МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий

Кафедра Информационных систем и технологий

Специальность 1-40 01 01 «Программное обеспечение информационных

технологий»

Специализация Программирование интернет-приложений

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

WEB-приложение «Торговая площадка»

Выполнил студент Борисов Антон Андреевич

(Ф.И.О.)

Руководитель проекта преп.-стаж. Сенюк В.К.

(учен. степень, звание, должность, подпись, Ф.И.О.)

Заведующий кафедрой к.т.н., доц. Смелов В.В.

(учен. степень, звание, должность, подпись, Ф.И.О.)

Консультанты преп.-стаж. Сенюк В.К.

(учен. степень, звание, должность, подпись, Ф.И.О.)

Нормоконтролер преп.-стаж. Сенюк В.К.

(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Минск 2021

Оглавление

[**Введение** 3](#_Toc72812670)

[**1.** **Постановка задачи** 4](#_Toc72812671)

[**1.1.** **Алгоритмы решения** 4](#_Toc72812672)

[**1.2.** **Обзор прототипа** 5](#_Toc72812673)

[**2.** **Проектирование приложения** 7](#_Toc72812674)

[**2.1.** **Архитектура** 7](#_Toc72812675)

[**2.2.** **Серверная часть приложения** 7](#_Toc72812676)

[**2.2.1.** **Уровни. API, Service, Repository** 8](#_Toc72812677)

[**2.2.2.** **Безопасность. Аутентификация и авторизация** 8](#_Toc72812678)

[**2.3.** **Взаимосвязь всех компонентов** 9](#_Toc72812679)

[**2.4.** **Диаграмма UML** 10](#_Toc72812680)

[**2.5.** **Сущности в базе данных и связи между ними** 12](#_Toc72812681)

[**3.** **Реализация приложения** 14](#_Toc72812682)

[**3.1.** **Серверная часть** 14](#_Toc72812683)

[**3.1.1.** **Конфигурация** 14](#_Toc72812684)

[**3.1.2.** **Реализация REST API. Контроллеры** 14](#_Toc72812685)

[**3.1.3.** **Сервисы.** 15](#_Toc72812686)

[**3.1.4.** **Репозитории. Взаимодействие с БД** 15](#_Toc72812687)

[**3.1.5.** **DTO. Конвертация разных типов объектов** 15](#_Toc72812688)

[**3.1.6.** **JWT. Аутентификация и авторизация** 16](#_Toc72812689)

[**3.1.7.** **Разработка документации** 16](#_Toc72812690)

[**3.1.8.** **Обработка исключений** 17](#_Toc72812691)

[**3.1.9.** **Обмен сообщениями с клиентом в реальном времени** 17](#_Toc72812692)

[**3.2.** **Клиентская часть** 18](#_Toc72812693)

[**3.2.1.** **Хранение состояния** 18](#_Toc72812694)

[**3.2.2.** **Маршрутизация** 19](#_Toc72812695)

[**3.2.3.** **Обмен сообщениями с сервером в реальном времени.** 19](#_Toc72812696)

[**4.** **Тестирование, проверка работоспособности и анализ полученных результатов** 20](#_Toc72812697)

[**5.** **Руководство пользователя** 22](#_Toc72812698)

[**5.1.** **Регистрация пользователя** 22](#_Toc72812699)

[**5.2.** **Аутентификация и авторизация пользователя** 23](#_Toc72812700)

[**5.3.** **Оформление заказа** 24](#_Toc72812701)

[**5.4.** **Аукцион** 25](#_Toc72812702)

[**Заключение** 26](#_Toc72812703)

[**Список используемых источников** 27](#_Toc72812704)

[**Приложение А** 28](#_Toc72812705)

[**Приложение Б** 30](#_Toc72812706)

[**Приложение В** 31](#_Toc72812707)

# 

# **Введение**

В данном курсовом проекте разработано WEB-приложение «Торговая площадка». Логически оно разделено на две части: серверную, написанную на NestJS, и клиентскую написанную на ReactJS. Серверная часть взаимодействует с базой данных и размещена на платформе «Heroku». Клиентская часть приложения взаимодействует с сервером через REST API и размещена на платформе «Netlify».

В современном мире информационные технологии позволяют автоматизировать и упростить работу во многих сферах деятельности человека. Базы данных, интернет, быстрый поиск, автоматическое вычисление данных позволяют упростить продажу вещей в интернете. Это и является целью моего курсового проекта ­– создание простого и удобного программного средства, позволяющего пользователям ознакомиться с ассортиментом торговой площадки, выставить товар на продажу, выбрать товар, положить его в корзину и заказать. Также продавцы на сайте получают доступ к заказам их продуктов, а значит есть возможность организовать доставку товаров.

Я выбрал в качестве темы курсового проекта «Торговая площадка» в связи с активным использованием онлайн площадок для приобретения и продажи товаров. Данное приложение содержит тот функционал, который поможет вам осуществлять покупки, не выходя из дома, тем самым экономить ваше время. А также продавать вещи, которыми вы уже не пользуетесь.

Основная цель курсового проекта: разработка web-приложения «Торговая площадка».

В первом разделе рассматриваются основные технологии, которые использовались в разработке данного приложения, а также его прототипы и актуальность задачи.

Во втором разделе описана архитектура курсового проекта.

В третьем разделе предоставлена информация о разработанных объектах базы данных.

В четвертом разделе представлены результаты тестирования приложения.

Пятый раздел содержит руководство пользователя для разработанного клиентского приложения.

В заключении описывается результат курсового проектирования и задачи, которые были решены в ходе разработки приложения.

1. **Постановка задачи**

Главная задача курсового проекта является разработка WEB- приложения, позволяющего пользователю просматривать каталог товаров, выставлять товары на торговую площадку, добавлять товары в корзину, а также удалять их из корзины, оформлять заказ, выставлять товары на аукцион, ставить ставки на определенный лот. В данном курсовом проекте требовалось реализовать следующие задачи:

* сохранение рабочей информации в централизованной базе данных;
* организация регистрации и входа в систему со стороны пользователей;
* возможность просмотра, добавления, удаления товарных позиций;
* осуществление заказа;
* возможность просмотра, добавления, удаления товаров на аукцион;
* возможность поставить ставку на аукцион;

## **Алгоритмы решения**

При реализации курсового проекта использовались технологии Node.js, фреймворк NestJs, ORM библиотека Sequelize-typescript, Socket.IO для реализации back-end. Для front-end части Material-UI, JavaScript, ReactJS, Redux, Redux-Saga, Sokect.IO-client. PostgreSQL в качестве базы данных.

Node.js –  программная платформа, основанная на движке [V8](https://ru.wikipedia.org/wiki/V8_(%D0%B4%D0%B2%D0%B8%D0%B6%D0%BE%D0%BA_JavaScript)) (транслирующем [JavaScript](https://ru.wikipedia.org/wiki/JavaScript) в [машинный код](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4)), превращающая JavaScript из узкоспециализированного языка в язык общего назначения. Node.js добавляет возможность [JavaScript](https://ru.wikipedia.org/wiki/JavaScript) взаимодействовать с устройствами [ввода-вывода](https://ru.wikipedia.org/wiki/%D0%92%D0%B2%D0%BE%D0%B4-%D0%B2%D1%8B%D0%B2%D0%BE%D0%B4) через свой [API](https://ru.wikipedia.org/wiki/API) (написанный на [C++](https://ru.wikipedia.org/wiki/C%2B%2B)), подключать другие внешние библиотеки, написанные на разных языках, обеспечивая вызовы к ним из JavaScript-кода. Node.js применяется преимущественно на сервере, выполняя роль [веб-сервера](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80), но есть возможность разрабатывать на Node.js и десктопные оконные приложения и даже программировать микроконтроллеры. В основе Node.js лежит [событийно-ориентированное](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D0%B9%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) и [асинхронное (или реактивное)](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B0%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) программирование с [неблокирующим вводом/выводом](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%B1%D0%BB%D0%BE%D0%BA%D0%B8%D1%80%D1%83%D1%8E%D1%89%D0%B0%D1%8F_%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F).

NestJS - это платформа для создания эффективных масштабируемых серверных приложений Node.js. Он использует прогрессивный JavaScript, построен на TypeScript и полностью поддерживает его (но при этом позволяет разработчикам кодировать на чистом JavaScript) и сочетает в себе элементы ООП (объектно-ориентированное программирование), FP (функциональное программирование) и FRP (функциональное реактивное программирование).

Под капотом Nest использует надежные фреймворки HTTP-серверов, такие как Express (по умолчанию), и при желании также может быть настроен на использование Fastify!

Sequelize – это ORM-библиотека для приложений на Node.js, которая осуществляет сопоставление таблиц в базе данных и отношений между ними с классами. При использовании Sequelize мы можем не писать SQL-запросы, а работать с данными как с обычными объектами. Для работы с Typescript необходимо установить пакет sequelize-typescript, который добавляет возможность объявлять классы моделй с помощью декораторов

WebSoket – это передовая технология, которая позволяет создавать интерактивное соединение между клиентом (браузером) и сервером для обмена сообщениями в режиме реального времени. Веб-сокеты, в отличие от HTTP, позволяют работать с двунаправленным потоком данных, что делает эту технологию совершенно уникальной. Socket.IO - это библиотека, которая упращает работу с веб сокетами на серверной стороне. Socket.IO-client на клиентской.

JavaScript – мультипарадигменный язык программирования. Поддерживает объектно-ориентированный, императивный и функциональный стили. Является реализацией стандарта ECMAScript (стандарт ECMA-262). JavaScript обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности веб-страницам.

Material-UI – это библиотека, которая включает компоненты React, реализующие Material Design от Google.

