МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий

Кафедра Информационные системы и технологии

Специальность 1-98 01 03

Специализация Программное обеспечение информационной безопасности мобильных систем

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Приложение службы доставки магазина»

Выполнил студент Шичко Владислав Сергеевич

(Ф.И.О.)

Руководитель проекта

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

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

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

Консультант:

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

Нормоконтролер:

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

Курсовой проект защищен с оценкой

Содержание

[ВВЕДЕНИЕ 3](#_Toc72356407)

[1. Аналитический обзор литературы 4](#_Toc72356408)

[2. Проектирование и разработка 5](#_Toc72356409)

[2.1. База данных 5](#_Toc72356412)

[2.1.1. Таблицы 5](#_Toc72356413)

[2.1.2. Хранимые процедуры 7](#_Toc72356414)

[2.1.3. Триггеры 7](#_Toc72356415)

[2.2. Мобильное приложение 8](#_Toc72356416)

[2.2.1. Архитектура мобильного приложения 8](#_Toc72356417)

[2.2.2. Интерфейс приложения 10](#_Toc72356418)

[2.2.3. Локальная база данных 13](#_Toc72356419)

[2.2.4. Синхронизация данных 13](#_Toc72356420)

[2.3. Web-приложение 14](#_Toc72356421)

[2.4. Web-сервер 15](#_Toc72356422)

[3. Тестирование разработанного продукта 17](#_Toc72356423)

[3.1. Позитивное тестирование 17](#_Toc72356424)

[3.2. Негативное тестирование 20](#_Toc72356425)

[4. Руководство пользователя 22](#_Toc72356426)

[4.1. Мобильное приложение 22](#_Toc72356427)

[4.2. Веб-приложение 27](#_Toc72356428)

[Заключение 30](#_Toc72356429)

[Список используемых источников 31](#_Toc72356430)

[Приложение А 32](#_Toc72356431)

[Приложение Б 33](#_Toc72356432)

[Приложение В 34](#_Toc72356433)

[Приложение Г 36](#_Toc72356434)

# ВВЕДЕНИЕ

Целью данного курсового проекта является создание приложения для службы доставки магазина. Данное программное средство позволит выполнять:

* добавление, удаление, редактирование информации о заказах;
* отслеживание состояния заказа;
* уведомление клиентов о состоянии заказа;
* сбор статистики;
* работу с мобильным приложением без доступа к сети;
* регистрацию курьеров.

Таким образом, упростится контроль за статистикой службы доставки.

Функционал приложения позволяет:

1. Выполнение функций администратора службы:
2. Возможность подтверждения аккаунтов курьеров;
3. Добавление, удаление, изменение информации о заказах;
4. Выполнение функций курьера службы:
5. Получение информации о заказах;
6. Изменение статуса заказа;
7. Регистрация и аутентификация.

# Аналитический обзор литературы

В современном мире сервисы доставки играют немаловажную роль в жизни людей. Доставка позволяет ускорить и упростить получение товара, купленного в сети Интернет.

В открытом доступе находится мало информации о подобных приложениях, поэтому рассмотрим одно из таких приложений «Курьерская служба 2008». Основой мобильного приложения сервиса доставки является список заказов, а также возможность их принять или отказаться (Рисунок 1.1). В информацию о заказе входят адрес и время доставки. Данное приложение позволяет связаться с заказчиком по телефону, а также изменять состояние заказа.

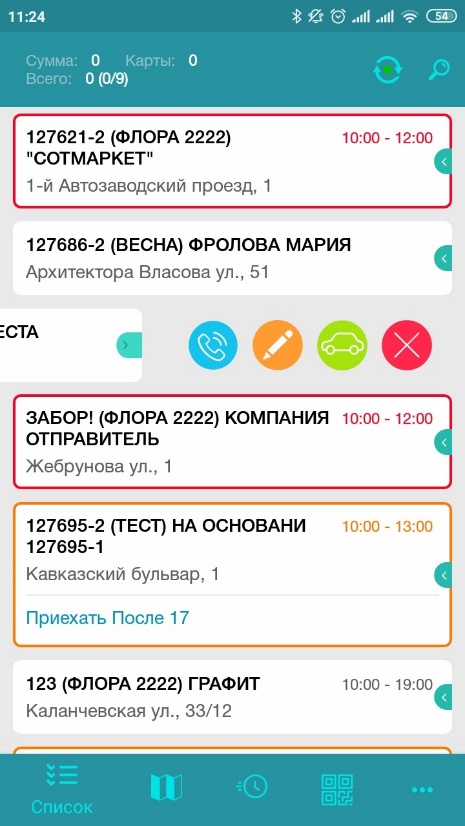


Рисунок 1.1 – Главная страница приложения «Курьерская служба 2008»

Наша задача спроектировать приложение, функционально соответствующее нашему техническому заданию, а также позволяющее работать с ним без соединения с сетью Интернет.

# Проектирование и разработка

Исходя из требований к курсовому проекту, необходимо разработать следующие модули:

* База данных;
* Мобильное приложение;
* Web-приложение;
* REST и SOAP сервисы.

Далее будут описаны проектирование и разработка каждого из модулей



# База данных

В качестве системы управления базами данных была выбрана Microsoft SQL Server 2012.

При проектировании и разработке базы данных программного средства нам понадобились следующие объекты базы данных:

* Таблицы;
* Хранимые процедуры;
* Триггеры.

База данных, используемая в приложении, будет состоять из 8 таблиц: customer, user, user\_role, role, orders, ordered, product, changes.

Диаграмма базы данных со структурой отношений представлена в приложении А.

# Таблицы

Таблица user содержит в себе всех пользователей, зарегистрированных в системе, состоит из:

* id – идентификатор пользователя;
* email – электронная почта пользователя;
* password – пароль пользователя;
* first\_name – имя пользователя;
* second\_name – фамилия пользователя;
* phone – номер телефона пользователя.

Идентификатор пользователя генерируется автоматически при регистрации. Пароль хранится в виде хеша, основанном на алгоритме Blowfish, для повышения безопасности.

Таблица role необходима для хранения ролей пользователей и содержит следующие столбцы:

* id – идентификатор роли;
* name – название роли.

Столбец name может принимать следующие значения:

* ROLE\_BASIC – роль, выдаваемая по умолчанию при регистрации, позволяет просматривать информацию об аккаунте;
* ROLE\_COURIER – роль курьера, позволяющая принимать и отказываться от заказов, а также изменять состояние заказа, и иметь доступ к статистике;
* ROLE\_ADMIN – роль администратора. Позволяет изменять данные всех основных таблиц. Принимать и отказываться от заказов администратор не может, так как не является курьером.

В данном приложении заказчик пользователем не является, так как необходимость регистрации при заказе в интернет-магазине не оправдана.

Таблица user\_role используется для реализации отношения многие-ко-многим между таблицами role и user. Имеет следующие столбцы:

* user\_id – идентификатор пользователя;
* role\_id – идентификатор роли.

Таблица orders необходима для хранения информации о заказах и состоит из следующих столбцов:

* id – идентификатор заказа;
* customer\_id – идентификатор заказчика;
* courier\_id – идентификатор курьера;
* address – адрес доставки;
* info – дополнительная информация;
* state – состояние заказа;
* ordered\_at – время заказа;
* delivered\_at – время доставки;
* preferred\_range\_start – начало предпочитаемого промежутка доставки;
* preferred\_range\_end – конец предпочитаемого промежутка доставки.

Столбец state может принимать следующие значения:

* Ordered – состояние, принимаемое заказом при его создании;
* Delivering – состояние, устанавливаемое курьером при начале доставки заказа;
* Delivered – состояние, устанавливаемое курьером при доставке заказа;
* Canceled – состояние, устанавливаемое администратором при отмене.

Курьер также может отказаться от заказа, при этом значение столбца courier\_id примет значение null, а состояние заказа изменится на Ordered.

Значение столбцов ordered\_at и delivered\_at устанавливаются автоматически при соответствующих событиях.

Таблица customer необходима для хранения информации о заказчике, которая указывается при создании заказа. Содержит следующие столбцы:

* id – идентификатор заказчика;
* email – электронная почта заказчика;
* first\_name – имя заказчика;
* second\_name – фамилия заказчика;
* phone – номер телефона заказчика.

Таблица product используется для хранения информации о товарах и состоит из столбцов:

* id – идентификатор товара;
* name – наименование товара;
* price – стоимость товара;
* weight – масса товара.

Таблица ordered используется для связи заказа с заказанными товарами и содержит следующие столбцы:

* id – идентификатор записи;
* order\_id – идентификатор заказа;
* product\_id – идентификатор товара;
* amount – количество товара.

Таблица changes необходима для хранения информации об изменениях записей в других таблицах:

* table\_name – наименование таблицы;
* item\_id – идентификатор измененной записи;
* operation – изменяющая операция (insert, update, delete);
* time\_stamp – время изменения.

# Хранимые процедуры

Хранимые процедуры используются в базе данных разрабатываемого приложения для получения статистики курьера. В приложении 4 хранимых процедуры:

* delivered\_orders\_amount\_by\_courier\_id – возвращает количество заказов доставленных курьером по его идентификатору;
* delivered\_orders\_in\_time\_by\_courier\_id – возвращает количество заказов доставленных курьером вовремя по его идентификатору;
* total\_delivered\_price\_by\_courier\_id – возвращает суммарную стоимость всех товаров, доставленных курьером;
* total\_delivered\_product\_amount\_by\_courier\_id – возвращает количество доставленных курьером товаров.

Код хранимых процедур приведен в приложении Б.

# Триггеры

В базе данных используются триггеры tr\_change\_<имя\_таблицы> для каждой таблицы кроме changes для внесения информации об изменениях в таблицу changes.

Также для таблицы orders объявлены два дополнительных триггера:

* set\_timestamp\_on\_delivered – задает логику установки значения в столбец delivered\_at при изменении состояния заказа;
* set\_timestamp\_ordered\_at – задает логику установки значения в столбец ordered\_at при изменении состояния заказа.

Код триггера tr\_change\_orders приведен в приложении В.

# Мобильное приложение

Мобильное приложение будет функционировать на базе операционной системы Android версии не ниже API 26.

Имеет адаптивный дизайн, который корректно отображается на различных разрешениях экрана.

Минимальные функциональные требования:

* ОЗУ 512mb;
* OS Android 8.0 и выше (API 26+);
* Network пропускная способность выше 200kb/s;
* Внутренняя память не менее 128mb.

# Архитектура мобильного приложения

Для построения архитектуры нашего приложения будут использоваться следующие паттерны:

* MVVM
* Observer
* Singleton
* Dao

Использование паттернов проектирования в нашем приложении упростит читаемость кода, исключит появления спагетти-кода.

Структура мобильного приложения насчитывает более 70 классов и представлена на рисунке 2.1. Архитектура приложения разбита на слои и соответствует MVVM. В модели используются 2 основных источника данных: API сервера и локальная база данных. Для работы с сетью использовался фреймворк Retrofit2, а для работы с локальной базой данных SQLite – библиотека Room. Пользовательский интерфейс создан на основе одной активности и 8 фрагментах с использованием Android Navigation. Используется Databinding для связи представления и модели представления. Все запросы к модели являются асинхронными, что не позволяет заблокировать поток интерфейса.

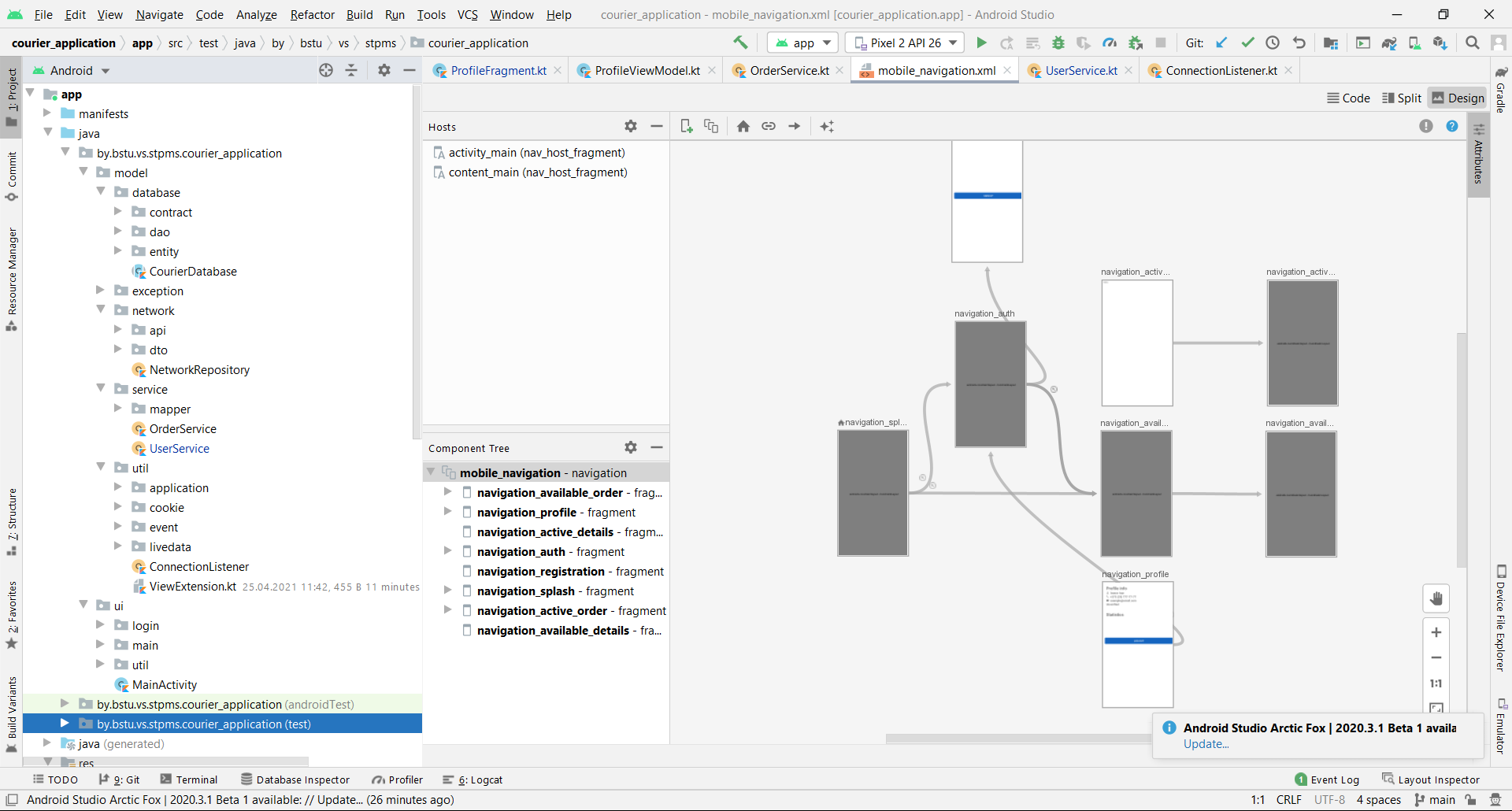


Рисунок 2.1 – Структура мобильного приложения

Структура мобильного приложения насчитывает более 70 классов и представлена на рисунке 2.1. Архитектура приложения разбита на слои и соответствует MVVM. В модели используются 2 основных источника данных: API сервера и локальная база данных. Для работы с сетью использовался фреймворк Retrofit2, а для работы с локальной базой данных SQLite – библиотека Room. Пользовательский интерфейс создан на основе одной активности и 8 фрагментах с использованием Android Navigation. Используется Databinding для связи представления и модели представления. Все запросы к модели являются асинхронными, что не позволяет заблокировать поток интерфейса.

