1. ФИНАЛЬНЫЙ ЭТАП

Задача командного тура

1.1. Легенда

Стремительно развиваясь, деревня Финтехово переросла в полноценный город. В деревне начал развиваться бизнес, поэтому здесь решили построить торговый центр (далее — ТЦ).

У владельца ТЦ есть множество площадей, которые предприниматели арендуют под магазины, кафе и т.д. При этом каждый расчётный период (например, месяц) арендаторы платят за занимаемое ими место.

Однако часто происходит следующая ситуация: арендатор задерживает платеж за арендованное помещение, но продолжает им пользоваться. Несмотря на его обещания погасить долг, владелец ТЦ не получает своих денег. В результате стороны договора не могут урегулировать спор и обращаются в суд.

В связи с тем, что люди стали чаще покупать товары в онлайн-сервисах, количество покупателей в ТЦ резко сократилось. В результате этого владельцы ТЦ понесли дополнительные убытки.

Часто блокчейн-технологии применяются там, где важно, чтобы все стороны какого-либо договора гарантированно исполнили свои финансовые обязательства, и это тот самый случай.

В финальной задаче сезона 2021/22 годов вам предстоит разработать децентрализованное приложение для контроля доходов арендодателя.

Как это будет работать?

- Арендодатель устанавливает условия для каждой арендной площади, управляет договорами с арендаторами.
- Каждый арендатор регистрируется в системе и привязывает свой криптокошелек. Он заключает договор с арендодателем и обязуется принимать оплату от покупателей в криптовалюте (*для простоты представим, что криптовалюта Ethereum используется повсеместно*).
- Каждый кассир, работающий на точке арендатора, также имеет доступ к системе. Он оформляет покупки и предоставляет покупателю *QR-код* для оплаты.
- Все вырученные средства поступают на счёт контракта арендодателя (*владельца ТЦ*). Каждый месяц часть этих средств идёт на оплату аренды, а остаток переводится на счет арендатора.
- Ни арендатор, ни арендодатель не могут самостоятельно изменить условия аренды: все действия происходят после согласия обеих сторон.

1.2. Набор заданий

Решение командной задачи разбито на подзадачи, сгруппированные в 3 набора.

Каждая подзадача (*user story*) формулирует необходимый функционал, который должен быть реализован командой, а также набор приемочных тестов (*acceptance criteria*), позволяющих проверить в полном ли объеме решена данная подзадача.

Каждый набор подзадач предлагается решать в отдельный соревновательный день (*итерацию*).

На каждой итерации решение подзадач проверяется с помощью системы автоматического тестирования, которая запускает решение участников с теми или иными параметрами в соответствии с приемочными тестами.

1.3. Условия проведения

- Участники во время командного этапа финального тура могут использовать интернет и заранее подготовленные библиотеки для решения задачи.
- Участники не ограничены в выборе языков программирования, если в задаче (подзадаче) не указано таких требований.
- Участники не могут пользоваться помощью тренера, наставника, сопровождающего лица или привлекать третьих лиц для решения задачи.
- Финальная задача формулируется участникам в первый день финального тура, но участники выполняют решение задачи поэтапно. Критерии прохождения каждого этапа формулируются для каждого дня финального тура. За подзадачи, решенные в конкретном этапе начисляются, баллы. Баллы за определенные подзадачи можно получить только в день, закрепленный за конкретным этапом.
- В начале первого дня состязаний участники каждой команды получают доступ к репозиторию на серверах gitlab.com. Каждая команда имеет свой собствен-

ный репозиторий. Члены других команд не имеют доступ к чужим репозиториям.

- В течение дня не ведется учет количества изменений, которые команды регистрируют в Git-репозитории.
- В конце каждого дня финального этапа жюри проверяет решение участников на соответствие приемочным тестам для каждой подзадачи, входящей в набор для соответствующего этапа.
- Баллы за все подзадачи, для которых прошло приемочное тестирование, определяют баллы, набранные командой в данный день соревнований. Для некоторых подзадач, на усмотрение жюри, может быть возможность получать баллы с дисконтом в последующие дни соревнований. Также есть подзадачи, которые позволяют набирать дополнительные баллы, если приемочные тесты для этих подзадач проходят в последующие дни (регрессионное тестирование).
- После завершения сорревновательного дня команды могут ознакомиться с логикой проверки решений и подать апелляцию не позже начала следующего сорревновательного дня, если не согласны с корректностью проведения тестов.
- После рассмотрения сути апелляции, жюри вправе провести тестирование еще раз и назначить команде баллы за соответствующие подзадачи.
- Описанные выше условия могут быть изменены членами жюри. Все изменения в условиях объявляются участникам перед началом каждого дня состязаний.

