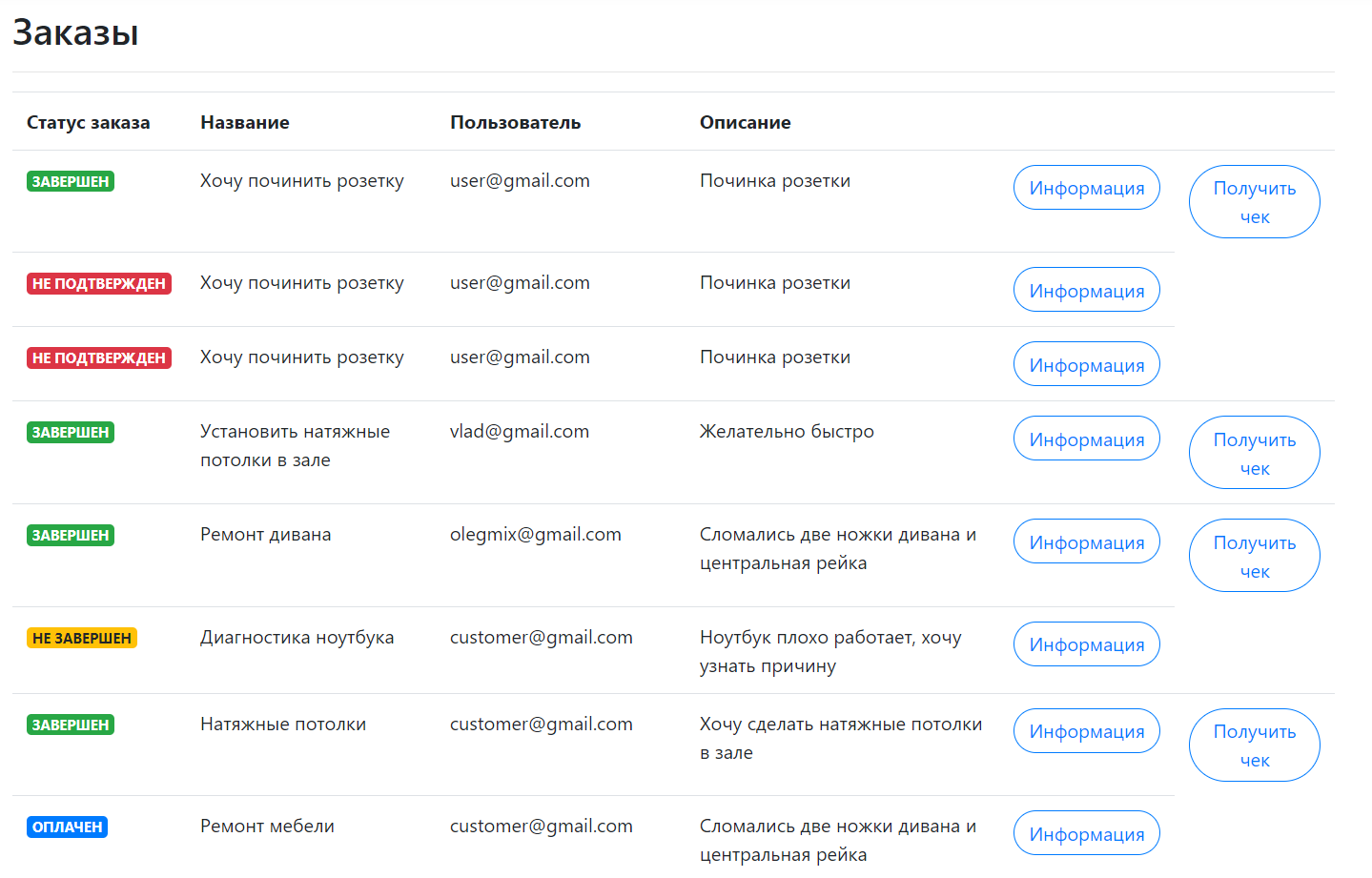


Рисунок 3.18 – Страница «Заказы»

После подтверждения заказ переходит в состояние «Не завершен» и отображается у рабочих той бригады, которая была указана в заказе.

У менеджера так же есть доступ к информации о специализациях, заданиях, бригадах, должностях. Он может изменять, добавлять и удалять все ранее перечисленные категории. Страница специализаций представлена на рисунке 3.19.

