

Министерство образования Республики Беларусь

Учреждение образования  
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

К ЗАЩИТЕ ДОПУСТИТЬ  
Зав. каф. ЭВМ  
\_\_\_\_\_ Б.В. Никульшин

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА  
к дипломному проекту  
на тему  
ЭЛЕКТРОННО-ИНФОРМАЦИОННЫЙ СЕРВИС ПО ОКАЗАНИЮ УСЛУГ  
ПЕРЕВОЗКИ И ДОСТАВКИ

БГУИР ДП 1-40 02 01 01 107 ПЗ

Студент

В.М. Филиппович

Руководитель

А.М. Ковальчук

Консультанты:

от кафедры ЭВМ

А.М. Ковальчук

по экономической части

О.А. Матяс

Нормоконтролер

Е.Е. Клинцевич

Рецензент

МИНСК 2022

Министерство образования Республики Беларусь

Учреждение образования  
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет: ФКСиС. Кафедра: ЭВМ.

Специальность: 40 02 01 «Вычислительные машины, системы и сети».

Специализация: 40 02 01-01 «Проектирование и применение локальных компьютерных сетей».

УТВЕРЖДАЮ  
Заведующий кафедрой ЭВМ  
\_\_\_\_\_ Б.В.Никульшин  
«\_\_\_\_\_» 2022 г.

ЗАДАНИЕ  
по дипломному проекту студента  
Филипповича Вадима Максимовича

**1** Тема проекта: «Электронно-информационный сервис по оказанию услуг перевозки и доставки» – утверждена приказом по университету от 1 апреля 2022 г. № 892-с.

**2** Срок сдачи студентом законченного проекта: 1 июня 2022 г.

**3** Исходные требования к проекту:

3.1 Операционные системы: Android, iOS, Windows, MacOS, Linux.

3.2 Среда разработки: IntelijIDEA, XCode, Android Studio.

3.3 Язык программирования: TypeScript.

**4** Содержание пояснительной записи (перечень подлежащих разработке вопросов):

Введение 1. Обзор литературы. 2. Системное проектирование.  
3. Функциональное проектирование. 4. Разработка программных модулей.  
5. Программа и методика испытаний. 6. Руководство пользователя.  
7. Технико-экономическое обоснование эффективности разработки платформы управления приложением пакетной обработки данных.  
Заключение. Список использованных источников. Приложения.

**5** Перечень графического материала (с точным указанием обязательных чертежей):

- 5.1** Вводный плакат. Плакат.
- 5.2** Электронно-информационный сервис по оказанию услуг перевозки и доставки. Схема структурная.
- 5.3** Электронно-информационный сервис по оказанию услуг перевозки и доставки. Схема программы.
- 5.4** Электронно-информационный сервис по оказанию услуг перевозки и доставки. Диаграмма последовательности.
- 5.5** Электронно-информационный сервис по оказанию услуг перевозки и доставки. Диаграмма классов и модулей.
- 5.6** Электронно-информационный сервис по оказанию услуг перевозки и доставки. Модель данных.
- 5.7** Заключительный плакат. Плакат.

**6** Содержание задания по технико-экономическому обоснованию: «Технико-экономическое обоснование разработки электронно-информационного сервиса по оказанию услуг перевозки и доставки».

ЗАДАНИЕ ВЫДАЛА

О.А. Матяс

КАЛЕНДАРНЫЙ ПЛАН

Наименование этапов дипломного проекта	Объем этапа, %	Срок выполнения этапа	Примечания
Подбор и изучение литературы. Сравнение аналогов. Уточнение задания на ДП	10	23.03 – 30.03	
Структурное проектирование	15	31.03 – 09.04	
Функциональное проектирование	25	10.04 – 25.04	
Разработка программных модулей	20	26.04 – 07.05	
Программа и методика испытаний	10	08.05 – 15.05	
Расчет экономической эффективности	5	16.05 – 18.05	
Оформление пояснительной записи	15	19.05 – 30.05	

Дата выдачи задания: 23.03.22

Руководитель

А. М. Ковальчук

ЗАДАНИЕ ПРИНЯЛ К ИСПОЛНЕНИЮ

\_\_\_\_\_

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ .....	9
1 ОБЗОР ЛИТЕРАТУРЫ .....	11
1.1 Исследование предметной области.....	11
1.1.1 Клиент-серверная архитектура.....	11
1.1.2 Распространение на платформах.....	12
1.2 Обзор аналогов.....	13
1.2.1 Yandex.Go .....	13
1.2.2 Uber .....	14
1.2.3 Bolt .....	15
1.2.4 135 .....	17
1.2.5 Maxim.....	18
1.3 Обзор технологий и инструментов .....	19
1.3.1 Языки программирования.....	19
1.3.1.1 Java .....	19
1.3.1.2 Swift.....	19
1.3.1.3 TypeScript.....	19
1.3.1.4 Python .....	20
1.3.2 Клиентская часть .....	20
1.3.2.1 Android SDK .....	20
1.3.2.2 iOS SDK .....	20
1.3.2.3 React Native.....	21
1.3.2.4 Flutter.....	21
1.3.3 Серверная часть .....	21
1.3.3.1 Spring.....	21
1.3.3.2 Nest .....	22
1.3.3.3 Django.....	22
1.3.4 Система управления базами данных.....	22
1.3.4.1 PostgreSQL.....	22
1.3.4.2 MongoDB .....	23
1.4 Постановка задачи .....	23
2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ .....	24
2.1 Мобильное приложение .....	24
2.1.1 Блок пользовательского интерфейса .....	24
2.1.2 Блок мобильной бизнес-логики.....	24
2.1.3 Блок работы с хранилищем .....	25
2.1.4 Блок работы с сервером .....	25
2.1.5 Блок работы с GPS.....	25
2.1.6 Блок вспомогательных утилит .....	26
2.2 Серверное приложение .....	26
2.2.1 Блок получения запроса клиента .....	26
2.2.2 Блок авторизации.....	26

2.2.3 Блок обработки данных клиента .....	27
2.2.4 Блок работы с СУБД.....	27
2.2.5 Блок оплаты.....	28
2.2.6 Блок подключения клиента.....	28
2.2.7 Блок RTC .....	28
2.2.8 Блок расчета стоимости и маршрута поездки.....	29
2.2.9 Блок работы с сервисами Google .....	29
<b>3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ.....</b>	<b>30</b>
<b>3.1 Мобильное приложение .....</b>	<b>30</b>
<b>3.1.1 Модуль пользовательского интерфейса .....</b>	<b>30</b>
3.1.1.1 Компонент App .....	31
3.1.1.2 Компонент NavigationEntry .....	31
3.1.1.3 Компонент HorizontalPicker.....	31
3.1.1.4 Компонент TextInput .....	31
3.1.1.5 Компонент LoginScreen.....	32
3.1.1.6 Компонент RegisterScreen.....	32
3.1.1.7 Компонент HistoryScreen .....	33
3.1.1.8 Компонент HomeDrawer .....	34
3.1.1.9 Компонент PaymentMethodComponent.....	34
3.1.1.10 Компонент AddCard .....	35
3.1.1.12 Компонент ProfileScreen .....	35
3.1.1.13 Компонент Status .....	36
3.1.1.14 Компонент Pointer.....	36
3.1.1.15 Компонент HomeScreen .....	37
3.1.1.16 Компонент RideRequest.....	38
3.1.1.17 Компонент DriverOnRideStatus .....	38
3.1.1.18 Компонент SearchBlock.....	38
3.1.1.19 Компонент SearchResultsBlock .....	39
3.1.1.20 Компонент ChooseClass .....	39
3.1.1.21 Компонент CustomerRideStatus .....	39
3.1.2 Модуль мобильной бизнес-логики .....	40
3.1.2.1 Функция appSaga .....	41
3.1.2.2 Функция initializationSaga .....	41
3.1.2.3 Функция loginSaga .....	41
3.1.2.4 Функция listenForLogin .....	41
3.1.2.5 Функция logoutSaga .....	41
3.1.2.6 Функция listenForLogout .....	41
3.1.2.7 Функция registerSaga .....	42
3.1.2.8 Функция listenForRegister .....	42
3.1.2.9 Функция fetchHistorySaga .....	42
3.1.2.10 Функция listenForFetchHistory .....	42
3.1.2.11 Функция getUserSaga.....	42
3.1.2.12 Функция listenFor GetUser .....	43
3.1.2.13 Функция getPaymentMethodsSaga .....	43

3.1.2.14 Функция listenForGetPaymentMethods .....	43
3.1.2.15 Функция removePaymentMethodSaga.....	43
3.1.2.16 Функция listenForRemovePaymentMethod.....	43
3.1.2.17 Функция setAsDefaultPaymentMethodSaga.....	43
3.1.2.18 Функция listenForSetAsDefaultPaymentMethod.....	44
3.1.2.19 Функция addCardSaga .....	44
3.1.2.20 Функция listenForAddCard .....	44
3.1.2.21 Функция paySaga .....	44
3.1.2.22 Функция listenForPay.....	44
3.1.2.23 Функция initializeMapSaga.....	44
3.1.2.24 Функция listenForInitializeMap .....	45
3.1.2.25 Функция receiveLocationUpdateSaga.....	45
3.1.2.26 Функция bootstrapGPSSubscription .....	45
3.1.2.27 Функция receiveWebSocketMessageSaga .....	45
3.1.2.28 Функция bootstrapWebSocketSubscription .....	45
3.1.2.29 Функция chooseRouteSaga .....	45
3.1.2.30 Функция listenForChooseRoute .....	46
3.1.2.31 Функция fetchPlacesSaga.....	46
3.1.2.32 Функция listenForFetchPlaces .....	46
3.1.2.33 Функция prepareRideDataSaga.....	46
3.1.2.34 Функция listenForPrepareRide.....	46
3.1.2.35 Функция requestRideSaga .....	46
3.1.2.36 Функция listenForRequestRide .....	47
3.1.2.37 Функция answerToRideRequestSaga .....	47
3.1.2.38 Функция listenForRequestRide .....	47
3.1.2.39 Функция setChosenLocationSaga .....	47
3.1.2.40 Функция listenForSetChosenLocation .....	47
3.1.2.41 Функция setRouteLocationSaga.....	48
3.1.2.42 Функция listenForSetRouteLocation.....	48
<b>3.1.3 Модуль работы с хранилищем .....</b>	<b>48</b>
3.1.3.1 Тип ApplicationState.....	48
3.1.3.2 Тип State<T> .....	49
3.1.3.3 Тип UserState .....	49
3.1.3.4 Тип User .....	49
3.1.3.5 Перечисление Gender .....	49
3.1.3.6 Тип Driver .....	50
3.1.3.7 Перечисление CarClass.....	50
3.1.3.8 Тип HistoryState .....	50
3.1.3.9 Тип Ride .....	50
3.1.3.10 Перечисление RideStatus.....	51
3.1.3.11 Тип PaymentsState .....	51
3.1.3.12 Тип PaymentMethod .....	51
3.1.3.13 Перечисление PaymentMethodType .....	51
3.1.3.14 Тип PaymentMethodDetails.....	52

3.1.3.15 Перечисление CarBrand .....	52
3.1.3.16 Тип HomeState.....	52
3.1.3.17 Тип PointerLocation.....	52
3.1.3.19 Тип Location .....	53
3.1.3.20 Тип PrepareRide.....	53
3.1.3.21 Тип RideRequest.....	53
3.1.3.22 Тип PrepareDriverRide .....	54
3.1.3.23 Тип ExtendedRideRequest .....	54
3.1.3.24 Тип ChooseRouteState .....	54
3.1.3.25 Тип DirectionChooseResult .....	54
3.1.3.26 Тип RideState .....	55
3.1.3.27 Функция userReducer.....	55
3.1.3.28 Функция historyReducer .....	55
3.1.3.29 Функция paymentsReducer .....	55
3.1.3.30 Функция homeReducer.....	55
3.1.4 Модуль работы с сервером .....	56
3.1.4.1 Класс RestGatewayAPI .....	56
3.1.4.2 Класс ConnectionGatewayAPI .....	56
3.1.4.3 Класс AuthAPI.....	56
3.1.4.4 Класс HistoryAPI.....	57
3.1.4.5 Класс HomeAPI .....	57
3.1.4.6 Класс PaymentsAPI .....	58
3.1.4.7 Класс ProfileAPI.....	58
3.1.5 Модуль работы с GPS .....	59
3.1.6 Модуль вспомогательных утилит .....	59
3.1.6.1 Класс NavigationService .....	59
3.1.6.2 Класс MapService .....	59
3.2 Серверное приложение .....	60
3.2.1 Модуль получения запроса клиента .....	60
3.2.1.1 Класс AuthController.....	60
3.2.1.2 Класс MapsController .....	60
3.2.1.3 Класс PaymentController.....	61
3.2.1.4 Класс UserController .....	61
3.2.2 Модуль авторизации .....	62
3.2.3 Модуль обработки данных клиента .....	62
3.2.4 Модуль работы с СУБД .....	63
3.2.4.1 Класс AuthData.....	63
3.2.4.2 Класс Driver.....	63
3.2.4.3 Класс Payment .....	64
3.2.4.4 Класс PaymentMethod.....	64
3.2.4.5 Класс PaymentMethodDetails .....	64
3.2.4.6 Класс Ride.....	64
3.2.4.7 Класс User.....	65
3.2.4.8 Классы репозиториев .....	65

3.2.5 Модуль оплаты .....	66
3.2.5.1 Класс PaymentService .....	66
3.2.5.2 Класс Stripe.....	67
3.2.6 Модуль подключения клиента .....	67
3.2.7 Модуль RTC .....	68
3.2.8 Модуль расчета стоимости и маршрута поездки .....	69
3.2.9 Модуль работы с сервисами Google .....	69
3.2.9.1 Класс GoogleMaps.....	69
3.2.9.2 Класс Places .....	69
3.2.9.3 Класс Geocoding.....	70
3.2.9.4 Класс Directions.....	70
<b>4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ .....</b>	<b>71</b>
4.1 Добавление платежной карты .....	71
4.2 Составление маршрута поездки .....	73
4.3 Отслеживание местоположения водителя .....	77
4.3 Поиск водителя .....	79
<b>5 ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ.....</b>	<b>83</b>
<b>6 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ .....</b>	<b>89</b>
6.1 Аппаратные и программные требования.....	89
6.2 Руководство по использованию приложения .....	89
<b>7 ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ ЭФФЕКТИВНОСТИ РАЗРАБОТКИ И РЕАЛИЗАЦИИ ЭЛЕКТРОННО-ИНФОРМАЦИОННОГО СЕРВИСА ПО ОКАЗАНИЮ УСЛУГ ПЕРЕВОЗКИ И ДОСТАВКИ .....</b>	<b>98</b>
7.1 Описание функций, назначения и потенциальных пользователей программного обеспечения.....	98
7.2 Расчет затрат на разработку и цены электронно-информационного сервиса по оказанию услуг перевозки и доставки.....	98
7.3 Оценка экономического эффекта от продажи электронно-информационного сервиса по оказанию услуг перевозки и доставки .....	101
7.3 Расчет показателей эффективности разработки программного обеспечения.....	102
<b>ЗАКЛЮЧЕНИЕ .....</b>	<b>103</b>
<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....</b>	<b>104</b>
<b>ПРИЛОЖЕНИЕ А .....</b>	<b>105</b>
<b>ПРИЛОЖЕНИЕ Б .....</b>	<b>106</b>
<b>ПРИЛОЖЕНИЕ В .....</b>	<b>107</b>

## **ВВЕДЕНИЕ**

Современную жизнь невозможно представить без портативных устройств, помогающих нам изо дня в день. Мобильные приложения используются повсеместно – от вычислений и навигации до медицины и финансов. Ежедневно сотни миллионов людей используют приложения для общения, развлечений и ведения бизнеса, многие компании строят свой бизнес исключительно в сфере мобильных приложений.

В настоящее время рынок мобильных приложений развивается быстрыми темпами. Рост числа мобильных устройств, усиление их влияния на повседневную жизнь человека, а также простота использования мобильных приложений для самых различных задач обуславливают рост рынка мобильных приложений.

Различные компании и сервисы используют мобильные приложения для привлечения аудитории и автоматизации процессов. Так, например, банковские приложения позволяют клиенту обращаться в поддержку банка, просматривать баланс и проводить различные финансовые операции прямо с телефона. Приложения файлообменники предоставляют возможность передавать компьютерные файлы пользователям по сети. Фото- и видео редакторы упрощают процесс обработки цифровых изображений и видеопотока прямо со смартфона или планшета. Популярные приложения агрегаторы скидок привлекают аудиторию к партнерам компании, распространяющей данное приложение.

С ростом рынка мобильных приложений изменился и рынок услуг, поэтому особый интерес вызывает проблема конкуренции на сложившемся обновленном рынке такси. Сервисы предоставления услуг такси в Беларуси находятся в стадии активного роста: агрегаторы, предоставляющие услуги заказа такси через интернет, конкурируют между собой, а также с общественным транспортом. В то же время они соревнуются и с частным транспортом, заставляя пользователей отказываться от покупки личного автомобиля в пользу более выгодного и удобного средства передвижения.

Мобильные приложения таких компаний выводят опыт пользования услугами на качественно новый уровень, значительно упрощая процесс оплаты и заказа машины благодаря автоматизации множества процессов. Например, вместо того, чтобы звонить в колл-центр и заказывать такси у оператора по определенному адресу, пользователь может выбрать текущую позицию прямо на карте своего смартфона. С использованием банковских систем интернет-расчетов можно рассчитываться за поездку, не имея при себе наличных денег или самой пластиковой карты.

Целью дипломного проекта является разработка мобильного приложения для пользования услугами такси. В приложении, разрабатываемом в рамках дипломного проекта, будет реализована возможность выбора точки назначения, поездки по наиболее оптимальному маршруту, оплаты различными способами и отслеживания прогресса поездки.

Разрабатываемое приложение предназначено для жителей городов и поселков, которым требуются услуги перевозки различных вещей, а также для тех, которые предпочитают путешествовать и перемещаться приватно и комфортно.

# 1 ОБЗОР ЛИТЕРАТУРЫ

## 1.1 Исследование предметной области

### 1.1.1 Клиент-серверная архитектура системы

«Клиент-сервер» - архитектура, которая чаще всего ложится в основу проектирования современных многопользовательских систем. Ее использование позволяет разделить обязанности и ответственности между клиентами и сервисом. Проектирование разрабатываемой в дипломном проекте системы будет осуществляться в соответствии с клиент-серверной архитектурой.

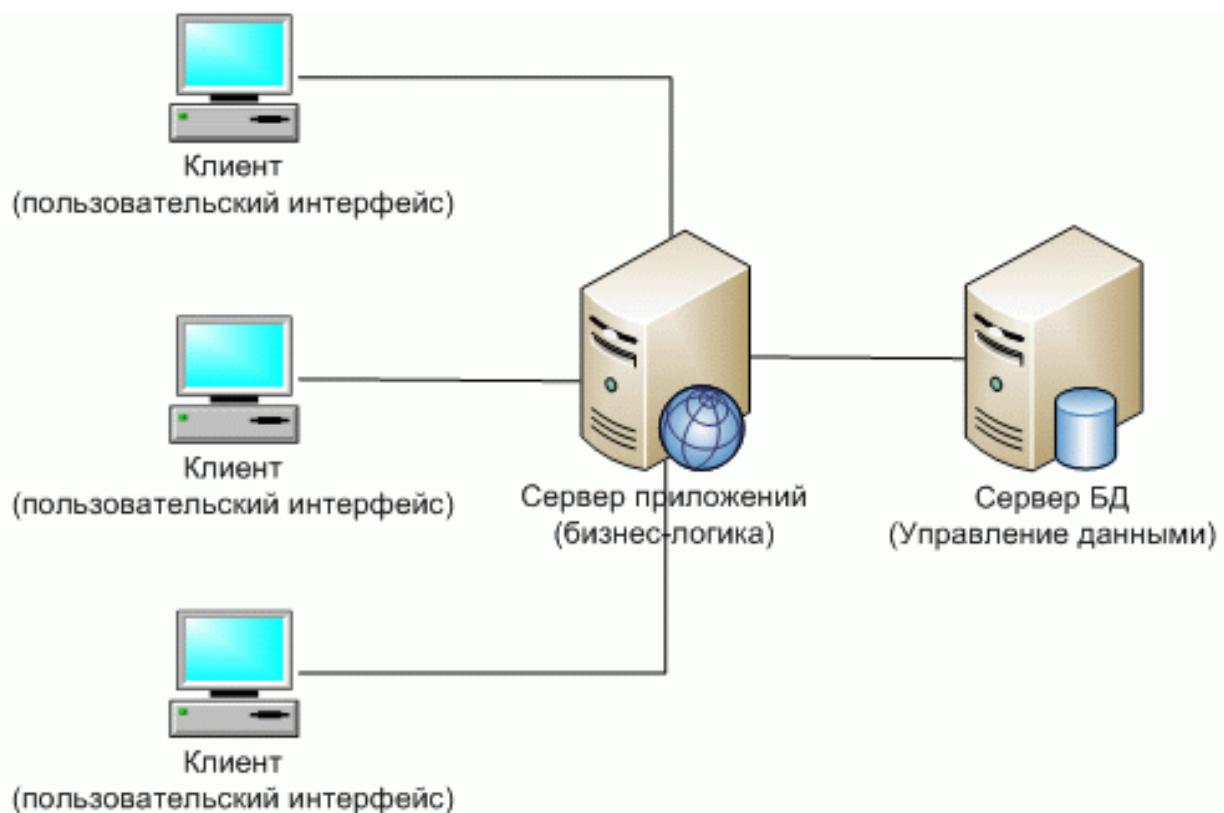


Рисунок 1.1 – Схема клиент-серверной архитектуры

Клиент-серверная архитектура заключает в себе идею использования одной или нескольких машин для создания одного продукта/системы, разделяя обязанности между частями этого продукта/системы. Разделение обязанностей достигается за счет деления системы на несколько программных модулей – клиентское ПО и серверное ПО. Обычно клиентское и серверное ПО взаимодействуют друг с другом посредством общения между устройствами системы с использованием различных сетевых протоколов, например HTTP, FTP, RPC и т.д., однако они также могут находиться и на

одном устройстве, используя для общения внутренние протоколы операционной системы, на которой они выполняются.

Клиент-серверная архитектура, в отличие от peer-to-peer, является централизованной, то есть конечные пользователи (клиенты), как и работоспособность всей системы, зависят от центральной точки – сервера, что является как преимуществом, так и недостатком.

Преимущества архитектуры:

- необходимость в наличии мощного устройства только со стороны сервера, клиентом же могут выступать современные устройства любой мощности, от персональных компьютеров до умных часов и смартфонов;
- необходимость обеспечивать защиту и хранить пользовательские данные только на стороне сервера ввиду того, что хранить их на устройстве конечного пользователя небезопасно;
- отсутствие дублирования кода между различными программными компонентами системы, что позволяет сделать клиенты легкими и быстрыми даже на самых слабых устройствах;

Недостатки архитектуры:

- отказ серверной части архитектуры вызывает неработоспособность всей системы;
- высокая стоимость серверного оборудования ввиду высоких требований к его производительности;
- поддержка серверного оборудования требует выделенных специалистов;

В настоящем проекте применение peer-to-peer архитектуры целесообразно, ввиду непостоянного количества клиентов в сети системы, а также их малой производительности.

### **1.1.2 Распространение на платформах**

На этапе исследования изучены приложения, представленные в магазинах App Store и Google Play. Аналоги разрабатываемого приложения распространяются по схеме In-App Purchases [1]. Существуют несколько типов таких приложений, самыми распространенными из них являются следующие:

- пользователь, скачивая приложение из магазина, получает доступ к его полной версии, и оплата идет непосредственно за услуги, предоставляемые сервисом. Покупки внутри приложений представляют из себя подписки или единоразовые платежи, дающие определенные преимущества в использовании, например скидки на поездки в такси определенного класса;
- пользователь, скачивая приложение из магазина, получает доступ к его ограниченной версии, покупка (подписка или единоразовый платеж) разблокируют недоступный ранее платный функционал.

Сервисы, представленные в магазинах предложений, являются представителями первого типа приложений.

## **1.2 Обзор аналогов**

### **1.2.1 Yandex.Go**

Yandex.Go [2] – это бесплатное приложение, ориентированное на рынок стран СНГ, для пользования услугами такси, перевозок, а так же доставки еды. Грамотно продуманный и реализованный графический интерфейс, позволяющий выполнять все самые популярные задачи одной рукой в несколько нажатий, обеспечивает удобство пользования. Интерфейс приложения представлен на рисунке 1.2.

В Yandex.Go реализована возможность оплаты поездки прямо из приложения. Пользователи могут привязывать карты различных банков и использовать одну из них для последующей оплаты. При списании средств не требуется одноразовый пароль (OTP), это делает оплату моментальной и исключает вмешательство пользователя в процесс.

Важным при использовании таких приложений является точность расчета времени прибытия и показа местоположения. В приложении информация о загруженности дорог и, соответственно, основанный на этих данных маршрут и расчетное время прибытия, вычисляются исходя из данных других сервисов компании Yandex. Необходимые данные вычисляются с использованием сервисов Yandex.Maps и Yandex.Navigator, которые анализируют данные со спутников, подключенных к сети устройств и различных государственных источников, с которыми сотрудничает компания Yandex.

Бесплатная версия приложения предоставляет пользователю функционал, необходимый для заказа такси и перевозки вещей. При покупке подписки пользователь получает доступ к различным бонусам в системе компании Yandex, например:

- бесплатный доступ к различным сервисам компании, таким как Kinopoisk, Yandex.Music и т.д.;

- скидки на поездки в такси определенного класса.

**Плюсы сервиса Yandex.Go:**

- возможность оплаты картой;
- высокая точность расчета времени прибытия;
- удобный пользовательский интерфейс.

**Минусы приложения:**

- разные версии приложений под каждую операционную систему, что удорожает процесс разработки;
- отсутствие возможности оплаты мобильными системами;
- функционирует только в некоторых странах СНГ.