Для сохранения входа в приложение используется Shared Preference, для которого специально написан класс для упрощённой работы с данными. Хранение осуществляется в формате XML.

Так же в приложении реализованы:

* конвертеры;
* адаптеры;
* классы-слушатели интернет-соединения;
* классы работы с локальной базой данных;
* классы валидации;
* методы синхронизации данных.

# Интерфейс приложения

Интерфейс приложения включает в себя 7 экранов для взаимодействия с пользователем. При первом запуске открывается AuthFragment (Рисунок 2.2). Помимо полей ввода логина и пароля, на нем расположены 2 кнопки. При нажатии на кнопку «LOGIN» происходит попытка входа в систему. В случае успеха открывается доступ к основной части приложения, в противном случае выводится сообщение об ошибке. При нажатии на кнопку «REGISTRATION» открывается экран регистрации, при помощи которого можно создать новую учетную запись.

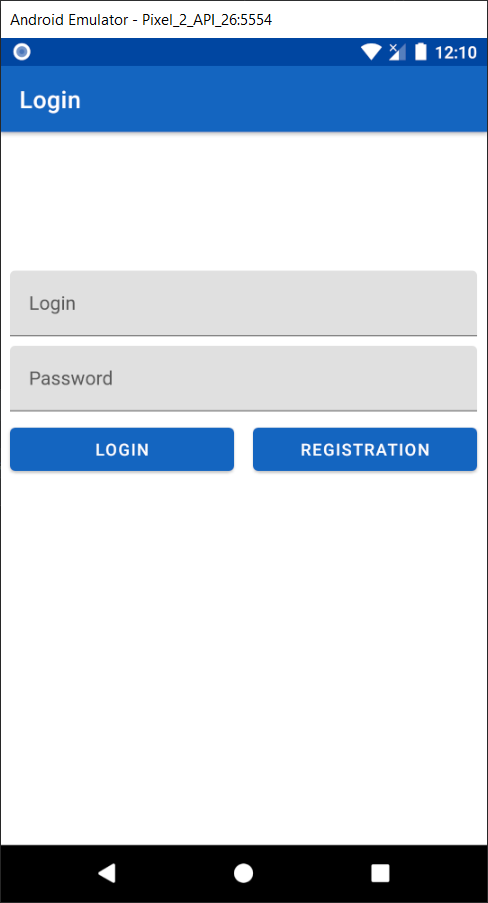


Рисунок 2.2 – Экран входа в приложение

После успешного входа происходит запоминание учетной записи посредством сохранения уникального куки, что позволяет пропускать форму входа при последующем использовании приложения.

Далее пользователь попадает на экран AvailableFragment (рисунок 2.3), на котором размещен список доступных для принятия заказов. Список будет доступен пользователю только в том случае, если у него есть роль ROLE\_COURIER, в противном случае, будет выведено соответствующее сообщение об ошибке.

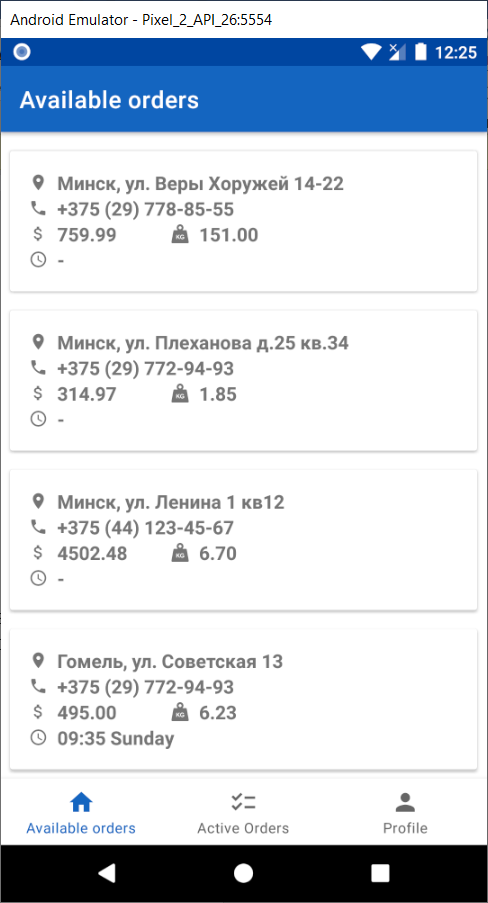


Рисунок 2.3 – Экран доступных заказов

Экран активных заказов имеет такой же интерфейс, за тем исключением, что принятые заказы сгруппированы по состоянию (Рисунок 2.4). Также список активных заказов доступен и без интернет-соединения для повышения автономности курьера. Подробности синхронизации будут рассмотрены в п.2.2.3.

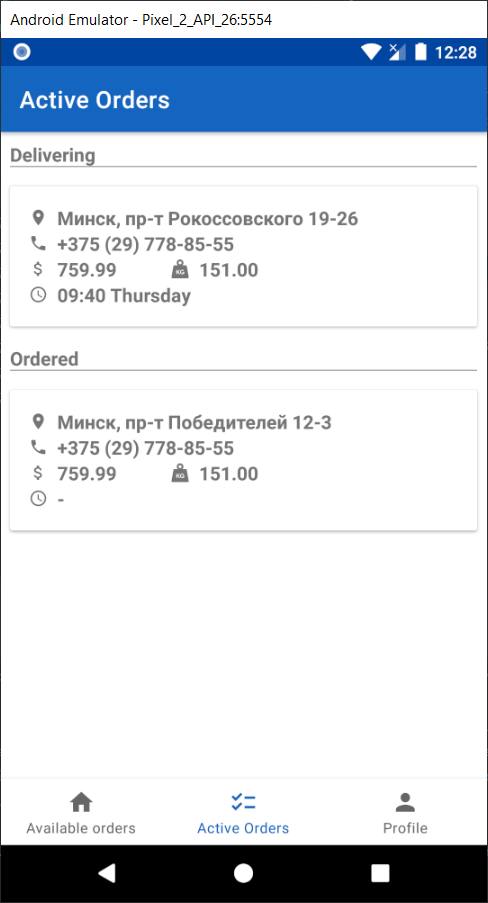


Рисунок 2.4 – Экран активных заказов.

При нажатии на любой заказ из списка откроется экран деталей заказа (Рисунок 2.5), на котором размещена дополнительная информация о заказе, такая, как точный интервал желаемого времени доставки, информация, оставленная заказчиком, список товаров заказа. Экран также позволяет изменять состояние заказа, а также принять или отклонить (зависит от того, в каком контексте он был открыт). Помимо этого, функционал позволяет открыть адрес при помощи google maps, а также позвонить по номеру заказчика, используя интерфейс приложения.

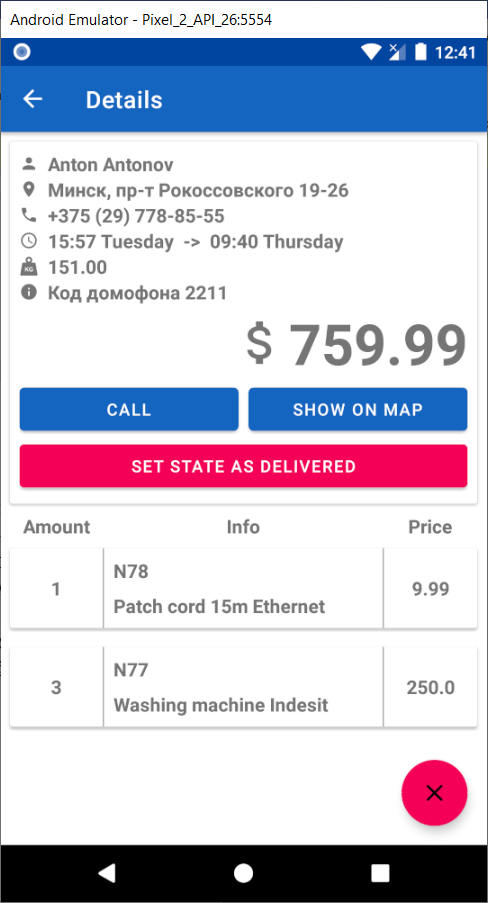


Рисунок 2.5 – Экран деталей заказа

Последним рассмотренным экраном приложения будет экран информации об аккаунте (рисунок 2.6). Это единственный экран, доступный пользователям без роли ROLE\_COURIER. Здесь отображена информация об аккаунте, является ли он подтвержденным (присутствует ли роль ROLE\_COURIER), а также статистика аккаунта. Здесь же есть кнопка «LOGOUT» позволяющая выйти из приложения, удалив соответствующие cookie из shared preferences файла.

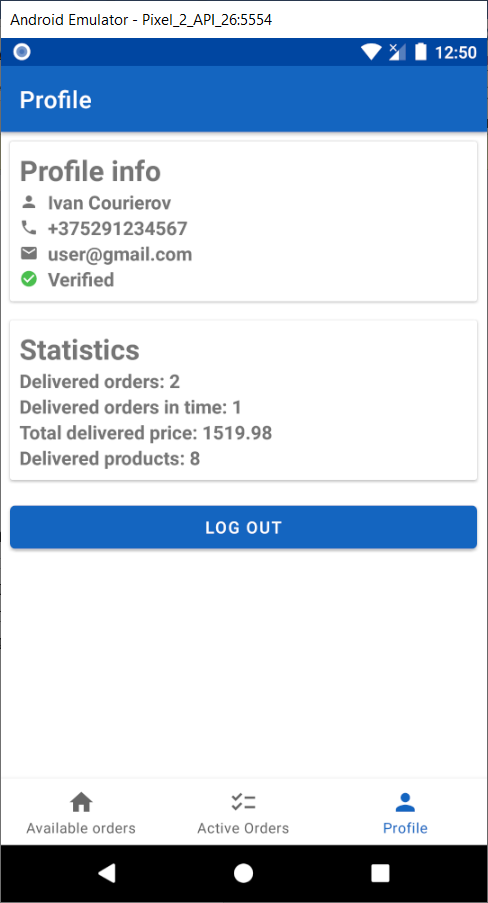


Рисунок 2.6 – Экран информации о профиле

# Локальная база данных

Локальная база данных SQLite имеет такую же архитектуру, что и основная база данных, за тем лишь исключением, что триггеры внесения записей в таблицу changes есть лишь у таблиц user и orders, а у таблицы changes есть дополнительный столбец isUpToDate, в который устанавливается значение 1 при записи данных, полученных с сервера, и 0 в противном случае (по умолчанию).

# Синхронизация данных

За синхронизацию данных о заказах отвечает класс OrderService (Приложение Г). В общем случае, для вывода на экран данных о заказах, реализуется следующая схема: если интернет-соединение есть, то получаем данные с сервера, записываем их в локальную базу данных, записываем 1 в столбец isUpToDate соответствующих строк таблицы changes и возвращаем на экран из локальной базы данных. В случае, если интернет-соединения нет, возвращаем данные из локальной базы данных. При изменении состояния заказа при наличии интернет-соединения, отправляется соответствующий запрос на сервер. В противном случае данные изменяются локально, эти изменения фиксируются триггером в таблицу changes со значением 0 в столбце isUpToDate (что говорит о том, что данные были изменены локально). Далее при появлении интернет-соединения вызывается соответствующий метод (sendDecline, sendUpdateState), который находит обновленные данные и отправляет их на сервер.

# Web-приложение

Web-приложение представляет собой приложение, написанное по принципу Single Page Application, с использованием Vue.js. Данное приложение предоставляет инструменты администратора по управлению данными. Для администратора предоставляются 2 страницы для управления пользователями и заказами. Помимо страниц администратора, есть общедоступные страницы входа, регистрации (которые по функционалу повторяют экраны мобильного приложения), а также страница информации о состоянии заказа, ссылка на которую отсылается заказчику на почту при оформлении заказа и содержит информацию о текущем состоянии, курьере и адресе доставки.

Страница управления пользователями (рисунок 2.7) представляет собой таблицу, позволяющую добавлять и удалять роли пользователей, а также получать статистику каждого пользователя приложения.

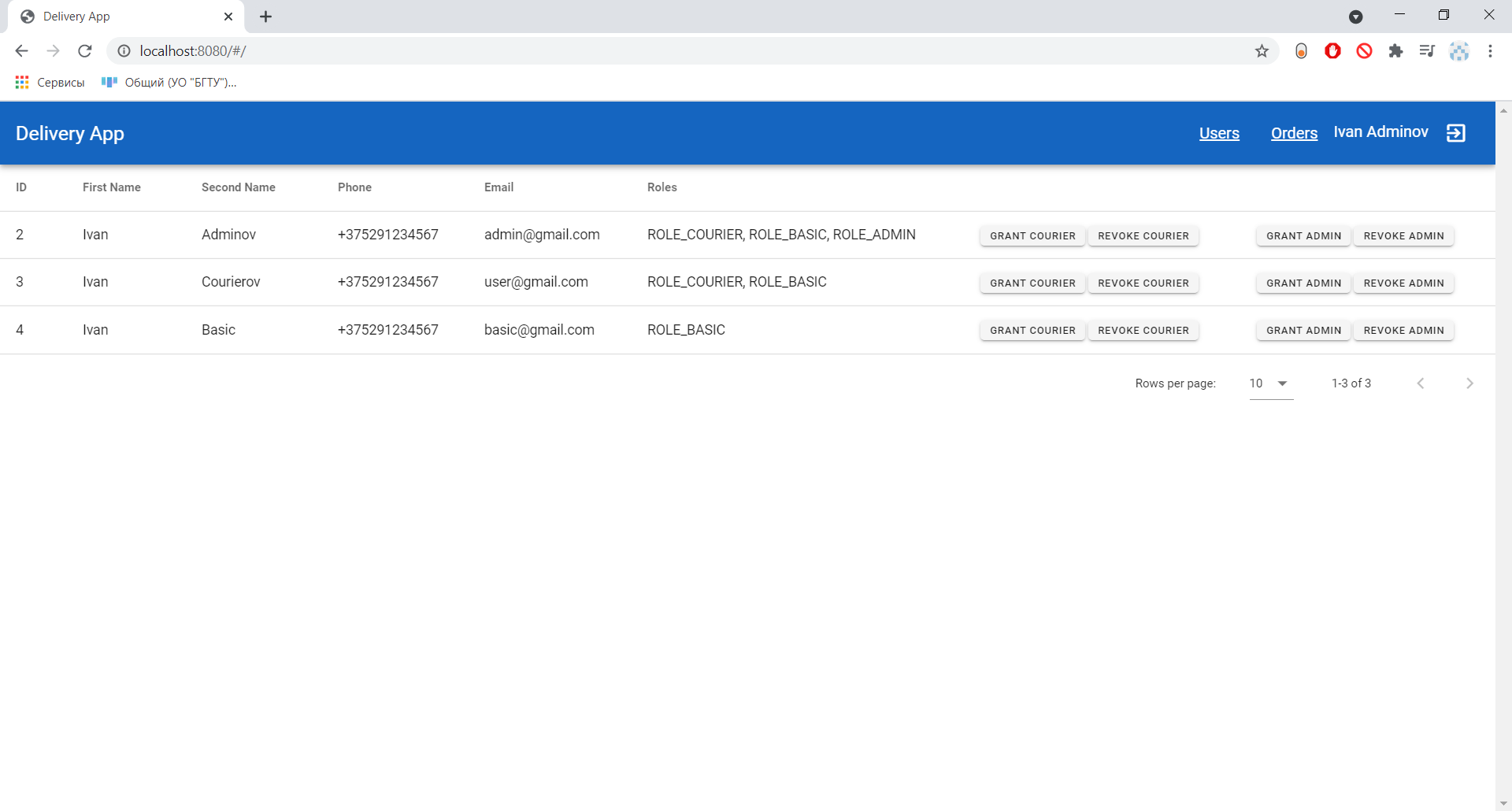


Рисунок 2.7 – Страница управления пользователями

Страница управления заказами содержит таблицу заказов, позволяющую просматривать всю информацию, а также отменять и присваивать состояние по умолчанию заказам (рисунок 2.8).

