

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: IntelliJ IDEA, 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 алгоритма.
- 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.
- 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: «Экономическое обоснование эффективности разработки и реализации 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

РЕФЕРАТ

Дипломный проект представлен следующим образом. Электронные носители: 1 компакт-диск. Чертежный материал: 6 листов формата А1. Пояснительная записка: 215 страниц, 17 рисунков, 3 таблицы, 13 литературных источников, 4 приложения.

Ключевые слова: мобильное приложение, вызов такси, доставка, перевозка.

Предметная область: предоставление услуг перевозки и доставки. Объект разработки: мобильное приложение для заказа такси и поддерживающее серверное приложение.

Целью данного дипломного проекта является разработка системы, которое позволит пользователям путешествовать по городу, используя частный транспорт. Данная система должна быть проста и удобна для использования, а также предоставлять весь необходимый функционал для оплаты.

Для разработки приложения были использованы интегрированные среды разработки IntelliJ IDEA, Xcode и Android Studio, язык программирования TypeScript, база данных PostgreSQL, фреймворки React Native и NestJS.

В результате работы над дипломным проектом была разработана система, предоставляющая возможности заказывать такси различных классов, просматривать маршрут поездки, отслеживать местоположение водителя и оплачивать поездки картами.

Данная система предназначена для людей, имеющих мобильное устройство под управлением операционной системы iOS или Android, и желающих совершить поездку на частном автомобиле.

По результатам технико-экономического обоснования эффективности разработки можно сделать вывод об экономической целесообразности разработки данного мобильного приложения.

Дипломный проект является успешно завершенным. Присутствует возможность усовершенствования и расширения функционала приложения. В рамках расширения функционала можно добавить возможность оплаты поездки через систему Apple Pay.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	10
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.....	28
2.2.9 Блок работы с сервисами Google	29
3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ.....	30
3.1 Мобильное приложение	30
3.1.1 Модуль пользовательского интерфейса	30
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	40
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	42
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.....	43
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	44
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	45
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	46
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.....	47
3.1.2.42 Функция listenForSetRouteLocation.....	48
3.1.3 Модуль работы с хранилищем	48
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
3.1.3.18 Тип ExtendedLocation	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	53
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	54
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	55
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	58
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	59
3.2.1 Модуль получения запроса клиента	59
3.2.1.1 Класс AuthController.....	60
3.2.1.2 Класс MapsController	60
3.2.1.3 Класс PaymentController.....	60
3.2.1.4 Класс UserController	60
3.2.2 Модуль авторизации	61
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	63
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	65
3.2.5.1 Класс PaymentService	65
3.2.5.2 Класс Stripe.....	66
3.2.6 Модуль подключения клиента	67
3.2.7 Модуль RTC	67
3.2.8 Модуль расчета стоимости и маршрута поездки	68
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.....	69
3.2.9.4 Класс Directions.....	70
4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ	71
4.1 Добавление платежной карты	71
4.2 Составление маршрута поездки	75
5 ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ	84
6 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ	89
6.1 Аппаратные и программные требования.....	89
6.2 Руководство по использованию приложения	89
7 ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ ЭФФЕКТИВНОСТИ РАЗРАБОТКИ И РЕАЛИЗАЦИИ ЭЛЕКТРОННО-ИНФОРМАЦИОННОГО СЕРВИСА ПО ОКАЗАНИЮ УСЛУГ ПЕРЕВОЗКИ И ДОСТАВКИ	97
7.1 Описание функций, назначения и потенциальных пользователей программного обеспечения.....	97
7.2 Расчет затрат на разработку и цены электронно-информационного сервиса по оказанию услуг перевозки и доставки.....	97
7.3 Оценка экономического эффекта от продажи электронно-информационного сервиса по оказанию услуг перевозки и доставки	100
7.3 Расчет показателей эффективности разработки программного обеспечения.....	101
ЗАКЛЮЧЕНИЕ	102
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	103
ПРИЛОЖЕНИЕ А	104
ПРИЛОЖЕНИЕ Б	162
ПРИЛОЖЕНИЕ В	163
ПРИЛОЖЕНИЕ Г	164

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.

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.

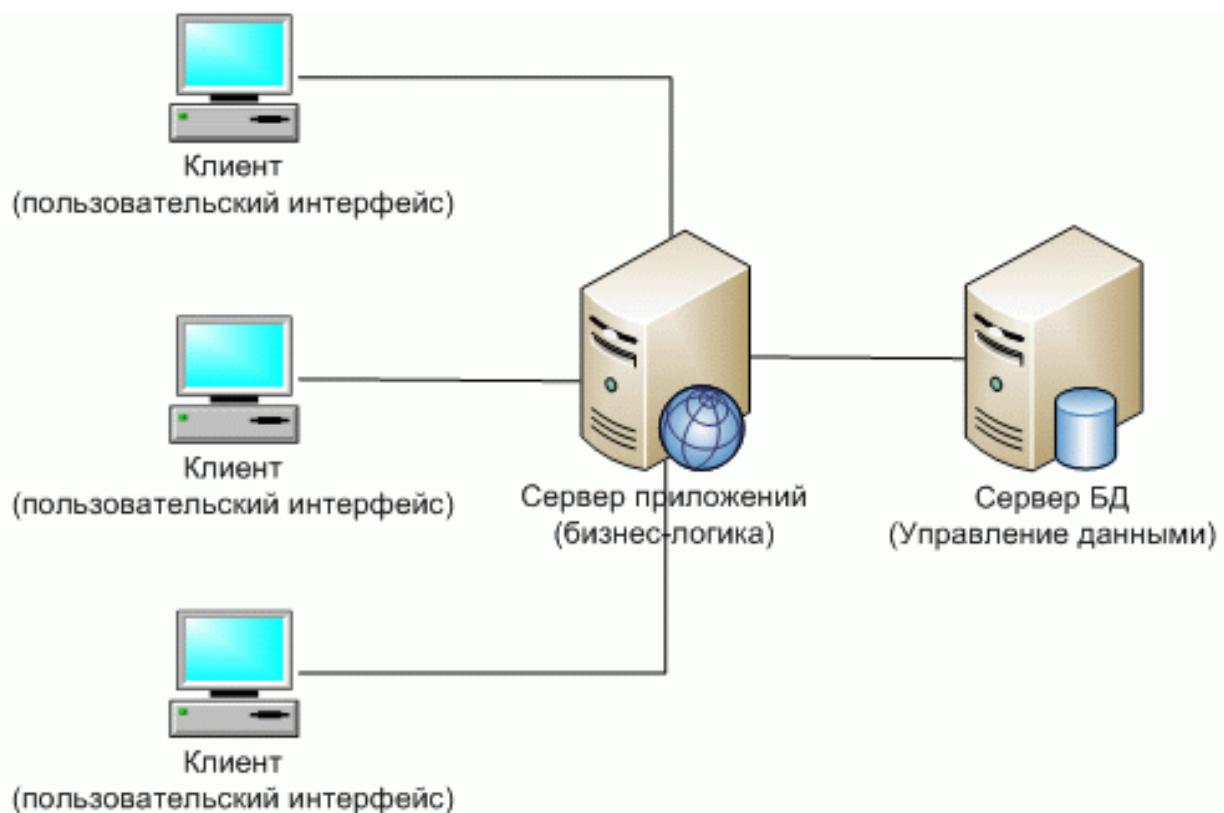


Рисунок 1.1 – Схема клиент-серверной архитектуры

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.

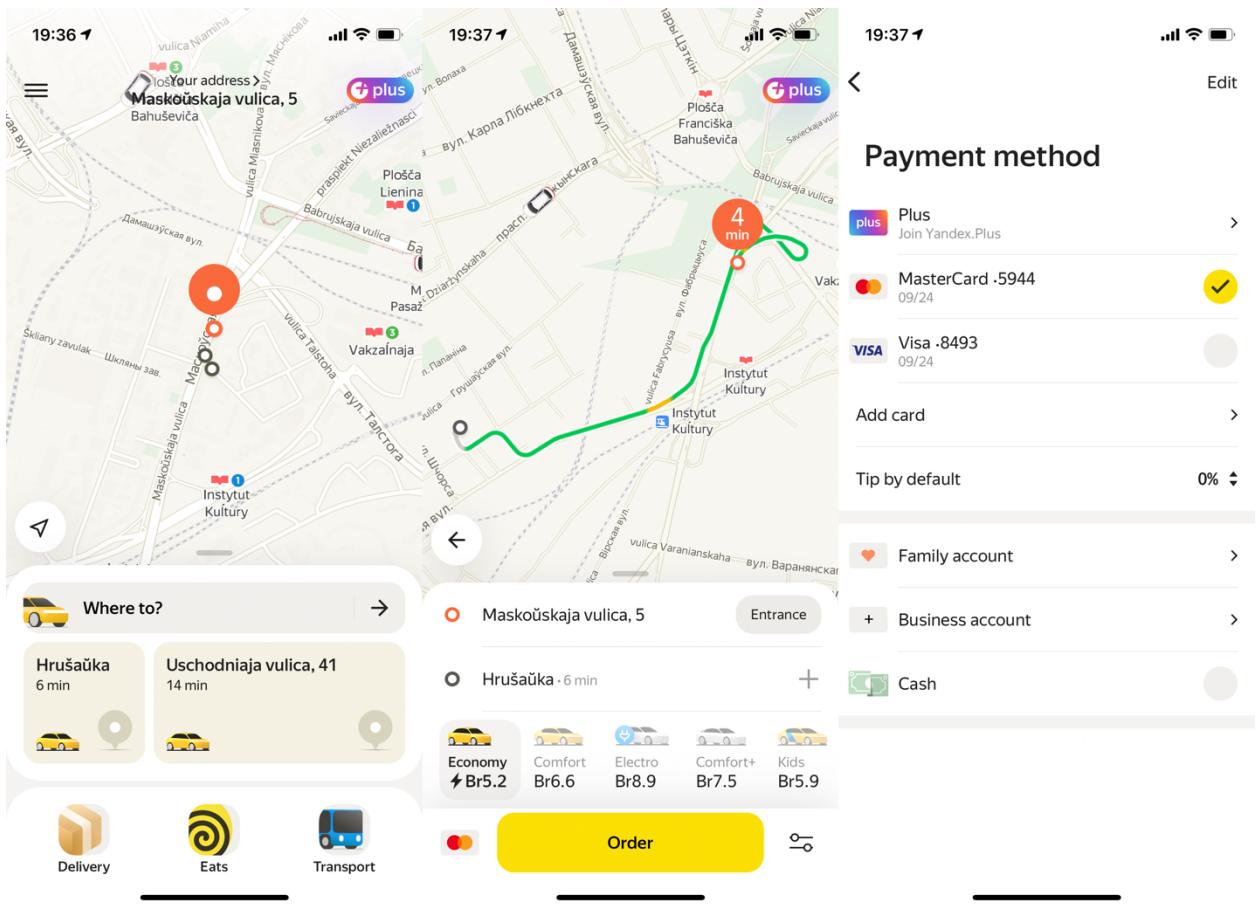


Рисунок 1.2 – Приложение Yandex.Go

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.

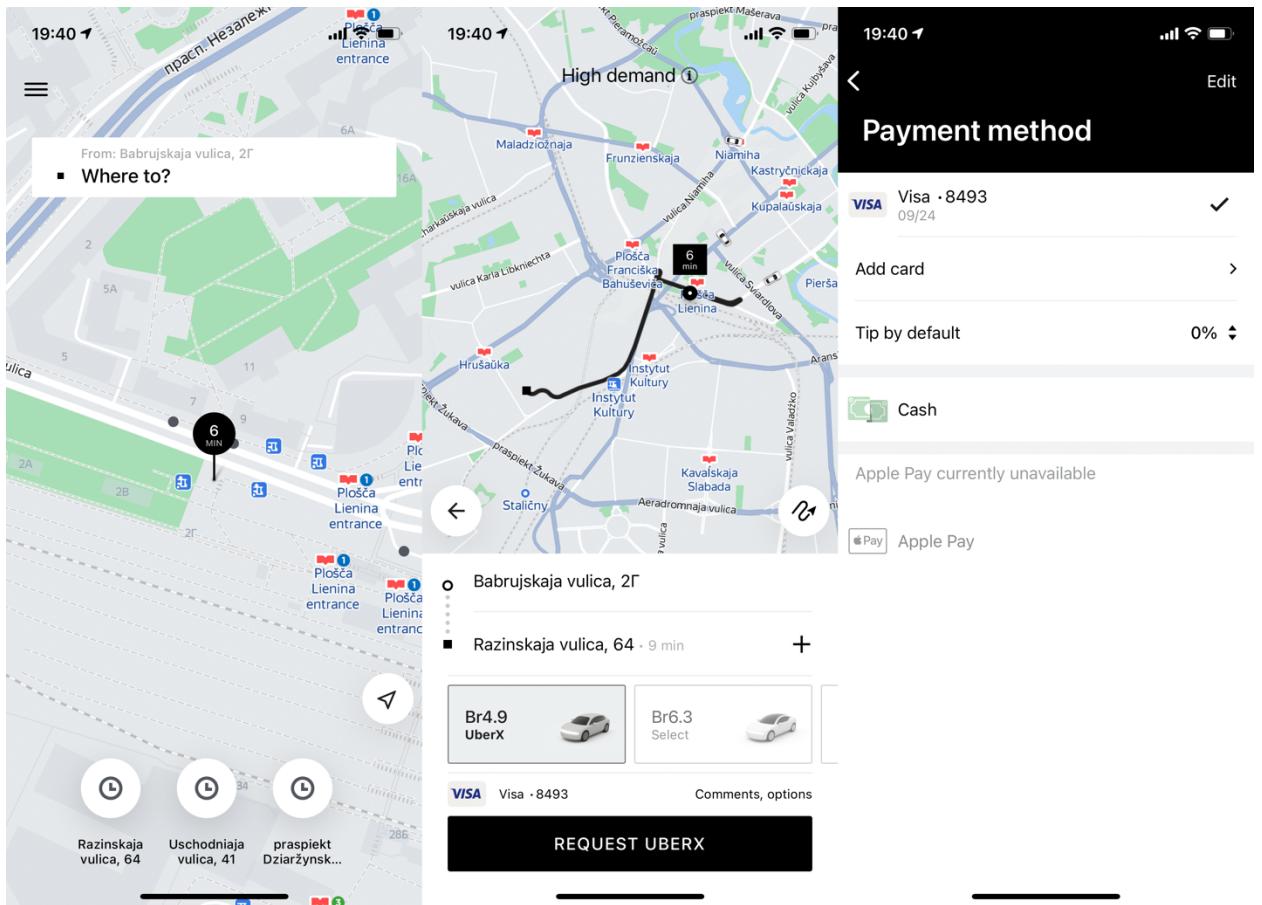


Рисунок 1.3 – Приложение Uber

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.

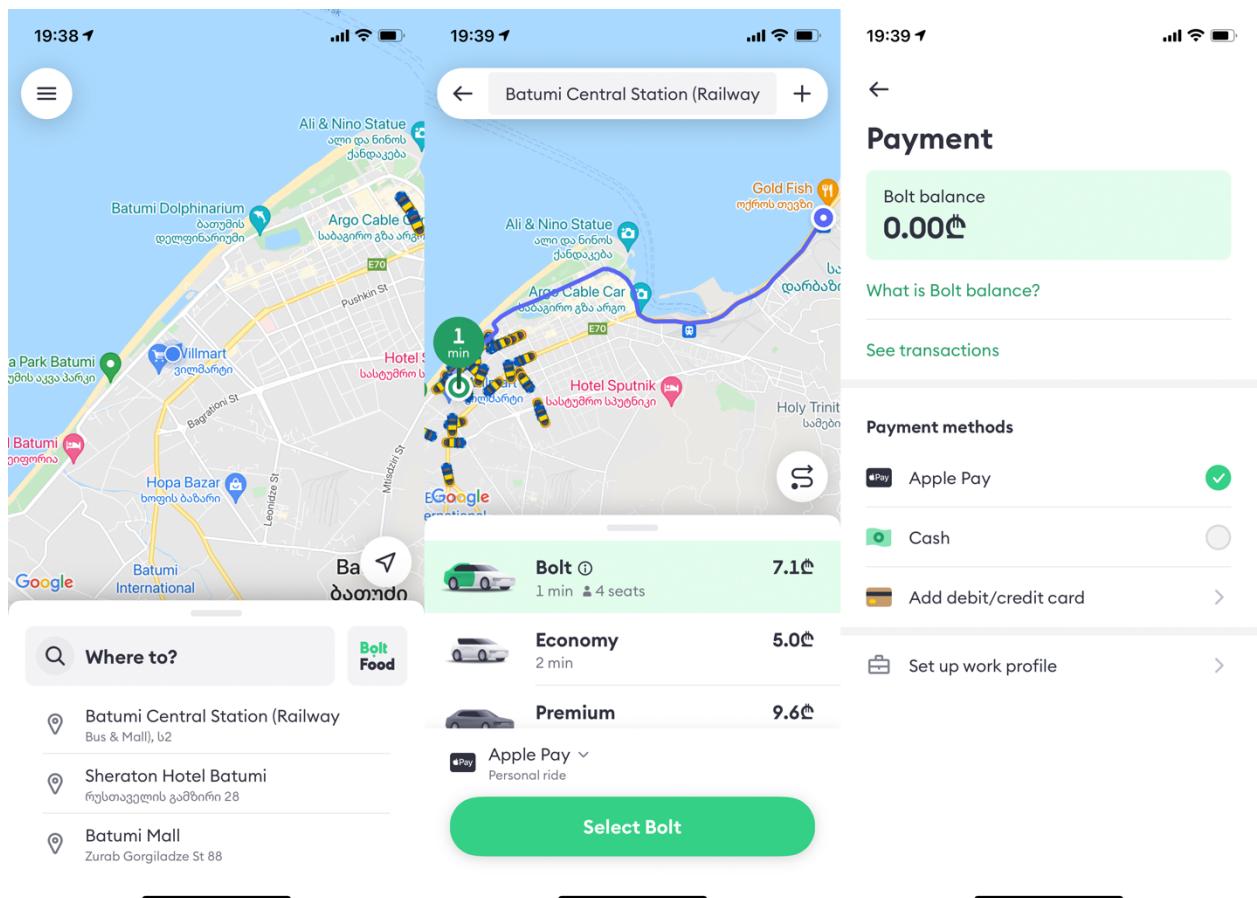


Рисунок 1.4 – Приложение Bolt

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.

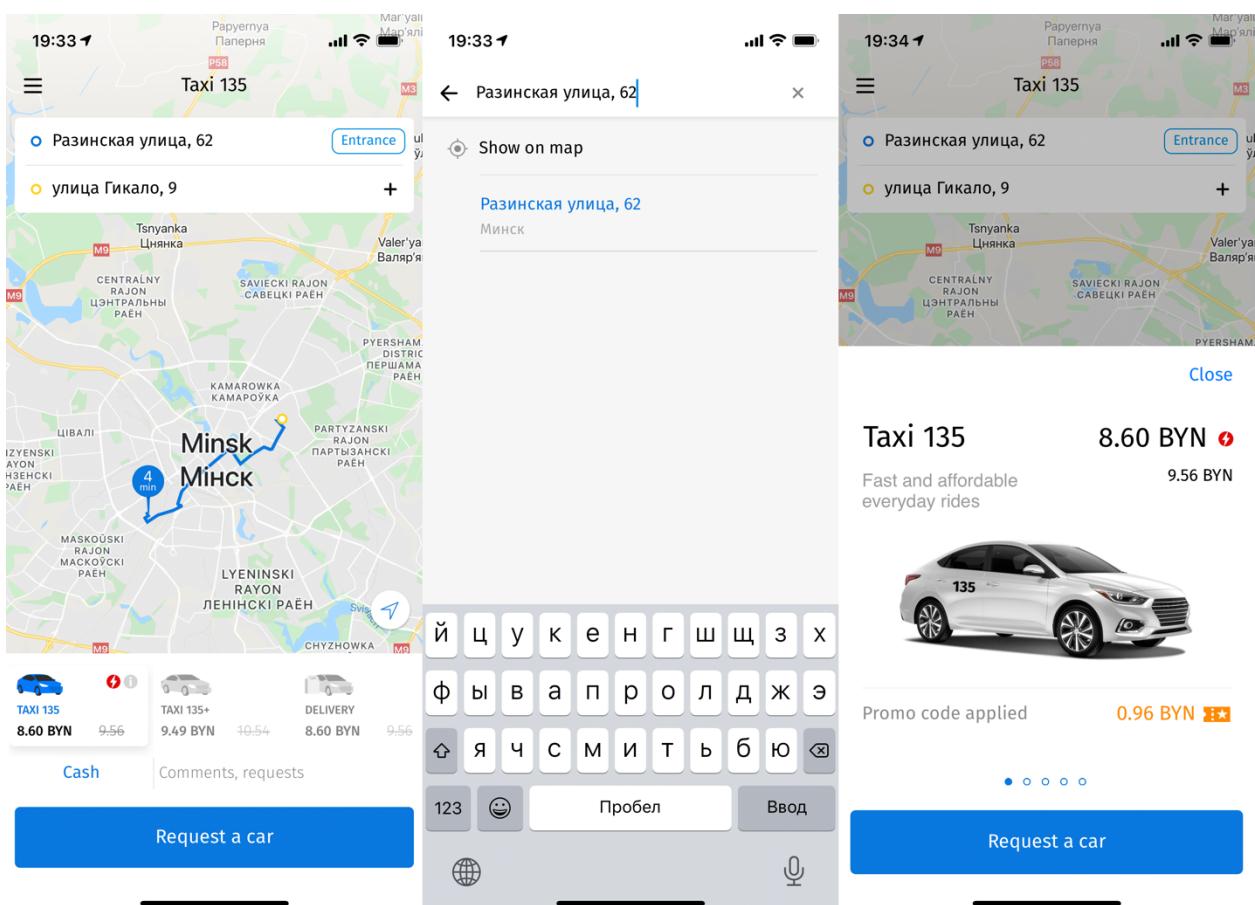


Рисунок 1.5 – Приложение 135

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.

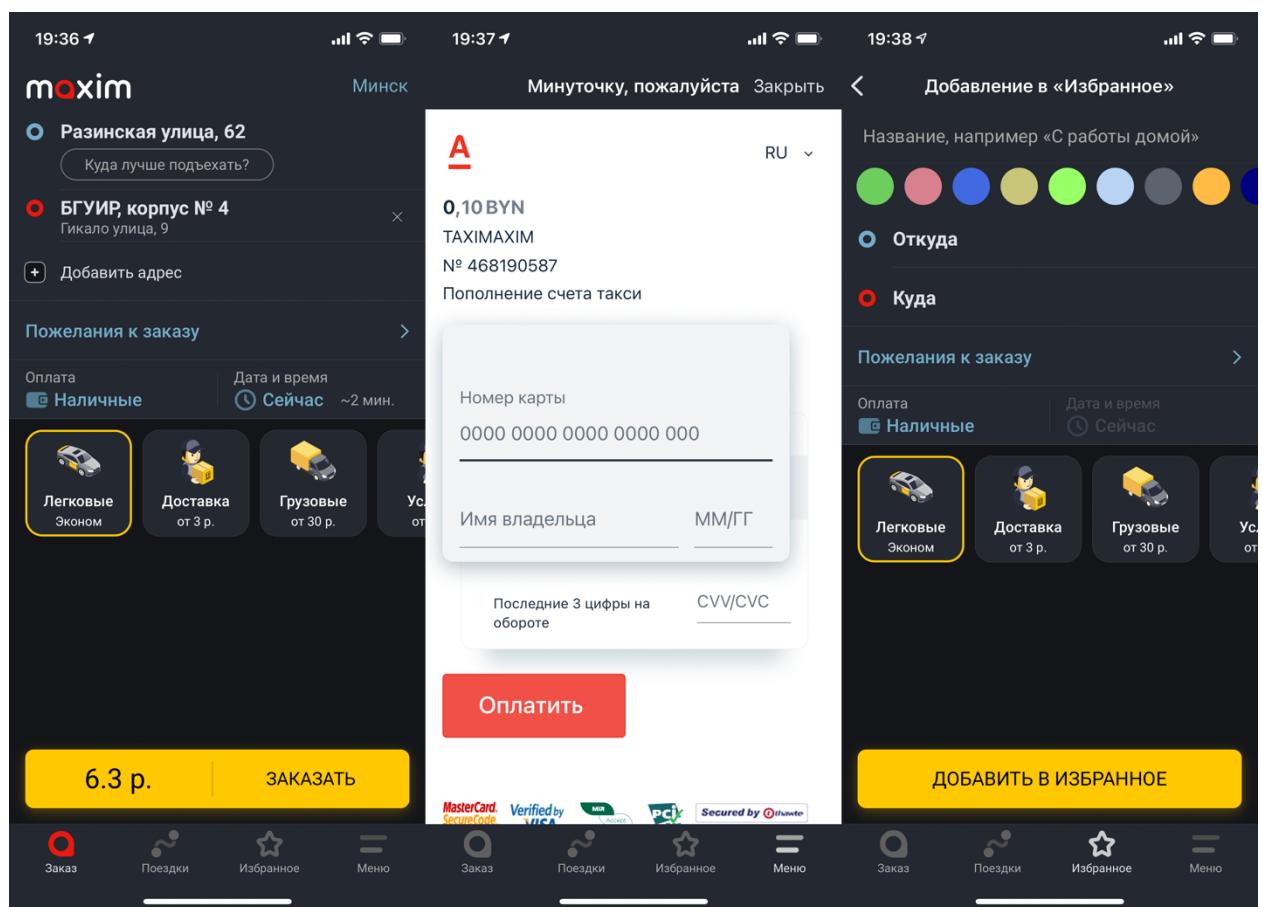


Рисунок 1.6 – Приложение Maxim

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 подписать 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 связность 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) отвечает за обмен сообщениями между пользователями и сбор данных об их текущей геопозиции. Данный модуль является ключевым в системе, так как позволяет отслеживать местоположение и производить поиск в режиме реального времени.