React — это декларативная, эффективная и гибкая JavaScript библиотека для создания пользовательских интерфейсов. Она позволяет собирать сложный UI из маленьких изолированных кусочков кода, называемых «компонентами».

Redux является предсказуемым контейнером состояния для JavaScript приложений. Это позволяет создавать приложения, которые ведут себя одинаково в различных окружениях (клиент, сервер и нативные приложения), а также просто тестируются.

Redux-Saga — это библиотека, которая призвана упростить и улучшить побочные эффекты асинхронных запросов (т.е. такие действия, как асинхронные операции, например, загрузки данных, и "грязные" действия, такие, как доступ к браузерному кешу), сделать лёгкими в тестировании и лучше справляться с ошибками.

## **Обзор прототипа**

В наше время в сети Интернет можно найти любую информацию, поэтому было решено поискать аналоги проектируемого приложения в иных ведущих торговых площадок страны.

Одной из них является площадка Куфар. Эта площадка предназначена для размещения объявлений, покупок интересующих товаров и доставки. Достаточно удобная система поиска товаров и размещения объявления. Приложение имеет довольно понятный любому пользователю интерфейс и функционал.

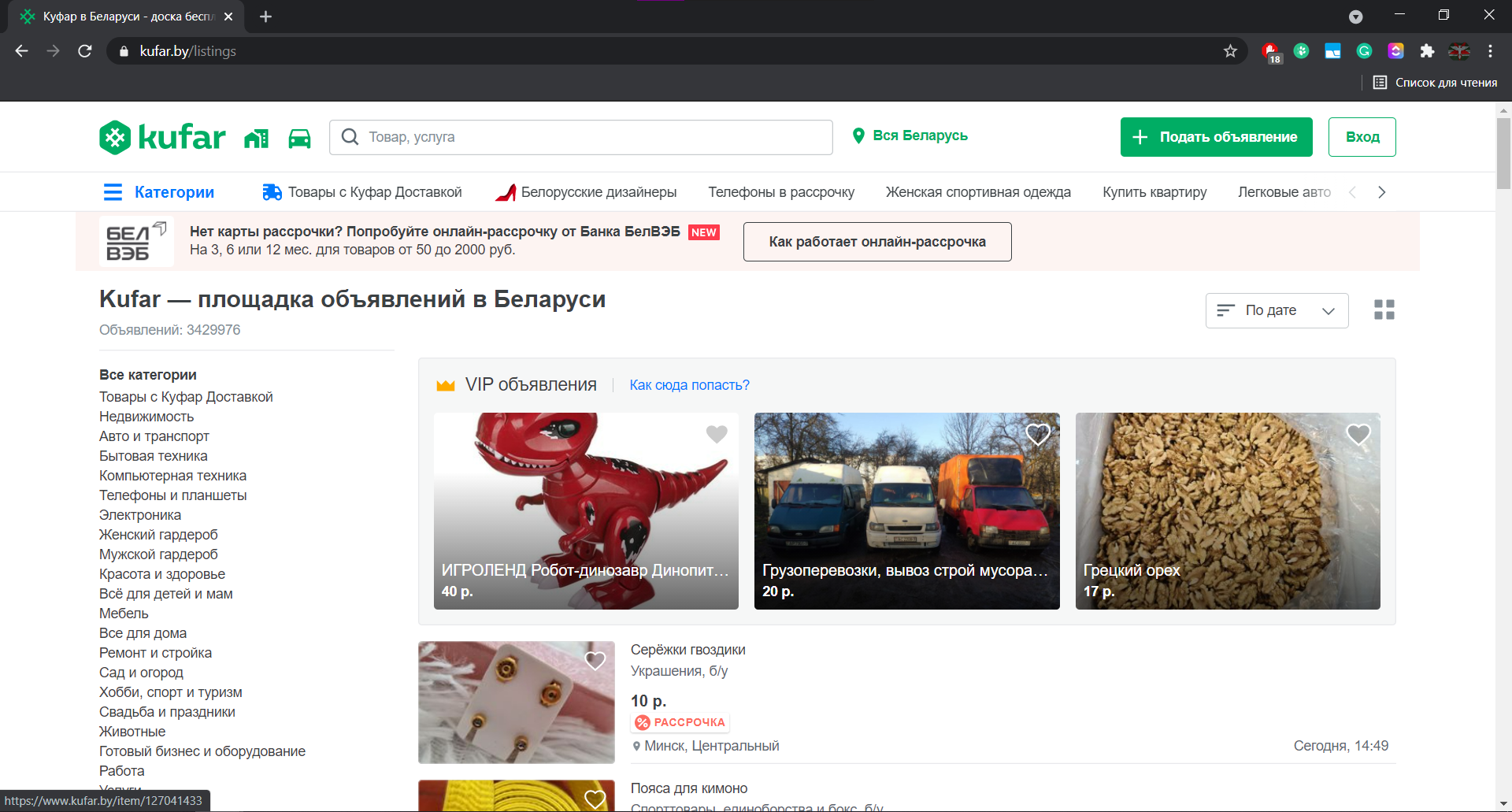


Рисунок 1.1 – Торговая площадка «Куфар»

Еще один интернет-магазин помог мне определиться с функциональностью моего WEB-приложения – это магазин «Wildberries» (рисунок 1.2).

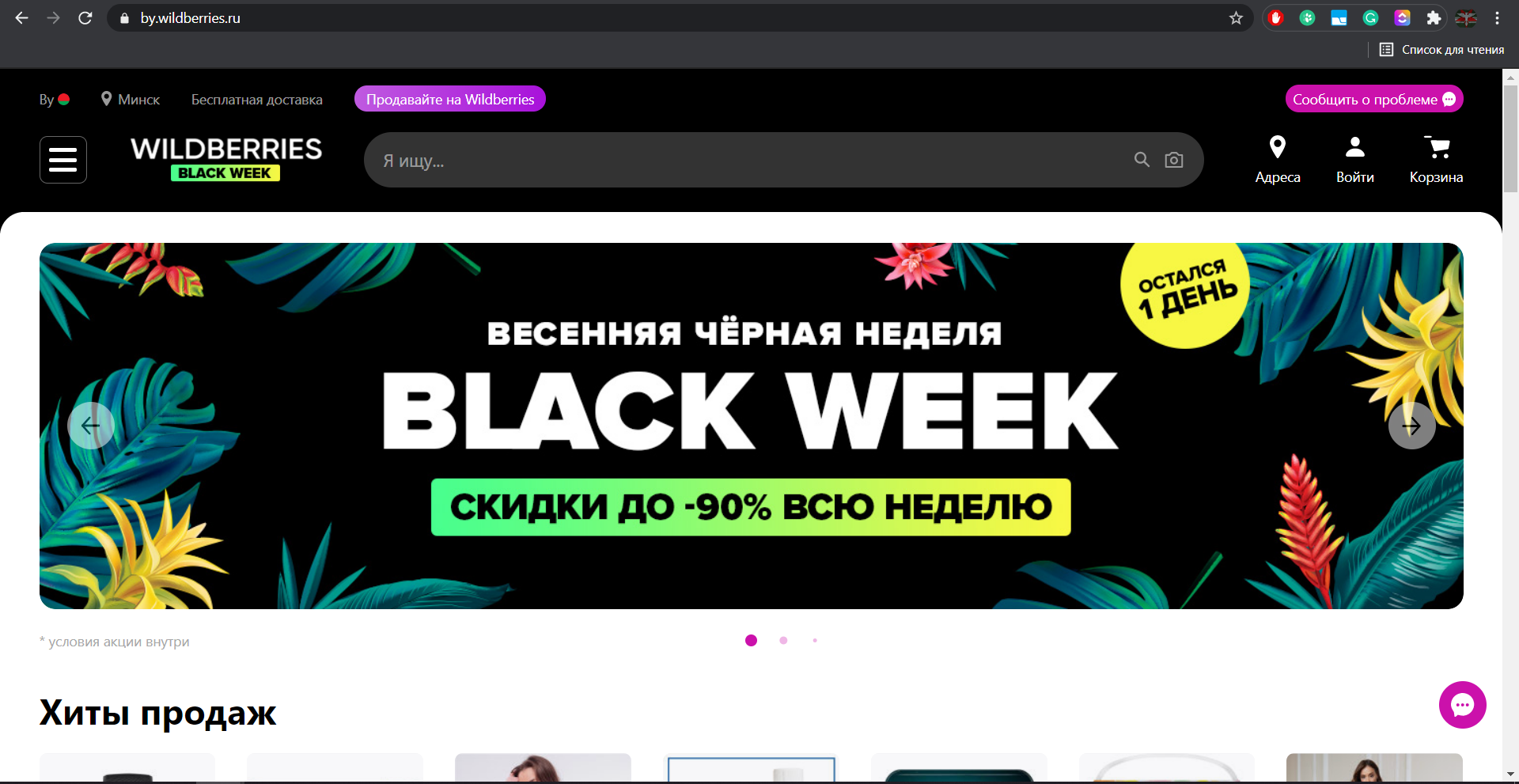


Рисунок 1.2 – Интернет-магазин «Wildberries»

# **Проектирование приложения**

Разработка архитектуры проекта – одна из важнейших задач в процессе работы над приложением, так как в зависимости от неё определяется уровень зависимости компонентами приложения, и насколько легко расширяемы её составные части.

Архитектура проекта – это его строение как оно видно (или должно быть видно) из вне его, т.е. представление программного средства как системы, состоящей из некоторой совокупности взаимодействующих подсистем. В качестве таких подсистем выступают обычно отдельные программы. Разработка архитектуры является первым этапом борьбы со сложностью программного средства, на котором реализуется принцип выделения относительно независимых компонент.