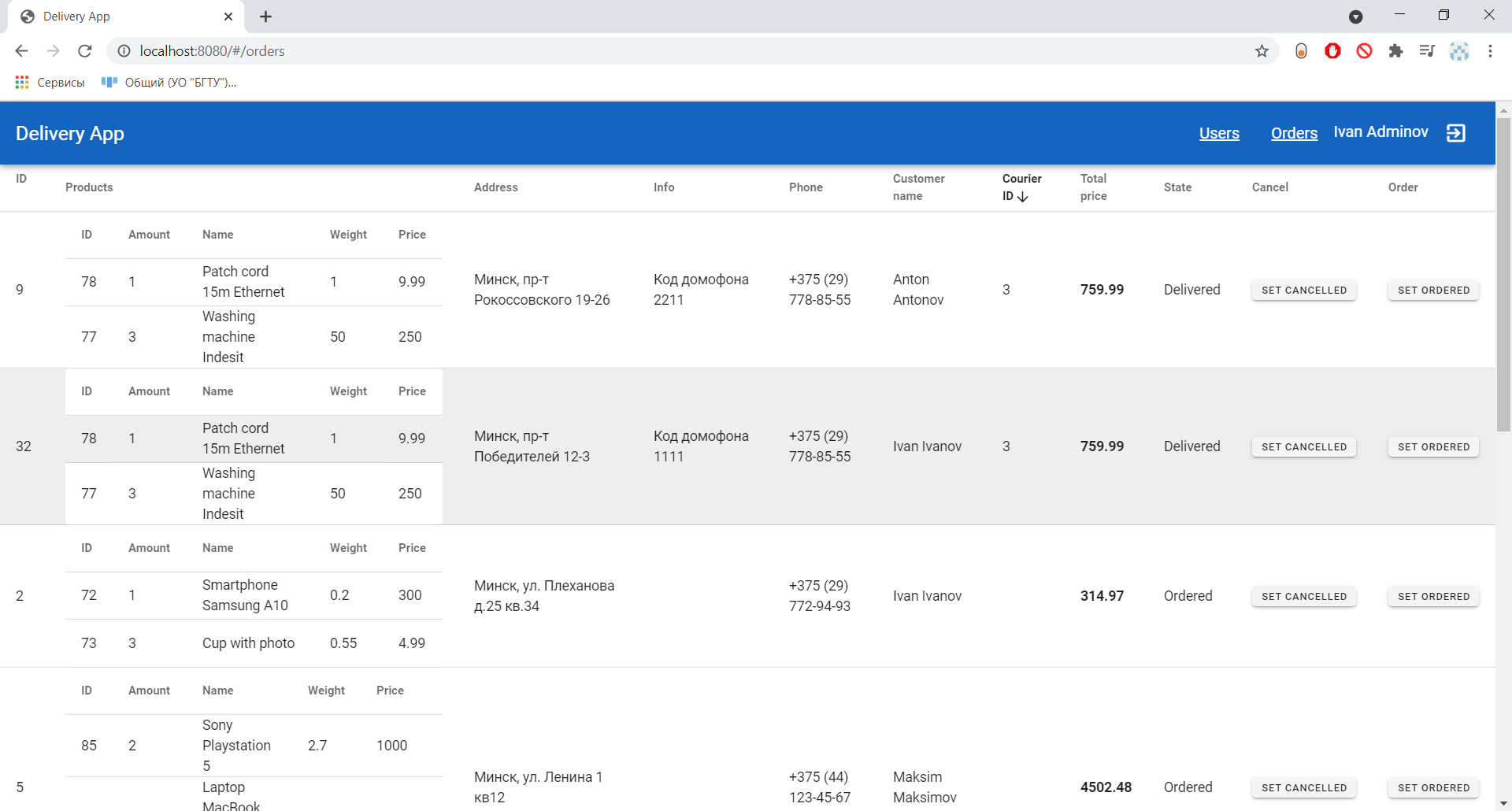


Рисунок 2.8 – Страница информации о заказах

# Web-сервер

Сервер приложения был сделан с использованием Spring Framework. Для взаимодействия с базой данных использовался Java Persistence API c целью упрощения работы с базой данных как с репозиторием.

Основные компоненты Web-сервера:

* entity – классы-сущности, представляющие собой отображение сущностей базы данных;
* repository – классы для обращения к базе данных как к коллекции данных;
* service – классы, использующие репозитории для реализации бизнес-логики;
* dto – классы, использующиеся для передачи данных между клиентом и сервером;
* controller – классы, получающие REST/SOAP запросы, обрабатывающие их при помощи сервисов и отправляющие ответ клиенту.

REST-сервисы использовались для реализации CRUD операций с базой данных, а также некоторых других задач, связанных с требованиями.

Также был разработан SOAP-сервис, необходимый для отправки заказчику состояния заказа в виде xml-файла, являющимся валидным в соответствии со схемой xsd.

Общая структура серверной части приложения изображена на рисунке 2.9.

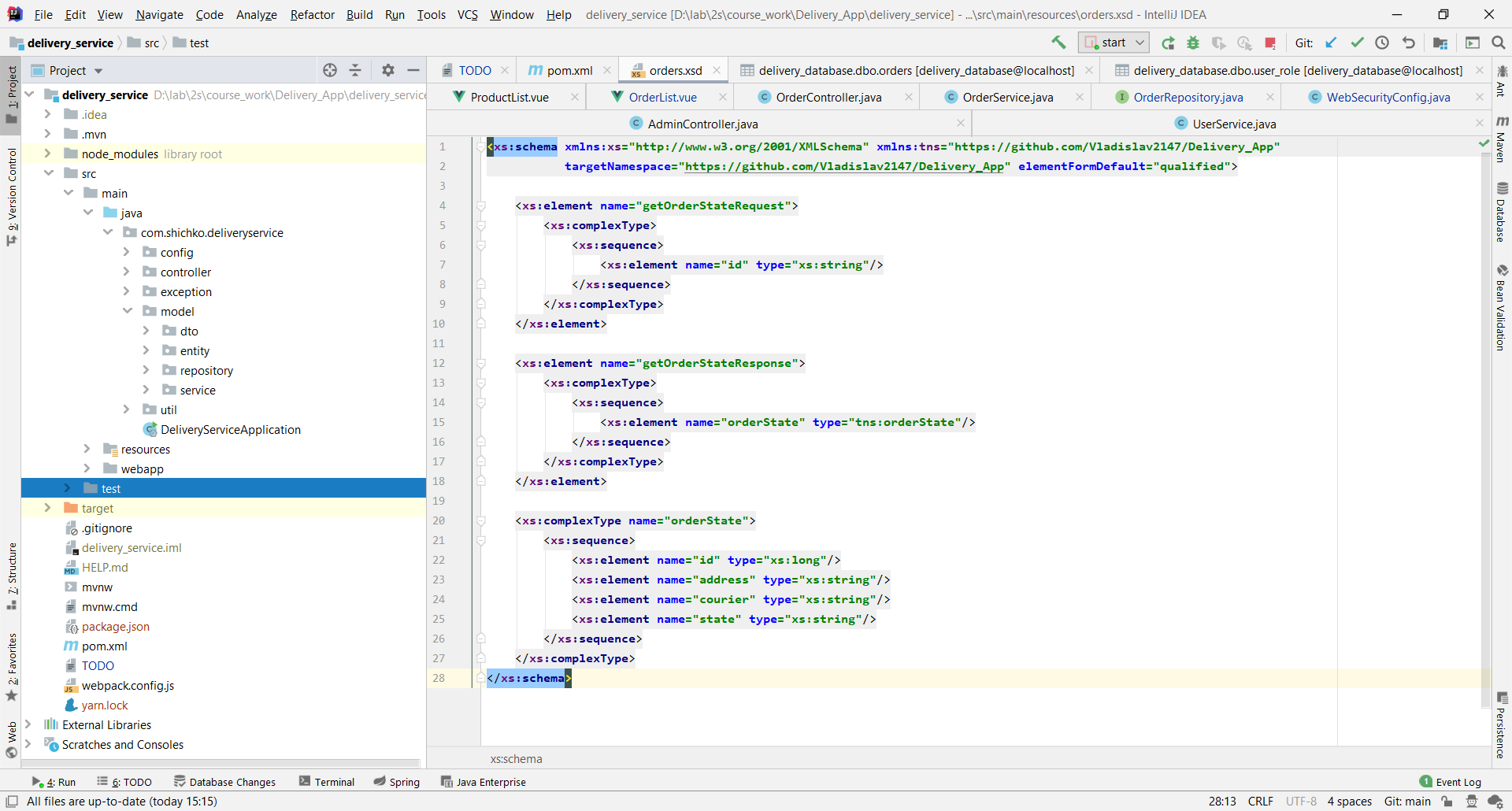


Рисунок 2.9 – Общая структура серверной части приложения

Для реализации логики аутентификации и авторизации использовалась библиотека spring-security, а также класс конфигурации WebSecurityConfig, что позволило разграничивать предоставляемые сервисом ресурсы по соответствующим ролям.

# Тестирование разработанного продукта

Важной частью при разработке приложения является его тестирование с целью проверки его работоспособности при различных ситуациях. Будет рассмотрено два варианта развития событий: все пункты проходят успешно (позитивное тестирование), все пункты проваливаются (негативное тестирование).

В данном разделе будут протестированы несколько сценариев использования с применением как позитивного, так и негативного тестирования.

# Позитивное тестирование

Позитивное тестирование – это один из видов тестирования, который позволяет проверить систему на корректное поведение. В ходе такого тестирования мы можем узнать, что система правильно функционирует.

Рассмотрим сценарий регистрации нового пользователя с использованием валидных данных.

При вводе валидных данных у формы не возникает ошибок (Рисунок 3.1 а), а при нажатии на кнопку SIGN UP происходит переход на форму логина с сообщением об успешной регистрации (Рисунок 3.1 б).

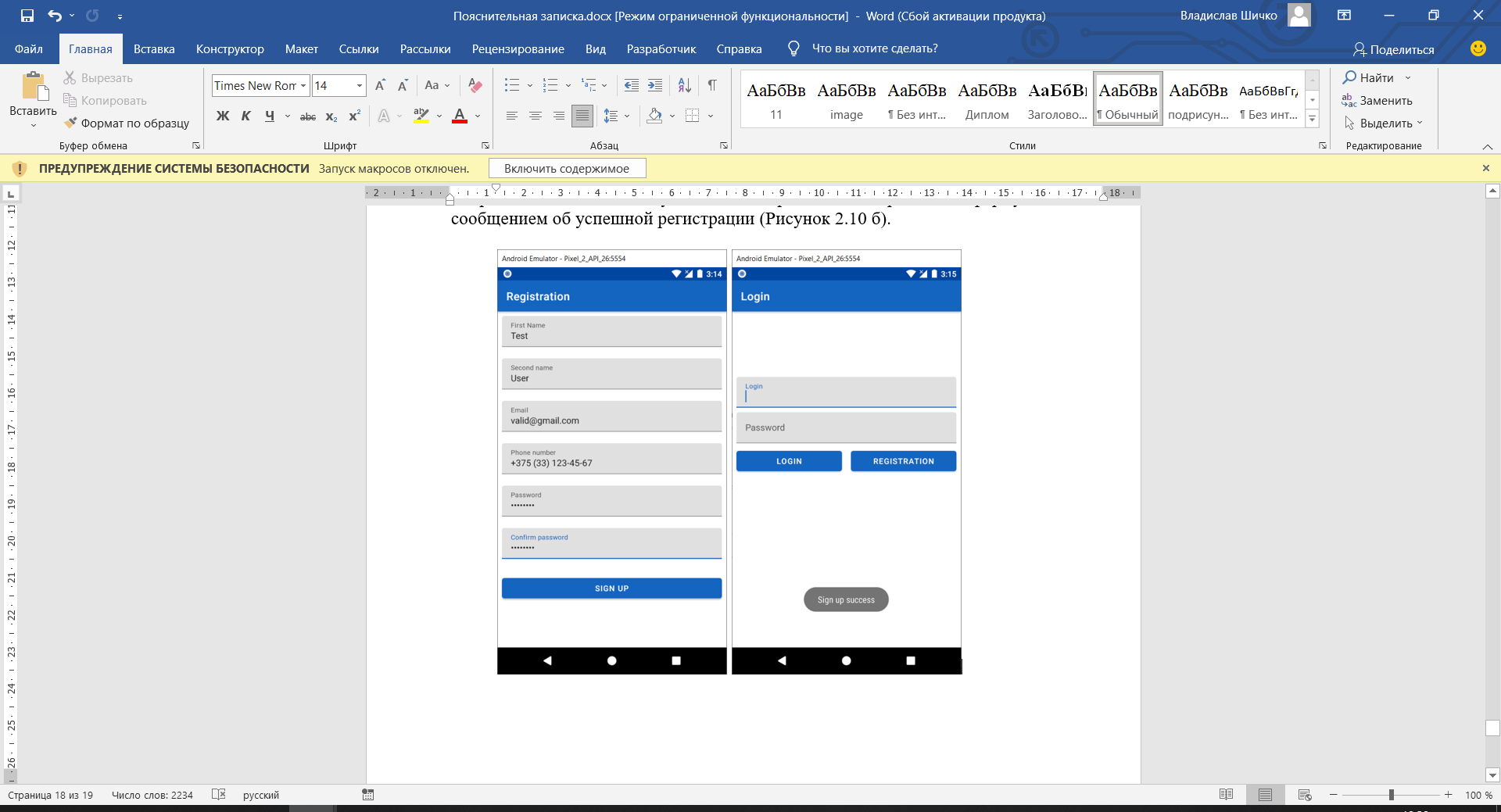


Рисунок 3.1 а – отсутствие ошибок формы, б – сообщение об успешной регистрации

Также в веб-приложении можно увидеть только что зарегистрировавшегося пользователя с ролью по умолчанию ROLE\_BASIC (Рисунок 3.2).

