1. **РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ**

**4.1 Добавление платежной карты**

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

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

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

Параметром функции addCardSaga(action: Action) является:

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

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

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

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

const result: CreatePaymentIntentResponse = yield call(paymentsAPI.createPaymentIntent, {  
 bynAmount: 3,  
 requestThreeDSecure: 'automatic',  
});

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

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

const secret = result.data.clientSecret;

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

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

Шаг 4. Отправка уникального идентификатора карты на сервер для последующего переиспользования.

На сервере сохраняется публичная информация о карте (последние четыре цифры, срок действия, бренд) и ее уникальный идентификатор в платежной системе Stripe.

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

Шаг 5. Выход на экран списка доступных платежных методов.

yield call(navigationService.goBack);

Шаг 6. Запуск функции обновления списка платежных методов.

yield call(getPaymentMethodsSaga);

Шаг 7. Отображение всплывающего уведомления об окончании операции.

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

Шаг 8. Конец алгоритма.

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

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

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

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

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

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

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

Шаг 2. Создание запроса на проведение оплаты.

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

Шаг 3. Сохранение платежного идентификатора пользователя для последующих оплат сохраненной картой. При использовании идентификатора карты без идентификатора владельца оплата будет отклонена.

user.stripeClientId = result.customerId;

await this.userRepository.save(user);

Шаг 4. Возврат результата клиенту.

return {  
 clientSecret: result.intent.client\_secret,  
};

Шаг 5. Конец алгоритма.

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

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

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

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

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

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

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

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

const details = new PaymentMethodDetails();  
details.exp = cardData.exp;  
details.stripePaymentId = cardData.stripePaymentId;  
details.brand = cardData.brand;  
details.holder = cardData.holder;  
details.lastFour = cardData.lastFour;  
await this.paymentMethodDetailsRepository.save(details);

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

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

Шаг 4. Конец алгоритма.

## 4.2 Составление маршрута поездки

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

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

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

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

Параметром функции fetchPlacesSaga(action: Action) является:

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

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

* toastService – объект, позволяющий отображать уведомление на экране устройства.
* homeAPI – объект, предоставляющий возможность отправки запросов на маршруты сервера, выполняющие операции с географической картой;

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

yield debounce(300, FETCH\_PLACES.TRIGGER, fetchPlacesSaga);

Шаг 2. Проверка на наличие необходимого количества символов.

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

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

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

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

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

yield put(FETCH\_PLACES.COMPLETED({ results: result.data, direction: action.payload.direction }));

При возникновении ошибки на экране пользователя будет отображено текстовое уведомление.

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

Шаг 5. Конец алгоритма.

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

Параметром функции setChosenLocationSaga(action: Action) является:

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

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

* homeAPI – объект, предоставляющий возможность отправки запросов на маршруты сервера, выполняющие операции с географической картой;
* getIsPointerMoving – функция, возвращающая состояние указателя на карте (находится ли он в движении или нет);
* toastService – объект, позволяющий отображать уведомление на экране устройства.

Шаг 1. Подписка на событие о перемещении карты.

yield debounce(500, SET\_CHOSEN\_LOCATION.TRIGGER, setChosenLocationSaga);

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

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

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

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

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

yield put(SET\_CHOSEN\_LOCATION.COMPLETED(result.data));

При возникновении ошибки на экране пользователя будет отображено текстовое уведомление.

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

Шаг 5. Конец алгоритма.

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

Параметром функции prepareRideDataSaga(action: Action) является:

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

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

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

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

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

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

yield call(bottomSheetService.minimize);

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

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

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

yield put(SET\_RIDE\_REQUEST(result.data));

При возникновении ошибки на экране пользователя будет отображено текстовое уведомление.

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

Шаг 5. Конец алгоритма.

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

Параметром функции decode(location: Location) является:

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

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

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

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

if (this.mockGeocoding) {

...

}

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

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

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

Сферическая геометрия отличается от обычной Эвклидовой и уравнения расстояния также принимают другую форму. В Эвклидовой геометрии, кратчайшее расстояние между двумя точками – прямая линия. На сфере прямых линий не бывает. Эти линии на сфере являются частью больших кругов – окружностей, центры которых совпадают с центром сферы.

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

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

if (data) return data;

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

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

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

Шаг 5. Возвращение ответа пользователю.

return {  
 latitude: location.latitude,  
 longitude: location.longitude,  
 readableLocation: decoded.data?.results[0]?.address\_components[0]?.short\_name ?? 'Неизвестно',  
};

Шаг 6. Конец алгоритма.

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

Параметром функции search(part: string) является:

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

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

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

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

if (this.mockGeocoding) {

...

}

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

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

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

const result = await this.maps.client.placeAutocomplete({  
 params: {  
 key: this.maps.mapsKey,  
 input: part,  
 language: Language.*ru*,  
 },  
});

Шаг 4. Преобразование формата данных и их возврат пользователю.

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

Шаг 5. Конец алгоритма.

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

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

* from – широта и долгота точки отправления;
* to – широта и долгота точки назначения.

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

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

Шаг 1. Выполнение запроса подсчета пути на картографические API.

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

Шаг 2. Извлечение необходимых данных из ответа API.

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

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

const time = Math.ceil(data.minutes);  
  
return {  
 calculatedTime: time,  
 route: data.route,  
 classes: {  
 [CarClass.Economy]: 3 + 0.3 \* time,  
 [CarClass.Comfort]: 4 + 0.4 \* time,  
 [CarClass.Business]: 5 + 0.5 \* time,  
 },  
};