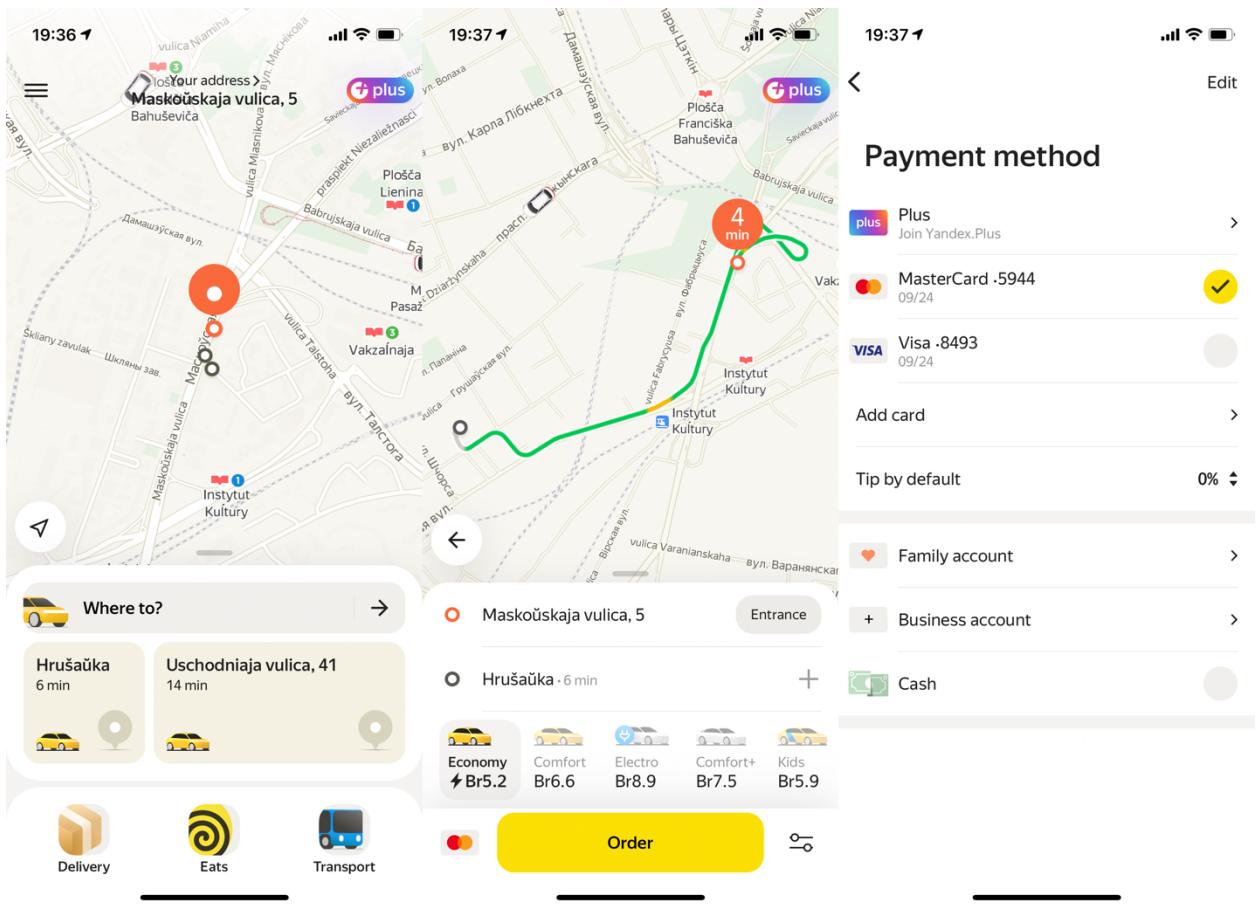


Рисунок 1.2 – Приложение Yandex.Go

## 1.2.2 Uber

Uber [3] – это мобильное приложение для пользования услугами такси международного уровня: сервис является крупнейшим игроком на многих мировых рынках, в том числе американский, европейский и рынок стран СНГ. Пользовательский интерфейс приложения представлен на рисунке 1.3. Бесплатная версия позволяет вызывать такси различных классов и заказывать доставку еды. Преимуществом приложения являются дополнительные системы оплаты.

В Uber, как и приведенном выше аналоге, пользователь может добавлять свои платежные карты в персональный аккаунт, однако помимо этого у пользователя появляется возможность оплаты платформенными сервисами, такими как Apple Pay, Google Pay и т.д., что позволяет пользователю избегать предоставления своих персональных данных третьим лицам.

Как и в большинстве аналогичных приложений, пользователь может получить доступ к дополнительному функционалу приложения с помощью подписки. Дополнительные функции, предоставляемые по подписке, связаны с получением различных скидок и бонусов. В бесплатной версии приложения вызов такси или доставки предполагает оплату по обычной цене с надбавкой в качестве платы за услуги сервиса. При покупке подписки у пользователя

появляется дополнительная скидка на такси и доставку, а также появляются дополнительные места, из которых можно заказывать еду.

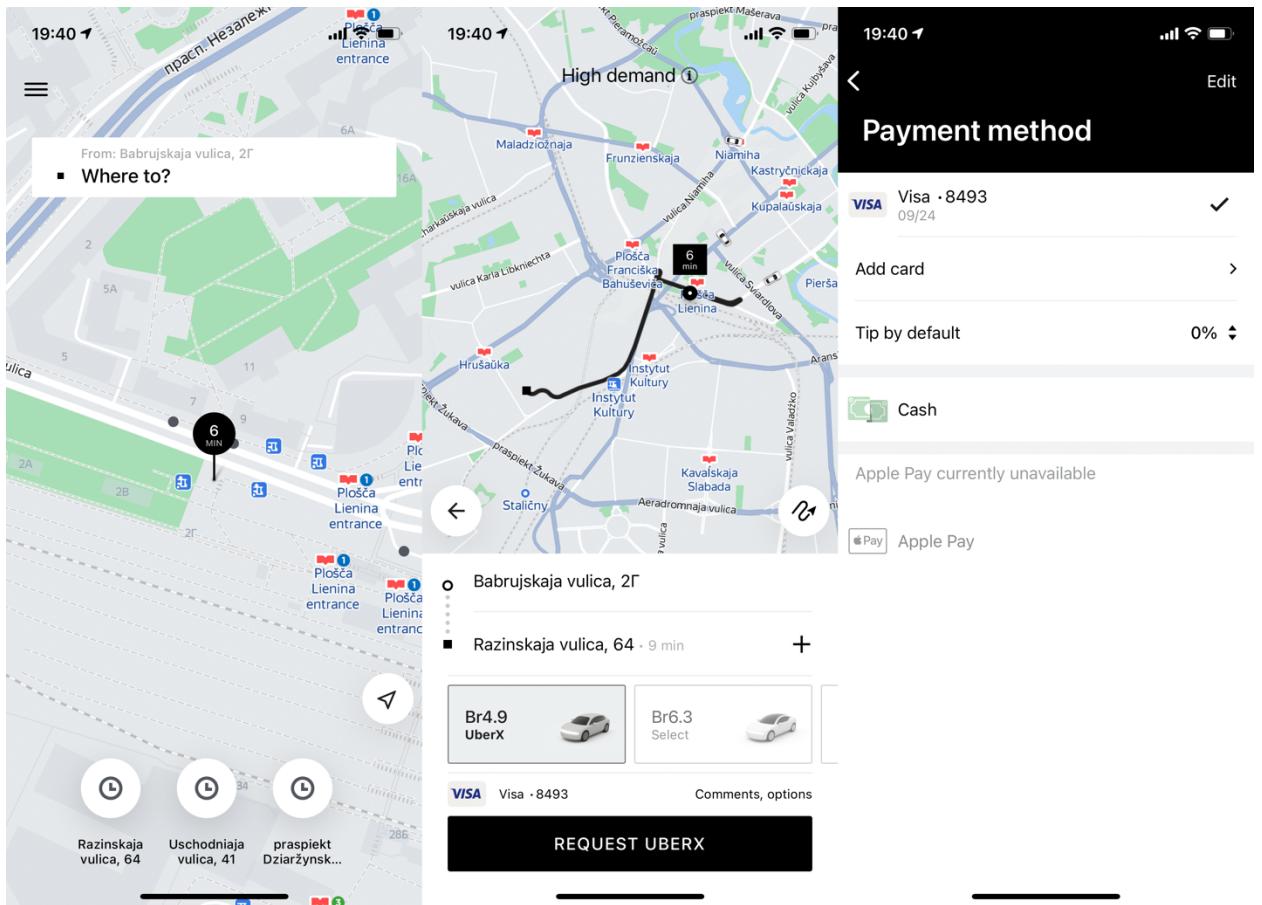


Рисунок 1.3 – Приложение Uber

#### Плюсы сервиса Uber:

- возможность оплаты картой и мобильными системами (Apple Pay, Google Pay);
- высокая точность расчета времени прибытия;
- работает в большинстве стран мира.

#### Минусы приложения:

- разные версии приложений под каждую операционную систему, что удорожает процесс разработки;
- недостатки в пользовательском интерфейсе (например расположение строки поиска сверху экрана, при использовании телефона одной рукой требуется перехватывать устройство).

### 1.2.3 Bolt

Bolt [4] дает возможность заказывать такси в различных точках мира. Пользовательский интерфейс приложения представлен на рисунке 1.4.

Особенностью сервиса является экологичность транспорта и высокая квалификация водителей, что делает его лучшим на рынке Европы. Сервис предоставляет удобный интерфейс и высокую отзывчивость как приложения, так и его серверов, что делает работу с ним плавной и быстрой. Преимуществами приложения Bolt является возможность оплаты через платформенные сервисы, а также расширенные возможности по авторизации.

Bolt предоставляет пользователям возможность заказа такси, однако, в отличие от конкурентов, оно позволяет использовать уже существующие аккаунты различных сервисов и социальных сетей для авторизации в своей системе, что избавляет пользователя от необходимости предоставления данных и заполнения форм регистрации.

Bolt является абсолютно бесплатным и не предоставляет подписок для своих пользователей, зарабатывая исключительно на перевозках и транспортировке.

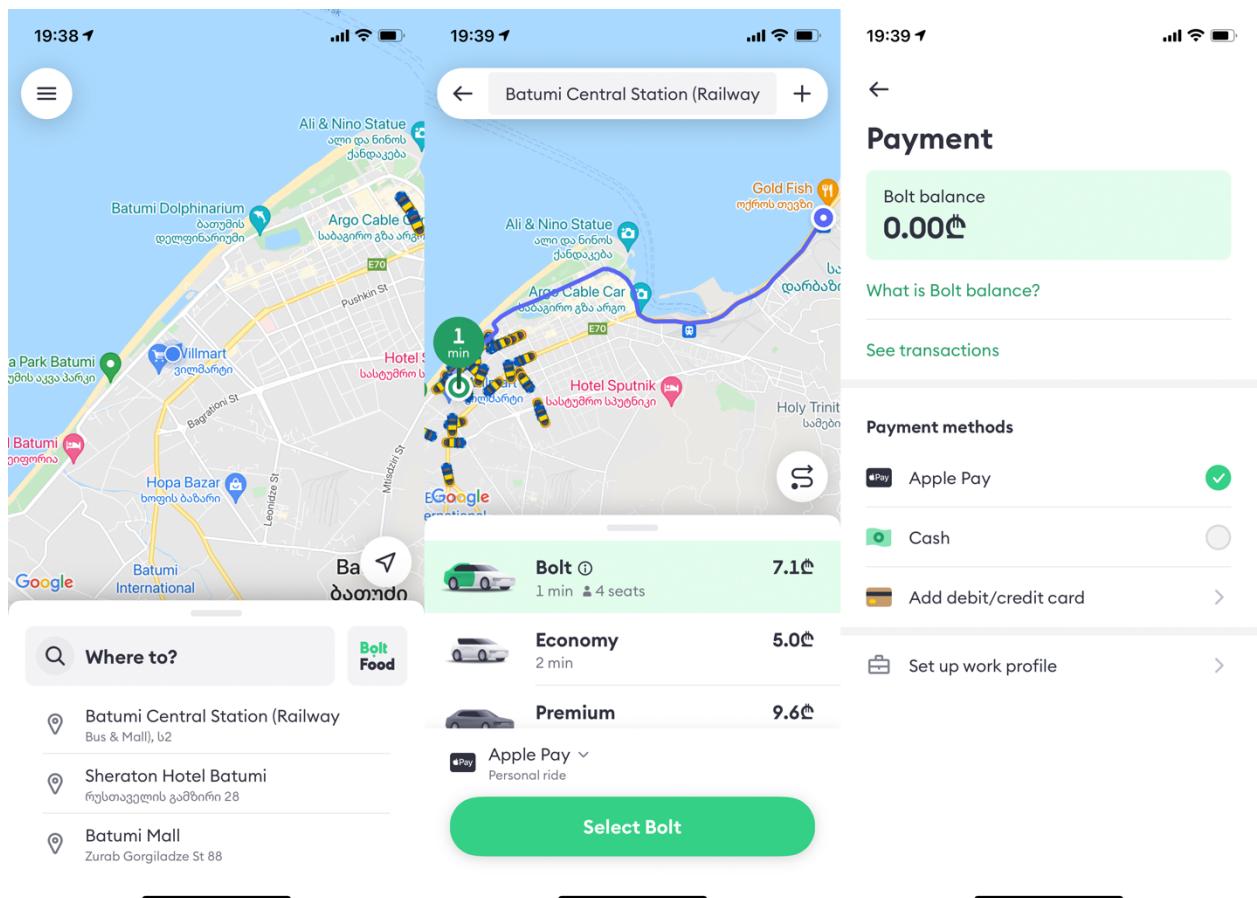


Рисунок 1.4 – Приложение Bolt

#### Плюсы сервиса Bolt:

- возможность оплаты картой и мобильными системами (Apple Pay, Google Pay);
- возможность авторизации с помощью сторонних сервисов;

- удобный пользовательский интерфейс и высокая скорость работы.

**Минусы приложения:**

- разные версии приложений под каждую операционную систему, что удорожает процесс разработки;
- списание средств происходит до подтверждения заказа водителем;
- функционирует в ограниченном количестве стран.

#### 1.2.4 135

135 [5] – сервис для пользования услугами такси и перевозками на территории Республики Беларусь. Он является самым крупным исключительно белорусским сервисом, а также одним из первых на рынке. Сервис предоставляет для пользователей не только мобильное приложение, но еще и сайт, что позволяет охватить наибольший круг пользователей. Внешний вид мобильного приложения представлен на рисунке 1.5.

Приложение 135, как и Bolt, является абсолютно бесплатным и не предоставляет моделей подписки.

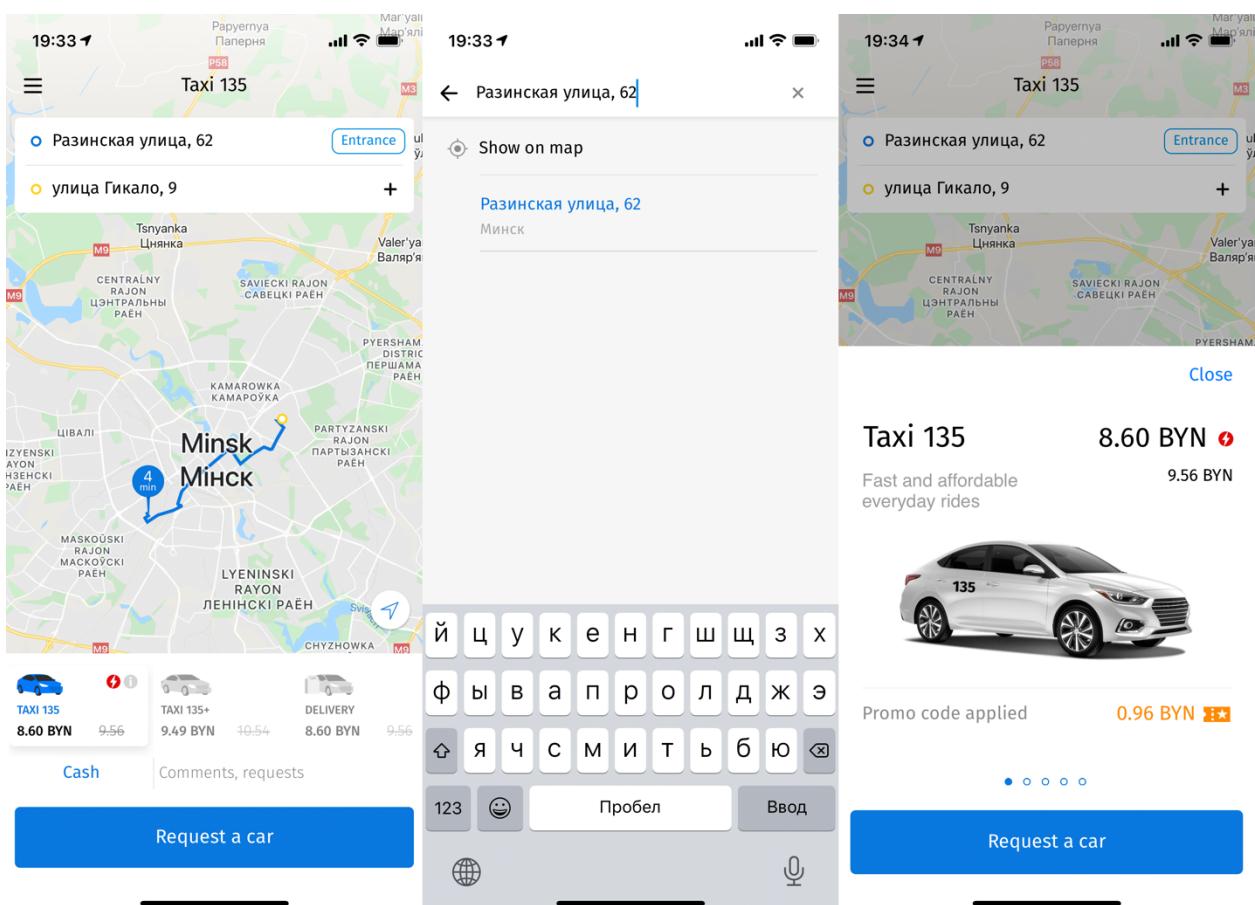


Рисунок 1.5 – Приложение 135

**Плюсы сервиса 135:**

- возможность оплаты картой;
- возможность вызова микроавтобусов при поездке большой компанией.

Минусы приложения:

- разные версии приложений под каждую операционную систему, что удорожает процесс разработки;
- малое количество машин;
- отсутствие возможности оплаты мобильными системами.

### 1.2.5 Maxim

Maxim [6] – сервис по вызову такси, работающий в более чем 1000 городах мира. Основанная в 2003 году, платформа начала набирать популярность в России, а позже начала распространяться в странах ближнего зарубежья. Пользовательский интерфейс представлен на рисунке 1.6. Несмотря на устаревший дизайн, приложение обладает достаточно широким функционалом, который отсутствует у аналогов.

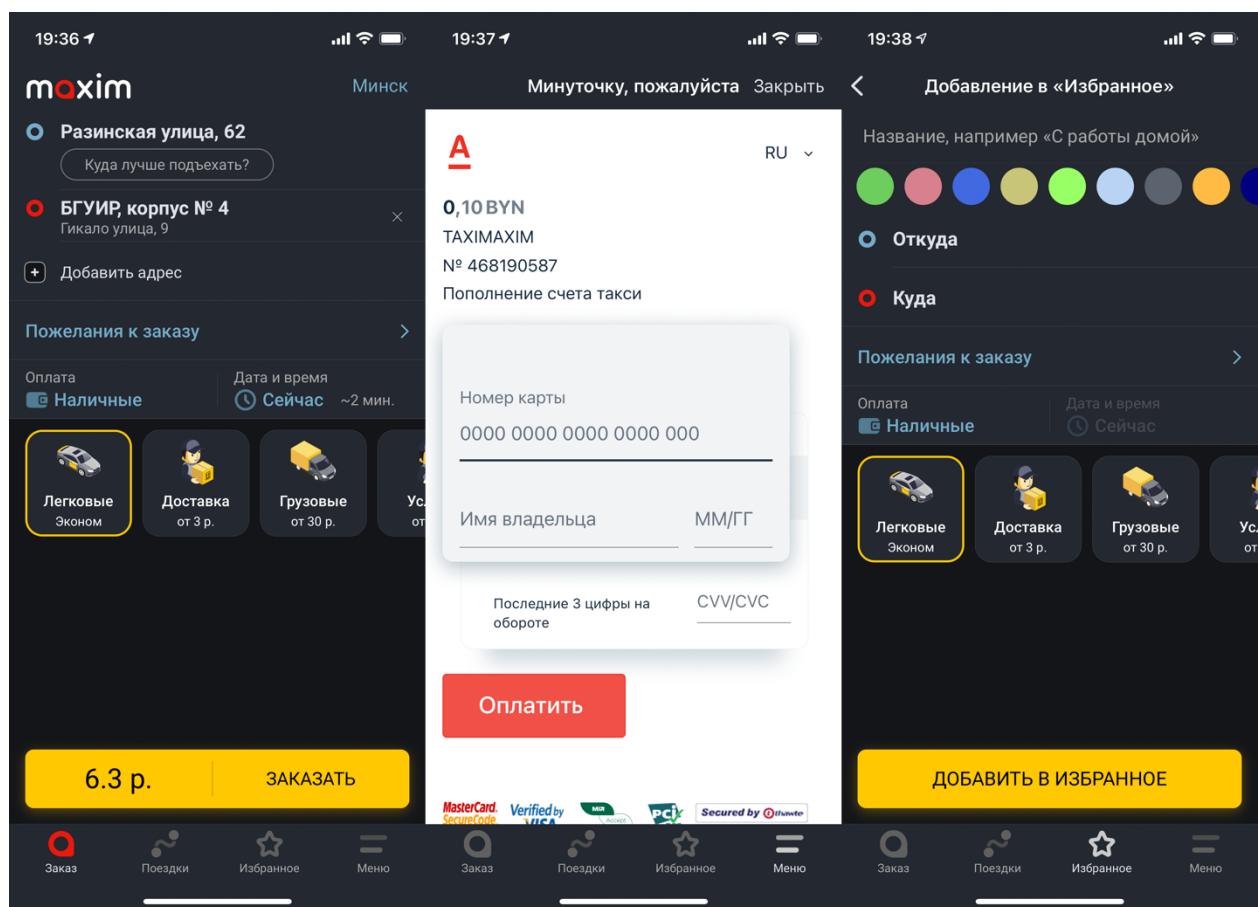


Рисунок 1.6 – Приложение Maxim

Приложение Maxim распространяется бесплатно и не предоставляет подписочную модель.

Плюсы сервиса Maxim:

- возможность оплаты картой;
- возможность добавление избранных маршрутов;
- возможность выбрать время подачи машины.

Минусы приложения:

- добавление карты происходит в веб-интерфейсе;
- устаревший дизайн;
- малое количество машин.

## 1.3 Обзор технологий и инструментов

### 1.3.1 Языки программирования

#### 1.3.1.1 Java

Java — строго типизированный объектно-ориентированный язык программирования общего назначения, разработанный компанией Sun Microsystems. Программы на Java транслируются в байт-код Java, выполняемый виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор.

#### 1.3.1.2 Swift

Swift — открытый мультипарадигмальный компилируемый язык программирования общего назначения. Создан компанией Apple в первую очередь для разработчиков iOS и macOS. Swift задумывался как более лёгкий для чтения и устойчивый к ошибкам программиста язык, нежели предшествовавший ему Objective-C. Программы на Swift компилируются при помощи LLVM. Swift может использовать рантайм Objective-C, что делает возможным использование обоих языков (а также C) в рамках одной программы.

#### 1.3.1.3 TypeScript

TypeScript — язык программирования, представленный Microsoft в 2012 году и позиционируемый как средство разработки веб-приложений, расширяющее возможности JavaScript. TypeScript отличается от JavaScript возможностью явного статического назначения типов, поддержкой использования полноценных классов (как в традиционных объектно-ориентированных языках), а также поддержкой подключения модулей, что

призвано повысить скорость разработки, облегчить читаемость, рефакторинг и повторное использование кода, помочь осуществлять поиск ошибок на этапе разработки и компиляции, и, возможно, ускорить выполнение программ.

### **1.3.1.4 Python**

Python — высокоуровневый язык программирования общего назначения с динамической строгой типизацией и автоматическим управлением памятью. Python является мультипарадигмальным языком программирования, поддерживающим императивное, процедурное, структурное, объектно-ориентированное программирование, метапрограммирование и функциональное программирование. Стандартная библиотека включает большой набор полезных переносимых функций, начиная с возможностей для работы с текстом и заканчивая средствами для написания сетевых приложений.

## **1.3.2 Клиентская часть**

Клиентская часть разрабатываемой системой будет представлена мобильным приложением. Для выбора оптимальной технологии требуется сделать обзор доступных вариантов.

### **1.3.2.1 Android SDK**

Android SDK [7] – официальный набор утилит от Google для разработки под операционную систему Android. Разработана с использованием языков Kotlin, Java и C++. Позволяет разрабатывать высокопроизводительные приложения для платформ Android, Android TV и WearOS. В состав SDK включены различные средства разработки, в том числе отладчик, набор библиотек, телефонный эмулятор на базе движка QEMU, набор документации, примеров приложений и руководств. Среда Android SDK может быть запущена на компьютерах, использующих ОС Linux, Mac OS X 10.5.8 и новее, Windows 7 и новее.

### **1.3.2.2 iOS SDK**

iOS SDK [8] – официальный набор утилит от Apple для разработки под операционную систему iOS. Разработана с использованием языков Objective-C, Swift и C++, позволяет разрабатывать оптимизированные приложения для модельного ряда устройств iPhone. Наряду с набором инструментов Xcode, SDK содержит iPhone Simulator, используемый для имитации внешнего вида iPhone на компьютере разработчика, ранее называвшийся «Aspen Simulator». Новые версии SDK сопровождают новые версии iOS. Чтобы тестировать приложения, получать техническую поддержку и распространять приложения

через App Store, разработчикам необходимо подписаться на программу Apple Developer Program.

### **1.3.2.3 React Native**

React Native [9] – фреймворк для создания кроссплатформенных приложений, разработанный компанией Facebook. Данный фреймворк позволяет разрабатывать приложение под несколько платформ (например, iOS и Android) с использованием лишь одного языка – TypeScript. Это преимущество компенсируется более низкой производительностью в сравнении с Android и iOS SDK. TypeScript-код, написанный разработчиком, выполняется в фоновом потоке, и взаимодействует с платформенными API через асинхронную систему обмена данными, называемую Bridge.

### **1.3.2.4 Flutter**

Flutter [10] - фреймворк для создания кроссплатформенных приложений, разработанный компанией Google. Как и React Native, данный фреймворк позволяет писать код на несколько платформ с использованием одного языка – Dart. Показатели производительности фреймворка также ниже, в сравнении с Android и iOS SDK. Из-за ограничений на динамическое выполнение кода в App Store, под iOS Flutter использует АОТ-компиляцию. Широко используется такая возможность платформы Dart, как «горячая перезагрузка», когда изменение исходного кода применяется сразу в работающем приложении без необходимости его перезапуска.

## **1.3.3 Серверная часть**

Серверное ПО является центром в клиент-серверной архитектуре: тут происходит обработка данных, платежей и прочая специфичная каждому приложению бизнес-логика. Рассмотрим самые популярные технологии для построения такого ПО.