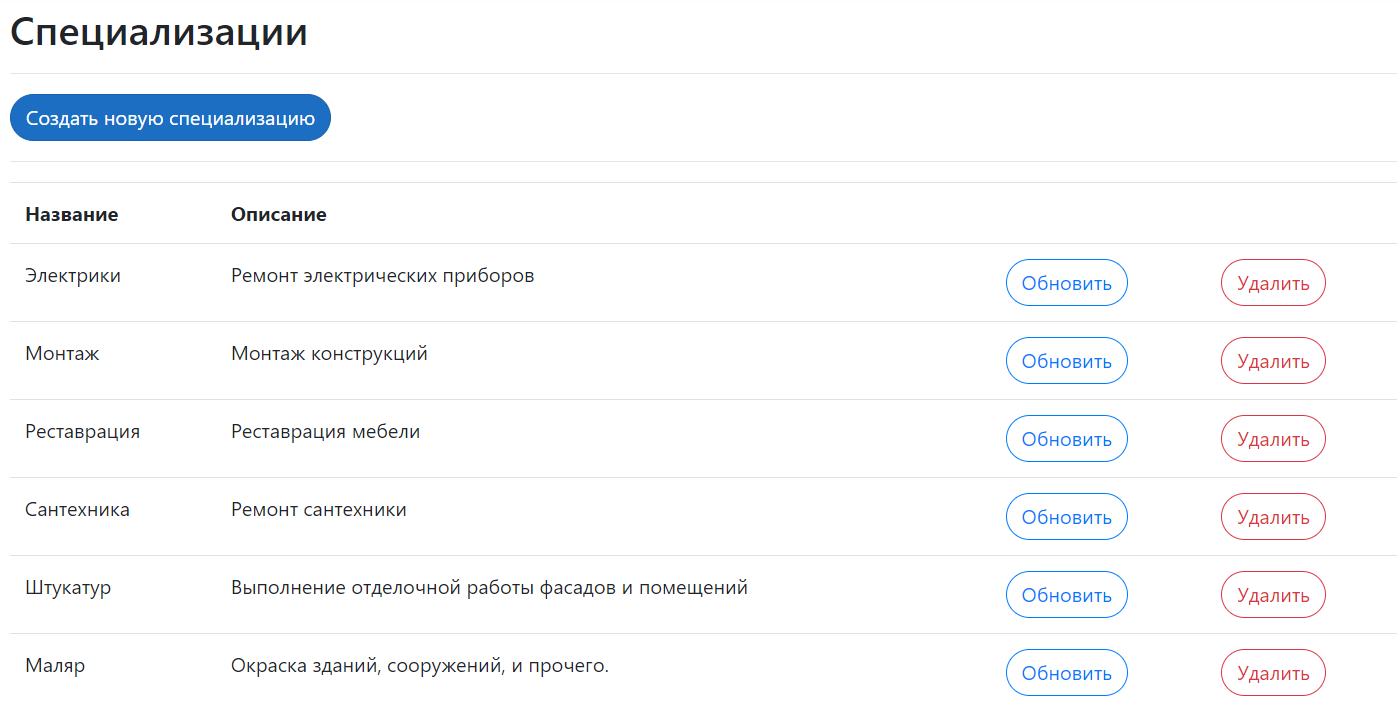


Рисунок 3.19 – Страница специализаций

При нажатии на кнопку «Создать новую специализацию», будет открыта страница для ввода необходимой информации для создания новой специализации. Страница для создания специализации представлена на рисунке 3.20.

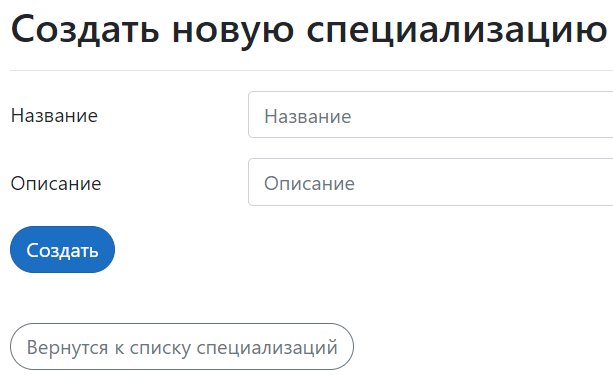


Рисунок 3.20 – Страница для создания специализации

После ввода необходимых данных и их подтверждения, новая специализация появится в списке специализаций.

При нажатии кнопки «Обновить», откроется страница для обновления существующей специализации. Страница для обновления представлена на рисунке 3.21.

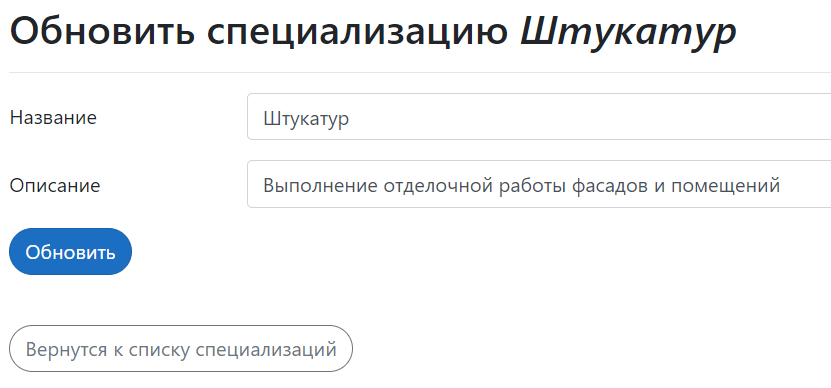
**

Рисунок 3.21 – Страница для обновления

После ввода необходимых данных и их подтверждения, обновленная специализация появится в списке специализаций.

