Содержание

Введение

Современные многопользовательские онлайн-игры требуют не только высокой производительности, но и надежной синхронизации действий между участниками, а также гибкого управления игровыми сессиями. Актуальность данной работы обусловлена растущим спросом на интерактивные игровые среды, обеспечивающие безопасное подключение, низкие задержки и удобство настройки для пользователей.

Целью проекта является разработка информационной системы для управления и синхронизации игровых сессий, реализованной на основе микросервисной архитектуры. В качестве примера выбрана классическая игра Pong, что позволяет продемонстрировать работу системы в условиях реального времени.

Основные задачи проекта:

1. Создание модульной архитектуры с использованием REST API для взаимодействия сервисов авторизации, управления хостами и синхронизации;
2. Разработка клиентских приложений: веб-интерфейса для управления настройками доступа (Dashboard) и игрового клиента (JPong);
3. Обеспечение безопасности через JWT-аутентификацию и проверку прав доступа;
4. Реализация синхронизации игровых сессий с минимальной задержкой и обработкой сетевых ошибок.

Для реализации сервисов была выбрана библиотека FastAPI на Python 3, а клиентская часть была реализована на React TS для панели управления и Jetpack Compose для реализации игры Pong.

1 Анализ предметной области и требований

Анализ предметной области и требований – это ключевой этап разработки информационной системы. Он помогает избежать недопонимания между разработчиками и пользователями, а также обеспечивает создание системы, которая действительно соответствует потребностям бизнеса.

1.1 Постановка задачи на разработку приложения

Эффективность взаимодействия пользователей в многопользовательских играх напрямую зависит от гибкости управления игровыми сессиями, безопасности подключений и удобства настройки параметров хоста. Разработка системы синхронизации игровых сессий направлена на оптимизацию процессов создания, подключения к играм и управления правами доступа, что повышает качество пользовательского опыта и расширяет возможности социального взаимодействия в игровой среде. В этой работе будет реализована классическая игра Pong как многопользовательская сессионная игра, чьи сессии и будет синхронизировать сервис.

Система охватывает следующие ключевые компоненты:

- пользователи — участники системы, которые могут регистрироваться, входить в аккаунт и управлять своими настройками;
- хосты — пользователи, создающие игровые сессии с возможностью настройки прав доступа (разрешение для друзей, блокировка нежелательных участников);
- друзья и блокировки — списки для управления доверенными и заблокированными пользователями;
- игровые сессии — вре́менные сессии (например, матчи в Pong), к которым могут подключаться игроки в роли участников или наблюдателей;

- синхронизация действий — обмен данными между участниками сессии для обеспечения согласованного состояния игры.

В рамках системы можно выделить следующие ключевые процессы:

1. авторизация и регистрация — создание учетной записи, вход в систему, проверка подлинности пользователя;
2. управление аккаунтом хоста:
    a) Редактирование списка друзей и заблокированных пользователей;
    b) Настройка разрешений: доступ для незарегистрированных пользователей и не друзей;
3. Взаимодействие с игровыми сессиями:
    a) Создание новой сессии;
    b) Подключение к сессии;
    c) Синхронизация действий между участниками;
    d) Выход из сессии.

Для реализации процессов система будет состоять из четырёх основных микросервисов:

1. auth — управляет регистрацией, аутентификацией, генерацией и проверкой токенов доступа;
2. hosts — отвечает за настройки аккаунта хоста: списки друзей, блокировок, прав доступа для незарегистрированных лиц и не друзей;
3. gateway — объединяет запросы к сервисам auth и hosts, выступает единой точкой входа для клиентов;
4. sync — синхронизирует состояние игровых сессий, обеспечивает передачу действий между участниками.

Сервисы взаимодействуют через API, что обеспечивает модульность и гибкость системы. Клиенты (Dashboard и JPong) отправляют запросы к Gateway, который перенаправляет их в соответствующие сервисы (Auth, Hosts). Сервис Sync работает асинхронно, обмениваясь данными с игровыми клиентами для

поддержания целостности сессий. Сервисы планируется переиспользовать в дальнейшем разработчиком этой работы, что стоит учитывать при проектировании.

1.2 Анализ функциональных требований

Функциональные требования определяют ключевые возможности системы, обеспечивающие управление игровыми сессиями, безопасное подключение участников и синхронизацию действий. Для системы «Среда синхронизации игровых сессий» требования, распределенные между сервисами и клиентами, описаны ниже.

Для сервиса Auth:

- система должна предоставлять возможность регистрации новых пользователей с генерацией уникального идентификатора;
- система должна обеспечивать аутентификацию пользователей через токены, включая проверку их валидности;
- система должна поддерживать обновление токенов доступа для авторизованных пользователей.

Для сервиса Hosts система должна предоставлять возможность настройки параметров хоста:

- управление списком друзей (добавление, удаление);
- управление списком заблокированных пользователей (блокировка, разблокировка);
- настройка разрешения доступа для незарегистрированных пользователей и ограничения подключения: только для друзей.

Также, для сервиса Hosts система должна проверять права доступа пользователей к игровым сессиям на основе настроек хоста и блокировок.

Для сервиса Sync система должна обеспечивать:

- создание временных игровых сессий с уникальным идентификатором хоста;

- синхронизацию действий между участниками сессии в реальном времени (например, перемещение объектов в играх или отправку сообщений в чате);
- обработку подключений/отключений игроков, включая автоматическое удаление неактивных сессий;
- защиту от подмены источника данных (source of truth) через уникальные ключи сессии.

Для сервиса Gateway (промежуточный слой):

- система должна объединять запросы к сервисам Auth и Hosts, выступая единой точкой входа;
- система должна проверять авторизацию пользователей перед перенаправлением запросов к целевым сервисам;
- система должна обрабатывать ошибки соединения с внешними сервисами и возвращать соответствующие статусы.

В процессе анализа задачи на разработку приложения была разработана диаграмма вариантов использования на рисунке 1. [1]

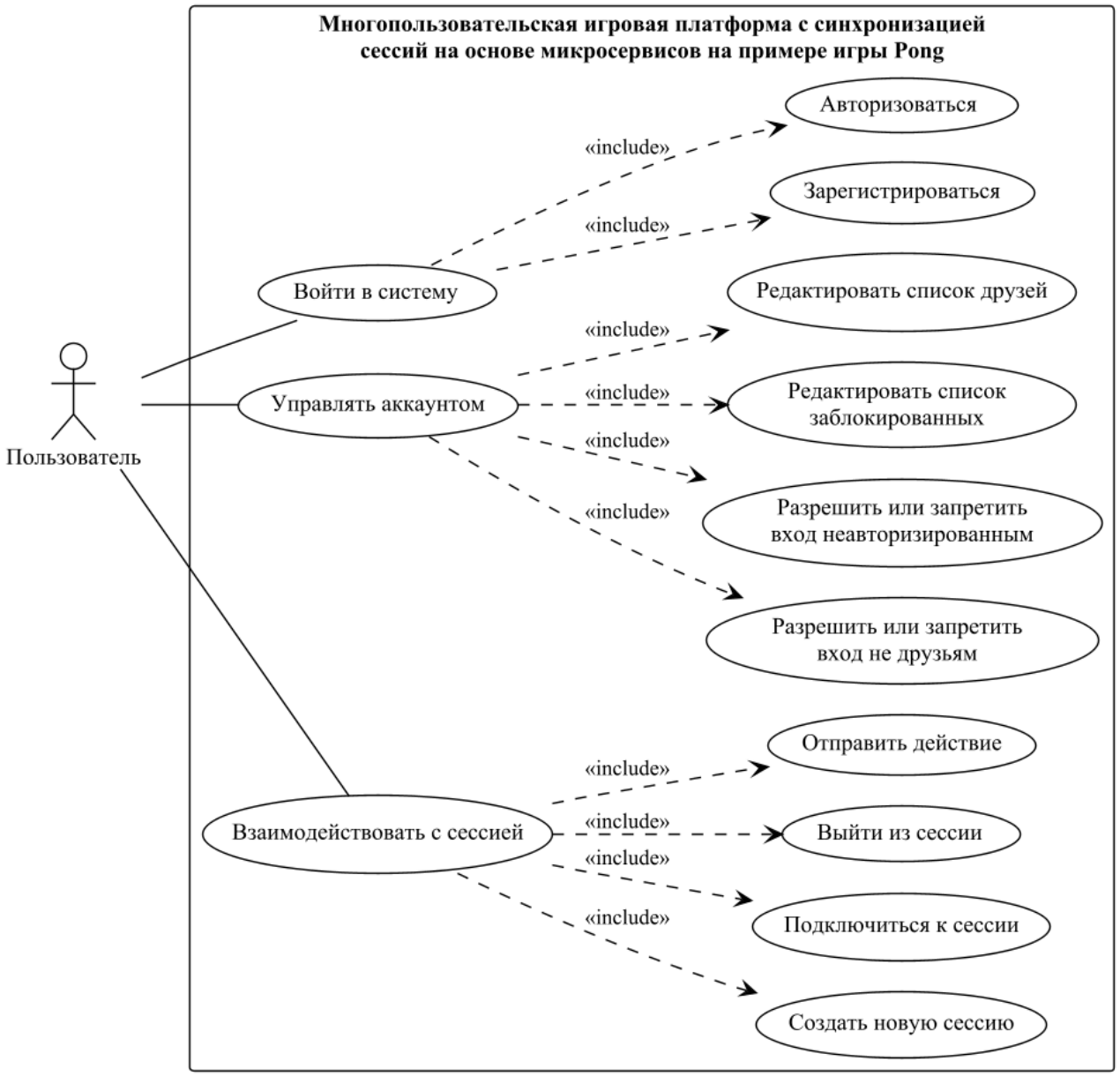

Рисунок 1 – Диаграмма вариантов использования для пользователя

Далее приведены спецификации некоторых из вариантов использования пользователя.

Вариант использования «Авторизоваться»:

ID: 1

Краткое описание: Пользователь авторизуется в системе для доступа к функционалу управления аккаунтом и игровыми сессиями.

Действующие лица: Пользователь, система.

Предусловие: Пользователь не авторизован в системе.

Основной поток:

1. Пользователь открывает клиентское приложение Dashboard;
2. Пользователь вводит свой идентификатор;
3. Система проверяет существование идентификатора;
4. Система предоставляет доступ к интерфейсу управления аккаунтом.

Постусловие: Пользователь авторизован. Ему доступно управление аккаунтом.

Вариант использования «Управлять аккаунтом»:

ID: 2

Краткое описание: Пользователь настраивает параметры своего аккаунта хоста.

Действующие лица: Пользователь, система.

Предусловие: Пользователь авторизован (см. вариант использования 1 – «Авторизоваться»).

Основной поток:

1. Пользователь открывает клиентское приложение Dashboard;
2. Система загружает текущие настройки аккаунта;
3. Пользователь выполняет одно из действией:
    a) Редактировать список друзей (добавить/удалить);
    b) Редактировать список блокировок (добавить/удалить);
    c) Разрешить/запретить вход не авторизированным;
    d) Разрешить/запретить вход не друзьям;

4. Система отправляет изменение и обновляет состояние интерфейса.