### **1.3.3.1 Spring**

Spring Framework [11] — универсальный фреймворк с открытым исходным кодом для Java-платформы. Spring дает большую свободу и гибкость разработчикам, при этом предоставляя эффективные и мощные утилиты для работы с такими важными вещами, как безопасность и хранение данных. Этот фреймворк предлагает последовательную модель и делает её применимой к большинству типов приложений, которые уже созданы на основе платформы Java. Считается, что Spring реализует модель разработки, основанную на лучших стандартах индустрии, и делает её доступной во многих областях Java.

### **1.3.3.2 Nest**

Nest (NestJS) — это фреймворк для создания эффективных масштабируемых серверных приложений NodeJS. Он сочетает в себе элементы ООП (объектно-ориентированное программирование), ФП (функционального программирования) и ФРП (функционально-реактивного программирования). Nest не только обеспечивает дополнительный уровень абстракции над распространенными платформами NodeJS (Express/Fastify), но также предоставляет свой собственный API разработчику, что дает ему свободу в использовании сторонних модулей.

### **1.3.3.3 Django**

Django — это высокоуровневый фреймворк, который способствует быстрой разработке и чистому прагматичному дизайну. Является самым популярным средством разработки веб-серверов на языке Python с широким набором встроенных средств и утилит. Django используется в сайтах Instagram, Mozilla, The Washington Times, Pinterest, YouTube, Google и др. На базе Django разработан ряд готовых решений со свободной лицензией, среди которых интернет-магазины, системы управления содержимым, а также более узконаправленные проекты.

## **1.3.4 Система управления базами данных**

База данных является важной частью многопользовательских систем. При проектировании таких приложений выбор базы данных проводится путем строгого анализа и отбора по целому ряду пунктов, например скорость и отказоустойчивость. Оптимальными являются несколько СУБД, описанных ниже.

### **1.3.4.1 PostgreSQL**

PostgreSQL [13] — свободная объектно-реляционная система управления базами данных. Существует в реализациях для множества операционных систем, в том числе Windows, macOS и Linux. PostgreSQL базируется на языке SQL и поддерживает многие из возможностей стандарта SQL 2011. Сильными сторонами PostgreSQL считаются:

- высокопроизводительные и надёжные механизмы транзакций и репликации;
- наследование;
- возможность индексирования геометрических (в частности, географических) объектов и наличие базирующегося на ней расширения PostGIS.

### **1.3.4.2 MongoDB**

MongoDB — документно-ориентированная система управления базами данных, не требующая описания схемы таблиц. На данный момент является самой популярной NoSQL системой, в качестве схемы данных используется JSON. Система масштабируется горизонтально, используя технику сегментирования объектов баз данных — распределение их частей по различным узлам кластера. Администратор выбирает ключ сегментирования, который определяет, по какому критерию данные будут разнесены по узлам (в зависимости от значений хэша ключа сегментирования). Благодаря тому, что каждый узел кластера может принимать запросы, обеспечивается балансировка нагрузки.

## **1.4 Постановка задачи**

Разрабатываемая система должна выдавать максимальную производительность при минимальных временных и денежных затратах на разработку.

Клиентская часть должна быть отзывчивой и не вызывать дискомфорт у пользователя, а также должна предоставлять следующие функции:

- функция регистрации и входа в систему;
- возможность добавления и оплаты картой;
- возможность выбора места отправки и назначения на карте или в поле для ввода адреса;
- просмотр истории поездок и профиля пользователя;
- отслеживание местоположения водителя и машины;
- принятие заказа, если пользователь приложения – водитель такси.

Серверная часть должна максимально быстро обрабатывать входящие запросы клиентов и поддерживать с ними соединение для передачи текущего местоположения. Серверное ПО должно соответствовать следующим требованиям:

- высокие показатели по пропускной способности;
- эффективное расходование ресурсов;
- минимальное время, необходимое для разработки, внесения изменений и разворачивания системы.

## **2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ**

Изучив теоретические аспекты, связанные с проектированием и разработкой системы, и выработав список требований, можно разбить систему на два приложения: мобильное и серверное. Оба приложения представляют из себя отдельные программные продукты, которые можно разбить на функциональные блоки. Каждый функциональный блок отвечает за ту или иную функцию приложения. Структурная схема, иллюстрирующая перечисленные блоки и связи между ними, приведена на чертеже ГУИР.400201.107 С1.

### **2.1 Мобильное приложение**

#### **2.1.1 Блок пользовательского интерфейса**

Блок пользовательского интерфейса обеспечивает отображение данных на экране устройства и взаимодействие пользователя с приложением. В построении пользовательского интерфейса для приложения применяются UI компоненты фреймворка React Native, в составе которых различные кнопки, текст и прочие компоненты систем iOS и Android. React Native предоставляет возможность изменения стилей графических компонентов и обработки событий, возникающих при взаимодействии пользователя с элементами интерфейса.

Система расположения UI элементов React Native работает таким образом, что при грамотном построении дерева компонентов пользовательский интерфейс получается адаптивным, то есть изменяет свой размер и положение в зависимости от размера и ориентации экрана.

#### **2.1.2 Блок мобильной бизнес-логики**

Блок мобильной бизнес-логики является связующим звеном между блоком интерфейса и блоками работы с хранилищем, сервером и GPS. Его задача заключается в обработке событий, генерируемых пользовательским интерфейсом (нажатия на кнопки, движение карты и т.д.), генерировании запросов и обработке ответов сервера, отслеживанием данных с модуля GPS, а также в сохранении этих данных.

В мобильном приложении описанная функциональность реализована с помощью библиотеки Redux Saga, которая позволяет проводить синхронные и асинхронные операции вне жизненного цикла компонента, из которого было получено событие. В Redux Saga присутствуют различные вспомогательные функции для упорядочивания обработки входящих событий, их сортировки и т.д., что сводит написание специфической бизнес-логики к использованию нескольких функций из стандартного пакета.

### **2.1.3 Блок работы с хранилищем**

Блок работы с хранилищем инкапсулирует в себе всю логику хранения и восстановления данных. Пользовательский интерфейс подписывается на изменение данного хранилища, и при каждом его изменении компонент, зависящий от измененного значения хранилища, будет перерисован. Это гарантирует отображение на экране только актуальных данных.

Блок хранилища разделен на две части – хранилища в постоянной и оперативной памяти. За хранение данных в оперативной памяти отвечает библиотека Redux, легкая и быстрая, позволяющая модульно разделять хранилище на логические блоки. За хранение данных в постоянной памяти используется библиотека EncryptedStorage – хранилище типа ключ-значение, написанное на C++, она обеспечивает высокую скорость чтения и записи и не блокирует поток отрисовки графического интерфейса.

### **2.1.4 Блок работы с сервером**

Блок работы с сервером обеспечивает соединение между клиентом и сервером. Блок работает как с одиночными запросами на REST эндпоинты, так и с постоянным подключением через WebSocket. Блок ответствен за авторизацию клиента, так как к каждому запросу прикрепляется авторизационный ключ, который однозначно идентифицирует пользователя на сервере.

Блок реализован с помощью библиотек Axios и Socket.IO. Axios позволяет полностью контролировать содержимое REST запроса, например устанавливать заголовки и тело запроса, время ожидания ответа и так далее. Socket.IO позволяет поддерживать постоянное подключение с сервером с помощью одноименного протокола, эта библиотека написана с использованием языков Java (Android) и Objective-C (iOS). Благодаря ей сервер может отправлять водителям уведомление с предложением взять новый заказ, в то же время постоянно принимая поток данных об обновленном местоположении пользователей с их устройств.

### **2.1.5 Блок работы с GPS**

Блок работы с GPS предоставляет возможность получать данные о местоположении пользователя в реальном времени. Это позволяет серверу находить водителей, которые находятся ближе всего к клиенту. Подписка на обновление геолокации и получение широты и долготы происходит с помощью утилит, поставляемых вместе с фреймворком React Native (Geolocation).

## **2.1.6 Блок вспомогательных утилит**

Блок утилит представляет из себя набор различных классов и функций, в которых инкапсулирована работа со сторонними библиотеками, о которых не упоминалось ранее. Блок является дополнительным слоем абстракции, он позволяет уменьшить связанность кода и зависимость от определенных библиотек во всей кодовой базе. Например, если бы мы не использовали обертки над такими библиотеками, то при обновлении ее версии нам бы пришлось менять логику ее использования во всех местах программы. Такое решение является неоптимальным ввиду больших затрат по времени. В случае же использования утилиты-обертки мы можем сменить вызовы библиотечных функций только в этой утилите, и вся остальная программа продолжит корректное исполнение.

## **2.2 Серверное приложение**

### **2.2.1 Блок получения запроса клиента**

Блок получения запроса клиента является слоем абстракции между веб-сервером (например Apache, Nginx) и бизнес-логикой. В этом месте заключены настройки REST эндпоинтов, проверка доступов пользователя и проверка передаваемых параметров. В данном блоке работа с зависимостями серверного фреймворка должна быть сосредоточена так, чтобы блок бизнес-логики был максимально переносимым. Это значит, что блок бизнес-логики не должен ничего знать о том, кто к нему обращается и в каком фреймворке этот блок используется. Например, бизнес-логика по генерированию авторизационного ключа не должна ничего знать о том, что этот код выполняется в контексте запроса пользователя к серверу, написанному на Nest. После выполнения всех проверок выполнение передается блоку бизнес-логики, либо же, при возникновении ошибки в проверяемых данных, данный блок вернет клиенту ошибку без последующей передачи управления блоку с логикой.

Для построения блока используются различные аннотации и классы библиотеки Nest. Например, для создания класса, инкапсулирующего в себе различное количество REST эндпоинтов, используется аннотация `@Controller`, а для создания пути, обрабатывающего запрос типа GET, используется аннотация `@Get`.

### **2.2.2 Блок авторизации**

Блок авторизации берет на себя задачу обработки и генерации секретных данных клиента. В сферу его ответственности входит сравнение авторизационных данных (электронная почта и пароль), вычисление хэша

пароля и генерация авторизационных ключей, использующихся в заголовках запросов. Авторизационные ключи позволяют однозначно определить клиента, отправившего запрос на сервер.

Для генерации таких ключей (называемых также токенами) используется стандарт JWT. Токен JWT состоит из трех частей: заголовка (header), полезной нагрузки (payload) и подписи или данных шифрования. Первые два элемента — это JSON объекты определенной структуры. Третий элемент вычисляется на основании первых и зависит от выбранного алгоритма (в случае использования неподписанного JWT может быть опущен). Токены могут быть перекодированы в компактное представление (JWS/JWE Compact Serialization): к заголовку и полезной нагрузке применяется алгоритм кодирования Base64-URL, после чего добавляется подпись и все три элемента разделяются точками («..»).

### 2.2.3 Блок обработки данных клиента

Блок обработки данных клиента является местом обработки всех хранимых персональных данных пользователя. Обязанности блока сводятся к созданию, чтению, обновлению и удалению пользовательских записей из базы данных с помощью блока работы с СУБД.

Блок написан на языке TypeScript без использования сторонних библиотек.

### 2.2.4 Блок работы с СУБД

Блок работы с базой данных сосредотачивает в себе работу с конкретной реализацией СУБД, что позволяет абстрагировать бизнес-логику от той или иной СУБД, благодаря этому подходу в блоках с логикой мы можем использовать различные системы хранения данных, предоставляя использующему классу лишь унифицированный интерфейс взаимодействия.

Блок построен с использованием библиотеки TypeORM и драйвера для работы с PostgreSQL. TypeORM – библиотека для языка TypeScript, предназначенная для решения задач объектно-реляционного отображения (ORM). Объектно-реляционного отображение – технология программирования, суть которой заключается в создании «виртуальной объектной базы данных». Благодаря этой технологии разработчики могут использовать язык программирования, с которым им удобно работать с базой данных, вместо написания операторов SQL или хранимых процедур. Это может значительно ускорить разработку приложений. ORM также позволяет переключать приложение между различными реляционными базами данных. Например, приложение может быть переключено с MySQL на PostgreSQL с минимальными изменениями кода.

## **2.2.5 Блок оплаты**

Блок оплаты получает данные о типе платежа, его размере и платежном средстве, используя их для проведения транзакции в сети Stripe.

Stripe – американская технологическая компания, разрабатывающая решения для приёма и обработки электронных платежей. Предоставляет утилиты для интеграции с различными языками программирования, в том числе и TypeScript. В числе поддерживаемых способов оплаты находятся банковские карты (Visa, Mastercard, American Express), мобильные платежные средства (Apple Pay, Google Pay), а также различные виды расчета с отсроченным платежом (Klarna, AfterPay).

Мобильный клиент будет запрашивать у данного микросервиса создание сессии оплаты Stripe, а при получении данной сессии будет подтверждать оплату напрямую на серверах платежного шлюза Stripe.

## **2.2.6 Блок подключения клиента**

Блок подключения клиента представляет из себя шлюз для подключения клиентов мобильного приложения к серверу по протоколу WebSocket. Это необходимо для двухстороннего обмена сообщения с клиентами в режиме реального времени ввиду того, что у клиентов отсутствуют публичные IP адреса, и сервера не могут напрямую посыпать запрос на мобильный клиент. Передача данных через протокол WebSocket значительно упрощает и ускоряет работу сервера в сравнении с часто повторяющейся генерацией REST запросов на определенный маршрут (long-polling).

Такой тип подключения будет использоваться для сбора данных о местоположении машин и клиентов в реальном времени, а также оповещении водителя и клиента об изменении статуса заказа. Реализация WebSocket соединения написана с использованием библиотеки socket.io, являющейся самой популярной и эффективной для платформы NodeJS.

## **2.2.7 Блок RTC**

Блок RTC (Real Time Communication) будет отвечать за обмен сообщениями между пользователями и сбор данных о текущей геопозиции пользователей и трансформации географических координат в уникальные идентификаторы секторов, получаемые из библиотеки S2.

S2 – библиотека от Google, проецирующая географические координаты на геоид и вычисляющая ID сектора исходя из переданных координат. При вычислении идентификатора сектора указывается размер сектора. Размер сектора – длина стороны квадрата, которыми будет разбит геоид. Например, при размере сектора в 1 км, геоид будет разбит на множество квадратов, имеющих размер 1км в ширину и 1 км в длину. Соответственно люди,

находящиеся рядом и имеющие схожие географические координаты, будут иметь одинаковые либо близкие друг к другу идентификаторы секторов.

S2 упрощает работу по поиску ближайших к пользователю водителей. Поиск без этой библиотеки предполагает сравнение координат, учет ширины и долготы, учет размера сектора в градусах (а не в километрах). Все это происходило бы прямо во время поиска водителя, что значительно замедляет этот процесс. Однако, используя библиотеку S2, задача поиска ближайшего водителя сводится к поиску в массиве подключенных водителей объект, в котором модуль разницы между ID сектора водителя и ID сектора пользователя минимален или равен нулю.

### **2.2.8 Блок расчета стоимости и маршрута поездки**

Блок расчета стоимости и маршрута поездки, используя выбранные пользователем местоположения начала и конца поездки, просчитывает маршрут и время в пути используя блок работы с сервисами Google. Исходя из времени в пути и загруженности дорог блок вычисляет стоимость поездки и возвращает результат в виде объекта, содержащего путь на картах Google, стоимость поездки и время в пути.

Блок написан на языке TypeScript без использования сторонних библиотек.

### **2.2.9 Блок работы с сервисами Google**

Блок работы с сервисами Google представляет из себя набор классов и утилит, которые инкапсулируют в себе работу с Google Maps API. Данный блок позволяет:

- получать кратчайший маршрут используя Directions API;
- получать данные об адресе по географическим координатам используя Geocoding API;
- получать расстояние между точками используя Distance Matrix API.

Блок написан на языке TypeScript с использованием официальной библиотеки от Google – `@googlemaps/google-maps-services-js`. Библиотека полностью типизирована и предоставляет все необходимые обертки и параметры для получения необходимого результата.

### **3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ**

Разработка системы осуществляется с использованием объектно-ориентированного и функционального подходов. Для понимания структуры системы и ее функционирования необходимо описать свойства и методы классов и модулей, а также взаимоотношения между ними. Диаграмма классов и модулей приведена на чертеже ГУИР.400201.107 РР.1.

#### **3.1 Мобильное приложение**

##### **3.1.1 Модуль пользовательского интерфейса**

Модуль пользовательского интерфейса реализован с использованием функционального подхода разработки ввиду особенностей фреймворка. React Native использует декларативный способ описания пользовательского интерфейса, а отдельные элементы интерфейса (кнопки, текст, картинки и т.д.) программируются с помощью функций языка TypeScript. Результатом выполнения такой функции является дерево других UI элементов, из которых составлен рассматриваемый компонент, представленный вызываемой функцией. Для построения разметки фреймворк вызывает данные функции рекурсивно, создавая тем самым дерево элементов, которое затем будет отрисовано на экране смартфона.

У такого компонента заметен очевидный недостаток: если для перерисовки компонента нужно вызвать его функцию заново, то задача сохранения его состояния становится нетривиальной. Когда компонент описывается классом, то каждый отрисованный на экране компонент представляет из себя экземпляр этого класса, в котором можно хранить его состояние. Результатом же выполнения функции является разметка, и не существует конкретного объекта компонента, в котором хранится все его состояние.

Для решения этой проблемы фреймворк предоставляет специальные функции, называемые хуками. Принцип работы хуков заключается в следующем: при использовании хука внутри функции-компонента переменная, созданная с помощью хука, будет создана в глобальном контейнере, а функция-компонент будет использовать ссылку на созданную извне переменную. Данный подход отличается от обычных глобальных переменных тем, что при изменении переменной, созданной хуком, генерируется событие, которое уведомляет об этом фреймворк. Сам же фреймворк, получив это событие, запускает процесс перерисовки компонентов, которые используют обновившуюся переменную.

Все переменные и функции-обработчики, описанные для каждого компонента, созданы с использованием хуков. Для простоты описания каждая функция, описывающая компонент, будет называться компонентом.

### **3.1.1.1 Компонент App**

Компонент App используется как точка входа в приложение. Процесс его отрисовки инициирует рекурсивный вызов всех вложенных в него функций-компонентов.

Компонент App содержит следующие переменные:

- store : Store | null – контейнер, в котором находится состояние выполняющегося приложения.

### **3.1.1.2 Компонент NavigationEntry**

Компонент NavigationEntry представляет из себя функцию, содержащую в себе логику настройки навигации в приложении.

Компонент NavigationEntry содержит следующие переменные:

- isAuthorized : boolean – переменная, содержащая в себе статус авторизации. Используется для отрисовки разных частей приложения для авторизованных и неавторизованных пользователей.

### **3.1.1.3 Компонент HorizontalPicker**

Компонент HorizontalPicker предоставляет элемент, содержащий в себе горизонтально прокручиваемый список вложенных элементов, доступных для выбора.

Компонент HorizontalPicker содержит следующие переменные:

- selected : number | null – переменная, содержащая индекс выбранного элемента.

Компонент HorizontalPicker содержит следующие функции:

- handleChoose (number) – обработчик нажатия на выбранный элемент;
- setSelected (number | null) – функция установки значения переменной selected.

### **3.1.1.4 Компонент TextInput**

Компонент TextInput используется в качестве поля для ввода текста для всего приложения.

Компонент TextInput содержит следующие переменные:

- opacity : SharedValue – переменная, использующаяся для анимации прозрачности компонента;
- wrapperStyle : AnimatedStyle – объект, содержащий стиль компонента, внутри которого находится поле для ввода;

Компонент TextInput содержит следующие функции:

- `handleFocus()` – обработчик, вызываемый при фокусировке текстового поля (т.е. нажатия на него и появления клавиатуры);
- `handleBlur()` – обработчик, вызываемый при выходе текстового поля из фокуса (т.е. при скрытии клавиатуры и т.д.);
- `handlePress()` – обработчик, вызываемый нажатии на компонент, в котором находится поле для ввода.

### **3.1.1.5 Компонент LoginScreen**

Компонент `LoginScreen` является реализацией экрана входа. Данный компонент содержит в себе поля для ввода электронной почты и пароля, а также кнопки подтверждения ввода данных и регистрации.

Компонент `LoginScreen` содержит следующие переменные:

- `email` : `MutableRefObject<string>` – переменная, содержащая в себе значение поля для ввода электронной почты;
- `password` : `MutableRefObject<string>` – переменная, содержащая в себе значение поля для ввода пароля;
- `navigation` : `NavigationProp` – объект, хранящий в себе функции, необходимые для навигации между экранами;
- `isLoading` : `boolean` – флаг, показывающий, находится ли запрос на авторизацию в состоянии выполнения;

Компонент `LoginScreen` содержит следующие функции:

- `dispatch(Action)` – функция, которая отсылает события с пользовательского интерфейса в модули хранилища и бизнес-логики;
- `handleLoginPress()` – функция-обработчик нажатия на кнопку входа;
- `handleRegisterPress()` – функция-обработчик нажатия на кнопку регистрации.

### **3.1.1.6 Компонент RegisterScreen**

Компонент `RegisterScreen` является реализацией экрана регистрации. Экран содержит в себе многостраничную форму на несколько текстовых полей и списков выбираемых элементов для сбора необходимой информации о пользователе, а также кнопки далее и назад для навигации между страницами формы.

Компонент `RegisterScreen` содержит следующие переменные:

- `firstName` : `MutableRefObject<string>` – переменная, содержащая в себе значение поля для ввода имени;
- `lastName` : `MutableRefObject<string>` – переменная, содержащая в себе значение поля для ввода фамилии;

- email : MutableRefObject<string> – переменная, содержащая в себе значение поля для ввода электронной почты;
- password : MutableRefObject<string> – переменная, содержащая в себе значение поля для ввода пароля;
- phone : MutableRefObject<string> – переменная, содержащая в себе значение поля для ввода номера телефона;
- carModel : MutableRefObject<string> – переменная, содержащая в себе значение поля для ввода модели машины;
- gender : MutableRefObject<Gender | null> – переменная, хранящая выбранный пол пользователя;
- carClass : MutableRefObject<CarClass | null> – переменная, хранящая класс машины водителя: эконом, комфорт или бизнес;
- currentPage : MutableRefObject<number> – переменная, содержащая в себе текущую страницу заполняемой формы;
- scroll : MutableRefObject<ScrollView | null> – ссылка на прокручивающийся список формы, элемент списка – страница формы регистрации;
- isLoading : boolean – флаг, показывающий, находится ли запрос на регистрацию в состоянии выполнения;
- isDriver : boolean – флаг, описывающий выбранную пользователем роль: клиент или водитель.

Компонент RegisterScreen содержит следующие функции:

- dispatch(Action) – функция, которая отсылает события с пользовательского интерфейса в модули хранилища и бизнес-логики;
- setIsDriver() – функция установки значения поля isDriver;
- handleContinue() – функция-обработчик нажатия на кнопку далее;
- handleGoBack() – функция-обработчик нажатия на кнопку назад.

### 3.1.1.7 Компонент HistoryScreen

Компонент HistoryScreen является представлением экрана просмотра истории поездок. Экран состоит из списка карточек, каждая из которых отображает информацию об определенной поездке пользователя.

Компонент HistoryScreen содержит следующие переменные:

- isLoading : boolean – состояние запроса на получение списка поездок пользователей: в процессе загрузки или нет;
- data : Array<Ride> – список поездок пользователя.

Компонент HistoryScreen содержит следующие функции:

- dispatch(Action) – функция, которая отсылает события с пользовательского интерфейса в модули хранилища и бизнес-логики;

- `getHistory()` – функция, инициирующая запрос на получение данных о поездках.

### **3.1.1.8 Компонент HomeDrawer**

Компонент `HomeDrawer` представляет из себя боковое меню, которое представлено на домашнем экране. С его помощью можно перейти на другие экраны приложения, например способы оплаты, история поездок или профиль пользователя.

Компонент `HomeDrawer` содержит следующие переменные:

- `user : User` – текущий авторизованный пользователь;
- `defaultPaymentMethod : PaymentMethod` – способ оплаты по умолчанию.

Компонент `HistoryScreen` содержит следующие функции:

- `dispatch(Action)` – функция, которая отсылает события с пользовательского интерфейса в модули хранилища и бизнес-логики;
- `handleProfilePress()` – функция-обработчик нажатия на раздел профиля;
- `handlePaymentOptionsPress()` – функция-обработчик нажатия на раздел способов оплаты;
- `handleOrderHistoryPress()` – функция-обработчик нажатия на раздел истории поездок;
- `handleLogoutPress()` – функция-обработчик нажатия на кнопку выхода.

### **3.1.1.9 Компонент PaymentMethodComponent**

Компонент `PaymentMethodComponent` отображает платежный метод пользователя, например наличные или карты.

Компонент `PaymentMethodComponent` содержит следующие функции:

- `dispatch(Action)` – функция, которая отсылает события с пользовательского интерфейса в модули хранилища и бизнес-логики;
- `handleTilePress()` – функция-обработчик нажатия на способ оплаты, при вызове устанавливает выбранный способ оплаты основным;
- `handleDeletePress()` – функция-обработчик нажатия на кнопку удаления способа оплаты.

### **3.1.1.10 Компонент AddCard**

Компонент `AddCard` представляет экран добавления банковской карточки на аккаунт пользователя.

Компонент AddCard содержит следующие переменные:

- isLoading : boolean – состояние запроса на добавление карты в базу данных: в процессе загрузки или нет;
- user : User – текущий авторизованный пользователь;
- cardData :

MutableRefObject<CardFieldInput.Details> – ссылка на объект, содержащий информацию о введенной карте.

Компонент AddCard содержит следующие функции:

- dispatch(Action) – функция, которая отсылает события с пользовательского интерфейса в модули хранилища и бизнес-логики;

Компонент AddCard содержит следующие функции:

- handleAddPress() – функция-обработчик нажатия кнопки добавления карты.

### **3.1.1.11 Компонент PaymentsList**

Компонент PaymentsList представляет экран, содержащий список всех доступных пользователю методов оплаты.

Компонент HistoryScreen содержит следующие переменные:

- isLoading : boolean – состояние запроса на получение списка способов оплаты: в процессе загрузки или нет;
- methods : Array<PaymentMethod> – список методов оплаты пользователя.

Компонент HistoryScreen содержит следующие функции:

- dispatch(Action) – функция, которая отсылает события с пользовательского интерфейса в модули хранилища и бизнес-логики;
- getPaymentsList() – функция, инициирующая запрос на получение данных о способах оплаты;
- onAddCardPress() – функция-обработчик нажатия на кнопку добавления карты.

### **3.1.1.12 Компонент ProfileScreen**

Компонент ProfileScreen описывает экран, содержащий доступную информацию о пользователе: имя, фамилия, адрес электронной почты, пол, номер мобильного телефона и информация о машине и балансе (если пользователь зарегистрирован как водитель).

Компонент ProfileScreen содержит следующие переменные:

- isLoading : boolean – состояние запроса на получение информации о пользователе: в процессе загрузки или нет;
- user : User – текущий авторизованный пользователь.