При нажатии кнопки «Удалить» появляется форма подтверждения, где отображаются данные по удаляемой специализации. Страница удаления представлена на рисунке 3.22.

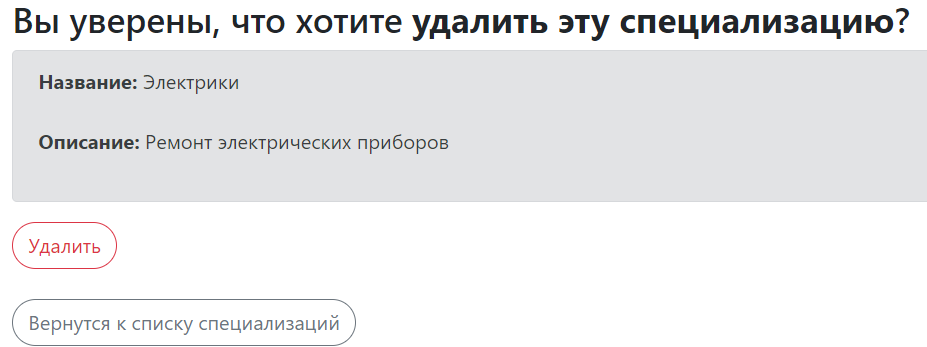


Рисунок 3.23 – Страница удаления

Страницы с управлением справочниками «Задания», «Бригады» и «Рабочие позиции» имеют такой же вид вид и функционал.

После подтверждения заказа, этот заказ появится у работников, выбранной бригады, во вкладке «Список задач», где у них есть возможность увидеть информацию о заказе и пользователе и отметить заказ как выполненный, после его выполнения. Страница «Список задач» представлена на рисунке 3.24.

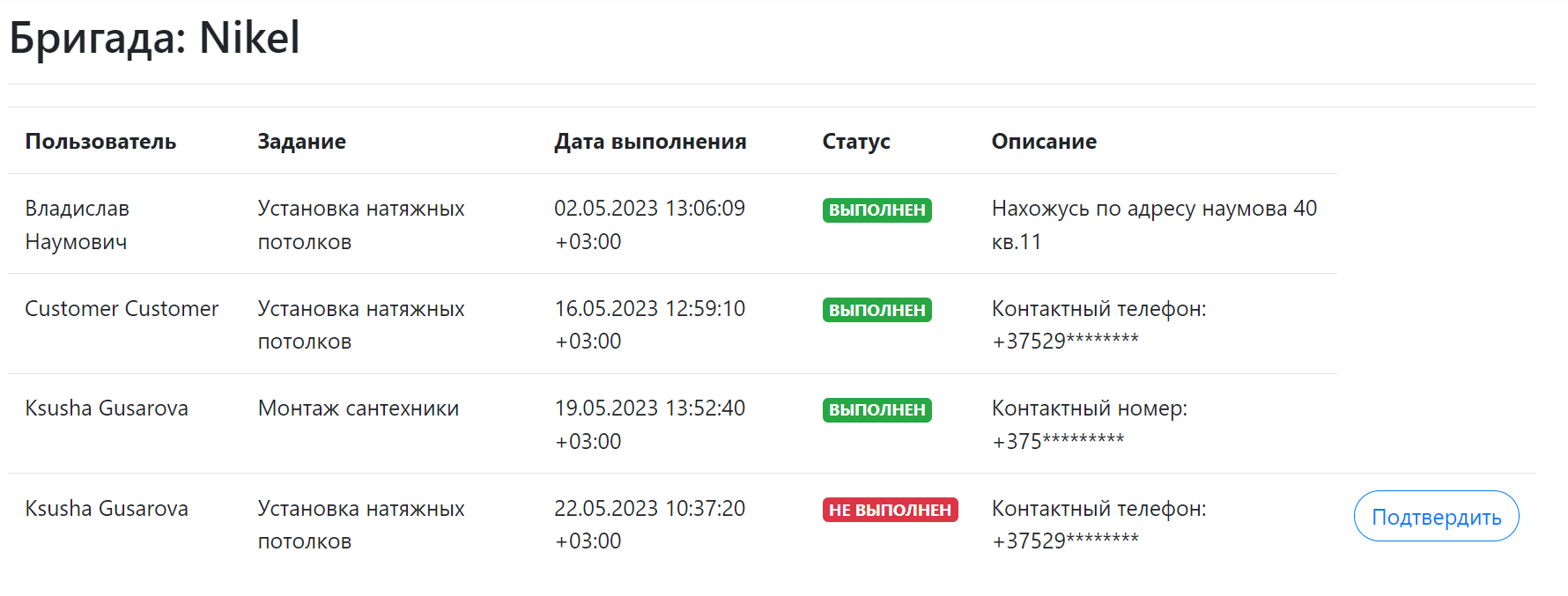


Рисунок 3.24 – Страница «Список задач»

После подтверждения заказа, его статус изменится на «Выполнен». И у пользователя появится возможность оставить отзыв, который будет виден на главном экране, а также получить чек этого заказа. Пример чека представлен на рисунке 3.25.