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 список доступных к заказу классов машин, кнопки вызова 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 `logoutSaga` производит действия, необходимые для выхода из аккаунта: отключение от сервера, очистка токенов из заголовков клиентов модуля работы с сервером, удаление авторизационных токенов из зашифрованных хранилищ операционных систем (`SecureSharedPref` / `Keychain`) и переход на экран входа.

3.1.2.6 Функция listenForLogout

Функция `listenForLogout` создает подписку функции `logoutSaga` 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 методов 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

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 `listenForRequestRide`

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.

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 или ответ от сервера) используются так называемые типы – аналог интерфейсов из Java и Kotlin 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, структура:

- `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 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 proximity;
- 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. Модель данных приведена на чертеже ГУИР.400201.107 РР.2.

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 из себя модель данных, общую для клиента и сервера, и описывает оплату поездки. Данная модель содержит в себе 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, например добавление или удаление платежного метода.

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

- driverDataHolder : Map<number, DriverSocketUserInfo> – структура данных, хранящая данные о статусе водителя по его идентификатору;
- tokenProvider : TokenProvider – экземпляр класса TokenProvider, предоставляющий возможность генерации авторизационных токенов;
- 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 – экземпляр класса Places из модуля работы с сервисами Google;
- directions : Directions – экземпляр класса Directions.

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`;
- `mockGeocoding : boolean` – флаг, показывающий необходимость подменить запрос на сервис тестовыми данными.

- geoUtils : GeoUtils – экземпляр класса GeoUtils;

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

В качестве дополнительного удобства клиентов в приложении по пользованию услугами такси должны существовать способы оплаты, отличные от наличного способа расчета за поездку, поэтому для оплаты поездок банковскими картами был разработан механизм регистрации данных карты пользователя в банковской платежной системе Stripe.

В качестве реализации механизма добавления карты разработаны методы `addCardSaga(action: Action)`, `createPaymentIntent(userId: number, data: CreatePaymentIntentInput)`, `addCard(userId: number, cardData: AddCardInput)`.

Для управления процессом с устройства пользователя используется функция `addCardSaga(action: Action)`. Рассмотрим ее алгоритм по шагам:

Параметром функции `addCardSaga(action: Action)` является:

- `action` – объект, хранящий в себе публичные данные карты, которые пользователь вписал в форму (последние четыре цифры, срок действия, бренд).

Для реализации функции `addCardSaga(action: Action)` использовались следующие входные данные:

- `paymentsAPI` – объект, предоставляющий возможность отправки запросов на платежные маршруты сервера;
- `confirmPayment` – функция, подтверждающая намерение оплаты пользователя через внутренние платежные шлюзы системы Stripe;
- `navigationService` – объект, позволяющий императивно управлять состоянием навигации приложения;
- `getPaymentMethodsSaga` – функция, получающая список доступных пользователю платежных методов;
- `toastService` – объект, позволяющий отображать уведомление на экране устройства;

Шаг 1. Создание запроса на проведение оплаты на минимальную сумму с автоматическим прохождением 3D-secure, то есть пропуск запроса на ввод одноразового СМС-пароля (при условии, что данная возможность предоставляется банком, обслуживающим карту).

```
const result: CreatePaymentIntentResponse = yield
call(paymentsAPI.createPaymentIntent, {
  bynAmount: 3,
```

```
    requestThreeDSecure: 'automatic',
}) ;
```

Для сохранения карты (то есть для получения возможности проведения оплаты автоматически без повторного ввода данных пользователем) в платежной системе используется механизм уникальных идентификаторов. Уникальный 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.

Шаг 2. Получение секретного ключа для подтверждения оплаты из ответа от сервера.

```
const secret = result.data.clientSecret;
```

Шаг 3. Проведение оплаты с использованием платежного шлюза Stripe, передавая тип платежного средства и уведомляя шлюз о намерении использовать карту в дальнейшем.

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

Шаг 4. Отправка уникального идентификатора карты на сервер для последующего переиспользования. На сервере сохраняется публичная информация о карте (последние четыре цифры, срок действия, бренд) и ее уникальный идентификатор в платежной системе Stripe.

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

Шаг 5. Выход на экран списка доступных платежных методов.

```
yield call(navigationService.goBack);
```

Шаг 6. Запуск функции обновления списка платежных методов.

```
yield call(getPaymentMethodsSaga);
```

Шаг 7. Отображение всплывающего уведомления об окончании операции.

```
yield call(toastService.showSuccess, 'Карта успешно добавлена', 'Желаем приятных поездок');
```

Шаг 8. Конец алгоритма.

При создании запроса на проведение оплаты используется функция `createPaymentIntent(userId: number, data: CreatePaymentIntentInput)`, реализованная в серверном приложении системы. Рассмотрим ее алгоритм по шагам:

Параметрами функции `createPaymentIntent(userId: number, data: CreatePaymentIntentInput)` являются:

- `userId` – уникальный идентификатор пользователя;
- `data` – переданный пользователем объект, хранящий в себе размер платежа и способ проведения авторизации 3D-secure.

Для реализации функции `createPaymentIntent(userId: number, data: CreatePaymentIntentInput)` использовались следующие входные данные:

- `userRepository` – объект, позволяющий взаимодействовать с таблицей пользователей подключенной базы данных;
- `stripe` – объект, являющийся абстракцией над Stripe SDK, позволяющий осуществлять запросы на платежные шлюзы данной платежной системы.

Шаг 1. Поиск в базе данных пользователя, совершающего запрос.

```
const user = await this.userRepository.findOneOrFail({ id: userId });
```

Шаг 2. Создание запроса на проведение оплаты.

```
const result = await this.stripe.createIntent(  
    user.email,  
    data.bynAmount,  
    data.requestThreeDSecure,  
    user.stripeClientId  
) ;
```

Шаг 3. Сохранение платежного идентификатора пользователя для последующих оплат сохраненной картой.

```
user.stripeClientId = result.customerId;  
  
await this.userRepository.save(user);
```

Шаг 4. Возврат результата клиенту.

```
return {  
    clientSecret: result.intent.client_secret,  
};
```

Шаг 5. Конец алгоритма.

При добавлении платежной карты в базу данных используется функция `addCard(userId: number, cardData: AddCardInput)`, реализованная в серверном приложении системы. Рассмотрим ее алгоритм по шагам:

Параметрами функции `addCard(userId: number, cardData: AddCardInput)` являются:

- `userId` – уникальный идентификатор пользователя;
- `cardData` – переданный пользователем объект, хранящий в себе публичные данные карты и уникальный идентификатор платежного средства в системе Stripe.

Для реализации функции `addCard(userId: number, cardData: AddCardInput)` использовались следующие входные данные:

- `userRepository` – объект, позволяющий взаимодействовать с таблицей пользователей подключенной базы данных;
- `PaymentMethodDetails` – класс, являющийся моделью представления информации о карте в базе данных;
- `paymentMethodRepository` – объект, позволяющий взаимодействовать с таблицей платежных методов подключенной базы данных;
- `paymentMethodDetailsRepository` – объект, позволяющий взаимодействовать с таблицей данных о платежных картах подключенной базы данных.

Шаг 1. Поиск в базе данных пользователя, совершающего запрос.

```
const user = await this.userRepository.findOneOrFail({ id:  
    userId });
```

Шаг 2. Создание и сохранение в базу данных информации о карте.

```
const details = new PaymentMethodDetails();  
details.exp = cardData.exp;  
details.stripePaymentId = cardData.stripePaymentId;  
details.brand = cardData.brand;  
details.holder = cardData.holder;
```

```
details.lastFour = cardData.lastFour;
await this.paymentMethodDetailsRepository.save(details);
```

Шаг 3. Создание и сохранение в базу данных платежного средства.

```
const card = new PaymentMethod();
card.isDefault = type === card.Cash;
card.type = type;
card.user = user;
card.details = details;
await this.paymentMethodRepository.save(card);
```

Шаг 4. Конец алгоритма.

4.2 Составление маршрута поездки

Для указания системе требований к поиску водителя для предстоящей поездки, необходимо дать пользователю возможность выбрать точки назначения, отправления и класс машины для поездки.

В подразделе описаны алгоритмы, представленные на чертежах ГУИР.400201.107 ПД1 и ГУИР.400201.027 РР.3.

В качестве реализации механизма добавления карты были разработаны методы `fetchPlacesSaga(action: Action)`, `setChosenLocationSaga(action: Action)`, `prepareRideDataSaga(action: Action)`, `decode(location: Location)`, `search(part: string)`, `getDirection(from: Location, to: Location)`.

При поиске точки назначения или отправления с помощью текстового ввода используется функция `fetchPlacesSaga(action: Action)`. Рассмотрим ее алгоритм по шагам:

Параметром функции `fetchPlacesSaga(action: Action)` является:

- `action` – объект, хранящий в себе точку поиска (отправление или назначение) и часть названия или адреса места, которое находится в данной точке.

Для реализации функции `fetchPlacesSaga(action: Action)` использовались следующие входные данные:

- `toastService` – объект, позволяющий отображать уведомление на экране устройства.

- `homeAPI` – объект, предоставляющий возможность отправки запросов на маршруты сервера, выполняющие операции с географической картой;

Шаг 1. Подписка на событие о поиске места по названию или адресу. Учитывая то, что пользователь может менять текст в строке поиска очень быстро и события о смене текста будут генерироваться раз в несколько миллисекунд, была проведена оптимизация с использованием функции debounce: функция fetchPlacesSaga будет вызвана лишь в том случае, если за последние 300 миллисекунд не было сгенерировано нового события о смене текста в поле для ввода.

```
yield debounce(300, FETCH_PLACES.TRIGGER, fetchPlacesSaga);
```

Шаг 2. Проверка на наличие необходимого количества символов.

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

Для оптимизации запросов на сервер будет производиться отправка строки лишь в том случае, если ее длина составляет три и более символа, иначе результат устанавливается пустым и происходит выход из функции.

Шаг 3. Создание запроса на получение списка возможных мест и адресов.

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

Шаг 4. Обработка ответа. При успешном ответе сервера тело ответа будет содержать массив возможных адресов.

```
yield put(FETCH_PLACES.COMPLETED({ results: result.data,  
direction: action.payload.direction }));
```

При возникновении ошибки на экране пользователя будет отображено текстовое уведомление.

```
yield call(toastService.showError, result.error);
```

Шаг 5. Конец алгоритма.

При поиске точки отправления с помощью географической карты используется функция setChosenLocationSaga(action: Action). Рассмотрим ее алгоритм по шагам:

Параметром функции `setChosenLocationSaga(action: Action)` является:

- `action` – объект, хранящий в себе выбранное положение на карте (широта и долгота выбранной точки).

Для реализации функции `setChosenLocationSaga(action: Action)` использовались следующие входные данные:

- `homeAPI` – объект, предоставляющий возможность отправки запросов на маршруты сервера, выполняющие операции с географической картой;

- `getIsPointerMoving` – функция, возвращающая состояние указателя на карте (находится ли он в движении или нет);

- `toastService` – объект, позволяющий отображать уведомление на экране устройства.

Шаг 1. Подписка на событие о перемещении карты.

```
yield debounce(500, SET_CHOSEN_LOCATION.TRIGGER,  
setChosenLocationSaga);
```

Шаг 2. Проверка нахождения в состоянии движения, генерировать новый запрос на сервер нецелесообразно.

```
const isMoving = yield select(getIsPointerMoving);  
  
if (isMoving) return;
```

Шаг 3. Создание запроса на получение списка адреса или места, имеющего переданные географические координаты.

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

Шаг 4. Обработка ответа. При успешном ответе сервера тело ответа будет содержать текстовое описание выбранной точки.

```
yield put(SET_CHOSEN_LOCATION.COMPLETED(result.data));
```

При возникновении ошибки на экране пользователя будет отображено текстовое уведомление.

```
yield call(toastService.showError, result.error);
```

Шаг 5. Конец алгоритма.

После выбора обеих точек поездки происходит переход приложение в состояние выбора класса машины и отображения маршрута, для этого используется функция `prepareRideDataSaga(action: Action)`. Рассмотрим ее алгоритм по шагам:

Параметром функции `prepareRideDataSaga(action: Action)` является:

- `action` – объект, хранящий в себе точки отправления и назначения.

Для реализации функции `prepareRideDataSaga(action: Action)` использовались следующие входные данные:

- `homeAPI` – объект, предоставляющий возможность отправки запросов на маршруты сервера, выполняющие операции с географической картой;
- `toastService` – объект, позволяющий отображать уведомление на экране устройства;
- `mapService` – объект, позволяющий императивно управлять различными свойствами карты, изображенной на главном экране;
- `bottomSheetService` – объект, позволяющий императивно управлять различными свойствами нижнего меню, изображенного на главном экране.

Шаг 1. Установка камеры карты в положение, в котором был бы виден весь маршрут от точки отправления до точки назначения.

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

Шаг 2. Установка нижнего меню в минимизированное положение, то есть в такое, при котором будет видно минимально необходимое количество информации.

```
yield call(bottomSheetService.minimize);
```

Шаг 3. Создание запроса на получение маршрута поездки и стоимости доступных классов машин.

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

Шаг 4. Обработка ответа. При успешном ответе сервера тело ответа будет содержать массив географических координат, составляющих ломанную линию маршрута на карте, и список различных доступных классов машин.

```
yield put(SET_RIDE_REQUEST(result.data));
```

При возникновении ошибки на экране пользователя будет отображено текстовое уведомление.

```
yield call(toastService.showError, result.error);
```

Шаг 5. Конец алгоритма.

При обработке запроса пользователя на получение текстового описания места по географическим координатам используется функция `decode(location: Location)`, реализованная в серверном приложении системы. Рассмотрим ее алгоритм по шагам:

Параметром функции `decode(location: Location)` является:

- `data` – объект, хранящий в себе широту и долготу точки, информацию о которой требуется вычислить.

Для реализации функции `decode(location: Location)` использовались следующие входные данные:

- `maps` – объект, являющийся оберткой над различными сервисами предоставления услуг картографии. По умолчанию – Google Maps API, возможно использовать альтернативу в виде GraphHopper API;
- `geoUtils` – объект, являющийся вспомогательным и используется для проведения различных вычислительных действий с географическими координатами;
- `places` – массив тестовых мест и их адресов.

Шаг 1. Проверка на включенный режим подмены данных от сторонних сервисов. Данный режим можно использовать в режиме разработки для избежания совершения запросов на реальные сервисы.

```
if (this.mockGeocoding) {  
    ...  
}
```

Шаг 2. При включенном режиме подмены данных (режим разработки) происходит поиск существующей в массиве `places` точки, которая удалена от запрашиваемой не более чем на километр, этой точности достаточно для разработчика.

```
const data = places.find((place) =>  
    this.geoUtils.getDistance(  
        location.latitude,  
        location.longitude,  
        place.latitude,  
        place.longitude,  
        'K'  
    ) < 1;  
);
```

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 Эвклидовой и уравнения расстояния также принимают другую форму. В Эвклидовой 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 при неправильном выборе могут привести к значительным искажениям длин в силу особенностей проекционных искажений.

Шаг 3. При нахождении данной точки среди тестовых она будет возвращена, в противном случае будет сгенерирована заглушка.

```
if (data) return data;

return {
  latitude: location.latitude,
  longitude: location.longitude,
  readableLocation: `Заглушка #${(Math.random() *
100).toFixed()}`,
};
```

Шаг 4. В реальном режиме работы поиск данных о переданной в функцию точке будет происходить с использованием сторонних картографических сервисов (Google Maps / GraphHopper).

```
const decoded = await this.maps.client.reverseGeocode({
  params: {
    key: this.maps.mapsKey,
    latlng: location,
    language: Language.ru,
  },
});
```

Шаг 5. Преобразование данных в другой формат и возвращение ответа пользователю.

```

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

```

Шаг 6. Конец алгоритма.

При обработке запроса пользователя на получение возможных мест и адресов по части их названия используется функция `search(part: string)`, реализованная в серверном приложении системы. Рассмотрим ее алгоритм по шагам:

Параметром функции `search(part: string)` является:

- `data` – объект, хранящий в себе широту и долготу точки, информацию о которой требуется вычислить.

Для реализации функции `search(part: string)` использовались следующие входные данные:

- `maps` – объект, являющийся оберткой над различными сервисами предоставления услуг картографии. По умолчанию – Google Maps API, возможно использовать альтернативу в виде GraphHopper API.

Шаг 1. Проверка на включенный режим подмены данных от сторонних сервисов. Данный режим можно использовать в режиме разработки для избежания совершения запросов на реальные сервисы.

```

if (this.mockGeocoding) {
    ...
}

```

Шаг 2. При включенном режиме подмены данных (режим разработки) происходит поиск и возврат пользователю существующих в массиве `places` точек, описание которых включало бы в себя запрашиваемую пользователем строку.

```

return places.filter((place) =>
    place.readableLocation.toLowerCase().includes(part.toLowerCase()));

```

Шаг 3. В реальном режиме работы поиск точек будет происходить с использованием сторонних картографических сервисов (Google Maps / GraphHopper).

```

const result = await this.maps.client.placeAutocomplete({
    params: {

```

```

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

```

Шаг 4. Преобразование формата данных и их возврат пользователю.

```

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

```

Шаг 5. Конец алгоритма.

При обработке запроса пользователя на получение маршрута поездки и доступных классов автомобилей используется функция `getDirection(from: Location, to: Location)`, реализованная в серверном приложении системы. Рассмотрим ее алгоритм по шагам:

Параметром функции `getDirection(from: Location, to: Location)` является:

- `from` – широта и долгота точки отправления;
- `to` – широта и долгота точки назначения.

Для реализации функции `getDirection(from: Location, to: Location)` использовались следующие входные данные:

– `maps` – объект, являющийся оберткой над различными сервисами предоставления услуг картографии. По умолчанию – Google Maps API, возможно использовать альтернативу в виде GraphHopper API.

Шаг 1. Выполнение запроса подсчета пути на картографические API (Google Maps / GraphHopper).

```

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

```

Шаг 2. Извлечение необходимых данных из ответа API и их преобразование в необходимый для клиента вид.

```
const data = {
    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,
    })),
};
```