1.4. Процедура проведения приемочного тестирования и критерии оценки

Для каждого дня соревнований (для каждой итерации) справедлива следующая процедура приемочного тестирования:

- В конце каждого дня финального этапа команды должны сформировать запрос на слияние (*Merge Request*) из своей ветки исходного кода в основную ветку (*master*) в Git-репозитории.
- Команда ответственна за то, чтобы в запросе на слияние не было конфликтов. Запрос на слияние с конфликтами может не рассматриваться жюри для выполнения приемочного тестирования.
- После того, как все команды отправили запросы на слияние, жюри одобряет все запросы и приступает к приемочному тестированию, для тех подзадач, которые входят в соответствующую итерацию. Для этого исходный код приложения команды загружается в систему автоматического тестирования (поддержку которой осуществляет функциональность GitLab CI/CD), где запускаются автоматические тесты на соответствие решения участников требованиям к приемочным тестам (*acceptance criteria*).
- Если все приемочные тесты для данной подзадачи пройдены успешно, команда получает баллы за данную подзадачу. Если хотя бы один тест не проходит, то баллы за данное подзадачу не начисляются.

Приемочные тесты для каждой подзадачи описаны в разделе “Подробное описание подзадач”.

Дальше перечислены баллы, которые получает команда за решение подзадач в

каждой итерации.

Максимальное количество баллов, которое может набрать команда за решение всех подзадач — 400.

Первая итерация

user story	**баллы**
US-001 Развертывание контракта	1
US-002 Аренда помещения	18
US-101 Сборка и запуск решения	5
US-102 Аутентификация через MetaMask на стороне сервера	30
US-201 Аутентификация через MetaMask на стороне клиента	15

Максимальное количество баллов за итерацию по части смарт-контракта — 19.
Максимальное количество баллов за итерацию по серверной части приложения — 35.
Максимальное количество баллов за итерацию по пользовательскому интерфейсу приложения — 15.
Максимальное количество баллов за всю итерацию — 69.

Вторая итерация

user story	**баллы**
US-003 Управление списком кассиров	25
US-004 Приём платежей	15
US-103 Добавление помещений	10
US-104 Задание адреса контракта для помещения	12
US-105 Редактирование помещений	6
US-106 Удаление помещений	8
US-107 Просмотр списка помещений арендодателем	8
US-108 Просмотр списка арендованных и доступных для аренды помещений	12
US-109 Установка публичного названия арендуемого помещения	11
US-202 Добавление помещений	20
US-203 Просмотр помещений	15
US-204 Редактирование помещений	10
US-205 Установка публичного названия арендуемого помещения	9
US-206 Разрешение аренды помещения	20
US-207 Удаление помещений	6

Максимальное количество баллов за итерацию по части смарт-контракта — 40.
Максимальное количество баллов за итерацию по серверной части приложения — 67.
Максимальное количество баллов за итерацию по пользовательскому интерфейсу приложения — 80.
Максимальное количество баллов за всю итерацию — 187.

Третья итерация

user story	баллы
US-005 Вывод средств арендатора	21
US-006 Вывод средств арендодателя	17
US-007 Удаление контракта	36
US-110 Регистрация квитанции на оплату	25
US-111 Получение квитанции на оплату	5
US-208 Управление кассирами	20
US-209 Оплата квитанций	20

Максимальное количество баллов за итерацию по части смарт-контракта — 74.
Максимальное количество баллов за итерацию по серверной части приложения — 30.
Максимальное количество баллов за итерацию по пользовательскому интерфейсу приложения — 40.
Максимальное количество баллов за всю итерацию — 144.

Критерии определения команды-победителя командного тура

- Сумма баллов, набранных за решения подзадач командного тура финального этапа, определяет итоговую результативность команды (измеряемую в баллах).
- Команды ранжируются по результативности.
- Команда победитель определяется, как команда с максимальной результативностью.

1.5. Подробное описание подзадач

Участникам необходимо будет реализовать систему из трёх взаимосвязанных компонентов:

- смарт-контракта соглашения об аренде,
- серверной части,
- пользовательского интерфейса.