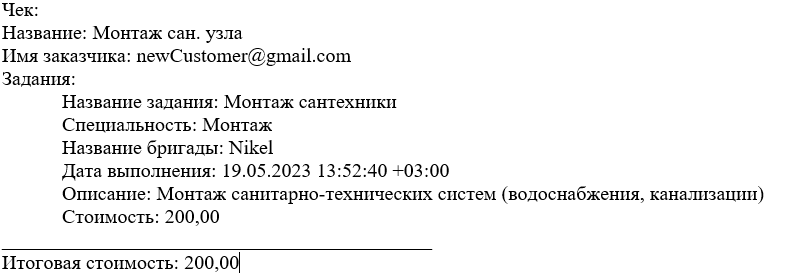


Рисунок 3.25 – Пример чека по заказу

В приложении, так же, есть роль администратора, который администрирует пользователей и обладает схожими возможностями с менеджером, за исключением обработки заказов. Администратору доступна возможность изменять роли пользователям, а, так же, добавлять новые роли. Страница с изменением ролей пользователей представлена на рисунке 3.26.

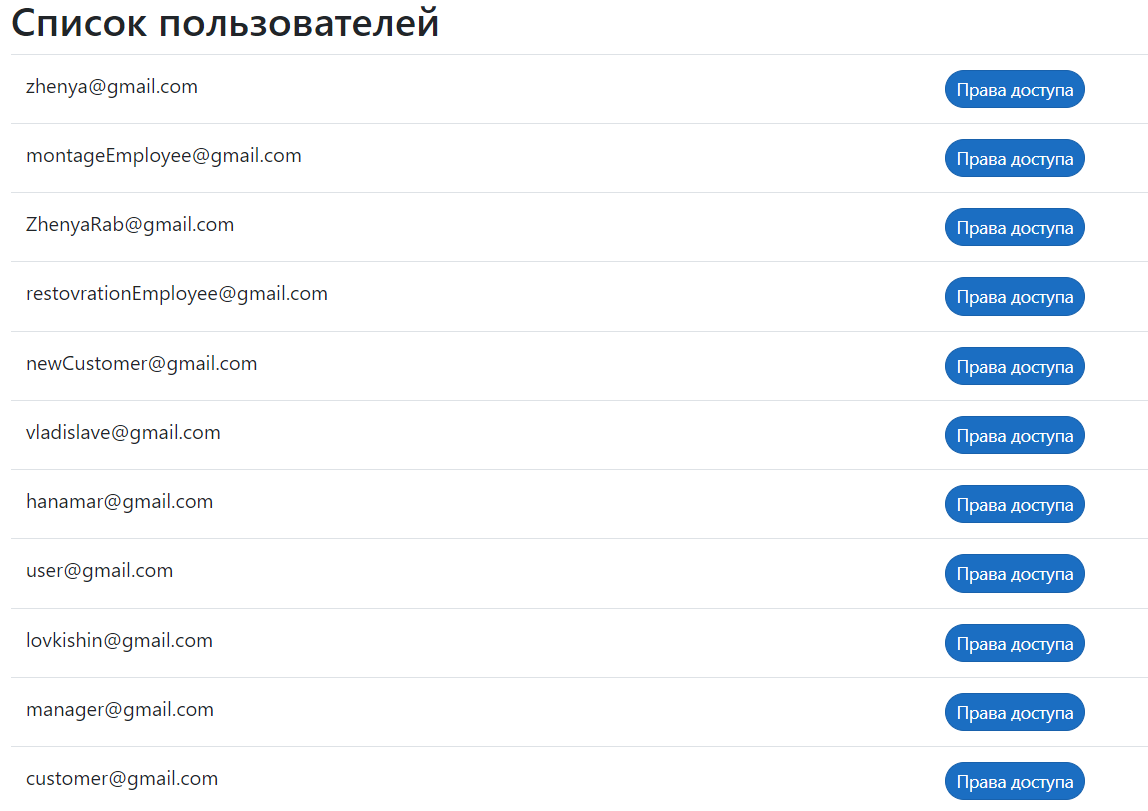
**

Рисунок 3.26 – Страница с изменением ролей пользователей

При переходе по кнопке «Права доступа» администратор может выбрать роль, которую необходимо присвоить пользователю. Страница с выбором ролей представлена на рисунке 3.*2*7.