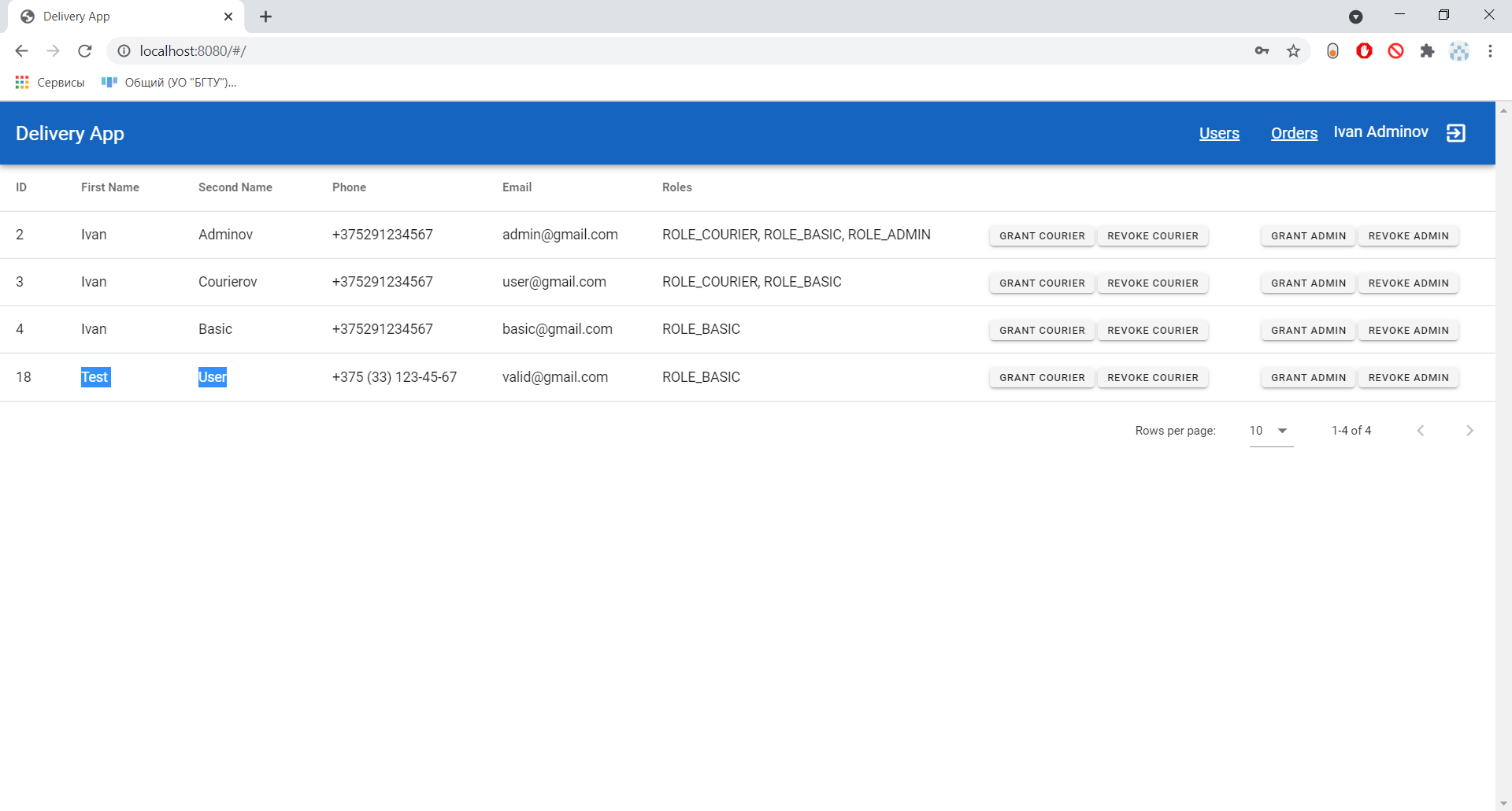


Рисунок 3.2 – Новый пользователь отображается в веб-приложении

Рассмотрим еще один позитивный сценарий: курьер изменяет статус заказа без интернет-соединения, в это время администратор отменяет заказ, и при установке курьером соединения статус заказа в приложении администратора остается отмененным.

Пусть курьер №3 принял заказ №2 (Рисунок 3.3).

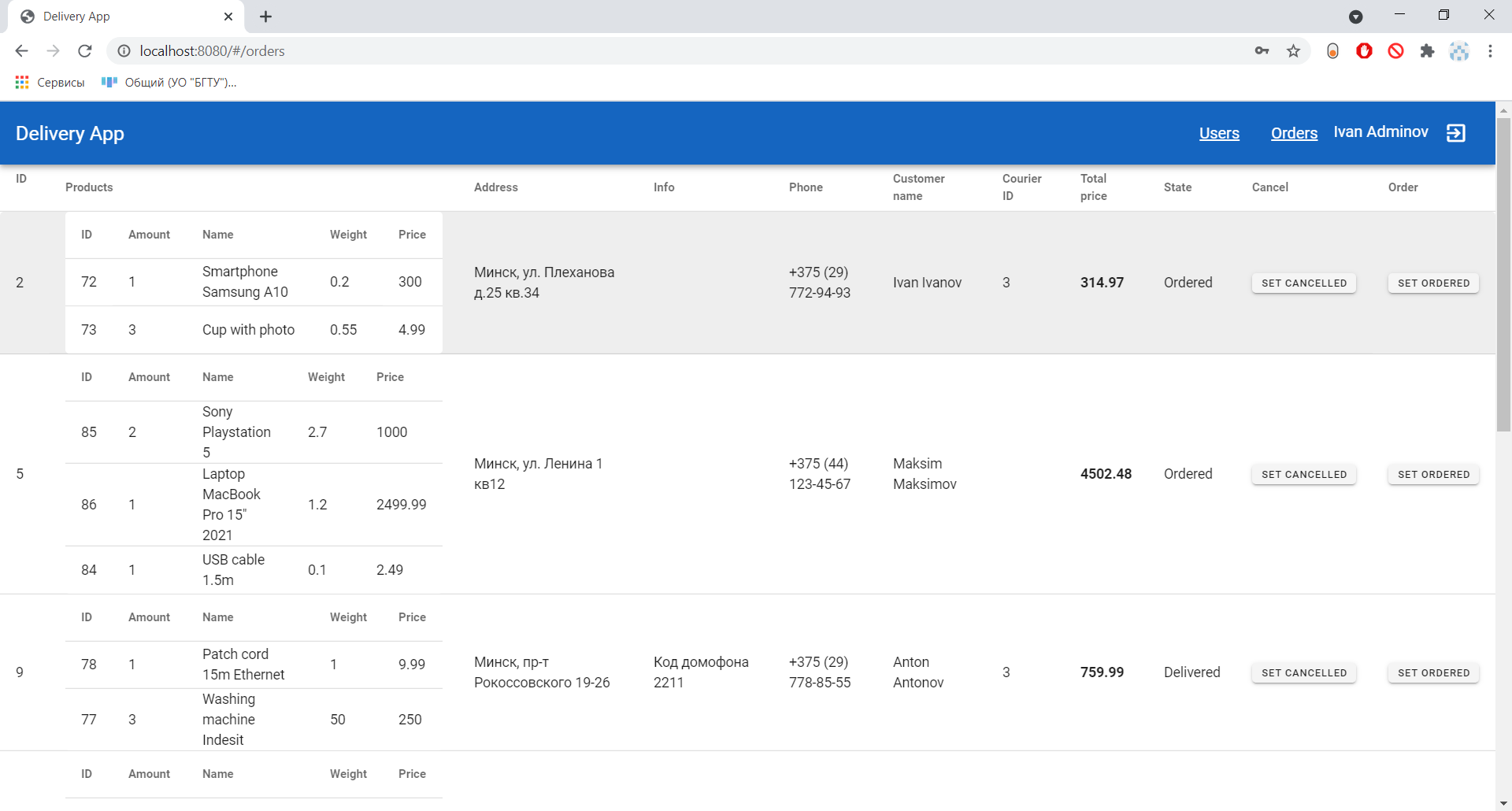


Рисунок 3.3 – Заказ №2 принял курьер №3

Далее он отключает интернет на устройстве, и изменяет состояние заказа №2 на «Delivering» (Доставляется) (Рисунок 3.4).

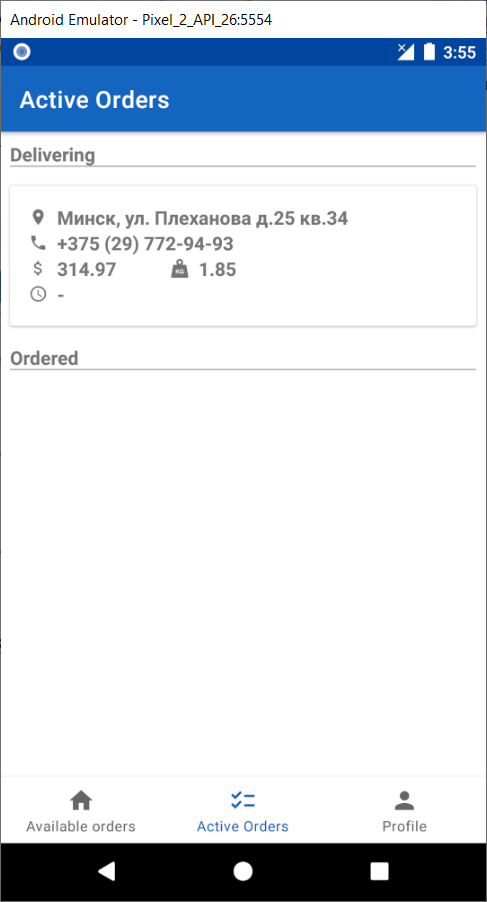


Рисунок 3.4 – Курьер изменил состояние заказа без подключения к сети

После этого администратор отменяет заказ, установив соответствующее состояние заказа. После этого курьер устанавливает соединение, но состояние заказа остается отмененным (Рисунок 3.5).

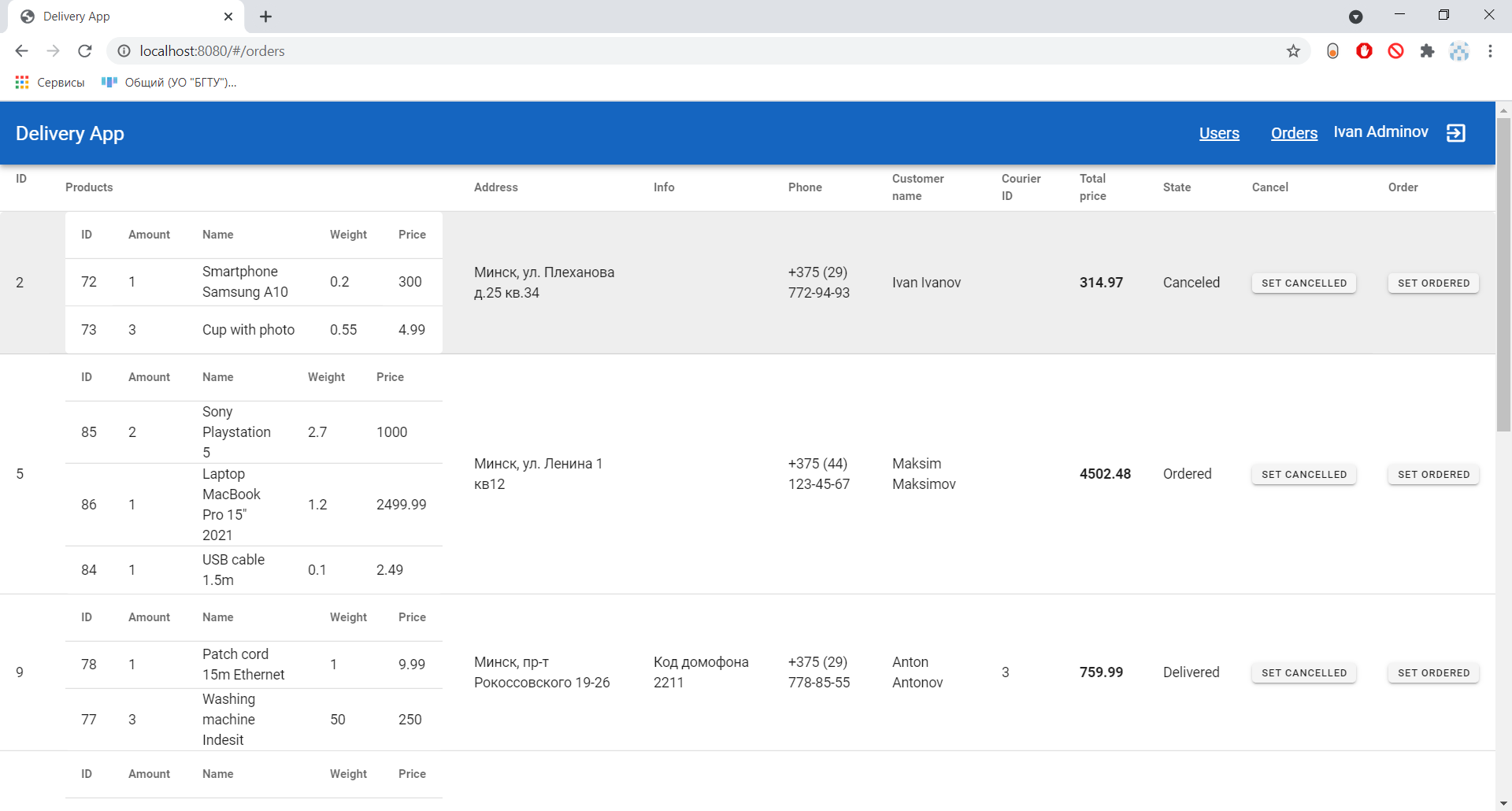


Рисунок 3.5 – Состояние заказа осталось отмененным

# Негативное тестирование

Негативное тестирование – это один из видов тестирования, который позволяет проверить систему на некорректное поведение. В ходе такого тестирования мы можем узнать, что система справится с непредвиденными ситуациями.

Сценарий: уже зарегистрированный ранее пользователь пытается произвести процедуру регистрации.

При попытке регистрации пользователя с электронной почтой существующего, возникает ошибка (Рисунок 3.6).