Постусловие: Настройки аккаунта обновлены. Изменения влияют на правила подключения к сессиям.

Вариант использования: Создать новую сессию

ID: 3

Предусловие: Пользователь должен быть зарегистрирован в системе.

Основной поток:

1. Пользователь создаёт сессию в сервисе Sync, указывая свой идентификатор;
2. Система проверяет зарегистрированность пользователя;
3. Система создаёт новую сессию;
4. Система возвращает ключ хоста и адрес новой сессии клиенту;
5. Пользователь присоединяется к сессии по адресу и ключу.

Постусловие: Пользователь подключён сессии.

Вариант использования: Подключиться к сессии

ID: 4

Предусловие: Пользователь зарегистрирован и подходит подходит под настройки подключения хоста.

1. Пользователь подключается к сессии, указывая свой идентификатор;
2. Система проверяет зарегистрированность пользователя;
3. Система проверяет, право подключения в зависимости от настроек хоста;
4. Пользователь присоединяется к сессии.

Постусловие: Пользователь подключён к сессии.

Вариант использования: Подключиться к сессии

ID: 5

Предусловие: Пользователь подключается анонимно.

1. Пользователь подключается к сессии, указывая свою анонимность;
2. Система проверяет, право подключения в зависимости от настроек хоста;
3. Пользователь присоединяется к сессии.

Постусловие: Пользователь подключён к сессии.

Выделенные варианты использования были реализованы при разработке приложения.

1.3 Анализ нефункциональных требований

Нефункциональные требования описывают характеристики системы, которые не относятся непосредственно к ее функциональности, но имеют важное значение для ее успешной работы.

Анализ нефункциональных требований к разрабатываемой информационной системе выявил ряд ключевых аспектов, определяющих качество и удобство использования программного обеспечения:

- производительность системы является критически важным фактором, поэтому обработка отправляемых сообщений в систему синхронизации и рассылка обновлённого состояния должны именить минимальную задержку;
- должна быть обеспечена авторизируемость и конфигурируемость, ведь система предполагает наличие работы с незарегистрированными пользователями и с конфигурациями доступа хостов;
- сопровождение кода и архитектуры должно быть обеспечено модульностью и использованием общепринятых стандартов программирования, что облегчит дальнейшую разработку и внесение изменений в систему.

1.4 Постановка задачи на разработку

В рамках курсового проекта необходимо разработать информационную систему, которая должна предоставить хостам удобный способ конфигурации доступа и всем пользователям обеспечить стабильную игру.

В информационной системе должны решаться следующие задачи:

- войти в систему;
- зарегистрироваться;
- авторизоваться;
- управлять аккаунтом;
- редактировать список друзей;
- редактировать список заблокированных;
- разрешить или запретить вход неавторизованным;
- разрешить или запретить вход не друзьям;
- взаимодействовать с сессией;
- создать новую сессию;
- подключиться к сессии;
- выйти из сессии;
- отправить действие.

В результате выполнения этих задач необходимо получить систему с архитектурой на основе сервисов, работающих по технологии REST API.

За образец игры Pong будет взята реализация с сайта https://www.ponggame.org/ [2]

2 Проектирование

В данном разделе курсового проекта будет предоставлено проектирование информационной системы «Многопользовательское сессионное игровое приложение pong», определены подходы и технологии для реализации системы.

2.1 Высокоуровневое проектирование программных средств

На рисунке 2 приведена диаграмма компонентов, из которых состоит система.

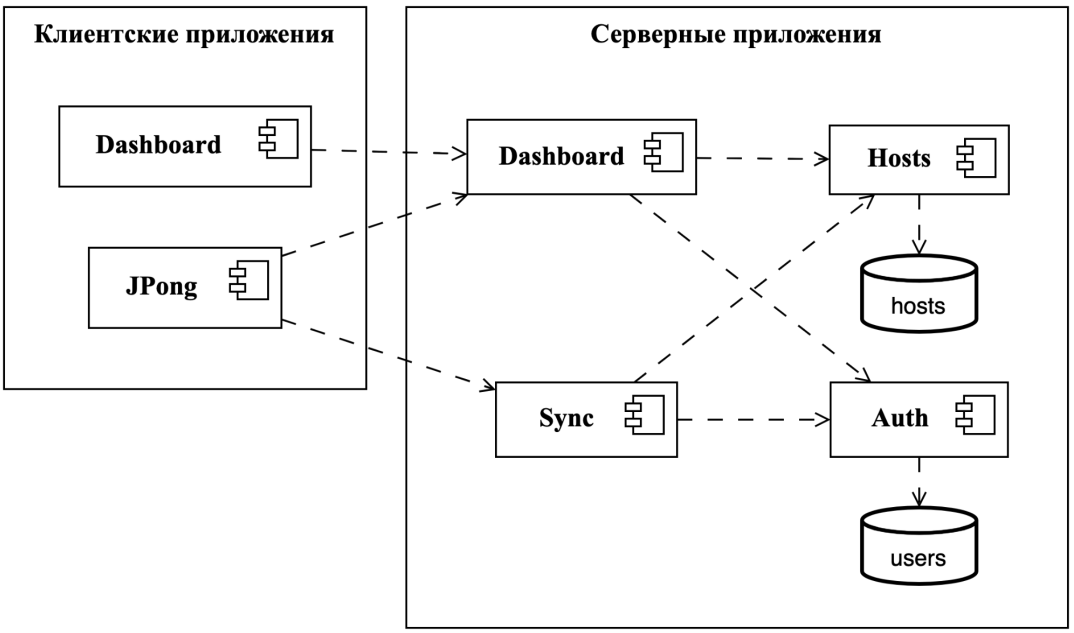

Рисунок 2 – Диаграмма компонентов приложения

Компонент Dashboard – клиентское приложение, предоставляющее интерфейс конфигурации настроек доступа к сессии хоста. Подробнее описано в разделе 2.3.1.

Компонент JPong – игровое клиентское приложение. Подробнее описано в разделе 2.3.2.

Компонент Hosts – сервис, управляющий конфигурацией параметров подключения к сессии хоста. Подробнее описан в разделе 2.2.1.

Компонент Auth – сервис, управляющий авторизацией пользователей. Подробнее описан в разделе 2.2.2.

Компонент Gateway – сервис, объединяющий во внешний API работу с сервисами Hosts и Auth. Подробнее описан в разделе 2.2.3.

Компонент Sync – сервис, обеспечивающий работу с сессиями. Работа с ним не происходит через Gateway чтобы обеспечить прямую, более быструю работу в соответствии с нефункциональным требованием. Подробнее описан в разделе 2.2.4.

2.2 Проектирование серверной части

Одним из основных этапов в проектировании клиент-серверного приложения является выбор типа построения API.

В качестве архитектуры для реализации серверного приложения был выбран REST API.

REST API – это архитектурный стиль, который описывает, как разработчику следует спроектировать интерфейс для взаимодействия своего приложения с другими. Если продолжить аналогию с естественным языком, то REST API описывает грамматику. Принципы и ограничения REST API были определены в 2000-м году Роем Филдингом – одним из создателей протокола HTTP. Говорят, что если интерфейс взаимодействия приложения соответствует принципам REST API, он является RESTful.

Как правило, для взаимодействия между клиентом и сервером достаточно четырех методов:

GET – получение информации об объекте (ресурсе);

POST – создание, а иногда и изменение объекта (ресурса);

PUT – изменение объекта (ресурса). Обычно используется для регулярного обновления данных;

DELETE – удаление информации об объекте (ресурсе).

При этом обмен сообщениями осуществляется обычно по протоколу HTTP(S).

Главная особенность REST API – обмен сообщениями без сохранения состояния. Каждое сообщение самодостаточное и содержит всю информацию, необходимую для его обработки. Сервер не хранит результаты предыдущих сессий с клиентскими приложениями. Это обеспечивает гибкость и масштабируемость серверной части, позволяет поддерживать асинхронные взаимодействия и реализовывать алгоритмы обработки любой сложности. Кроме того, такой формат взаимодействия является универсальным – он не зависит от технологий, используемых на клиенте и на сервере, и не привязывает разработчиков к определенному провайдеру [3].

Далее будет подробнее рассмотрен каждый компонент серверной части приложения.

2.2.1 Проектирование сервиса конфигурации хоста

В данном разделе будет подробнее рассмотрено проектирование логики работы и методов сервиса Hosts. Единственная цель существования этого сервиса – взаимодействие с аккаунтом хоста, а потому прежде чем описать его API, стоит описать структуру данных, с которой будет производиться взаимодействие:

- uuid – строка, universal unique identifier (универсально уникальный идентификатор) (UUID) идентификатор хоста;
- only_friends – булево значение, только ли друзья могут подключаться (режим белого списка);
- allow_nonames – булево значение, разрешено ли подключение неавторизованным лицам;
- friends – список UUID дружественных игроков, только они смогут подключаться, если полу only_friends установлено в true;

- banlist – список UUID заблокированных пользователей, которые не смогут никогда подключиться к хосту;

Все методы API сервиса Hosts осуществляют взаимодействие с конкретным хостом, чей UUID указывается в начале URL запросов в виде параметра {host}. Методы API сервиса Hosts представлены ниже:

1. POST /{host} – создаёт и возвращает новую запись о хосте;
2. GET /{host} – возвращает запись о хосте;
3. PUT /{host}/only_friends – устанавливает поле only_friends;
4. PUT /{host}/allow_nonames – устанавливает, разрешено ли подключение для неавторизованных пользователей;
5. POST /{host}/friends/{friend} – добавление нового друга;
6. DELETE /{host}/friends{friend} – удаление из списка друзей;
7. POST /{host}/banlist/{banned} – блокирование пользователя от подключения;
8. DELETE /{host}/banlist/{banned} – разблокирование пользователя от подключения;
9. GET /{host}/welcomes/{guest} – проверка, принимает ли хост (host) гостя (guest).

В вышеперечисленном списке {host}, {friend}, {banned} и {guest} – места, куда подставляются UUID хоста, друга, пользователя для блокировки и гостя в сессию соответственно.

Запросы PUT /{host}/only_friends и PUT /{host}/allow_nonames принимают json объекты с булевым полем only_friends и allow_nonames соответственно.

Общие возвращаемые значения:

1. код ошибки 404 и сообщение «not found this» в случае ненахождения записи о хосте;
2. код ошибки 404 и сообщение «not found other в случае ненахождения записи о втором операнде (friend / banned / guest);

3. код ошибки 400 в случае неверного запроса;

4. код успеха 200 в случае успеха.

Запрос GET /{host}/welcomes/{guest} в случае, если хост (host) «приветствует» гостя (guest), возвращает код 200 или код 404 без сообщения, если не гостю нет доступа в сессии этого хоста.