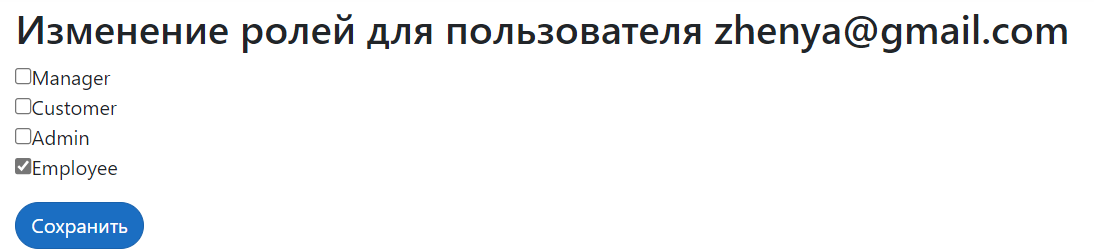


Рисунок 3.27 – Страница с выбором ролей для пользователя

Администратор, так же, может получить статистику по работам бригад, перейдя по вкладке «Статистика», где получит возможность получить годовую статистику и статистику за необходимый месяц. Страницы со статистикой представлены на рисунках 3.28 – 3.29.

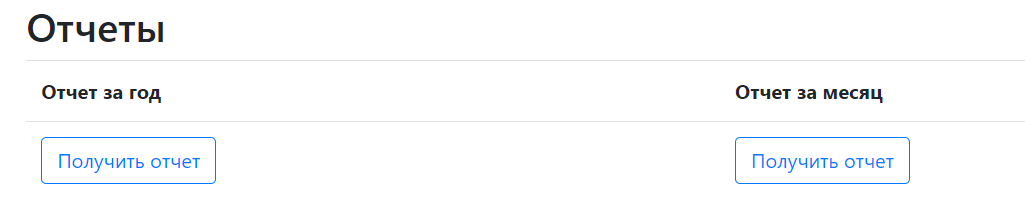


Рисунок 3.28 – Страница с выбором необходимой статистики

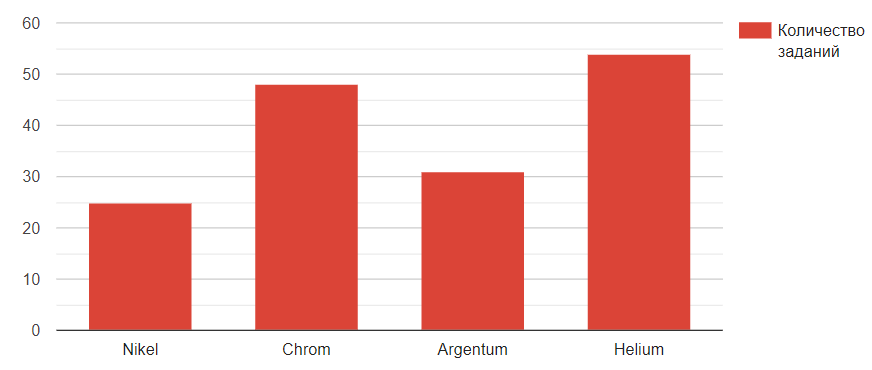


Рисунок 3.29 – Страница с графиком статистики

***3.4.2*** Для создания мобильного приложения был использован язык программирования *Java* и разработано андроид-приложение, которое состоит из нескольких активностей и разметок. Приложение имеет модульную структуру, где разметки используют различные фрагменты для возможности их повторного использования. При разработке было использовано руководство разработчика на *Android*. Для связи с *RESTful* сервером использовалась библиотека *Retrofit* – типобезопасный *HTTP* клиент для *Android* [13].

Андроид-приложение дублирует функционал работника. Список задач работника представлен на рисунке 3.30.

