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 Блок пользовательского интерфейса

Блок 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);

– handeBlur() – 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 блоки 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<ScrolView | 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 блоки 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 блоки 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;

– `defaultPaymnentMethod : PaymentMethod` – способ оплаты по умолчанию.

Компонент `HistoryScreen` содержит следующие функции:

– `dispatch(Action)` – функция, которая отсылает события с пользовательского интерфейса в блоки 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 блоки 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 блоки 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 блоки 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 блоки 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 блоке 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 отображает запрос 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 блоки хранилища и бизнес-логики;

– `answer(WSMessageType)` – функция, отправляющая выбор пользователя в блок 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 блоки 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 блоки хранилища и бизнес-логики;

– `onChangeText(string, string)` – функция, посылающая событие об изменении значения поля ввода локации в блок 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 блоки хранилища и бизнес-логики;

– `onResultPress(ExtendedLocation)` – функция-обработчик нажатия на элемент списка, посылающая событие о выборе в блок бизнес-логики.

3.1.1.20 Компонент ChooseClass

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

Компонент `ChooseClass` содержит следующие переменные:

– `rideRequest : Optional<RideRequest>` – информация о стоимости, времени и маршруте поездки;

– `isRideRequestLoading : boolean` – флаг, показывающий состояние загрузки информации о поездке;

– `isCarSearching : boolean` – флаг, показывающий, находится ли клиент в состоянии поиска машины;

– `selectedClass : number | null` – индекс выбранного класса машины в массиве доступных классов.

Компонент `ChooseClass` содержит следующие функции:

– `setSelectedClass(number | null)` – функция установки значения переменной `selectedClass`;

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

– `handleCancel()` – функция-обработчик нажатия на кнопку отмены, при нажатии отсылает событие об отмене поиска в блок бизнес-логики;

– `handleGoPress()` – функция-обработчик нажатия на кнопку поиска машины, при нажатии отсылает событие о начале поиска в блок 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 Блок мобильной бизнес-логики

Блок 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 блока пользовательского интерфейса их можно вызвать только отсылая определенные события с использованием функции `dispatch`. Функции оперируют объектами, полученными из блоков хранилища, сетевого

блока, блока 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 блока 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 посылается на сервер.

3.1.2.4 Функция listenForLogin

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

3.1.2.5 Функция logoutSaga

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

3.1.2.6 Функция listenForLogout

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

3.1.2.9 Функция fetchHistorySaga

Функция `fetchHistorySaga` выполняет запрос на получение списка поездок пользователя, используя блок 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 блока пользовательского интерфейса будет вызвана функция `fetchHistorySaga`.

3.1.2.11 Функция getUserSaga

Функция `getUserSaga` выполняет запрос на получение данных о пользователе, используя блок 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 listenForGetUser

Функция `listenForGetUser` создает подписку функции `getUserSaga` на событие `GET_USER.TRIGGER`. При отправке этого

события с блока пользовательского интерфейса будет вызвана функция `getUserSaga`.

3.1.2.13 Функция getPaymentMethodsSaga

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

3.1.2.14 Функция listenForGetPaymentMethods

Функция `listenForGetPaymentMethods` создает подписку функции `getPaymentMethodsSaga` на событие `GET_PAYMENT_METHODS.TRIGGER`. При отправке этого события с блока 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 блока пользовательского интерфейса будет вызвана функция `removePaymentMethodSaga`.

3.1.2.17 Функция setAsDefaultPaymentMethodSaga

Функция `setAsDefaultPaymentMethodSaga` устанавливает метод оплаты методом по умолчанию и работает аналогично функции `removePaymentMethodSaga`, с теми лишь исключениями, что наличный способ оплаты не отсеивается, и запрос на установку метода по умолчанию выполняется на отдельный маршрут на сервере (эндпоинт).

3.1.2.18 Функция listenForSetAsDefaultPaymentMethod

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

3.1.2.41 Функция setRouteLocationSaga

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

3.1.2.42 Функция listenForSetRouteLocation

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

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 блока пользовательского интерфейса будет вызвана функция `setRouteLocationSaga`.

3.1.3 Блок работы с хранилищем

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

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

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

предыдущий, тем самым заставляя всех подписчиков произвести перерисовку для отображения актуальных данных.

3.1.3.1 Тип ApplicationState

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

- `user : UserState` – хранилище данных о профиле пользователя;
- `history : HistoryState` – хранилище данных о истории поездок;
- `payments : PaymentsState` – хранилище данных о платежных методах;
- `home : HomeState` – хранилище всех необходимых данных для вызова такси.