Шаг 3. Просчет стоимости поминутной стоимости классов с учетом посадки и возврат данных пользователю.

```
const time = Math.ceil(data.minutes);

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

Шаг 4. Конец алгоритма.

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. 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 (авторизация, 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.

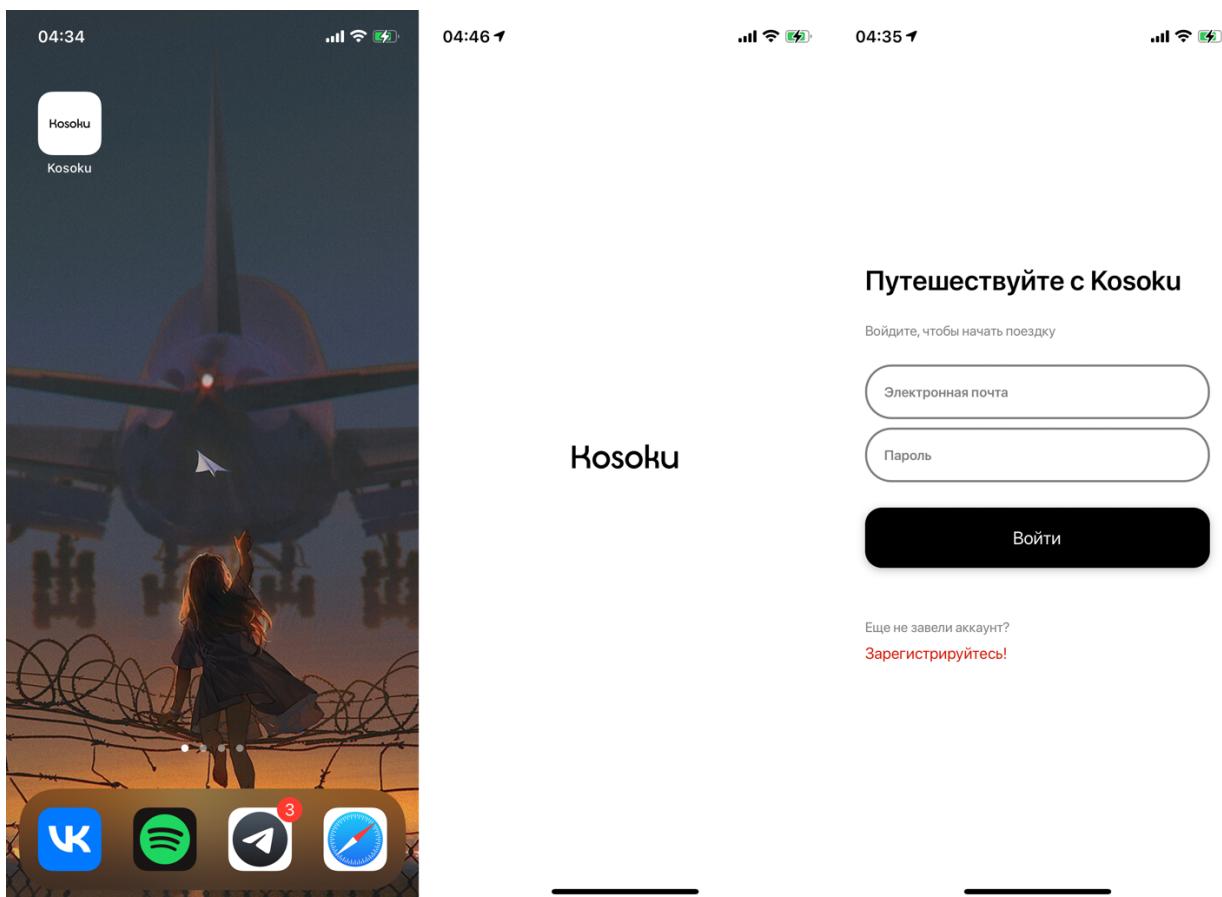


Рисунок 6.1 – Запуск приложения

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

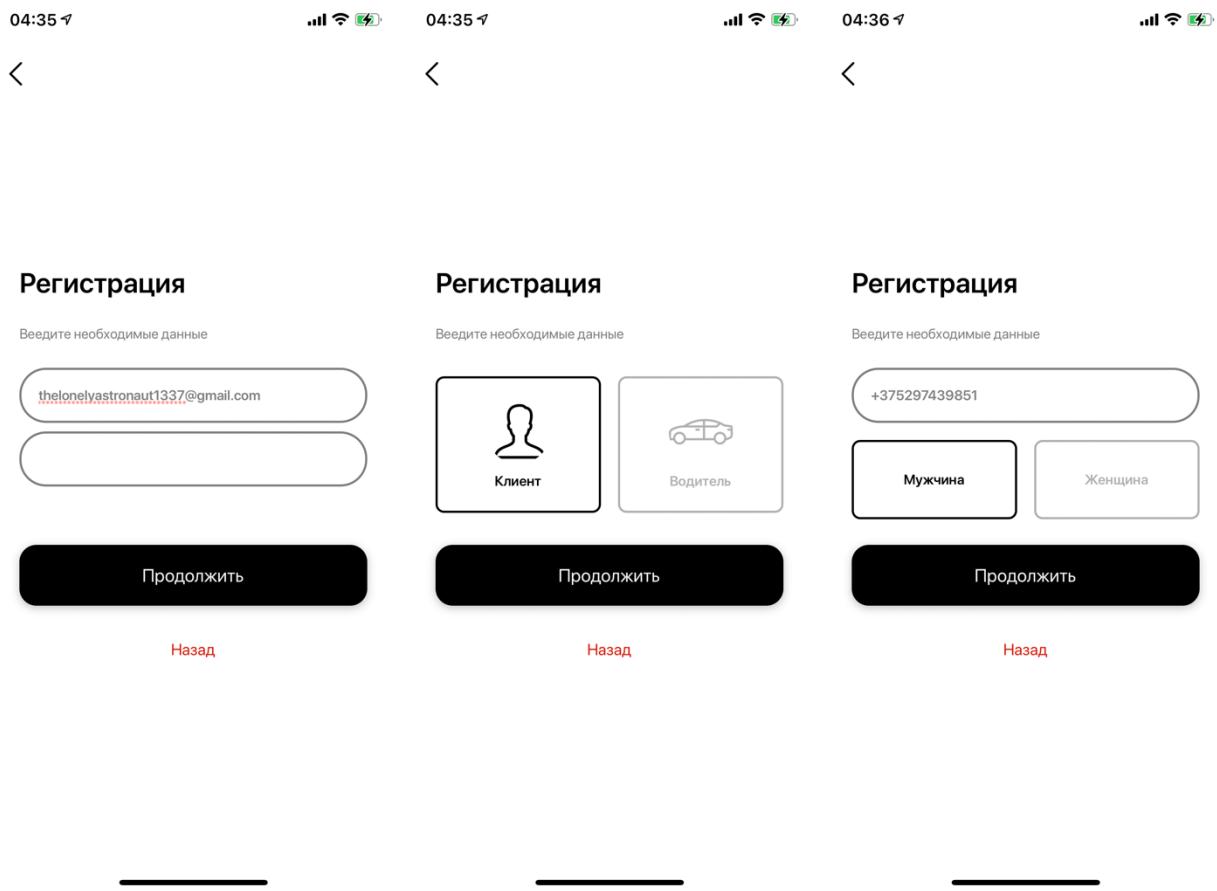


Рисунок 6.2 – Процесс регистрации

При регистрации пользователю требуется указать следующие поля:

- имя;
- фамилия;
- электронная почта;
- пароль;
- тип аккаунта (клиент или водитель);
- номер телефона, пол;
- 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.

С данного экрана пользователь может начать процесс вызова такси (в случае, если пользователь – клиент сервиса) или открыть боковое меню для дальнейшей навигации по приложению (переход к профилю, выбор способа оплаты, просмотр истории поездок или выход)

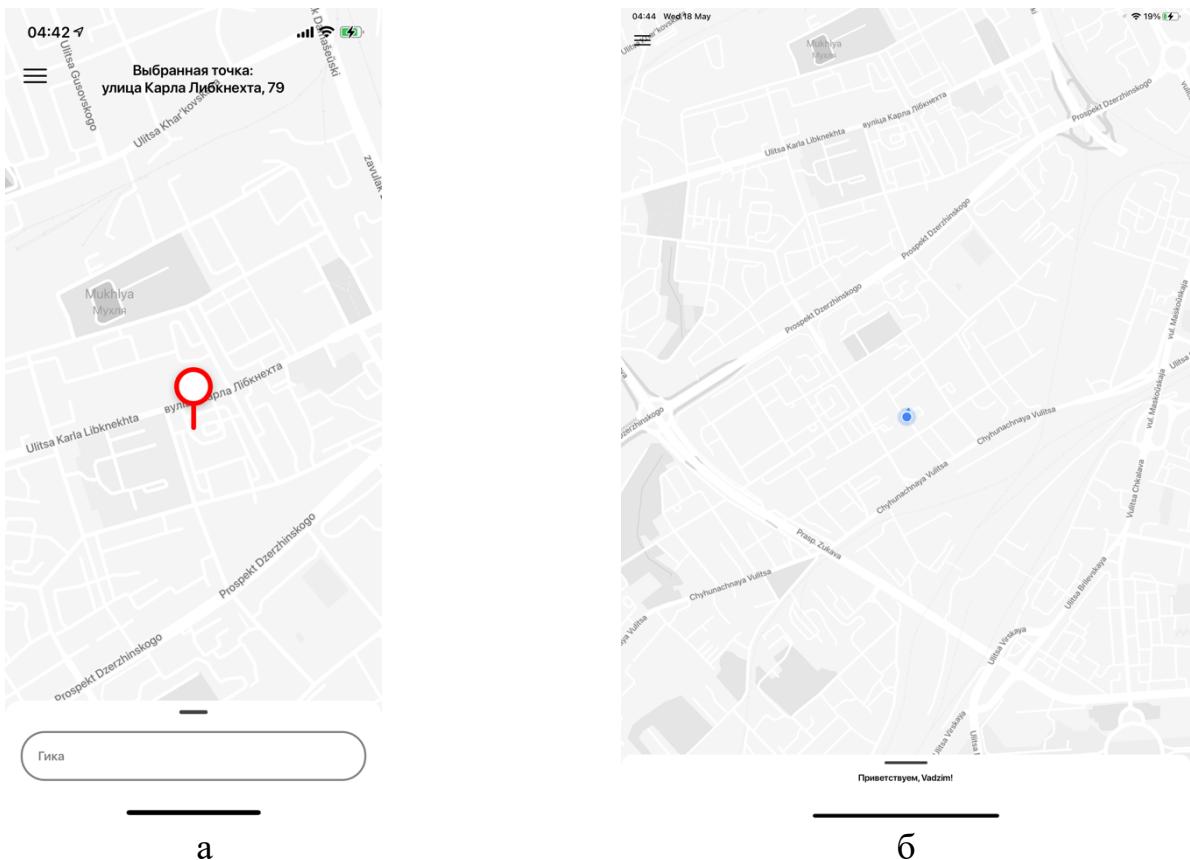


Рисунок 6.3 – Домашний экран (а – клиент, б – водитель)

При нажатии на имя профиля в боковом меню пользователь будет перенаправлен на страницу, содержащую информацию о профиле. Пример данной страницы изображен на рисунке 6.4.

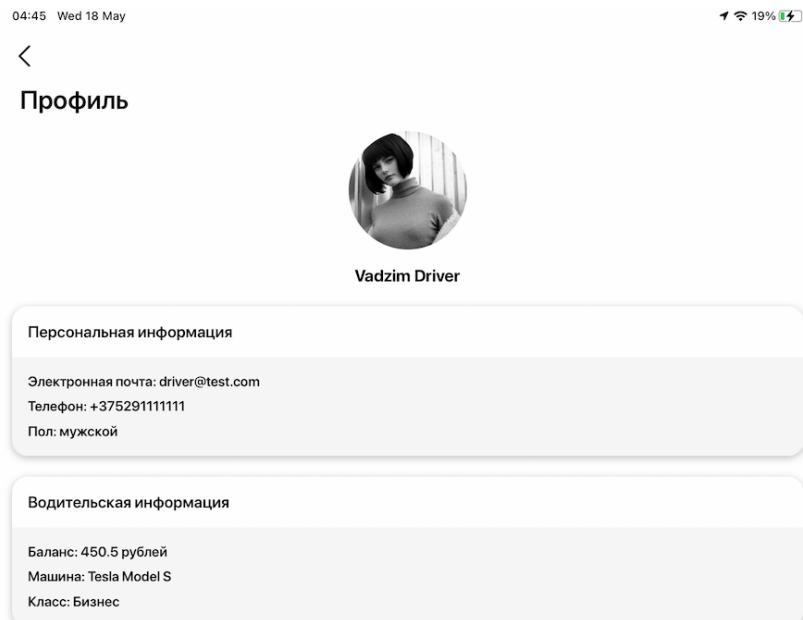


Рисунок 6.4 – Экран профиля

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 6.5.

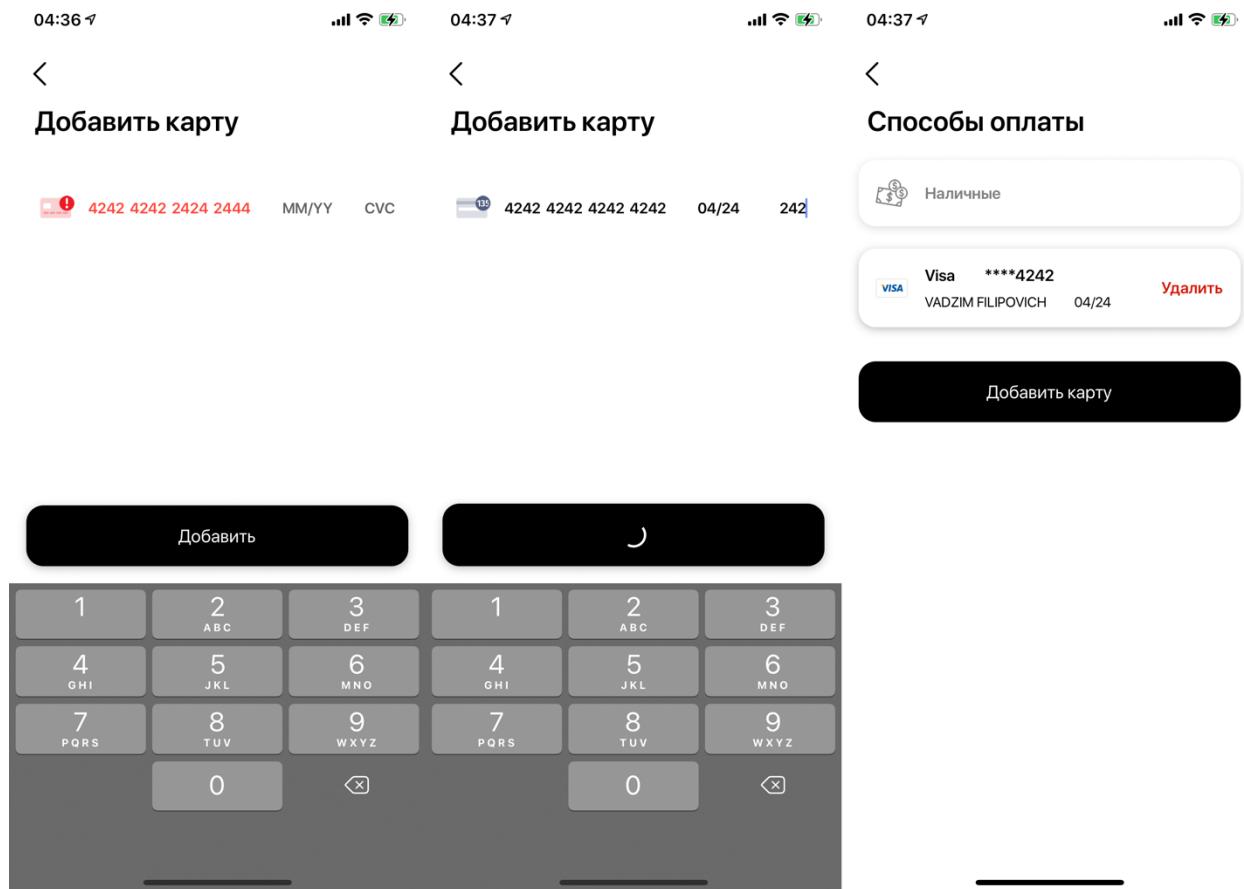


Рисунок 6.5 – Процесс добавления карты

При нажатии на историю поездок в боковом меню пользователь будет перенаправлен на экран со списком предыдущих поездок, изображенный на рисунке 6.6.

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 6.7.

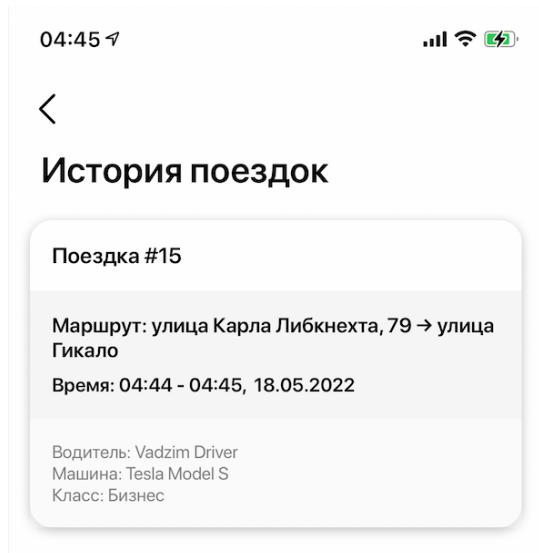


Рисунок 6.6 – История поездок

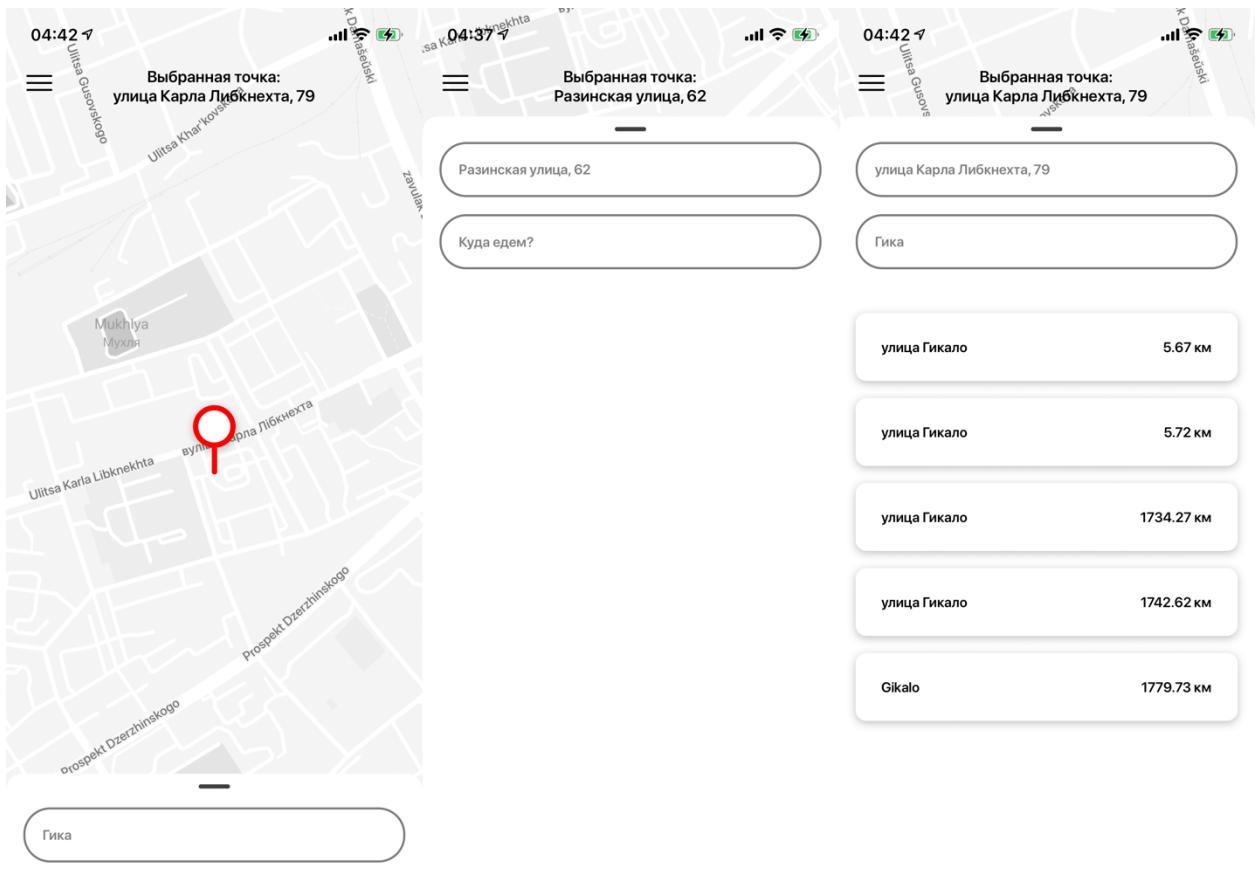


Рисунок 6.7 – Состояние выбора маршрута

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 6.8.

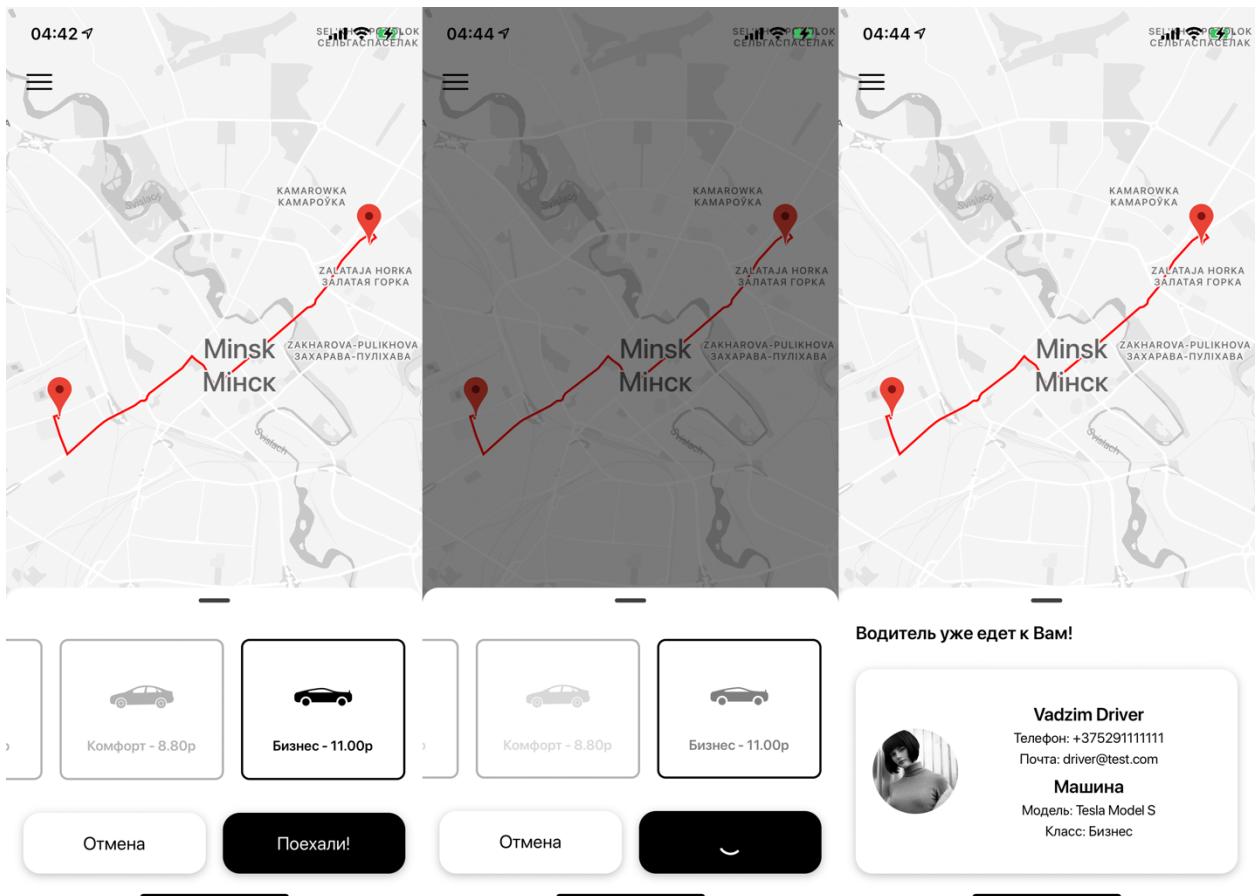


Рисунок 6.8 – Состояние выбора класса машины

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 6.9. 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 6.10. Данный рисунок также демонстрирует пример интерфейса на планшетных устройствах.

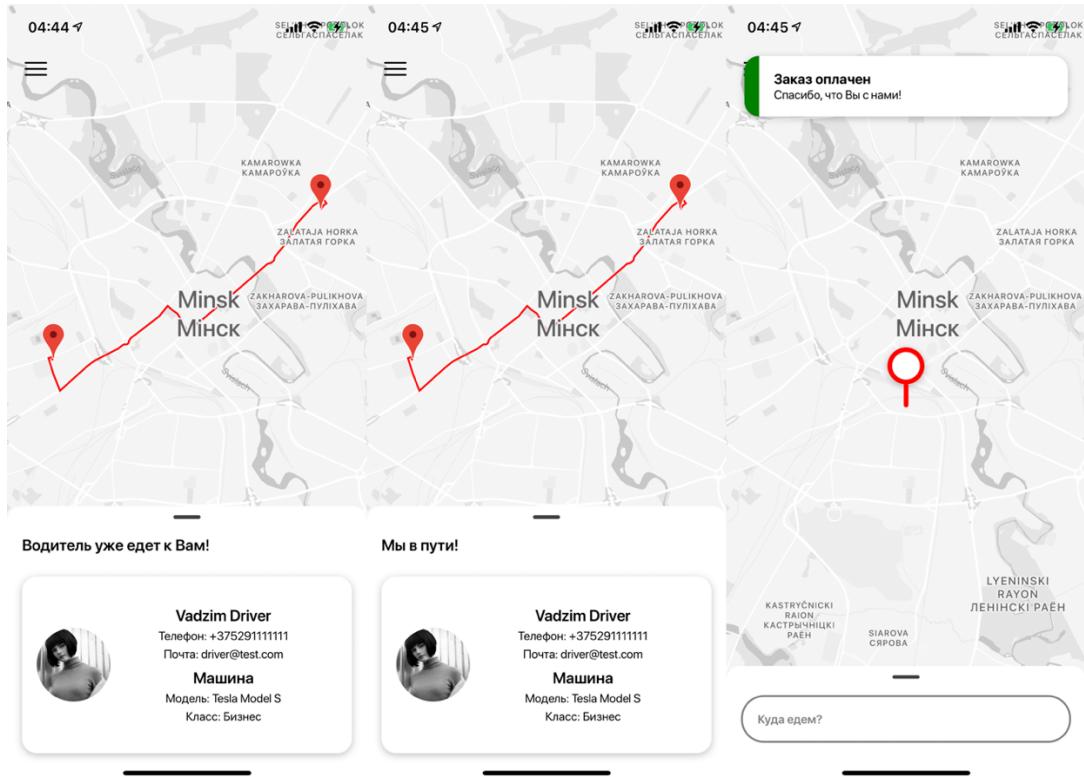


Рисунок 6.9 – Состояние поездки клиента

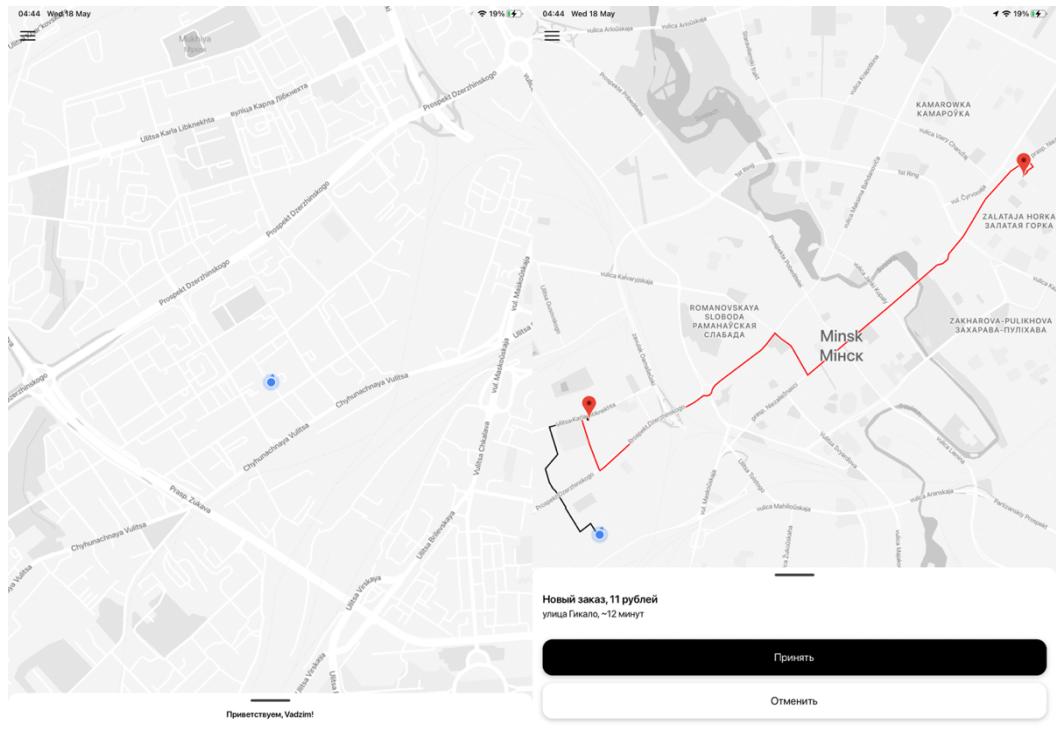


Рисунок 6.10 – Состояние запроса о рейсе

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 6.11.

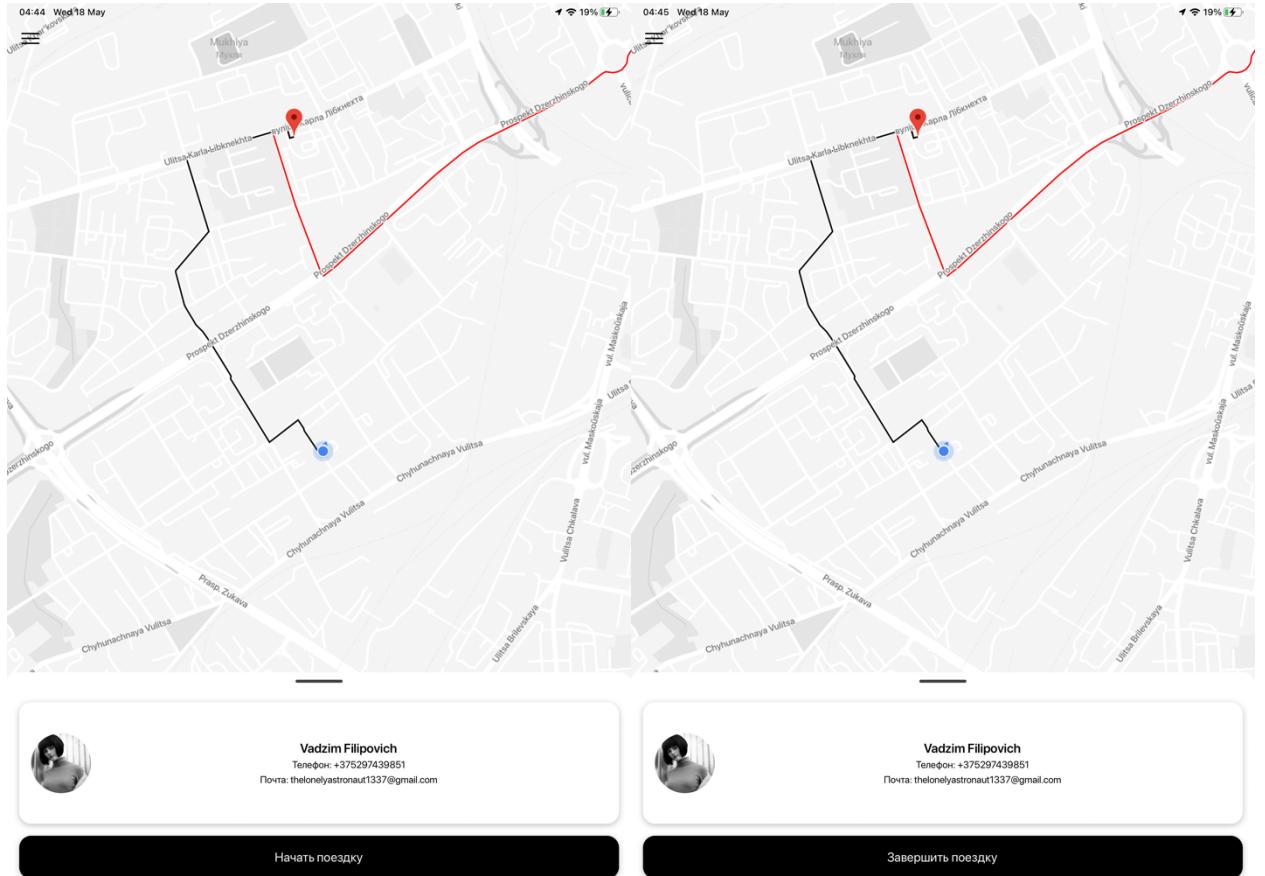


Рисунок 6.11 – Состояние поездки водителя

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:

$$Z_0 = K_{\text{пр}} \cdot \sum_{i=1}^n Z_{\text{ч}_i} \cdot t_i, \quad (7.1)$$

где n – количество исполнителей, занятых разработкой ПО;

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

$Z_{\text{ч}_i}$ – часовая заработка 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 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 заработка 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)*

Листинг кода

Файл app.reducer.ts