Компонент `ProfileScreen` содержит следующие функции:

- `dispatch(Action)` – функция, которая отсылает события с пользовательского интерфейса в модули хранилища и бизнес-логики;
- `getUser()` – функция, инициирующая запрос на получение данных о пользователе.

### 3.1.1.13 Компонент `Status`

Компонент `Status` отображает данные о текущем местоположении пользователя и статусе получения этого местоположения. При выборе отправной точки данный компонент покажет текстовое описание места, координаты которого были выбраны. В процессе получения этих данных будет отображено сообщение о состоянии загрузки.

Компонент `Status` содержит следующие переменные:

- `isLoading` : `boolean` – состояние запроса на получение информации о выбранном местоположении;
- `isMoving` : `boolean` – флаг, описывающий состояние карты: производится ли перемещение указателя или нет;
- `isMoving` : `boolean` – является ли пользователь водителем;
- `isChoosingRoute` : `boolean` – находится ли клиент в состоянии выбора маршрута;
- `data` : `Optional<ExtendedLocation>` – текущее местоположение указателя с информацией о выбранном месте на русском языке.

### 3.1.1.14 Компонент `Pointer`

Компонент `Pointer` используется для отображения и выбора точки начала маршрута на карте. При перемещении карты компонент анимируется.

Компонент `Pointer` содержит следующие переменные:

- `sharedValue` : `SharedValue` – переменная для анимации компонента;
- `legStyle` : `Animated.Style` – стиль компонента с возможностью анимации.

Компонент `Pointer` содержит следующие функции:

- `start()` – функция, запускающая анимацию;
- `stop()` – функция, останавливающая анимацию.

### 3.1.1.15 Компонент `HomeScreen`

Компонент `HomeScreen` является ключевым в модуле пользовательского интерфейса и используется для отображения карты и

нижнего меню, содержащего возможность вызова такси (для клиента) и получения информации о заказе (для клиента и водителя). Двигая карту в режиме выбора маршрута, клиент выбирает точку отправления, информация о выбранной точке располагается вверху экрана в компоненте Status. В нижнем меню можно выбрать пункт назначения, класс машины и посмотреть состояние поездки, если она началась.

Компонент Pointer содержит следующие переменные:

- insets : EdgeInsets – объект, содержащий отступы от краев экрана, которые требуется сделать для правильного отображения контента. Например, на телефонах iPhone сверху экрана имеется вырез, и чтобы информация сверху экрана не была скрыта из-за этого выреза, сверху нужно добавить отступ, размером с высоту выреза;
- rideRequest : Optional<RideRequest> – запрос на поездку, отсылаемый водителю сервером по протоколу WebSocket;
- animatedSheetPosition : SharedValue – переменная, хранящая позицию нижнего меню в пикселях от верхней части экрана;
- isChoosingRoute : boolean – находится ли клиент в состоянии выбора маршрута;
- isPreparing : boolean – находится ли клиент в состоянии выбора класса машины;
- isDriver : boolean – является ли пользователь клиентом или водителем;
- isCarSearching : boolean – находится ли клиент в состоянии поиска автомобиля;
- isDriverIdle : boolean – является ли водитель свободным (т.е. нет выполняемого заказа);
- isDriverRequested : boolean – является ли водитель кандидатом на поездку (т.е. поступил ли запрос на выполнение заказа);
- isUserOnRide : boolean – находится ли пользователь в пути (т.е. есть ли текущий заказ);
- to : ExtendedLocation – переменная, хранящая в себе информацию о координате пункта назначения поездки;
- from : ExtendedLocation – переменная, хранящая в себе информацию о координате пункта начала поездки;
- isDraggable : boolean – флаг, разрешающий менять местоположение камеры карты.

Компонент HomeScreen содержит следующие функции:

- onBottomSheetAnimate() – функция, вызывающаяся при открытии или закрытии нижнего меню.

### **3.1.1.16 Компонент RideRequest**

Компонент RideRequest используется для отображения запроса о поездке на экране водителя. Компонент показывает информацию о маршруте, стоимости и длительности поездки и отрисовывается в нижнем меню.

Компонент RideRequest содержит следующие переменные:

- rideRequest : Optional<RideRequest> – запрос на поездку, отсылаемый водителю сервером по протоколу WebSocket;

Компонент RideRequest содержит следующие функции:

- dispatch(Action) – функция, которая отсылает события с пользовательского интерфейса в модули хранилища и бизнес-логики;
- answer(WSMessageType) – функция, отправляющая выбор пользователя в модуль бизнес-логики.

### **3.1.1.17 Компонент DriverOnRideStatus**

Компонент DriverOnRideStatus используется для отображения информации о поездке экране водителя. Компонент показывает информацию о маршруте, стоимости и длительности поездки и отрисовывается в нижнем меню.

Компонент DriverOnRideStatus содержит следующие переменные:

- rideStatus : RideStatus – текущий статус поездки;
- buttonTitle : string – надпись на кнопке, меняется в зависимости от статуса поездки;
- client : User – текущий клиент;

Компонент DriverOnRideStatus содержит следующие функции:

- dispatch(Action) – функция, которая отсылает события с пользовательского интерфейса в модули хранилища и бизнес-логики;
- handleButtonPress() – функция-обработчик нажатия на кнопку завершения или начала поездки.

### **3.1.1.18 Компонент SearchBlock**

Компонент SearchBlock содержит в себе поле для ввода точки отправления, поле для ввода точки назначения и списка найденных подходящих локаций и отрисовывается в нижнем меню.

Компонент SearchBlock содержит следующие переменные:

- to : Optional<ExtendedLocation> – переменная, хранящая в себе информацию о координате пункта назначения поездки;
- from : Optional<ExtendedLocation> – переменная, хранящая в себе информацию о координате пункта начала поездки;

- `formWrapperStyle` : `Animated.Style` – стиль формы для ввода локаций с возможностью анимации;
- `pointerPosition` : `Optional<ExtendedLocation>` – переменная, хранящая в себе информацию о положении указателя на карте.

Компонент `SearchBlock` содержит следующие функции:

- `dispatch(Action)` – функция, которая отсылает события с пользовательского интерфейса в модули хранилища и бизнес-логики;
- `onChangeText(string, string)` – функция, посылающая событие об изменении значения поля ввода локации в модуль бизнес-логики.

### **3.1.1.19 Компонент SearchResultsBlock**

Компонент `SearchResultBlock` представляет из себя список доступных для выбора локаций, которые были найдены исходя из значений в полях поиска компонента `SearchBlock`. Нажатие на элемент списка приводит к выбору этой позиции как точку начала или конца маршрута.

Компонент `SearchResultBlock` содержит следующие переменные:

- `toResults` : `Array<ExtendedLocation>` – переменная, хранящая в себе результаты поиска отправной точки маршрута;
- `fromResults` : `Array<ExtendedLocation>` – переменная, хранящая в себе результаты поиска конечной точки маршрута;
- `latest` : `string` – последнее выбранное текстовое поле: откуда или куда;
- `toRender` : `Array<ExtendedLocation>` – массив адресов, которые получены из переменных `toResults` и `fromResults` исходя из значения `latest`, и отсортированные по дальности удаления от пользователя.

Компонент `SearchResultBlock` содержит следующие функции:

- `dispatch(Action)` – функция, которая отсылает события с пользовательского интерфейса в модули хранилища и бизнес-логики;
- `onResultPress(ExtendedLocation)` – функция-обработчик нажатия на элемент списка, посылающая событие о выборе в модуль бизнес-логики.

### **3.1.1.20 Компонент ChooseClass**

Компонент `ChooseClass` отрисовывается в нижнем меню и содержит в себе прокручивающийся список доступных к заказу классов машин, а также кнопки вызова такси и отмены.

Компонент `ChooseClass` содержит следующие переменные:

- `rideRequest` : `Optional<RideRequest>` – информация о стоимости, времени и маршруте поездки;
- `isRideRequestLoading` : `boolean` – флаг, показывающий состояние загрузки информации о поездке;
- `isCarSearching` : `boolean` – флаг, показывающий, находится ли клиент в состоянии поиска машины;
- `selectedClass` : `number | null` – индекс выбранного класса машины в массиве доступных классов.

Компонент `ChooseClass` содержит следующие функции:

- `setSelectedClass(number | null)` – функция установки значения переменной `selectedClass`;
- `dispatch(Action)` – функция, которая отсылает события с пользовательского интерфейса в модули хранилища и бизнес-логики;
- `handleCancel()` – функция-обработчик нажатия на кнопку отмены, при нажатии отсылает событие об отмене поиска в модуль бизнес-логики;
- `handleGoPress()` – функция-обработчик нажатия на кнопку поиска машины, при нажатии отсылает событие о начале поиска в модуль бизнес-логики.

### **3.1.1.21 Компонент CustomerRideStatus**

Компонент `CustomerRideStatus` отображается в нижнем меню и содержит состояние текущей поездки. Способен отображать длительность поездки, позиции водителя (находится ли он в пути к точке отправления или точке назначения), а также информацию о самом водителе.

Компонент `CustomerRideStatus` содержит следующие переменные:

- `rideStatus` : `RideStatus` – текущий статус поездки;
- `driver` : `User` – информация о водителе;
- `title` : `string` – информация о позиции водителя.

## **3.1.2 Модуль мобильной бизнес-логики**

Модуль мобильной бизнес-логики реализован с использованием функционального подхода разработки ввиду используемой библиотеки Redux Saga. Функции, в которых заключена бизнес-логика, не требуют сохранения собственного состояния, поэтому являются хорошей альтернативой классов, содержащих бизнес-логику. Данные функции могут вызывать друг друга, однако из модуля пользовательского интерфейса их можно вызвать только отсылая определенные события с использованием функции `dispatch`. Функции оперируют объектами, полученными из модуля хранилища, сетевого

модуля, модуля работы с GPS и данными, переданными через события из пользовательского интерфейса.

### **3.1.2.1 Функция appSaga**

Функция appSaga является функцией инициализации работы всех слушателей, которые получают события из пользовательского интерфейса, и, в зависимости от переданного события, запускают определенную функцию из модуля бизнес-логики.

### **3.1.2.2 Функция initializationSaga**

Функция initializationSaga является функцией инициализации работы приложения, внутри нее происходят следующие операции:

- запрос разрешения на использование GPS;
- авторизация пользователя по сохраненным в зашифрованном хранилище токенам;
- конфигурация мобильного клиента платежной системы.

### **3.1.2.3 Функция loginSaga**

Функция loginSaga описывает алгоритм входа в систему со стороны клиента. На вход функции передаются электронная почта и пароль, затем запрос на авторизацию посыпается на сервер.

### **3.1.2.4 Функция listenForLogin**

Функция listenForLogin создает подписку функции loginSaga на событие LOGIN.TRIGGER. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция loginSaga.

### **3.1.2.5 Функция logoutSaga**

Функция loginSaga производит действия, необходимые для выхода из аккаунта: отключение от сервера, очистка токенов из заголовков клиентов модуля работы с сервером, удаление токенов из зашифрованного хранилища и переход на экран входа.

### **3.1.2.6 Функция listenForLogout**

Функция `listenForLogout` создает подписку функции `logotSaga` на событие `LOGOUT.TRIGGER`. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция `logoutSaga`.

### **3.1.2.7 Функция registerSaga**

Функция `registerSaga` работает аналогично функции `loginSaga`, однако на вход ей передаются данные, которые пользователь ввел в компоненте `RegisterScreen`. Эти данные отправляются на сервер для регистрации нового пользователя, дальнейший алгоритм работы функций `loginSaga` и `registerSaga` одинаковый.

### **3.1.2.8 Функция listenForRegister**

Функция `listenForRegister` создает подписку функции `registerSaga` на событие `REGISTER.TRIGGER`. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция `registerSaga`.

### **3.1.2.9 Функция fetchHistorySaga**

Функция `fetchHistorySaga` выполняет запрос на получение списка поездок пользователя, используя модуль работы с сервером. При успешном выполнении данные будут помещены в хранилище состояния приложения, при ошибке будет показано уведомление с деталями.

### **3.1.2.10 Функция listenForFetchHistory**

Функция `listenForFetchHistory` создает подписку функции `fetchHistorySaga` на событие `FETCH_HISTORY.TRIGGER`. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция `fetchHistorySaga`.

### **3.1.2.11 Функция getUserSaga**

Функция `getUserSaga` выполняет запрос на получение данных о пользователе, используя модуль работы с сервером. При этом сервер получает эти данные исходя из авторизационного токена, отправляемого в заголовке с мобильного устройства. При успешном выполнении данные будут помещены в хранилище состояния приложения, при ошибке будет показано уведомление с деталями.

### **3.1.2.12 Функция listenFor GetUser**

Функция `listenFor GetUser` создает подписку функции `getUserSaga` на событие `GET_USER_TRIGGER`. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция `getUserSaga`.

### **3.1.2.13 Функция getPaymentMethodsSaga**

Функция `getPaymentMethodsSaga` работает аналогично функциям `fetchHistorySaga` и `getUserSaga`, с тем лишь исключением, что с сервера запрашиваются данные доступных методах оплаты пользователя.

### **3.1.2.14 Функция listenForGetPaymentMethods**

Функция `listenForGetPaymentMethods` создает подписку функции `getPaymentMethodsSaga` на событие `GET_PAYMENT_METHODS_TRIGGER`. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция `getPaymentMethodsSaga`.

### **3.1.2.15 Функция removePaymentMethodSaga**

Функция `removePaymentMethodSaga` используется для удаления способа оплаты и получает на вход уникальный идентификатор способа оплаты, который отправляется на сервер в качестве тела запроса на удаление.

### **3.1.2.16 Функция listenForRemovePaymentMethod**

Функция `listenForRemovePaymentMethod` создает подписку функции `removePaymentMethodSaga` на событие `REMOVE_PAYMENT_METHOD_TRIGGER`. При отправке этого события с б модуля пользовательского интерфейса будет вызвана функция `removePaymentMethodSaga`.

### **3.1.2.17 Функция setAsDefaultPaymentMethodSaga**

Функция `setAsDefaultPaymentMethodSaga` устанавливает метод оплаты методом по умолчанию и работает аналогично функции `removePaymentMethodSaga`, с теми лишь исключениями, что наличный

способ оплаты не отсеивается, и запрос на установку метода по умолчанию выполняется на отдельный маршрут на сервере (эндпоинт).

### **3.1.2.18 Функция listenForSetAsDefaultPaymentMethod**

Функция `listenForSetAsDefaultPaymentMethod` создает подписку функции `setAsDefaultPaymentMethodSaga` на событие `SET_AS_DEFAULT_PAYMENT_METHOD.TRIGGER`. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция `setAsDefaultPaymentMethodSaga`.

### **3.1.2.19 Функция addCardSaga**

Функция `addCardSaga` принимает на вход платежную систему карты (Visa, MasterCard, American Express), четыре последние цифры номера карты, ее срок действия и имя владельца. Данная функция тесно связана с алгоритмом работы сервера по добавлению карты.

### **3.1.2.20 Функция listenForAddCard**

Функция `listenForAddCard` создает подписку функции `addCardSaga` на событие `ADD_CARD.TRIGGER`. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция `addCardSaga`.

### **3.1.2.21 Функция paySaga**

Функция `paySaga` используется для автоматической оплаты завершенной поездки и принимает на вход уникальный идентификатор поездки, которую требуется оплатить.

### **3.1.2.22 Функция listenForPay**

Функция `listenForPay` создает подписку функции `paySaga` на событие `PAY.TRIGGER`. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция `paySaga`.

### **3.1.2.23 Функция initializeMapSaga**

Функция `initializeMapSaga` производит первоначальные настройки карты: получение текущего местоположения, установки указателя на карте и фокусировка камеры карты на координаты текущего местоположения.

### **3.1.2.24 Функция listenForInitializeMap**

Функция `listenForSetAsDefaultPaymentMethod` создает подписку функции `initializeMapSaga` на событие `INITIALIZE_MAP_TRIGGER`. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция `initializeMapSaga`.

### **3.1.2.25 Функция receiveLocationUpdateSaga**

Функция `receiveLocationUpdateSaga` принимает на вход геолокацию пользователя и отправляет ее на сервер для актуализации, чтобы в дальнейшем сервис мог подобрать ближайшего для клиента водителя.

### **3.1.2.26 Функция bootstrapGPSSubscription**

Функция `bootstrapGPSSubscription` создает канал уведомлений между модулем GPS и модулем бизнес-логики, подписывая функцию `receiveLocationUpdateSaga` на получение обновленного местоположения пользователя.

### **3.1.2.27 Функция receiveWebSocketMessageSaga**

Функция `receiveWebSocketMessageSaga` принимает на вход очередное сообщение от сервера и получает его тип. В зависимости от типа сообщения и команды сервера функция будет перенаправлять выполнение в другие участки кода.

### **3.1.2.28 Функция bootstrapWebSocketSubscription**

Функция `bootstrapWebSocketSubscription` создает канал уведомлений между модулем работы с сервером (подписка через WebSocket) и модулем бизнес-логики, подписывая функцию `receiveWebSocketMessageSaga` на получение нового сообщения от сервера.

### **3.1.2.29 Функция chooseRouteSaga**

Функция `chooseRouteSaga` производит перевод приложения в режим выбора маршрута, т.е. удаляет все данные о выбранном ранее маршруте и классе машины.

### **3.1.2.30 Функция listenForChooseRoute**

Функция `listenForChooseRoute` создает подписку функции `chooseRouteSaga` на событие `CHOOSE_ROUTE.TRIGGER`. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция `chooseRouteSaga`.

### **3.1.2.31 Функция fetchPlacesSaga**

Функция `fetchPlacesSaga` принимает на вход строку с частью адреса, который ввел клиент в строку поиска, и получает список подходящих адресов с полным именем каждого адреса.

### **3.1.2.32 Функция listenForFetchPlaces**

Функция `listenForFetchPlaces` создает подписку функции `fetchPlacesSaga` на событие `FETCH_PLACES.TRIGGER`. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция `fetchPlacesSaga`.

### **3.1.2.33 Функция prepareRideDataSaga**

Функция `prepareRideDataSaga` переводит клиента из состояния выбора точки отправления и назначения в состояние просмотра маршрута и выбора класса машины.

### **3.1.2.34 Функция listenForPrepareRide**

Функция `listenForPrepareRide` создает подписку функции `prepareRideDataSaga` на событие `PREPARE_RIDE.TRIGGER`. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция `prepareRideDataSaga`.

### **3.1.2.35 Функция requestRideSaga**

Функция `requestRideSaga` принимает на вход выбранные клиентом класс машины и стоимость проездки в этом классе и выполняет ряд запросов для создания поездки и поиска водителя в реальном времени с учетом текущего местоположения пользователя и выбранной точки отправления. В случае успешного подбора происходит переход приложения в режим поездки и отображения информации о ней.

### **3.1.2.36 Функция listenForRequestRide**

Функция `listenForRequestRide` создает подписку функции `requestRideSaga` на событие `REQUEST_RIDE.TRIGGER`. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция `requestRideSaga`.

### **3.1.2.37 Функция answerToRideRequestSaga**

Функция `answerToRideRequestSaga` принимает на вход условия поездки, полученные от сервера по протоколу WebSocket, и переводит приложение в состояние предложения о поездке. Данная функция вызывается только у водителей.

### **3.1.2.38 Функция listenForAnswerToRideRequest**

Функция `listenForAnswerToRideRequest` создает подписку функции `answerToRideRequestSaga` на событие `ANSWER_TO_RIDE_REQUEST.TRIGGER`. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция `answerToRideRequestSaga`.

### **3.1.2.39 Функция setChosenLocationSaga**

Функция `setChosenLocationSaga` принимает на вход геопозицию, выбранную пользователем указателем на карте, и переводит географические координаты в текстовое описание этого места.

### **3.1.2.40 Функция listenForSetChosenLocation**

Функция `listenForSetChosenLocation` создает подписку функции `setChosenLocationSaga` на событие `SET_CHOSEN_LOCATION.TRIGGER`. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция `setChosenLocationSaga`, однако, ввиду того что это событие генерируется при движении карты, то оно может генерироваться десятки раз за секунду и нагружать сервер. Для исключения такого варианта используется механизм `debounce`: после получения события функция начинает выполняться не сразу, а спустя какое-то время; если же за период ожидания придет еще одно событие, то запланированное выполнение предыдущей функции отменится и будет запланировано заново. Это гарантирует выполнение только одного события при их быстром потоке.

### **3.1.2.41 Функция setRouteLocationSaga**

Функция `setRouteLocationSaga` принимает на вход одну из позиций маршрута, записывает в хранилище состояния и перемещает камеру.

### **3.1.2.42 Функция listenForSetRouteLocation**

Функция `listenForSetRouteLocation` создает подписку функции `setRouteLocationSaga` на событие `SET_ROUTE_LOCATION_TRIGGER`. При отправке этого события с модуля пользовательского интерфейса будет вызвана функция `setRouteLocationSaga`.

## **3.1.3 Модуль работы с хранилищем**

Модуль работы с хранилищем реализован с использованием функционального подхода разработки ввиду используемой библиотеки Redux.

Состояние приложения – обычный объект языка TypeScript, с той лишь оговоркой, что его изменение приводит к обновлению компонентов, которые подписаны на эти изменения. Данный объект не является объектом класса, как это принято в языках Java и C++. Объекты языка TypeScript могут создаваться без создания специальных конструкций, которых их описывают (например, без создания классов), это удобно, если данные объекты используются всего единожды. Для типизации таких объектов (например, состояние приложения) используются так называемые типы – аналог интерфейсов из Java в языке TypeScript.

Функции, в которых заключена логика по изменению состояния, называются `reducers`, или же обычными обработчиками событий. При поступлении очередного события от пользовательского интерфейса или бизнес логики данные обработчики будут вызываться друг за другом до тех пор, пока не найдется обработчик пришедшего события. В результате работы этой цепочки получается новый объект состояния, который заменяет предыдущий, тем самым заставляя всех подписчиков произвести перерисовку для отображения актуальных данных.

### **3.1.3.1 Тип ApplicationState**

Тип `ApplicationState` описывает тип хранилища состояния приложения и содержит следующие обязательные поля для экземпляра хранилища:

- `user` : `UserState` – хранилище данных о профиле пользователя;
- `history` : `HistoryState` – хранилище данных о истории поездок;

- payments : PaymentsState – хранилище данных о платежных методах;
- home : HomeState – хранилище всех необходимых данных для вызова такси.

### **3.1.3.2 Тип State<T>**

Тип State<T> шаблонный, он создает обертку над передаваемым типом, которая описывает состояние работы с данными хранилища. Состоит из следующих полей:

- isLoading : boolean – флаг, хранящий состояние о процессе загрузки данных;
- error : Optional<Error> – последняя возникшая ошибка при работе с хранилищем, может быть пустой в случае успешной работы и стабильного интернет-соединения;
- data : Optional<T> – данные, которые необходимо хранить. Поле может быть пустое в случае возникновения ошибки по расчете этих данных.

### **3.1.3.3 Тип UserState**

Тип UserState описывает структуру хранилища данных о профиле пользователя и является переименованным типом State<User>.

### **3.1.3.4 Тип User**

Тип User представляет из себя модель пользователя, зарегистрированного в системе. Данный класс содержит следующие поля:

- id : number – уникальный идентификатор пользователя;
- email : string – почтовый адрес;
- phone : string – номер телефона;
- firstName : string – имя;
- lastName : string – фамилия;
- gender : Gender – пол;
- driver : Optional<Driver> – данные о машине и балансе, если пользователь – водитель.

### **3.1.3.5 Перечисление Gender**

Перечисление Gender содержит доступный для выбора пол пользователя:

- Male – мужской пол;
- Female – женский пол.

### **3.1.3.6 Тип Driver**

Тип Driver содержит информацию о машине и балансе водителя. Класс содержит следующие поля:

- id : number – уникальный идентификатор;
- carBrand : string – марка и модель машины;
- carClass : CarClass – класс машины;
- balance : number – баланс водителя.

### **3.1.3.7 Перечисление CarClass**

Перечисление CarClass содержит возможные уровни машин, доступных к заказу:

- Economy – эконом-класс;
- Comfort – комфорт-класс;
- Business – бизнес-класс.

### **3.1.3.8 Тип HistoryState**

Тип HistoryState описывает структуру хранилища данных о профиле пользователя и является переименованным типом State<Array<Ride>>.

### **3.1.3.9 Тип Ride**

Тип Ride описывает структуру объекта поездки и содержит следующие поля:

- id : number – уникальный идентификатор поездки;
- client : User – пользователь, заказавший такси;
- driver : User – водитель, принял заказ;
- cost : number – стоимость поездки;
- payment: Optional<Payment> - транзакция оплаты поездки;
- startTime : number – время начала поездки;
- endTime : Optional<number> – время конца поездки;
- to : string – место назначения;
- from : string – место отправления;
- status : RideStatus – текущий статус поездки;

- paid : boolean – флаг, показывающий состояние оплаты поездки.

### 3.1.3.10 Перечисление RideStatus

Перечисление RideStatus отображает различные статусы, описывающие прогресс поездки:

- Starting – поездка начинается, водитель двигается к точке направления;
- InProgress – поездка началась, водитель двигается от точки направления к точке назначения;
- Completed – поездка завершена, водитель прибыл в точку назначения;
- NoRide – нет активной поездки.

### 3.1.3.11 Тип PaymentsState

Тип PaymentsState описывает структуру хранилища способов оплаты:

- list : State<Array<PaymentMethod>> – состояние списка методов оплаты;
- addCard : State<unknown> – состояние прогресса добавления карты;
- payment : State<unknown> – состояние процесса оплаты.

### 3.1.3.12 Тип PaymentMethod

Тип PaymentMethod описывает структуру объекта, хранящего данные о способе оплаты:

- id : number – уникальный идентификатор способа оплаты;
- type : PaymentMethodType – тип средства оплаты;
- isDefault : boolean – флаг, показывающий, является ли средство оплаты средством по умолчанию;
- details : Optional<PaymentMethodDetails> – детали банковской карты.

### 3.1.3.13 Перечисление PaymentMethodType

Перечисление PaymentMethodType отображает тип способа оплаты:

- Cash – оплата наличными;
- Card – оплата картой.

### **3.1.3.14 Тип PaymentMethodDetails**

Тип `PaymentMethodDetails` хранит информацию о картах и содержит следующие поля:

- `id` : `number` – уникальный идентификатор способа оплаты;
- `lastFour` : `string` – четыре последние цифры карты;
- `exp` : `string` – срок действия карты;
- `holder` : `string` – имя владельца карты;
- `brand` : `Brand` – тип карточки;
- `stripePaymentId` : `string` – уникальный номер карты в платежной системе Stripe.

### **3.1.3.15 Перечисление CarBrand**

Перечисление `CarBrand` отображает тип платежной карты:

- `Visa`;
- `Mastercard`;
- `AmericanExpress`.