Основные задачи разработки архитектуры проекта:

Выделение программных подсистем и отображение на них внешних функций (заданных по внешнем описании) программного средства;

Определение способов взаимодействия между выделенными программными подсистемами.

С учетом принимаемых на этом этапе решений производится дальнейшая конкретизация и функциональных спецификаций.

## **Архитектура**

Для разработки подобного типа проекта было решено использовать стандартную трёхуровневую архитектуру, так как такая архитектура позволяет добиться низкой связности уровней, что, в свою очередь, обеспечивает для приложения надёжность, простоту долгосрочной поддержки, а также оперативное устранение неисправностей при необходимости.

Клиентская часть приложения является Single-Page Application, то есть приложение, использует единственный HTML-документ как оболочку для всех веб-страниц и организующий взаимодействие с пользователем через динамически подгружаемый HTML, CSS, JavaScript код, генерируемый на основе ответов, поступивших на AJAX запросы. Также на клиенте реализовывается логика обработки полученных от сервера сообщений об ставках на определенный лот на аукционе.

Сервер будет выполнять мероприятия по поддержке аутентификации и авторизации, а также будет ответственным за всё взаимодействие с базой данных.

## **Серверная часть приложения**

При проектировании классов и интерфейсов для серверной части приложения необходимо стараться держать низкий уровень избыточности и связности между составляющими частями, а также усиленно следить за корректностью кода, поскольку от этого будет зависеть работа всех подключенных клиентов. Добиться всего этого можно с помощью грамотного распределения логики сервера по разным уровням, определяя формат объектов, которые будут использоваться на каждом уровне, и преобразования объектов при передаче с уровня на уровень, а также используя подходящие паттерны проектирования.

Для того, чтобы обмениваться данными между клиентами и сервисами был выбран архитектурный стиль REST. По своему определению он широко использует возможности протокола HTTP и поэтому не имеет состояния (stateless). Это современный подход, использующийся в большинстве веб-сервисов и поэтому имеющий множество примеров реализаций различных функций на его основе, таких как аутентификация, выполнение CRUD-операций и т.д. В REST-приложениях в подавляющем большинстве случаев используется JSON для передачи данных, его также было решено взять как формат данных при передаче между клиентами и сервером.

## **Уровни. API, Service, Repository**

Приняв во внимание вышенаписанное, лучше всего разбить серверную логику на следующие уровни API, Service и Repository.

API-уровень. Это самый верхний уровень сервера, и на нём будут располагаться классы-контроллеры. Контроллеры отвечают за обработку входящих запросов и возврат ответов клиенту.

Service-уровень. На этом уровне находятся классы, ответственные за бизнес-логику, среди них как те, которые просто делегируют вызов к репозиторию, так и те, которые выполняют более сложную логику.

Repository-уровень. Он содержит в себе классы-репозитории, использующиеся, когда необходимо совершить какие-либо операции с базой данных. Методы этих классов могут обращаться к БД как с помощью автоматически сгенерированных SQL-запросов, так и с помощью написанных непосредственно программистом. Стоит упомянуть, что создание данных классов в NestJS не требуется, так как для нас эти классы предоставляет пакет sequelize-typescript.

## **Безопасность. Аутентификация и авторизация**

Важнейшим аспектом проектирования серверной части приложения является то, как пользователи будут входить в систему и как система будет понимать, что очередной присланный запрос прислан уже находящимся в системе пользователем.

Стандартными подходами при реализации этих функций является либо аутентификация с помощью логина и пароля, при этом используя собственную БД в качестве их места хранения, либо использование сторонних сервисов, предоставляющих своё API для того, чтобы можно было осуществить вход. Это может быть API на основе протокола OAuth, либо какие-то собственные разработки. При рассмотрении данного вопроса лучше всего обратить внимание на то, какими будут приложение-клиенты, то, в каком окружении, на каких системах они будут запускаться. В нашем случае это приложение под Android и веб-клиент. В связи с этим было принято решение о авторизации с использованием JWT-токена.

После того, как пользователь совершит вход в приложение, необходимо поддерживать его статус и позволять выполнять запросы к API, в то время как запросы не аутентифицированных пользователей – отклонять. Как было описано ранее, серверная часть работает без сохранения состояния по протоколу HTTP, в таком случае достаточно распространённой практикой является использование JWT-токена. Он идеально вписывается в описанные методы взаимодействия с клиентами и поможет поддержать требуемый уровень безопасности.

## **Взаимосвязь всех компонентов**

В решении курсового проекта была использована REST архитектура. Рассмотрим компоненты, используемые в данной архитектуре.

Клиент – это программа, использующая API. Клиент делает запросы к API, чтобы получить некоторую информацию или что-то изменить в приложении. Веб-браузер является клиентом - он взаимодействует с API-интерфейсом, чтобы получить от него содержимое страницы. Запрошенная информация отправляется обратно в браузер и отображается на экране.

Ресурс - это любая информация, которую API может предоставить клиенту. Например, ресурсом в данном API может быть пользователь, фотография или товар. Каждый ресурс имеет уникальное имя, называемое идентификатором ресурса.

Сервер используется приложением, которое принимает запросы клиентов и содержит ресурсы, которые требуются клиенту. Сервер имеет API для взаимодействия с клиентами, не предоставляя им прямого доступа к контенту, хранящемуся в базе данных.

Для реализации серверной части была использована N-Layer архитектура. Эта архитектура выбрана, так как трехуровневая архитектура обеспечивает множество преимуществ для производственной среды и среды разработки за счет модульного разделения пользовательского интерфейса, бизнес-логики и уровней хранения данных. Это дает большую гибкость, позволяя обновлять определенную часть приложения независимо от других частей. Эта дополнительная гибкость может улучшить общее время вывода продукта на рынок и сократить время цикла разработки, давая возможность заменять или обновлять независимые уровни, не затрагивая другие части системы. Данный курсовой проект построен по примеру ниже приведенной схемы классической трехуровневой системы. Ее схематическое представление продемонстрировано на рисунке 2.1.

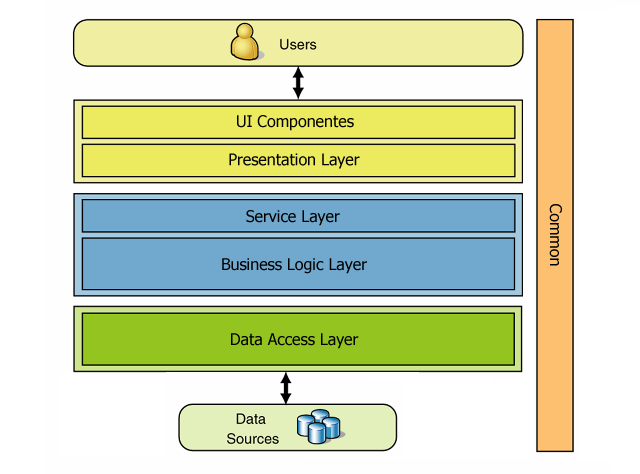


Рисунок 2.1 – Общая схема N-Layer архитектуры

Data Access Layer (уровень доступа к данным) содержит сущности, соответствующие таблицам базы данных, а также схемы взаимодействия друг с другом.

Business Logic Layer (уровень бизнес-логики) при тесном взаимодействии с Service Layer (уровень сервисов) являются посредниками между сущностями базы данных и моделями для UI, сочетая в себе сервисы и классы, реализующий основной функционал всего разрабатываемого приложения.

Presentation Layer (уровень представления), поддерживаемый UI Components (User Interface Components – компоненты пользовательского интерфейса), представляет собой генерирующуюся на сервере разметку или получаемые с уровня бизнес-логики данные для REST API в формате JSON, преобразующиеся впоследствии в разметку на стороне клиента. Уровень представления состоит состоящее из контроллеров, координирующих пользовательские запросы, и представлений, формирующееся в разметку HTML**.**

## **Диаграмма UML**

ДиаграммаUML–это графическое представление набора элементов,изображаемое в виде связанного графа с вершинами (сущностями) и ребрами (отношениями).

В языке UML вариант использования изображается в виде овала, помеченного именем представляемого варианта. Варианты использования могут быть связаны с участвующими в них действующими лицами (actors), изображаемыми в виде человечков и представляющими различные роли пользователей системы или внешние системы, взаимодействующие с ней.

Варианты использования могут быть связаны друг с другом тремя видами связей: обобщением (generalization), расширением (extend relationship) и включением (include relationship). Действующие лица также могут быть связаны друг с другом с помощью связей обобщения (generalization).

Для каждой роли были разработаны UML-диаграммы вариантов использования. При первом открытии сайта пользователь не аутентифицирован. Диаграмма вариантов использование неаутентифицированного пользователя отображена на рисунке 2.2.



Рисунок 2.2 – Диаграмма использования неаутентифицированный пользователя

Как видно из диаграммы, неаутентифицированный пользователь может зарегистрироваться, авторизироваться, просмотреть каталог товаров, а также лоты на аукционе, производить поиск по продуктам, а также добавлять и удалять товары в корзине. При попытке совершения заказа пользователю будет выведено сообщение о том что ему необходимо авторизироваться.

Итак, зарегистрировавшись и войдя в аккаунт, пользователь получает роль аутентифицированного пользователя. Диаграмма вариантов использования приложения в зависимости от роли отображена на Рисунке 2.3.