```
import { combineReducers } from 'redux';

import { userReducer } from '../../../../../core/data/store/user.reducer';
import { historyReducer } from '../../../../../features/history/data/store/history.reducer';
import { homeReducer } from '../../../../../features/home/data/store/home.reducer';
import { paymentsReducer } from '../../../../../features/payments/data/store/payments.reducer';

import { ApplicationState } from './app.types';

export const appReducer = combineReducers<ApplicationState>({
    user: userReducer,
    history: historyReducer,
    payments: paymentsReducer,
    home: homeReducer,
});
```

Файл app.saga.ts

```
import { SagaIterator } from 'redux-saga';
import { spawn } from 'redux-saga/effects';

import { listenForLogin } from '../../../../../features/auth/domain/login.saga';
import { listenForLogout } from '../../../../../features/auth/domain/logout.saga';
import { listenForRegister } from '../../../../../features/auth/domain/register.saga';
import { listenForFetchHistory } from '../../../../../features/history/domain/fetch-history.saga';
import { listenForChooseRoute } from '../../../../../features/home/domain/choose-route.saga';
import { listenForDeclineRideRequest } from '../../../../../features/home/domain/decline-ride.saga';
import { listenForFetchPlaces } from '../../../../../features/home/domain/fetch-places.saga';
import { listenForInitializeMap } from '../../../../../features/home/domain/initialize-map.saga';
import { listenForPrepareRide } from '../../../../../features/home/domain/prepare-ride-data.saga';
import { listenForAnswerToRideRequest, listenForRequestRideSaga } from
' '../../../../../features/home/domain/request-ride.saga';
import { listenForSetChosenLocation } from '../../../../../features/home/domain/set-chosen-location.saga';
import { listenForSetRideStatus } from '../../../../../features/home/domain/set-ride-status.saga';
import { listenForSetRouteLocation } from '../../../../../features/home/domain/set-route-location.saga';
import { listenForAddCard } from '../../../../../features/payments/domain/add-card.saga';
import { listenForGetPaymentMethods } from '../../../../../features/payments/domain/get-payment-
methods.saga';
import { listenForPay } from '../../../../../features/payments/domain/pay.saga';
import { listenForRemovePaymentMethod } from '../../../../../features/payments/domain/remove-payment-
method.saga';
import { listenForSetAsDefaultPaymentMethod } from '../../../../../features/payments/domain/set-as-
default.saga';
import { listenFor GetUser } from '../../../../../features/profile/domain/get-user.saga';

import { listenForInitialization } from './initialization.saga';

export function* appSaga(): SagaIterator {
    yield spawn(listenForInitialization);
    yield spawn(listenForLogin);
    yield spawn(listenForRegister);
    yield spawn(listenForLogout);
    yield spawn(listenForFetchHistory);
    yield spawn(listenFor GetUser);
    yield spawn(listenForGetPaymentMethods);
    yield spawn(listenForRemovePaymentMethod);
    yield spawn(listenForSetAsDefaultPaymentMethod);
    yield spawn(listenForAddCard);
    yield spawn(listenForPay);
    yield spawn(listenForInitializeMap);
    yield spawn(listenForSetChosenLocation);
    yield spawn(listenForRequestRideSaga);
    yield spawn(listenForAnswerToRideRequest);
    yield spawn(listenForSetRouteLocation);
    yield spawn(listenForFetchPlaces);
    yield spawn(listenForPrepareRide);
```

```

        yield spawn(listenForChooseRoute);
        yield spawn(listenForDeclineRideRequest);
        yield spawn(listenForSetRideStatus);
    }
}

```

Файл initialization.saga.ts

```

import { initStripe } from '@stripe/stripe-react-native';
import { SagaIterator } from 'redux-saga';
import { takeLatest, call, put, spawn } from 'redux-saga/effects';

import { connectionGatewayAPI } from '../../../../../core/data/api/connection-gateway-api.data';
import { Tokens } from '../../../../../core/model/tokens.model';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { tokenService } from '../../../../../core/utils/services/token-service.utils';
import { environmentConfig } from '../../../../../core/utils/third-party/environment-config.utils';
import { splashScreen } from '../../../../../core/utils/third-party/splash-screen.utils';
import { authAPI } from '../../../../../features/auth/data/api/auth-api.data';
import { LOGIN } from '../../../../../features/auth/data/auth.actions';
import { AuthResponse } from '../../../../../features/auth/model/auth-response.model';
import { fetchHistorySaga } from '../../../../../features/history/domain/fetch-history.saga';
import { bootstrapWebSocketSubscription } from '../../../../../features/home/domain/receive-ws-
message.saga';
import { geolocationService } from '../../../../../features/home/utils/services/geolocation-
service.utils';
import { getPaymentMethodsSaga } from '../../../../../features/payments/domain/get-payment-methods.saga';
import { profileAPI } from '../../../../../features/profile/data/api/profile-api.data';
import { UserResponse } from '../../../../../features/profile/model/user-response.model';
import { INITIALIZE } from '../data/store/app.actions';

export function* initializationSaga(): SagaIterator {
    yield call(geolocationService.initialize);
    yield spawn(bootstrapWebSocketSubscription);
    const tokens: Tokens = yield call(tokenService.readTokensFromStorage);

    if (tokens) {
        yield call(tokenService.setAuthToken, tokens);
        const newTokens: AuthResponse = yield call(authAPI.refreshToken, tokens.refreshToken);

        if (newTokens.data) {
            yield call(tokenService.writeTokensToStorage, newTokens.data);
            yield call(tokenService.setAuthToken, newTokens.data);

            const user: UserResponse = yield call(profileAPI.getUser);

            if (user.data) {
                yield call(connectionGatewayAPI.setIsClient, !user.data.driver);
                yield call(connectionGatewayAPI.connect);

                yield call(fetchHistorySaga);
                yield call(getPaymentMethodsSaga);

                yield put(LOGIN.COMPLETED(user.data));
            }
        } else {
            yield call(tokenService.deleteTokensFromStorage);
        }
    }
    const key = environmentConfig.get('stripePublishableKey') ?? '';
    if (!key) {
        yield call(toastService.showError, new Error('Не удалось настроить платежную систему'));
    } else {
        yield call(initStripe, { publishableKey: key });
    }

    yield call(splashScreen.hide);
}
export function* listenForInitialization(): SagaIterator {
    yield takeLatest(INITIALIZE, initializationSaga);
}

```

Файл app.component.tsx

```

import React from 'react';
import { Provider } from 'react-redux';

```

```

import { ThemeProvider } from 'styled-components';

import { Toast } from '../../core/presentation/toast/toast.component';
import { defaultTheme } from '../../core/utils/theme/themes.utils';
import { useStore } from '../utils/hooks/use-store.utils';

import { NavigationEntry } from './navigation-entry.component';

export const App: React.FC = () => {
  const store = useStore();

  return store ? (
    <Provider store={store}>
      <ThemeProvider theme={defaultTheme}>
        <NavigationEntry />
        <Toast />
      </ThemeProvider>
    </Provider>
  ) : null;
};

```

Файл use-store.utils.ts

```

import { useEffect, useRef, useState } from 'react';
import { applyMiddleware, createStore, Store } from 'redux';
import createSagaMiddleware from 'redux-saga';

import { Optional } from '../../../../../core/model/optional.model';
import { appReducer } from '../../../../../data/store/app.reducer';
import { appSaga } from '../../../../../domain/app.saga';

export const useStore = (): Optional<Store> => {
  const [, setIsReady] = useState(false);
  const store = useRef<Store>();

  useEffect(() => {
    (async () => {
      const sagaMiddleware = createSagaMiddleware();
      store.current = createStore(appReducer, applyMiddleware(sagaMiddleware));

      sagaMiddleware.run(appSaga);

      setIsReady(true);
    })();
  }, []);

  return store.current;
};

```

Файл connection-gateway-api.data.ts

```

import { logger } from '../../../../../utils/logger.utils';
import { environmentConfig } from '../../../../../utils/third-party/environment-config.utils';

// eslint-disable-next-line
// @ts-ignore
navigator.__defineGetter__('userAgent', function () {
  // you have to import react native first !!
  return 'react-native';
});

// eslint-disable-next-line
import { io, Socket } from 'socket.io-client';
// eslint-disable-next-line import/order
import { toastService } from '../../../../../utils/services/toast-service.utils';
// eslint-disable-next-line import/order
import { RideStatus } from '../../../../../model/ride.model';

export enum WSMimeType {
  RideRequest = 'RIDE_REQUEST',
  RideStatusChange = 'RIDE_STATUS_CHANGE',
  RideAccept = 'RIDE_ACCEPT',
  RideDecline = 'RIDE_DECLINE',
  RideStopSearch = 'RIDE_STOP_SEARCH',
  RideTimeout = 'RIDE_TIMEOUT',
}

```

```

    LocationUpdate = 'LOCATION_UPDATE',
    InternalReconnect = 'INTERNAL_RECONNECT',
    RestoreState = 'RESTORE_STATE',
}

export type WSMassage = {
    type: WSMassageType;
    payload?: RideStatus | unknown;
};

export type Listener = (data: WSMassage) => void;

export class ConnectionGatewayAPI {
    private authToken = '';
    private isClient = false;
    private listeners: Array<Listener> = [];
    private socket: Socket | null = null;

    setAuthToken = (token: string) => (this.authToken = token);
    setIsClient = (flag: boolean) => (this.isClient = flag);

    addEventListener = (callback: Listener): (() => void) => {
        this.listeners.push(callback);

        return () => this.listeners.splice(this.listeners.indexOf(callback), 0);
    };

    connect = async () => {
        try {
            this.socket = io(environmentConfig.get('connectionGatewayAPI'), {
                transports: ['websocket'],
                auth: {
                    Authorization: `Bearer ${this.authToken}`,
                },
                reconnection: true,
            });

            Object.values(WSMassageType).forEach((event) => {
                this.socket?.on(event, (data) => {
                    this.listeners.forEach((listener) => listener({ type: event, payload: data }));
                });
            });

            this.socket
                .on('connect_error', (error) => {
                    logger.log(error);
                    toastService.showError(undefined, 'Соединение потеряно', 'Восстанавливаем...');

                    this.retryConnection();
                })
                .on('connect', () => {
                    toastService.showSuccess('Соединение установлено', 'Приятного пользования');
                    this.listeners.forEach((listener) => listener({ type: WSMassageType.InternalReconnect }));
                })
                .on('disconnect', () => {
                    toastService.showError(undefined, 'Соединение закрыто', 'Восстанавливаем...');

                });
        } catch (e) {
            logger.log(e);
        }
    };

    send = async <T>(event: WSMassageType, data: T) => {
        this.socket?.emit(event, {
            isClient: this.isClient,
            data,
        });
    };

    disconnect = async () => {
        this.socket?.close();

        Object.values(WSMassageType).forEach((event) => {
            this.socket?.removeAllListeners(event);
        });
    };
}

```

```

};

private retryConnection = () => {
    this.socket?.connect();
};

}

export const connectionGatewayAPI = new ConnectionGatewayAPI();

```

Файл rest-gateway-api.data.ts

```

import axios, { AxiosInstance } from 'axios';

import { Optional } from '../../../../../model/optional.model';
import { Response } from '../../../../../model/response.model';
import { environmentConfig } from '../../../../../utils/third-party/environment-config.utils';

export class RestGatewayAPI {
    private authToken = '';
    axios: AxiosInstance;

    constructor() {
        this.axios = this.initialize();
    }

    private initialize = (): AxiosInstance => {
        const _axios = axios.create({
            baseURL: environmentConfig.get('restGatewayAPI'),
            timeout: 60000,
        });

        _axios.interceptors.request.use((config) => {
            if (this.authToken) {
                config.headers = {
                    ...config.headers,
                    Authorization: 'Bearer ' + this.authToken,
                };
            }

            return config;
        });

        return _axios;
    };

    setAuthToken = (token: string) => {
        this.authToken = token;
        this.axios = this.initialize();
    };

    post = async <K, T>(endpoint: string, data: K): Promise<Response<T>> => {
        try {
            const result = await this.axios.post(endpoint, data);
            const _data = typeof result.data === 'string' ? JSON.parse(result.data) : result.data;

            if (_data.error) {
                return {
                    status: 400,
                    error: new Error(_data.error),
                };
            }

            return {
                status: result.status,
                data: _data,
            };
        } catch (e) {
            return {
                status: 400,
                error: e as Error,
            };
        }
    };

    get = async <K, T>(endpoint: string, params?: Optional<K>): Promise<Response<T>> => {
        try {

```

```

        const result = await this.axios.get(endpoint, {
            params,
        });

        const _data = typeof result.data === 'string' ? JSON.parse(result.data) :
result.data;

        if (_data.error || _data.message) {
            return {
                status: 400,
                error: new Error(_data.error || _data.message),
            };
        }

        return {
            status: result.status,
            data: _data,
        };
    } catch (e) {
        return {
            status: 400,
            error: e as Error,
        };
    }
};

}

export const restGatewayAPI = new RestGatewayAPI();

```

Файл user.reducer.ts

```

import { createReducer } from '@reduxjs/toolkit';

import { LOGIN, LOGOUT, REGISTER } from '../../../../../features/auth/data/store/auth.actions';
import { GET_USER } from '../../../../../features/profile/data/store/profile.actions';

import { UserState } from './user.types';

const initialState: UserState = {
    isLoading: false,
    error: null,
    data: null,
};

export const userReducer = createReducer<UserState>(initialState, (builder) => {
    builder
        .addCase(LOGIN.STARTED, (state) => {
            state.isLoading = true;
        })
        .addCase(LOGIN.COMPLETED, (state, action) => {
            state.isLoading = false;
            state.error = null;
            state.data = action.payload;
        })
        .addCase(LOGIN.FAILED, (state, action) => {
            state.isLoading = false;
            state.error = action.payload;
            state.data = null;
        })
        .addCase(GET_USER.STARTED, (state) => {
            state.isLoading = true;
        })
        .addCase(GET_USER.COMPLETED, (state, action) => {
            state.isLoading = false;
            state.error = null;
            state.data = action.payload;
        })
        .addCase(GET_USER.FAILED, (state, action) => {
            state.isLoading = false;
            state.error = action.payload;
            state.data = null;
        })
        .addCase(REGISTER.STARTED, (state) => {
            state.isLoading = true;
        })
        .addCase(REGISTER.COMPLETED, (state, action) => {
            state.isLoading = false;

```

```

        state.error = null;
        state.data = action.payload;
    })
.addCase(REGISTER.FAILED, (state, action) => {
    state.isLoading = false;
    state.error = action.payload;
    state.data = null;
})
.addCase(LOGOUT.COMPLETED, (state) => {
    state.data = null;
});
);

```

Файл navigation-service.utils.ts

```

import { NavigationContainerRef } from '@react-navigation/native';

export class NavigationService {
    // eslint-disable-next-line
    private _navigationRef: NavigationContainerRef<any> | null = null;

    setNavigationRef = (ref: NavigationContainerRef<never>) => (this._navigationRef = ref);

    goBack = () => {
        this._navigationRef?.goBack();
    };
}
export const navigationService = new NavigationService();

```

Файл toast-service.utils.ts

```

import Toast from 'react-native-toast-message';

export class ToastService {
    showError = (error?: Error, title?: string, message?: string) => {
        Toast.show({
            type: 'error',
            text1: title ?? 'Ошибка',
            text2: message ?? error?.message ?? '',
            autoHide: true,
            position: 'top',
        });
    };

    showSuccess = (title: string, message: string) => {
        Toast.show({
            type: 'success',
            text1: title,
            text2: message,
            autoHide: true,
            position: 'top',
        });
    };
}

export const toastService = new ToastService();

```

Файл token-service.utils.ts

```

import { connectionGatewayAPI } from '../../../../../data/api/connection-gateway-api.data';
import { restGatewayAPI } from '../../../../../data/api/rest-gateway-api.data';
import { Optional } from '../../../../../model/optional.model';
import { Tokens } from '../../../../../model/tokens.model';
import { encryptedStorage } from '../third-party/encrypted-storage.utils';

export class TokenService {
    private storageKey = 'KOSOKU::AUTH_TOKENS';

    readTokensFromStorage = async (): Promise<Optional<Tokens>> => {
        return encryptedStorage.read(this.storageKey);
    };

    writeTokensToStorage = async (tokens: Tokens) => {
        await encryptedStorage.write(this.storageKey, tokens);
    };
}

```

```

    deleteTokensFromStorage = async () => {
      await encryptedStorage.delete(this.storageKey);
    };

    setAuthToken = async (tokens: Tokens) => {
      restGatewayAPI.setAuthToken(tokens.token);
      connectionGatewayAPI.setAuthToken(tokens.token);
    };

    removeAuthToken = async () => {
      restGatewayAPI.setAuthToken('');
      connectionGatewayAPI.setAuthToken('');
    };
  }

export const tokenService = new TokenService();

```

Файл encrypted-storage.utils.ts

```

import RNEncryptedStorage from 'react-native-encrypted-storage';

import { Optional } from '../../../../../model/optional.model';

export class EncryptedStorage {
  read = async <T>(key: string): Promise<Optional<T>> => {
    const data = await RNEncryptedStorage.getItem(key);

    return data ? (JSON.parse(data) as T) : null;
  };

  write = async <T>(key: string, data: T) => {
    await RNEncryptedStorage.setItem(key, JSON.stringify(data));
  };

  delete = async (key: string) => {
    await RNEncryptedStorage.removeItem(key);
  };
}

export const encryptedStorage = new EncryptedStorage();

```

Файл environment-config.utils.ts

```

import Config from 'react-native-config';

const readableEnv = {
  stripePublishableKey: 'STRIPE_PUBLISHABLE_KEY',
  restGatewayAPI: 'REST_GATEWAY_API',
  connectionGatewayAPI: 'CONNECTION_GATEWAY_API',
};

export type EnvironmentConfigVars = Record<keyof typeof readableEnv, string>;

export class EnvironmentConfig {
  get = (key: keyof EnvironmentConfigVars): string => {
    return Config[readableEnv[key]];
  };
}

export const environmentConfig = new EnvironmentConfig();

```

Файл auth-api.data.ts

```

import { RestGatewayAPI, restGatewayAPI } from '../../../../../core/data/api/rest-gateway-api.data';
import { AuthResponse } from '../../../../../model/auth-response.model';
import { LoginPayload, RegisterPayload } from '../store/auth.actions';

export class AuthAPI {
  constructor(private restGatewayAPI: RestGatewayAPI) {}

  login = async (data: LoginPayload): Promise<AuthResponse> => {
    return this.restGatewayAPI.post('/api/v1/auth/login', data);
  };

```

```

register = async (data: RegisterPayload): Promise<AuthResponse> => {
    return this.restGatewayAPI.post('/api/v1/auth/register', data);
};

refreshToken = async (refreshToken: string): Promise<AuthResponse> => {
    return this.restGatewayAPI.post('/api/v1/auth/refresh', { refreshToken });
};

}

export const authAPI = new AuthAPI(restGatewayAPI);

```

Файл login.saga.ts

```

import { SagaIterator } from 'redux-saga';
import { call, takeLatest, put } from 'redux-saga/effects';

import { connectionGatewayAPI } from '../../../../../core/data/api/connection-gateway-api.data';
import { logger } from '../../../../../core/utils/logger.utils';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { tokenService } from '../../../../../core/utils/services/token-service.utils';
import { fetchHistorySaga } from '../../../../../history/domain/fetch-history.saga';
import { getPaymentMethodsSaga } from '../../../../../payments/domain/get-payment-methods.saga';
import { profileAPI } from '../../../../../profile/data/api/profile-api.data';
import { UserResponse } from '../../../../../profile/model/user-response.model';
import { authAPI } from '../data/api/auth-api.data';
import { LOGIN } from '../data/store/auth.actions';
import { AuthResponse } from '../model/auth-response.model';

export function* loginSaga(action: ReturnType<typeof LOGIN.TRIGGER>): SagaIterator {
    yield put(LOGIN.STARTED());

    const result: AuthResponse = yield call(authAPI.login, action.payload);

    if (result.data) {
        yield call(tokenService.writeTokensToStorage, result.data);
        yield call(tokenService.setAuthToken, result.data);

        const user: UserResponse = yield call(profileAPI.getUser);

        if (user.data) {
            yield call(connectionGatewayAPI.setIsClient, !user.data.driver);
            yield call(connectionGatewayAPI.connect);
            yield call(fetchHistorySaga);
            yield call(getPaymentMethodsSaga);

            yield put(LOGIN.COMPLETED(user.data));
        }
    }

    if (user.error) {
        yield call(logger.log, user.error);
        yield call(toastService.showError, user.error);
        yield put(LOGIN.FAILED(user.error));
    }
}

export function* listenForLogin(): SagaIterator {
    yield takeLatest(LOGIN.TRIGGER, loginSaga);
}

```

Файл logout.saga.ts

```

import { SagaIterator } from 'redux-saga';
import { call, put, takeLatest } from 'redux-saga/effects';

import { connectionGatewayAPI } from '../../../../../core/data/api/connection-gateway-api.data';
import { tokenService } from '../../../../../core/utils/services/token-service.utils';
import { LOGOUT } from '../data/store/auth.actions';

export function* logoutSaga(): SagaIterator {
    yield call(connectionGatewayAPI.disconnect);
}

```

```

        yield call(tokenService.deleteTokensFromStorage);
        yield call(tokenService.removeAuthToken);

        yield put(LOGOUT.COMPLETED());
    }

export function* listenForLogout(): SagaIterator {
    yield takeLatest(LOGOUT.TRIGGER, logoutSaga);
}

```

Файл register.saga.ts

```

import { SagaIterator } from 'redux-saga';
import { call, takeLatest, put } from 'redux-saga/effects';

import { connectionGatewayAPI } from '../../../../../core/data/api/connection-gateway-api.data';
import { logger } from '../../../../../core/utils/logger.utils';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { tokenService } from '../../../../../core/utils/services/token-service.utils';
import { fetchHistorySaga } from '../../../../../history/domain/fetch-history.saga';
import { getPaymentMethodsSaga } from '../../../../../payments/domain/get-payment-methods.saga';
import { profileAPI } from '../../../../../profile/data/api/profile-api.data';
import { UserResponse } from '../../../../../profile/model/user-response.model';
import { authAPI } from '../data/api/auth-api.data';
import { REGISTER } from '../data/store/auth.actions';

export function* registerSaga(action: ReturnType<typeof REGISTER.TRIGGER>): SagaIterator {
    yield put(REGISTER.STARTED());

    const result = yield call(authAPI.register, action.payload);

    if (result.data) {
        yield call(tokenService.writeTokensToStorage, result.data);
        yield call(tokenService.setAuthToken, result.data);

        const user: UserResponse = yield call(profileAPI.getUser);

        if (user.data) {
            yield call(connectionGatewayAPI.setIsClient, !user.data.driver);
            yield call(connectionGatewayAPI.connect);
            yield call(fetchHistorySaga);
            yield call(getPaymentMethodsSaga);

            yield put(REGISTER.COMPLETED(user.data));
        }
    }

    if (user.error) {
        yield call(logger.log, user.error);
        yield call(toastService.showError, user.error);
        yield put(REGISTER.FAILED(user.error));
    }
}

if (result.error) {
    yield call(logger.log, result.error);
    yield call(toastService.showError, result.error);
    yield put(REGISTER.FAILED(result.error));
}
}

export function* listenForRegister(): SagaIterator {
    yield takeLatest(REGISTER.TRIGGER, registerSaga);
}

```

Файл login.component.tsx

```

import { useNavigation } from '@react-navigation/native';
import React, { useCallback, useRef } from 'react';
import { Keyboard } from 'react-native';
import { useDispatch, useSelector } from 'react-redux';

import { getIsUserLoading } from '../../../../../core/data/store/user.selectors';
import { SafeBackground } from '../../../../../core/presentation/background/background.styled';
import { Button } from '../../../../../core/presentation/button/button.component';
import { TextInput } from '../../../../../core/presentation/text-input/text-input.component';
import { screens } from '../../../../../navigation/utils/screens';

```