Диаграмма классов представлена на рисунке 3, а отношения функций к точкам доступа представлены в таблице 1:

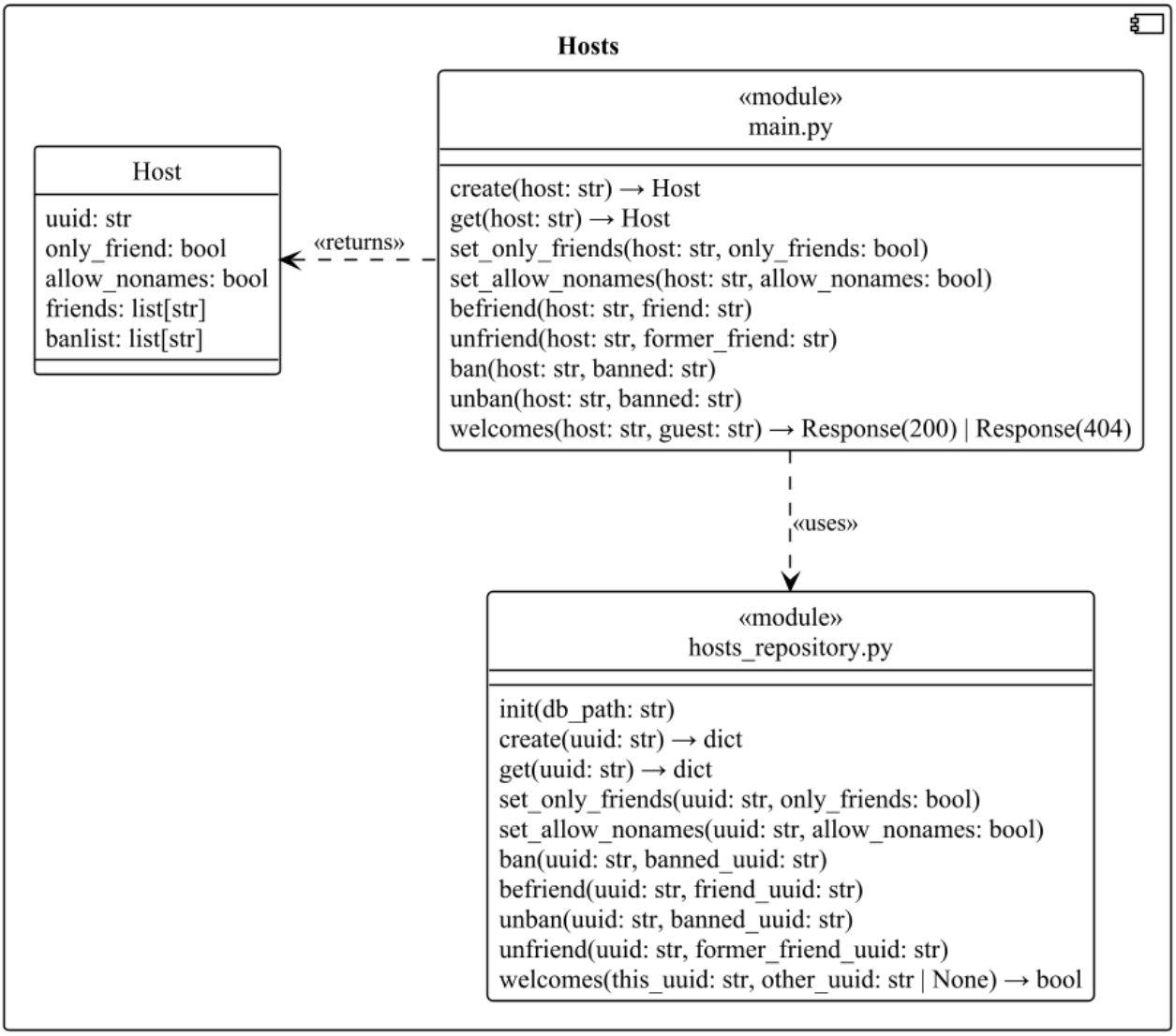

Рисунок 3 – Диаграмма классов компонента Hosts

Таблица 1 – Отношение функций модуля main.py к точкам доступа Hosts

Функция	**Точка доступа**
create	POST /{host}
get	GET /{host}
set_only_friends	PUT /{host}/only_friends
set_allow_nonames	PUT /{host}/allow_nonames
befriend	POST /{host}/friends/{friend}
unfriend	DELETE /{host}/friends/{former_friend}
ban	POST /{host}/banlist/{banned}
unban	DELETE /{host}/banlist/{banned}
welcomes	GET /{host}/welcomes/{guest}

Описанные функции будут использованы при реализации серверной части приложения.

2.2.2 Проектирование сервиса авторизации

В данном разделе рассматривается проектирование сервиса авторизации, который отвечает за генерацию, валидацию и обновление токенов доступа, а также за управление учётными данными пользователей. Основная цель сервиса – обеспечить безопасную аутентификацию и авторизацию через JWT-токены (JSON Web Tokens).

Для пользователей определена следующая структура данных:

- user_id – уникальный числовой идентификатор;
- uuid – уникальный строковый идентификатор пользователя в формате UUID версии 1.

Для самого JWT-токена содержание полей было определено следующим образом:

- iss (issuer) – url сервиса выпустившего токен;

- sub (subject) – UUID пользователя, кому выдан токен;
- aud (audience) – совпадает с subject;
- exp (expires) – время истечения токена в формате UNIX timestamp [4];
- iat (issued at) – время создания токена в формате UNIX timestamp;
- jti (JWT ID) – уникальный идентификатор токена.

Методы сервиса Auth представлены ниже:

1. POST / – регистрирует нового пользователя и возвращает JSON с полем uuid, где содержится UUID нового пользователя;
2. GET /login/{uuid} – возвращает код 200 и JSON с полем token, где в виде строки указан токен доступа для запрашиваемого пользователя, иначе, в случае отсутствия пользователя, возвращается код ошибки 404;
3. GET /renew – проверяет валидность токена и возвращает новый токен, если он истёк или неправилен, то возвращает код ошибки 401 и сообщение о том, что не так;
4. HEAD /{uuid} – через URL принимает UUID пользователя и через заголовок принимает токен, возвращает код 200, если токен валиден и 401 в ином случае.

Диаграмма классов с модулем, который будет обрабатывать точки доступа Auth и соотношения функций модуля к методам сервиса приведены на рисунке 4 и таблице 2 соответственно:

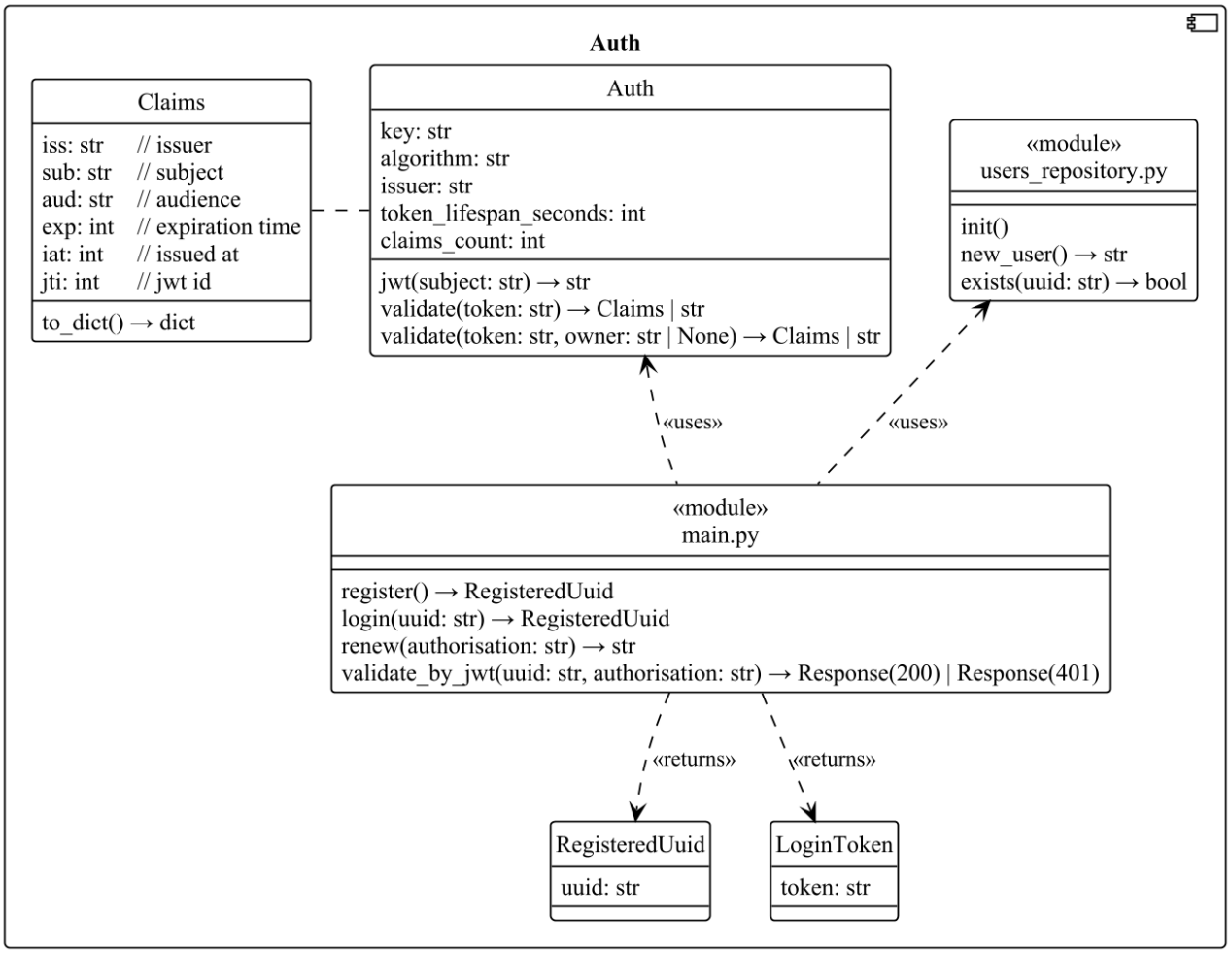

Рисунок 4 – Диаграмма классов компонента Auth

Таблица 2 – Отношение функций модуля main.py к точкам доступа Auth

Функция	**Точка доступа**
register	POST /
login	GET /login/{uuid}
renew	GET /renew
validate_by_jwt	HEAD /{uuid}

Описанные методы будут использованы при реализации серверной части приложения.

2.2.3 Проектирование управляющего сервиса

Сервис Gateway выступает центральным узлом для взаимодействия клиентов с другими компонентами системы: сервисами Auth и Hosts. Его основная задача – маршрутизация запросов, управление доступом через JWT-токены, обработка CORS и обеспечение отказоустойчивости. Gateway не хранит данные самостоятельно, а делегирует их обработку соответствующим сервисам.

Сервис Gateway переиспользует модели данных и ошибки из вышеописанных сервисов, лишь добавляя ошибку 503, если недоступен сервис Hosts или Auth, с сообщением в теле ответа «hosts» или «auth» соответственно.