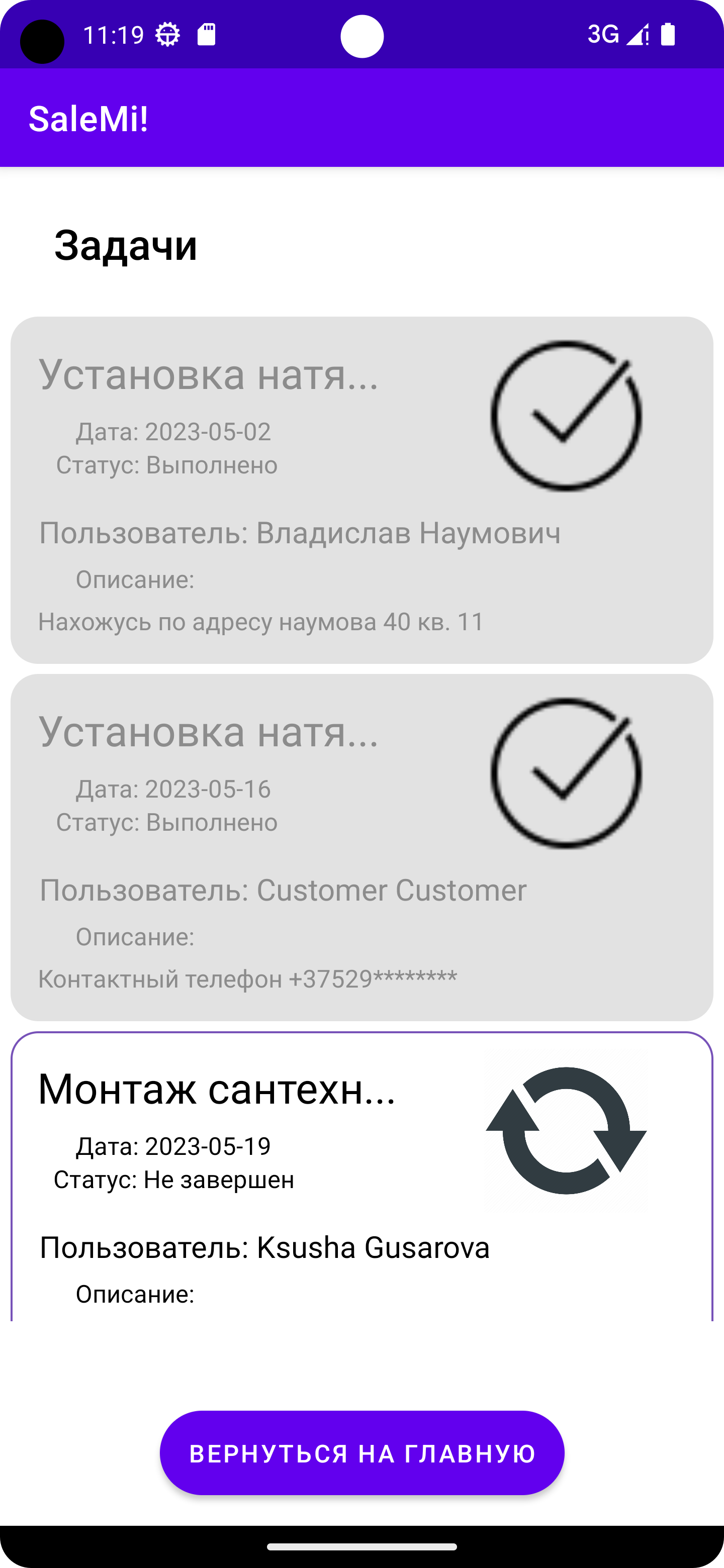


Рисунок 3.30 – Список задач работника в андроид-приложении

Структура андроид-приложения представлена на рисунке 3.31.

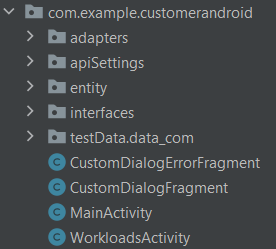


Рисунок 3.31 – Структура андроид-приложения

# **4 ТЕСТИРОВАНИЕ, ВЕРЕФИКАЦИЯ И ВАЛИДАЦИЯ**

## **4.1 Модульное тестирование**

Тестирование ПО - процесс исследования (испытания) ПО, имеющий своей целью проверку соответствия между реальным и ожидаемым поведением ПО на конечном наборе тестов, выбранных определённым образом (согласно ISO/IEC TR 19759:2005). Цель тестирования или цель разработки и выполнения тестов – обеспечить очищение ПО от ошибок до приемлемого уровня (обеспечить 100% покрытие кода тестами, естественно, не представляется возможным, поэтому необходимо в первую очередь исправить очевидные ошибки), убедиться, что ПО отвечает оригинальным требованиям и спецификации.

Согласно [14], модульное (компонентное) тестирование (*unit testing*, *module testing*, *component testing*) направлено на проверку отдельных неболь-ших частей приложения, которые (как правило) можно исследовать изолированно от других подобных частей. При выполнении данного тестирования могут проверяться отдельные функции или методы классов, сами классы, взаимодействие классов, небольшие библиотеки, отдельные части приложения. Часто данный вид тестирования реализуется с использованием специальных технологий и инструментальных средств автоматизации тестирования, значительно упрощающих и ускоряющих разработку соответствующих тест-кейсов.

По уровню детализации приложения (по уровню тестирования) тестирование разделяют на:

* модульное (юнит) тестирование*;*
* интеграционное.

Качественно написанные тесты должны следовать принципам *FIRST:*

* ***f****ast* (быстрый)*;*
* ***i****solated* (изолированные от других тест-кейсов);
* ***r****epeatable* (повторяемые в любом количестве и порядке);
* ***s****elf-validating* (явно определяют, «прошёл» или «не прошёл»);
* ***t****imely/****t****horough* (своевременно написанные, т.е. «свежий код – свежие тесты»; обеспечивают покрытие достаточной функциональности для удовлетворения требований – покрывают все ветки кода).

Юнит тестирование представлено юнит-тестами слоя бизнес-логики, а именно его сервисов: *Brigade*Service, *Order*Service, *Report*Service, *User*Service. Тесты используют объекты библиотеки Moq для имитации пове-дения объектов репозиториев – таким образом, они не используют внешний источник данных.

Юнит тесты проверяют, пустой ли объект отправлен на обработку, а также валидацию у следующих объектов:

* заказ (пустое имя, пустая дата, пустое задание);
* задание (отрицательная цена, пустые поля);
* справочники (пустые или некорректные значения);
* задание работника (отметка о выполненности, раньше даты задания)
* баланс (отрицательное значение)
* пользователи (несколько ролей у пользователя)
* отзыв (пустой текст).