```

import { LOGIN } from '../../../../../data/store/auth.actions';
import { FormHolder } from '../components/form-holder/form-holder.styled';
import { FormTitle, FormSubtitle, FormBottomSubtitle, PressableText } from '../components/form-title/form-title.styled';

export const LoginScreen: React.FC = () => {
    const email = useRef('');
    const password = useRef('');
    const dispatch = useDispatch();
    const navigation = useNavigation();
    const isLoading = useSelector(getIsUserLoading);

    const handleLoginPress = useCallback(() => {
        Keyboard.dismiss();
        dispatch(LOGIN.TRIGGER({ email: email.current, password: password.current }));
    }, [dispatch]);

    const handleRegisterPress = useCallback(() => {
        if (isLoading) return;
        // eslint-disable-next-line
        // @ts-ignore
        navigation.navigate(screens.auth.register);
    }, [navigation, isLoading]);

    return (
        <SafeBackground>
            <FormHolder>
                <FormTitle>Путешествуйте с Kosoku</FormTitle>
                <FormSubtitle>Войдите, чтобы начать поездку</FormSubtitle>
                <TextInput
                    keyboardType={'email-address'}
                    placeholder={'Электронная почта'}
                    onChangeText={(text) => (email.current = text)}
                />
                <TextInput
                    secureTextEntry={true}
                    placeholder={'Пароль'}
                    onChangeText={(text) => (password.current = text)}
                />
                <Button isLoading={isLoading} title={'Войти'} onPress={handleLoginPress} />
                <FormBottomSubtitle>Еще не завели аккаунт?</FormBottomSubtitle>
                <PressableText onPress={handleRegisterPress}>Зарегистрируйтесь!</PressableText>
            </FormHolder>
        </SafeBackground>
    );
};

```

Файл register.component.tsx

```

import React, { useCallback, useRef, useState } from 'react';
import { Keyboard, ScrollView } from 'react-native';
import { useDispatch, useSelector } from 'react-redux';

import { getIsUserLoading } from '../../../../../core/data/store/user.selectors';
import { CarClass } from '../../../../../core/model/car-class.model';
import { Gender } from '../../../../../core/model/gender.model';
import { SafeBackground } from '../../../../../core/presentation/background/background.styled';
import { Button } from '../../../../../core/presentation/button/button.component';
import { Header } from '../../../../../core/presentation/header/header.component';
import { DefaultCard } from '../../../../../core/presentation/horizontal-picker/default-card.component';
import { HorizontalPicker } from '../../../../../core/presentation/horizontal-picker/horizontal-picker.component';
import { TextInput } from '../../../../../core/presentation/text-input/text-input.component';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { defaultTheme } from '../../../../../core/utils/theme/themes.utils';
import { REGISTER } from '../../../../../data/store/auth.actions';
import { FormHolder } from '../components/form-holder/form-holder.styled';
import {

    FormSubtitle,
    FormTitle,
    PressableText,
    PressableTextWrapper,
} from '../components/form-title/form-title.styled';
import { HorizontalPaging, Page, PagingWrapper, width } from '../components/vertical-paging/vertical-paging.styled';

```

```

export const RegisterScreen: React.FC = () => {
    const firstName = useRef('');
    const lastName = useRef('');
    const email = useRef('');
    const password = useRef('');
    const [isDriver, setIsDriver] = useState<boolean | null>(null);
    const phone = useRef('');
    const gender = useRef<Gender | null>(null);

    const carModel = useRef('');
    const carClass = useRef<CarClass | null>(null);

    const scroll = useRef<ScrollView | null>();
    const currentPage = useRef(0);
    const dispatch = useDispatch();

    const isLoading = useSelector(getIsUserLoading);

    const handleContinue = useCallback(() => {
        Keyboard.dismiss();

        if (currentPage.current === 0) {
            if (!firstName.current.length || !lastName.current.length) {
                toastService.showError(new Error('Заполните все поля'));
                return;
            }
        }

        if (currentPage.current === 1) {
            if (!email.current.length || !password.current.length) {
                toastService.showError(new Error('Заполните все поля'));
                return;
            }
        }

        if (currentPage.current === 2) {
            if (isDriver === null) {
                toastService.showError(new Error('Выберите тип аккаунта'));
                return;
            }
        }

        if (currentPage.current === 3) {
            if (!gender.current || !phone.current.length) {
                toastService.showError(new Error('Предоставьте необходимые данные'));
                return;
            }
        }

        if (!isDriver) {
            dispatch(
                REGISTER.TRIGGER({
                    email: email.current,
                    password: password.current,
                    phone: phone.current,
                    firstName: firstName.current,
                    lastName: lastName.current,
                    gender: gender.current,
                })
            );
        }

        return;
    });
}

if (currentPage.current === 4) {
    if (!carModel.current.length || !carClass.current) {
        toastService.showError(new Error('Предоставьте необходимые данные'));
        return;
    }

    dispatch(
        REGISTER.TRIGGER({
            email: email.current,
            password: password.current,
            phone: phone.current,
            firstName: firstName.current,
            lastName: lastName.current,
            gender: gender.current as Gender,
        })
    );
}

```

```

        driver: {
          carBrand: carModel.current,
          carClass: carClass.current,
          balance: 0,
        },
      })
    );
  }

  return;
}

currentPage.current += 1;

scroll.current?.scrollTo({
  x: width * currentPage.current,
  y: 0,
  animated: true,
});
}, [dispatch, isDriver]);

const handleGoBack = useCallback(() => {
  if (currentPage.current == 0 || isLoading) return;

  currentPage.current -= 1;

  scroll.current?.scrollTo({
    x: width * currentPage.current,
    y: 0,
    animated: true,
  });
}, [isLoading]);

return (
  <SafeBackground>
    <Header disabled={isLoading} title={''} />
    <FormHolder>
      <FormTitle>Регистрация</FormTitle>
      <FormSubtitle>Ведите необходимые данные</FormSubtitle>
      <PagingWrapper>
        <HorizontalPaging ref={(ref) => (scroll.current = ref)}>
          <Page>
            <TextInput placeholder={'Имя'} onChangeText={(text) =>
(firstName.current = text)} />
            <TextInput placeholder={'Фамилия'} onChangeText={(text) =>
(lastName.current = text)} />
          </Page>
          <Page>
            <TextInput
              placeholder={'Электронная почта'}
              keyboardType={'email-address'}
              onChangeText={(text) => (email.current = text)}
            />
            <TextInput
              secureTextEntry={true}
              placeholder={'Пароль'}
              onChangeText={(text) => (password.current = text)}
            />
          </Page>
          <Page>
            <HorizontalPicker
              content={roleRenderList}
              onSelectionChange={(index) => setIsDriver(index === 1)}
            />
          </Page>
          <Page>
            <TextInput
              keyboardType={'phone-pad'}
              placeholder={'Номер телефона'}
              onChangeText={(text) => (phone.current = text)}
            />
            <HorizontalPicker
              content={genderRenderList}
              onSelectionChange={(index) => (gender.current = index ? Gender.Female : Gender.Male)}
            />
          </Page>
        {isDriver && (
          <Page>

```

```

        <TextInput
            placeholder='Модель машины'
            onChangeText={(text) => (carModel.current = text)}
        />
        <HorizontalPicker
            content={carClassRenderList}
            onSelectionChange={(index) =>
                (carClass.current = index
                    ? index == 2
                        ? CarClass.Business
                        : CarClass.Economy
                    : CarClass.Economy)
            }
        />
    </Page>
)
</HorizontalPaging>
</PagingWrapper>
<Button title='Продолжить' onPress={handleContinue} isLoading={isLoading} />
<PressableTextWrapper>
    <PressableText style={{ marginLeft: 0 }} onPress={handleGoBack}>
        Назад
    </PressableText>
</PressableTextWrapper>
</FormHolder>
</SafeBackground>
);
};

const splitWidth = (width - defaultTheme.spacer * 11) / 2;
const splitWidthThree = (width - defaultTheme.spacer * 14) / 3;

const roleRenderList = [
    <DefaultCard
        title='Клиент'
        key='Клиент'
        asset={require('../../../assets/icons/client.png')}
        style={{ minWidth: splitWidth }}
    />,
    <DefaultCard
        title='Водитель'
        key='Водитель'
        asset={require('../../../assets/icons/driver.png')}
        style={{ minWidth: splitWidth }}
    />,
];
;

const genderRenderList = [
    <DefaultCard title='Мужчина' key='Мужчина' style={{ minWidth: splitWidth }} />,
    <DefaultCard title='Женщина' key='Женщина' style={{ minWidth: splitWidth }} />,
];

const carClassRenderList = [
    <DefaultCard title='Эконом' key='Эконом' style={{ minWidth: splitWidthThree }} />,
    <DefaultCard title='Комфорт' key='Комфорт' style={{ minWidth: splitWidthThree }} />,
    <DefaultCard title='Бизнес' key='Бизнес' style={{ minWidth: splitWidthThree }} />,
];

```

Файл history-api.data.ts

```

import { RestGatewayAPI, restGatewayAPI } from '../../../../../core/data/api/rest-gateway-api.data';
import { HistoryResponse } from '../../../../../model/history-response.model';

export class HistoryAPI {
    constructor(private restGatewayAPI: RestGatewayAPI) {}

    getHistory = async (): Promise<HistoryResponse> => {
        return this.restGatewayAPI.get('/api/v1/user/history');
    };
}
export const historyAPI = new HistoryAPI(restGatewayAPI);

```

Файл history.reducer.ts

```
import { createReducer } from '@reduxjs/toolkit';
```

```

import { FETCH_HISTORY } from './history.actions';
import { HistoryState } from './history.types';

const initialState: HistoryState = {
  isLoading: true,
  error: null,
  data: null,
};

export const historyReducer = createReducer<HistoryState>(initialState, (builder) => {
  builder
    .addCase(FETCH_HISTORY.STARTED, (state) => {
      state.isLoading = true;
    })
    .addCase(FETCH_HISTORY.COMPLETED, (state, action) => {
      state.isLoading = false;
      state.data = action.payload;
      state.error = null;
    })
    .addCase(FETCH_HISTORY.FAILED, (state, action) => {
      state.isLoading = false;
      state.data = null;
      state.error = action.payload;
    });
});

```

Файл fetch-history.saga.ts

```

import { SagaIterator } from 'redux-saga';
import { call, put, takeLatest } from 'redux-saga/effects';

import { logger } from '../../../../../core/utils/logger.utils';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { historyAPI } from '../data/api/history-api.data';
import { FETCH_HISTORY } from '../data/store/history.actions';
import { HistoryResponse } from '../model/history-response.model';

export function* fetchHistorySaga(): SagaIterator {
  yield put(FETCH_HISTORY.STARTED());
  const result: HistoryResponse = yield call(historyAPI.getHistory);
  if (result.data) {
    yield put(FETCH_HISTORY.COMPLETED(result.data));
  }
  if (result.error) {
    yield call(logger.log, result.error);
    yield call(toastService.showError, result.error);
    yield put(FETCH_HISTORY.FAILED(result.error));
  }
}

export function* listenForFetchHistory(): SagaIterator {
  yield takeLatest(FETCH_HISTORY.TRIGGER, fetchHistorySaga);
}

```

history.component.tsx

```

import React, { useCallback, useEffect } from 'react';
import { FlatList, ListRenderItemInfo } from 'react-native';
import { useDispatch, useSelector } from 'react-redux';

import { SafeBackground } from '../../../../../core/presentation/background/background.styled';
import { Header } from '../../../../../core/presentation/header/header.component';
import { FETCH_HISTORY } from '../data/store/history.actions';
import { getHistoryData, getIsHistoryLoading } from '../data/store/history.selectors';
import { Order } from '../model/order.model';

import { OrderTile } from './components/order-tile/order-tile.component';

const renderItem = (item: ListRenderItemInfo<Order>) => <OrderTile order={item.item} />

export const HistoryScreen: React.FC = () => {
  const isLoading = useSelector(getIsHistoryLoading);
  const data = useSelector(getHistoryData) ?? [];

```

```

const dispatch = useDispatch();

const getHistory = useCallback(() => {
    dispatch(FETCH_HISTORY.TRIGGER());
}, [dispatch]);

useEffect(() => {
    getHistory();
    // eslint-disable-next-line
}, []);

return (
    <SafeBackground edges={['top']}>
        <Header title='История поездок' />
        <FlatList
            data={[...data].reverse()}
            onRefresh={getHistory}
            refreshing={isLoading}
            showsVerticalScrollIndicator={false}
            keyExtractor={(item) => item.id.toString()}
            renderItem={renderItem}
        />
    </SafeBackground>
);
};

```

Файл home-api.data.ts

```

import {
    ConnectionGatewayAPI,
    connectionGatewayAPI,
    WSMimeType,
} from '../../../../../core/data/api/connection-gateway-api.data';
import { RestGatewayAPI, restGatewayAPI } from '../../../../../core/data/api/rest-gateway-api.data';
import { ExtendedRideRequest, RideStatus } from '../../../../../core/model/ride.model';
import { ExtendedLocation, Location } from '../../../../../model/location.model';
import { DecodeResponse, DirectionsResponse, PlacesResponse } from '../../../../../model/network.model';

export class HomeAPI {
    constructor(private restGatewayAPI: RestGatewayAPI, private connectionGatewayAPI: ConnectionGatewayAPI) {}

    decode = async (location: Location): Promise<DecodeResponse> => {
        return this.restGatewayAPI.post('/api/v1/maps/decode', { location });
    };

    updateMyLocation = async (location: Location) => {
        return this.connectionGatewayAPI.send(WSMimeType.LocationUpdate, location);
    };

    requestRide = async (data: ExtendedRideRequest) => {
        return this.connectionGatewayAPI.send(WSMimeType.RideRequest, data);
    };

    answerToRideRequest = async (answer: WSMimeType) => {
        return this.connectionGatewayAPI.send(answer, { type: answer });
    };

    fetchPlaces = async (toSearch: string): Promise<PlacesResponse> => {
        return this.restGatewayAPI.post('/api/v1/maps/places', { toSearch });
    };

    calculateRideData = async (to: ExtendedLocation, from: ExtendedLocation): Promise<DirectionsResponse> => {
        return this.restGatewayAPI.post('/api/v1/maps/direction', { to, from });
    };

    declineRideRequest = async () => {
        return this.connectionGatewayAPI.send(WSMimeType.RideStopSearch, { type: WSMimeType.RideStopSearch });
    };

    updateRideStatus = async (status: RideStatus) => {
        return this.connectionGatewayAPI.send(WSMimeType.RideStatusChange, status);
    };
}

```

```
export const homeAPI = new HomeAPI(restGatewayAPI, connectionGatewayAPI);
```

Файл home.reducer.ts

```

    })
    .addCase(SET_CHOSEN_LOCATION.COMPLETED, (state, action) => {
        state.pointerLocation.isLoading = false;
        state.pointerLocation.data = {
            isMoving: false,
            location: action.payload,
        };
        state.pointerLocation.error = null;
    })
    .addCase(SET_CHOSEN_LOCATION.FAILED, (state, action) => {
        state.pointerLocation.data = {
            isMoving: false,
            location: null,
        };
        state.pointerLocation.error = action.payload;
        state.pointerLocation.isLoading = false;
    })
    .addCase(POINTER_MOVE.START, (state) => {
        state.pointerLocation.data = {
            location: state.pointerLocation.data?.location,
            isMoving: true,
        };
        state.pointerLocation.isLoading = true;
    })
    .addCase(POINTER_MOVE.STOP, (state) => {
        state.pointerLocation.data = {
            location: state.pointerLocation.data?.location,
            isMoving: false,
        };
    })
    .addCase(SET_RIDE_REQUEST, (state, action) => {
        state.prepareRide.rideRequest.data = action.payload;
        state.prepareRide.rideRequest.isLoading = false;
    })
    .addCase(REQUEST_RIDE.COMPLETED, (state, action) => {
        state.prepareRide.isCarSearching = action.payload;
    })
    .addCase(SET_RIDE_STATUS.COMPLETED, (state, action) => {
        state.ride.status = action.payload;
    })
    .addCase(SET_ROUTE_LOCATION.COMPLETED, (state, action) => {
        const key = action.payload.to ? 'to' : 'from';

        state.chooseRoute[key].data = {
            pickedLocation: action.payload[key],
            searchResults: [],
        };
        state.chooseRoute[key].isLoading = false;
        state.chooseRoute[key].error = null;
    })
    .addCase(FETCH_PLACES.STARTED, (state, action) => {
        state.chooseRoute[action.payload.direction].isLoading = true;
    })
    .addCase(FETCH_PLACES.COMPLETED, (state, action) => {
        state.chooseRoute[action.payload.direction].data = {
            pickedLocation:
                state.chooseRoute[action.payload.direction]?.data?.pickedLocation,
            searchResults: action.payload.results,
        };

        state.chooseRoute.latestChange = action.payload.direction;
    })
    .addCase(FETCH_PLACES.FAILED, (state, action) => {
        state.chooseRoute[action.payload.direction].error = action.payload.error;
        state.chooseRoute.latestChange = action.payload.direction;
    })
    .addCase(PREPARE_RIDE.STARTED, (state, action) => {
        state.prepareRide.isPreparing = true;
        state.prepareRide.rideRequest.isLoading = true;
        state.prepareRide.to = action.payload.to;
        state.prepareRide.from = action.payload.from;

        state.chooseRoute.isChoosingRoute = false;
        state.chooseRoute.latestChange = undefined;
        state.chooseRoute.from = {
            isLoading: false,
            data: null,
            error: null,
        }
    })
}

```

```

    };
    state.chooseRoute.to = {
      isLoading: false,
      data: null,
      error: null,
    };
  });
.addCase(CHOOSE_ROUTE.COMPLETED, (state) => {
  state.prepareRide = { ...initialState.prepareRide };

  state.chooseRoute.isChoosingRoute = true;
})
.addCase(DECLINE_RIDE_REQUEST.COMPLETED, (state) => {
  state.prepareRide.isCarSearching = false;
})
.addCase(SET_DRIVER_RIDE_REQUEST, (state, action) => {
  if (!action.payload) {
    state.ride.toPickUp = state.prepareDriverRide.rideRequest?.toPickUp;

    state.ride.route = state.prepareDriverRide.rideRequest?.route;

    state.ride.to = state.prepareDriverRide.rideRequest?.to;
    state.ride.from = state.prepareDriverRide.rideRequest?.from;

    state.prepareRide.to = state.prepareDriverRide.rideRequest!.to;
    state.prepareRide.from = state.prepareDriverRide.rideRequest!.from;

    state.prepareRide.rideRequest.data = {
      route: state.prepareDriverRide.rideRequest!.route,
    } as RideRequest;
  }

  state.prepareDriverRide.rideRequest = action.payload;
})
.addCase(SET_RIDE, (state, action) => {
  state.ride.ride = action.payload;
  state.ride.status = action.payload.status;

  state.ride.route = state.prepareRide.rideRequest.data?.route;
  state.ride.to = state.prepareRide.to ?? state.prepareDriverRide.rideRequest?.to;
  state.ride.from = state.prepareRide.from ?? state.prepareDriverRide.rideRequest?.from;

  state.prepareDriverRide.rideRequest?.from;

  // Reset
  state.prepareRide = { ...initialState.prepareRide };
})
.addCase(RESET_HOME_STATE, (state) => {
  state.ride = { ...initialState.ride };
  state.prepareRide = { ...initialState.prepareRide };
  state.chooseRoute = { ...initialState.chooseRoute };
  state.prepareDriverRide = { ...initialState.prepareDriverRide };
})
.addCase(RESTORE_DRIVE_STATE, (state, action) => {
  state.ride.ride = action.payload.ride;
  state.ride.status = action.payload.ride.status;
  state.ride.route = action.payload.request.route;
  state.ride.toPickUp = action.payload.toPickUp;
  state.ride.to = action.payload.request.to;
  state.ride.from = action.payload.request.from;
  state.chooseRoute.isChoosingRoute = false;
  state.prepareRide.isPreparing = false;
});
);

```

Файл choose-route.saga.ts

```

import { SagaIterator } from 'redux-saga';
import { call, put, takeLatest } from 'redux-saga/effects';

import { CHOOSE_ROUTE } from '../data/store/home.actions';
import { geolocationService } from '../utils/services/geolocation-service.utils';
import { mapService } from '../utils/services/map-service.utils';

export function* chooseRouteSaga(): SagaIterator {
  yield put(CHOOSE_ROUTE.COMPLETED());

  if (geolocationService.latestLocation) {

```

```

        yield call(mapService.animateCamera, geolocationService.latestLocation, 15);
    }
}

export function* listenForChooseRoute(): SagaIterator {
    yield takeLatest(CHOSE_ROUTE.TRIGGER, chooseRouteSaga);
}

```

Файл fetch-places.saga.ts

```

import { SagaIterator } from 'redux-saga';
import { call, debounce, put } from 'redux-saga/effects';

import { logger } from '../../../../../core/utils/logger.utils';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { homeAPI } from '../data/api/home-api.data';
import { FETCH_PLACES } from '../data/store/home.actions';
import { PlacesResponse } from '../model/network.model';

export function* fetchPlacesSaga(action: ReturnType<typeof FETCH_PLACES.TRIGGER>): SagaIterator {
    yield put(FETCH_PLACES.STARTED(action.payload));

    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());

    if (result.data) {
        yield put(FETCH_PLACES.COMPLETED({ results: result.data, direction: action.payload.direction }));
    }

    if (result.error) {
        yield call(logger.log, result.error);
        yield call(toastService.showError, result.error);
        yield put(FETCH_PLACES.FAILED({ error: result.error, direction: action.payload.direction }));
    }
}

export function* listenForFetchPlaces(): SagaIterator {
    yield debounce(300, FETCH_PLACES.TRIGGER, fetchPlacesSaga);
}

```

Файл initialize-map.saga.ts

```

import { SagaIterator } from 'redux-saga';
import { call, put, spawn, takeLatest } from 'redux-saga/effects';

import { logger } from '../../../../../core/utils/logger.utils';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { INITIALIZE_MAP, SET_CHOSEN_LOCATION } from '../data/store/home.actions';
import { geolocationService } from '../../utils/services/geolocation-service.utils';
import { mapService } from '../../utils/services/map-service.utils';

import { bootstrapGPSSubscription } from './receive-gps-position.saga';

export function* initializeMapSaga(): SagaIterator {
    try {
        const currentLocation = yield call(geolocationService.getLocation);

        yield call(mapService.animateCamera, currentLocation, 15);
        yield put(SET_CHOSEN_LOCATION.TRIGGER(currentLocation));

        yield spawn(bootstrapGPSSubscription);
    } catch (e) {
        yield put(SET_CHOSEN_LOCATION.TRIGGER());
        yield call(logger.log, e);
        yield call(toastService.showError, new Error((e as Error).message));
    }
}

export function* listenForInitializeMap(): SagaIterator {

```

```
        yield takeLatest(INITIALIZE_MAP.TRIGGER, initializeMapSaga);
    }
```

Файл prepare-ride-data.saga.ts

```
import { SagaIterator } from 'redux-saga';
import { call, delay, put, takeLatest } from 'redux-saga/effects';

import { logger } from '../../../../../core/utils/logger.utils';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { homeAPI } from '../data/api/home-api.data';
import { PREPARE_RIDE, SET_RIDE_REQUEST } from '../data/store/home.actions';
import { DirectionsResponse } from '../model/network.model';
import { bottomSheetService } from '../utils/services/bottom-sheet-service.utils';
import { mapService } from '../utils/services/map-service.utils';

import { chooseRouteSaga } from './choose-route.saga';

export function* prepareRideDataSaga(action: ReturnType<typeof PREPARE_RIDE.TRIGGER>): SagaIterator {
    yield put(PREPARE_RIDE.STARTED(action.payload));

    yield delay(100);

    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);

    if (result.status >= 200 && result.status < 300) {
        yield put(SET_RIDE_REQUEST(result.data));
    }

    if (result.error) {
        yield call(logger.log, result.error);
        yield call(toastService.showError, result.error);
        yield put(SET_RIDE_REQUEST());
        yield call(chooseRouteSaga);
    }
}

export function* listenForPrepareRide(): SagaIterator {
    yield takeLatest(PREPARE_RIDE.TRIGGER, prepareRideDataSaga);
}
```

Файл receive-gps-position.saga.ts

```
import { eventChannel, EventChannel, SagaIterator, Subscribe } from 'redux-saga';
import { call, select, takeLatest } from 'redux-saga/effects';

import { getExistingUser } from '../../../../../core/data/store/user.selectors';
import { homeAPI } from '../data/api/home-api.data';
import { Location } from '../model/location.model';
import { geolocationService } from '../utils/services/geolocation-service.utils';

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);
}
```

Файл receive-ws-message.saga.ts

```
import { eventChannel, EventChannel, SagaIterator, Subscribe } from 'redux-saga';
import { call, put, select, takeLatest } from 'redux-saga/effects';

import { connectionGatewayAPI, WSMassage, WSMassageType } from
'../../../../core/data/api/connection-gateway-api.data';
import { getExistingUser } from '../../../../core/data/store/user.selectors';
import { ExtendedDriverRideRequest, Ride, RideStatus } from '../../../../core/model/ride.model';
import { User } from '../../../../core/model/user.model';
import { logger } from '../../../../core/utils/logger.utils';
import { toastService } from '../../../../core/utils/services/toast-service.utils';
import { PAY } from '../../../../payments/data/store/payments.actions';
import {