Большинство запросов являются промежуточными шагами к запросу к сервису Hosts, но с проверкой через сервис Auth: если у пользователя есть право выполнить действие (например, редактировать конфигурацию хоста имеет право лишь сам хост, обладая выданным именно ему токеном), то запрос выполняется дальше, иначе возвращается код ошибки 401. В начале перечислим запросы, работающие этим общим образом (подробное описание приведено в разделе 2.2.1):

1. GET /hosts/{host};
2. PUT /hosts/{host}/only_friends;
3. PUT /hosts/{host}/allow_nonames;
4. POST /hosts/{host}/friends (с отличием: UUID друга передаётся как query параметр);
5. DELETE /hosts/{host}/friends/{friend};
6. POST /hosts/{host}/banlist (с отличием: UUID заблокированного пользователя передаётся как query параметр);
7. DELETE /hosts/{host}/banlist/{banned}.

Без изменений остались пара методов из сервиса Auth:

1. GET /hosts/{host}/access_token соответствует GET /login/{host};
2. GET /hosts/access_token/renew соответствует GET /renew.

Метод регистрации (сигнатура: POST /hosts, без параметров) объединяет работу обоих вышеперечисленных сервисов: в начале нужно создать пользователя в сервисе Auth, что породит UUID, а затем зарегистрировать нового пользователя в сервисе Hosts. Возвращается запись о хосте, описанная в разделе 2.2.1.

Диаграмма классов с модулем, который будет обрабатывать точки доступа Gateway и соотношения функций модуля к методам сервиса представлены на рисунке 5 и таблице 3 соответственно:

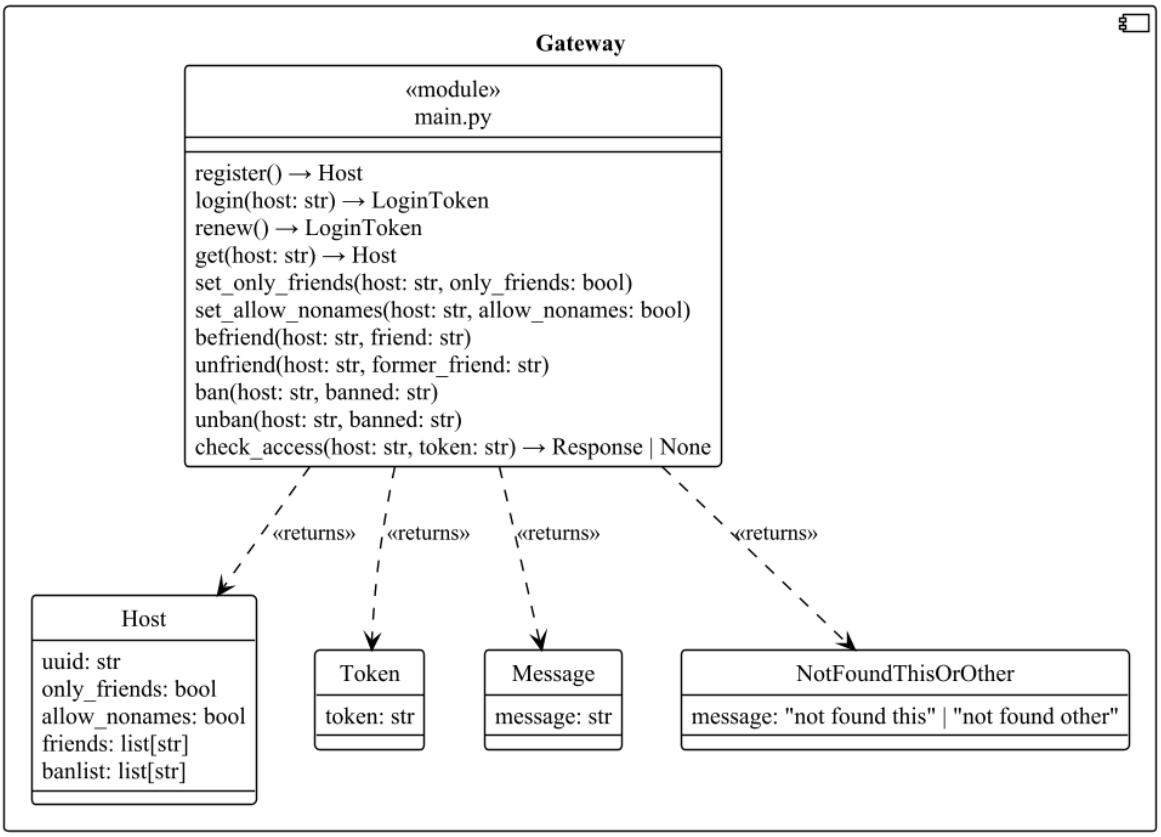

Рисунок 5 – Диаграмма классов компонента Gateway

Таблица 3 – Отношение функций модуля main.py к точкам доступа Gateway

Функция	**Точка доступа**
register	POST /hosts
login	GET /hosts/{host}/access_token

Продолжение таблицы 3

set_only_friends	PUT /hosts/{host}/only_friends
set_allow_nonames	PUT /hosts/{host}/allow_nonames
befriend	POST /hosts/{host}/friends?friend={friend}
unfriend	DELETE /hosts/{host}/friends/{former_friend}
ban	POST /hosts/{host}/banlist?friend={banned}
unban	DELETE /hosts/{host}/banlist/{banned}

2.2.4 Проектирование сервиса синхронизации

Сервис Sync отвечает за синхронизацию игровых сессий в реальном времени, управление подключениями игроков через WebSocket, а также обеспечение целостности данных между участниками. Он взаимодействует с сервисами авторизации и хостов для проверки прав доступа, но не хранит постоянные данные самостоятельно.

Методы с указанием протоколов сервиса представлены ниже:

1. HTTP GET / – возвращает все данные всех сессий;
2. HTTP GET /session/{host} – создаёт новую сессию и возвращает JSON с полями session_id (строка = UUID хоста) и source_of_truth_key (строка), которые будут нужны хосту для подключения.
3. WS /session/{session_id} – служит для подключения к сессии по её идентификатору, также принимает player_id через query и заголовок authorisation с токеном, если пользователь хочет присоединиться, используя свой аккаунт, а не анонимно, возвращает идентификатор игрока (UUID для не анонимных и сгенерированный ID для анонимных);
4. WS /session/{session_id}/player/{player_id} – служит для подключения к сессии игроков (дублируя в большей части функционал WS /session/{session_id}), но в случае, если этот игрок уже подключён, то не обнуляет данные игрока, хост для подключения также должен указать ранее выданный source_of_truth_key в заголовке authentication_info.

При подключении к сессиям, сервис проверяет доступ нового игрока через запрос GET /{hosts}/welcomes/{guest} к сервису Hosts, а проверка аутентификации 5 через сервис Auth. Механизмы работы изображены на рисунках 7 и 8, устройство сущности сессии, изображённой на диаграммах описано на диаграмме классов на рисунке 6.

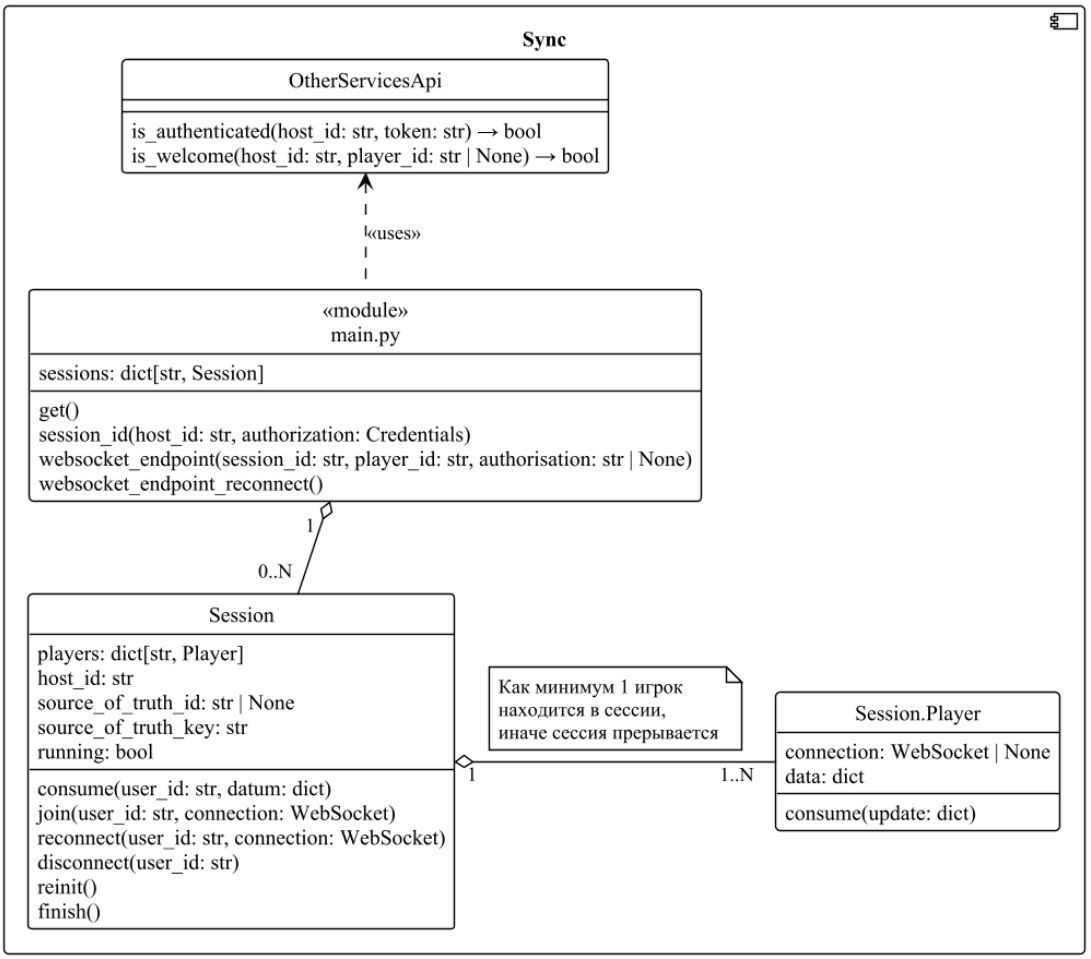

Рисунок 6 – Диаграмма классов компонента Sync

Таблица 4 – Отношения функций модуля main.py к точкам доступа Sync

Функция	**Точка доступа**
get	GET /
session_id	GET /session/{host_id}
websocket_endpoint	WEBSOCKET /session/{session_id}
wepsocket_endpoint_reconnect	WEBSOCKET /session/{session_id}/player/{player_id}