3.1.3.2 Тип State<T>

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

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

3.1.3.3 Тип UserState

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

3.1.3.3 Тип User

Тип `User` представляет из себя модель пользователя, зарегистрированного в системе. Данный тип 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.

3.1.3.4 Перечисление Gender

Перечисление `Gender` содержит доступный для выбора пол пользователя:
– `Male` – мужской пол;
– `Female` – женский пол.

3.1.3.5 Тип Driver

Тип `Driver` содержит информацию о машине и балансе водителя. Обязательные поля:
– `carBrand : string` – марка и модель машины;
– `carClass : CarClass` – класс машины;
– `balance : number` – баланс водителя.

3.1.3.6 Перечисление CarClass

Перечисление `CarClass` содержит возможные уровни машин, доступных к заказу:
– `Economy` – эконом-класс;
– `Comfort` – комфорт-класс;
– `Budiness` – бизнес-класс.

3.1.3.7 Тип HistoryState

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

3.1.3.8 Тип Ride

Тип `Ride` описывает структуру объекта поездки и сдержит следующие поля:
– `id : number` – уникальный идентификатор поездки;
– `client : User` – пользователь, заказавший такси;

– `driver : User` – водитель, принявший заказ;
– `cost : number` – стоимость 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.

3.1.3.9 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.

3.1.3.10 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.

3.1.3.11 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<CardMethodDetials> – детали банковской карты.

3.1.3.12 Перечисление PaymentMethodType

Перечисление PaymentMethodType отображает тип способа оплаты:
– Cash – оплата наличными;
– Card – оплата картой.

3.1.3.13 Тип CardMethodDetails

Перечисление PaymentMethodType отображает тип 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.

3.1.3.14 Перечисление CarBrand

Перечисление CarBrand отображает тип платежной карты:
– Visa;
– Mastercard;
– AmericanExpress.

3.1.3.15 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.

3.1.3.16 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.

3.1.3.17 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.

3.1.3.18 Тип Location

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

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

3.1.3.19 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.

3.1.3.20 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 : string` – маршрут поездки;
- `classes : Record<CarClass, number>` – доступные классы машин и цены на них.

3.1.3.21 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.

3.1.3.22 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 : string` – маршрут поездки;
- `to : ExtendedLocation` – место назначения;
- `from : ExtendedLocation` – место отправления;
- `cost : number` – стоимость поездки;
- `carClass : CarClass` – класс машины.

3.1.3.23 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.

3.1.3.24 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.

3.1.3.25 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.

3.1.3.26 Функция userReducer

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

3.1.3.27 Функция historyReducer

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

3.1.3.28 Функция paymentsReducer

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

3.1.3.29 Функция homeReducer

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

3.1.4 Блок работы с сервером

Блок работы с сервером реализован с использованием обьектно-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 решистрацию;

– `register(boolean) : 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 описании локации по координатам;

– updateMyLocation(number) – отправка текущего местоположения на сервер;

– requestRide(ExtendedRideRequest) – запрос на поиск машин поблизости;

– answerToRideRequest(WSMessageType) – отправка ответа водителя 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) : Response<Array<Ride>> – 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 Блок работы с GPS

Блок 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 Блок 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` – последняя полученная локация от операционной системы;

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:

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

Класс `MapService` содержит следующие методы:

– `setNavigationRef(NavigationContainerRef<any>)` – установка значения переменной `navigationRef`;

– `goBack()` – выход на предыдущий экран.

3.2 Серверное приложение

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

3.2.1 Блок 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 блока авторизации.

Класс `AuthController` содержит следующие методы:

– `constructor(AuthService)` – конструктор класса;

– `login(LoginInput) : Promise<AuthOutput>` – принимает запрос на вход;

– `register(RegisterInput) : Promise<AuthOutput>` – принимает запрос на регистрацию;

– `refreshToken(RefreshInputInput) : Promise<AuthOutput>` – 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 блока 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 блока 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 блока 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 Блок авторизации

Блок 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 блока работы с СУБД;

– `tokenProvider : TokenProvider` – экземпляр класса `TokenProvider`, предоставляющий возможность генерации авторизационных токенов;

– `userService : UserService` – экземпляр класса `UserService` из блока 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<AuthOutput>` – функция входа в систему;

– `register(RegisterInput) : Promise<AuthOutput>` – функция регистрации в системе;

– `refreshToken(RefreshInputInput) : Promise<AuthOutput>` – функция обновления токенов.