  REQUEST_RIDE,
  RESET_HOME_STATE,
  RESTORE_DRIVE_STATE,
  RestoreDriveStatePayload,
  SET_DRIVER_RIDE_REQUEST,
  SET_RIDE,
  SET_RIDE_STATUS,
} from '../data/store/home.actions';
import { getRide } from '../data/store/home.selectors';
import { geolocationService } from '../../utils/services/geolocation-service.utils';
import { mapService } from '../../utils/services/map-service.utils';

import { receiveLocationUpdateSaga } from './receive-gps-position.saga';

export const webSocketSubscribe: Subscribe<WSMassage> = (emitter) =>
connectionGatewayAPI.addEventListener(emitter);

export const webSocketMessagesReceiver = (): EventChannel<WSMassage> =>
eventChannel<WSMassage>(webSocketSubscribe);

export function* receiveWebSocketMessageSaga(message: WSMassage): SagaIterator {
  const user: User = yield select(getExistingUser);

  if (!user) return;

  yield call(logger.log, message);

  switch (message.type) {
    case WSMassageType.InternalReconnect: {
      const location = yield call(geolocationService.getLocation);
      yield call(receiveLocationUpdateSaga, location);
      break;
    }
    case WSMassageType.LocationUpdate: {
      break;
    }
    case WSMassageType.RideRequest: {
      const data = message.payload as { data: ExtendedDriverRideRequest };

      if (!user.driver) {
        break;
      }

      yield put(SET_DRIVER_RIDE_REQUEST(data.data));
      yield call(mapService.animateToRegion, data.data.to, data.data.from);

      break;
    }
    case WSMassageType.RideStatusChange: {
      const rideStatus = (message.payload as any).status as RideStatus;

      yield put(SET_RIDE_STATUS.COMPLETED(rideStatus));

      if (rideStatus === RideStatus.Completed) {
        const ride: Ride = yield select(getRide);
        yield put(RESET_HOME_STATE());
        yield put(PAY.TRIGGER({ amount: ride.cost, rideId: ride.id }));
      }

      break;
    }
    case WSMassageType.RideAccept: {
      const data = message.payload as Ride;
    }
  }
}
```

```

        yield put(REQUEST_RIDE.COMPLETED(false));
        yield put(SET_RIDE_STATUS.COMPLETED(RideStatus.Starting));
        yield put(SET_RIDE(data));

        break;
    }
    case WSMimeType.RideDecline: {
        yield put(SET_RIDE_STATUS.COMPLETED(RideStatus.NoRide));

        if (!user.driver) {
            yield put(REQUEST_RIDE.COMPLETED(false));
            yield call(toastService.showError, undefined, 'Поездка отменена', 'Повторите попытку позже');
        } else {
            // TODO
        }

        break;
    }
    case WSMimeType.RestoreState: {
        const data = message.payload as RestoreDriveStatePayload;
        yield put(RESTORE_DRIVE_STATE(data));

        break;
    }
    case WSMimeType.RideTimeout: {
        yield put(SET_DRIVER_RIDE_REQUEST());

        if (geolocationService.latestLocation) {
            yield call(mapService.animateCamera, geolocationService.latestLocation, 15);
        }

        break;
    }
    default: {
        yield call(toastService.showError, new Error('Неизвестная серверная команда'));
    }
}
}

export function* bootstrapWebSocketSubscription(): SagaIterator {
    const webSocketEventChannel = yield call(webSocketMessagesReceiver);

    yield takeLatest(webSocketEventChannel, receiveWebSocketMessageSaga);
}

```

Файл request-ride.saga.ts

```

import { SagaIterator } from 'redux-saga';
import { call, put, select, takeLatest } from 'redux-saga/effects';

import { RideRequest } from '../../../../../core/model/ride.model';
import { homeAPI } from '../data/api/home-api.data';
import { ANSWER_TO_RIDE_REQUEST, REQUEST_RIDE, SET_DRIVER_RIDE_REQUEST } from '../data/store/home.actions';
import { getPrepareRideFromLocation, getPrepareRideToLocation, getRideRequest } from '../data/store/home.selectors';
import { geolocationService } from '../utils/services/geolocation-service.utils';
import { mapService } from '../utils/services/map-service.utils';

export function* requestRideSaga(action: ReturnType<typeof REQUEST_RIDE.TRIGGER>): SagaIterator {
    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,
    });

    yield put(REQUEST_RIDE.COMPLETED(true));
}

```

```

export function* listenForRequestRideSaga(): SagaIterator {
    yield takeLatest(REQUEST_RIDE.TRIGGER, requestRideSaga);
}

export function* answerToRideRequestSaga(action: ReturnType<typeof ANSWER_TO_RIDE_REQUEST>): SagaIterator {
    yield put(SET_DRIVER_RIDE_REQUEST());

    if (geolocationService.latestLocation) {
        yield call(mapService.animateCamera, geolocationService.latestLocation, 15);
    }

    yield call(homeAPI.answerToRideRequest, action.payload);
}

export function* listenForAnswerToRideRequest(): SagaIterator {
    yield takeLatest(ANSWER_TO_RIDE_REQUEST, answerToRideRequestSaga);
}

```

Файл set-chosen-location.saga.ts

```

import { SagaIterator } from 'redux-saga';
import { call, debounce, put, select } from 'redux-saga/effects';

import { logger } from '../../../../../core/utils/logger.utils';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { homeAPI } from '../data/api/home-api.data';
import { SET_CHOSEN_LOCATION } from '../data/store/home.actions';
import { getIsPointerMoving } from '../data/store/home.selectors';
import { DecodeResponse } from '../model/network.model';

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);
}

```

Файл set-ride-status.saga.ts

```

import { SagaIterator } from 'redux-saga';
import { call, put, select, takeLatest } from 'redux-saga/effects';

import { RideStatus } from '../../../../../core/model/ride.model';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { homeAPI } from '../data/api/home-api.data';
import { RESET_HOME_STATE, SET_RIDE_STATUS } from '../data/store/home.actions';
import { getRide } from '../data/store/home.selectors';

export function* setRideStatusSaga(action: ReturnType<typeof SET_RIDE_STATUS.TRIGGER>): SagaIterator {
    yield call(homeAPI.updateRideStatus, action.payload);
    yield put(SET_RIDE_STATUS.COMPLETED(action.payload));
}

```

```

        if (action.payload === RideStatus.Completed) {
            const ride = yield select(getRide);
            yield call(toastService.showSuccess, 'Поездка завершена', `Ваша прибыль - ${ride.cost}р`);
            yield put(RESET_HOME_STATE());
        }
    }

export function* listenForSetRideStatus(): SagaIterator {
    yield takeLatest(SET_RIDE_STATUS.TRIGGER, setRideStatusSaga);
}

```

Файл set-route-location.saga.ts

```

import { SagaIterator } from 'redux-saga';
import { call, put, select, takeLatest } from 'redux-saga/effects';

import { Optional } from '../../../../../core/model/optional.model';
import { PREPARE_RIDE, SET_ROUTE_LOCATION } from '../data/store/home.actions';
import {
    getChooseRouteFromLocation,
    getChooseRouteToLocation,
    getPointerLocationData,
} from '../data/store/home.selectors';
import { ExtendedLocation } from '../model/location.model';
import { mapService } from '../utils/services/map-service.utils';

export function* setRouteLocationSaga(action: ReturnType<typeof SET_ROUTE_LOCATION.TRIGGER>): SagaIterator {
    if (action.payload.from) {
        const location: Optional<ExtendedLocation> = yield select(getPointerLocationData);

        if (
            location?.longitude !== action.payload.from.longitude ||
            location?.latitude !== action.payload.from.latitude
        ) {
            yield call(mapService.animateCamera, action.payload.from, 12);
        }
    }

    yield put(SET_ROUTE_LOCATION.COMPLETED({ ...action.payload }));

    const from = yield select(getChooseRouteFromLocation);
    const to = yield select(getChooseRouteToLocation);

    if (from && to) {
        yield put(PREPARE_RIDE.TRIGGER({ from, to }));
    }
}

export function* listenForSetRouteLocation(): SagaIterator {
    yield takeLatest(SET_ROUTE_LOCATION.TRIGGER, setRouteLocationSaga);
}

```

Файл location.model.ts

```

import { LatLng } from 'react-native-maps';

export type Location = LatLng;

export type ExtendedLocation = Location & {
    readableLocation?: string;
};

```

Файл network.model.ts

```

import { CarClass } from '../../../../../core/model/car-class.model';
import { Response } from '../../../../../core/model/response.model';
import { RideRequest } from '../../../../../core/model/ride.model';

import { ExtendedLocation } from './location.model';

export type DecodeResponse = Response<ExtendedLocation>;
export type PlacesResponse = Response<Array<ExtendedLocation>>;

```

```
export type DirectionsResponse = Response<RideRequest>;
```

Файл home.screen.tsx

```
import BottomSheet from '@gorhom/bottom-sheet';
import { DrawerNavigationProp } from '@react-navigation/drawer';
import React, { useCallback, useMemo } from 'react';
import { Keyboard, View, StyleSheet } from 'react-native';
import MapView, { Marker, Polyline, PROVIDER_GOOGLE } from 'react-native-maps';
import { useSharedValue } from 'react-native-reanimated';
import { useSafeAreaInsets } from 'react-native-safe-area-context';
import { useSelector } from 'react-redux';

import mapStyle from '../../../../../assets/maps/style.json';
import { getIsDriver } from '../../../../../core/data/store/user.selectors';
import { Background } from '../../../../../core/presentation/background/background.styled';
import {
  getIsCarSearching,
  getIsChoosingRoute,
  getIsDriverIdle,
  getIsUserOnRide,
  getIsDriverRequested,
  getIsRidePreparing,
  getPrepareRideFromLocation,
  getPrepareRideToLocation,
  getRideRequest,
  getDriverRideRequest,
  getRideStateRoute,
  getRideStateToPickUp,
  getRide,
  getRideStateToPoint,
  getRideStateFromPoint,
  getHomeState,
} from '../../../../../data/store/home.selectors';
import { useMapSetup } from '../utils/hooks/use-map-setup.utils';
import { bottomSheetService } from '../utils/services/bottom-sheet-service.utils';
import { mapService } from '../utils/services/map-service.utils';

import { burgerAsset, BurgerMenu, BurgerWrapper } from './components/burger-menu/burger-menu.styled';
import { ChooseClass } from './components/choose-class/choose-class.component';
import { CustomerRideStatus } from './components/customer-ride-status/customer-ride-status.component';
import { DriverIdleStatus } from './components/driver-status/driver-idle-status.component';
import { DriverOnRideStatus } from './components/driver-status/driver-on-ride-status.component';
import { Pointer } from './components/pointer/pointer.component';
import { RideRequest } from './components/ride-request/ride-request.component';
import { SearchBlock } from './components/search-block/search-block.component';
import { Status } from './components/status/status.component';
import { StatusWrapper } from './components/status/status.styled';

export const HomeScreen = ({ navigation }: { navigation: DrawerNavigationProp<never> }) => {
  const insets = useSafeAreaInsets();
  const animatedSheetPosition = useSharedValue(0);

  const clientRideRequest = useSelector(getRideRequest);
  const driverRideRequest = useSelector(getDriverRideRequest);

  const isPreparing = useSelector(getIsRidePreparing);
  const isChoosingRoute = useSelector(getIsChoosingRoute);
  const isDriver = useSelector(getIsDriver);
  const isCarSearching = useSelector(getIsCarSearching);
  const isDriverIdle = useSelector(getIsDriverIdle);
  const isDriverRequested = useSelector(getIsDriverRequested);
  const isUserOnRide = useSelector(getIsUserOnRide);

  const clientTo = useSelector(getPrepareRideToLocation);
  const clientFrom = useSelector(getPrepareRideFromLocation);
  const rideTo = useSelector(getRideStateToPoint);
  const rideFrom = useSelector(getRideStateFromPoint);

  const rideStateRoute = useSelector(getRideStateRoute);
  const rideStateToPickUp = useSelector(getRideStateToPickUp);

  console.log(rideStateToPickUp);
```

```

const to = useMemo(() => {
    if (rideTo) return rideTo;

    if (isDriver && driverRideRequest?.to) {
        return driverRideRequest.to;
    } else if (clientTo) {
        return clientTo;
    }

    return null;
}, [isDriver, clientTo, driverRideRequest, rideTo]);

const from = useMemo(() => {
    if (rideFrom) return rideFrom;

    if (isDriver && driverRideRequest?.from) {
        return driverRideRequest.from;
    } else if (clientFrom) {
        return clientFrom;
    }

    return null;
}, [isDriver, clientFrom, driverRideRequest, rideFrom]);

const rideRequest = useMemo(
    () => (isDriver ? driverRideRequest : clientRideRequest),
    [isDriver, driverRideRequest, clientRideRequest]
);

const route = useMemo(() => rideStateRoute ?? rideRequest?.route, [rideStateRoute,
rideRequest]);
const toPickUp = useMemo(
    () => rideStateToPickUp ?? driverRideRequest?.toPickUp,
    [rideStateToPickUp, driverRideRequest]
);

const isDraggable = useMemo(
    () => (!isDriver && !isCarSearching && !isPreparing) || (!clientRideRequest && isDriver),
    [isDriver, isCarSearching, clientRideRequest, isPreparing]
);

const onBottomSheetAnimate = useCallback((fromIndex: number, toIndex: number) => {
    if (fromIndex === 1 && toIndex === 0) {
        Keyboard.dismiss();
    }
}, []);
const { onRegionChange, onRegionChangeComplete, pointerRef, snapPoints } = useMapSetup();

return (
    <Background disableMargins={true}>
        <MapView
            ref={mapService.getMapRef()}
            provider={PROVIDER_GOOGLE}
            style={{ flex: 1 }}
            customMapStyle={mapStyle}
            showsUserLocation={isDriver}
            onRegionChange={!isDriver && isChoosingRoute ? onRegionChange : undefined}
            scrollEnabled={isDraggable}
            zoomEnabled={isDraggable}
            pitchEnabled={isDraggable}
            rotateEnabled={false}
            onRegionChangeComplete={!isDriver && isChoosingRoute ? onRegionChangeComplete :
undefined}
        >
            <!--from && !to && (
                <>
                    <Marker coordinate={from} />
                    <Marker coordinate={to} />
                    {!route && <Polyline coordinates={route} strokeWidth={2}
strokeColor={'red'} />
                    {!toPickUp && <Polyline coordinates={toPickUp} strokeWidth={2} />}
                </>
            )-->
        </MapView>
        {/* eslint-disable-next-line react-native/no-color-literals */}
        {isCarSearching && (
            <View style={{ ...StyleSheet.absoluteFillObject, backgroundColor:
'rgba(0,0,0,0.5)' }} />
        )
    
```

```

        )
        {!isDriver && isChoosingRoute && <Pointer ref={pointerRef} />}
        {isDriver && !clientRideRequest && <Pointer />}
        <StatusWrapper style={{ top: insets.top }}>
            <Status />
        </StatusWrapper>
        <BurgerWrapper onPress={navigation.openDrawer}>
            <BurgerMenu source={burgerAsset} style={{ top: insets.top }} />
        </BurgerWrapper>
        <BottomSheet
            animatedPosition={animatedSheetPosition}
            onAnimate={onBottomSheetAnimate}
            snapPoints={snapPoints}
            index={0}
            ref={bottomSheetService.setRef}
        >
            {isDriver ? (
                <>
                    {isDriverIdle && <DriverIdleStatus />}
                    {isDriverRequested && <RideRequest />}
                    {isUserOnRide && <DriverOnRideStatus />}
                </>
            ) : (
                <>
                    {isChoosingRoute && <SearchBlock
                    animatedSheetPosition={animatedSheetPosition} />}
                    {isPreparing && <ChooseClass />}
                    {isUserOnRide && <CustomerRideStatus />}
                </>
            )}
        </BottomSheet>
    </Background>
);
};

```

Файл use-map-setup.utils.ts

```

import React, { useCallback, useEffect, useMemo, useRef } from 'react';
import { Dimensions } from 'react-native';
import { useDispatch, useSelector } from 'react-redux';

import { getExistingUser } from '../../../../../core/data/store/user.selectors';
import { INITIALIZE_MAP, POINTER_MOVE, SET_CHOSEN_LOCATION } from
'../../../../data/store/home.actions';
import {
    getIsChoosingRoute,
    getIsDriverIdle,
    getIsUserOnRide,
    getIsDriverRequested,
    getIsPointerMoving,
} from '../../../../../data/store/home.selectors';
import { Location } from '../../../../../model/location.model';
import { PointerRef } from '../../../../../presentation/components/pointer/pointer.component';

export type MapUtils = {
    onRegionChange: (location: Location) => void;
    onRegionChangeComplete: (location: Location) => void;
    pointerRef: React.MutableRefObject<PointerRef | null>;
    snapPoints: Array<string | number>;
};

// User snaps
export const chooseRouteClientSnaps = [Dimensions.get('window').height * 0.15,
Dimensions.get('window').height * 0.88];
export const revertChooseRouteClientSnaps = [
    Dimensions.get('window').height * 0.85,
    Dimensions.get('window').height * 0.12,
];
export const prepareRideSnaps = [300];

// Driver snaps
export const idleSnaps = [100];
export const requestedSnaps = [300];
export const onRideSnaps = [300];

export const useMapSetup = (): MapUtils => {

```

```

const dispatch = useDispatch();
const wasStarted = useRef(false);
const isPointerMoving = useSelector(getIsPointerMoving);
const pointerRef = useRef<PointerRef | null>(null);
const user = useSelector(getExistingUser);
const isChoosingRoute = useSelector(getIsChoosingRoute);

const isDriverIdle = useSelector(getIsDriverIdle);
const isDriverRequested = useSelector(getIsDriverRequested);
const isUserOnRide = useSelector(getIsUserOnRide);

const snapPoints = useMemo(
() =>
    user?.driver
        ? isDriverIdle
            ? idleSaps
            : isDriverRequested
            ? requestedSaps
            : isUserOnRide
            ? onRideSaps
            : idleSaps
        : isChoosingRoute
        ? chooseRouteClientSaps
        : prepareRideSaps,
    [user, isChoosingRoute, isUserOnRide, isDriverIdle, isDriverRequested]
);

useEffect(() => {
    dispatch(INITIALIZE_MAP.TRIGGER());
    // eslint-disable-next-line
}, []);

const onRegionChange = useCallback(() => {
    if (!wasStarted.current || !isPointerMoving) {
        if (!wasStarted.current) {
            wasStarted.current = true;
            pointerRef.current?.start();
        }

        dispatch(POINTER_MOVE.START());
    }
}, [dispatch, isPointerMoving]);

const onRegionChangeComplete = useCallback(
(location: Location) => {
    wasStarted.current = false;
    pointerRef.current?.stop();
    dispatch(POINTER_MOVE.STOP());
    dispatch(SET_CHOSEN_LOCATION.TRIGGER(location));
},
[dispatch]
);

return {
    onRegionChange,
    onRegionChangeComplete,
    pointerRef,
    snapPoints,
};
};

}

```

Файл bottom-sheet-service.utils.ts

```

import { BottomSheetMethods } from '@gorhom/bottom-sheet/lib/typescript/types';
import { Optional } from '../../../../../core/model/optional.model';

export class BottomSheetService {
    private bottomSheetRef: Optional<BottomSheetMethods>;

    setRef = (ref: BottomSheetMethods) => (this.bottomSheetRef = ref);

    expand = () => {
        setTimeout(() => this.bottomSheetRef?.expand(), 100);
    };

    snapToIndex = (index: number) => {

```

```

        setTimeout(() => this.bottomSheetRef?.snapToIndex(index), 100);
    };

    minimize = () => {
        this.bottomSheetRef?.snapToIndex(0);
    };

    close = () => {
        setTimeout(() => this.bottomSheetRef?.forceClose(), 100);
    };
}

export const bottomSheetService = new BottomSheetService();

```

Файл geolocation-service.utils.ts

```

import Geolocation, {
    GeolocationError,
    GeolocationOptions,
    GeolocationResponse,
} from '@react-native-community/geolocation';

import { logger } from '../../../../../core/utils/logger.utils';
import { Location } from '../../../../../model/location.model';

export class GeolocationService {
    latestLocation: Location | null = null;

    private commonGeolocationOptions: GeolocationOptions = {
        enableHighAccuracy: true,
    };

    initialize = () => {
        Geolocation.setRNConfiguration({
            authorizationLevel: 'whenInUse',
            skipPermissionRequests: false,
        });
        Geolocation.requestAuthorization();
    };

    getLocation = async (): Promise<Location> => {
        return new Promise((resolve, reject) => {
            Geolocation.getCurrentPosition(
                (result: GeolocationResponse) => {
                    resolve({
                        latitude: result.coords.latitude,
                        longitude: result.coords.longitude,
                    });
                },
                (error: GeolocationError) => {
                    if (this.latestLocation) {
                        resolve(this.latestLocation);
                    } else {
                        reject(error);
                    }
                },
                {
                    ...this.commonGeolocationOptions,
                    timeout: 5000,
                }
            );
        });
    };

    subscribeToPositionChange = (listener: (data: Location) => void): (() => void) => {
        const subID = Geolocation.watchPosition(
            (location) => {
                this.latestLocation = {
                    latitude: location.coords.latitude,
                    longitude: location.coords.longitude,
                };
                listener({ ...this.latestLocation });
            },
            logger.log,
            this.commonGeolocationOptions
        );
    };
}

```

```

    );
    return () => Geolocation.clearWatch(subID);
}
}

export const geolocationService = new GeolocationService();

```

Файл map-service.utils.ts

```

import React from 'react';
import MapView from 'react-native-maps';

import { Location } from '../../model/location.model';
import { prepareRideSnaps } from '../hooks/use-map-setup.utils';

class MapService {
    private map = React.createRef<MapView>();

    getMapRef = () => this.map;

    animateCamera = (location: Location, zoom?: number) => {
        this.map.current?.animateCamera(
            {
                center: location,
                zoom,
            },
            {
                duration: 400,
            }
        );
    };

    animateToRegion = (location1: Location, location2: Location) => {
        this.map.current?.fitToCoordinates([location1, location2], {
            animated: true,
            edgePadding: {
                top: 70,
                bottom: prepareRideSnaps[0] + 20,
                left: 50,
                right: 50,
            },
        });
    };
}

export const mapService = new MapService();

```

Файл geo-distance.utils.ts

```

export const 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;
}

```

```
    }
};
```

Файл payments-api.data.ts

```
import { RestGatewayAPI, restGatewayAPI } from '../../../../../core/data/api/rest-gateway-api.data';
import { CreatePaymentIntentInput } from '../../../../../model/create-payment-intent-input.model';
import { CardMethodDetails } from '../../../../../model/method-details.model';
import { PaymentFinishedInput, PaymentFinishedResponse } from '../../../../../model/payment-
finished.model';
import {
    AddCardResponse,
    CreatePaymentIntentResponse,
    GetPaymentMethodsResponse,
    RemovePaymentMethodResponse,
    SetAsDefaultMethodResponse,
} from '../../../../../model/payments-response.model';

export class PaymentsAPI {
    constructor(private restGatewayAPI: RestGatewayAPI) {}

    getPaymentMethods = async (): Promise<GetPaymentMethodsResponse> => {
        return this.restGatewayAPI.get('/api/v1/payments');
    };

    setAsDefaultMethod = async (methodId: number): Promise<SetAsDefaultMethodResponse> => {
        return this.restGatewayAPI.post('/api/v1/payments/default', { methodId });
    };

    addCard = async (details: CardMethodDetails): Promise<AddCardResponse> => {
        return this.restGatewayAPI.post('/api/v1/payments/add', details);
    };

    createPaymentIntent = async (details: CreatePaymentIntentInput): Promise<CreatePaymentIntentResponse> => {
        return this.restGatewayAPI.post('/api/v1/payments/intent', details);
    };

    removePaymentMethod = async (methodId: number): Promise<RemovePaymentMethodResponse> => {
        return this.restGatewayAPI.post('/api/v1/payments/remove', { methodId });
    };

    paymentFinished = async (data: PaymentFinishedInput): Promise<PaymentFinishedResponse> => {
        return this.restGatewayAPI.post('/api/v1/payments/confirm', data);
    };
}

export const paymentsAPI = new PaymentsAPI(restGatewayAPI);
```

Файл payments.reducer.ts

```
import { createReducer } from '@reduxjs/toolkit';

import {
    ADD_CARD,
    GET_PAYMENT_METHODS,
    PAY,
    REMOVE_PAYMENT_METHOD,
    SET_AS_DEFAULT_PAYMENT_METHOD,
} from './payments.actions';
import { PaymentsState } from './payments.types';

const initialState: PaymentsState = {
    list: {
        isLoading: false,
        error: null,
        data: null,
    },
    addCard: {
        isLoading: false,
        error: null,
        data: null,
    },
    payment: {
        isLoading: false,
        error: null,
```

```

        data: null,
    },
};

export const paymentsReducer = createReducer<PaymentsState>(initialState, (builder) => {
    builder
        .addCase(GET_PAYMENT_METHODS.STARTED, (state) => {
            state.list.isLoading = true;
        })
        .addCase(GET_PAYMENT_METHODS.COMPLETED, (state, action) => {
            state.list.error = null;
            state.list.data = action.payload;
            state.list.isLoading = false;
        })
        .addCase(GET_PAYMENT_METHODS.FAILED, (state, action) => {
            state.list.data = null;
            state.list.error = action.payload;
            state.list.isLoading = false;
        })
        .addCase(SET_AS_DEFAULT_PAYMENT_METHOD.STARTED, (state) => {
            state.list.isLoading = true;
        })
        .addCase(SET_AS_DEFAULT_PAYMENT_METHOD.COMPLETED, (state) => {
            state.list.error = null;
            state.list.isLoading = false;
        })
        .addCase(SET_AS_DEFAULT_PAYMENT_METHOD.FAILED, (state, action) => {
            state.list.error = action.payload;
            state.list.isLoading = false;
        })
        .addCase(REMOVE_PAYMENT_METHOD.STARTED, (state) => {
            state.list.isLoading = true;
        })
        .addCase(REMOVE_PAYMENT_METHOD.COMPLETED, (state) => {
            state.list.error = null;
            state.list.isLoading = false;
        })
        .addCase(REMOVE_PAYMENT_METHOD.FAILED, (state, action) => {
            state.list.error = action.payload;
            state.list.isLoading = false;
        })
        .addCase(ADD_CARD.STARTED, (state) => {
            state.addCard.isLoading = true;
        })
        .addCase(ADD_CARD.COMPLETED, (state) => {
            state.addCard.isLoading = false;
            state.addCard.error = null;
        })
        .addCase(ADD_CARD.FAILED, (state, action) => {
            state.addCard.isLoading = false;
            state.addCard.error = action.payload;
        })
        .addCase(PAY.STARTED, (state) => {
            state.payment.isLoading = true;
        })
        .addCase(PAY.COMPLETED, (state) => {
            state.payment.isLoading = false;
            state.payment.error = null;
        })
        .addCase(PAY.FAILED, (state, action) => {
            state.payment.isLoading = false;
            state.payment.error = action.payload;
        });
    );
});

```

Файл add-card.saga.ts