### **3.1.3.16 Тип HomeState**

Тип `HomeState` описывает структуру объекта, хранящего состояние домашнего экрана:

- `pointerLocation` : `State<PointerLocation>` – состояние указателя на карте;
- `chooseRoute` : `ChooseRouteState` – состояние приложения в режиме выбора маршрута;
- `prepareRide` : `PrepareRide` – состояние приложения в режиме выбора класса (используется клиентом);
- `prepareDriverRide` : `PrepareDriverRide` – состояние приложения в режиме предложения поездки (используется водителем);
- `ride` : `RideState` – состояние текущей существующей поездки.

### **3.1.3.17 Тип PointerLocation**

Тип `PointerLocation` описывает состояние указателя на точку начала поездки на карте:

- `pointerLocation` : `Optional<ExtendedLocation>` – положение маркера на карте;
- `isMoving` : `boolean` – двигается карта или нет.

### **3.1.3.18 Тип ExtendedLocation**

Тип `ExtendedLocation` описывает структуру объекта, хранящего в себе данные о геолокации и названии определенного места на карте:

- `readableDescription` : `Optional<string>` – русскоязычное описание места на карте;
- `latitude` : `number` – широта;
- `longitude` : `number` – долгота.

### **3.1.3.19 Тип Location**

Тип `ExtendedLocation` описывает структуру объекта, хранящего в себе данные о геолокации:

- `latitude` : `number` – широта;
- `longitude` : `number` – долгота.

### **3.1.3.20 Тип PrepareRide**

Тип `PrepareRide` описывает состояние приложения в режиме выбора класса (используется клиентом):

- `isPreparing` : `boolean` – находится ли клиент в состоянии выбора класса машины;
- `rideRequest` : `State<RideRequest>` – объект, содержащий первоначальную информацию о поездке, просчитанную сервером на основе конечной и начальной точки маршрута;
- `isCarSearching` : `boolean` – находится ли приложение в состоянии поиска машины;
- `from` : `ExtendedLocation` – точка отправления;
- `to` : `ExtendedLocation` – точка назначения.

### **3.1.3.21 Тип RideRequest**

Тип `RideRequest` описывает объект, содержащий первоначальную информацию о поездке, просчитанную сервером на основе конечной и начальной точки маршрута:

- `calculatedTime` : `number` – расчетное время в пути;
- `route` : `Array<Location>` – маршрут поездки;
- `classes` : `Record<CarClass, number>` – доступные классы машин и цены на них.

### **3.1.3.22 Тип PrepareDriverRide**

Тип `PrepareRide` состояние приложения в режиме предложения поездки (используется водителем):

- `rideRequest` : `State<ExtendedRideRequest>` – объект, содержащий информацию о поездке, просчитанную сервером на основе конечной и начальной точки маршрута, стоимости и выбранного клиентом класса машины.

### **3.1.3.23 Тип ExtendedRideRequest**

Тип `ExtendedRideRequest` описывает объект, содержащий информацию о поездке, просчитанную сервером на основе конечной и начальной точки маршрута, стоимости и выбранного клиентом класса машины:

- `calculatedTime` : `number` – расчетное время в пути;
- `route` : `Array<Location>` – маршрут поездки;
- `to` : `ExtendedLocation` – место назначения;
- `from` : `ExtendedLocation` – место отправления;
- `cost` : `number` – стоимость поездки;
- `carClass` : `CarClass` – класс машины.

### **3.1.3.24 Тип ChooseRouteState**

Тип `ChooseRouteState` описывает состояние приложения в режиме выбора пункта начала и конца поездки:

- `lastChange` : `string` – последнее измененное поле для ввода;
- `isChoosingRoute` : `boolean` – находится ли приложение в режиме выбора маршрута;
- `to` : `State<DirectionChooseResult>` – состояние выбора конечной точки;
- `from` : `State<DirectionChooseResult>` – состояние выбора начальной точки.

### **3.1.3.25 Тип DirectionChooseResult**

Тип `DirectionChooseResult` описывает состояние выбора точки для определенного направления движения:

- `pickedLocation` : `Optional<ExtendedLocation>` – выбранная точка;

- `searchResults` : `Array<ExtendedLocation>` – места, найденные при вводе текста с адресом либо названием здания или организации.

### **3.1.3.26 Тип RideState**

Тип `RideState` описывает состояние приложения в режиме поездки:

- `status` : `RideStatus` – текущее состояние поездки;
- `driverPosition` : `Optional<Location>` – текущее положение машины с водителем;
- `ride` : `Optional<Ride>` – общие сведения о поездке.

### **3.1.3.27 Функция userReducer**

Функция `userReducer` принимает на вход два аргумента: состояние `UserState` и событие. Задача функции – менять компонент состояния `UserState` в зависимости от передаваемого события.

### **3.1.3.28 Функция historyReducer**

Функция `historyReducer` работает аналогично функции `userReducer`, с тем исключением, что первый параметр имеет тип `HistoryState`.

### **3.1.3.29 Функция paymentsReducer**

Функция `paymentsReducer` работает аналогично функции `historyReducer`, с тем исключением, что первый параметр имеет тип `PaymentsState`.

### **3.1.3.30 Функция homeReducer**

Функция `homeReducer` работает аналогично функции `historyReducer`, с тем исключением, что первый параметр имеет тип `HomeState`.

## **3.1.4 Модуль работы с сервером**

Модуль работы с сервером реализован с использованием объектно-ориентированного подхода с использованием библиотек `Axios` и `Socket.io`.

### **3.1.4.1 Класс RestGatewayAPI**

Класс RestGatewayAPI предоставляет базовый функционал для работы с серверным приложением по протоколу HTTP.

Класс RestGatewayAPI содержит следующие поля:

- authToken : string – авторизационный токен;
- axios : AxiosInstance – экземпляр HTTP клиента.

Класс RestGatewayAPI содержит следующие методы:

- initialize() : AxiosInstance – авторизационный токен;
- setAuthToken(string) – экземпляр HTTP клиента;
- post<K, T>(string, K) : Promise<Response<T>> – функция, выполняющая запрос на сервер методом POST;
- get<K, T>(string, Optional<K>) : Promise<Response<T>> – функция, выполняющая запрос на сервер методом GET.

### **3.1.4.2 Класс ConnectionGatewayAPI**

Класс ConnectionGatewayAPI предоставляет базовый функционал для работы с серверным приложением по протоколу WebSocket.

Класс ConnectionGatewayAPI содержит следующие поля:

- authToken : string – авторизационный токен;
- isClient : boolean – клиент или водитель;
- listeners : Array<(WSMessage) => void> – массив слушателей сообщений;
- socket : Socket | null – сокет для соединения с сервером.

Класс ConnectionGatewayAPI содержит следующие методы:

- setAuthToken(string) – установка authToken;
- setIsClient(boolean) – установка isClient;
- addEventListener((WSMessage) => void) : () => void – добавление слушателя;
- connect() – подключение к серверу;
- send<T>(WSMessageType, T) – отправка сообщения;
- disconnect() – отключение от сервера;
- retryConnection() – переподключение после обрыва.

### **3.1.4.3 Класс AuthAPI**

Класс AuthAPI предоставляет доступ к авторизационным маршрутам сервера.

**Класс AuthAPI** содержит следующие поля:

- `restGatewayAPI` : `RestGatewayAPI` – абстракция над Axios.
- Класс AuthAPI содержит следующие методы:
  - `login(string, string)` : `Promise<Response<Tokens>>` – запрос на регистрацию;
    - `register(RegisterPayload)` : `Promise<Response<Tokens>>` – запрос на регистрацию;
  - `refreshToken(string)` : `Promise<Response<Tokens>>` – запрос на обновление токена.

#### **3.1.4.4 Класс HistoryAPI**

Класс HistoryAPI предоставляет возможность получения списка поездок пользователя.

Класс HistoryAPI содержит следующие поля:

- `restGatewayAPI` : `RestGatewayAPI` – абстракция над Axios.
- Класс HistoryAPI содержит следующие методы:
  - `getHistory()` : `Promise<Response<Array<Ride>>>` – получение истории поездок.

#### **3.1.4.5 Класс HomeAPI**

Класс HomeAPI предоставляет возможность использования различных эндпоинтов для получения информации о названии мест, просчете маршрутов и стоимости поездок.

Класс HomeAPI содержит следующие поля:

- `restGatewayAPI` : `RestGatewayAPI` – абстракция над Axios.
- `connectionGatewayAPI` : `ConnectionGatewayAPI` – абстракция над Socket.io.

Класс HomeAPI содержит следующие методы:

- `decode(Location)` : `Promise<Response<ExtendedLocation>>` – получение текстового описания локации по координатам;
- `updateMyLocation(Location)` – отправка текущего местоположения на сервер;
- `requestRide(ExtendedRideRequest)` – запрос на поиск машин поблизости;
- `answerToRideRequest(WSMessageType)` – отправка ответа водителю на предложение о поездке;

- `fetchPlaces(string)` :  
`Promise<Response<Array<ExtendedLocation>>` – получение текстового описания мест, похожих по названию с передаваемой строкой;
- `calculateRideData(ExtendedLocation, ExtendedLocation)` : `Promise<Response<RideRequest>>` – получение первоначальной информации о поездке;
- `declineRideRequest()` – отмена поиска машины;
- `updateRideStatus(RideStatus)` – обновление статуса поездки.

### **3.1.4.6 Класс PaymentsAPI**

Класс PaymentsAPI предоставляет возможность использования платежной системы Stripe и ее серверной части.

Класс PaymentsAPI содержит следующие поля:

- `restGatewayAPI : RestGatewayAPI` – абстракция над Axios.

Класс PaymentsAPI содержит следующие методы:

- `getPaymentMethods()` :  
`Promise<Response<Array<PaymentMethod>>>` – получение списка способов оплаты пользователя;
- `setAsDefaultMethod(number)` :  
`Promise<Response<unknown>>` – установка способа оплаты по умолчанию;
- `addCard(CardMethodDetails)` :  
`Promise<Response<unknown>>` – добавление карты;
- `createPaymentIntent(CreatePaymentIntentInput)` :  
`Promise<Response<string>>` – генерация секретного ключа оплаты;
- `removePaymentMethod(number)` :  
`Promise<Response<unknown>>` – удаление способа оплаты;
- `paymentFinished(PaymentFinishedInput)` :  
`Promise<Response<unknown>>` – завершение платежа.

### **3.1.4.7 Класс ProfileAPI**

Класс ProfileAPI предоставляет возможность информации о пользователе.

Класс ProfileAPI содержит следующие поля:

- `restGatewayAPI : RestGatewayAPI` – абстракция над Axios.

Класс ProfileAPI содержит следующие методы:

- `getUser()` : `Promise<Response<User>>` – получение текущего пользователя.

### 3.1.5 Модуль работы с GPS

Модуль работы с GPS представлен классом `GeolocationService`. Данный класс работает с API операционных систем для получения информации о текущем местоположении устройства.

Класс `GeolocationService` содержит следующие поля:

- `latestLocation` : `Optional<Location>` – последняя полученная локация от операционной системы;

Класс `GeolocationService` содержит следующие методы:

- `initialize()` – функция инициализации и получения запроса на разрешение использования геолокации;
- `getLocation()` : `Location` – получение текущей локации;
- `subscribeToPositionChange((Location) => void)` :  
() => void – получение текущей локации.

### 3.1.6 Модуль вспомогательных утилит

#### 3.1.6.1 Класс `NavigationService`

Класс `NavigationService` используется как слой абстракции между кодом бизнес-логики и UI библиотекой.

Класс `NavigationService` содержит следующие поля:

- `navigationRef` : `NavigationContainerRef<any>` | null – ссылка на навигационный контейнер.

Класс `NavigationService` содержит следующие методы:

- `setNavigationRef(NavigationContainerRef<any>)` – установка значения переменной `navigationRef`;
- `goBack()` – выход на предыдущий экран.

#### 3.1.6.2 Класс `MapService`

Класс `MapService` используется как слой абстракции между кодом бизнес-логики и UI библиотекой.

Класс `MapService` содержит следующие поля:

- `map` : `MutableRefObject<MapView>` – ссылка на компонент карты.

Класс `MapService` содержит следующие методы:

- `getMapRef() : MutableRefObject<MapView>` – получение ссылки на карту;
- `animateCamera(Location, zoom)` – фокус на точку на карте;
- `animateToRegion(Location, Location)` – фокус на регион на карте.

## **3.2 Серверное приложение**

Сервер написан с использованием фреймворка NestJS, для которого обязательна разработка в объектно-ориентированном стиле.

### **3.2.1 Модуль получения запроса клиента**

#### **3.2.1.1 Класс AuthController**

Класс `AuthController` представляет из себя точку входа запроса клиента на авторизационные маршруты.

Класс `AuthController` содержит следующие поля:

- `authService : AuthService` – экземпляр класса `AuthService` из модуля авторизации.

Класс `AuthController` содержит следующие методы:

- `constructor(AuthService)` – конструктор класса;
- `login(LoginInput) : Promise<Tokens>` – принимает запрос на вход;
- `register(RegisterInput) : Promise<Tokens>` – принимает запрос на регистрацию;
- `refreshToken(RefreshInput) : Promise<Tokens>` – принимает запрос на обновление токенов.

#### **3.2.1.2 Класс MapsController**

Класс `MapsController` представляет из себя точку входа запроса клиента на маршруты, связанные с работой карты.

Класс `MapsController` содержит следующие поля:

- `mapsService : MapsService` – экземпляр класса `MapsService` из модуля расчета стоимости маршрута и поездки.

Класс `MapsController` содержит следующие методы:

- `constructor(MapsService)` – конструктор класса;
- `decode(DecodeInput) : Promise<DecodeOutput>` – принимает запрос на получение текстового описания места по его географическим координатам;

- `direction(DirectionInput) : Promise<DecodeOutput>`
- принимает запрос на просчет маршрута и стоимости поездки между двумя точками;
- `places(PlacesInput) : Promise<PlacesOutput>` – принимает запрос на поиск мест, попадающих под поисковый запрос.

### **3.2.1.3 Класс PaymentController**

Класс `PaymentController` представляет из себя точку входа запроса клиента на маршруты, связанные с платежной системой.

Класс `PaymentController` содержит следующие поля:

- `paymentService : PaymentService` – экземпляр класса `PaymentService` из модуля оплаты.

Класс `PaymentController` содержит следующие методы:

- `constructor(PaymentService)` – конструктор класса;
- `getPaymentMethods(Request) : Promise<Array<PaymentMethod>>` – принимает запрос на получение всех способов оплаты пользователя;
- `setDefaultMethod(Request, SetAsDefaultOrRemoveInput) : Promise<StatusWrapper>` – принимает запрос на установку метода оплаты по умолчанию;
- `addCard(Request, AddCardInput) : Promise<StatusWrapper>` – принимает запрос на добавление карты.
- `createPaymentIntent(Request, CreatePaymentIntentInput) : Promise<CreatePaymentOutput>` – принимает запрос на создание секретного ключа оплаты;
- `confirmPayment(Request, ConfirmPaymentInput) : Promise<StatusWrapper>` – принимает запрос на подтверждение оплаты;
- `removePaymentMethod(Request, SetAsDefaultOrRemoveInput) : Promise<StatusWrapper>` – принимает запрос на удаление метода оплаты.

### **3.2.1.4 Класс UserController**

Класс `UserController` представляет из себя точку входа запроса клиента на маршруты, связанные с информацией о профиле пользователя.

Класс `UserController` содержит следующие поля:

- `userService : UserService` – экземпляр класса `UserService` из модуля обработки данных клиента.

Класс `UserController` содержит следующие методы:

- `constructor(UserService)` – конструктор класса;

- `getUser(Request) : Promise<User>` – принимает запрос на получение профиля пользователя;
- `getHistory(Request) : Promise<Array<Ride>>` – принимает запрос на получение истории поездок пользователя;
- `getCurrentStatus(Request) : Promise<Ride | null>` – принимает запрос на получение текущей поездки пользователя.

### **3.2.2 Модуль авторизации**

Модуль авторизации представлен единственным классом `AuthService`, который сосредотачивает в себе всю логику обработки авторизационных запросов.

Класс `AuthService` содержит следующие поля:

- `authDataRepository : AuthDataRepository` – экземпляр класса `AuthDataRepository` из модуля работы с СУБД;
- `tokenProvider : TokenProvider` – экземпляр класса `TokenProvider`, предоставляющий возможность генерации авторизационных токенов;
- `userService : UserService` – экземпляр класса `UserService` из модуля обработки данных клиента;
- `hasher : Hasher` – экземпляр класса `Hasher`, позволяющий переводить пароли в хэш и сравнивать хэш с незашифрованной строкой.

Класс `AuthService` содержит следующие методы:

- `constructor(AuthDataRepository, TokenProvider, UserService, Hasher)` – конструктор класса;
- `login(string, string) : Promise<Tokens>` – функция входа в систему;
- `register(RegisterInput) : Promise<Tokens>` – функция регистрации в системе;
- `refreshToken(RefreshInput) : Promise<Tokens>` – функция обновления токенов.

### **3.2.3 Модуль обработки данных клиента**

Модуль обработки данных клиента представлен единственным классом `UserService`, который сосредотачивает в себе всю логику обработки запросов о получении данных о профиле.

Класс `UserService` содержит следующие поля:

- `userRepository : UserRepository` – экземпляр класса `UserRepository` из модуля работы с СУБД;

- `driverRepository` : `DriverRepository` – экземпляр класса `DriverRepository` из модуля работы с СУБД;
- `rideRepository` : `RideRepository` – экземпляр класса `RideRepository` из модуля работы с СУБД;
- `paymentService` : `PaymentService` – экземпляр класса `PaymentService` из модуля оплаты;

Класс `UserService` содержит следующие методы:

- `constructor(UserRepository, DriverRepository, RideRepository, PaymentService)` – конструктор класса;
- `getUser(number) : Promise<User>` – получение пользователя из базы данных;
- `createUser(RegisterInput) : Promise<User>` – создание нового пользователя;
- `getHistory(number) : Promise<Array<Ride>>` – получение списка поездок пользователя;
- `getLastestRide(number) : Promise<Ride | null>` – получение последней поездки пользователя.

### **3.2.4 Модуль работы с СУБД**

Данный модуль обеспечивает взаимодействие с базой данных для реализации функционала хранения и восстановления данных пользователя. В системе использована реляционная база данных PostgreSQL.

#### **3.2.4.1 Класс AuthData**

Класс `AuthData` представляет модель данных, содержащую следующие поля:

- `id : number` – уникальный идентификатор записи в таблице;
- `email : string` – почтовый адрес пользователя;
- `passwordHash : string` – хэш пароля.

#### **3.2.4.2 Класс Driver**

Класс `Driver` представляет модель данных, содержащую следующие поля:

- `id : number` – уникальный идентификатор записи в таблице;
- `carBrand : string` – модель машины;
- `carClass : CarClass` – класс машины;
- `balance : number` – счет водителя.

### **3.2.4.3 Класс Payment**

Класс `Payment` представляет модель данных, содержащую следующие поля:

- `id` : `number` – уникальный идентификатор записи в таблице;
- `user` : `User` – пользователь, инициировавший транзакцию;
- `method` : `PaymentMethod` – платежный метод;
- `amount` : `number` – размер платежа;
- `timestamp` : `number` – дата платежа.

### **3.2.4.4 Класс PaymentMethod**

Класс `PaymentMethod` представляет модель данных, содержащую следующие поля:

- `id` : `number` – уникальный идентификатор записи в таблице;
- `type` : `PaymentMethodType` – тип способа оплаты;
- `isDefault` : `boolean` – является ли средство оплаты способом по-умолчанию;
- `isVisible` : `boolean` – доступно ли средство для просмотра;
- `details` : `Optional<PaymentMethodDetails>` – детали карты;
- `user` : `User` – владелец способа оплаты.

### **3.2.4.5 Класс PaymentMethodDetails**

Класс `PaymentMethodDetails` представляет модель данных, содержащую следующие поля:

- `id` : `number` – уникальный идентификатор записи в таблице;
- `lastFour` : `string` – четыре последние цифры карты;
- `exp` : `string` – срок действия карты;
- `holder` : `string` – имя держателя карты;
- `brand` : `CardBrand` – тип карты;
- `stripePaymentId` : `string` – идентификатор способа оплаты в системе Stripe.

### **3.2.4.6 Класс Ride**

Класс `Ride` представляет модель данных, содержащую следующие поля:

- `id` : `number` – уникальный идентификатор записи в таблице;

- client : User – клиент, заказавший такси;
- driver : User – водитель, выполнивший заказ;
- payment : Optional<Payment> – оплата заказа;
- cost : number – стоимость заказа;
- startTime : number – время начала поездки;
- endTime : Optional<number> – время конца поездки;
- to : string – пункт назначения;
- from : string – пункт отправления;
- status : RideStatus – статус поездки;
- paid : boolean – оплачена ли поездка.

### 3.2.4.7 Класс User

Класс User представляет модель данных, содержащую следующие поля:

- id : number – уникальный идентификатор записи в таблице;
- email : string – электронная почта;
- phone : string – номер телефона;
- firstName : string – имя;
- lastName : string – фамилия;
- gender : Gender – пол;
- stripeClientId : Optional<string> – идентификатор клиента Stripe;
- driver : Optional<Driver> – данные о машине и балансе, если пользователь зарегистрирован как водитель.

### 3.2.4.8 Классы репозиториев

Классы репозиториев представляют из себя классы-помощники, которые упрощают задачу сохранения и поиска моделей в базе данных. Данные классы не содержат полей и методов, однако всю необходимую функциональность им предоставляют аннотации библиотеки TypeORM. Такими классами являются AuthDataRepository, DriverRepository, PaymentRepository, PaymentMethodRepository, PaymentMethodDetailsRepository, RideRepository и UserRepository.

### **3.2.5 Модуль оплаты**

#### **3.2.5.1 Класс PaymentService**

Класс `PaymentService` сосредотачивает в себе всю логику обработки платежей и различных действий с платежными средствами.

Класс `PaymentService` содержит следующие поля:

- `paymentRepository` : `PaymentRepository` – экземпляр класса `PaymentRepository` из модуля работы с СУБД;
- `paymentMethodRepository` : `PaymentMethodRepository` – экземпляр класса `PaymentMethodRepository` из модуля работы с СУБД;
- `paymentMethodDetailsRepository` : `PaymentMethodDetailsRepository` – экземпляр класса `PaymentMethodDetailsRepository` из модуля работы с СУБД;
- `userRepository` : `UserRepository` – экземпляр класса `UserRepository` из модуля работы с СУБД;
- `rideRepository` : `RideRepository` – экземпляр класса `RideRepository` из модуля работы с СУБД;
- `stripe` : `Stripe` – экземпляр класса `Stripe`.

Класс `PaymentService` содержит следующие методы:

- `constructor(PaymentRepository, PaymentMethodRepository, PaymentMethodDetailsRepository, UserRepository, RideRepository, Stripe)` – конструктор класса;
- `getPaymentMethods(number)` : `Promise<Array<PaymentMethod>>` – получает список всех способов оплаты пользователя;
- `setDefaultMethod(number, number)` – принимает запрос на установку метода оплаты по умолчанию;
- `addCard(number, AddCardInput)` – добавляет карту пользователю;
- `createPaymentIntent(number, CreatePaymentIntentInput)` : `Promise<CreatePaymentOutput>` – создает секретный ключ оплаты;
- `confirmPayment(number, ConfirmPaymentInput)` : `Promise<StatusWrapper>` – подтверждает оплату;
- `removePaymentMethod(number, number)` : `Promise<StatusWrapper>` – удаляет метод оплаты.

### **3.2.5.2 Класс Stripe**

Класс Stripe является оберткой, заключающей в себе всю логику работы с зависимостями Stripe SDK для платформы NodeJS.

Класс Stripe содержит следующие поля:

- stripeSecretKey : string – ключ доступа к Stripe API;
- usdRate : number – курс белорусского рубля к доллару;
- stripe : StripeSDK – экземпляр класса StripeSDK.

Класс Stripe содержит следующие методы:

– createIntent(string, number, string, Optional<string>) : Promise<CreateIntentOutput> – создает секретный ключ для оплаты.

### **3.2.6 Модуль подключения клиента**

Модуль подключения клиента состоит из единственного класса класса RTCController, который представляет из себя точку входа запроса клиента на маршруты, связанные с коммуникацией в реальном времени по протоколу WebSocket.

Класс RTCController содержит следующие поля:

– rtcService : RTCSERVICE – экземпляр класса RTCSERVICE из модуля RTC.

Класс RTCController содержит следующие методы:

– constructor(RTCSERVICE) – конструктор класса;

– handleConnection(Socket) – функция, вызываемая при подключении нового клиента;

– handleDisconnect(Socket) – функция, вызываемая при отключении клиента;

– handleLocationUpdate(WithUserRole<Location>, Socket) – функция, вызываемая при получении события обновления местоположения пользователя;

– handleRideRequest(WithUserRole<ExtendedRideRequest>, Socket) – функция, вызываемая при получении события о запросе водителей от клиента.

– handleRideStopSearch(Socket) – функция, вызываемая при получении события об отмене поиска водителей от клиента;

– handleRideStatusChange(WithUserRole<RideStatus>, Socket) – функция, вызываемая при получении события об изменении статуса поездки от водителя.

### 3.2.7 Модуль RTC

Модуль RTC представлен единственным классом RTCService, который сосредотачивает в себе всю логику обработки событий с WebSocket-сервера.

Класс RTCService содержит следующие поля:

- idBasedSocketHolder : Map<number, Socket> – структура данных, хранящая сокет пользователя по его идентификатору;
- socketBasedIdHolder : WeakMap<Socket, number> – структура данных, хранящая идентификатор пользователя по его сокету;
- clientDataHolder : Map<number, SocketUserInfo> – структура данных, хранящая данные о статусе клиента по его идентификатору;
- driverDataHolder : Map<number, DriverSocketUserInfo> – структура данных, хранящая данные о статусе водителя по его идентификатору;
- tokenProvider : TokenProvider – экземпляр класса TokenProvider, предоставляющий возможность генерации авторизационных токенов;
- userService : UserService – экземпляр класса UserService из модуля обработки данных клиента;
- geoUtils : GeoUtils – экземпляр класса GeoUtils;
- rideRepository : RideRepository – экземпляр класса RideRepository из модуля работы с СУБД.

Класс RTCService содержит следующие методы:

- constructor(TokenProvider, UserService, GeoUtils, RideRepository) – конструктор класса;
- getUserFromSocket(Socket) : Promise<User> – получает данные о пользователе исходя из текущего подключения;
- addUser(Socket) – добавление пользователей в список подключенных;
- removeUser(Socket) – удаление пользователя из списка подключенных;
- handleLocationUpdate(WithUserRole<Location>, Socket) – функция, вызываемая при получении события обновления местоположения пользователя;
- handleRideRequest(WithUserRole<ExtendedRideRequest>, Socket) – функция, вызываемая при получении события о запросе поиска водителей от клиента.
- handleStopSearch(Socket) – функция, вызываемая при получении события об отмене поиска водителей от клиента;

- `handleRideStatusChange(WithUserRole<RideStatus>, Socket)` – функция, вызываемая при получении события об изменении статуса поездки от водителя.