3.2.3 Блок обработки данных клиента

Блок 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 блока работы с СУБД;

– `driverRepository : DriverRepository` – экземпляр класса `DriverRepository` из блока работы с СУБД;

– `rideRepository : RideRepository` – экземпляр класса `RideRepository` из блока работы с СУБД;

– `paymentService : PaymentService` – экземпляр класса `PaymentService` из блока 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 Блок работы с СУБД

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

3.2.4.1 Класс AuthData

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

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

– `email : string` – почтовый адрес пользователя;

– `passwordHash : string` – хэш пароля.

3.2.4.2 Класс Driver

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

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

3.2.4.3 Класс Payment

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

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

3.2.4.4 Класс PaymentMethod

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

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

3.2.4.5 Класс PaymentMethodDetails

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

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

– `stripePaymentId : string` – идентификатор способа оплаты в системе Stripe.

3.2.4.6 Класс Ride

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

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

3.2.4.7 Класс User

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

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

3.2.4.8 Классы репозиториев

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

PaymentRepository, PaymentMethodRepository, PaymentMethodDetailsRepository, RideRepository и UserRepository.

3.2.5 Блок оплаты

3.2.5.1 Класс PaymentService

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

Класс PaymentService содержит следующие поля:

– paymentRepository : PaymentRepository – экземпляр класса PaymentRepository из блока работы с СУБД;

– paymentMethodRepository : PaymentMethodRepository – экземпляр класса PaymentMethodRepository из блока работы с СУБД;

– paymentMethodDetailsRepository : PaymentMethodDetailsRepository – экземпляр класса PaymentMethodDetailsRepository из блока работы с СУБД;

– userRepository : UserRepository – экземпляр класса UserRepository из блока работы с СУБД;

– rideRepository : RideRepository – экземпляр класса RideRepository из блока 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 Stipe 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 Блок подключения клиента

Блок 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 блока 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 Блок RTC

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

Класс RTCService содержит следующие поля:

– idBasedSocketHolder : Map<number, Socket> – структура данных, хранящая сокет пользователя по его идентификатору;

– socketBasedIdHolder : WeakMap<Socket, number> – структура данных, хранящая идентификатор пользователя по его сокету;

– clientDataHolder : Map<number, SocketUserInfo> – структура данных, хранящая данные о статусе клиента по его идентификатору;

– driverDataHolder : Map<number, DriverSocketUserInfo> – структура данных, хранящая данные о статусе водителя по его идентификатору;

– tokenProvider : TokenProvider – экземпляр класса TokenProvider, предоставляющий возможность генерации авторизационных токенов;

– userService : UserService – экземпляр класса UserService из блока обработки данных клиента;

– geoUtils : GeoUtils – экземпляр класса GeoUtils;

– rideRepository : RideRepository – экземпляр класса RideRepository из блока работы с СУБД.

Класс RTCService содержит следующие методы:

– constructor(TokenProvider, UserService, GeoUtils, RideRepository) – конструктор класса;

– getUserFromSocket(Socket) : Promise<User> – получает данные о пользователя исходя из текущего подключения;

– addUser(Socket) – добавление пользователей в список подключенных;

– removeUser(Sokcet) – 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 Блок расчета стоимости и маршрута поездки

Блок 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 блока работы с сервисами Google;

– places : Places – эк экземпляр класса Places из блока работы с сервисами Google;

– directions : Directions – экземпляр класса Directions из блока работы с сервисами Google.

Класс MapsService содержит следующие методы:

– constructor(Geocoding, Places, Directions) – конструктор класса;

– decode(Location) : Promise<ExtendedLocation> – вычисляет текстовое описания места по его географическим координатам;

– getDirection(from, to) : Promise<DecodeOutput> – 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 Блок 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 описания места.

– `maps : GoogleMaps` – экземпляр класса `GoogleMaps`;

– `geoUtils : GeoUtils` – экземпляр класса `GeoUtils`;

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

Класс `Places` содержит следующие методы:

– `constructor(GeoUtils, GoogleMaps)` – конструктор класса;

– `decode(Location): Promise<ExtendedLocation>` – производит определение места и возвращает его текстовое описание исходя из переданных координат.

3.2.9.4 Класс Directions

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

Класс `Directions` содержит следующие поля:

– `maps : GoogleMaps` – экземпляр класса `GoogleMaps`;

– `mockDirections : boolean` – флаг, показывающий необходимость подменить запрос на сервис тестовыми данными.

Класс `Directions` содержит следующие методы:

– `constructor(GoogleMaps)` – конструктор класса;

– `getDirection(Location, Location): Promise<Array<Location>>` – поиск кратчайшего маршрута между двумя точками;