4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

4.1 Добавление платежной карты

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

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

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

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

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

Из-за соображений конфиденциальности и безопасности процесс проведения платежа разбит на две части: создание запроса на платеж и сам платеж соответственно. Создание запроса на платеж производится на сервере

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

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

const paymentIntent = await
this.stripe.paymentIntents.create(params);
```

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

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

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

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

Информация о платеже содержит необходимый нам уникальный идентификатор платежного средства. Для проведения дальнейших оплат

данный идентификатор, четыре последние цифры номера карты, срок действия и имя держателя отправляется на сервер для сохранения в базу данных:

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

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

4.2 Составление маршрута поездки

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

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

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

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

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

```
if (this.mockPlaces) {
   return places.filter((place) =>
place.readableLocation.toLowerCase().includes(part.toLowerC
ase()));
} else {
   const result = await
this.maps.client.placeAutocomplete({
      params: {
         key: this.maps.mapsKey,
```

```
            input: part,
            language: Language.ru,
        },
    });

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

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

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

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

    if (isMoving) return;

    yield put(SET_CHOSEN_LOCATION.STARTED());

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

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

        if (result.error) {
            yield call(logger.log, result.error);
```

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

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

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

```
if (this.mockGeocoding) {
   const data = places.find((place) => {
      const distance = this.geoUtils.getDistance(
         location.latitude,
         location.longitude,
         place.latitude,
         place.longitude,
         'K'
      );

      return distance < 1;
   });

   if (data) return data;

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

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

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

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

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

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

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

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

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

```
      language: Language.ru,
   },
});

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

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

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

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

4.3 Отслеживание местоположения водителя

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

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

```
this.socket =
io(environmentConfig.get('connectionGatewayAPI'), {
```

```
    transports: ['websocket'],
    auth: {
        Authorization: `Bearer ${this.authToken}`,
    },
    reconnection: true,
});
```

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

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

```
export const gpsSubscribe: Subscribe<Location> = (emitter)
=> geolocationService.subscribeToPositionChange(emitter);

export const createGPSMessagesReceiver = ():
EventChannel<Location> =>
eventChannel<Location>(gpsSubscribe);

export function* receiveLocationUpdateSaga(location:
Location): SagaIterator {
    const user = yield select(getExistingUser);

    if (!user) return;

    yield call(homeAPI.updateMyLocation, location);
}

export function* bootstrapGPSSubscription(): SagaIterator {
    const gpsEventChannel = yield
call(createGPSMessagesReceiver);

    yield takeLatest(gpsEventChannel,
receiveLocationUpdateSaga);
}
```

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

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

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

toSearchIn.set(userId, info);

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

this.idBasedSocketHolder.get(info.companionId)?.emit(WSMessa
geType.LocationUpdate, {
      from: userId,
      location,
   });
}
```

4.4 Поиск водителя

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

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

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

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

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

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

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

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

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

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

Для расчета расстояния на сфере используется метод вычисления расстояний на большом круге. Длина дуги большого круга – кратчайшее расстояние между любыми двумя точками находящимися на поверхности сферы, измеренное вдоль линии соединяющей эти две точки (такая линия носит название ортодромии) и проходящей по поверхности сферы или другой поверхности вращения. Сферическая геометрия отличается от обычной

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, если неправильно выбраны, могут привести к значительным искажениям длин в силу особенностей проекционных искажений.

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

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

      socket.on(msg, listener);

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

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

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

При успешном подборе водителя создается новый экземпляр поездки, устанавливается статус водителя в состояние занятого и отправляется событие об успешном начале поездки водителю и клиенту. В свою очередь мобильное приложение переходит в состояние отображения информации о поездке, в котором на карте отрисовывается маршрут и текущее местоположение водителя, а также отображается меню, содержащее в себе информацию о водителе или клиенте, машине и времени поездки.