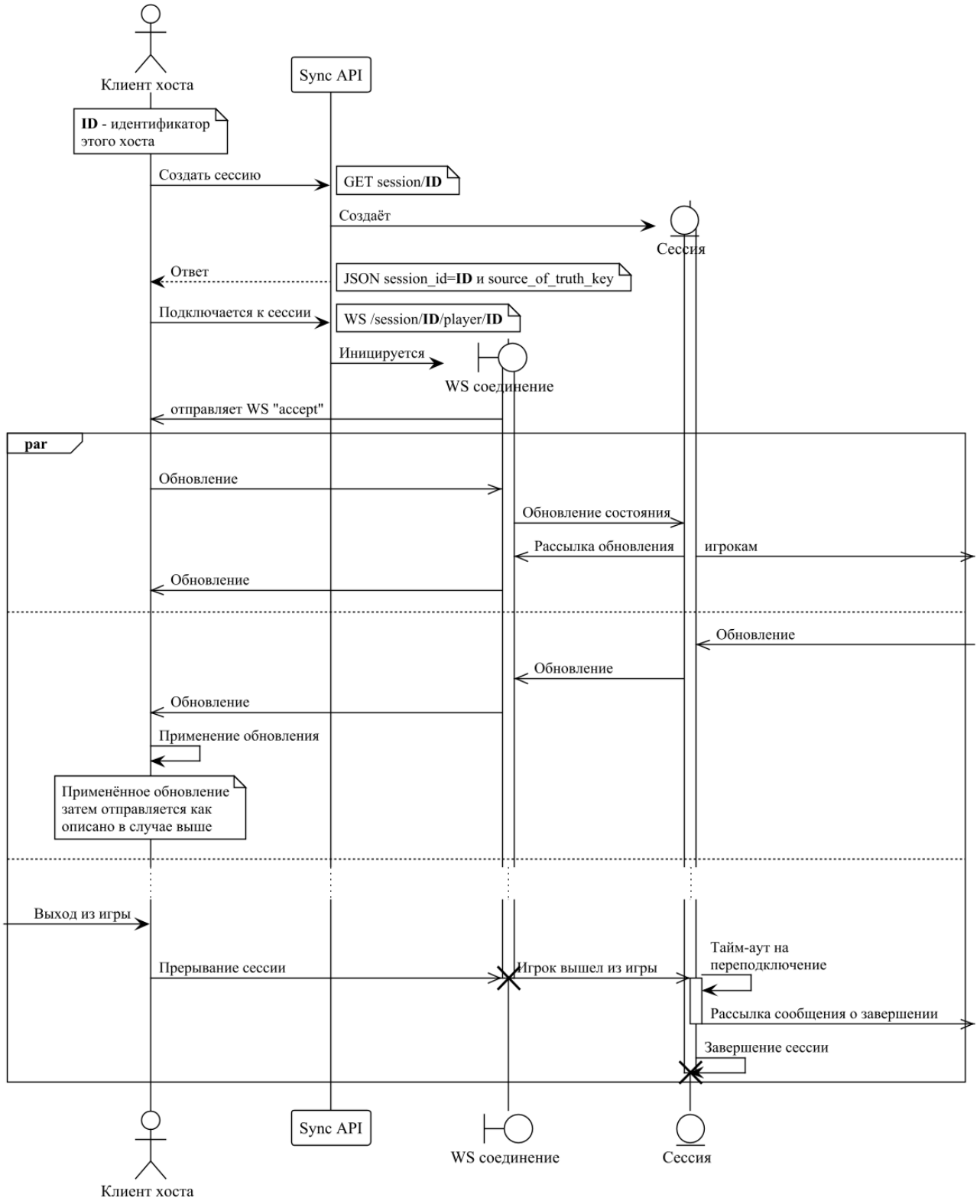

Рисунок 7 – Диаграмма последовательности работы хоста

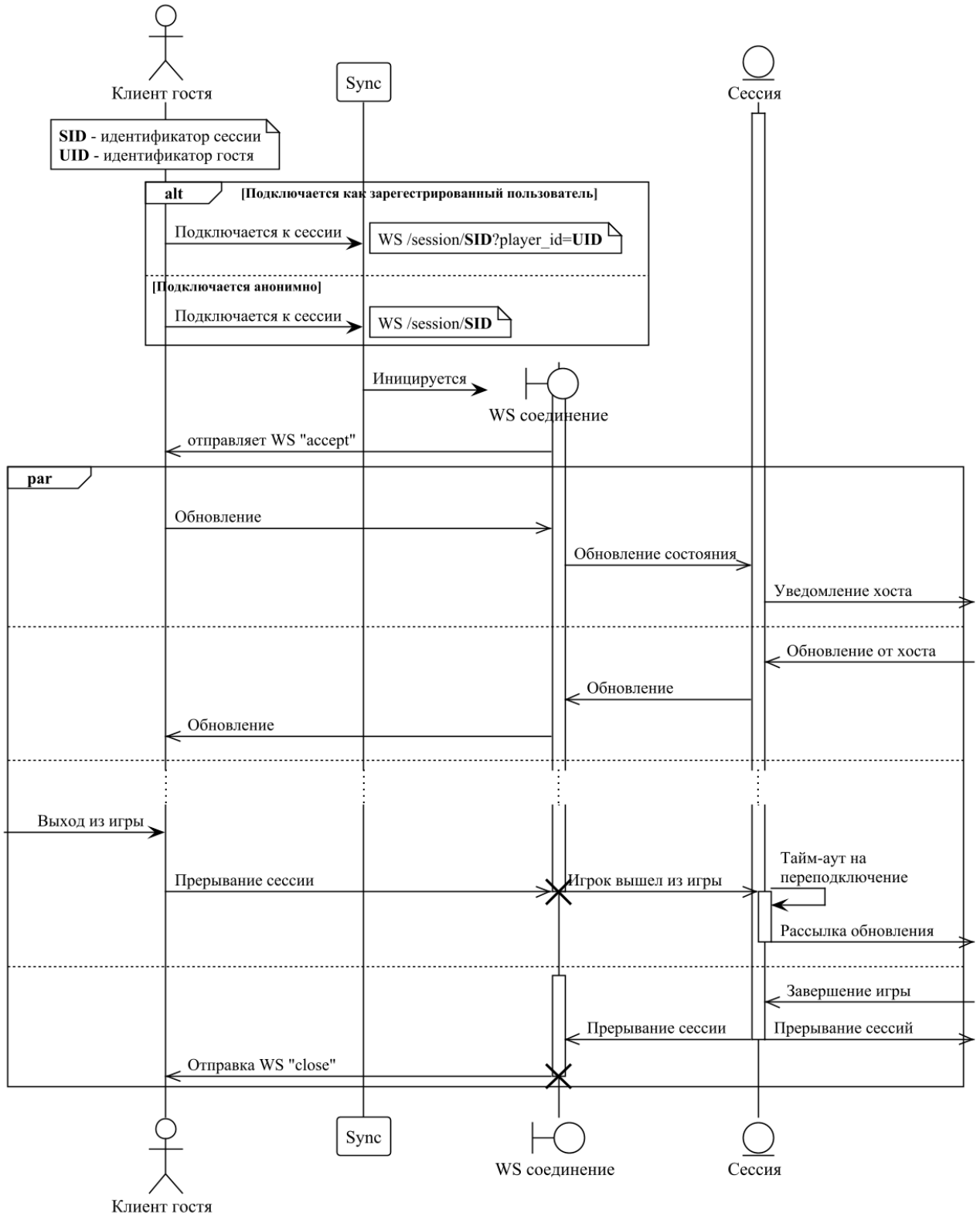

Рисунок 8 – Диаграмма последовательности работы гостя

На рисунках 7 и 8 изображено, как пользователи (хост и гость) осуществляют работу с сессиями.

2.3 Проектирование клиентской части

Логика управления конфигурацией аккаунта гостя отделена от логики синхронизации, необходимой для самой игры, что позволяет также сделать и независимые решения клиентской части чтобы пользователи могли менять свои настройки доступа вне зависимости от игры.

2.3.1 Проектирование приложения панели управления хоста

Приложение Dashboard взаимодействует с сервисом Gateway – получает и отображает текущую конфигурацию, а также позволяет управлять ею.

Пользователь системы должен иметь возможность создать аккаунт или войти в него. Конфигурация аккаунта состоит из двух булевых переменных: разрешение входить в сессию незарегистрированным пользователям и разрешение входить в сессию не друзьям. Также есть два списка: друзья и заблокированные пользователи. Всё вышеперечисленное представлено на рисунках 6 и 7:

Рисунок 9 – Макет страницы входа

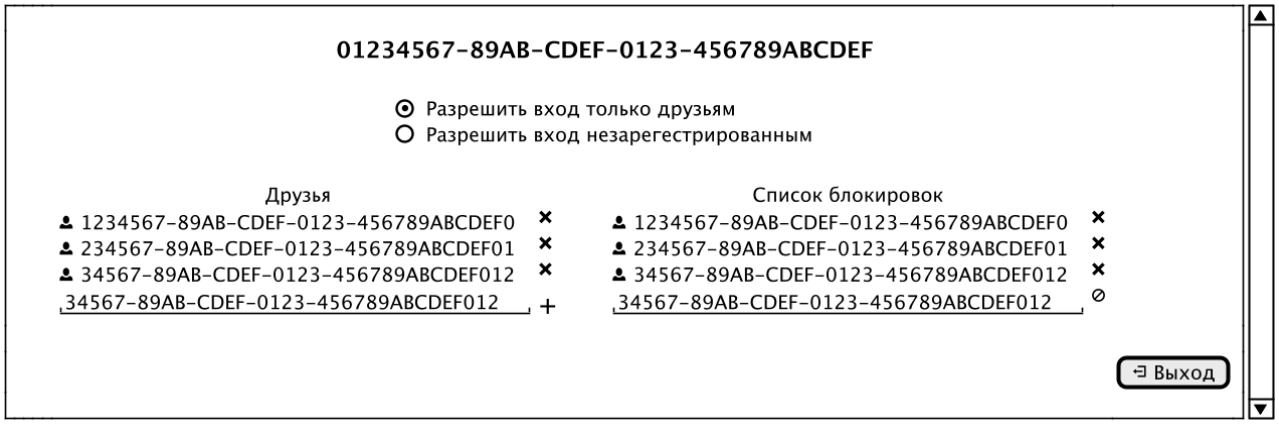

Рисунок 10 – Макет основной страницы

Макеты интерфейса, разработанные в данном разделе, будут использоваться при разработке интерфейса клиентского приложения.

2.3.2 Проектирование игрового приложения

Приложение JPong представляет собой игровой клиент, где можно стать хостом сессии или подключиться к уже запущенной игровой сессии. Это поведение представлено на рисунках 8-10:

Рисунок 11 – Макет страницы присоединения к сессии (авторизованно)

Рисунок 12 – Макет страницы присоединения к сессии (анонимно)

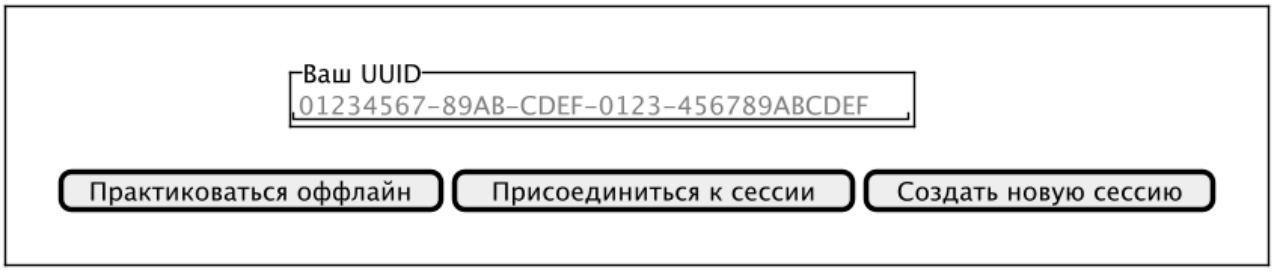

Рисунок 13 – Макет страницы запуска сессии

Т.к. существует два варианта для игры в мультиплеере (как хост и как гость) и они требуют разные наборы данных, были спроектированы две страницы:

1. страница присоединения к сессии включает в себя опциональное поле ввода UUID игрока и поле ввода ID сессии (UUID хоста), особенности:
   a) для точной передачи намерения пользователя быть или не быть анонимным (подключиться без авторизации) у поля «Ваш UUID» добавлен переключатель;
   b) кнопка «Создать новую сессию» должна переводить на страницу запуска сессии;
   c) кнопка «Присоединиться к сессии» должна быть недоступна, если поле ввода UUID не содержит валидного значения и пользователь выбрал входить используя свой аккаунт;
2. страница запуска к сессии содержит лишь поле ввода UUID хоста, особенности:
   a) кнопка «Присоединится к сессии» должна переводить на страницу присоединения к сессии;
   b) кнопка «Создать новую сессию» должна быть недоступна, если поле ввода UUID не содержит валидного значения.