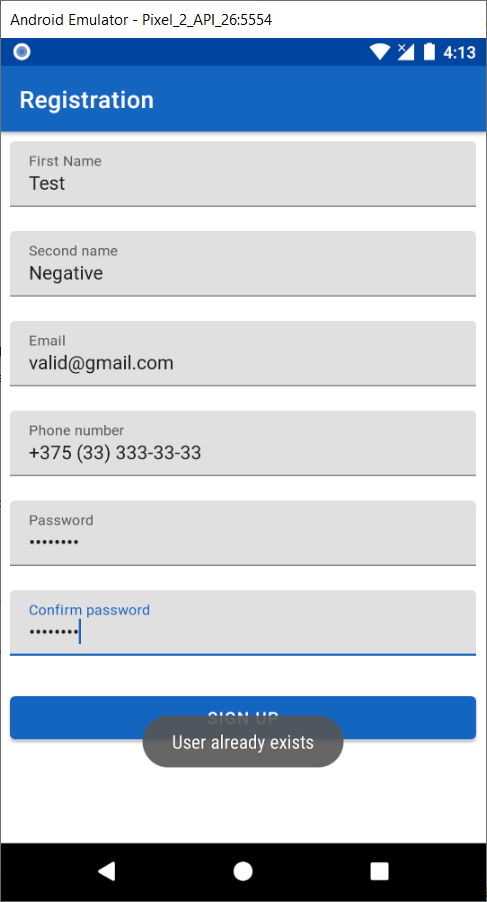


Рисунок 3.6 – Сообщение ошибки при попытке регистрации второго аккаунта на электронную почту

Также рассмотрим еще один негативный сценарий: пользователь без роли курьера (не подтвержденный) заходит в мобильное приложение и пытается принять заказ.

При входе в мобильное приложение с использованием аккаунта без соответствующей роли список доступных заказов не отображается, а вместо него появляется соответствующее сообщение об ошибке (Рисунок 3.7)

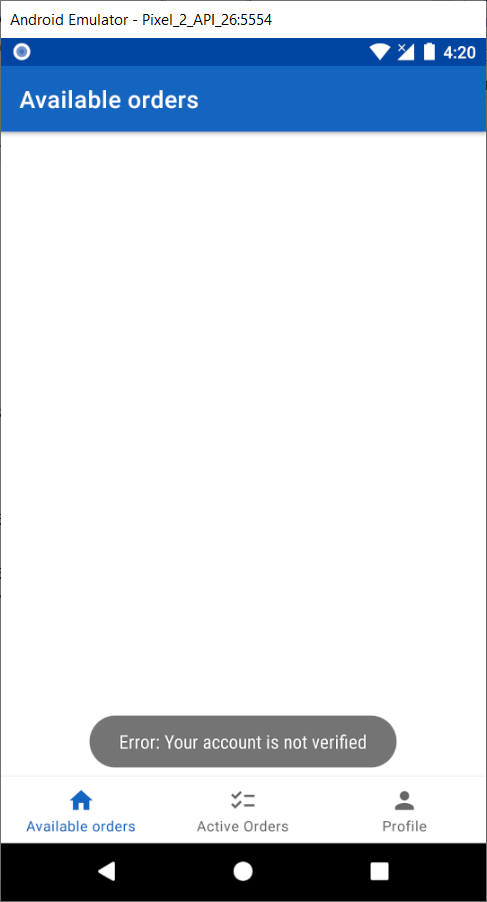


Рисунок 3.8 – Сообщение об ошибке при входе с неподтвержденного аккаунта

# Руководство пользователя

# Мобильное приложение

При первом входе в приложение открывается экран Входа в аккаунт (Рисунок 4.1). Здесь можно войти в приложение, введя логин и пароль, и нажав кнопку «LOGIN». Если логин и пароль верны, то осуществится вход в приложение на экран доступных заказов. Если нет, то появится соответствующее сообщение.

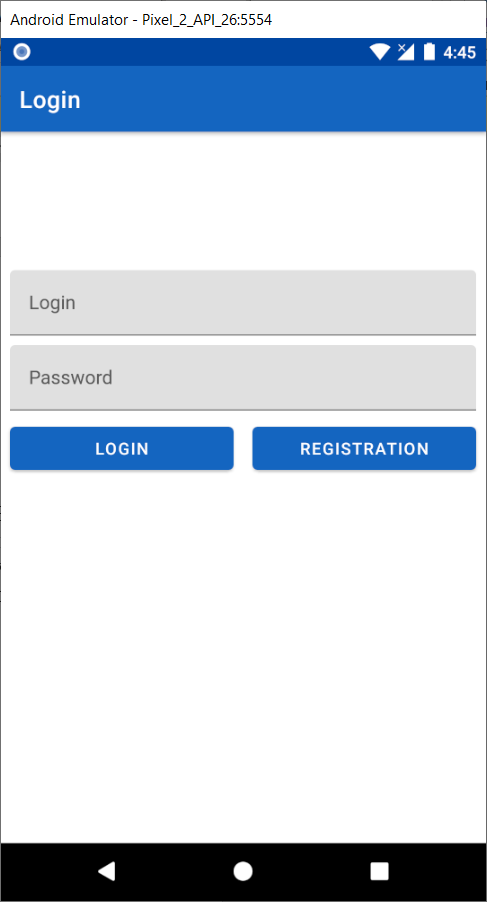


Рисунок 4.1 – Экран входа в аккаунт

При нажатии на кнопку «REGISTRATION» произойдет переход на экран регистрации (Рисунок 4.2). На экране регистрации при вводе корректных данных и нажатии на кнопку «SIGN UP» произойдет регистрация с выводом соответствующего сообщения и возвратом на экран Входа в аккаунт. При вводе некорректных данных появится сообщение об ошибке.

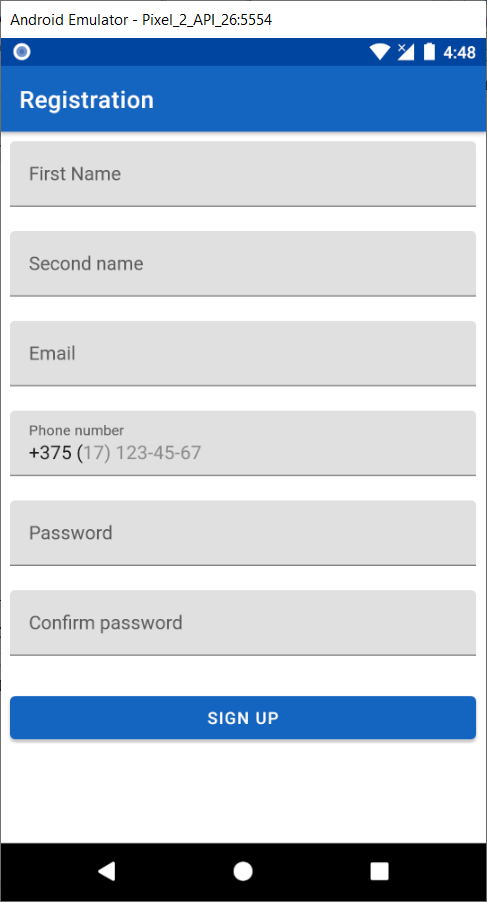


Рисунок 4.2 – Экран регистрации

При успешном входе в аккаунт или при запуске приложения (если до этого уже осуществлялся вход в аккаунт) откроется экран доступных заказов (Рисунок 4.3). В случае, если текущий аккаунт подтвержден, на этом экране отобразится список доступных заказов, в противном случае появится соответствующее сообщение об ошибке. В нижней части экрана есть навигационное меню для перехода между основными экранами.

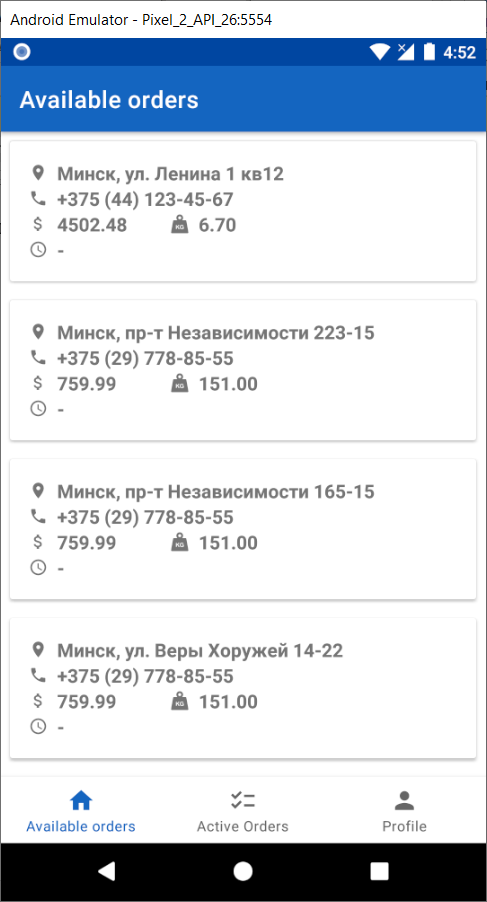


Рисунок 4.3 – экран с доступными заказами

При нажатии на заказ из списка доступных заказов откроется экран Деталей доступного заказа (Рисунок 4.4). На этом экране расположена информация о заказе, а также несколько кнопок. При нажатии на кнопку «CALL» откроется приложение Контакты с введенным номером телефона для звонка. При нажатии на кнопку «SHOW ON MAP» откроется приложение Google maps с введенным адресом. При нажатии на круглую кнопку в нижнем правом углу произойдет принятие заказа и возврат на экран доступных заказов.

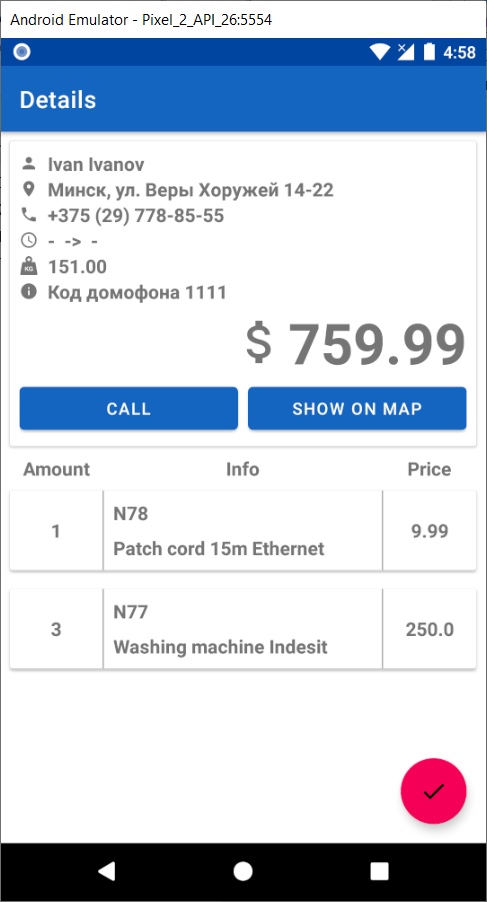


Рисунок 4.4 – Экран Деталей доступного заказа

При нажатии на кнопку «Active Orders» в нижнем меню произойдет переход на экран Активных заказов (Рисунок 4.5). Здесь будут находиться два списка заказов: принятые (Ordered) и доставляемые (Delivering). Информация на этом экране так же доступна только подтвержденным курьерам. Если аккаунт не имеет роли ROLE\_COURIER, то появится сообщение об ошибки. Также, этот экран, в отличие от экрана Доступных заказов, поддерживает автономную работу (без подключения к сети), т.е. список активных заказов курьера выведется на экран в любом случае.

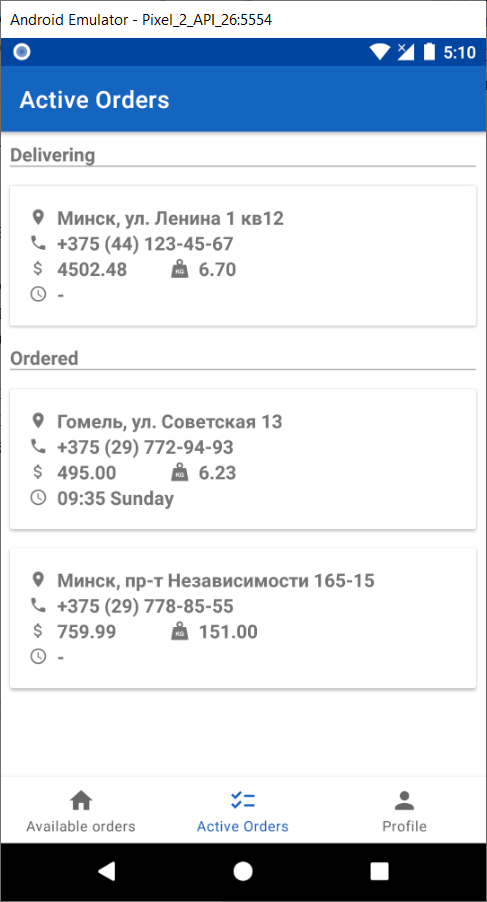


Рисунок 4.5 – Экран Активных заказов

При нажатии на заказ из списка активных заказов откроется экран Деталей активного заказа. На нем также расположена информация о заказе, однако присутствуют 4 кнопки:

* CALL – при нажатии откроется приложение Контакты с введенным номером телефона для звонка;
* SHOW ON MAP – при нажатии откроется приложение Google maps с введенным адресом;
* SET STATE – при нажатии изменяется состояние следующим образом: если текущее состояние Ordered, то оно меняется на Delivering, если Delivering, то меняется на Delivered. После изменения состояния осуществляется возврат на экран Активных заказов;
* DECLINE – находится в правом нижнем углу, служит для отказа от заказа. После отказа осуществляется переход на экран Активных заказов.

Особенностью этого экрана так же является его автономность. При отсутствии соединения курьер также может изменять состояние заказа или отказаться от него. При появлении соединения с сетью, произойдет автоматическая синхронизация с сервером. Вид экрана представлен на рисунке 4.6.

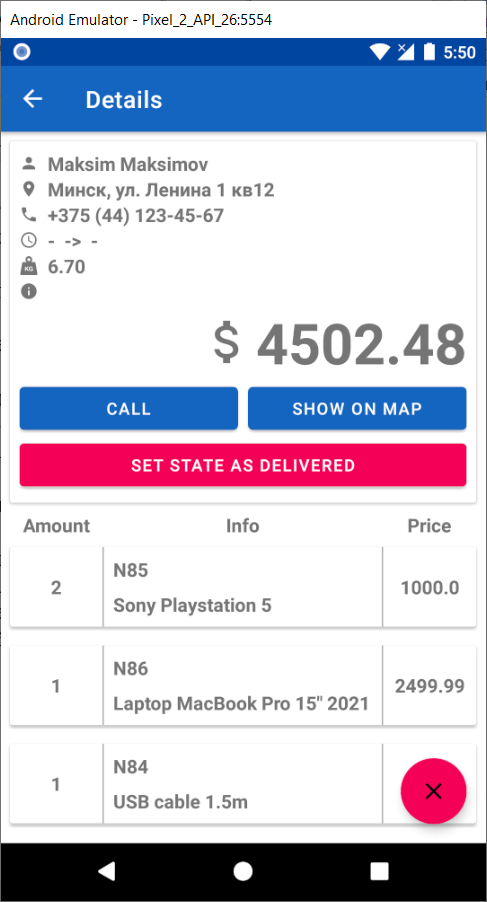


Рисунок 4.6 – Экран Деталей активного заказа

Последним экраном мобильного приложения является экран профиля (Рисунок 4.7). Для перехода на него необходимо нажать на 3 кнопку с надписью «Profile» в нижнем меню. На этом экране расположена информация о текущем пользователе, а также его статистика. При нажатии на кнопку «LOG OUT» происходи выход из учетной записи и переход на экран Входа.