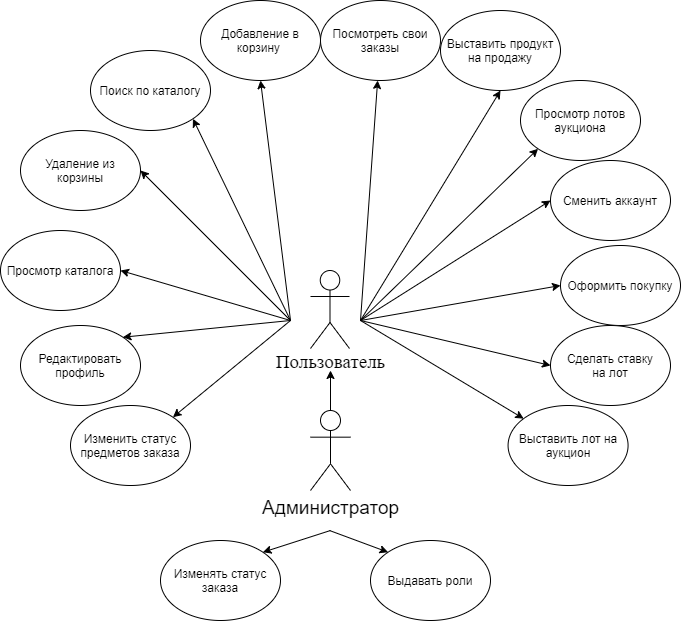


Рисунок 2.3 – Диаграмма использования аутентифицированного пользователя

Из диаграммы видно, что администратор имеет тот же функционал, как и обычный пользователь за исключением того, что присутствуют дополнительные доступные действия для подтверждения статуса заказов и выдачи роли.

## **Сущности в базе данных и связи между ними**

Изучение предметной области и составленный впоследствии функционал показали, что можно выделить 8 таблиц, которые позволяют полностью охватить обозначенный функционал (рис. 2.1). Из них 7 основных:

* пользователь (users) – таблица, предназначенная для хранения основных данных о пользователе, таких как имя, телефон, электронная почта, пароль;
* продукт (products) – таблица, предназначенная для хранения данных о продукте;
* аукцион (auctions) – таблица, предназначенная для хранения данных о товарах выставленных на аукцион;
* аукцион (bids) – таблица, предназначенная для хранения ставок сделанных на аукционе;
* заказы (orders) – таблица, предназначенная для хранения информации о заказах пользователя;
* товары заказа(order\_items) – таблица, предназначенная для хранения информации о товарах в заказе;
* роль (roles) – таблица нужна для того, чтобы хранить данные о существующих ролях;

Оставшиеся таблица user\_roles вспомогательная, и нужна для того, чтобы поддерживать отношения типа «многие-ко-многим».

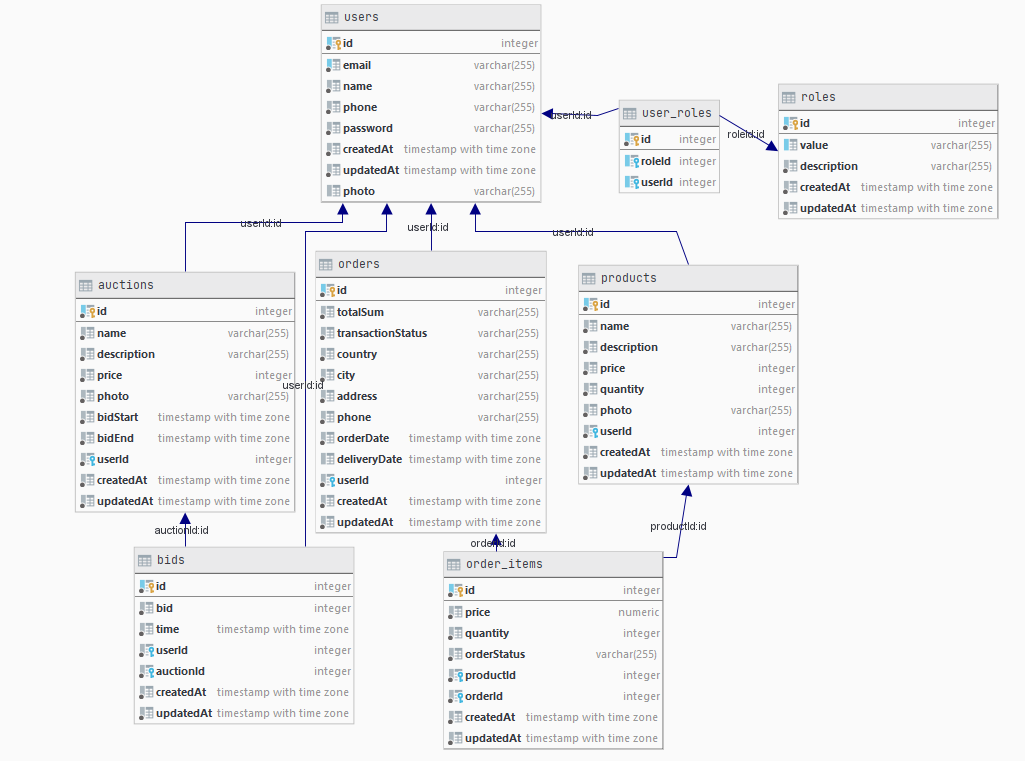


Рисунок 2.1 – Схема базы данных

Отдельно стоит упомянуть способ хранения картинок в базе данных. Там их хранение осуществляется в виде относительных ссылок на файловую систему внутри сервера, либо абсолютных ссылок на Интернет-ресурс, откуда её будет необходимо качать для просмотра.

# **Реализация приложения**

Приложение делится на две части сервер и клиент. Сервер располагается на облачной платформе Heroku на которой поднята так же база данных PostgreSQL. Клиентская часть располагается на платформе Netlify.

* 1. **Серверная часть**

Для разработки серверной части приложения использовался фреймворк NestJS, язык который используется в основе данного фреймворка Typescript.

* + 1. **Конфигурация**

Секретные данные для работы приложения, такие как строка подключения к базе данных или секрет для JWT-токена, содержатся в .env файлах, которые в зависимости от переданной переменной среды NODE\_ENV выбираются с помощью npm-пакета cross-env. Env файл отвечает за то, в каком режиме будет работать сервер: production или development.

В файле tsconfig.json описана конфигурация Typescript.

Файл nest-cli.json отвечает за настройку nest-cli.

Файлы eslint и prettier отвечают за настройку контроля качества кода которого необходимо придерживаться во время разработки программного продукта.

В классе main.ts содержатся метод, настраивающий сервер. В нем создается объект сервера подключается конфигурация Swagger и устанавливаются глобальные middleware и запускается сервер на порту хранящимся в env файле. Помимо всего прочего, там настраивается CORS для корректного доступа с веб-клиента и устанавливается глобальный префикс.

Отдельно стоит уделить внимание форматам даты и времени, которые будут использоваться. Это важный момент, так как очевидно, что тип, представляющий время и дату, является сложным, а значит, что на разных платформах, в разных языках программирования, этот тип может быть интерпретирован совершенно по-разному, поэтому возникает необходимость введения определённого единого стиля. Таковым был выбран стандарт ISO 8601, так как он является самым распространённым и имеет поддержку во всех используемых языках программирования. Он выглядит так: «YYYY-MM-DDThh:mm:ss».

* + 1. **Реализация REST API. Контроллеры**

Классы-контроллеры являются представлением API. Есть условный шаблон, по которому описывается каждый контроллер. Указывается декоратор над классом контроллера, характеризующая его. Далее внутри класса идут методы, каждый из которых помечен другим декоратором, который показывает, какой тип HTTP-метода соответствует этому методу класса, какой тип данных он принимает и какой возвращает. Благодаря NestJS, нет необходимости в том, чтобы осуществлять самостоятельную валидацию, данных, приходящих в виде JSON. Достаточно указать в аргументах метода, класс DTO в котором описать с помощью декораторов ограничения на входные данные. NestJS с помощью глобального middleware ValidationPipe проверит входные данные и в случае если входные данные окажутся не валидными отправит ответ со статусом ошибки 400 и пояснительным текстом. В контроллеры с помощью механизма Dependency Injection встраиваются сервисы, которые в нем используются.

* + 1. **Сервисы.**

Сервисы – классы, которым контроллеры делегируют выполнение различных действий, после поступления на них запроса. Они также помечаются своим особым декоратором, но скорее для того, чтобы фреймворк смог создать по ним объекты и включить их в свой пул для последующего внедрения зависимостей.

Как было уже описано ранее, большинство сервисов лишь выполняют различные действия с репозиториями, иногда лишь делая небольшие преобразования в объектах, чтобы ORM-система смогла правильно их обработать. Работа сервисов совершается в неблокирующем асинхронном режиме. В конструктор сервиса так же встраиваются различные модели.

* + 1. **Репозитории. Взаимодействие с БД**

Для того, чтобы взаимодействовать с базой данных, используется ORM sequelize-typescript. Для этого создаются Модели – классы, представляющие собой таблицы в базе данных, у которых поля соответствуют колонкам этих таблиц. Эти классы принимают на вход методы репозиториев. Репозиторий в нашем случае – это сами классы моделей, которые позволяют работать с ними как с репозиториями.

Каждый класс модели помечается декоратором Table в котором можно указать название таблицы, которой он будет соответствовать.

Описание полей класса также сопровождается декораторами в которых указывается: тип данных, ограничение целостности, ограничения типа данных. Также можно описать декораторами схему для документации.

* + 1. **DTO. Конвертация разных типов объектов**

DTO – важнейшая часть API-системы, так как эти объекты представляют то, как должны будут выглядеть запросы и ответы к ней. Легко понять тот факт, что если запросы будут по большей части одинаковыми, то ответы могут отличаться как по своему наполнению, так и по структуре. Например, если приходит запрос на получение одного пользователя, то не имеет смысла отправлять вместе с ним полную информацию о всех аукционах, в которых он поучаствовал, так как большинство из отправленных данных не будет использоваться получателем и канал передачи будет впустую нагружаться. Также будут возникать проблемы как с многочисленными зависимостями, если они будут тянуть друг друга, так и с циклическими зависимостям, которые будут вызывать переполнение стека. Одним из решений перечисленных проблем является определение класса для каждого из типов запросов. Главным его недостатком является то, что придётся писать достаточно большое количество мало чем отличающегося кода. Тем не менее, преимуществ такой подход несёт больше, поэтому был выбран именно он. Для реализации этого подхода использовались:

* классы для каждого тела запроса;
* классы, которые наследовали либо все поля либо часть из них.
* классы для тела ответа

Пример такого класса можно увидеть в приложении A.