```

import { confirmPayment } from '@stripe/stripe-react-native';
import { ConfirmPaymentResult, PaymentIntents } from '@stripe/stripe-react-native/src/types/index';
import { SagaIterator } from 'redux-saga';
import { call, delay, put, takeLatest } from 'redux-saga/effects';

import { logger } from '../../../../../core/utils/logger.utils';
import { navigationService } from '../../../../../core/utils/services/navigation-service.utils';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { paymentsAPI } from '../data/api/payments-api.data';

```

```

import { ADD_CARD } from '../data/store/payments.actions';
import { CreatePaymentIntentResponse } from '../model/payments-response.model';

import { getPaymentMethodsSaga } from './get-payment-methods.saga';

function* handleError(error: Error): SagaIterator {
    yield call(logger.log, error);
    yield call(toastService.showError, error);
    yield put(ADD_CARD.FAILED(error));
}

export function* addCardSaga(action: ReturnType<typeof ADD_CARD.TRIGGER>): SagaIterator {
    yield put(ADD_CARD.STARTED());

    const result: CreatePaymentIntentResponse = yield call(paymentsAPI.createPaymentIntent, {
        bynAmount: 300, // rubles * 100
        requestThreeDSecure: 'automatic',
    });

    if (result.data) {
        const secret = result.data.clientSecret;

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

        if (paymentIntent?.status === PaymentIntents.Status.Succeeded) {
            const result = yield call(paymentsAPI.addCard, {
                ...action.payload,
                stripePaymentId: paymentIntent.paymentMethodId,
            });

            if (result.status >= 200 && result.status < 300) {
                yield put(ADD_CARD.COMPLETED());

                yield delay(100);
                yield call(navigationService.goBack);
                yield call(getPaymentMethodsSaga);
                yield call(toastService.showSuccess, 'Карта успешно добавлена', 'Желаем приятных поездок');
            }
        }

        if (result.error) {
            yield call(handleError, result.error);
        }
    }

    if (error) {
        const _error = new Error(error.message);
        yield call(handleError, _error);
    }
}

if (result.error) {
    yield call(handleError, result.error);
}
}

export function* listenForAddCard(): SagaIterator {
    yield takeLatest(ADD_CARD.TRIGGER, addCardSaga);
}

```

Файл get-payment-methods.saga.ts

```

import { SagaIterator } from 'redux-saga';
import { call, put, takeLatest } from 'redux-saga/effects';

import { logger } from '../../../../../core/utils/logger.utils';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { paymentsAPI } from '../data/api/payments-api.data';
import { GET_PAYMENT_METHODS } from '../data/store/payments.actions';
import { GetPaymentMethodsResponse } from '../model/payments-response.model';

export function* getPaymentMethodsSaga(): SagaIterator {
    yield put(GET_PAYMENT_METHODS.STARTED());
}

```

```

const result: GetPaymentMethodsResponse = yield call(paymentsAPI.getPaymentMethods);

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

if (result.error) {
    yield call(logger.log, result.error);
    yield call(toastService.showError, result.error);
    yield call(GET_PAYMENT_METHODS.FAILED, result.error);
}
}

export function* listenForGetPaymentMethods(): SagaIterator {
    yield takeLatest(GET_PAYMENT_METHODS.TRIGGER, getPaymentMethodsSaga);
}

```

Файл pay.saga.ts

```

import { confirmPayment } from '@stripe/stripe-react-native';
import { ConfirmPaymentResult, PaymentIntents } from '@stripe/stripe-react-native/src/types/index';
import { SagaIterator } from 'redux-saga';
import { call, put, select, takeLatest } from 'redux-saga/effects';

import { logger } from '../../../../../core/utils/logger.utils';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { paymentsAPI } from '../data/api/payments-api.data';
import { PAY } from '../data/store/payments.actions';
import { getDefaultPaymentMethod } from '../data/store/payments.selectors';
import { PaymentMethodType } from '../model/method-type.model';
import { PaymentMethod } from '../model/payment-method.model';
import { CreatePaymentIntentResponse } from '../model/payments-response.model';

function* handleError(error: Error): SagaIterator {
    yield call(toastService.showError, error);
    yield call(logger.log, error);
    yield put(PAY.FAILED(error));
}

export function* paySaga(action: ReturnType<typeof PAY.TRIGGER>): SagaIterator {
    yield put(PAY.STARTED());

    const defaultMethod: PaymentMethod = yield select(getDefaultPaymentMethod);
    let paymentId = (Math.random() + 1).toString(36).substring(2);

    if (defaultMethod.type === PaymentMethodType.Card) {
        const result: CreatePaymentIntentResponse = yield call(paymentsAPI.createPaymentIntent, {
            requestThreeDSecure: 'automatic',
            bynAmount: action.payload.amount,
        });

        if (result.data) {
            const secret = result.data.clientSecret;

            if (defaultMethod.type === PaymentMethodType.Card) {
                const { error, paymentIntent }: ConfirmPaymentResult = yield call(confirmPayment, secret, {
                    type: 'Card',
                    // eslint-disable-next-line
                    paymentMethodId: defaultMethod.details!.stripePaymentId,
                });

                if (error) {
                    yield call(handleError, new Error(error.message));
                    return;
                }

                if (paymentIntent?.status !== PaymentIntents.Status.Succeeded) {
                    yield call(logger.log, paymentIntent);
                    return;
                }

                paymentId = paymentIntent.id;
            }
        }
    }
}

```

```

        if (result.error) {
            yield call(handleError, result.error);
        }
    }

    const _result = yield call(paymentsAPI.paymentFinished, {
        methodId: defaultMethod.id,
        amount: action.payload.amount,
        rideId: action.payload.rideId, // Mock
        paymentId,
    });

    if (_result.status >= 200 && _result.status < 300) {
        yield put(PAY.COMPLETED());
        yield call(toastService.showSuccess, 'Заказ оплачен', 'Спасибо, что Вы с нами!');
    }

    if (_result.error) {
        yield call(handleError, _result.error);
    }
}

export function* listenForPay(): SagaIterator {
    yield takeLatest(PAY.TRIGGER, paySaga);
}

```

Файл remove-payment-method.saga.ts

```

import { SagaIterator } from 'redux-saga';
import { call, put, takeLatest } from 'redux-saga/effects';

import { logger } from '../../../../../core/utils/logger.utils';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { paymentsAPI } from '../data/api/payments-api.data';
import { REMOVE_PAYMENT_METHOD } from '../data/store/payments.actions';
import { RemovePaymentMethodResponse } from '../model/payments-response.model';

import { getPaymentMethodsSaga } from './get-payment-methods.saga';

export function* removePaymentMethodSaga(action: ReturnType<typeof
REMOVE_PAYMENT_METHOD.TRIGGER>): SagaIterator {
    yield put(REMOVE_PAYMENT_METHOD.STARTED());

    const result: RemovePaymentMethodResponse = yield call(paymentsAPI.removePaymentMethod,
action.payload);

    if (result.status >= 200 && result.status < 300) {
        yield put(REMOVE_PAYMENT_METHOD.COMPLETED());
        yield call(getPaymentMethodsSaga);
    }

    if (result.error) {
        yield call(logger.log, result.error);
        yield call(toastService.showError, result.error);
        yield call(REMOVE_PAYMENT_METHOD.FAILED, result.error);
    }
}

export function* listenForRemovePaymentMethod(): SagaIterator {
    yield takeLatest(REMOVE_PAYMENT_METHOD.TRIGGER, removePaymentMethodSaga);
}

```

Файл set-as-default.saga.ts

```

import { SagaIterator } from 'redux-saga';
import { call, put, takeLatest } from 'redux-saga/effects';

import { logger } from '../../../../../core/utils/logger.utils';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { paymentsAPI } from '../data/api/payments-api.data';
import { SET_AS_DEFAULT_PAYMENT_METHOD } from '../data/store/payments.actions';
import { SetAsDefaultMethodResponse } from '../model/payments-response.model';

import { getPaymentMethodsSaga } from './get-payment-methods.saga';

```

```

export function* setAsDefaultPaymentMethodSaga(
  action: ReturnType<typeof SET_AS_DEFAULT_PAYMENT_METHOD.TRIGGER>
): SagaIterator {
  yield put(SET_AS_DEFAULT_PAYMENT_METHOD.STARTED());

  const result: SetAsDefaultMethodResponse = yield call(paymentsAPI.setAsDefaultMethod,
  action.payload);

  if (result.status >= 200 && result.status < 300) {
    yield put(SET_AS_DEFAULT_PAYMENT_METHOD.COMPLETED());
    yield call(getPaymentMethodsSaga);
  }

  if (result.error) {
    yield call(logger.log, result.error);
    yield call(toastService.showError, result.error);
    yield call(SET_AS_DEFAULT_PAYMENT_METHOD.FAILED, result.error);
  }
}

export function* listenForSetAsDefaultPaymentMethod(): SagaIterator {
  yield takeLatest(SET_AS_DEFAULT_PAYMENT_METHOD.TRIGGER, setAsDefaultPaymentMethodSaga);
}

```

Файл add-card.component.tsx

```

import { CardField, CardFieldInput } from '@stripe/stripe-react-native';
import React, { useCallback, useRef } from 'react';
import { Dimensions, Keyboard, KeyboardAvoidingView } from 'react-native';
import { useDispatch, useSelector } from 'react-redux';

import { getExistingUser } from '../../../../../core/data/store/user.selectors';
import { SafeBackground } from '../../../../../core/presentation/background/background.styled';
import { Button } from '../../../../../core/presentation/button/button.component';
import { Header } from '../../../../../core/presentation/header/header.component';
import { SpacerPressable } from '../../../../../core/presentation/spacer/spacer.styled';
import { useNavigationBlocker } from '../../../../../core/utils/hooks/use-navigation-blocker.utils';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { colors } from '../../../../../core/utils/theme/colors.utils';
import { FontType, getFontName } from '../../../../../core/utils/theme/fonts.utils';
import { defaultTheme } from '../../../../../core/utils/theme/themes.utils';
import { ADD_CARD } from '../../../../../data/store/payments.actions';
import { getIsCardAdding } from '../../../../../data/store/payments.selectors';
import { toValidCardBrand, toValidExp } from '../../../../../utils/formatters.utils';

export const AddCard: React.FC = () => {
  const isLoading = useSelector(getIsCardAdding);
  const dispatch = useDispatch();
  const cardData = useRef<CardFieldInput.Details>();
  const user = useSelector(getExistingUser);

  const handleAddPress = useCallback(() => {
    if (
      cardData.current?.validCVC !== CardFieldInput.ValidationState.Valid ||
      cardData.current?.validNumber !== CardFieldInput.ValidationState.Valid ||
      cardData.current?.validExpiryDate !== CardFieldInput.ValidationState.Valid
    ) {
      return;
    }

    const data = cardData.current as CardFieldInput.Details;
    const brand = toValidCardBrand(data.brand);

    if (!brand) {
      toastService.showError(new Error(`Мы не работаем с картами ${data.brand}`));
      return;
    }

    dispatch(
      ADD_CARD.TRIGGER({
        lastFour: data.last4,
        exp: toValidExp(data.expiryMonth, data.expiryYear),
        brand: brand,
        holder: `${user.firstName.toUpperCase()} ${user.lastName.toUpperCase()}`,
      })
    );
  }, [dispatch, user]);
}

```

```

useNavigationBlocker(isLoading);

return (
  <SafeBackground>
    <Header title={'Добавить карту'} />
    <KeyboardAvoidingView style={{ flex: 1 }} behavior={'padding'}>
      <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)}
      />
      <SpacerPressable onPress={Keyboard.dismiss} />
      <Button isLoading={isLoading} title={'Добавить'} onPress={handleAddPress} />
    </KeyboardAvoidingView>
  </SafeBackground>
);
);

```

Файл payments-list.component.tsx

```

import { StackNavigationProp } from '@react-navigation/stack';
import React, { useCallback, useEffect } from 'react';
import { RefreshControl, ScrollView } from 'react-native';
import { useDispatch, useSelector } from 'react-redux';

import { SafeBackground } from '../../../../../core/presentation/background/background.styled';
import { Button } from '../../../../../core/presentation/button/button.component';
import { Header } from '../../../../../core/presentation/header/header.component';
import { defaultTheme } from '../../../../../core/utils/theme/themes.utils';
import { screens } from '../../../../../navigation/utils/screens';
import { GET_PAYMENT_METHODS } from '../../../../../data/store/payments.actions';
import { getIsPaymentsListLoading, getPaymentMethodList } from '../../../../../data/store/payments.selectors';
import { PaymentMethodComponent } from '../components/payment-method/payment-method.component';
import { width } from '../components/payment-method/payment-method.styled';

export const PaymentsList = ({ navigation }: { navigation: StackNavigationProp<never> }) => {
  const isLoading = useSelector(getIsPaymentsListLoading);
  const methods = useSelector(getPaymentMethodList) ?? [];
  const dispatch = useDispatch();

  const getPaymentsList = useCallback(() => {
    dispatch(GET_PAYMENT_METHODS.TRIGGER());
  }, [dispatch]);

  useEffect(() => {
    getPaymentsList();
    // eslint-disable-next-line
  }, []);

  const onAddCardPress = useCallback(() => {
    navigation.navigate(screens.main.payments.addCard as never);
  }, [navigation]);

  return (
    <SafeBackground edges={['top']}>
      <Header title={'Способы оплаты'} />
      <ScrollView
        refreshControl={<RefreshControl refreshing={isLoading} onRefresh={getPaymentsList} />}
        contentContainerStyle={{ alignItems: 'center' }}
      >
        {[...methods].reverse().map((method) => (
          <PaymentMethodComponent method={method} key={method.id.toString()} />
        ))}
        <Button
          title={'Добавить карту'}
          onPress={onAddCardPress}
        </Button>
      
```

```

        style={{ width: width - defaultTheme.spacer * 4 }}
    />
  </ScrollView>
<SafeBackground>
);
};

```

Файл profile-api.data.ts

```

import { restGatewayAPI } from '../../../../../core/data/api/rest-gateway-api.data';
import { UserResponse } from '../../../../../model/user-response.model';

export class ProfileAPI {
  getUser = async (): Promise<UserResponse> => {
    return restGatewayAPI.get('/api/v1/user');
  };
}

export const profileAPI = new ProfileAPI();

```

Файл get-user.saga.ts

```

import { SagaIterator } from 'redux-saga';
import { call, put, takeLatest } from 'redux-saga/effects';

import { logger } from '../../../../../core/utils/logger.utils';
import { toastService } from '../../../../../core/utils/services/toast-service.utils';
import { profileAPI } from '../data/api/profile-api.data';
import { GET_USER } from '../data/store/profile.actions';

export function* getUserSaga(): SagaIterator {
  yield put(GET_USER.STARTED());
  const result = yield call(profileAPI.getUser);
  if (result.data) {
    yield put(GET_USER.COMPLETED(result.data));
  }
  if (result.error) {
    yield call(logger.log, result.error);
    yield call(toastService.showError, result.error);
    yield put(GET_USER.FAILED(result.error));
  }
}

export function* listenFor GetUser(): SagaIterator {
  yield takeLatest(GET_USER.TRIGGER, getUserSaga);
}

```

Файл profile.component.tsx

```

import React, { useCallback, useEffect } from 'react';
import { RefreshControl, ScrollView } from 'react-native';
import { useDispatch, useSelector } from 'react-redux';

import { getExistingUser, getIsUserLoading } from '../../../../../core/data/store/user.selectors';
import { Avatar } from '../../../../../core/presentation/avatar/avatar.styled';
import { SafeBackground } from '../../../../../core/presentation/background/background.styled';
import { Header } from '../../../../../core/presentation/header/header.component';
import { toReadableCarClass } from '../../../../../core/utils/formatters/car-class.utils';
import { toReadableGender } from '../../../../../core/utils/formatters/gender.utils';
import { mockedAvatar } from '../../../../../mocks/users';
import { GET_USER } from '../data/store/profile.actions';

import { Name, Title, CommonText } from './components/profile-text/profile-text.styled';
import { ProfileTileWrapper, DataWrapper } from './components/profile-tile/profile-tile.styled';

export const ProfileScreen: React.FC = () => {
  const user = useSelector(getExistingUser);
  const dispatch = useDispatch();
  const isLoading = useSelector(getIsUserLoading);

  const getUser = useCallback(() => {
    if (user.driver) {

```

```

        // To update amount of money
        dispatch(GET_USER.TRIGGER());
    }
    // eslint-disable-next-line
}, [dispatch]);

useEffect(() => {
    getUser();
    // eslint-disable-next-line
}, []);

if (!user) {
    return null;
}

return (
    <SafeBackground edges={['top']}>
        <Header title={'Профиль'} />
        <ScrollView
            refreshControl={<RefreshControl refreshing={isLoading} onRefresh={getUser} />}
            contentContainerStyle={{ alignItems: 'center' }}
        >
            <Avatar source={mockedAvatar} style={{ width: 120, height: 120, borderRadius: 60 }} />
            <Name>
                {user.firstName} {user.lastName}
            </Name>
            <ProfileTileWrapper>
                <Title>Персональная информация</Title>
                <DataWrapper>
                    <CommonText>Электронная почта: {user.email}</CommonText>
                    <CommonText>Телефон: {user.phone}</CommonText>
                    <CommonText>Пол: {toReadableGender(user.gender)}</CommonText>
                </DataWrapper>
            </ProfileTileWrapper>
            {!user.driver && (
                <ProfileTileWrapper>
                    <Title>Водительская информация</Title>
                    <DataWrapper>
                        <CommonText>Баланс: {user.driver.balance} рублей</CommonText>
                        <CommonText>Машина: {user.driver.carBrand}</CommonText>
                        <CommonText>Класс: {toReadableCarClass(user.driver.carClass)}</CommonText>
                    </DataWrapper>
                </ProfileTileWrapper>
            )}
        </ScrollView>
    </SafeBackground>
);
};

```

Файл directions.integration.ts

```

import { Injectable } from '@nestjs/common';
import { Location } from '../../../../../utils/types/location.types';
import { GoogleMaps } from '../../../../google-maps/google-maps.integration';
import { Language } from '@googlemaps/google-maps-services-js';

@Injectable()
export class Directions {
    private mockDirections = process.env.MOCK_DIRECTIONS?.toLowerCase() === 'true' ||
    !process.env.MOCK_GEOCODING;

    constructor(private maps: GoogleMaps) {}

    async getDirection(
        from: Location,
        to: Location
    ): Promise<{
        minutes: number;
        route: Array<Location>;
    }> {
        const result = await this.maps.client.directions({
            params: {
                key: this.maps.mapsKey,
                origin: from,
                destination: to,
            }
        });
        return result;
    }
}

```

```

        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,
    })),
};

}
}
}

```

Файл geocoding.integration.ts

```

import { Injectable } from '@nestjs/common';
import { ExtendedLocation } from '../../../../../utils/types/location.types';
import { Language } from '@googlemaps/google-maps-services-js';
import { Location } from '../../../../../utils/types/location.types';
import { places } from '../../../../../mocks/places';
import { GeoUtils } from '../../../../../utils/geo-utils.utils';
import { GoogleMaps } from '../google-maps/google-maps.integration';

@Injectable()
export class Geocoding {
    private mockGeocoding = process.env.MOCK_GEOCODING?.toLowerCase() === 'true' ||
    !process.env.MOCK_GEOCODING;

    constructor(private geoUtils: GeoUtils, private maps: GoogleMaps) {}

    async decode(location: Location): Promise<ExtendedLocation> {
        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()} для экономии ;р`,
        };
    } 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',
        };
    }
}

```

Файл google-maps.integration.ts

```

import { Injectable } from '@nestjs/common';
import { Client, LatLngLiteral, LatLngLiteralVerbose } from '@googlemaps/google-maps-services-js';

```

```

import axios from 'axios';
import {
    PlaceAutocompleteRequest,
    PlaceAutocompleteResponse,
} from '@googlemaps/google-maps-services-js/dist/places/autocomplete';
import { DirectionsRequest, DirectionsResponse } from '@googlemaps/google-maps-services-
js/dist/directions';
import {
    ReverseGeocodeRequest,
    ReverseGeocodeResponse,
} from '@googlemaps/google-maps-services-js/dist/geocode/reversegeocode';

// eslint-disable-next-line @typescript-eslint/no-var-requires
const GraphHopperRouting = require('graphhopper-js-api-client/src/GraphHopperRouting');
// eslint-disable-next-line @typescript-eslint/no-var-requires
const GraphHopperGeocoding = require('graphhopper-js-api-client/src/GraphHopperGeocoding');
// eslint-disable-next-line @typescript-eslint/no-var-requires
const GHInput = require('graphhopper-js-api-client/src/GHInput');

@Injectable()
export class GoogleMaps {
    readonly mapsKey = process.env.DIRECTIONS_GATEWAY_API_KEY || '';
    private readonly useGraphHopper =
        process.env.USE_GRAPHHOPPER?.toLowerCase() === 'true' || !process.env.MOCK_GEOCODING;
    readonly client: Client;

    constructor() {
        if (this.useGraphHopper) {
            const ghRouting = new GraphHopperRouting({
                key: this.mapsKey,
                vehicle: 'car',
                elevation: false,
            }) as {
                doRequest: () => Promise<{
                    paths: Array<any>;
                }>;
                addPoint: (point: typeof GHInput) => void;
                clearPoints: () => void;
            };
            const ghGeocoding = new GraphHopperGeocoding({
                key: this.mapsKey,
            }) as {
                doRequest: ({{
                    locale,
                    query,
                    point,
                }}: {
                    locale: string;
                    query?: string;
                    point?: typeof GHInput;
                }) => Promise<{
                    hits: Array<{
                        street: string;
                        housenumber: string;
                        city: string;
                        country: string;
                        name?: string;
                        point: LatLngLiteral;
                    }>;
                }>;
            };
            this.client = {
                // eslint-disable-next-line @typescript-eslint/ban-ts-comment
                // @ts-ignore
                placeAutocomplete: async (request: PlaceAutocompleteRequest): Promise<PlaceAutocompleteResponse> => {
                    const result = await ghGeocoding.doRequest({
                        locale: 'ru',
                        query: request.params.input,
                    });

                    return {
                        data: {
                            predictions: result.hits.map((hit) => ({
                                description: hit.name ?? `${hit.street}, ${hit.housenumber}`,
                                location: hit.point,
                            })),
                        },
                    };
                },
            };
        }
    }
}

```

Файл places.integration.ts

```
import { Injectable } from '@nestjs/common';
import { ExtendedLocation } from '../../utils/types/location.types';
import { places } from '../../../../../mocks/places';
import { GoogleMaps } from '../google-maps/google-maps.integration';
import { Language } from '@googlemaps/google-maps-services-js';

@Injectable()
export class Places {
    private mockPlaces = process.env.MOCK_PLACES?.toLowerCase() === 'true' ||
!process.env.MOCK_PLACES;

    constructor(private maps: GoogleMaps) {}
}
```

```

        async search(part: string): Promise<Array<ExtendedLocation>> {
            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,
                }));
            }
        }
    }
}

```

Файл stripe.integration.ts

```

import { Injectable } from '@nestjs/common';
import StripeSDK from 'stripe';

export type CreateIntentOutput = {
    customerId: string;
    intent: StripeSDK.PaymentIntent;
};

@Injectable()
export class Stripe {
    private stripeSecretKey = process.env.STRIPE_SECRET_KEY || '';
    private usdRate = process.env.USD_RATE ? Number(process.env.USD_RATE) : 3;
    private stripe = new StripeSDK(this.stripeSecretKey as string, {
        apiVersion: '2020-08-27',
        typescript: true,
    });

    async createIntent(
        email: string,
        bynAmount: number,
        requestThreeDSecure,
        customerId?: string
    ): Promise<CreateIntentOutput> {
        const customer = customerId ? { id: customerId } : await this.stripe.customers.create({
            email
        });

        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);

        return {
            intent: paymentIntent,
            customerId: customer.id,
        };
    }
}

```

Файл places.ts

```
import { ExtendedLocation } from '../utils/types/location.types';
```

```

export const places: Array<ExtendedLocation> = [
  {
    latitude: 53.8874276,
    longitude: 27.5425487,
    readableLocation: 'БЦ Титул, Минск, Беларусь',
  },
  {
    latitude: 55.7622171,
    longitude: 37.613645,
    readableLocation: 'KAIF Provenance, Москва, Россия',
  },
  {
    latitude: 53.911848,
    longitude: 27.5927425,
    readableLocation: 'Гикало 9, Минск, Беларусь',
  },
  {
    latitude: 53.5912576,
    longitude: 27.8535574,
    readableLocation: 'Аптека 27, Руденск, Беларусь',
  },
  {
    latitude: 41.6524898,
    longitude: 41.6335758,
    readableLocation: 'Пляж Батуми, Батуми, Грузия',
  },
  {
    latitude: 41.6696123,
    longitude: 44.9621603,
    readableLocation: 'Аэропорт, Тбилиси, Грузия',
  },
  {
    latitude: 42.4758754,
    longitude: 44.4773676,
    readableLocation: 'Гудаури, Грузия',
  },
];

```

Файл auth-data.model.ts