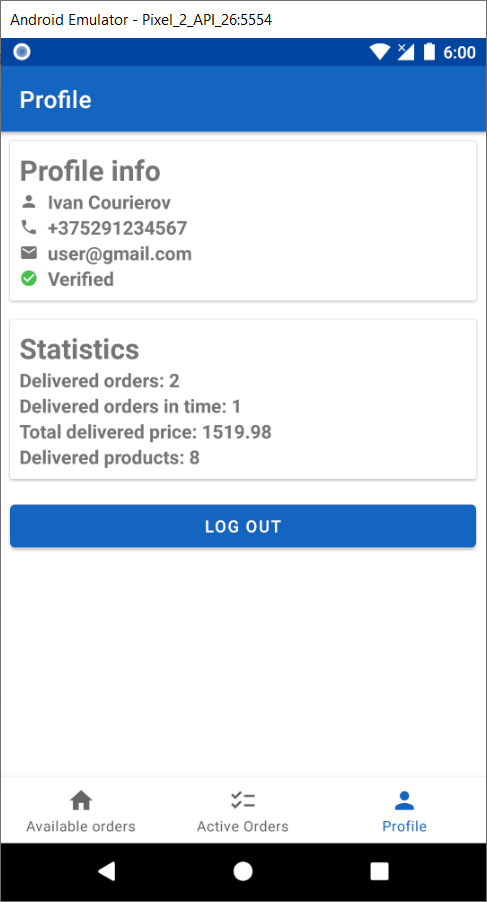


Рисунок 4.7 – Экран Профиля

# Веб-приложение

При первом запуске приложения для страниц, требующих аутентификацию, на экране появится соответствующее сообщение (Рисунок 4.8) с кнопкой «Login» для перехода на страницу входа. На страницу входа можно так же попасть, нажав на соответствующую иконку справа в шапке приложения.

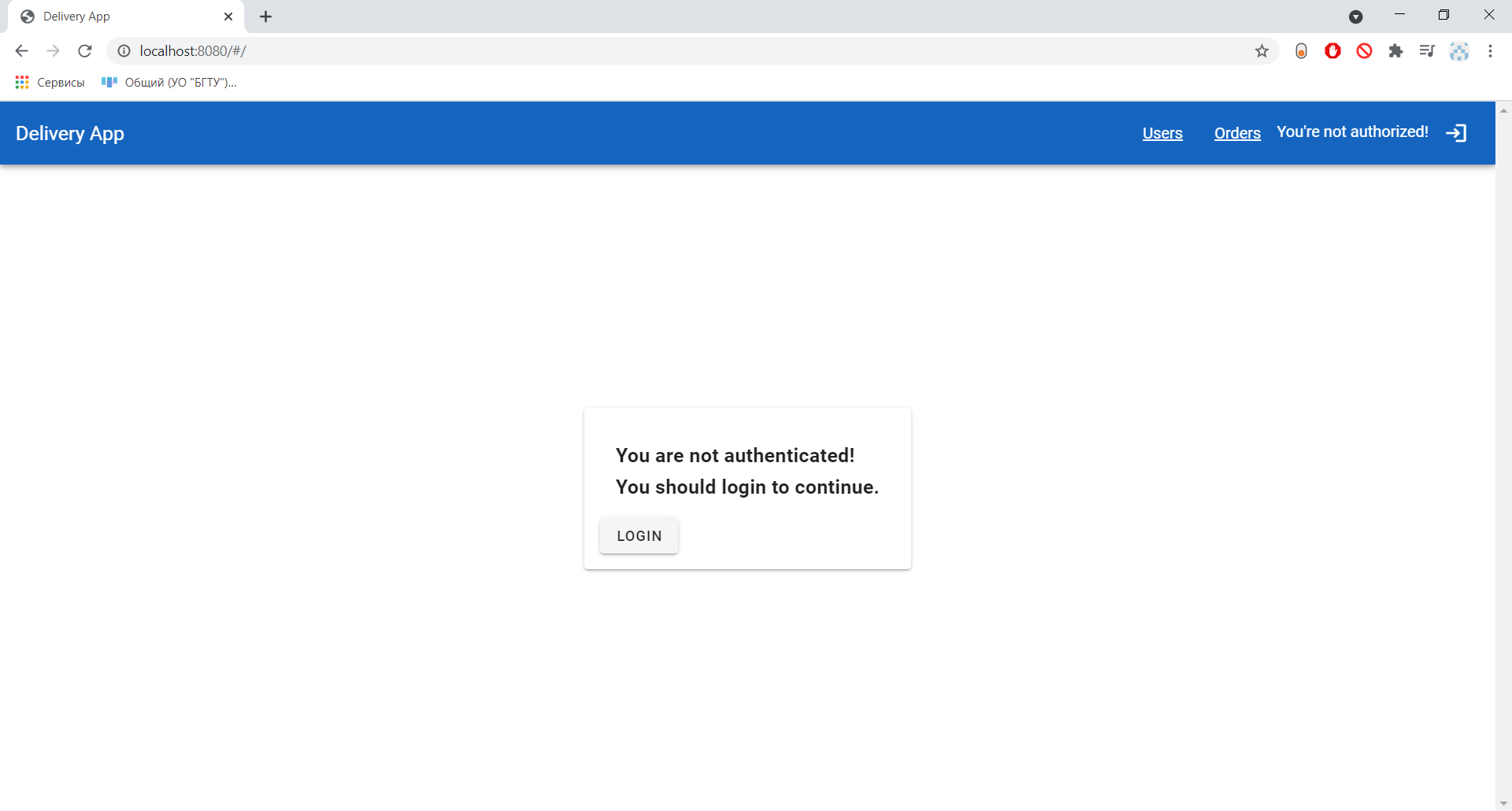


Рисунок 4.8 – Сообщение об ошибке

Страницы входа и регистрации аналогичны соответствующим экранам мобильного приложения за тем исключением, что это приложение предназначено только для администраторов и при попытке входа при помощи аккаунта не администратора возникнет ошибка.

При успешной аутентификации и авторизации пользователю доступны две страницы: страница пользователей и страница заказов. Переключаться между страницами можно по ссылками в шапке приложения.

На странице пользователей (Рисунок 4.9) присутствует таблица пользователей приложения, позволяющая выдавать роли при нажатии на соответствующие кнопки.

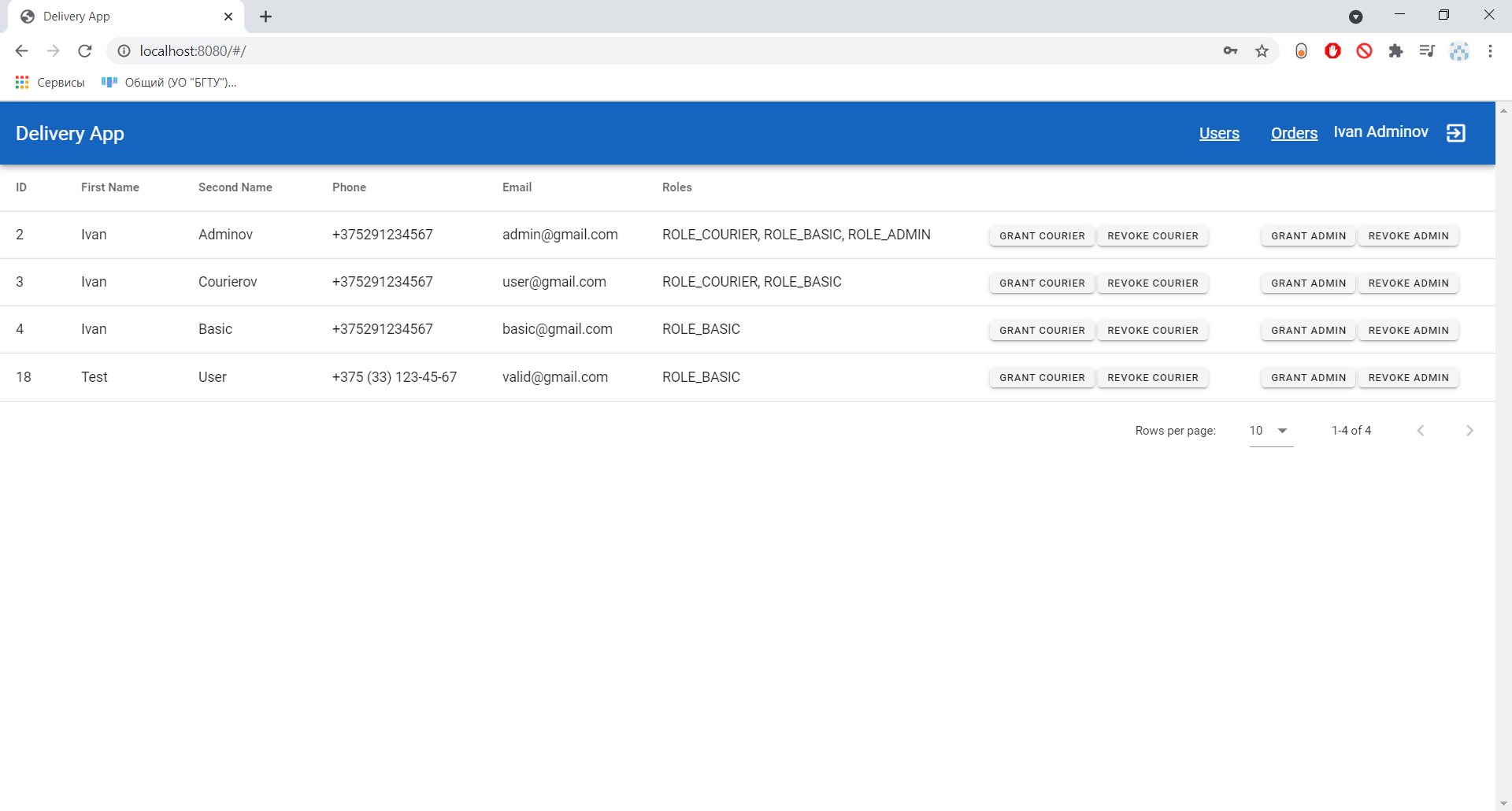


Рисунок 4.9 – Страница пользователей

Также при нажатии на строку в таблице появится окно со статистикой соответствующего пользователя (Рисунок 4.10).

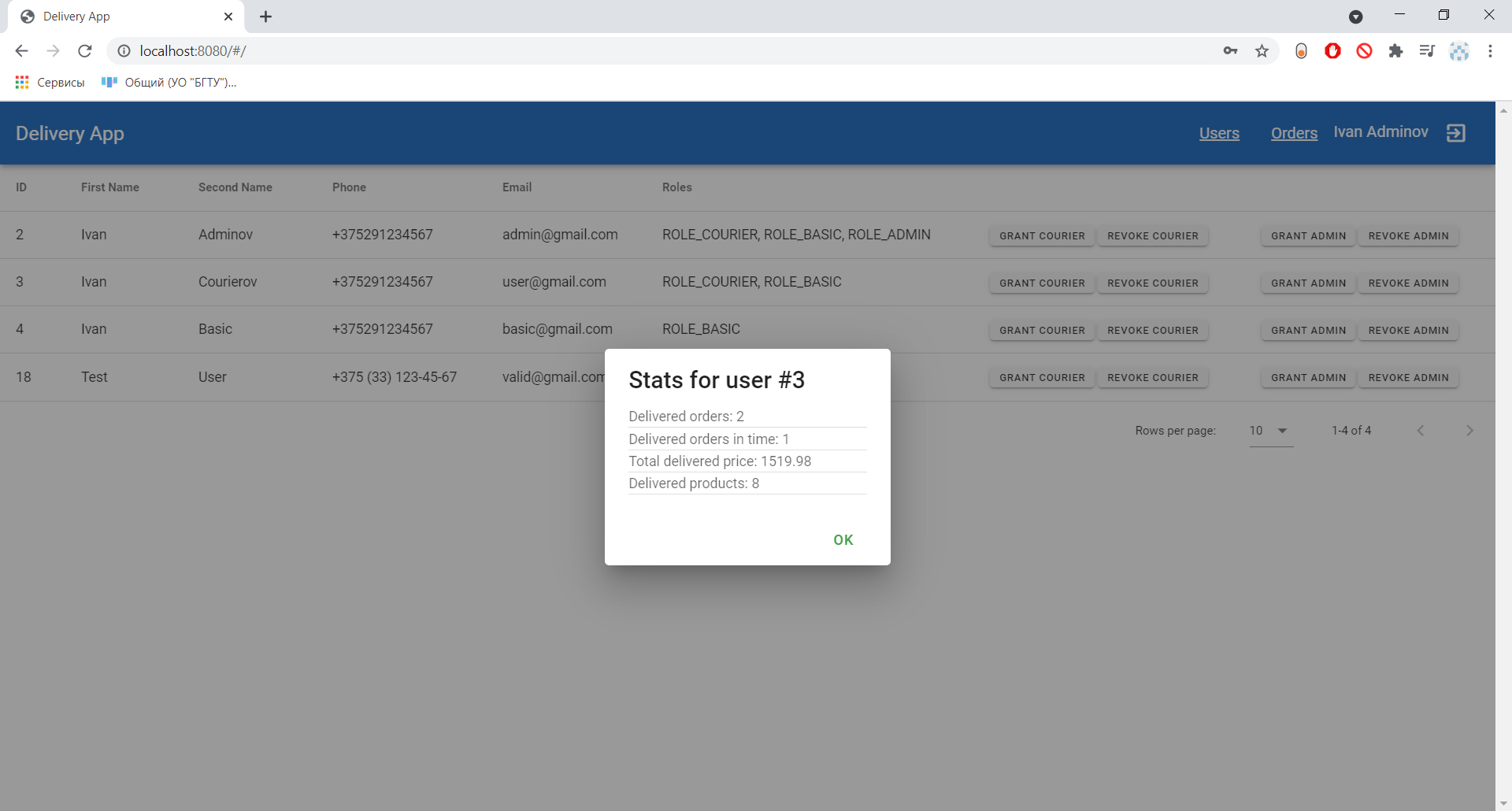


Рисунок 4.10 – Окно статистики пользователя

На странице заказов (Рисунок 4.11) размещена таблица с информацией о заказах. Для каждой строки в таблице заданы две кнопки: «Set Canceled» и «Set Ordered». При нажатии на кнопку «Set Canceled» соответствующему заказу задается состояние «Отменен», а при нажатии на «Set Ordered» заказ принимает состояние по умолчанию (Заказан) с той целью, чтобы его можно было сделать доступным для курьеров.