При валидации используется npm-пакет class-validator. Для преобразования ответа модели в DTO используется пакет class-transform.

Так же в приложении реализована загрузка тела с Content-Type двух видов multipart/form-data и application/json. Так как при загрузке информации с телом переданным как multipart/form-data вся информация приходит в виде строки, использовался пакет class-transorm для преобразования типа строк в тип указынный в DTO с помощью декораторов @Transform

* + 1. **JWT. Аутентификация и авторизация**

Как было написано ранее, для нужд аутентификации и авторизации будет использоваться JWT-токен. При аутентификации, он будет формироваться и отправляться аутентифицирующемуся клиенту, затем этот клиент должен будет его отправлять с каждым запросом, а задача сервера – проверять этот токен на валидность.

Для реализации аутентификации использовался пакет passport. Было реализовано несколько стратегий (так в терминологии passport.js называют способ аутентификации).

LocalStrategy – стратегия в которой пользователь аутентифицируется по логину и паролю.

JwtStrategy – стратегия в которой пользователь аутентифицируется по содержащемуся в заголовке Authorization, bearer токен.

Аутентификация пользователя к методам контроллера в Nest.js реализована с помощью Guards. В них как раз и реализован функционал из пакета passport.

Для авторизации пользователя использовались так же Guards которые проверяли на наличие необходимой роли в jwt токене, который добавляется после прохождения JwtAuthGuard в запрос как поле user.

Код RolesGuard представлен в приложении Б.

* + 1. **Разработка документации**

Swagger – это технология, которая позволяет документировать REST-сервисы. Она представляет интерфейc между клиентскими приложениями и сервером (рис. 3.1). Для написания документации в Nest.js используются декораторы, у которых название содержит префикс API. Документировать можно контроллеры, классы моделей, а так же типы представленные DTO классами.

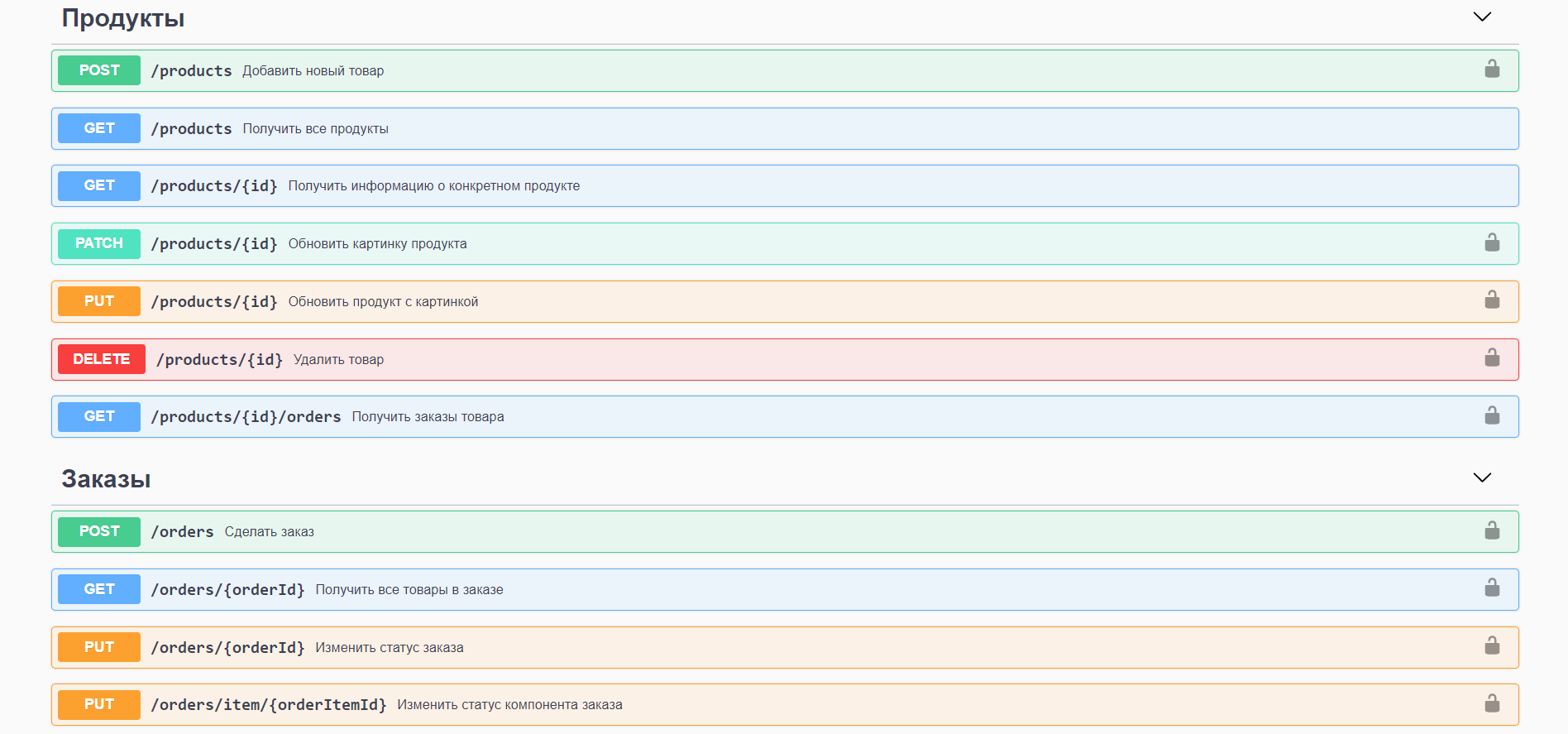


Рисунок 3.1 – Swagger схема

* + 1. **Обработка исключений**

Одна из замечательных особенностей Nest.js заключается в том, что он содержит встроенный уровень исключений, который обрабатывает все те исключения, которые не были обнаружены кодом приложения, и отправляет соответствующий ответ на запрос.

В случае нераспознанного исключения (ни HttpException, ни его унаследованный класс) этот встроенный уровень исключений генерирует ответ на запрос со статус кодом 500 и сообщением Internal server error.

* + 1. **Обмен сообщениями с клиентом в реальном времени**

 Для обмена сообщениями между браузером и веб-сервером в режиме реального времени используется протокол связи поверх TCP-соединения - Websocket.

В Nest для работы с веб-сокетами класс, аннотированный декоратором WebSocketGateway. Технически шлюзы не зависят от платформы, что делает их совместимыми с любой библиотекой WebSockets после создания адаптера. Из коробки поддерживаются две платформы WS: socket.io и ws. В данном проекте использовалась библиотека Socket.IO. С помощью данного протокола в приложении реализована логика работы аукциона. Пользователь при переходе на определенную страницу выставленного лота, подключается к комнате с названием равным идентификатору лота. При ставке на данный лот, всем пользователям, которые присоединились к комнате получают оповещение об новой ставке.

* 1. **Клиентская часть**

Для клиентской части приложения был использован фреймворк React. С помощью React создавать интерактивные пользовательские интерфейсы приятно и просто. Достаточно описать, как части интерфейса приложения выглядят в разных состояниях. React будет своевременно их обновлять, когда данные изменяются.

Декларативные представления сделают код более предсказуемым и упростят отладку. А создание инкапсулированных компоненты с собственным состоянием, позволяет объединить их в сложные пользовательские интерфейсы.

Поскольку логика компонента написана на JavaScript, а не содержится в шаблонах, можно с лёгкостью передавать самые разные данные по всему приложению и держать состояние вне DOM.

* + 1. **Хранение состояния**

[Redux](https://redux.js.org/) – это контейнер с предсказуемым состоянием для приложений JavaScript и очень ценный инструмент для управления состоянием приложения. Структура папок при реализации данного контейнера представлена на Рисунке 3.2.

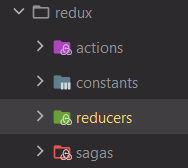


Рисунок 3.2 –Структура папок Redux

Папка reducers содержит все Reducer-ы которые, определяют, как изменяется состояние приложения в ответ на action-ы (действия), отправленные в store.

Reducer – это чистая функция, которая принимает в качестве аргументов предыдущее состояние и action и возвращает новое состояние. Action – это объект, задающий тип и (опционально) нагрузку.

Папка actions содержит Action-ы – это простые объекты JavaScript, которые представляют полезную нагрузку, отправляющую данные из приложения в store. Action-ы принимают тип и опционально полезную нагрузку (type и payload).

Redux store – это объект, объединяющий action-ы (которые представляют то, что произошло) и reducer-ы (которые обновляют состояние в соответствии с этими action-ами). В приложении Redux может быть только один store.

Папка sagas используется для хранения саг. Redux синхронно управляет состоянием приложения. Для работы с асинхронными вызовами используется Redux-saga – это библиотека, нацеленная делать побочные-эффекты проще.

Побочный эффект - это любой код, который выполняется асинхронно или обращается к внешнему источнику приложения.

Саги реализованы как функции генератора, которые передают объекты промежуточному программному обеспечению redux-saga. Пример саги приведен в Приложении В.