Также должен быть разработан экран самой игры Pong. Экран должен будет содержать по одной ракетке для каждого игрока, счёт и мячик, передвигающийся и прыгающий в реальном времени. Выход из игры будет осуществляться закрытием окна.

Можно выделить три основных части этого клиентского приложения: окно входа, окно игры и сетевая работа. Это разделение позволяет нам разделить игровое приложение на три модуля:

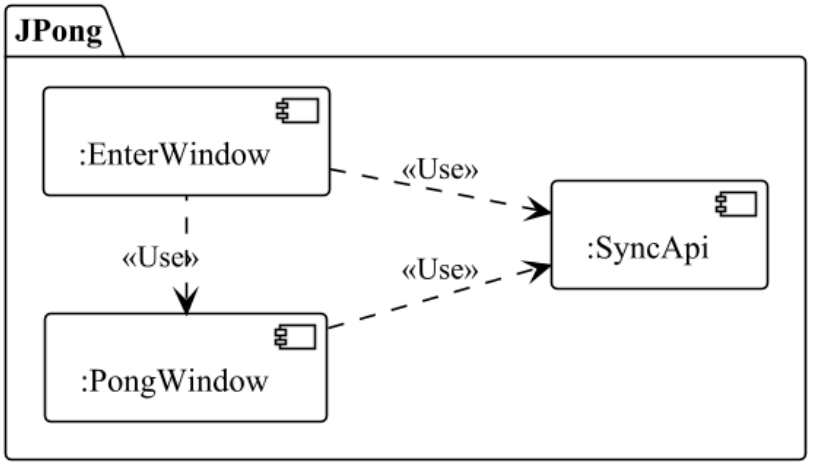

Рисунок 14 – Модули приложения JPong

Макеты интерфейса и разграниченные модули будут использоваться при разработке клиентского приложения.

3 Реализация приложения

В данном разделе курсового проекта будет описана практическая реализация многопользовательской игровой платформы с синхронизацией сессий на основе микросервисов на примере игры Pong.

3.1 Реализация серверной части

В ходе реализации серверной части было разработано четыре микросервиса, описанные в разделах выше: Auth, Hosts, Gateway, Sync.

Каждый микросервис реализован как отдельный проект, но схема связи между ними не изменилась (Рисунок 11):

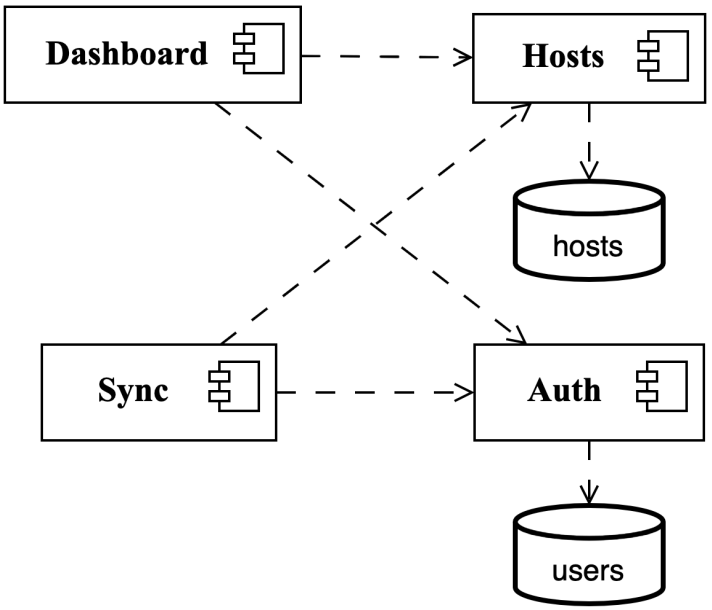

Рисунок 15 – Диаграмма компонентов серверной части

При разработке использовался язык Python 3 и библиотека FastAPI так как микросервисы, на которые разбит этот проект, внутри себя не имеют сложной логики, а Python 3 и FastAPI позволяет реализовать их с наименьшим количеством кода избыточного кода.

FastAPI — это современный высокопроизводительный веб-фреймворк для языка Python, предназначенный для разработки RESTful API и микросервисов.

Он сочетает простоту использования с возможностью создания масштабируемых и надежных решений, что делает его идеальным выбором для проектов, где требуется минимализм кода без ущерба для функциональности. Основные преимущества FastAPI включают встроенную поддержку асинхронности, автоматическую генерацию документации и строгую типизацию данных через интеграцию с Pydantic. [5]

Пример кода обработки точек доступа представлен на листинге 1:

Листинг 1 – Обработка запроса /session/{host_id} из сервиса Sync

```
@app.get('/session/{host_id}')
async def session_id(
    host_id: str,
    services: OtherServicesApi = Depends(provide_services),
    authorization: HTTPAuthorizationCredentials =
     Depends(http_security)
):
  host_id = host_id.upper()

  if
      not authorization or
      not await services.is_authenticated(
        host_id,
        authorization.credentials
      ):
    return Response(status_code=401)

  session = Session(host_id)
  session.source_of_truth_id = host_id
  session.players[host_id] = Session.Player(None, {})
  sessions[host_id] = session
  return {
    'session_id': host_id,
    'source_of_truth_key': session.source_of_truth_key
  }
```

Для хранения данных конфигурации в сервисе Hosts использовалась библиотека TinyDB.

TinyDB — это легковесная документоориентированная база данных на языке Python, предназначенная для хранения данных в формате JSON [6].

В рамках курсовой работы TinyDB была выбрана для управления конфигурацией пользователей из-за своей простоты, минималистичного API и

отсутствия зависимостей от внешних систем. Примером работы с БД послужит функция проверки, есть ли у игрока доступ к сессии гостя, что представлено в листинге 2:

Листинг 2 – Фрагмент кода работы с БД в сервисе Hosts.

```
def welcomes(
  this_uuid: str,
  other_uuid: str | None,
  this: dict | None = None
) -> bool:
  this_uuid = this_uuid.upper()
  other_uuid = other_uuid and other_uuid.upper()

  if this is None:
    this = get(this_uuid)

  if not this['allow_nonames'] and other_uuid is None:
    return False
  if this['only_friends'] and other_uuid not in this['friends']:
    return False
  if other_uuid in this['banlist']:
    return False

  return True
```

3.2 Реализация клиентской части приложения

Клиентская часть системы включает два независимых приложения: Dashboard (панель управления хоста) и JPong (игровой клиент), разработанные с использованием технологий, соответствующих задачам каждого компонента. Для реализации Dashboard был выбран React с TypeScript, что позволило создать динамический и отзывчивый веб-интерфейс для управления конфигурацией. Игровое приложение JPong реализовано на Kotlin с Jetpack Compose — этот выбор обусловлен опытом автора в разработке с использованием этих технологий, а также удобностью фреймворка для создания производительного интерфейса с реальной синхронизацией игрового процесса. Разделение логики управления аккаунтом и игровой сессией, заложенное на этапе проектирования, обеспечило модульность архитектуры: пользователи

могут изменять настройки доступа через Dashboard независимо от работы JPong. В следующих подразделах детально описывается реализация каждого клиентского приложения в соответствии с требованиями, сформулированными в разделе 2.3.

3.2.1 Реализация панели управления хоста

Приложение Dashboard, разработанное на React TS, предоставляет интерфейс для управления конфигурацией аккаунта через взаимодействие с сервисом Gateway.

React TS — это комбинация библиотеки React [7] и языка TypeScript [8], предназначенная для создания динамических веб-приложений с повышенной надежностью и поддерживаемостью кода. React предоставляет компонентный подход к построению интерфейсов, а TypeScript добавляет статическую типизацию поверх JavaScript, что позволяет выявлять ошибки на этапе компиляции и улучшает инструментарий разработчика. React TS часто выбирают для проектов, где важны гибкость, производительность и современная инструментальная поддержка.

Основная страница приложения включает форму авторизации и регистрации, представленную на рисунках 16 и 17 и панель управления настройками доступа, представленную на рисунках 18 и 19.