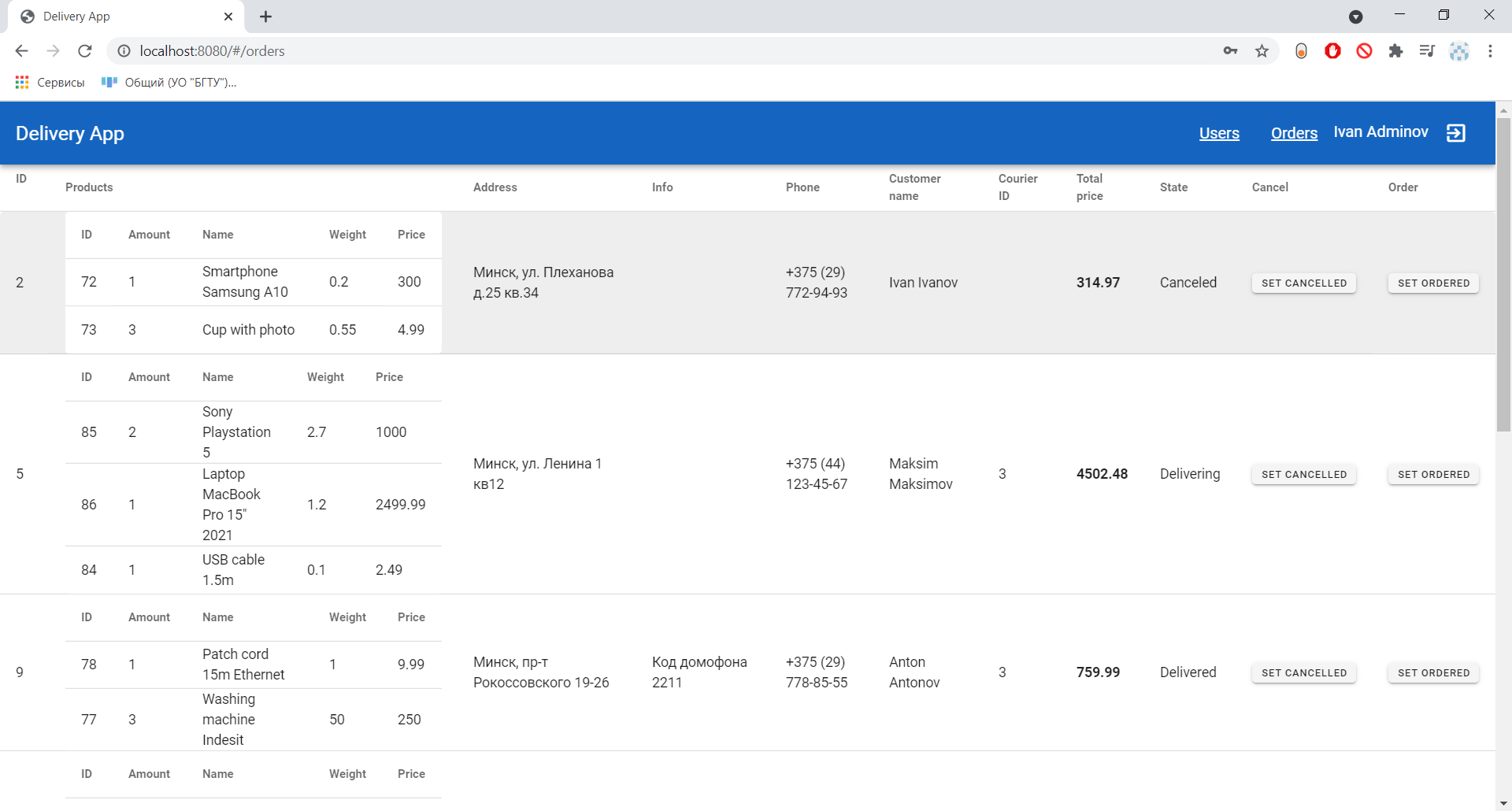


Рисунок 4.11 – Страница заказов

Последняя страница сайта отображает состояние заказа, доступна всем пользователям сайта, ссылка на нее передается в уведомлении заказчика по электронной почте при оформлении заказа (Рисунок 4.12).

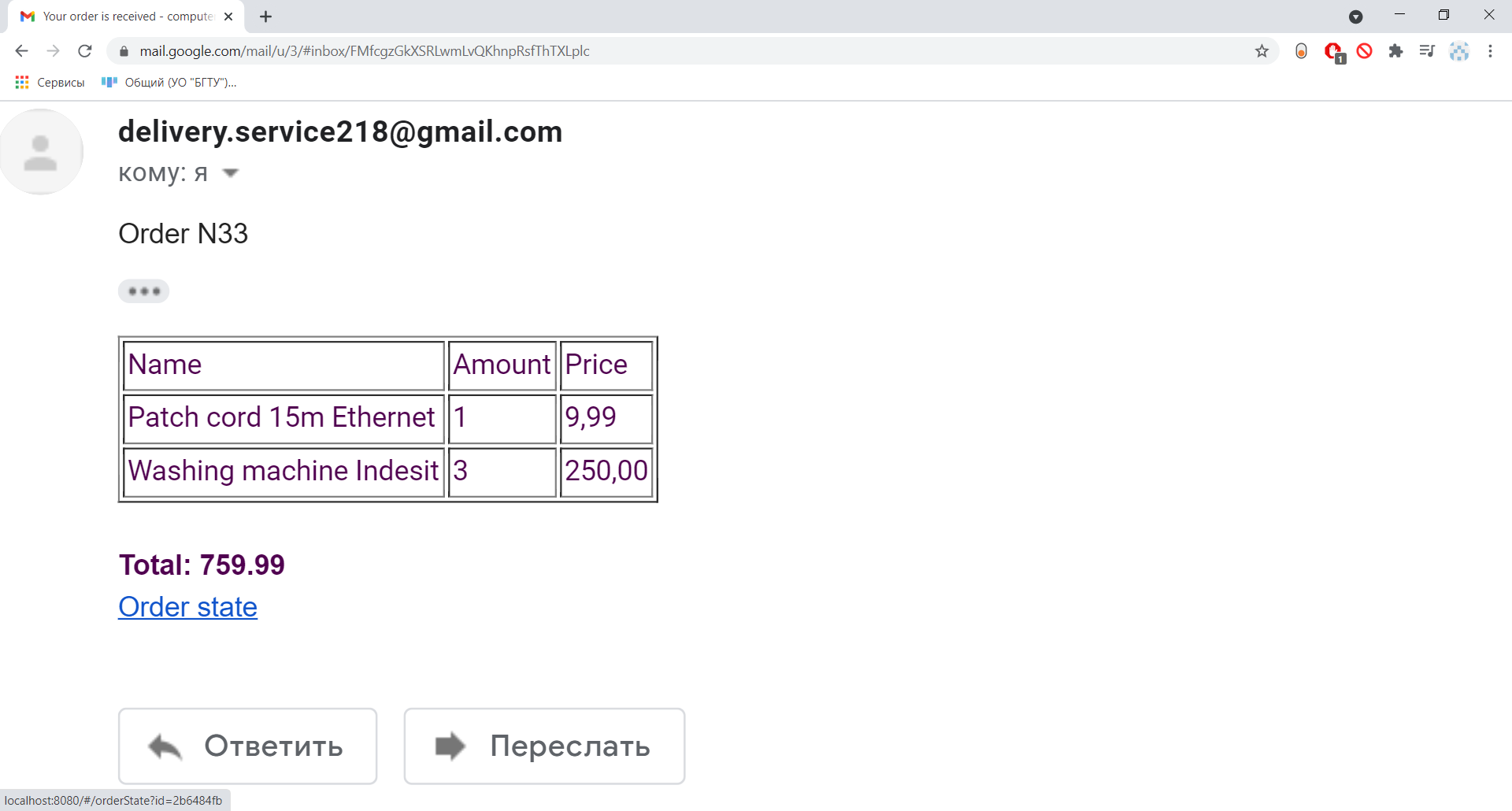


Рисунок 4.12 – Уведомление заказчика

Помимо ссылки на страницу состояния, передается параметром идентификатор заказа, зашифрованный сервером, и в последствии им и расшифруемый. Необходимо это для того, чтобы пользователи не могли напрямую обращаться к состоянию чужих заказов по их идентификатору.

Страница состояния заказа приведена на рисунке 4.13. Ее взаимодействие с сервером осуществляется по протоколу SOAP.

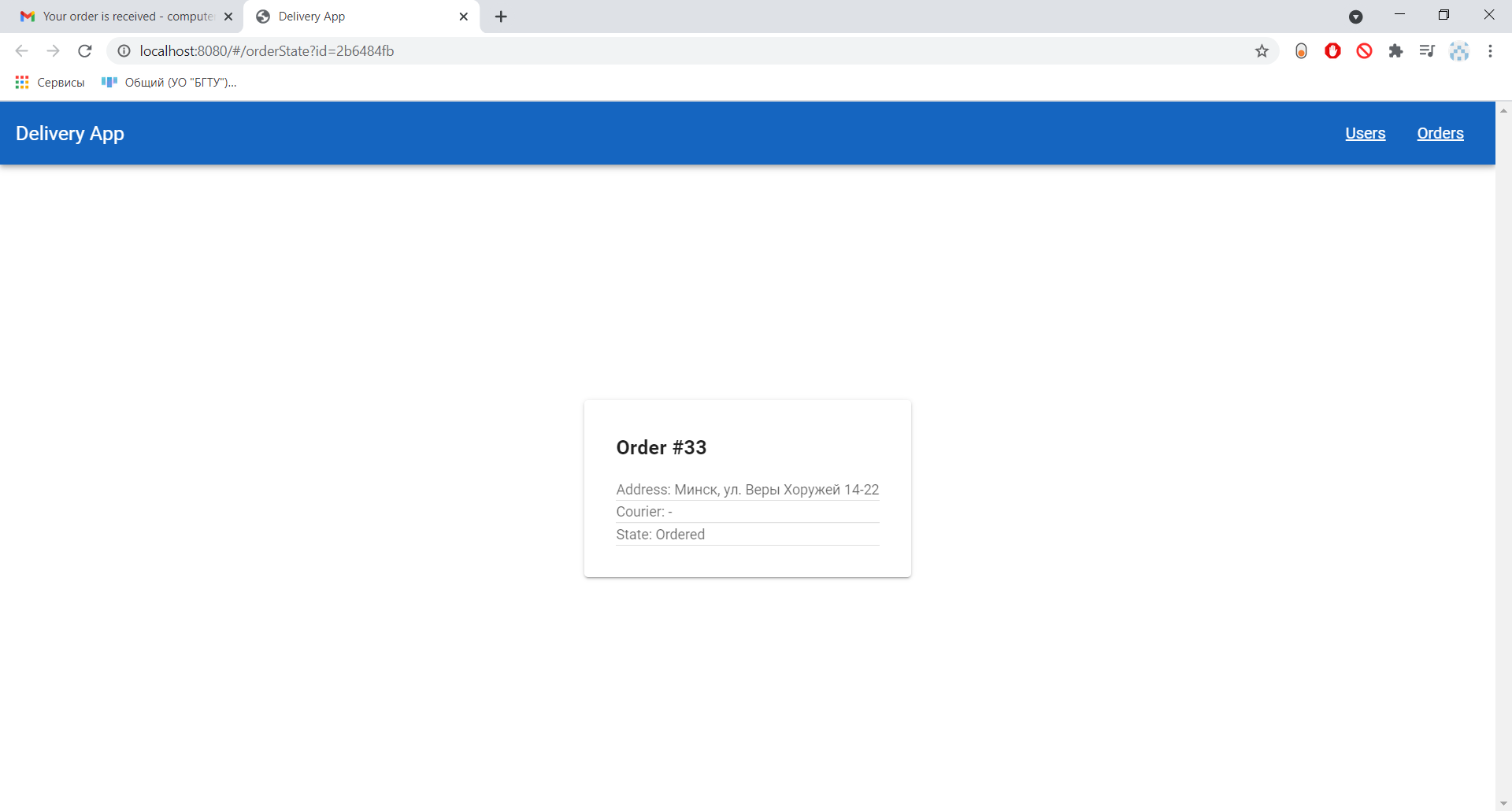


Рисунок 4.13 – Информация на странице состояния заказа

# Заключение

В процессе разработки была достигнута поставленная цель по созданию программного средства «Приложение службы доставки магазина», которое способно выполнять заявленные функции. Одной из основных целей курсового проекта стало проектирование базы данных для дальнейшей интеграции с приложением, которое помогло облегчить взаимодействие с базой данных посредством программного интерфейса. При разработке программного средства реализованы следующие пункты:

* добавление, удаление, редактирование информации о заказах;
* отслеживание состояния заказа;
* уведомление клиентов о состоянии заказа;
* сбор статистики;
* работу с мобильным приложением без доступа к сети;
* регистрацию курьеров.

Было успешно проведено негативное и позитивное тестирование.

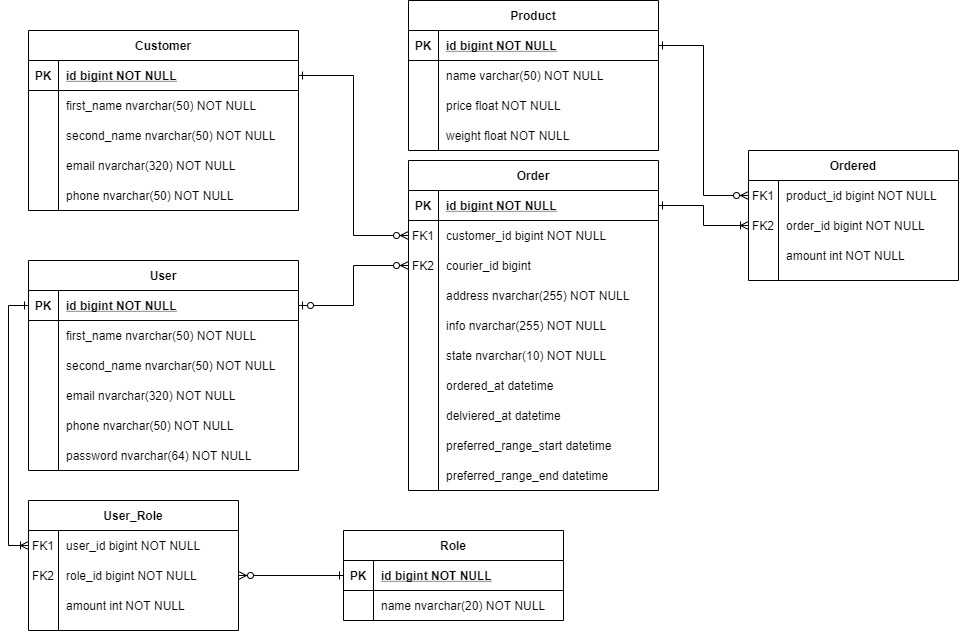
Составлено руководство пользователя.

В соответствии с полученным результатом работы программы можно сделать вывод, что разработанное программное средство работает верно, а требования технического задания выполнены в полном объёме.