* + 1. **Маршрутизация**

Маршрутизация в приложении реализована с помощью библиотеки react-router и react-router-dom. Маршрутизация проверяется на наличии токена в локальном хранилище. Если токен находится в хранилище тогда пользователь считается авторизованным и ему доступны все маршруты за исключением страницы логина и регистрации. На Рисунке 3.3 представлена реализация маршрутизации.



Рисунок 3.3 – Маршруты

* + 1. **Обмен сообщениями с сервером в реальном времени.**

Для реализации двунаправленного соединения с сервером в режиме реального времени используется протокол связи поверх TCP-соединения - Websocket. Для работы с протоколом, как и на сервере была использована библиотека, упрощающая работу с сокетами socket-io-client. При загрузке компонента отвечающего за показ информации об лоте на аукционе, мы подключаемся к серверу по протоколу WebSocket и подписываемся на событие newBid при возникновении которого, обновляем информацию об ставках на данный лот.

1. **Тестирование, проверка работоспособности и анализ полученных результатов**

Тестирование проекта выполнялось в течение разработки посредством таких средств, как отладка и логгирование. После введения нового либо изменения существующего компонента программы, он и зависящий от него код также подвергался тестированию. Анализ полученных результатов происходил по факту получения ошибок и проблемные места сразу исправлялись. После завершения разработки программы по нескольку раз были подвергнуты испытаниям все элементы управления в пользовательском интерфейсе, также они тестировались по ходу написания записки. Если происходили ошибки, то их причина и местоположение в коде легко фиксировались благодаря ведению логов.

После написания приложения также проводились различные тесты, и проверки, такие как проверка на корректную покупку, проверка на корректную работу ставок, проверка корректного создания продукта и нового лота на аукционе, а также изменение профиля.

В курсовом проекте организована валидация полей на случай неочевидного или неправильного поведения пользователя.

Валидация приложения – это один из основных гарантов надёжности приложения и всякий программист должен предусмотреть и предотвратить непредвиденное поведение пользователя.

При регистрации обрабатываются вводимые данные пользователя. Так как поле «email» является уникальным, в базе данных не могут храниться пользователи с одинаковыми логинами, если в базе данных уже существует пользователь с таким же логином, то клиента уведомят об ошибке, это продемонстрировано на рисунке 4.1.

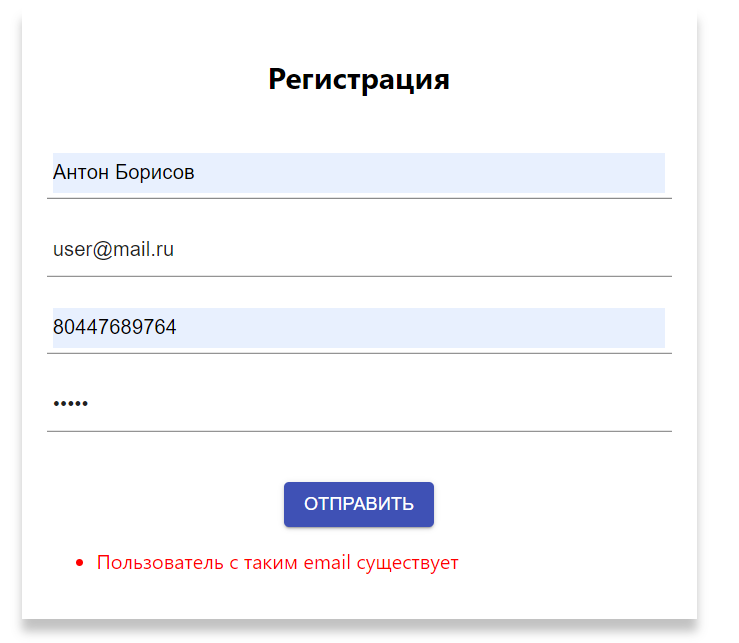


Рисунок 4.1 – Некорректная регистрация

При входе в систему также обрабатываются вводимые данные пользователя. Если пользователь неверно ввел данные в поля для входа, его уведомят об этом, пример уведомления об ошибке предоставлен на рисунке 4.2.

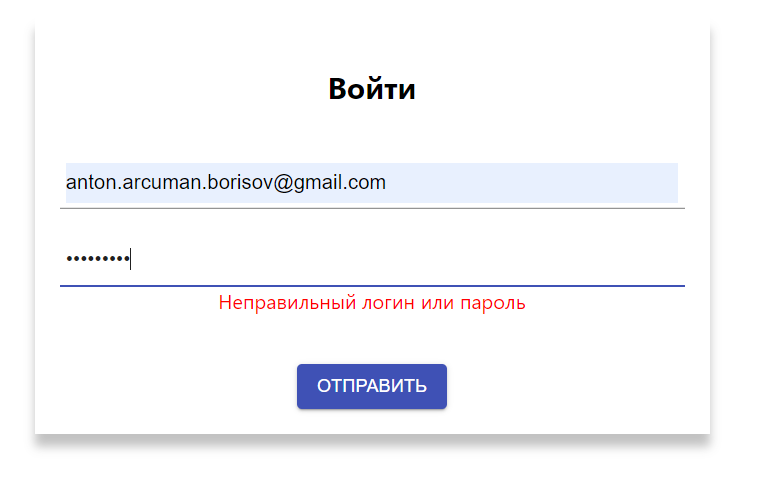
****

Рисунок 4.2 – Неверно введены данные

В итоге приложение стало хорошо отлаженным и готовым к полноценному использованию.

1. **Руководство пользователя**

Данная глава содержит описание некоторых функций приложения для более легкого восприятия конечного пользователя.

* 1. **Регистрация пользователя**

Для регистрации необходимо заполнить форму, которая продемонстрирована на рисунке 5.1.

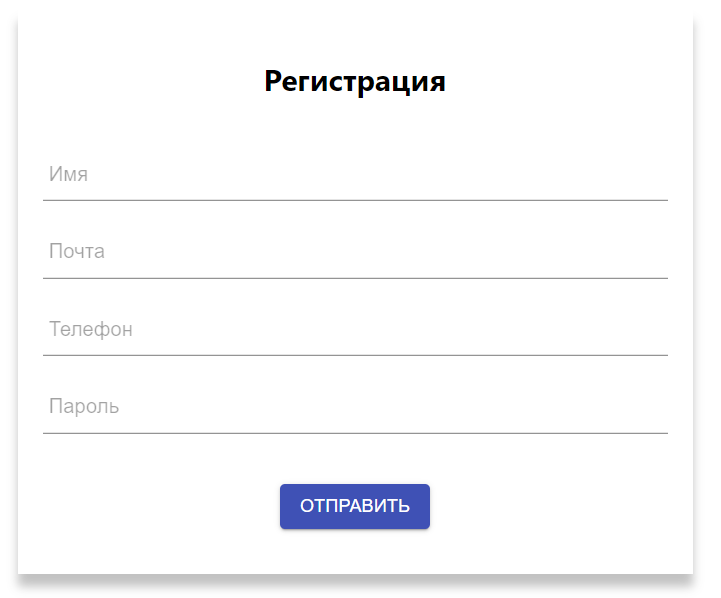


Рисунок 5.1 – Форма регистрации

После отправки пользователем запроса на регистрацию, происходит валидация всех полей формы. Если валидация будет неуспешной, пользователь увидит сообщения об ошибках в соответствующих полях для ввода информации. В случае успешной регистрации пользователь переадресовывается на главную страницу приложения.

* 1. **Аутентификация и авторизация пользователя**

Для аутентификации необходимо ввести пароль, логин и нажать на кнопку «Войти», что позволит проверить, существует ли пользователь с введенными данными в системе. Все поля являются обязательными для заполнения.

Если аутентификация прошла успешно и соблюдены все выше описанные правила, то происходит перенаправление на главную страницу, что означает, что пользователь был успешно авторизован. Скриншот формы авторизации представлен на рисунке 5.2.

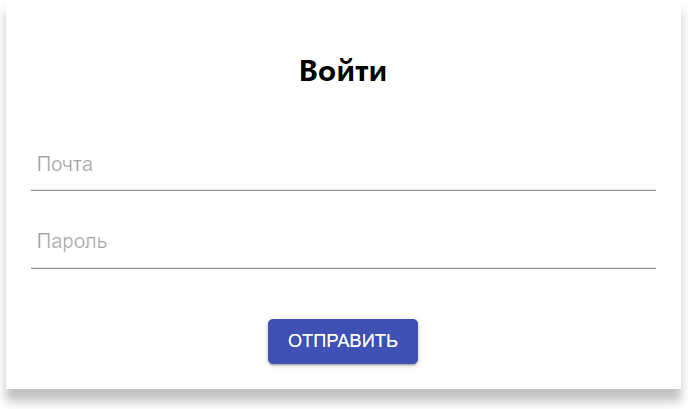


Рисунок 5.2 – Фрагмент экрана с формой для авторизации пользователя

При входе в систему авторизированному пользователю выдается токен, который записывается в локальное хранилище, тем самым позволяя разграничить возможности пользователей на сайте.

В данном курсовом проекте логика приложения задумана так, что пользователь, который не вошел в систему имеет доступ только к просмотру товарных позиций и аукциона, добавления (удаления) товаров. Если пользователь авторизовался, то у него появляется возможность совершать покупки, ставки, а также возможность осуществлять заказ.

* 1. **Оформление заказа**

Для того, чтобы пользователь смог оформить заказ ему необходимо вначале войти на торговую площадку под своим аккаунтом, потому что, если пользователь не авторизирован он лишается возможности оформить заказ. После того как пользователь вошел в систему и выбрал понравившийся ему товар, он может добавить его нажав на иконку «Корзина» (рисунок 5.3).