### 3.2.8 Модуль расчета стоимости и маршрута поездки

Модуль расчета стоимости и маршрута поездки представлен единственным классом `MapsService`, который сосредотачивает в себе всю логику обработки запросов о просчете данных маршрута и геолокации.

Класс `MapsService` содержит следующие поля:

- `geocoding` : `Geocoding` – экземпляр класса `Geocoding` из модуля работы с сервисами Google;
- `places` : `Places` – экзэмпляр класса `Places` из модуля работы с сервисами Google;
- `directions` : `Directions` – экземпляр класса `Directions` из модуля работы с сервисами Google.

Класс `MapsService` содержит следующие методы:

- `constructor(Geocoding, Places, Directions)` – конструктор класса;
- `decode(Location) : Promise<ExtendedLocation>` – вычисляет текстовое описание места по его географическим координатам;
- `getDirection(Location, Location) : Promise<DirectionOutput>` – просчитывает маршрута и стоимость поездки между двумя точками;
- `searchPlaces(string) : Promise<Array<ExtendedLocation>>` – производит поиск мест, подходящих по названию с переданной строкой.

### 3.2.9 Модуль работы с сервисами Google

#### 3.2.9.1 Класс GoogleMaps

Класс `GoogleMaps` хранит в себе клиент для работы с Google Maps API.

Класс `GoogleMaps` содержит следующие поля:

- `client` : `Client` – объект клиента для доступа к сервисам Google Maps.

#### 3.2.9.2 Класс Places

Класс `Places`, который сосредотачивает в себе использование Google Places API для поиска мест по текстовой строке.

Класс `Places` содержит следующие поля:

- `maps` : `GoogleMaps` – экземпляр класса `GoogleMaps`;
- `mockPlaces` : `boolean` – флаг, показывающий необходимость подменить запрос на сервис тестовыми данными.

Класс `Places` содержит следующие методы:

- `constructor(GoogleMaps)` – конструктор класса;
- `search(string) : Promise<Array<ExtendedLocation>>` – производит поиск возможных мест, используя часть названия этого места.

### 3.2.9.3 Класс Geocoding

Класс `Geocoding` сосредотачивает в себе использование Google Geocoding API для перевода координат в текстовое описание места.

- `maps` : `GoogleMaps` – экземпляр класса `GoogleMaps`;
- `geoUtils` : `GeoUtils` – экземпляр класса `GeoUtils` ;
- `mockGeocoding` : `boolean` – флаг, показывающий необходимость подменить запрос на сервис тестовыми данными.

Класс `Places` содержит следующие методы:

- `constructor(GeoUtils, GoogleMaps)` – конструктор класса;
- `decode(Location) : Promise<ExtendedLocation>` – производит определение места и возвращает его текстовое описание исходя из переданных координат.

### 3.2.9.4 Класс Directions

Класс `Directions` сосредотачивает в себе использование Google Directions API для нахождения маршрута и длительности поездки между двумя точками.

Класс `Directions` содержит следующие поля:

- `maps` : `GoogleMaps` – экземпляр класса `GoogleMaps`;
- `mockDirections` : `boolean` – флаг, показывающий необходимость подменить запрос на сервис тестовыми данными.

Класс `Directions` содержит следующие методы:

- `constructor(GoogleMaps)` – конструктор класса;
- `getDirection(Location, Location) : Promise<Array<Location>>` – поиск кратчайшего маршрута между двумя точками.

## 4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

### 4.1 Добавление платежной карты

После регистрации в системе и входа в аккаунт у клиента появляется возможность выбора платежного средства: наличные или банковская карта. Для проведения различных действий по добавлению карты, возврату и списанию средств с карты используется платежная система Stripe.

При переходе пользователя на экран добавления карты пользователь получит возможность ввести свои данные в форму, предоставляемую официальной библиотекой Stripe:

```
<CardField
    style={{
        width: Dimensions.get('window').width -
    defaultTheme.spacer * 4,
        height: 100,
        marginLeft: defaultTheme.spacer * 2,
    }
    postalCodeEnabled={false}
    cardStyle={{
        fontFamily: getFontName(FontType.semibold),
        textColor: colors.black,
    }
    onCardChange={(data) => (cardData.current = data)}
/>
```

Форма, предоставляемая разработчиками платежной системы Stripe, является безопасной с точки зрения обработки данных о карте: после ввода данных информация о CVV коде не предоставляется, а вместо уникального 16-тизначного номера карты возвращаются лишь последние четыре цифры для отображения пользователю. Имеющейся у разработчика после заполнения формы информации недостаточно для проведения платежа, поэтому для сохранения конфиденциальности данных мобильная библиотека Stripe будет проводить платеж самостоятельно с помощью серверной части данной библиотеки.

Для сохранения карты (то есть для получения возможности проведения оплаты автоматически без повторного ввода данных пользователем) в платежной системе используется механизм уникальных идентификаторов. Данный идентификатор генерируется платежной системой при первой оплате новой картой, соответственно для сохранения карты требуется произвести платеж на минимальную доступную сумму (в системе Stripe – 1 USD). В последующих платежах полученный идентификатор переиспользуется.

Из-за соображений конфиденциальности и безопасности процесс проведения платежа разбит на две части: создание запроса на платеж и сам

платеж соответственно. Создание запроса на платеж производится на сервере и представляет из себя регистрацию нового платежа в специальном платежном шлюзе Stripe с последующей генерацией уникального ключа, который необходим для проведения на стороне клиента. При этом запрос на платежный шлюз будет содержать уникальный ключ продавца в системе Stripe, выдающийся каждому, кто регистрирует компанию в данной платежной системе.

```
const params: StripeSDK.PaymentIntentCreateParams = {  
    amount: Math.ceil((bynAmount * 100) / this.usdRate),  
    currency: 'usd',  
    customer: customer.id,  
    payment_method_options: {  
        card: {  
            request_three_d_secure: requestThreeDSecure ||  
'automatic',  
            },  
        },  
    payment_method_types: ['card'],  
};  
  
const paymentIntent = await  
this.stripe.paymentIntents.create(params);
```

Сгенерированный объект paymentIntent содержит в себе уникальный ключ для проведения платежа.

Получив данный ключ, мобильное приложение начинает процесс непосредственной оплаты. Для этого полученный ключ и флаг сохранения карты передаются в специальную функцию оплаты, которую предоставляет мобильная библиотека:

```
const { error, paymentIntent }: ConfirmPaymentResult = yield  
call(confirmPayment, secret, {  
    type: 'Card',  
    setupFutureUsage: 'OffSession',  
});
```

Так как данные о карте не предоставляются разработчику из соображений безопасности, то это значит, что сам запрос на платежный шлюз с конфиденциальными данными должен проводиться самой библиотекой. При старте платежа библиотека, используя ссылку на созданную ранее форму ввода данных, самостоятельно извлекает из нее все необходимые данные, и, с помощью полученного от сервера ключа, делает запрос на внутренние сервера системы Stripe. В результате успешного проведения платежа возвращается информация о платеже, иначе – ошибка с детальной информацией о причине отказа об оплате.

Информация о платеже содержит необходимый нам уникальный идентификатор платежного средства. Для проведения дальнейших оплат данный идентификатор, четыре последние цифры номера карты, срок действия и имя держателя отправляется на сервер для сохранения в базу данных:

```
const result = yield call(paymentsAPI.addCard, {
    ...action.payload,
    stripePaymentId: paymentIntent.paymentMethodId,
}) ;
```

В дальнейшем при использовании приложения пользователю не придется вводить свои персональные данные для проведения платежа по оплате поездки.

## 4.2 Составление маршрута поездки

Составление маршрута поездки включает в себя два этапа: выбор точек отправления и назначения и построения маршрута на карте. Выбор точки отправления можно производить как в режиме ручного поиска по адресу, так и с помощью выбора точки прямо на карте, расположенной на экране смартфона.

В режиме ручного поиска пользователь вводит часть адреса в строке поиска, данная строка отправляется на сервер для поиска полного адреса с использованием сервисов поиска адресов:

```
if (action.payload.toSearch.trim().length < 3) {
    yield put(FETCH_PLACES.COMPLETED({ results: [], direction:
action.payload.direction }));
    return;
}

const result: PlacesResponse = yield call(homeAPI.fetchPlaces,
action.payload.toSearch.trim());
```

Для оптимизации таких запросов на сервер будет производиться отправка строки лишь в том случае, если ее длина составляет три и более символа. При получении строки сервером происходит обращение к сторонним сервисам для получения полного адреса:

```
if (this.mockPlaces) {
    return places.filter((place) =>
place.readableLocation.toLowerCase().includes(part.toLowerCase()))
);
} else {
    const result = await this.maps.client.placeAutocomplete({
```

```

    params: {
      key: this.maps.mapsKey,
      input: part,
      language: Language.ru,
    },
  )};

  return result.data.predictions.map((prediction) => ({
    readableLocation: prediction.description,
    longitude: (prediction as any).location.lng,
    latitude: (prediction as any).location.lat,
  }));
}

```

В режиме разработки для сокращения количества запросов сервера на сторонние сервисы используется собственная коллекция тестовых адресов. После получения полного адреса, строка с названием места/улицы и его географические координаты возвращаются клиенту для последующей отрисовки на карте.

При выборе точки отправления пользователю доступна возможность выбора точки назначения на карте. Для этого на главном экране располагается карта с указателем на выбранное местоположение, первоначально указывающий на текущее местоположение пользователя. При перемещении карты указатель будет показывать новую точку начала поездки. Местоположение выбранной точки на карте доступно разработчику в виде ее географических координат. Для проверки информации о точке пользователю требуется получить информацию о том, что в ней находится, поэтому выбранные географические координаты отправляются на сервер для обработки сервисами обратного геокодирования:

```

export function* setChosenLocationSaga(action: ReturnType<typeof
SET_CHOSEN_LOCATION.TRIGGER>): SagaIterator {
  const isMoving = yield select(getIsPointerMoving);

  if (isMoving) return;

  yield put(SET_CHOSEN_LOCATION.STARTED());

  if (action.payload) {
    const result: DecodeResponse = yield
    call(homeAPI.decode, action.payload);

    if (result.data) {
      yield
      put(SET_CHOSEN_LOCATION.COMPLETED(result.data));
    }
  }

  if (result.error) {

```

```

        yield call(logger.log, result.error);
        yield call(toastService.showError, result.error);
        yield put(SET_CHOSEN_LOCATION.FAILED(result.error));
    }
} else {
    yield put(SET_CHOSEN_LOCATION.COMPLETED());
}
}

export function* listenForSetChosenLocation(): SagaIterator {
    yield debounce(500, SET_CHOSEN_LOCATION.TRIGGER,
setChosenLocationSaga);
}

```

Учитывая то, что пользователь может перемещать карту очень быстро и события о смене позиции указателя будут генерироваться раз в несколько миллисекунд, была проведена оптимизация с использованием функции debounce: функция setChosenLocationSaga будет вызвана лишь в том случае, если за последние 500 миллисекунд не было сгенерировано нового события о перемещении указателя на карте. Данная оптимизация позволяет предотвратить генерацию множества ненужных запросов на сервер, тем самым исключая возможность его перегрузки. При получении географических координат сервер преобразует его в читаемый адрес с помощью сервисов обратного геокодирования:

```

if (this.mockGeocoding) {
    const data = places.find((place) => {
        const distance = this.geoUtils.getDistance(
            location.latitude,
            location.longitude,
            place.latitude,
            place.longitude,
            'K'
        );
        return distance < 1;
    });
    if (data) return data;
    return {
        latitude: location.latitude,
        longitude: location.longitude,
        readableLocation: `Заглушка ${Math.random() * 100).toFixed()} `,
    };
} else {
    const decoded = await this.maps.client.reverseGeocode({
        params: {

```

```

        key: this.maps.mapsKey,
        latlng: location,
        language: Language.ru,
    },
});

return {
    latitude: location.latitude,
    longitude: location.longitude,
    readableLocation:
decoded.data?.results[0]?.address_components[0]?.short_name ?? 'Неизвестно',
};
}

```

В режиме разработки для сокращения количества запросов сервера на сторонние сервисы используется собственная коллекция тестовых адресов. После получения полного адреса, строка с названием места/улицы и его географические координаты возвращаются клиенту для последующей отрисовки на карте.

После выбора точек отправления и назначения приложение переходит в состояние отображения маршрута. В данном состоянии пользователь видит маршрут поездки между точками на карте и стоимости поездки в различных классах машин. Для получения данной информации на сервер отправляется запрос, содержащий точки отправления и назначения:

```

yield call(mapService.animateToRegion, action.payload.from,
action.payload.to);
yield call(bottomSheetService.minimize);

const result: DirectionsResponse = yield
call(homeAPI.calculateRideData, action.payload.to,
action.payload.from);

```

Перед отправкой запроса происходит анимация камеры карты до положения, в котором на экране смартфона были бы видны точки назначения и отправки. После получения точек сервер использует сервисы для построения маршрутов, в результате работы которых предоставляется массив координат, составляющий маршрут, а также время в пути с учетом текущей ситуации на дорогах:

```

const result = await this.maps.client.directions({
params: {
    key: this.maps.mapsKey,
    origin: from,
    destination: to,
    language: Language.ru,
},

```

```

}) ;

return {
    minutes: result.data.routes[0].legs[0].duration.value / 60,
    route: result.data.routes[0].overview_path.map((path) => ({
        latitude: path.lat,
        longitude: path.lng,
    })),
};

```

Полученное время далее используется для вычисления стоимости поездки в каждом классе с учетом стоимости подачи машины:

```

const direction = await this.directions.getDirection(from, to);
const time = Math.ceil(direction.minutes);

return {
    calculatedTime: time,
    route: direction.route,
    classes: {
        [CarClass.Economy]: 3 + 0.3 * time,
        [CarClass.Comfort]: 4 + 0.4 * time,
        [CarClass.Business]: 5 + 0.5 * time,
    },
};

```

### 4.3 Отслеживание местоположения водителя

Для отображения текущего местоположения на экране клиента водителя и поиска ближайшего к клиенту водителя сервер должен владеть актуальной информацией о местонахождении водителя. Местоположение предоставляется GPS спутниками, обменивающимися информацией о координатах с операционной системой мобильного устройства. Для регулярной доставки данных со спутника на сервер используется протокол WebSocket, который был выбран ввиду его возможности обмена сообщениями и поддерживания соединения между сервером и клиентом, затрачивая при этом небольшое количество ресурсов.

При успешной авторизации приложение устанавливает соединение с сервером, помещая авторизационный токен в заголовок запроса, который однозначно идентифицирует подключающегося пользователя:

```

this.socket = io(environmentConfig.get('connectionGatewayAPI'),
{
    transports: ['websocket'],
    auth: {
        Authorization: `Bearer ${this.authToken}`,
    },
}

```

```
    reconnection: true,  
});
```

При возникновении неполадок в соединении приложение будет пытаться восстановить соединение до тех пор, пока оно в конечном итоге не будет восстановлено.

После установления соединения в приложении создается асинхронный обработчик событий об изменении местоположения, который будет отсылать соответствующее событие на сервер при получении новых данных от GPS спутника:

```
export const gpsSubscribe: Subscribe<Location> = (emitter) =>  
geolocationService.subscribeToPositionChange(emitter);  
  
export const createGPSMessagesReceiver = ():  
EventChannel<Location> => eventChannel<Location>(gpsSubscribe);  
  
export function* receiveLocationUpdateSaga(location: Location):  
SagaIterator {  
    const user = yield select(getExistingUser);  
  
    if (!user) return;  
  
    yield call(homeAPI.updateMyLocation, location);  
}  
  
export function* bootstrapGPSSubscription(): SagaIterator {  
    const gpsEventChannel = yield  
call(createGPSMessagesReceiver);  
  
    yield takeLatest(gpsEventChannel,  
receiveLocationUpdateSaga);  
}
```

При этом этот обработчик не будет уничтожен при выходе с аккаунта, обновления будут отбрасываться в случае отсутствия авторизованного в приложении пользователя.

После получения события об изменении местоположения сервер обновляет данные о конкретном водителе в in-memory хранилище, что позволяет анализировать наиболее актуальную информацию о местоположении при подборе потенциального водителя для поездки. При нахождении водителя в состоянии поездки данные о его местоположении будут отправлены клиенту для обновления маркера на его карте:

```
const toSearchIn = location.isClient ? this.clientDataHolder :  
this.driverDataHolder;  
const userId = this.socketBasedIdHolder.get(socket);  
// We setting this data on init, so it cant be undefined
```

```

const info = toSearchIn.get(userId) as SocketUserInfo;
info.location = location.data;

toSearchIn.set(userId, info);

// If this user on ride - send update to companion
if (info.companionId) {

    this.idBasedSocketHolder.get(info.companionId)?.emit(WSMessageTy
pe.LocationUpdate, {
        from: userId,
        location,
    });
}

```

#### 4.4 Поиск водителя

После получения данных о маршруте и выборе класса машины, мобильное приложение отправляет на сервер запрос о поиске водителя для совершения поездки:

```

const from = yield select(getPrepareRideFromLocation);
const to = yield select(getPrepareRideToLocation);
const request: RideRequest = yield select(getRideRequest);

yield call(homeAPI.requestRide, {
    to,
    from,
    ...action.payload,
    calculatedTime: request.calculatedTime,
    route: request.route,
});

```

Запрос содержит в себе точки отправления и назначения, класс машины, расчетное время и маршрут, которые далее будут отправляться потенциальным водителям. Все запросы в процессе поиска водителя происходят с помощью протокола WebSocket, который позволяет поддерживать постоянное соединение с сервером и обмениваться сообщениями в режиме реального времени.

После получения запроса о начале поездки сервер начинает поиск машин с возможностью отмены пользователем: при нажатии пользователем на кнопку отмены на сервер будет отправлено событие, которое прервет выполнение функции поиска водителя:

```

async handleStopSearch(socket: Socket) {
    const clientId = this.socketBasedIdHolder.get(socket);
    const client = this.clientDataHolder.get(clientId);

```

```

        client.stopSearch && client.stopSearch();
    }
}

```

В интересах клиента и для сокращения времени ожидания предпочтение в поиске водителя отдается тому, кто находится ближе к клиенту. Массив свободных водителей сортируется по возрастанию расстояния между водителем и выбранной точкой начала поездки по алгоритму вычисления расстояния на сфере:

```

getDistance = (lat1: number, lon1: number, lat2: number, lon2: number, unit: string): number => {
    if (lat1 == lat2 && lon1 == lon2) {
        return 0;
    } else {
        const radlat1 = (Math.PI * lat1) / 180;
        const radlat2 = (Math.PI * lat2) / 180;
        const theta = lon1 - lon2;
        const radtheta = (Math.PI * theta) / 180;

        let dist =
            Math.sin(radlat1) * Math.sin(radlat2) +
        Math.cos(radlat1) * Math.cos(radlat2) * Math.cos(radtheta);
        if (dist > 1) {
            dist = 1;
        }
        dist = Math.acos(dist);
        dist = (dist * 180) / Math.PI;
        dist = dist * 60 * 1.1515;
        if (unit == 'K') {
            dist = dist * 1.609344;
        }
        if (unit == 'N') {
            dist = dist * 0.8684;
        }
        return dist;
    }
};

```

Для расчета расстояния на сфере используется метод вычисления расстояний на большом круге. Длина дуги большого круга – кратчайшее расстояние между любыми двумя точками находящимися на поверхности сферы, измеренное вдоль линии соединяющей эти две точки (такая линия носит название ортодромии) и проходящей по поверхности сферы или другой поверхности вращения. Сферическая геометрия отличается от обычной Евклидовой и уравнения расстояния также принимают другую форму. В Евклидовой геометрии, кратчайшее расстояние между двумя точками – прямая линия. На сфере, прямых линий не бывает. Эти линии на сфере

являются частью больших кругов – окружностей, центры которых совпадают с центром сферы.

Вычисление расстояния этим методом более эффективно и во многих случаях более точно, чем вычисление его для спроектированных координат (в прямоугольных системах координат), поскольку, во-первых, для этого не надо переводить географические координаты в прямоугольную систему координат (осуществлять проекционные преобразования) и, во-вторых, многие проекции, если неправильно выбраны, могут привести к значительным искажениям длин в силу особенностей проекционных искажений.

После сортировки водителей сервер начинает поочередный опрос каждого с целью подтверждения приема нового заказа. Ввиду того, что запросы выполняются по асинхронному протоколу WebSocket, приостановить исполнение кода на стороне виртуальной машины до получения ответа от водителя невозможно, поэтому для этого была создана специальная программная абстракция:

```
return new Promise((resolve, reject) => {
    waitFor.forEach((msg) => {
        const listener = (data) => {
            socket.removeListener(msg, listener);
            wasResolved = true;
            resolve({ event: msg, data });
        };
        socket.on(msg, listener);

        setTimeout(() => {
            if (wasResolved) return;

            socket.emit(onTimeoutEvent);
            reject();
        }, timeout);
    });
});
```

Данный код предоставляет объект, который способен блокировать асинхронный поток кода до получения ответа от конкретного водителя. Для этого на сокет водителя добавляется новый слушатель, который при получении ответа от водителя разблокирует поток выполнения кода. Если в течении некоторого времени ответ получен не был, то данный запрос считается отклоненным. При получении отказа на предложение о поездке сервер отправляет запрос следующему водителю, который расположен к точке отправления ближе всех оставшихся водителей.

При успешном подборе водителя создается новый экземпляр поездки, устанавливается статус водителя в состояние занятого и отправляется событие об успешном начале поездки водителю и клиенту. В свою очередь мобильное

приложение переходит в состояние отображения информации о поездке, в котором на карте отрисовывается маршрут и текущее местоположение водителя, а также отображается меню, содержащее в себе информацию о водителе или клиенте, машине и времени поездки.

## **5 ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ**

Тестирование программного обеспечения (ПО) позволяет определить соответствие ожидаемого и фактического поведения данного ПО, а также выявить его дефекты. Одним из основных видов тестирования программного обеспечения является функциональное тестирование. Функциональное тестирование обеспечивает проверку соответствия разработанного программного обеспечения его исходным функциональным требованиям. Данный вид тестирования предполагает проверку правильности работы реализованных функций на различных наборах входных данных.

По степени доступа к программному коду можно выделить следующие виды тестирования: метод белого ящика, метод черного ящика и метод серого ящика. Метод белого ящика предполагает тестирование программы с доступом к ее коду. Данный метод позволяет учесть больше возможных вариантов поведения программы по сравнению с методом черного ящика, однако данный метод не всегда учитывает реальную среду выполнения программы и позицию конечного пользователя.

Метод черного ящика предполагает тестирование программы без доступа к ее коду. Данный метод позволяет исследовать поведение программы с позиции конечного пользователя, но при таком подходе зачастую тестируется лишь ограниченный набор из возможных вариантов поведения данной программы.

Метод серого ящика предполагает наличие частичного доступа к программному коду или частичное понимание архитектуры тестируемого ПО. Метод серого ящика сочетает в себе достоинства и недостатки двух вышеупомянутых методов тестирования. Данный метод будет использован в тестировании разработанного мобильного приложения.

По степени автоматизации тестирование ПО можно разделить на ручное и автоматизированное. Ручное тестирование предполагает выполнение тест-кейсов человеком вручную. Автоматизированное тестирование предполагает использование специального программного обеспечения для выполнения тест-кейсов и сравнения ожидаемого и фактического результата работы тестируемого программного обеспечения. Однако и в случае автоматизированного тестирования разработка тест-кейсов и оценка результатов тестирования осуществляется человеком.

Для проверки соответствия реализованного мобильного приложения его функциональным требованиям было проведено ручное тестирование. Тестовый сценарий представляет собой набор шагов, параметров и условий для проверки правильности реализации какой-либо функции. В таблице 5.1 приведены тестовые сценарии для проверки правильности реализации основных функций приложения, ожидаемые и фактические результаты поведения приложения, а также выводы о прохождении тестов. Вывод о прохождении теста делается в результате сравнения ожидаемого и фактического результатов поведения приложения.

Таблица 5.1 – Функциональное тестирование приложения

Тестовый сценарий	Ожидаемый результат	Фактический результат	Тест пройден
1	2	3	4
Начать регистрацию и ввести уже существующий в системе электронный адрес и нажать на кнопку регистрации.	Ошибка о невозможности использования данного адреса должна отобразиться сверху экрана.	Ошибка о невозможности использования данного адреса отобразилась сверху экрана.	Да.
Зайти на экран входа и ввести несуществующую пару электронного адреса и пароля. Нажать на кнопку входа.	Ошибка о некорректности введенных данных должна отобразиться сверху экрана.	Ошибка о некорректности введенных данных отобразилась сверху экрана.	Да.
Зайти на экран входа и ввести существующую пару электронного адреса и пароля. Нажать на кнопку входа.	Успешное завершение процесса авторизации и последующий переход на главный экран.	Успешное завершение процесса авторизации и последующий переход на главный экран.	Да.
Зайти на экран входа и ввести существующую пару электронного адреса и пароля. Нажать на кнопку входа. Выключить приложение и открыть его заново.	Должен произойти автоматический вход в систему и переход на главный экран.	Произошел автоматический вход в систему и переход на главный экран.	Да.
Открыть приложение, выполнить авторизацию и перейти на главный экран с картой. Раскрыть боковое меню и нажать на кнопку выхода.	Должен произойти выход на экран авторизации.	Произошел выход на экран авторизации.	Да.

Продолжение таблицы 5.1

1	2	3	4
Раскрыть боковое меню, перейти на экран способов оплаты. Нажать на любой способ оплаты.	Выбранный способ оплаты должен быть установлен в качестве способа оплаты по умолчанию.	Выбранный способ оплаты установлен в качестве способа оплаты по умолчанию.	Да.
Раскрыть боковое меню, перейти на экран способов оплаты. Нажать на кнопку добавления карты.	Должен произойти переход на экран добавления карты с расположенной на нем формой.	Произошел переход на экран добавления карты с расположенной на нем формой.	Да.
Раскрыть боковое меню, перейти на экран способов оплаты. Нажать на кнопку добавления карты. Ввести некорректные данные и нажать на кнопку добавления.	Должна появиться ошибка о некорректности введенных данных.	Появилась ошибка о некорректности введенных данных.	Да.
Раскрыть боковое меню, перейти на экран способов оплаты. Нажать на кнопку добавления карты. Ввести корректные данные и нажать на кнопку добавления.	Должен произойти переход на экран, содержащий список доступных платежных методов. Новая карта должна появиться в списке.	Произошел переход на экран, содержащий список доступных платежных методов. Новая карта появилась в списке.	Да.
Раскрыть боковое меню, перейти на экран способов оплаты. Нажать на кнопку добавления карты. Заполнить поля формы любыми данными и нажать на кнопку добавления. Попытаться покинуть экран.	Попытки покинуть экран во время загрузки должны быть блокированы.	Попытки покинуть экран во время загрузки заблокированы.	Да.

