2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

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

2.1 Мобильное приложение

2.1.1 Блок пользовательского интерфейса

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

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

2.1.2 Блок мобильной бизнес-логики

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

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

2.1.3 Блок работы с хранилищем

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

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

2.1.4 Блок работы с сервером

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

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

2.1.5 Блок работы с GPS

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

2.1.6 Блок вспомогательных утилит

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

2.2 Серверное приложение

2.2.1 Блок получения запроса клиента

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

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

2.2.2 Блок авторизации

Блок авторизации берет на себя задачу обработки и генерации секретных данных клиента. В сферу его ответственности входит сравнение

авторизационных данных (электронная почта и пароль), вычисление хэша пароля и генерация авторизационных ключей, использующихся в заголовках запросов. Авторизационные ключи позволяют однозначно определить клиента, отправившего запрос на сервер.

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

2.2.3 Блок обработки данных клиента

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

Блок написан на языке TypeScript без использования сторонних библиотек.

2.2.4 Блок работы с СУБД

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

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

2.2.5 Блок оплаты

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

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

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

2.2.6 Блок подключения клиента

Блок подключения клиента представляет из себя шлюз для подключения клиентов мобильного приложения к серверу по протоколу WebSocket. Это необходимо для двухстороннего обмена сообщения с клиентами в режиме реального времени ввиду того, что у клиентов отсутствуют публичные IP адреса, и сервера не могут напрямую посылать 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.