Контракт соглашения об аренде: высокоуровневое описание

Главная задача разрабатываемого нами смарт-контракта — сокращение количества судебных разбирательств между арендатором и арендодателем. Для предотвращения подобных разбирательств необходимо подготовить вещественные доказательства, например, историю транзакций, отображающую заработок арендатора. История транзакций должна быть проведена по заранее прописанным в смарт-контракте правилам и подтверждена криптографическими цифровыми подписями.

Смарт-контракт предоставляет следующие возможности его пользователям:

- интерфейс оплаты для клиентов арендатора,
- учёт доходов арендатора,
- автоматический вычет арендной платы.

Каждый смарт-контракт существует только на время аренды одного помещения одним арендатором. Когда срок аренды истекает, контракт должен быть удалён из сети блокчейн. При необходимости новый контракт может быть развёрнут в сети для того же помещения, а затем арендован тем же арендатором или другим человеком.

Таким образом жизненный цикл смарт-контракта состоит из трёх стадий:

- Создание свободного контракта, ожидающего начало аренды. Одновременно с этим создается юридический документ устанавливающий силу контракта с этим адресом на реальную арендную площадь.
- Помещение арендовано. Одновременно с этим арендатор подписывает упрощенный юридический документ, делегирующий всю работу на контракт.
- Аренда заканчивается по одной из трёх причин: преждевременный разрыв контракта, задолженность арендатора или штатное истечение срока.

(*) На протяжении жизни контракта у него может быть только один арендатор с момента начала аренды.

Серверная часть приложения: высокоуровневое описание

Задача серверной части — взять на себя работу с той частью данных, которая не может храниться на стороне смарт-контракта по экономическим и техническим соображениям.

Серверная часть приложения предоставляет следующие возможности её пользователям:

- аутентификация с помощью аккаунта в сети блокчейн,
- манипуляция со списком сдаваемых в аренду помещений для арендодателя,
- манипуляция арендованными помещениями для арендатора,
- выписывание квитанций для оплаты товаров или услуг арендатора для кассиров,
- получение квитанции из базы данных для клиентов арендатора.

Роли пользователей определяются следующим образом:

- Адрес аккаунта **арендодателя** передаётся приложению при запуске.
- Если в базе данных существует комната, арендованная пользователем, то он считается **арендатором**.
- Если в базе данных существует арендованная комната, в списке кассиров которой содержится адрес в сети блокчейн аккаунта пользователя, то этот пользователь — кассир.
- Аутентифицированный пользователь может не иметь роли, например, если он является потенциальным арендатором.
- Часть функций, например, получение квитанций, доступно и неаутентифицированным пользователям.

Пользовательский интерфейс приложения: высокоуровневое описание

Задача пользовательского интерфейса приложения — предоставить арендодателю и арендаторам удобный интерфейс для управления помещениями как на сервере, так и в смарт-контракте.

Первая итерация

Контракт соглашения об аренде: первая итерация

Цель итерации: реализация первого пункта жизненного цикла контракта.

Решение должно быть предоставлено в виде исходного кода на языке `Solidity`, компилируемым solc v0.8.11.

Контракт должен поддерживать следующий интерфейс:

- метод `nonpayable constructor(uint roomInternalId)` регистрирует контракт в сети блокчейн.
- метод `payable rent(uint deadline, address tenant, uint rentalRate, uint billingPeriodDuration, uint billingsCount, Sign landlordSign)` переводит контракт в арендованное состояние.
- метод `view getRoomInternalId() returns (uint)` возвращает id помещения, к которому привязан контракт.
- метод `view getLandlord() returns (address)` возвращает адрес владельца контракта.
- метод `view getTenant() returns (address)` возвращает адрес арендатора.
- метод `view getBillingPeriodDuration() returns (uint)` возвращает продолжительность расчётного периода.
- метод `view getRentStartTime() returns (uint)` возвращает временную метку начала аренды.
- метод `view getRentEndTime() returns (uint)` возвращает временную метку окончания аренды.
- метод `view getRentalRate() returns (uint)` возвращает арендную ставку.