Таким образом, создано 60 юнит теста, структура которых отображена на рисунке 4.1.



Рисунок 4.1 – Юнит тесты в обозревателе тестов

Интеграционное тестирование представлено интеграционными тестами слоя бизнес-логики, а именно его сервисов: *Brigade*Service, *Order*Service, *Report*Service, *User*Service. Данный вид тестирования проверяет корректность работы приложения с внешним источником данных, соединение с тестовой локальной базой данных, идентичной той, что используется в программном комплексе, в данном случае активно используется.

Интеграционные тесты проверяют корректность выполнения операций создания, обновления, удаления сущностей в тестовой базе данных. Каждый тест следует принципам *FIRST*, стоит также отметить, что по завершению выполнения теста происходит возврат тестовой базы данных в исходное состояние.

Таким образом, создано 84 интеграционных теста, структура которых отображена на рисунке 4.2.

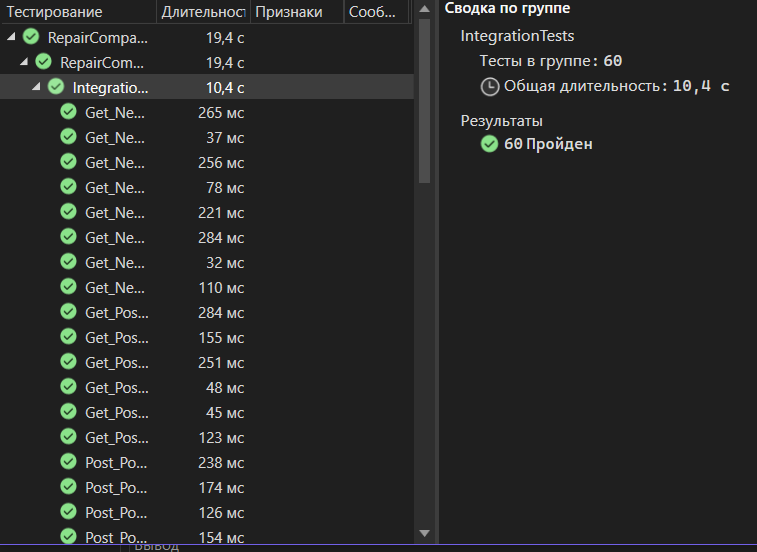


Рисунок 4.2 – Интеграционные тесты в обозревателе тестов

## **4.2 Ручное тестирование программного комплекса**

Все компоненты комплекса, а именно серверный компонент, веб-приложение и андроид-приложение прошли процесс отладки и исправления возможных ошибок. Среди них – некорректный ввод данных в форму – в соответствии с рисунком 4.2; попытка отметить уже выполненное задание – в соответствии с рисунком 4.3.