Продолжение таблицы 5.1

1	2	3	4
Перейти на главный экран.	Камера карты должна отобразить регион текущего местоположения пользователя.	Камера карты должна отобразить регион текущего местоположения пользователя.	Да.
Перейти на главный экран. Начать процесс выбора точки отправления. Переместить указатель на карте.	Сверху экрана должна появиться информация и выбранном месте. В нижнем меню данная точка должна быть установлена как точка отправления.	Сверху экрана должна появилась текстовая информация и выбранном месте. В нижнем меню данная точка установлена как точка отправления.	Да.
Перейти на главный экран. Начать процесс выбора точки назначения. Переместить указатель на карте.	Сверху экрана должна появиться информация и выбранном месте. В нижнем меню данная точка должна быть установлена как точка назначения.	Сверху экрана должна появилась текстовая информация и выбранном месте. В нижнем меню данная точка установлена как точка отправления.	Нет.
Перейти на главный экран. Раскрыть нижнее меню. Ввести в строку поиска адреса отправления или назначения меньше трех символов.	Список доступных адресов должен оставаться пустым.	Список доступных остался пустым.	Да.
Перейти на главный экран. Раскрыть нижнее меню. Ввести в строку поиска адреса отправления больше трех символов.	Список доступных к выбору адресов должен отобразиться под строкой поиска.	Список доступных к выбору адресов отобразился под строкой поиска.	Да.

Окончание таблицы 5.1

1	2	3	4
Перейти на главный экран. Выбрать точку назначения и отправления.	Приложение должно перейти в состояние выбора класса машины: отобразить карту с маршрутом и список доступных классов такси.	Приложение отобразило карту с маршрутом и список доступных классов такси.	Да.
Перевести приложение в состояние выбора класса машины, выбрать класс и начать поиск. Нет доступных водителей.	Сообщение об ошибке поиска должно быть отображено сверху экрана. Поиск должен быть прекращен.	Сообщение об ошибке поиска отобразилось сверху экрана. Поиск был прекращен.	Да.
Перевести приложение в состояние выбора класса машины, выбрать класс и начать поиск. Водитель найден.	Приложение должно перейти в состояние поездки: на карте должно отображаться текущее местоположение водителя и информация о нем.	Приложение должно перешло в состояние поездки: на карте отобразилось текущее местоположение водителя и информация о нем.	Да.
Перевести состояние приложение в состояние поездки. Закрыть приложение и открыть его заново.	Информация о поездке и местоположении водителя должно быть восстановлено. Приложение должно быть повторно переведено в состояние поездки.	Информация о поездке и местоположении водителя восстановлена. Приложение переведено в состояние поездки.	Да.
Перевести состояние приложение в состояние поездки. Завершить поездку.	Уведомление о состоянии оплаты должно быть отображено сверху экрана.	Уведомление о состоянии оплаты отобразилось сверху экрана.	Да.

По результатам функционального тестирования можно сделать вывод о правильной реализации функционала приложения. Все тесты, связанные с авторизацией, проведением платежей и процессом поездки, были успешно пройдены. Были обнаружены проблемы с возможностью выбора пункта назначения на карте. Тем не менее такое поведение приложения не является критичным, так как весь первостепенный функционал (поиск машин, отслеживание положения водителя, оплата и авторизация) отрабатывает исправно, пункт назначения можно выбрать путем заполнения адреса в строке поиска.

## 6 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

### 6.1 Аппаратные и программные требования

Для запуска приложения необходима операционная система iOS версии 11.0 и выше, или OS Android версии 5 и выше. Приложение может быть запущено как на смартфонах, так и на планшетах.

Аппаратные требования для запуска приложения:

- не менее 1 ГБ оперативной памяти;
- не менее 250 МБ свободного места в памяти устройства.

При первом запуске приложения на новом устройстве появится запрос от системы на разрешение доступа к геопозиции устройства. При запрете доступа часть функций приложения, связанная с получением местоположения, будет недоступна.

### 6.2 Руководство по использованию приложения

Для запуска приложения необходимо нажать на иконку приложения Kosoku на главном экране устройства (см. рисунок 6.1). При первом запуске приложения перед пользователем появится экран входа.

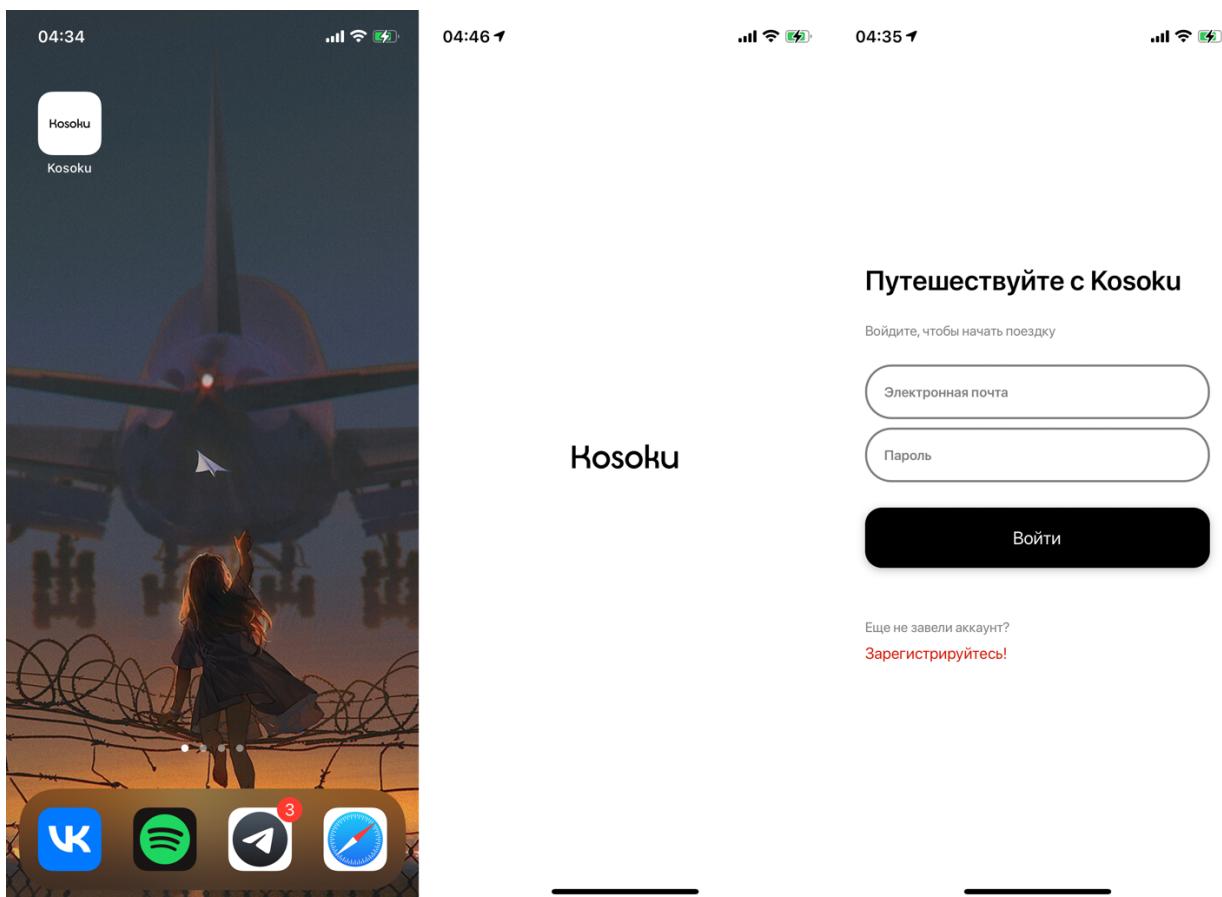


Рисунок 6.1 – Запуск приложения

Пользователь может ввести свои текущие авторотационные данные либо зарегистрировать новый аккаунт. Процесс регистрации изображен на рисунке 6.2.

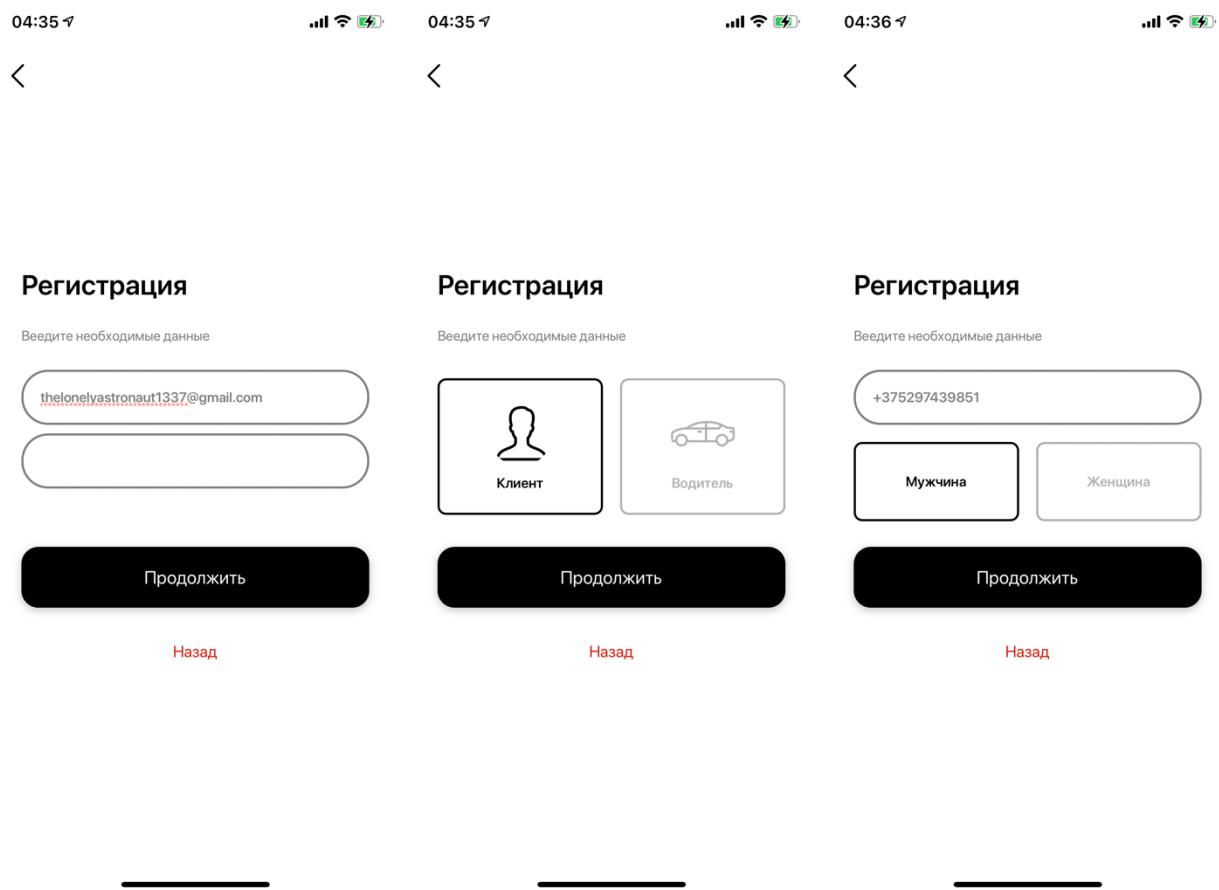


Рисунок 6.2 – Процесс регистрации

При регистрации пользователю требуется указать следующие поля:

7 имя;

8 фамилия;

9 электронная почта;

10 пароль;

11 тип аккаунта (клиент или водитель);

12 номер телефона, пол;

13 информация о машине (в случае, если был выбран водитель).

После успешного прохождения авторизации либо регистрации пользователь будет перенаправлен на главный экран. Экраны клиента и водителя изображены на рисунке 6.3.

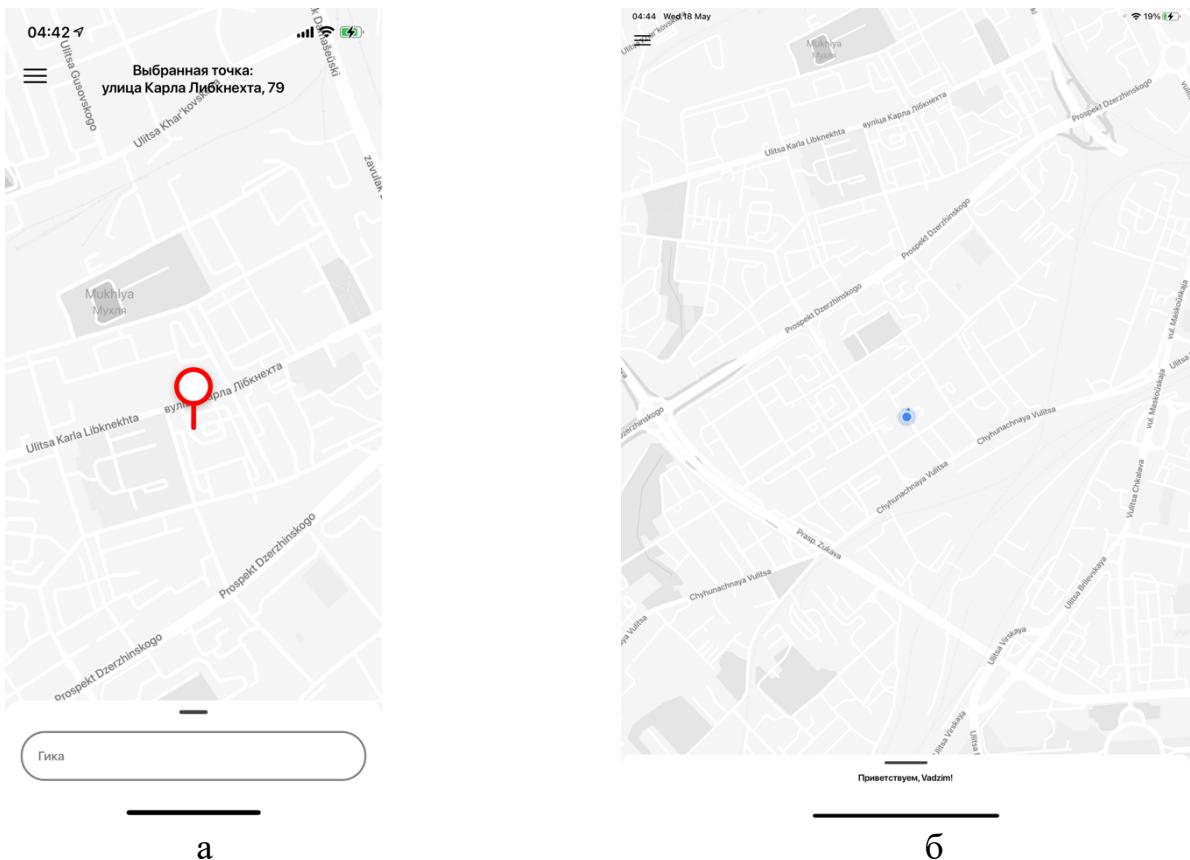


Рисунок 6.3 – Домашний экран (а – клиент, б – водитель)

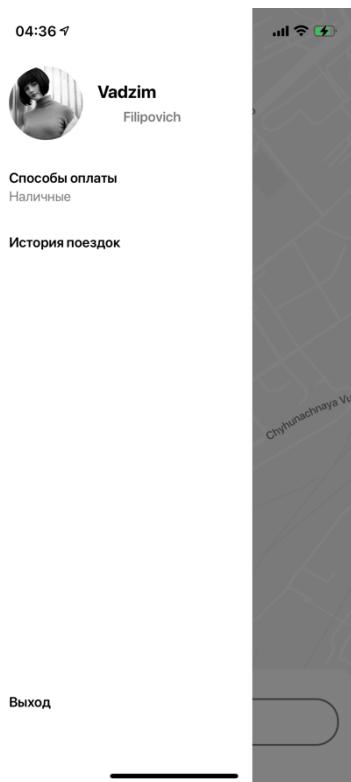


Рисунок 6.4 – Боковое меню

С данного экрана пользователь может начать процесс вызова такси (в случае, если пользователь – клиент сервиса) или открыть боковое меню для дальнейшей навигации по приложению. Боковое меню изображено на рисунке 6.4.

При нажатии на имя профиля в боковом меню пользователь будет перенаправлен на страницу, содержащую информацию о профиле. Пример данной страницы изображен на рисунке 6.5.

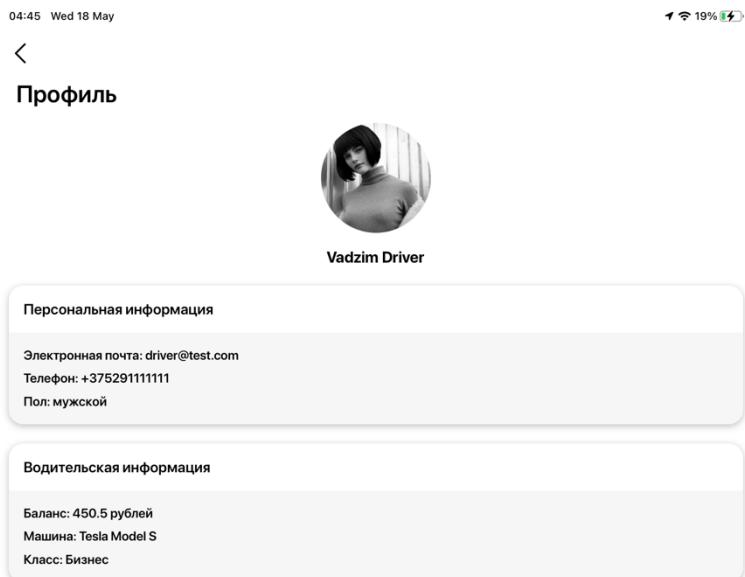


Рисунок 6.5 – Экран профиля

При нажатии на способ оплаты в боковом меню пользователь будет перенаправлен на экран способов оплаты. На данном экране расположена кнопка добавления карты, нажатие на которую приведет к навигации пользователя на страницу, содержащую форму по добавлению карты. Процесс добавления карты изображен на рисунке 6.6.

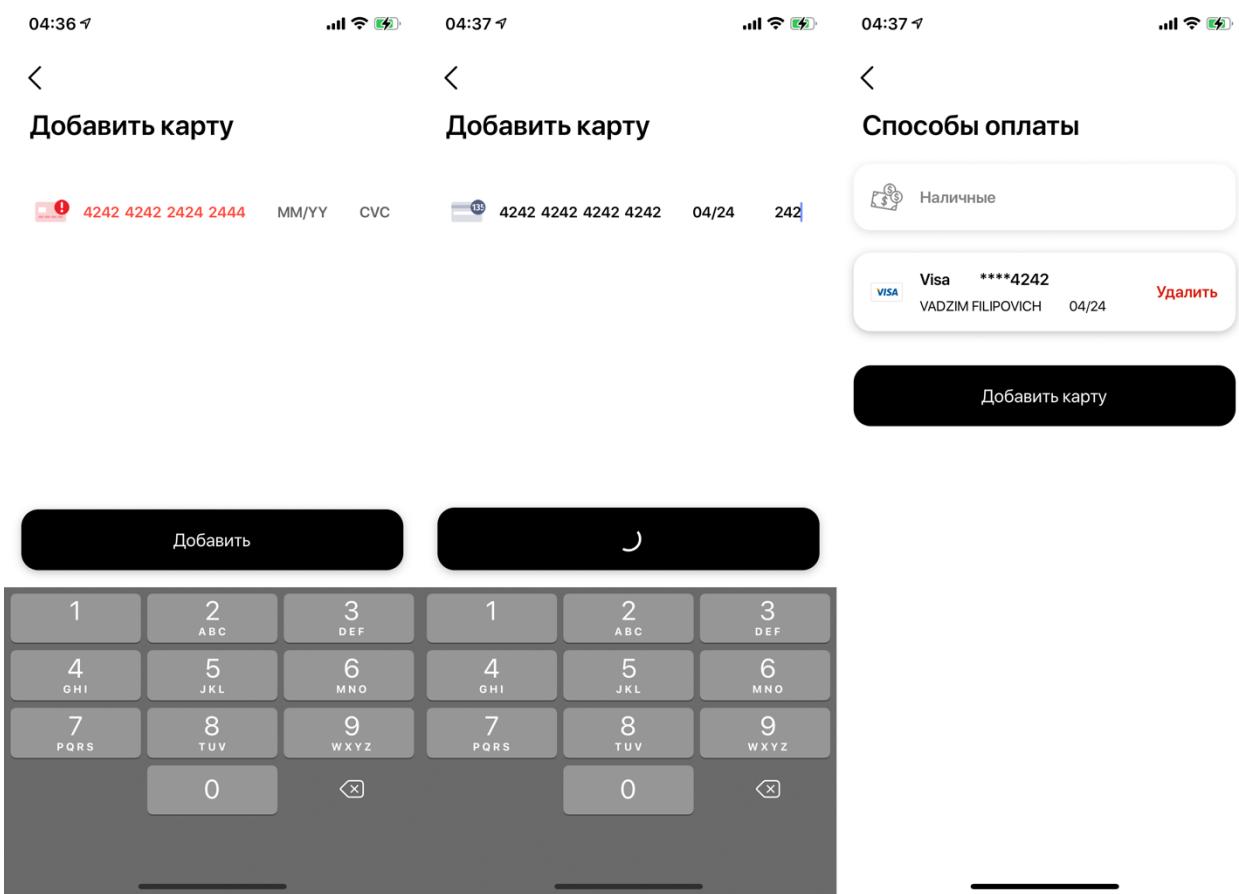


Рисунок 6.6 – Процесс добавления карты

При нажатии на историю поездок в боковом меню пользователь будет перенаправлен на экран со списком предыдущих поездок, изображенный на рисунке 6.7.

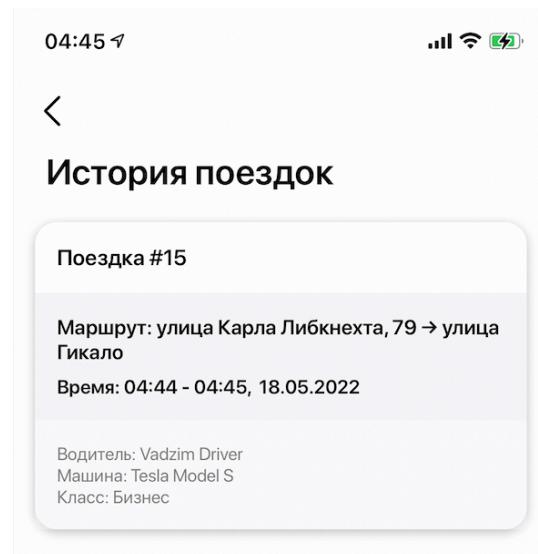


Рисунок 6.7 – История поездок

При нажатии на кнопку выхода в боковом меню текущая сессия будет завершена, и пользователь будет перенаправлен на экран авторизации.

На главном экране приложения содержится карта, а также маркер выбранной начальной позиции и ее описание в случае клиента, или же текущее местоположение в случае водителя.

Для клиента доступно выдвигающееся боковое меню, содержащее в себе два поля для ввода для точки отправления и точки назначения соответственно. Под данными полями отображается список предлагаемых адресов. В таком режиме приложение находится в состоянии выбора маршрута, пример данного состояния изображен на рисунке 6.8.

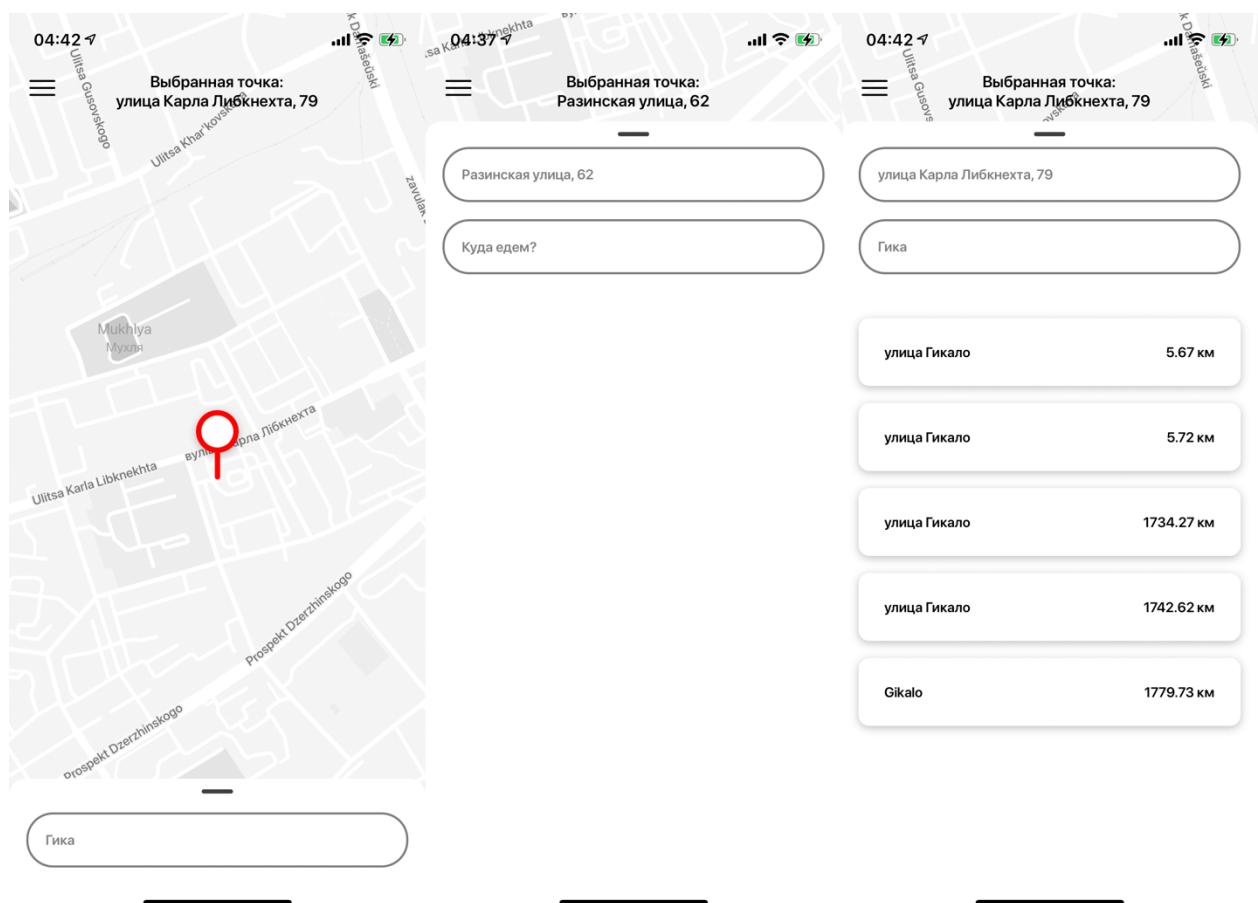


Рисунок 6.8 – Состояние выбора маршрута

При выборе пользователем двух точек, приложение переходит в состояние выбора класса машины. На экране смартфона отрисована карта, содержащая оптимальный маршрут между двумя точками, а также ряд доступных классов машин, включая стоимость поездки на каждом из них. Выбор класса делает кнопку начала поездки доступной для нажатия. При ее активации приложение переходит в режим поиска машины, в случае нажатия на кнопку отмены – возврат в состояние выбора маршрута. В случае успешного нахождения свободного водителя приложение переходит в

состояние поездки. Описанное состояние выбора класса изображено на рисунке 6.9.