Где Sign — следующая структура данных:

```
struct Sign
{
    uint8 v;
    bytes32 r;
    bytes32 s;
}
```

А параметр `landlordSign` сформирован из следующей EIP-712 сигнатуры:

`RentalPermit(uint256 deadline, address tenant, uint256 rentalRate, uint256 billingPeriodDuration, uint256 billingsCount)`

под доменом

`EIP712Domain(string name, string version, address verifyingContract)`

где

`name` = `"Rental Agreement"`

`version` = `"1.0"`

(*) Коды ошибок и примеры поведения методов в разных контекстах приведены в секции приемочного тестирования этой итерации.

Контракт соглашения об аренде: приемочное тестирование первой итерации

US-001 Развертывание контракта (1 балл)

Описание: Я, как арендодатель, могу развернуть смарт-контракт в сети блокчейн, указав ID помещения, арендную плату и период аренды.

Критерий оценивания (AC-001-01) : Развёртывание контракта (1 балл)

---

1. В директории `blockchain/contracts/` содержится контракт с именем `RentalAgreement`.
2. Аккаунт с адресом `<landlord-address>` вызывает у контракта метод `constructor` с параметром `(2419200)`. Транзакция исполняется успешно. Контракт регистрируется в сети блокчейн.
3. Аккаунт с адресом `<landlord-address>` вызывает у контракта метод `getRoomInternalId()`. Метод возвращает значение 2419200. Транзакция в блокчейн не отправляется.
4. Аккаунт с адресом `<landlord-address>` вызывает у контракта метод `getLandlord()`. Метод возвращает адрес `<landlord-address>`. Транзакция в блокчейн не отправляется.

US-002 Аренда помещения (18 баллов)

Зависит от успешного прохождения US-001

Описание: Я, как арендатор, могу арендовать помещение, имея подпись арендодателя и достаточное количество средств на счету для оплаты первого расчетного периода.

Критерий оценивания (AC-002-01) : Акт заключения сделки (17 баллов)

---

1. Аккаунт с адресом `<landlord-address>` вызывает у контракта метод `constructor` с параметром `(2419200)`. Транзакция исполняется успешно. Контракт регистрируется в сети блокчейн.
2. Аккаунт с адресом `<tenant-address>` вызывает у контракта метод `rent()` с

параметрами `(310, '<tenant-address>', 100, 1000, 5, ...)` и передаёт `100` wei. Транзакция исполняется успешно. Block timestamp: 300.

3. Аккаунт с адресом `<landlord-address>` вызывает у контракта метод `getTenant()`. Метод возвращает адрес `<tenant-address>`. Транзакция в блокчейн не отправляется.
4. Аккаунт с адресом `<landlord-address>` вызывает у контракта метод `getRentalRate()`. Метод возвращает значение 100. Транзакция в блокчейн не отправляется.
5. Аккаунт с адресом `<landlord-address>` вызывает у контракта метод `getBillingPeriodDuration()`. Метод возвращает значение 1000. Транзакция в блокчейн не отправляется.
6. Аккаунт с адресом `<landlord-address>` вызывает у контракта метод `getRentStartTime()`. Метод возвращает значение 300. Транзакция в блокчейн не отправляется.
7. Аккаунт с адресом `<landlord-address>` вызывает у контракта метод `getRentEndTime()`. Метод возвращает значение 5300. Транзакция в блокчейн не отправляется.
8. Аккаунт с адресом `<tenant-address>` вызывает у контракта метод `rent()` с параметрами `(410, '<tenant-address>', 100, 1000, 5, ...)` и передаёт `100` wei. В ходе вызова метода контракта выполняется команда `revert("The contract is being in not allowed state")`. Транзакция в блокчейн не отправляется. Block timestamp: 400.

Критерий оценивания (AC-002-02) : Валидация параметров акта (1 балл)

Зависит от успешного прохождения AC-002-01

1. Аккаунт с адресом `<landlord-address>` вызывает у контракта метод `constructor` с параметром `(2419200)`. Транзакция исполняется успешно. Контракт регистрируется в сети блокчейн.
2. Аккаунт с адресом `<tenant-address>` вызывает у контракта метод `rent()` с параметрами `(1642676872, '<tenant-address>', 100, 1000, 5, A)`, где `A` - сообщение `RentalPermit`, подписанное приватным ключом аккаунта `<address-of-somebody>`. Транзакция снабжается суммой в `100` wei. В ходе вызова метода контракта выполняется команда `revert("Invalid landlord sign")`. Транзакция в блокчейн не отправляется.
3. Аккаунт с адресом `<tenant-address>` вызывает у контракта метод `rent()` с параметрами `(100000, '<tenant-address>', 100, 1000, 5, A)`, где `A` - сообщение `RentalPermit`, подписанное приватным ключом аккаунта `<landlord-address>`. Транзакция снабжается суммой в `100` wei. В ходе вызова метода контракта выполняется команда `revert("The operation is outdated")`. Транзакция в блокчейн не отправляется.
4. Аккаунт с адресом `<address-of-somebody>` вызывает у контракта метод `rent()` с параметрами `(1642676872, '<tenant-address>', 100, 1000, 5, A)`, где `A` - сообщение `RentalPermit`, подписанное приватным ключом аккаунта `<landlord-address>`. Транзакция снабжается суммой в `100` wei. В ходе вызова метода контракта выполняется команда `revert("The caller account and`