# Список используемых источников

1. Spring Tutorial [Электронный ресурс] – Режим доступа: https://www.baeldung.com/spring-tutorial. - Дата доступа:14.05.2021
2. Android Documentation [Электронный ресурс] – Режим доступа: https://developer.android.com/docs. - Дата доступа: 07.05.2021
3. Vue js [Электронный ресурс] – Режим доступа: https://vuejs.org/. - Дата доступа: 12.05.2021
4. Vuetify [Электронный ресурс] – Режим доступа: https://vuetifyjs.com/en/. - Дата доступа: 15.05.2021
5. Spring Security [Электронный ресурс] – Режим доступа: https://spring.io/projects/spring-security. - Дата доступа: 17.04.2021
6. Spring Producing SOAP web-service [Электронный ресурс] – Режим доступа: https://spring.io/guides/gs/producing-web-service/. - Дата доступа: 04.05.2021

# Приложение А



# Приложение Б

CREATE PROCEDURE [dbo].[delivered\_orders\_amount\_by\_courier\_id]

@id bigint = 0,

@count int = 0 output

AS

BEGIN

set @count = (select count(\*) from orders where courier\_id = @id and state = 'Delivered')

END

CREATE PROCEDURE [dbo].[delivered\_orders\_in\_time\_by\_courier\_id]

@id bigint = 0,

@count int = 0 output

AS

BEGIN

set @count = (select count(\*) from orders where ((preferred\_range\_start is null and preferred\_range\_end is null)

or (delivered\_at between preferred\_range\_start and preferred\_range\_end)) and courier\_id = @id and state = 'Delivered')

END

CREATE PROCEDURE [dbo].[total\_delivered\_price\_by\_courier\_id]

@id bigint = 0,

@total real = 0 output

AS

BEGIN

set @total = (select sum(table1.amount \* table1.price)

from (

select amount, (select price from product where product.id = ordered.product\_id) as price

from ordered

where ordered.order\_id IN (select id from orders where courier\_id = @id AND [state] = 'Delivered')

) as table1)

END

CREATE PROCEDURE [dbo].[total\_delivered\_product\_amount\_by\_courier\_id]

@id bigint = 0,

@amount int = 0 output

AS

BEGIN

set @amount = (select sum(amount) from ordered where ordered.order\_id IN (select id from orders where courier\_id = @id AND [state] = 'Delivered'))

END

# Приложение В

CREATE trigger [dbo].[tr\_change\_orders] on [dbo].[orders] after INSERT, DELETE, UPDATE

as

declare @id bigint, @table\_name varchar(10) = 'orders';

declare @ins int = (select count(\*) from inserted),

@del int = (select count(\*) from deleted);

if @ins > 0 and @del = 0

begin

set @id = (select id from inserted)

if (select count(\*) from changes where table\_name = @table\_name and item\_id = @id) = 0

begin

insert into [changes](table\_name, item\_id, operation) values(@table\_name, @id, 'insert');

end;

else

begin

update changes set

operation = 'insert',

time\_stamp = getdate()

where table\_name = @table\_name and item\_id = @id;

end;

end;

else

if @ins = 0 and @del > 0

begin

set @id = (select id from deleted)

if (select count(\*) from changes where table\_name = @table\_name and item\_id = @id) = 0

begin

insert into [changes](table\_name, item\_id, operation) values(@table\_name, @id, 'delete');

end;

else

begin

update changes set

operation = 'delete',

time\_stamp = getdate()

where table\_name = @table\_name and item\_id = @id;

end;

end;

else

if @ins > 0 and @del > 0

begin

set @id = (select id from inserted)

if (select count(\*) from changes where table\_name = @table\_name and item\_id = @id) = 0

begin

insert into [changes](table\_name, item\_id, operation) values(@table\_name, @id, 'update');

end;

else

begin

update changes set

operation = 'update',

time\_stamp = getdate()

where table\_name = @table\_name and item\_id = @id;

end;

end;

return;

# Приложение Г

object OrderService {  
 private final val TAG = "OrderService"  
 private val orderApi = NetworkRepository.orderApi()  
 private val db: CourierDatabase = CourierDatabase.getDatabase(*context*)  
  
 *//Получение списка доступных заказов (только онлайн)* fun getAvailableOrders(ordersLiveData: MutableLiveData<Event<List<Order>>>) {  
 ordersLiveData.postValue(Event.loading())  
 NetworkRepository.orderApi().*availableOrders*.enqueue(object: Callback<List<OrderDto>> {  
 override fun onResponse(call: Call<List<OrderDto>>, response: Response<List<OrderDto>>) {  
 try {  
 when (response.code()) {  
 *//Успешное получение данных* 200 -> {  
 val mapper = Mappers.getMapper(OrderMapper::class.*java*)  
 val orders = mapper.dtosToEntities(response.body())  
 ordersLiveData.postValue(Event.success(orders))  
 }  
 *//Возвращается сервером, если пользователь не имеет роли курьера* 403 -> {  
 throw CourierNetworkException("Your account is not verified")  
 }  
 else -> {  
 throw CourierNetworkException("Network Troubles")  
 }  
 }  
 } catch (e: CourierNetworkException) {  
 ordersLiveData.postValue(Event.error(e))  
 }  
 }  
  
 override fun onFailure(call: Call<List<OrderDto>>, t: Throwable) {  
 ordersLiveData.postValue(Event.error(t))  
 }  
 })  
 }  
  
 *//Получение списка активных заказов (доступно без подключения к интернету)* fun getActiveOrders(ordersLiveData: MutableLiveData<Event<List<Order>>>) {  
 ordersLiveData.postValue(Event.loading())  
 *//Если подключение к интернету есть* if (isOnline(*context*)) {  
 NetworkRepository.orderApi().*activeOrders*.enqueue(object: Callback<List<OrderDto>> {  
 override fun onResponse(call: Call<List<OrderDto>>, response: Response<List<OrderDto>>) {  
 try {  
 when (response.code()) {  
 *//Успешное получение данных* 200 -> {  
 *CoroutineScope*(Dispatchers.IO).*launch* **{** val mapper = Mappers.getMapper(OrderMapper::class.*java*)  
 val orders = mapper.dtosToEntities(response.body())  
 *//Обновление локальной базы данных  
 //Очистка записей связанных с заказами* db.orderedDao.truncate()  
 db.orderDao.truncate()  
 *//Вставка актуальной информации о активных заказах в локальную бд* for (order in orders) {  
 insertOrderWithReplaceToLocalDb(order)  
 }  
 *//Получение списка активных заказов из локальной бд* ordersLiveData.postValue(Event.success(getActiveOrdersFromLocalDb()))  
 **}** }  
 *//Возвращается сервером, если пользователь не имеет роли курьера* 403 -> {  
 throw CourierNetworkException("Your account is not verified")  
 }  
 else -> {  
 throw CourierNetworkException("Network Troubles")  
 }  
 }  
 } catch (e: CourierNetworkException) {  
 ordersLiveData.postValue(Event.error(e))  
 }  
 }  
  
 override fun onFailure(call: Call<List<OrderDto>>, t: Throwable) {  
 ordersLiveData.postValue(Event.error(t))  
 }  
 })  
 } else {  
 *CoroutineScope*(Dispatchers.IO).*launch* **{** *//Получение списка активных заказов из локальной бд* ordersLiveData.postValue(Event.success(getActiveOrdersFromLocalDb()))  
 **}** }  
  
  
 }  
  
 *//Принять заказ (только онлайн)* fun accept(orderLiveData: MutableLiveData<Order>, responseLiveData: MutableLiveData<Event<ResponseBody>>) {  
 responseLiveData.postValue(Event.loading())  
 orderLiveData.*value*?.*let* **{** order **->** NetworkRepository.orderApi().acceptOrder(order.*id*).enqueue(object: Callback<ResponseBody>{  
 override fun onResponse(call: Call<ResponseBody>, response: Response<ResponseBody>) {  
 when (response.code()) {  
 200 -> {  
 *CoroutineScope*(Dispatchers.IO).*launch* **{** insertOrderWithReplaceToLocalDb(order)  
 responseLiveData.postValue(Event.success(response.body()))  
 **}** }  
 else -> {  
 val gson = Gson()  
 val jsonObject: JsonObject = gson.fromJson(response.errorBody()?.string(), JsonObject::class.*java*)  
 responseLiveData.postValue(Event.error(CourierNetworkException(jsonObject["message"].*asString*)))  
 }  
 }  
 }  
  
 override fun onFailure(call: Call<ResponseBody>, t: Throwable) {  
 responseLiveData.postValue(Event.error(t))  
 }  
  
 })  
 **}** }  
  
 *//Отмена заказа (доступно без подключения к интернету)* suspend fun decline(orderLiveData: MutableLiveData<Order>, responseLiveData: MutableLiveData<Event<ResponseBody>>) {  
 orderLiveData.*value*?.*let* **{** *//Удаляем запись о заказе из локальной бд  
 //При удалении в таблице changes появится соответствующая запись, при возобновлении соединения,  
 //по этой записи можно будет отменить заказ на сервере для синхронизации* deleteOrderAndOrderedFromLocalDb(**it**)  
 *//Если есть интернет, то выполняем отмену заказа и на сервере* if (isOnline(*context*)) {  
 responseLiveData.postValue(Event.loading())  
 NetworkRepository.orderApi().declineOrder(**it**.*id*).enqueue(object: Callback<ResponseBody>{  
 override fun onResponse(call: Call<ResponseBody>, response: Response<ResponseBody>) {  
 when (response.code()) {  
 200 -> responseLiveData.postValue(Event.success(response.body()))  
 else -> {  
 val gson = Gson()  
 val jsonObject: JsonObject = gson.fromJson(response.errorBody()?.string(), JsonObject::class.*java*)  
 responseLiveData.postValue(Event.error(CourierNetworkException(jsonObject["message"].*asString*)))  
 }  
 }  
 }  
  
 override fun onFailure(call: Call<ResponseBody>, t: Throwable) {  
 responseLiveData.postValue(Event.error(t))  
 }  
  
 })  
 } else {  
 responseLiveData.postValue(Event.success(null))  
 }  
 **}** }  
  
 suspend fun updateState(orderLiveData: MutableLiveData<Order>, responseLiveData: MutableLiveData<Event<ResponseBody>>) {  
 orderLiveData.*value*?.*let* **{** val nextState = **it**.*state*.*next* db.orderDao.updateState(**it**.*id*, nextState.name)  
  
 if (isOnline(*context*)) {  
 responseLiveData.postValue(Event.loading())  
 NetworkRepository.orderApi().updateState(**it**.*id*, nextState.name).enqueue(object: Callback<ResponseBody>{  
 override fun onResponse(call: Call<ResponseBody>, response: Response<ResponseBody>) {  
 when (response.code()) {  
 200 -> responseLiveData.postValue(Event.success(response.body()))  
 else -> {  
 val gson = Gson()  
 val jsonObject: JsonObject = gson.fromJson(response.errorBody()?.string(), JsonObject::class.*java*)  
 responseLiveData.postValue(Event.error(CourierNetworkException(jsonObject["message"].*asString*)))  
 }  
 }  
 }  
  
 override fun onFailure(call: Call<ResponseBody>, t: Throwable) {  
 responseLiveData.postValue(Event.error(t))  
 }  
  
 })  
 } else {  
 responseLiveData.postValue(Event.success(null))  
 }  
 **}** }  
  
  
 *//Метод вызывается при появлении соединения для отправки информации о заказах, от которых курьер отказался* suspend fun sendDecline() {  
 *//Получаем список ID заказов, которые были удалены из таблицы заказов во время оффлайн использования* val declinedOrdersIdList = db.changeDao.findAllByTableAndOperation("orders", "delete").*map* **{ it**.*itemId* **}** Log.d(TAG, "sendDecline: declined in offline - $declinedOrdersIdList")  
 for (id in declinedOrdersIdList) {  
 *//Отправляем соответствующий отмене заказа запрос на сервер* NetworkRepository.orderApi().declineOrder(id).enqueue(object: Callback<ResponseBody>{  
 override fun onResponse(call: Call<ResponseBody>, response: Response<ResponseBody>) {  
 Log.d(TAG, "sendDecline status: " + response.code())  
 }  
  
 override fun onFailure(call: Call<ResponseBody>, t: Throwable) {  
 Log.d(TAG, "sendDecline error: " + t.message)  
 }  
  
 })  
 *//Удаляем запись в таблице изменений для предотвращения последующего повторения* db.changeDao.deleteByTableAndItemId("orders", id)  
 }  
 }  
  
 *//Метод вызывается при появлении соединения для отправки информации о заказах, состояние которых было обновлено курьером* suspend fun sendUpdateState() {  
 *//Получаем список заказов, состояние которых было обновлено во время оффлайн использования* val updateStateOrdersIdList = db.changeDao.findAllByTableAndOperation("orders", "update").*map* **{ it**.*itemId* **}** Log.d(TAG, "sendUpdateState: updated in offline - $updateStateOrdersIdList")  
 for (id in updateStateOrdersIdList) {  
 *//Получаем состояние заказа из БД по id* val state = db.orderDao.getStateById(id)  
 *//Отправляем соответствующий обновлению состояния заказа запрос на сервер* NetworkRepository.orderApi().updateState(id, state).enqueue(object: Callback<ResponseBody>{  
 override fun onResponse(call: Call<ResponseBody>, response: Response<ResponseBody>) {  
 Log.d(TAG, "sendUpdate status: " + response.code())  
 }  
  
 override fun onFailure(call: Call<ResponseBody>, t: Throwable) {  
 Log.d(TAG, "sendUpdate error: " + t.message)  
 }  
  
 })  
 *//Удаляем запись в таблице изменений для предотвращения последующего повторения* db.changeDao.deleteByTableAndItemId("orders", id)  
 }  
 }  
  
 private suspend fun insertOrderWithReplaceToLocalDb(order: Order) {  
 db.customerDao.insertWithReplace(order.*customer*)  
 Log.d("OrderService", "customer")  
 db.productDao.insertAll(order.*ordered*.*map* **{ it**.*product* **}**)  
 Log.d("OrderService", "products $order")  
 db.orderDao.insert(order)  
 Log.d("OrderService", "order")  
 db.orderedDao.insertAll(order.*ordered*.*toList*())  
 Log.d("OrderService", "ordered")  
 db.changeDao.setUpToDate("orders", order.*id*)  
 Log.d("OrderService", "change")  
 }  
  
 private suspend fun deleteOrderAndOrderedFromLocalDb(order: Order) {  
 db.orderedDao.deleteAll(order.*ordered*)  
 db.orderDao.delete(order)  
 }  
  
 private suspend fun fillOrderFromDb(order: Order) {  
 order.*customer* = db.customerDao.findById(order.*customerId*)  
 val orderedList = db.orderedDao.findByOrderId(order.*id*)  
 for (ordered in orderedList) {  
 val product = db.productDao.findById(ordered.*productId*)  
 ordered.*product* = product  
 }  
 order.*ordered* = orderedList  
  
 }  
  
 private suspend fun getActiveOrdersFromLocalDb(): List<Order> {  
 *//Получаем активные заказы из локальной бд* val orders = db.orderDao.getAll()  
 for (order in orders) {  
 fillOrderFromDb(order)  
 }  
 return orders  
 }  
  
}