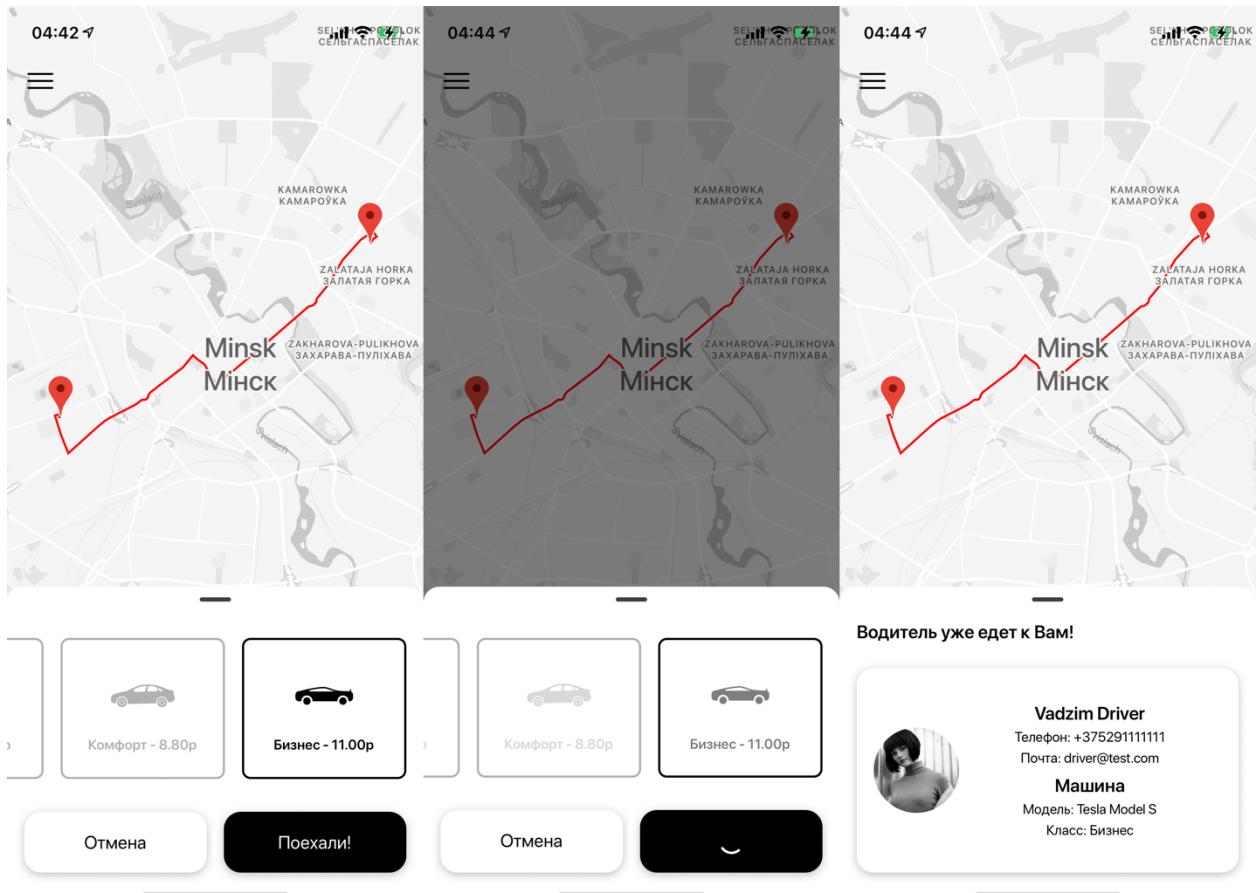


Рисунок 6.9 – Состояние выбора класса машины

При переходе приложения в состояние поездки на экране пользователя отображается информация о водителе, машине, его местоположении и статусе поездки. Если водитель едет к точке отправления, будет отображено сообщение “Водитель уже едет к вам”, если водитель едет к точке назначения – “Мы уже в пути”. При завершении поездки приложение возвращается к состоянию выбора маршрута и производится оплата. Описанное состояние изображено на рисунке 6.10.

Домашний экран водителя отличается от домашнего экрана пользователя, ключевое отличие – нет состояния выбора маршрута и выбора класса. При получении запроса о поездке с сервера приложение переходит в состояние запроса о рейсе. При отмене запроса экран вернется в состояние просмотра карты, в случае принятия приложение перейдет в состояние поездки. Пример состояния запроса о рейсе изображен на рисунке 6.11. Данный рисунок также демонстрирует пример интерфейса на планшетных устройствах.

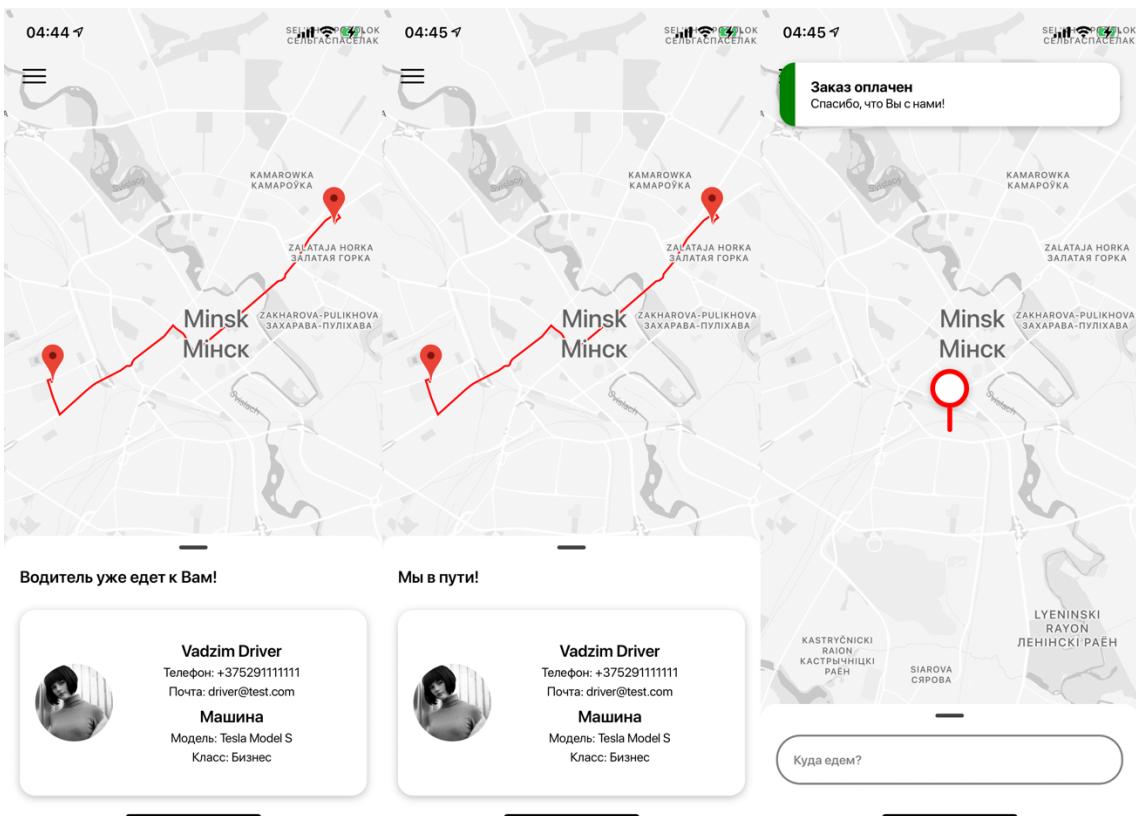


Рисунок 6.10 – Состояние поездки клиента

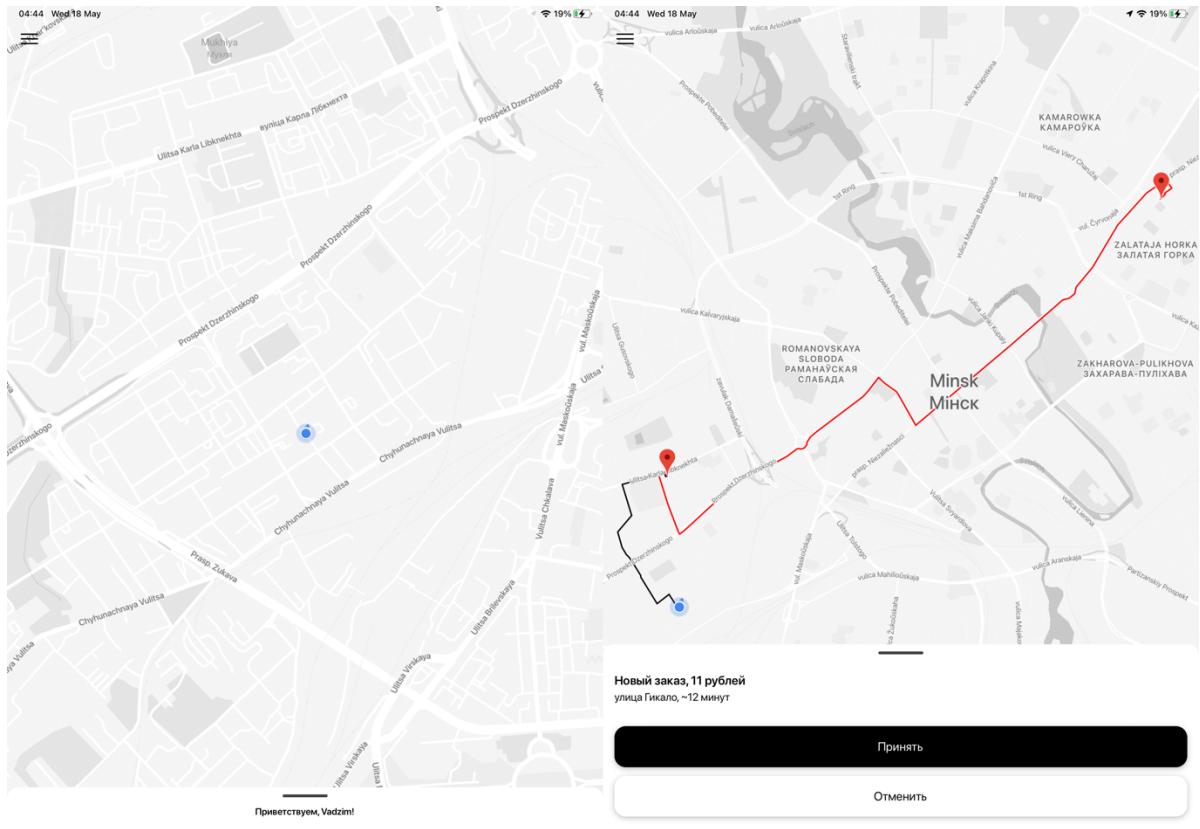


Рисунок 6.11 – Состояние запроса о рейсе

Состояние поездки водителя отличается от такового состояния клиента тем, что на карте черным маршрутом изображен путь до точки отправления, а также имеются кнопки начала и завершения поездки, нажатие на которые изменяет статус у клиента. Состояние поездки водителя изображено на рисунке 6.12.

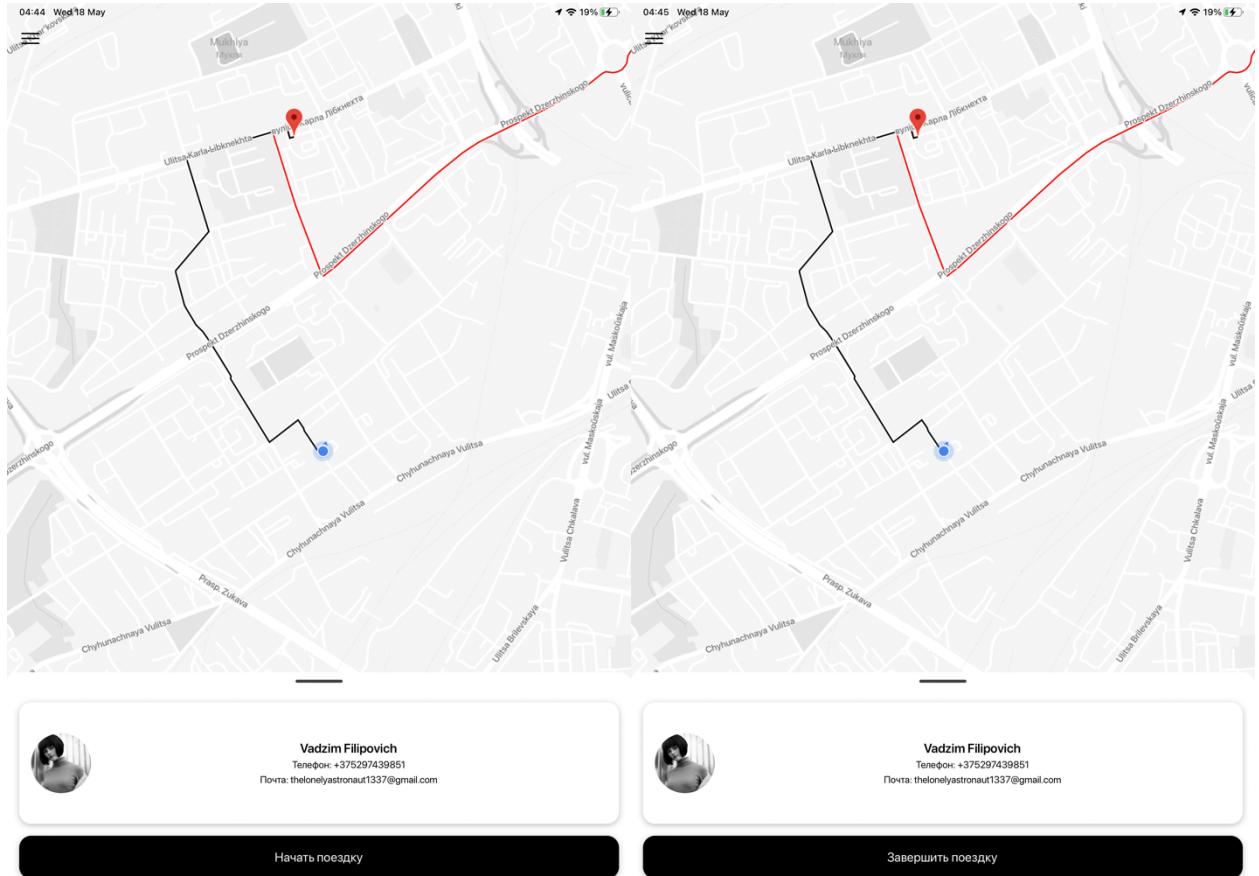


Рисунок 6.12 – Состояние поездки водителя

## **7 ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ ЭФФЕКТИВНОСТИ РАЗРАБОТКИ И РЕАЛИЗАЦИИ ЭЛЕКТРОННО-ИНФОРМАЦИОННОГО СЕРВИСА ПО ОКАЗАНИЮ УСЛУГ ПЕРЕВОЗКИ И ДОСТАВКИ**

### **7.1 Описание функций, назначения и потенциальных пользователей программного обеспечения**

Разрабатываемое программное обеспечение представляет собой систему, состоящую из мобильного приложения, предназначенного для вызова такси и оплаты поездки на устройствах под управлением операционных систем iOS и Android, и серверного приложения, предназначенного для обработки пользовательских запросов.

Разрабатываемая система позволит:

- выбирать маршрут и отслеживать прогресс поездки;
- оплачивать поездки банковской картой;
- просматривать историю предыдущих поездок.

В магазине приложений App Store представлен ряд аналогичных приложений. Основными преимуществами разрабатываемого мобильного приложения по сравнению с аналогами будут простота использования и кроссплатформенность разрабатываемой системы.

Разработка мобильного приложения осуществляется командой разработчиков по индивидуальному заказу. Причиной для разработки приложения является увеличение количества пользователей службы такси. Конечными пользователями данного приложения будут люди, имеющие мобильное устройство под управлением операционных систем iOS и Android, пользующиеся услугами такси.

### **7.2 Расчет затрат на разработку и цены электронно-информационного сервиса по оказанию услуг перевозки и доставки**

Расчет основной заработной платы участников команды осуществляется по формуле:

$$Z_o = K_{\text{пр}} \cdot \sum_{i=1}^n Z_{\text{ч}_i} \cdot t_i, \quad (7.1)$$

где  $n$  – количество исполнителей, занятых разработкой ПО;

$K_{\text{пр}}$  – коэффициент премий (1,6);

$Z_{\text{ч}_i}$  – часовая заработка плата  $i$ -го исполнителя (руб.);

$t_i$  – трудоемкость работ, выполняемых  $i$ -м исполнителем (ч.).

Данные о заработной плате команды разработчиков предоставлены компанией на 28.04.2022. Часовая заработка плата определяется путем

деления месячной заработной платы на количество рабочих часов в месяце. Количество рабочих часов в месяце принято равным 168 часам. Размер премии составляет 60% от размера основной заработной платы. Расчет затрат на основную заработную плату команды разработчиков представлен в таблице 7.1.

Таблица 7.1 – Расчет затрат на основную заработную плату команды разработчиков

№	Участник команды	Месячная заработная плата, р.	Часовая заработка плата, р.	Трудоемкость работ, ч.	Зарплата по тарифу, р.
1	Инженер-программист (разработчик мобильного приложения)	2000,00	11,90	120	1428,00
2	Инженер-программист (разработчик серверного приложения)	1400,00	8,33	300	2499,00
Премия (60%)					2356,20
Итого затраты на основную заработную плату разработчиков					6283,20

Затраты на дополнительную заработную плату команды разработчиков определяется по формуле:

$$Z_d = \frac{Z_o \cdot H_d}{100}, \quad (7.2)$$

где  $H_d$  – норматив дополнительной заработной платы (10%).

В результате вычисления по формуле (7.2) затраты на дополнительную заработную плату составят:

$$Z_d = \frac{6283,2 \cdot 10}{100} = 628,32 \text{ р.}$$

Отчисления в фонд социальной защиты населения и на обязательное страхование определяются в соответствии с действующими законодательными актами по формуле:

$$P_{\text{соц}} = \frac{(Z_o + Z_d) \cdot H_{\text{соц}}}{100}, \quad (7.3)$$

где  $H_{\text{соц}}$  – норматив отчислений в фонд социальной защиты населения и на обязательное страхование (34,6%).

В результате вычисления по формуле (7.3) отчисления на социальные нужды составят:

$$P_{\text{соц}} = \frac{(6283,2 + 628,32) \cdot 34,6}{100} = 2391,39 \text{ р.}$$

Прочие расходы включаются в себестоимость разработки ПО в процентах от затрат на основную заработную плату команды разработчиков по формуле:

$$P_{\text{пр}} = \frac{Z_o \cdot H_{\text{пз}}}{100}, \quad (7.4)$$

где  $H_{\text{пз}}$  – норматив прочих затрат (20%).

В результате вычисления по формуле (7.4) прочие затраты составят:

$$P_{\text{пр}} = \frac{6283,2 \cdot 20}{100} = 1256,64 \text{ р.}$$

Итого общая сумма затрат на разработку программного обеспечения вычисляется по формуле:

$$Z_p = Z_o + Z_d + P_{\text{соц}} + P_{\text{пр}}. \quad (7.5)$$

В результате вычисления по формуле (7.5) общая сумма затрат составит:

$$Z_p = 6283,20 + 628,32 + 2391,39 + 1256,64 = 10559,55 \text{ р.}$$

Плановая прибыль, которую требуется включить в итоговую стоимость программного обеспечения, рассчитывается по формуле:

$$\Pi_{\text{п.с}} = \frac{Z_p \cdot P_{\text{п.с}}}{100}, \quad (7.6)$$

где  $P_{\text{п.с}}$  – планируемая рентабельность затрат на разработку (25%).

В результате вычисления по формуле (7.6) плановая прибыль составит:

$$\Pi_{\text{п.с}} = \frac{10559,55 \cdot 25}{100} = 2639,89 \text{ р.}$$

Отпускная цена разработанного программного продукта вычисляется по формуле:

$$\Pi_{\text{п.с}} = Z_p + \Pi_{\text{п.с.}} \quad (7.7)$$

В результате вычисления по формуле (7.7) отпускная цена составит:

$$\Pi_{\text{п.с}} = 10559,55 + 2639,89 = 13199,44 \text{ р.}$$

Итоговые данные расчета цены программного обеспечения приведены в таблице 7.2.

Таблица 7.2 – Расчет цены ПО на основе затрат

№	Наименование статьи затрат	Формула/таблица для расчета	Значение, р.
1	Основная заработкая плата разработчиков	Таблица 7.1	6283,20
2	Дополнительная заработкая плата разработчиков	Формула 7.2	628,32
3	Отчисление на социальные нужды	Формула 7.3	2391,39
4	Прочие расходы	Формула 7.4	1256,64
5	Общая сумма затрат на разработку	Формула 7.5	10559,55
6	Плановая прибыль, включаемая в цену ПО	Формула 7.6	2639,89
7	Отпускная цена ПО	Формула 7.7	13199,44

### 7.3 Оценка экономического эффекта от продажи электронно-информационного сервиса по оказанию услуг перевозки и доставки

Разработка мобильного приложения осуществляется по индивидуальному заказу сторонней организации. В данном случае экономическим эффектом для организации-разработчика является чистая прибыль, полученная от реализации программного обеспечения заказчику. Цена программного продукта сформирована на основе затрат на разработку и установлена на уровне 15835,79 рублей.

Так как организация является плательщиком налога на прибыль, то экономическим эффектом от реализации ПО является чистая прибыль, которая рассчитывается по формуле:

$$\Delta\Pi_q = \Pi_{n.c} \cdot \left(1 - \frac{H_n}{100}\right), \quad (7.8)$$

где  $H_n$  – ставка налога на прибыль (18%).

В результате вычисления по формуле (7.8) чистая прибыль составит:

$$\Delta\Pi_q = 2639,89 \cdot \left(1 - \frac{18}{100}\right) = 2164,70\text{р.}$$

#### **7.4 Расчет показателей эффективности разработки программного обеспечения**

Для оценки экономической эффективности проекта необходимо сравнить чистую прибыль компании с общей суммой затрат на разработку.

Рентабельность затрат на разработку ПО с учетом налога на прибыль рассчитывается по формуле:

$$P_3 = \frac{\Delta\Pi_q}{Z_p} \cdot 100\%, \quad (7.9)$$

В результате вычисления по формуле (7.9) рентабельность затрат на разработку составит:

$$P_3 = \frac{2164,70}{10559,55} \cdot 100\% = 20,49\%.$$

В результате экономического обоснования эффективности разработки и реализации электронно-информационного сервиса по оказанию услуг перевозки и доставки были рассчитаны затраты на разработку ПО, чистая прибыль от продажи ПО заказчику и рентабельность инвестиций. Общая сумма затрат на разработку ПО составила 10559,55р. Чистая прибыль от реализации ПО заказчику составила 2164,70р. В качестве показателя эффективности инвестиций в разработку программного обеспечения была рассчитана рентабельность затрат на разработку равная 20,49%.

Так как рентабельность затрат равна 20,49%, то инвестирование средств в разработку данного программного продукта является экономически целесообразным.

## **ЗАКЛЮЧЕНИЕ**

Во время работы над дипломным проектом была изучена предметная область, рассмотрены существующие аналоги проектируемого приложения, выявлены их преимущества и недостатки. На этапе проектирования был определен список требований, которым должна соответствовать система.

В результате работы над дипломным проектом было разработано мобильное приложение для пользования услугами такси и ряд сервер к нему. Также были разработаны необходимые графические материалы для данного программного обеспечения. Разработанное мобильное приложение обладает рядом преимуществ и недостатков.

Основными преимуществами системы являются:

- кроссплатформенность всех модулей системы;
- возможность оплаты поездки наличным либо безналичным способом;
- возможность отслеживать текущее местоположение машины;
- гибкая и расширяемая архитектура;
- удобный пользовательский интерфейс.

Разработанная система обладает широкими возможностями для улучшения, например:

- чат между водителем и пользователем;
- поддержка мобильных систем оплаты (Apple Pay, Google Pay);
- регистрация в системе при помощи существующих аккаунтов в социальных сетях;
- система скидок и бонусов.

Целевой аудиторией приложения являются люди, имеющие мобильное устройство под управлением операционной системы iOS или Android, и желающие совершить поездку на частном автомобиле.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] In-App Purchase [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://developer.apple.com/in-app-purchase/> – Дата доступа: 12.04.2022.
- [2] Yandex.Go [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://apps.apple.com/ru/app/id472650686/> – Дата доступа: 12.04.2022
- [3] Uber [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://apps.apple.com/us/app/uber-request-a-ride/id368677368/> – Дата доступа: 12.04.2022.
- [4] Bolt [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://apps.apple.com/ee/app/bolt-fast-affordable-rides/id675033630/> – Дата доступа: 12.04.2022.
- [5] 135 [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://apps.apple.com/us/app/taxi-135/id768636008/> – Дата доступа: 12.04.2022.
- [6] Maxim [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://taximaxim.by/ru/about/> – Дата доступа: 12.04.2022.
- [7] Android SDK [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://developer.android.com/> – Дата доступа: 12.04.2022.
- [8] iOS SDK [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://developer.apple.com/> – Дата доступа: 12.04.2022.
- [9] React Native [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://reactnative.dev/> – Дата доступа: 12.04.2022.
- [10] Flutter [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://flutter.dev/> – Дата доступа: 12.04.2022.
- [11] Spring [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://spring.io/> – Дата доступа: 12.04.2022.
- [12] Nest [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://nestjs.com/> – Дата доступа: 12.04.2022.
- [13] PostgreSQL [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://www.postgresql.org/> – Дата доступа: 12.04.2022.

**ПРИЛОЖЕНИЕ А**  
*(обязательное)*

**Вводный плакат**

**ПРИЛОЖЕНИЕ Б**  
*(обязательное)*

**Схема структурная**

**ПРИЛОЖЕНИЕ В**  
*(обязательное)*

**Диаграмма классов и модулей**