`the account specified as a tenant do not match`"). Транзакция в блокчейн не отправляется.

5. Аккаунт с адресом `<landlord-address>` вызывает у контракта метод `rent()` с параметрами `(1642676872, '<landlord-address>', 100, 1000, 5, A)`, где `A` - сообщение `RentalPermit`, подписанное приватным ключом аккаунта `<landlord-address>`. Транзакция снабжается суммой в `100` wei. В ходе вызова метода контракта выполняется команда `revert("The landlord cannot become a tenant")`. Транзакция в блокчейн не отправляется.
6. Аккаунт с адресом `<tenant-address>` вызывает у контракта метод `rent()` с параметрами `(1642676872, '<tenant-address>', 0, 1000, 5, A)`, где `A` - сообщение `RentalPermit`, подписанное приватным ключом аккаунта `<landlord-address>`. В ходе вызова метода контракта выполняется команда `revert("Rent amount should be strictly greater than zero")`. Транзакция в блокчейн не отправляется.
7. Аккаунт с адресом `<tenant-address>` вызывает у контракта метод `rent()` с параметрами `(1642676872, '<tenant-address>', 100, 0, 5, A)`, где `A` - сообщение `RentalPermit`, подписанное приватным ключом аккаунта `<landlord-address>`. Транзакция снабжается суммой в `100` wei. В ходе вызова метода контракта выполняется команда `revert("Rent period should be strictly greater than zero")`. Транзакция в блокчейн не отправляется.
8. Аккаунт с адресом `<tenant-address>` вызывает у контракта метод `rent()` с параметрами `(1642676872, '<tenant-address>', 100, 1000, 0, A)`, где `A` - сообщение `RentalPermit`, подписанное приватным ключом аккаунта `<landlord-address>`. Транзакция снабжается суммой в `100` wei. В ходе вызова метода контракта выполняется команда `revert("Rent period repeats should be strictly greater than zero")`. Транзакция в блокчейн не отправляется.
9. Аккаунт с адресом `<tenant-address>` вызывает у контракта метод `rent()` с параметрами `(1642676872, '<tenant-address>', 100, 1000, 5, A)`, где `A` - сообщение `RentalPermit`, подписанное приватным ключом аккаунта `<landlord-address>`. Транзакция снабжается суммой в `99` wei. В ходе вызова метода контракта выполняется команда `revert("Incorrect deposit")`. Транзакция в блокчейн не отправляется.

Серверная часть приложения: первая итерация

Цель итерации: реализация первого пункта из списка функционала.

Серверная часть должна поддерживать запросы согласно следующей схеме `GraphQL`:

```
type Query {
    authentication: Authentication
}

type Mutation {
    requestAuthentication(address: String!): String!
    authenticate(
        address: String!
        signedMessage: InputSignature!
    ): Authentication!
```

```
}

type Authentication {
    address: String!
    isLandlord: Boolean!
}

input InputSignature {
    v: String!
    r: String!
    s: String!
}
```

Серверная часть приложения: приемочное тестирование первой итерации

US-101 Сборка и запуск решения (5 баллов)

Описание: Я, как системный администратор, могу собрать и запустить приложение с помощью `docker-compose`.

Критерий оценивания (AC-101-01) : Сборка и запуск приложения через docker-compose (3 балла)

---

1. Файл `docker-compose.yaml` написан участниками и находится в корне директории с решением. Решение участников принимает на вход переменные окружения `$LANDLORD_ADDRESS` в качестве адреса аккаунта арендодателя и `$RPC_URL` в качестве URL-адреса узла RPC блокчейна.
   Команда `docker-compose up --detach --build --force-recreate` в директории с исходным кодом исполняется успешно.
2. Команда `docker-compose ps | grep solution-web` выводит строку с информацией о собранном контейнере.