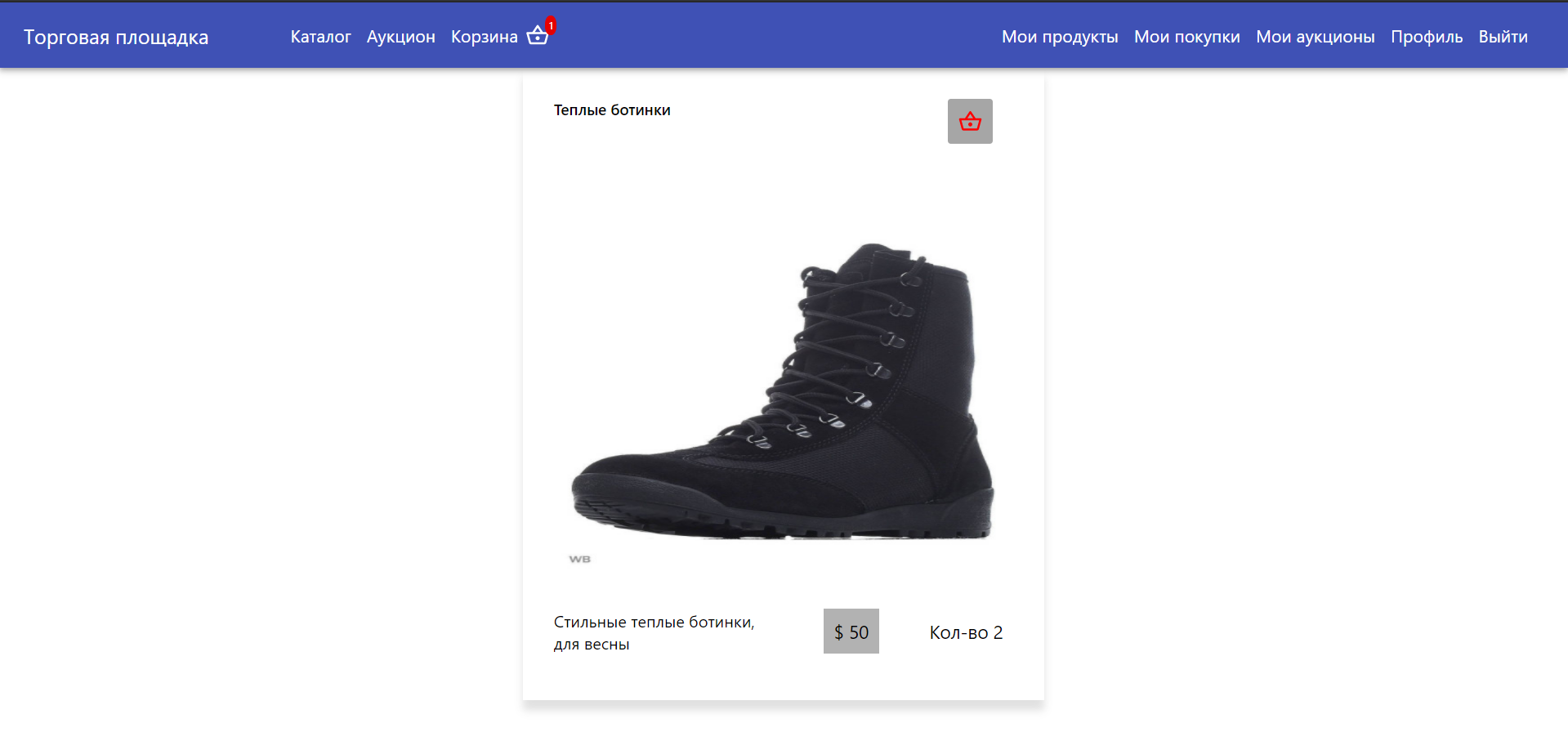


Рисунок 5.3 – Добавление товара в корзину

После того как пользователь добавил в корзину товары, которые хочет приобрести ему следует перейти в корзину. Корзина находится на главной странице слева в верхнем углу. Перейдя в корзину внизу нужно нажать на кнопку «Покупка». После нажатия на данную кнопку появится форма для заполнения (рисунок 5.4).

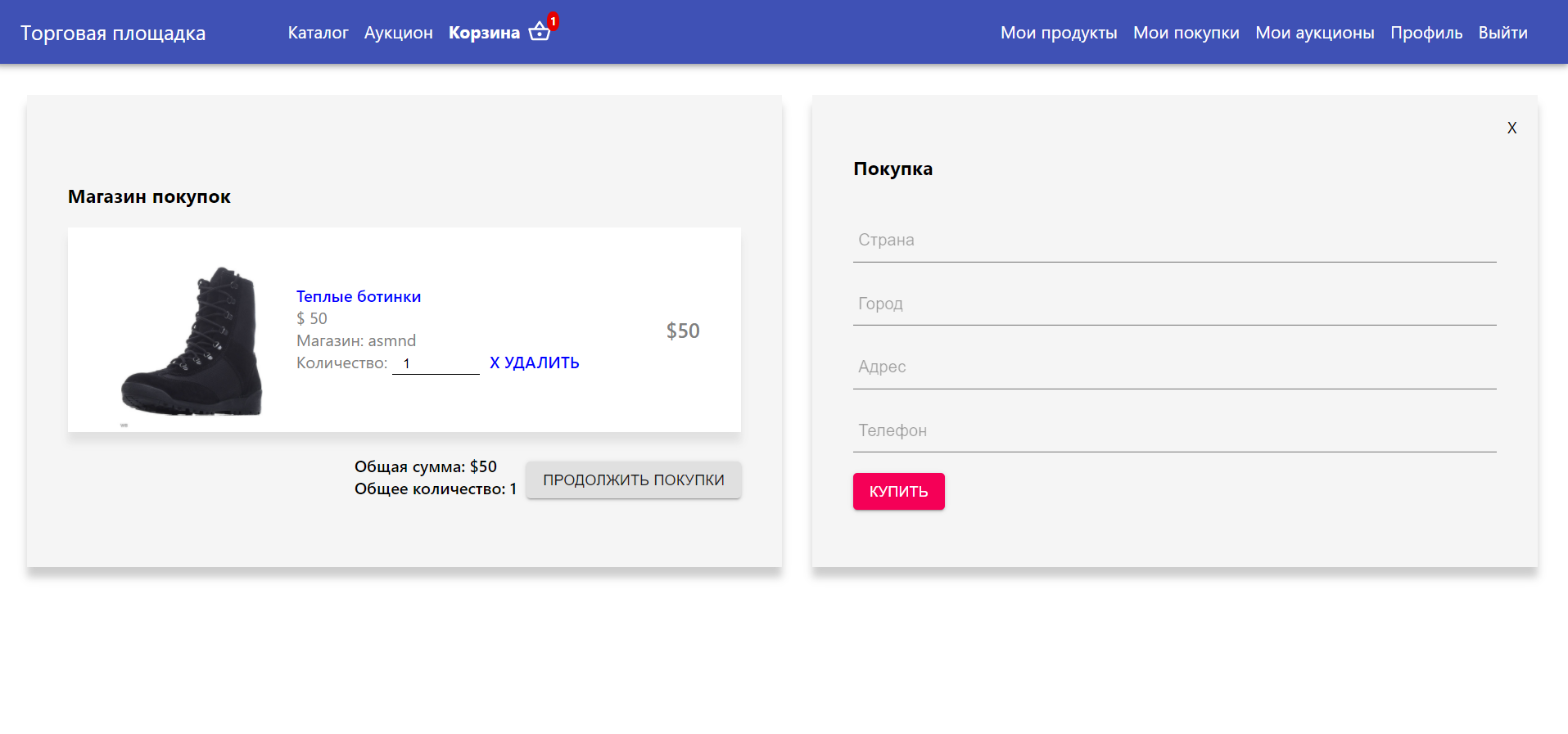


Рисунок 5.4 – Форма для оформления заказа

Можно заметить, что цену изменить невозможно, программа сама подсчитывает сумму всех товаров в корзине и выводит пользователю итоговую цену.

Когда все поля буду корректно заполнены пользователем, необходимо нажать на кнопку «Купить». Ваш заказ будет оформлен, пользователь сможет просмотреть их в разделе «Мои покупки».

* 1. **Аукцион**

В данном курсовом проекте был разработан аукцион. На котором пользователи могут делать ставки в реальном времени. Аукцион был реализован с помощью WebSoket. Более подробно работа WebSoket описана в первой главе.

Чтобы поучаствовать в аукционе необходимо перейти на страницу аукционы и выбрать интересующий вас лот (рисунок 5.6).

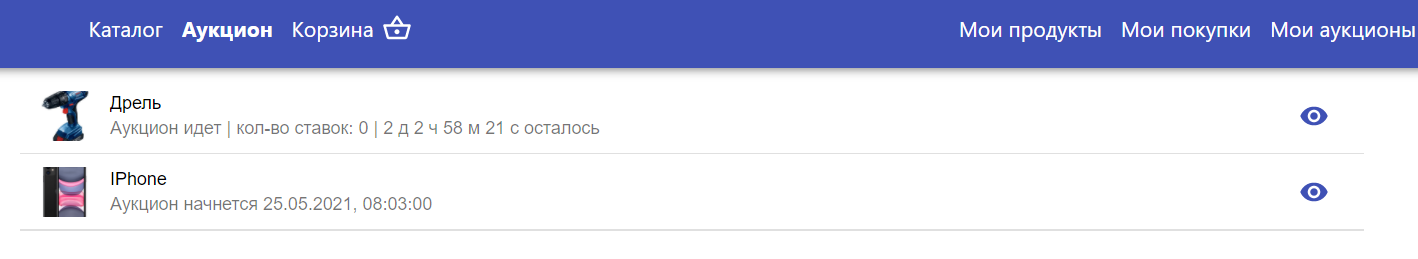


Рисунок 5.6 – Страница аукциона

После перехода на страницу конкретного лота можно увидеть текущие ставки и поставить свою, которая должна быть не меньше чем предыдущая (рисунок 5.7).