Рисунок 16 – Скриншот страницы входа без ввода

Рисунок 17 – Скриншот страницы входа

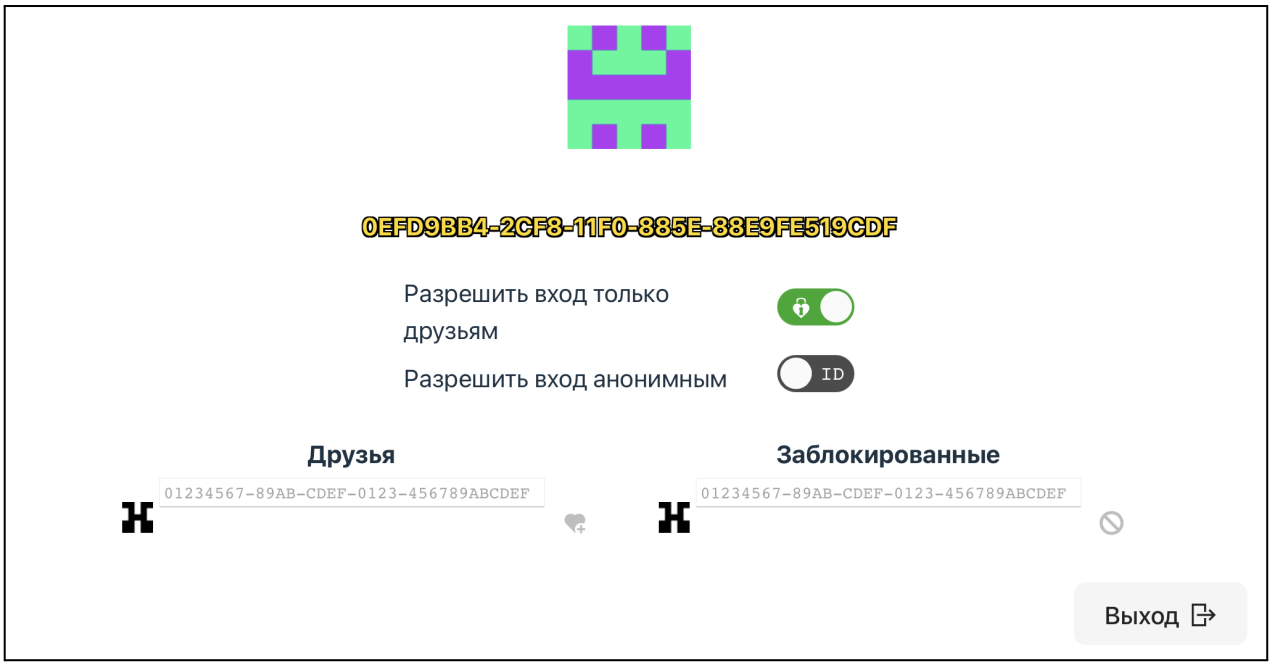

Рисунок 18 – Скриншот страницы новосозданной конфигурации хоста

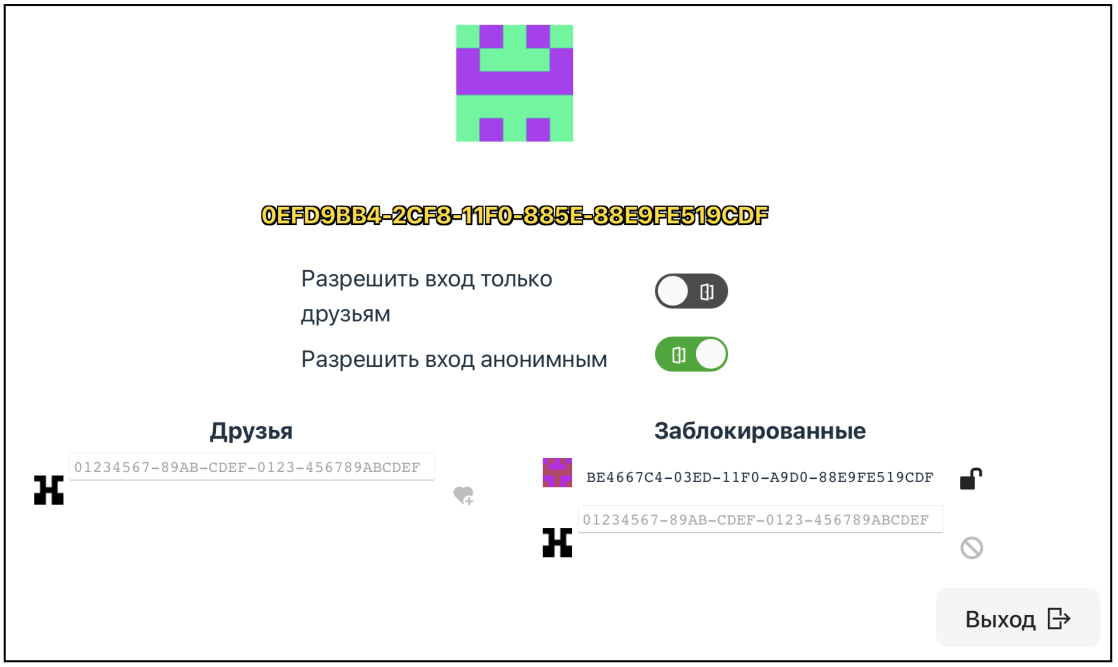

Рисунок 19 – Скриншот страницы конфигурации хоста, принимающего всех за исключением одного пользователя

Состояние интерфейса (например, валидность UUID, доступность кнопок) обрабатываются реактивно [9], что обеспечивает простое в представлении пользователю обновление при изменении полей. Валидация полей (например, проверка формата UUID) выполнена на клиенте для снижения нагрузки на сервер и улучшения UX. Графические элементы интерфейса созданы соответствуя макетам из раздела 2.3.1 с исключением добавления генерируемых уникальных иконок для облегчения распознавания аккаунтов пользователей.

3.2.2 Реализация игрового приложения

Игровой клиент JPong реализован на Kotlin с использованием Jetpack Compose.

Jetpack Compose for Desktop — это современный фреймворк от JetBrains для создания настольных приложений на языке Kotlin, использующий декларативный подход к построению пользовательских интерфейсов. Он является частью экосистемы Jetpack Compose, изначально разработанной для

Android, и позволяет переносить опыт разработки мобильных интерфейсов на десктопные платформы (Windows, macOS, Linux). С помощью Compose for Desktop разработчики могут создавать адаптивные и производительные мультиплатформенные приложения, используя единую кодовую базу [10].

Приложение состоит из двух окон: окна входа и окна самой игры, они продемонстрированы на рисунках 20-22:

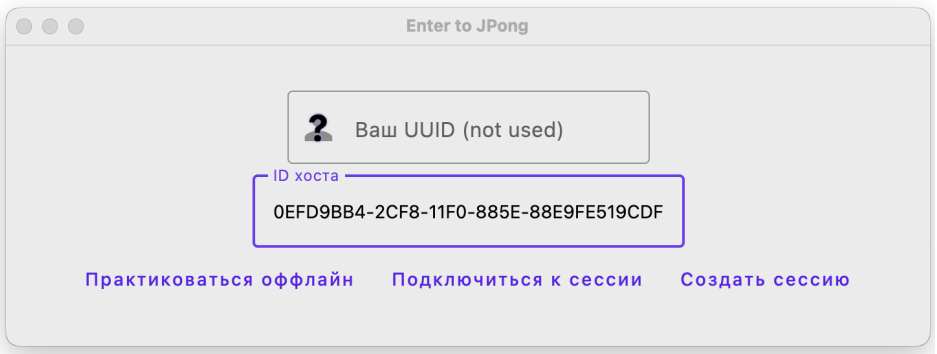

Рисунок 20 – Окно входа

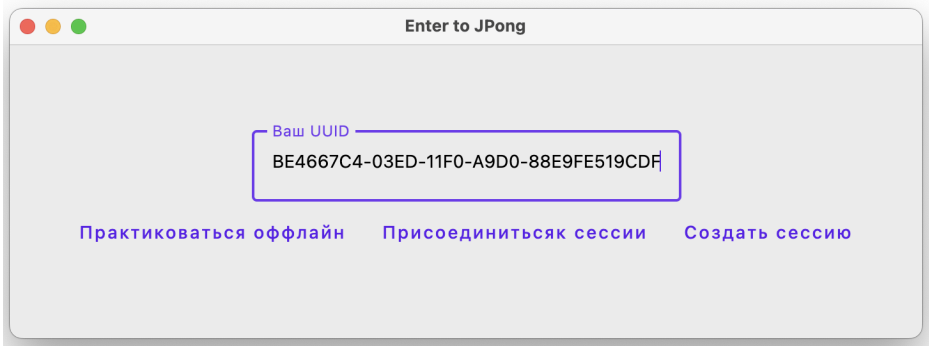

Рисунок 21 – Окно создания сессии

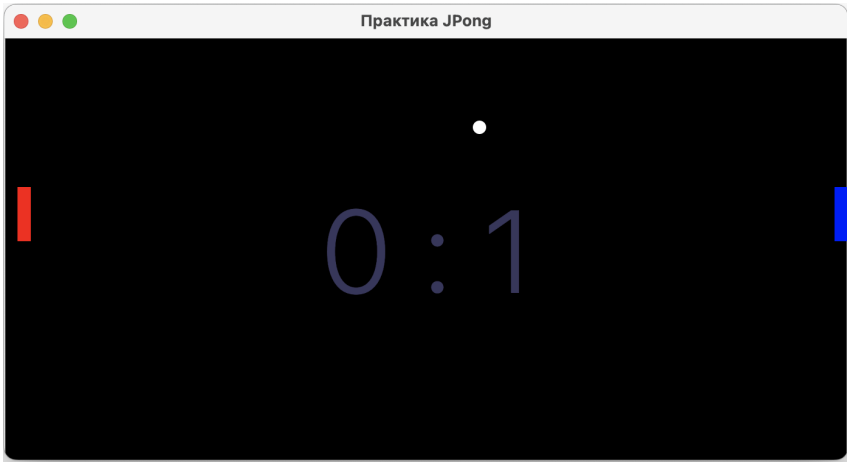

Рисунок 22 – Игровое окно

Код был разделён на три модуля (EnterWindow, PongWindow, SyncApi), что отображено на рисунке 23:

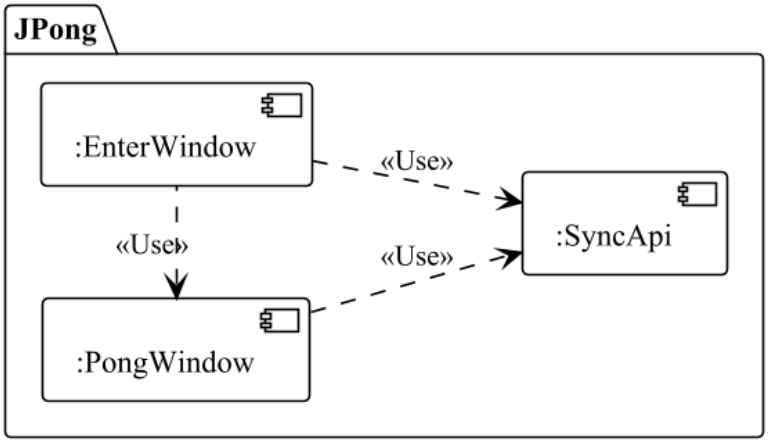

Рисунок 23 – Модули приложения JPong

Модуль SyncApi отвечает за взаимодействие с сессией. Он предоставляет функции для присоединение к ней и создание новой сессии, а для последующей работы с ней существует интерфейс Session, продемонстрированный на листинге 3:

Листинг 3 – Интерфейс Session

```
interface Session<DataT, UpdateT : Apply<DataT>, CommandT> {
  val sessionId: String
  val isHost: Boolean
  val thisPlayer: Player<DataT>
  val players: List<Player<DataT>>
  val playersStateFlow: StateFlow<List<Player<DataT>>>
  val commands: SharedFlow<Belonging<CommandT>>
  val updates: SharedFlow<Belonging<UpdateT>>
  val closeReason: Deferred<CloseReason>

  suspend fun send(command: CommandT)
  suspend fun send(update: UpdateT)

  fun exit()

  interface Apply<T> {
    fun apply(data: T): T
  }

  data class Belonging<T>(val playerId: String, val value: T)

  enum class CloseReason { GracefulExit, NotWelcome }
```

```
}
```

Интерфейс Session позволяет отправлять и подписываться на потоки обновлений от разных пользователей. Что уже в коде модуля PongWindow позволит настроить роли хоста игры (периодически рассылающего актуальное состояние) и гостя (отправляющего лишь свои действия), что продемонстрировано в листинге 4:

Листинг 4 – Функция работы с сессией хоста игры

```
@Composable
fun rememberHostGameHandle(
  initial: GameState,
  session: PongSession,
  settings: Settings
): GameHandle {
  val handle = remember(session) {
    OnlineGameHandle(initial, session)
  }

  LaunchedEffect(handle) {
    handle.launch()
  }

  LaunchedEffect(session) {
    /*логика назначения роли второго игрока*/
  }

  LaunchedEffect(session, handle.myPaddleDirection) {
    session.send(handle.myPaddleDirectionUpdate.serial())
  }

  LaunchedEffect(session) {
    session.updates
      .filter { it.playerId == handle.guestId }
      .map { GameState.Update.of(it.value) }
      .map { it.applyWithEstimation(handle.state, settings) }
      .collectLatest { handle.state = it }
  }

  LaunchedEffect(session) {
    while (isActive) {
      session.send(handle.state.toUpdate().serial())
      delay(settings.syncInterval)
    }
  }

  return handle
}
```

В условиях нестабильной сети, где передача данных между клиентами происходит с задержкой, критически важной становится синхронизация игрового состояния. Для минимизации визуальных артефактов (например, "телепортаций" мяча или ракеток) и синхронизации состояния игры в модуле PongWindow реализован метод applyWithEstimation, корректирующий полученные обновления с учётом времени задержки. Этот метод применяется к получаемым обновлением (session.updates) перед применением их к локальному состоянию (handle.state). Сам код прогнозирования можно прочесть в приложении А, а для консультации использовалось описание работы Source Multiplayer Networking [11].