Критерий оценивания (AC-101-02) : Проверка доступности сервера (2 балла)

---

1. GET-запрос к серверу по адресу `http://solution-web/graphql` выполняется успешно (код ответа 200).

US-102 Аутентификация через MetaMask на стороне сервера (30 баллов)

Зависит от успешного прохождения US-101

Описание: Я, как пользователь приложения, могу аутентифицироваться в нём, используя свой кошелёк MetaMask.

Критерий оценивания (AC-102-01) : Генерация сообщений для аутентификации (5 баллов)

---

1. Здесь и далее запросы к серверу отправляются по адресу `http://solution-web/graphql`. На сервер отправляется следующий запрос:

```
mutation {
    message: requestAuthentication(
        address: "<address-0>"
    )
}
```

Запрос выполняется успешно и возвращает следующий JSON-объект:

```
{
    "data": {
        "message": "<message-0>"
    }
}
```

2. На сервер отправляется следующий запрос:

```
mutation {
    message: requestAuthentication(
        address: "<address-0>"
    )
}
```

Запрос выполняется успешно и возвращает следующий JSON-объект:

```
{
    "data": {
        "message": "<message-1>"
    }
}
```

3. На сервер отправляется следующий запрос:

```
mutation {
    message: requestAuthentication(
        address: "<address-1>"
    )
}
```

Запрос выполняется успешно и возвращает следующий JSON-объект:

```
{
    "data": {
        "message": "<message-2>"
    }
}
```

4. `<message-0>`, `<message-1>` и `<message-2>` попарно отличаются друг от друга.

Критерий оценивания (AC-102-02) : Успешная аутентификация (10 баллов)

---

1. На сервер отправляется следующий запрос:

   ```
   query {
       authentication { address, isLandlord }
   }
   ```

   Запрос успешно и возвращает следующий JSON-объект:

   ```
   {
       "data": {
           "authentication": null
       }
   }
   ```

2. На сервер отправляется следующий запрос:

   ```
   mutation {
       message: requestAuthentication(
           address: "<address-0>"
       )
   }
   ```

   Запрос выполняется успешно и возвращает следующий JSON-объект:

   ```
   {
       "data": {
           "message": "<message-0>"
       }
   }
   ```

3. Владелец аккаунта `<address-0>` подписывает сообщение `<message-0>` своим приватным ключом и получает объект подписи `<signature-0>`, состоящий из трёх компонентов: `<v-0>`, `<r-0>` и `<s-0>`, здесь и далее представляющие из себя числа в шестнадцатеричной системе, начинающиеся с `0x`.
4. На сервер отправляется следующий запрос:

   ```
   mutation {
       authentication: authenticate(
           address: "<address-0>"
           signedMessage: {
               signature: {
                   v: "<v-0>",
                   r: "<r-0>",
                   s: "<s-0>"
               }
           }
       ) {
           address
           isLandlord
   ```

```
    }
}
```

Запрос выполняется успешно и возвращает следующий JSON-объект:

```
{
    "data": {
        "authentication": {
            "address": "<address-0>",
            "isLandlord": false
        }
    }
}
```

Переданные в ответе куки могут использоваться при совершении действий от имени владельца аккаунта `<address-0>`.

5. На сервер отправляется следующий запрос:

```
query {
    authentication { address, isLandlord }
}
```

Запрос выполняется успешно и возвращает следующий JSON-объект:

```
{
    "data": {
        "authentication": {
            "address": "<address-0>",
            "isLandlord": false
        }
    }
}
```

6. На сервер отправляется следующий запрос:

```
mutation {
    message: requestAuthentication(
        address: "<landlord-address>"
    )
}
```

Запрос выполняется успешно и возвращает следующий JSON-объект:

```
{
    "data": {
        "message": "<message-1>"
    }
}
```

7. Арендодатель с аккаунта `<landlord-address>` подписывает сообщение `<message-1>` своим приватным ключом и получает объект подписи `<signature-1>`, состоящий из трёх компонентов: `<v-1>`, `<r-1>` и `<s-1>`.
8. На сервер отправляется следующий запрос:

```
mutation {
    authentication: authenticate(
        address: "<landlord-address>"
        signedMessage: {
            signature: {
                v: "<v-1>",
                r: "<r-1>",
                s: "<s-1>"
            }
        }
    ) {
        address
        isLandlord
    }
}
```

Запрос выполняется успешно и возвращает следующий JSON-объект:

```
{
    "data": {
        "authentication": {
            "address": "<landlord-address>",
            "isLandlord": true
        }
    }
}
```

Переданные в ответе куки могут использоваться при совершении действий от имени владельца аккаунта `<landlord-address>`.