```

import { Column, Entity, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class AuthData {
  @PrimaryGeneratedColumn()
  id: number;

  @Column({ unique: true })
  email: string;

  @Column()
  passwordHash: string;
}

```

Файл driver.model.ts

```

import { CarClass } from '../utils/types/car-class.types';
import { Column, Entity, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class Driver {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  carBrand: string;

  @Column()
  carClass: CarClass;

  @Column({ type: 'float', default: 0.0 })
  balance: number;
}

```

Файл payment.model.ts

```
import { Column, Entity, ManyToOne, PrimaryGeneratedColumn } from 'typeorm';
import { User } from './user.model';
import { PaymentMethod } from './payment-method.model';

@Entity()
export class Payment {
    @PrimaryGeneratedColumn()
    id: number;

    @ManyToOne(() => User)
    user: User;

    @ManyToOne(() => PaymentMethod)
    method: PaymentMethod;

    @Column({ type: 'float', default: 0.0 })
    amount: number;

    @Column({ type: 'bigint' })
    timestamp: number;
}
```

Файл payment-method.model.ts

```
import { Column, Entity, JoinColumn, ManyToOne, OneToOne, PrimaryGeneratedColumn } from
'typeorm';
import { PaymentMethodType } from '../utils/types/payment-method-type.types';
import { PaymentMethodDetails } from './payment-method-details.model';
import { User } from './user.model';

@Entity()
export class PaymentMethod {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    type: PaymentMethodType;

    @Column()
    isDefault: boolean;

    @Column({ default: true })
    isVisible: boolean;

    @OneToOne(() => PaymentMethodDetails, { nullable: true, onDelete: 'CASCADE' })
    @JoinColumn()
    details?: PaymentMethodDetails;

    @ManyToOne(() => User)
    user: User;
}
```

Файл payment-method-details.model.ts

```
import { Column, Entity, PrimaryGeneratedColumn } from 'typeorm';
import { CardBrand } from '../utils/types/card-brand.types';

@Entity()
export class PaymentMethodDetails {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    lastFour: string;

    @Column()
    exp: string;

    @Column()
    holder: string;

    @Column()
}
```

```

brand: CardBrand;

@Column({ unique: true })
stripePaymentId: string;
}

```

Файл ride.model.ts

```

import { Column, Entity, JoinColumn, ManyToOne, OneToOne, PrimaryGeneratedColumn } from
'typeorm';
import { User } from './user.model';
import { Payment } from './payment.model';
import { RideStatus } from '../utils/types/ride-status.types';

@Entity()
export class Ride {
    @PrimaryGeneratedColumn()
    id: number;

    @ManyToOne(() => User)
    client: User;

    @ManyToOne(() => User)
    driver: User;

    @OneToOne(() => Payment, { nullable: true })
    @JoinColumn()
    payment?: Payment;

    @Column({ type: 'float', default: 0.0 })
    cost: number;

    @Column({ nullable: true, type: 'bigint' })
    startTime?: number;

    @Column({ nullable: true, type: 'bigint' })
    endTime?: number;

    @Column()
    to: string;

    @Column()
    from: string;

    @Column()
    status: RideStatus;

    @Column({ default: false })
    paid: boolean;
}

```

Файл user.model.ts

```

import { Column, Entity, JoinColumn, OneToOne, PrimaryGeneratedColumn } from 'typeorm';
import { Gender } from '../utils/types/gender.types';
import { Driver } from './driver.model';

@Entity()
export class User {
    @PrimaryGeneratedColumn()
    id: number;

    @Column({ unique: true })
    email: string;

    @Column()
    phone: string;

    @Column()
    firstName: string;

    @Column()
    lastName: string;

    @Column()
    gender: Gender;
}

```

```

@Column({ nullable: true })
stripeClientId?: string;

@OneToOne(() => Driver)
@JoinColumn()
driver: Driver;
}

```

Файл jwt.guard.ts

```

import { ExecutionContext, Injectable, UnauthorizedException } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';
import { UserRepository } from '../repository/user.repository';

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
    constructor(private userRepository: UserRepository) {
        super();
    }

    canActivate(context: ExecutionContext) {
        // Add your custom authentication logic here
        // for example, call super.logIn(request) to establish a session.
        return super.canActivate(context);
    }

    handleRequest(err, user) {
        // You can throw an exception based on either "info" or "err" arguments
        if (err || !user) {
            throw err || new UnauthorizedException();
        }

        return user;
    }
}

```

Файл jwt.provider.ts

```

import { Injectable } from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
import { GenerateTokenInput } from '../dto/input/auth-dto.input';
import { AuthOutput } from '../dto/output/auth-dto.output';

@Injectable()
export class TokenProvider {
    constructor(private jwtService: JwtService) {}

    decode(token: string): GenerateTokenInput | string {
        const data = this.jwtService.decode(token);

        return typeof data === 'string'
            ? data
            : {
                email: data.email,
                id: data.id,
            };
    }

    async generateToken(data: GenerateTokenInput): Promise<AuthOutput> {
        return {
            token: this.jwtService.sign(data),
            refreshToken: this.jwtService.sign(data, {
                expiresIn: '640000s',
            }),
        };
    }
}

```

Файл jwt.strategy.ts

```

import { ExtractJwt, Strategy } from 'passport-jwt';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable } from '@nestjs/common';
import { GenerateTokenInput } from '../dto/input/auth-dto.input';

```

```

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: process.env.JWT_SECRET,
    });
  }

  async validate(payload: any): Promise<GenerateTokenInput> {
    return { id: payload.id, email: payload.email };
  }
}

```

Файл auth.service.ts

```

import { Injectable, UnauthorizedException } from '@nestjs/common';
import { AuthDataRepository } from '../repository/auth-data.repository';
import { AuthOutput } from '../dto/output/auth-dto.output';
import { GenerateTokenInput, RefreshTokensInput, RegisterInput } from '../dto/input/auth-dto.input';
import { TokenProvider } from '../security/jwt.provider';
import { UserService } from './user.service';
import { AlreadyExistsException } from '../utils/exceptions/already-exists.exception';
import { AuthData } from '../model/auth-data.model';
import { Hasher } from '../utils/hasher.utils';

@Injectable()
export class AuthService {
  constructor(
    private authDataRepository: AuthDataRepository,
    private tokenProvider: TokenProvider,
    private userService: UserService,
    private hasher: Hasher
  ) {}

  async login(email: string, password: string): Promise<AuthOutput> {
    const existingData = await this.authDataRepository.findOne({ email });

    if (!existingData) {
      throw new AlreadyExistsException('Данная почта не зарегистрирована');
    }

    const isPasswordValid = await this.hasher.compare(password, existingData.passwordHash);

    if (!isPasswordValid) {
      throw new AlreadyExistsException('Неверный пароль');
    }

    return this.tokenProvider.generateToken({ email, id: existingData.id });
  }

  async register(data: RegisterInput): Promise<AuthOutput> {
    const existingData = await this.authDataRepository.findOne({ email: data.email });

    if (existingData) {
      throw new AlreadyExistsException('Данная почта уже используется');
    }

    const authData = new AuthData();

    authData.email = data.email;
    authData.passwordHash = await this.hasher.generateHash(data.password);

    await this.authDataRepository.save(authData);

    const user = await this.userService.createUser(data);

    return this.tokenProvider.generateToken({ email: user.email, id: user.id });
  }

  getUserFromToken(token: string): GenerateTokenInput | string {
    return this.tokenProvider.decode(token);
  }
}

```

```

    async refreshTokens(data: RefreshTokensInput): Promise<AuthOutput> {
        const user = this.getUserFromToken(data.refreshToken);

        if (typeof user !== 'object') {
            throw new UnauthorizedException();
        }

        return this.tokenProvider.generateToken(user as GenerateTokenInput);
    }
}

```

Файл maps.service.ts

```

import { Injectable } from '@nestjs/common';
import { ExtendedLocation } from '../utils/types/location.types';
import { Geocoding } from '../integration/geocoding/geocoding.integration';
import { Location } from '../utils/types/location.types';
import { Places } from '../integration/places/places.integration';
import { Directions } from '../integration/directions/directions.integration';
import { DirectionsOutput } from '../dto/output/map-dto.output';
import { CarClass } from '../utils/types/car-class.types';

@Injectable()
export class MapsService {
    constructor(private geocoding: Geocoding, private places: Places, private directions: Directions) {}

    async decode(data: Location): Promise<ExtendedLocation> {
        return this.geocoding.decode(data);
    }

    async searchPlaces(toSearch: string): Promise<Array<ExtendedLocation>> {
        return this.places.search(toSearch);
    }

    async getDirection(from: Location, to: Location): Promise<DirectionsOutput> {
        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,
            },
        };
    }
}

```

Файл payment.service.ts

```

import { Injectable } from '@nestjs/common';
import { PaymentMethodRepository } from '../repository/payment-method.repository';
import { PaymentMethod } from '../model/payment-method.model';
import { PaymentMethodType } from '../utils/types/payment-method-type.types';
import { PaymentMethodDetails } from '../model/payment-method-details.model';
import { PaymentMethodDetailsRepository } from '../repository/payment-method-details.repository';
import { User } from '../model/user.model';
import { AddCardInput, ConfirmPaymentInput, CreatePaymentIntentInput } from '../dto/input/payment-dto.input';
import { UserRepository } from '../repository/user.repository';
import { PaymentRepository } from '../repository/payment.repository';
import { RideRepository } from '../repository/ride.repository';
import { Payment } from '../model/payment.model';
import { CreatePaymentIntentOutput } from '../dto/output/payment-dto.output';
import { Stripe } from '../integration/stripe/stripe.integration';

@Injectable()
export class PaymentService {
    constructor(
        private paymentRepository: PaymentRepository,
        private paymentMethodRepository: PaymentMethodRepository,
        private paymentMethodDetailsRepository: PaymentMethodDetailsRepository,
        private userRepository: UserRepository,
    )
}

```

```

    private rideRepository: RideRepository,
    private stripe: Stripe
) {}

async createPaymentMethod(user: User, type: PaymentMethodType): Promise<PaymentMethod> {
    const paymentMethod = new PaymentMethod();

    paymentMethod.isDefault = type === PaymentMethodType.Cash;
    paymentMethod.type = type;
    paymentMethod.user = user;

    await this.paymentMethodRepository.save(paymentMethod);

    return paymentMethod;
}

async getPaymentMethods(id: number): Promise<Array<PaymentMethod>> {
    return this.paymentMethodRepository.find({ where: [{ user: { id }, isVisible: true }], relations: ['details'] });
}

async setDefaultMethod(userId: number, methodId: number) {
    const methods = await this.paymentMethodRepository.find({ where: [{ user: { id: userId } }] });

    methods.forEach((method) => (method.isDefault = method.id === methodId));

    await this.paymentMethodRepository.save(methods);
}

async addCard(userId: number, cardData: AddCardInput) {
    const user = await this.userRepository.findOneOrFail({ id: userId });

    const details = new PaymentMethodDetails();

    details.exp = cardData.exp;
    details.stripePaymentId = cardData.stripePaymentId;
    details.brand = cardData.brand;
    details.holder = cardData.holder;
    details.lastFour = cardData.lastFour;

    await this.paymentMethodDetailsRepository.save(details);

    const card = await this.createPaymentMethod(user, PaymentMethodType.Card);

    card.details = details;

    await this.paymentMethodRepository.save(card);
}

async removeMethod(userId: number, methodId: number) {
    const method = await this.paymentMethodRepository.findOneOrFail({
        where: [{ id: methodId }],
        relations: ['details'],
    });

    if (method.type === PaymentMethodType.Cash) return;

    if (method.isDefault) {
        const cash = await this.paymentMethodRepository.findOneOrFail({
            user: { id: userId },
            type: PaymentMethodType.Cash,
        });

        cash.isDefault = true;

        await this.paymentMethodRepository.save(cash);
    }

    method.isVisible = false;
    await this.paymentMethodRepository.save(method);
}

async confirmPayment(userId: number, paymentData: ConfirmPaymentInput) {
    const user = await this.userRepository.findOneOrFail({ id: userId });
    const paymentMethod = await this.paymentMethodRepository.findOneOrFail({
        id: paymentData.methodId,
        user: { id: userId },
    });
}

```

```

    });

    const ride = await this.rideRepository.findOneOrFail({ id: paymentData.rideId });
    const payment = new Payment();

    payment.method = paymentMethod;
    payment.amount = paymentData.amount;
    payment.timestamp = Date.now();
    payment.user = user;

    await this.paymentRepository.save(payment);

    ride.payment = payment;
    ride.paid = true;

    await this.rideRepository.save(ride);
}

async createPaymentIntent(userId: number, data: CreatePaymentIntentInput): Promise<CreatePaymentIntentOutput> {
    const user = await this.userRepository.findOneOrFail({ id: userId });

    const result = await this.stripe.createIntent(
        user.email,
        data.bynAmount,
        data.requestThreeDSecure,
        user.stripeClientId
    );

    if (result.customerId !== user.stripeClientId) {
        user.stripeClientId = result.customerId;

        await this.userRepository.save(user);
    }

    return {
        clientSecret: result.intent.client_secret,
    };
}
}
}

```

Файл rtc.service.ts

```

import { Injectable } from '@nestjs/common';
import { TokenProvider } from '../security/jwt.provider';
import { Socket } from 'socket.io';
import { GenerateTokenInput } from '../dto/input/auth-dto.input';
import { UserService } from './user.service';
import { User } from '../model/user.model';
import { Location } from '../utils/types/location.types';
import { DriverSocketUserInfo, SocketUserInfo, WithUserRole, WSMimeType } from
    '../utils/types/rtc-events.types';
import { RideStatus } from '../utils/types/ride-status.types';
import { GeoUtils } from '../utils/geo-utils.utils';
import { waitForSocketResponse } from '../utils/awaitable-socket.utils';
import { logger } from '../utils/logger.utils';
import { ExtendedRideRequest } from '../utils/types/ride-request.types';
import { RideRepository } from '../repository/ride.repository';
import { Ride } from '../model/ride.model';
import { Directions } from '../integration/directions/directions.integration';

@Injectable()
export class RTCService {
    private readonly idBasedSocketHolder: Map<number, Socket>;
    private readonly socketBasedIdHolder: WeakMap<Socket, number>;
    private readonly clientDataHolder: Map<number, SocketUserInfo>;
    private readonly driverDataHolder: Map<number, DriverSocketUserInfo>;

    constructor(
        private tokenProvider: TokenProvider,
        private userService: UserService,
        private geoUtils: GeoUtils,
        private rideRepository: RideRepository,
        private directions: Directions
    ) {
        this.clientDataHolder = new Map<number, SocketUserInfo>();
        this.driverDataHolder = new Map<number, DriverSocketUserInfo>();
        this.idBasedSocketHolder = new Map<number, Socket>();
    }
}

```

```

        this.socketBasedIdHolder = new WeakMap<Socket, number>();
    }

    private async getUserFromSocket(socket: Socket): Promise<User> {
        const token = (socket.handshake.auth?.Authorization as string)?.split('Bearer ')[1];

        if (!token) {
            socket.disconnect();
            return;
        }

        const data = this.tokenProvider.decode(token) as GenerateTokenInput;

        if (!data.id) {
            socket.disconnect();
            return;
        }

        return this.userService.getUser(data.id);
    }

    async addUser(socket: Socket) {
        const user = await this.getUserFromSocket(socket);

        if (user.driver) {
            if (this.driverDataHolder.has(user.id)) {
                await this.handleRestoreState(socket, user,
                    this.driverDataHolder.get(user.id).rideId);
            } else {
                this.driverDataHolder.set(user.id, { carClass: user.driver.carClass });
            }
        } else {
            if (this.clientDataHolder.has(user.id)) {
                await this.handleRestoreState(socket, user,
                    this.clientDataHolder.get(user.id).rideId);
            } else {
                this.clientDataHolder.set(user.id, {});
            }
        }

        this.idBasedSocketHolder.set(user.id, socket);
        this.socketBasedIdHolder.set(socket, user.id);
    }

    async handleRestoreState(socket: Socket, user: User, rideId?: number) {
        if (!rideId) return;

        let request: ExtendedRideRequest;
        let toPickUp: Array<Location> | null = null;
        const ride = await this.rideRepository.findOneOrFail(
            { id: rideId },
            { relations: ['client', 'driver', 'driver.driver'] }
        );

        if (user.driver) {
            request = this.driverDataHolder.get(user.id).rideRequest;
            toPickUp = this.driverDataHolder.get(user.id).toPickUp;
        } else {
            request = this.clientDataHolder.get(user.id).rideRequest;
        }

        setTimeout(() => {
            socket.emit(WSMessageType.RestoreState, {
                ride,
                request,
                toPickUp,
            });
        }, 400);
    }

    async removeUser(socket: Socket) {
        const user = await this.getUserFromSocket(socket);

        this.idBasedSocketHolder.delete(user.id);
        this.socketBasedIdHolder.delete(socket);
    }

    async handleLocationUpdate(location: WithUserRole<Location>, socket: Socket) {

```

```

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(WSMessageType.LocationUpdate, {
        from: userId,
        location,
    });
}
}

async handleRideRequest(request: WithUserRole<ExtendedRideRequest>, socket: Socket) {
    const userId = this.socketBasedIdHolder.get(socket);
    const user = this.clientDataHolder.get(userId);
    let driver: [number, SocketUserInfo] | null = null;
    let driverData: DriverSocketUserInfo | null = null;
    let wasStopped = false;

    await this.handleStopSearch(socket);

    return new Promise(async (resolve, reject) => {
        user.stopSearch = () => {
            reject('Поиск остановлен');
            wasStopped = true;
        };

        const sortedList = this.geoUtils
            .toSortedList(user.location, this.driverDataHolder)
            .filter(
                (value) => (value[1] as DriverSocketUserInfo).carClass === request.data.carClass
                && !value[1].rideId
            );

        for await (const entry of sortedList) {
            const socket = this.idBasedSocketHolder.get(entry[0]);
            const toPickUp = await this.directions.getDirection(entry[1].location,
request.data.from);

            socket.emit(WSMessageType.RideRequest, {
                ...request,
                data: {
                    ...request.data,
                    toPickUp: toPickUp.route,
                },
            });

            try {
                const result = await waitForSocketResponse(
                    socket,
                    20000,
                    WSMessageType.RideTimeout,
                    WSMessageType.RideAccept,
                    WSMessageType.RideDecline
                );
            }

            if (result.event === WSMessageType.RideAccept) {
                driver = entry;
                driverData = this.driverDataHolder.get(driver[0]);

                driverData.toPickUp = toPickUp.route;
                driverData.rideRequest = request.data;

                break;
            }
        } catch (e) {
            logger.log('Next car...');
        }
    });

    user.stopSearch = null;

    if (!driver) {

```

```

        socket.emit(WSMessageType.RideDecline);
        return;
    }

    if (wasStopped) {
        this.idBasedSocketHolder.get(driver[0]).emit(WSMessageType.RideDecline);
        return;
    }

    // Can't stop ride starting after this
    user.stopSearch = null;
    user.companionId = driver[0];
    user.rideRequest = request.data;

    driverData.companionId = userId;

    const dbDriver = await this.userService.getUser(driver[0]);
    const dbClient = await this.userService.getUser(userId);

    const ride = new Ride();

    ride.client = dbClient;
    ride.driver = dbDriver;
    ride.cost = request.data.cost;
    ride.to = request.data.to.readableLocation;
    ride.from = request.data.from.readableLocation;
    ride.status = RideStatus.Starting;
    ride.startTime = Date.now();

    await this.rideRepository.save(ride);

    driverData.rideId = ride.id;
    user.rideId = ride.id;

    socket.emit(WSMessageType.RideAccept, ride);
    this.idBasedSocketHolder.get(driver[0]).emit(WSMessageType.RideAccept, ride);
},

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

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

async handleRideStatusChange(status: WithUserRole<RideStatus>, socket: Socket) {
    const driverId = this.socketBasedIdHolder.get(socket);
    const driverData = this.driverDataHolder.get(driverId);
    const clientData = this.clientDataHolder.get(driverData.companionId);

    const companionId = driverData.companionId;
    const ride = await this.rideRepository.findOneOrFail(driverData.rideId);

    ride.status = status.data;

    if (status.data === RideStatus.Completed) {
        const driver = await this.userService.getUser(driverId);
        driver.driver.balance = driver.driver.balance + ride.cost;
        await this.userService.saveUser(driver);

        driverData.rideId = null;
        driverData.companionId = null;
        driverData.rideRequest = null;
        driverData.toPickUp = null;

        clientData.rideId = null;
        clientData.companionId = null;
        clientData.rideRequest = null;

        ride.endTime = Date.now();
    }

    this.idBasedSocketHolder.get(companionId)?.emit(WSMessageType.RideStatusChange, { status: status.data });

    await this.rideRepository.save(ride);
}

```

```
    }
}
```

Файл user.service.ts

```
import { Injectable } from '@nestjs/common';
import { UserRepository } from '../repository/user.repository';
import { User } from '../model/user.model';
import { RegisterInput } from '../dto/input/auth-dto.input';
import { Driver } from '../model/driver.model';
import { DriverRepository } from '../repository/driver.repository';
import { RideRepository } from '../repository/ride.repository';
import { Ride } from '../model/ride.model';
import { PaymentService } from './payment.service';
import { PaymentMethodType } from '../utils/types/payment-method-type.types';

@Injectable()
export class UserService {
    constructor(
        private userRepository: UserRepository,
        private driverRepository: DriverRepository,
        private rideRepository: RideRepository,
        private paymentService: PaymentService
    ) {}

    async getUser(id: number): Promise<User> {
        return this.userRepository.findOneOrFail({ id }, { relations: ['driver'] });
    }

    async saveUser(user: User) {
        await this.userRepository.save(user);
        await this.driverRepository.save(user.driver);
    }

    async createUser(data: RegisterInput): Promise<User> {
        const user = new User();

        user.email = data.email;
        user.gender = data.gender;
        user.phone = data.phone;
        user.lastName = data.lastName;
        user.firstName = data.firstName;

        if (data.driver) {
            const driver = new Driver();

            driver.carClass = data.driver.carClass;
            driver.carBrand = data.driver.carBrand;
            driver.balance = 0;

            await this.driverRepository.save(driver);

            user.driver = driver;
        }

        await this.userRepository.save(user);

        await this.paymentService.createPaymentMethod(user, PaymentMethodType.Cash);
    }

    return user;
}

async getHistory(id: number): Promise<Array<Ride>> {
    const user = await this.getUser(id);
    const key = user.driver ? 'driver' : 'client';

    return this.rideRepository.find({
        where: [{ [key]: { id } }],
        relations: ['driver', 'client', 'driver.driver'],
    });
}

async getLatestRide(id: number): Promise<Ride | null> {
    return this.rideRepository.findOne({
        where: [{ paid: false, client: { id } }],
    });
}
```

```
    }
}
```

Файл rtc-events.types.ts

```
import { Location } from './location.types';
import { CarClass } from './car-class.types';
import { ExtendedRideRequest } from './ride-request.types';

export enum WSMimeType {
  RideRequest = 'RIDE_REQUEST',
  RideStatusChange = 'RIDE_STATUS_CHANGE',
  RideAccept = 'RIDE_ACCEPT',
  RideDecline = 'RIDE_DECLINE',
  RideTimeout = 'RIDE_TIMEOUT',
  RideStopSearch = 'RIDE_STOP_SEARCH',
  LocationUpdate = 'LOCATION_UPDATE',
  RestoreState = 'RESTORE_STATE',
}

export type WithUserRole<T> = {
  isClient: boolean;
  data: T;
};

export type SocketUserInfo = {
  location?: Location;
  companionId?: number;
  stopSearch?: () => void;
  rideId?: number;
  rideRequest?: ExtendedRideRequest;
};

export type DriverSocketUserInfo = SocketUserInfo & {
  carClass: CarClass;
  toPickUp?: Array<Location>;
};
```

Файл awaitable-socket.utils.ts

```
import { Socket } from 'socket.io';
import { WSMimeType } from './types/rtc-events.types';

export const waitForSocketResponse = async (
  socket: Socket,
  timeout: number,
  onTimeoutEvent: WSMimeType,
  ...waitFor: WSMimeType[]
): Promise<{ event: WSMimeType; data: any }> => {
  let wasResolved = false;

  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);
  });
};
```

Файл geo-utils.utils.ts

```
import { Injectable } from '@nestjs/common';
import { Socket } from 'socket.io';
```

```

import { SocketUserInfo } from './types rtc-events.types';
import { Location } from './types location.types';

@Injectable()
export class GeoUtils {
    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;
    }
};

toSortedList = (from: Location, unsorted: Map<number, SocketUserInfo>) => {
    const array = Array.from(unsorted, ([name, value]) => [name, value] as [number,
    SocketUserInfo]);
    return array.sort((a, b) => {
        return (
            this.getDistance(from.latitude, from.longitude, a[1].location.latitude,
            a[1].location.longitude, 'K') -
            this.getDistance(from.latitude, from.longitude, b[1].location.latitude,
            b[1].location.longitude, 'K')
        );
    });
}
}

```

ПРИЛОЖЕНИЕ Б

Модель данных

ПРИЛОЖЕНИЕ В
(обязательное)

Спецификация программного дипломного проекта

ПРИЛОЖЕНИЕ Г
(обязательное)

Ведомость документов