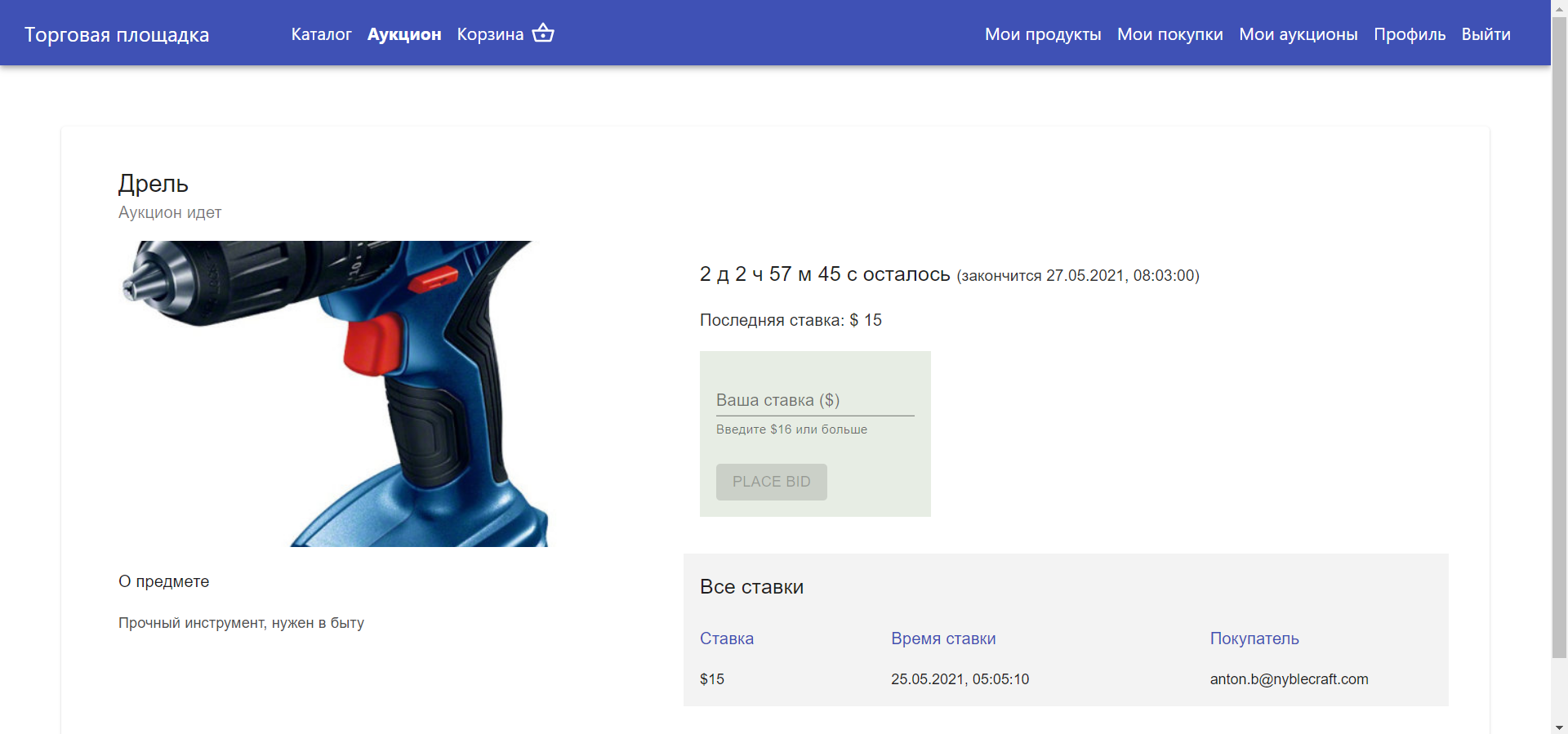


Рисунок 5.7 – Страница лота

# **Заключение**

При работе над курсовым проектом было проведено изучение предметной области, выявление в ней проблем и составление функционала программного средства, с их учётом, после чего последовало проектирование и разработка. Также в ходе работы проанализировано и проработано множество сторон и нюансов проектирования и разработки серверной части приложения и клиентской, например, выбор подходящих паттернов и технологий, составление и тестирование алгоритмов, структурирование проекта, построение пользовательского интерфейса.

В итоге были реализован весь заявленный функционал. Как клиентская, так и серверная части проекта имеют хороший потенциал для будущих модификаций во многом благодаря грамотному структурированию, внимательности при написании кода, а также выборе известных и проверенных сопутствующих технологий. Было получено большое количество опыта по работе с такими фреймворками как React и NestJS, базой данных PostgreSQL, ORM системой Sequileze-Typescript и такими платформами как Netlify и Heroku, улучшены навыки по проектированию, а также применению и связыванию друг с другом различных технологий.

# **Список используемых источников**

1. Асинхронный веб, или что такое веб-сокеты [Электронный ресурс]. — Режим доступа: <https://tproger.ru/translations/what-are-web-sockets/> Дата доступа: 13.03.2021.
2. Что такое Sequelize [Электронный ресурс]. — Режим доступа: <https://metanit.com/web/nodejs/9.1.php> Дата доступа: 10.03.2021.
3. JavaScript [Электронный ресурс]. — Режим доступа:  <https://ru.wikipedia.org/wiki/JavaScript> Дата доступа: 02.03.2021.
4. Документация по React [Электронный ресурс]. — Режим доступа: https://ru.reactjs.org/ Дата доступа: 27.03.2021.
5. Что такое Node.js [Электронный ресурс]. — Режим доступа: <https://metanit.com/web/nodejs/1.1.php> Дата доступа: 07.03.2021.
6. Документация по NestJS [Электронный ресурс]. — Режим доступа: https://nestjs.com/ Дата доступа: 07.03.2021.
7. MaterialUI [Электронный ресурс]. — Режим доступа: https://material-ui.com/versions/Дата доступа: 27.03.2021.
8. Socket.IO [Электронный ресурс]. — Режим доступа: https://socket.io/ Дата доступа: 07.04.2021.
9. N-layer архитектура [Электронный ресурс]. — Режим доступа: <https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier> Дата доступа: 07.03.2021.

# **Приложение А**

@Exclude()  
export class BaseProductDto {  
 @Expose()  
 @ApiProperty({  
 example: 'Nivea',  
 description: 'Brand',  
 })  
 @IsString({ message: 'Должно быть строкой' })  
 @IsNotEmpty({ message: 'Обязательное поле' })  
 @MaxLength(40, { message: 'Не больше 40' })  
 name: string;  
  
 @Expose()  
 @ApiProperty({  
 example: 'Lorem daasdas фывыфвыфв',  
 description: 'Описание',  
 })  
 @IsString({ message: 'Должно быть строкой' })  
 @IsNotEmpty({ message: 'Обязательное поле' })  
 @MaxLength(255, { message: 'Не больше 255' })  
 description: string;  
  
 @Expose()  
 @ApiProperty({  
 example: 1,  
 description: 'Цена',  
 })  
 @Transform((value) => {  
 if (+value.value) {  
 return +value.value;  
 }  
 return -1;  
 })  
 @IsNotEmpty({ message: 'Обязательное поле' })  
 @Min(0, { message: 'Минимальное значение 0' })  
 price: number;  
  
 @Expose()  
 @ApiProperty({  
 example: 1,  
 description: 'Цена',  
 minimum: 0,  
 })  
 @Transform((value) => {  
 if (+value.value) {  
 return +value.value;  
 }  
 return -1;  
 })  
 @IsNotEmpty({ message: 'Обязательное поле' })  
 @Min(1, { message: 'Минимальное значени 1' })  
 quantity: number;  
}  
  
@Exclude()  
export class CreateProductDto extends BaseProductDto {  
 @Expose()  
 @ApiProperty({ type: 'string', format: 'binary' })  
 photo: any;  
}

# **Приложение Б**

@Injectable()  
export class RolesGuard implements CanActivate {  
 constructor(private reflector: Reflector) {}  
  
 canActivate(context: ExecutionContext): boolean {  
 console.log('RolesGuard');  
 const requiredRoles = this.reflector.getAllAndOverride<Role[]>(ROLES\_KEY, [  
 context.getHandler(),  
 context.getClass(),  
 ]);  
 if (!requiredRoles) {  
 return true;  
 }  
 const { user } = context.switchToHttp().getRequest();  
 console.log(user);  
 if (!user) {  
 return false;  
 }  
 return requiredRoles.some((role) => user.roles?.includes(role));  
 }  
}

# **Приложение В**

import axios from "axios";  
import {put, takeEvery} from "redux-saga/effects";  
import {GET\_ALL\_AUCTION\_REQ, GET\_ALL\_AUCTION\_SUCCESS, GET\_ALL\_AUCTION\_FAILED} from '../../constants/auctions'  
import {BASE\_URL} from "../../../constants/constants";  
  
export function\* getAllSaga(action) {  
 try {  
 const res = yield axios.get(`${BASE\_URL}api/auctions`)  
 .then(res => res)  
 console.log(res);  
 yield put({  
 type: GET\_ALL\_AUCTION\_SUCCESS,  
 payload: res.data  
 })  
 } catch (e) {  
 console.log(e);  
 yield put({  
 type: GET\_ALL\_AUCTION\_FAILED,  
 payload: e.response.data.message  
 })  
 }  
}  
  
function\* watchGetAllSaga() {  
 yield takeEvery(GET\_ALL\_AUCTION\_REQ, getAllSaga);  
}  
  
export default watchGetAllSaga;