9. На сервер отправляется следующий запрос:

```
query {
    authentication { address, isLandlord }
}
```

Запрос выполняется успешно и возвращает следующий JSON-объект:

```
{
    "data": {
        "authentication": {
            "address": "<landlord-address>",
            "isLandlord": true
        }
    }
}
```

Критерий оценивания (AC-102-03) : Неудачная аутентификация (15 баллов)

---

1. На сервер отправляется следующий запрос:

```
mutation {
    authentication: authenticate(
        address: "<landlord-address>"
        signedMessage: {
            signature: {
                v: "<v-1>",
                r: "<r-1>",
                s: "<s-1>"
            }
        }
    ) {
        address
        isLandlord
    }
}
```

Запрос выполняется с ошибкой и возвращает следующий JSON-объект:

```
{
    "errors": [
        {
            "message": "Authentication failed"
        }
    ]
}
```

2. На сервер отправляется следующий запрос:

```
mutation {
    message: requestAuthentication(
        address: "<address-0>"
    )
}
```

Запрос выполняется успешно и возвращает следующий JSON-объект:

```
{
    "data": {
        "message": "<message-2>"
    }
}
```

3. На сервер отправляется следующий запрос:

```
mutation {
    authentication: authenticate(
        address: "<address-0>"
        signedMessage: { signature: "<invalid-signature>" }
    ) {
        address
        isLandlord
    }
}
```

Запрос выполняется с ошибкой и возвращает следующий JSON-объект:

```
{
    "errors": [
        {
            "message": "Authentication failed"
        }
    ]
}
```

4. Пользователь с аккаунта `<address-1>` подписывает сообщение `<message-2>` своим приватным ключом и получает объект подписи `<signature-2>`, состоящий из трёх компонентов: `<v-2>`, `<r-2>` и `<s-2>`. На сервер отправляется следующий запрос:

```
mutation {
    authentication: authenticate(
        address: "<address-0>"
        signedMessage: {
            signature: {
                v: "<v-2>",
                r: "<r-2>",
                s: "<s-2>"
            }
        }
    ) {
        address
        isLandlord
    }
}
```

Запрос выполняется с ошибкой и возвращает следующий JSON-объект:

```
{
    "errors": [
        {
            "message": "Authentication failed"
        }
    ]
}
```

Пользовательский интерфейс приложения: первая итерация

Цель итерации: реализация аутентификации.

Пользовательский интерфейс приложения: приемочное тестирование первой итерации

US-201 Аутентификация через MetaMask на стороне клиента (15 баллов)

Описание: Я, как пользователь приложения, могу аутентифицироваться в нём, используя свой кошелёк MetaMask.

Критерий оценивания (AC-201-01) : Аутентификация (11 баллов)

1. В браузере открывается страница по адресу `http://solution-web/`.
2. На странице нажимается кнопка `.authentication__authenticate`.
3. MetaMask показывает всплывающее окно, в котором просит пользователя выбрать аккаунт для подключения. Пользователь выбирает аккаунт с адресом `<address-0>` и предоставляет необходимые разрешения.
4. MetaMask показывает новое всплывающее окно, в котором просит пользователя подтвердить подписание сообщения. Сообщение отображается в человекочитаемом виде. Пользователь подтверждает подписание сообщения, нажав кнопку *Sign*.
5. Кнопка `.authentication__authenticate` исчезает со страницы, и появляется новый элемент `.account__address` с текстом `<address-0>`.

Критерий оценивания (AC-201-02) : Проверка актуальности аутентификации (4 балла)

1. В браузере открывается страница по адресу `http://solution-web/`.
2. Пользователь проходит аутентификацию, используя аккаунт с адресом `<address-0>`.
3. Пользователь закрывает страницу, меняет активный аккаунт в MetaMask на `<address-1>`, и снова открывает `http://solution-web/`.
4. На странице появляется кнопка `.authentication__authenticate`, а также элемент `.authentication__warning` с текстом `Your MetaMask account is different from the one you authenticated with before`.