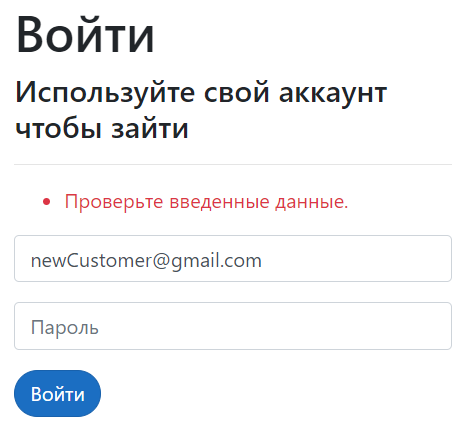


Рисунок 4.2 – Некорректный ввод данных в форму

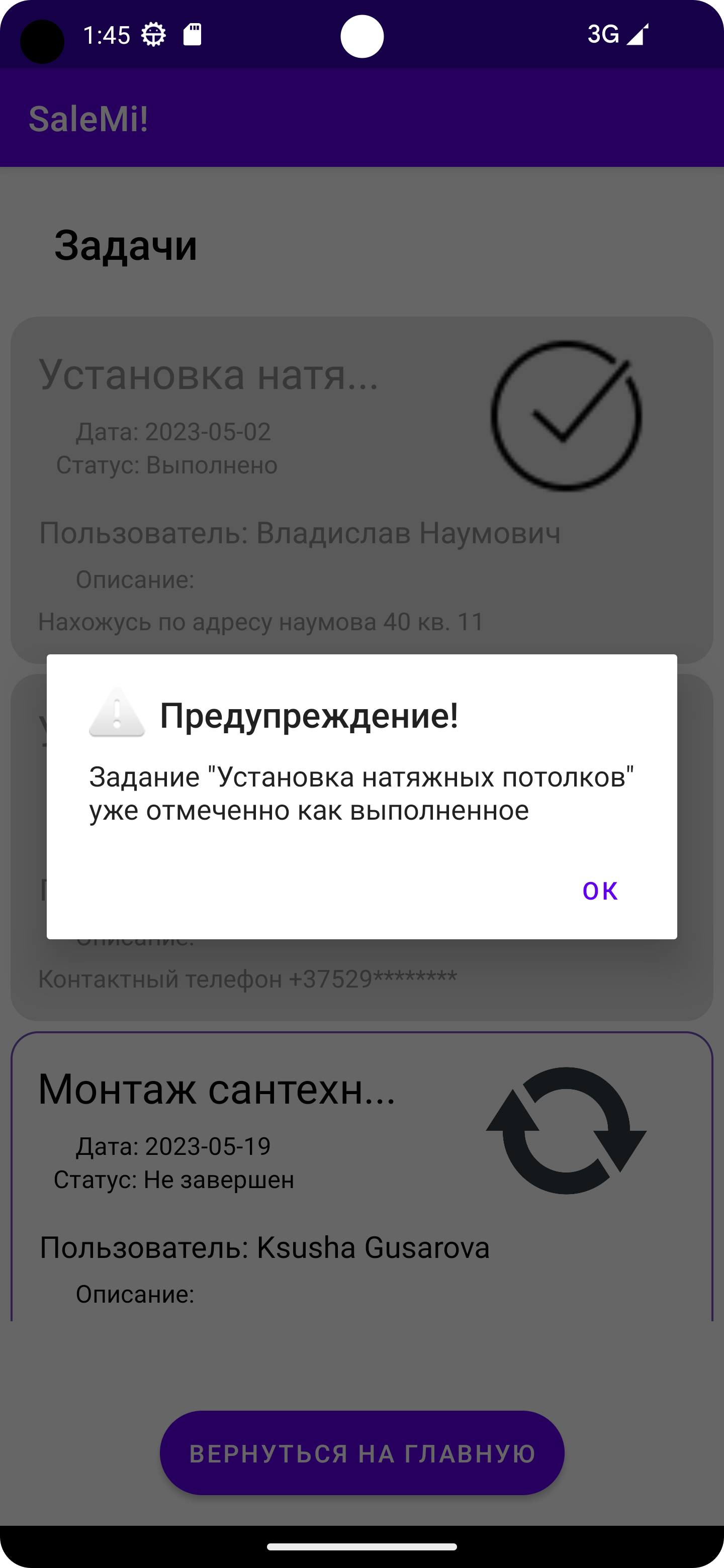


Рисунок 4.3 – Попытка отметить уже выполненное задание

По каждой сущности были успешно проведены ручные тесты основных *CRUD* операций. Пример некорректного ввода в форму при создании сущности приведён на рисунке 4.4.

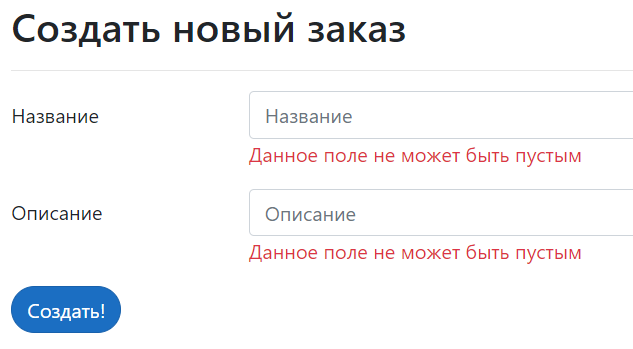


Рисунок 4.4 – Проверка корректности ввода данных в форму

Были разработаны общие для всех сущностей БД тест-кейсы для проверки *CRUD* операций над таблицами, описанные в таблице 4.1.

Таблица 4.1 – Тест-кейсы для проверки *CRUD* операций

|  |  |  |  |
| --- | --- | --- | --- |
| № | Действие | Шаги выполнения | Ожидаемый  результат |
| 1 | 2 | 3 | 4 |
| 1 | Работа с добавлением данных в таблицу БД | 1. Открыть страницу с таблицей элементов  2. Нажать на кнопку добавить  3. Ввести данные в форму  5. Нажать кнопку добавить | Добавление в базу данных произойдет успешно и обновлённые данные будут отображены в таблице на странице в приложении |
| 2 | Работа с редактированием сущности в базе данных | 1. Открыть страницу с таблицей сущностей  2. Выбрать запись в таблице неисправностей, нажать соответствующую ей кнопку  «Обновить»  3. После открытия окна с формой, внести необходимые изменения и нажать кнопку «Обновить» | Запись будет изменена в базе данных и будет отображена в таблице на странице в приложении |
| 3 | Работа приложения при попытке добавления/изменения записи с вводом некорректных данных в БД | 1. Открыть страницу с таблицей сущностей  2. Нажать на кнопку добавить или выбрать запись в таблице неисправностей, нажать соответствующую ей кнопку  «Обновить»  3. В форму ввода ввести некорректные данные  4. Нажать кнопку | Будет выведено сообщение, которое опишет ошибку, которая произошла на стадии добавления |
| 4 | Работа с поиском элементов | 1. Открыть страницу с таблицей сущностей  2. Нажать на форму поиска  3. Ввести нужную информацию по сущности | Сущности с любым совпадением будут отображены в таблице |