4 Тестирование

Тестирование программного обеспечения (ПО) – это процесс оценки и проверки программного продукта с целью выявления ошибок и обеспечения его качества. Существует множество классификаций и видов тестирования. В рамках данного курсового проекта было проведено функциональное тестирование.

Функциональное тестирование проверяет соответствие программы или системы заранее определенным функциональным требованиям и ожиданиям. Основная цель функционального тестирования — убедиться, что программа выполняет свои функции и операции согласно спецификациям, а также работает правильно и без сбоев.

Результаты тестирования приложений приведены в таблицах 5 и 6:

Таблица 5 – Результаты тестирования приложения Dashboard

№ теста	Состав теста	Ожидаемый результат	Наблюдаемый результат
1	Проверка авторизации по несуществующему UUID	Не происходит входа в аккаунт, пишется сообщение об ошибке	Соответствует ожидаемым результатам (Рисунок Б.1)
2	Проверка регистрации	Происходит переход на страницу аккаунта	Соответствует ожидаемым результатам (Рисунок Б.2)
3	Проверка изменения и сохранения конфигурации хоста	Совершённые изменения сохраняются	Соответствует ожидаемым результатам (Рисунок Б.3)
4	Проверка вставки существующего пользователя в список друзей или заблокированных	Вставка происходит	Соответствует ожидаемым результатам (Рисунок Б.4)

Таблица 6 – Результаты тестирования приложения JPong

№ теста	Состав теста	Ожидаемый результат	Наблюдаемый результат
1	Создание сессии под несущесвующим UUID	Вывод сообщения об ошибке	Соответствует ожидаемым результатам (Рисунок Б.5)
2	Создание сессии под существующим UUID	Переход в окно игровой сессии	Соответствует ожидаемым результатам (Рисунок Б.6)
3	Анонимное подключение к сессии	Переход в окно игровой сессии	Соответствует ожидаемым результатам (Рисунок Б.7)
4	Не анонимное подключение к сессии	Переход в окно игровой сессии	Соответствует ожидаемым результатам (Рисунок Б.8)
5	Подключение к сессии, где хост не принимает подключения	Вывод сообщения об отказе к подключению	Соответствует ожидаемым результатам (Рисунок Б.9)

Также должна быть проверена корректность игрового процесса Pong в для приложения JPong в многопользовательском режиме (Таблица 7).

Таблица 7 – Результаты тестирования игрового процесса JPong

№ теста	Состав теста	Ожидаемый результат	Наблюдаемый результат
1	Мяч летит и отскакивает от стенок	Мяч летит и отскакивает от стенок	Соответствует ожидаемым результатам
2	Произвести движение ракеткой	Ракетка двигается	Соответствует ожидаемым результатам
3	Игровое состояние синхронизовано у обоих игроков	Игровое состояние синхронизовано у обоих игроков	Соответствует ожидаемым результатам (Рисунок Б.10)

Все основные функции протестированы, приложение работает корректно.

Заключение

В ходе выполнения курсового проекта была разработана информационная система «Многопользовательская игровая платформа с синхронизацией сессий на основе микросервисов». Проект направлен на создание гибкой и масштабируемой архитектуры для управления игровыми сессиями в режиме реального времени, используя классическую игру Pong в качестве примера. Основной упор был сделан на обеспечение безопасности, производительности и удобства взаимодействия пользователей.

Ключевые этапы работы включали анализ требований, проектирование микросервисной архитектуры, реализацию серверной и клиентской частей, а также тестирование системы. На этапе анализа были определены функциональные требования, такие как регистрация пользователей, управление правами доступа и синхронизация игровых действий. Нефункциональные требования, включая низкие задержки и отказоустойчивость, легли в основу выбора технологий.

Результаты проекта подтвердили корректность работы системы:

- пользователи могут создавать сессии, настраивать доступ и взаимодействовать в реальном времени;
- механизмы прогнозирования движений компенсируют сетевые задержки, минимизируя визуальные артефакты;
- перспективы развития системы связаны с расширением функционала, например, добавлением чата или системы рейтингов. Для оптимизации сетевого взаимодействия планируется внедрение протокола UDP, а переход на распределенные базы данных повысит масштабируемость.

Разработанная платформа демонстрирует эффективность микросервисного подхода и служит основой для создания других многопользовательских приложений. Её архитектура адаптирована для будущих обновлений, что обеспечивает долгосрочную актуальность решения.

Список использованных источников

1. Балашова, И. Ю. Современные информационные технологии в проектировании программных систем и комплексов : учеб. пособие / И. Ю. Балашова, Д. В. Такташкин; под ред. П. П. Макарычева. – Пенза : Изд-во ПГУ, 2019. − 106 с.;
2. Pong Game [Электронный ресурс]. URL: https://www.ponggame.org/ (дата обращения: 07.12.2024);
3. Понимание REST API: основные понятия и принципы работы: [электронный ресурс]. URL: https://cloud.yandex.ru/ru/docs/glossary/rest-api (дата обращения: 24.12.2024);
4. Unix Time Stamp: [Электронный ресурс]. URL: https://www.unixtimestamp.com/ (дата обращения: 07.12.2024);
5. FastAPI: FastAPI – это современный, быстрый (высокопроизводительный) веб-фреймворк для создания API с помощью Python на основе стандартных подсказок типа Python.: [электронный ресурс]. URL: https://fastapi.tiangolo.com/ (дата обращения: 07.12.2024);
6. TinyDB: TinyDB – TinyDB - это легкая документоориентированная база данных, оптимизированная для вашего счастья: [Электронный ресурс]. URL: https://github.com/msiemens/tinydb (дата обращения: 07.12.2024);
7. React: Библиотека для веб и нативных пользовательских интерфейсов: [Электронный ресурс]. URL: https://react.dev/ (дата обращения: 07.12.2024);
8. TypeScript: JavaScipt С Синтаксом Для Типов: [Электронный ресурс]. URL: https://www.typescriptlang.org/ (дата обращения: 07.12.2024);
9. Хабр: Введение в реактивное программирование: [Электронный ресурс]. URL: https://habr.com/ru/articles/877332/ (дата обращения: 07.12.2024);

10. Compose Multiplatform – великолепные интерфейсы для любых приложений: [Электронный ресурс]. URL: https://www.jetbrains.com/ru-ru/compose-multiplatform/ (дата обращения: 07.12.2024)

11. Сетевое программирование в Source – Valve Developer Community: [Электронный ресурс]. URL: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking (дата обращения: 07.12.2024).

Приложение А. Код сервисов

Листинг А.1 – Код сервиса Auth

requirements.txt
```
uvicorn~=0.34.0
attrs~=25.1.0
fastapi~=0.115.7
pyjwt~=2.10.1
pydantic~=2.10.6
```

config.py
```
protocol = 'http'
host = '0.0.0.0'
port = 8000

run_url = f'{protocol}://{host}:{port}'

db_path = 'users.db'

auth_key = 'very secret key'
auth_algorithm = 'HS256'
auth_token_lifespan_seconds = 7 * 24 * 60 * 60
```

main.py
```
import sys
from typing import Annotated

import uvicorn
from fastapi import FastAPI, Header, status, Request
from fastapi.responses import JSONResponse, Response
from pydantic import create_model

from auth import Auth, Claims
import users_repository as users

from config import *
if '--test' in sys.argv:
  pass


app = FastAPI()
auth = Auth(auth_key, auth_algorithm, run_url,
token_lifespan_seconds=auth_token_lifespan_seconds)


@app.post('/')
def register() -> create_model('RegisteredUuid', uuid=(str,
None)):
  return {'uuid': users.new_user()}
```

```
@app.get('/login/{uuid}')
def login(uuid: str) -> create_model('LoginToken', token=(str,
None)):
  uuid = uuid.upper()

  print(f'{uuid=}')
  if not users.exists(uuid):
    return JSONResponse(
       content={'message': 'This user does not exists'},
       status_code=404
    )

  return {'token': auth.jwt(uuid)}


@app.get('/renew')
def renew(authorisation: Annotated[str | None, Header()] = None)
-> str:
  if authorisation is None:
    return Response(status_code=status.HTTP_401_UNAUTHORIZED)

  if authorisation.startswith('Bearer '):
    authorisation = authorisation.replace('Bearer ', '')
  result = auth.validate(authorisation)
  if isinstance(result, str):
    return JSONResponse(status_code=status.HTTP_401_UNAUTHORIZED,
content={'message': result})

  claims: Claims = result
  return login(claims.sub)['token']


@app.head('/{uuid}')
def validate_by_jwt(
  uuid: str,
  authorisation: Annotated[str | None, Header()] = None
):
  uuid = uuid.upper()

  if authorisation is None:
    print('401 No authentication token provided', file=sys.stderr)
    return Response(status_code=status.HTTP_401_UNAUTHORIZED)

  validation_error_message =
auth.validate(authorisation.replace('Bearer ', ''), uuid)

  if isinstance(validation_error_message, str):
    print(401, validation_error_message)
    return Response(status_code=status.HTTP_401_UNAUTHORIZED)

  return Response(status_code=200)
```

```
if __name__ == '__main__':
  users.init()
  uvicorn.run(app, host=host, port=port, log_level='trace')
```

auth.py

```
from datetime import datetime, UTC

import jwt
from attrs import define
from jwt import InvalidTokenError, InvalidAudienceError,
ExpiredSignatureError


@define
class Claims:
  iss: str  # issuer
  sub: str  # subject
  aud: str  # audience
  exp: int  # expiration time (utc seconds)
  iat: int  # issued at (utc seconds)
  jti: int  # jwt id

  def to_dict(self):
    return {
      'iss': self.iss,
      'sub': self.sub,
      'aud': self.aud,
      'exp': self.exp,
      'iat': self.iat,
      'jti': self.jti
    }


class Auth:
 def __init__(self, key: str, algorithm: str, issuer: str,
token_lifespan_seconds: int = 900):
    self.key = key
    self.algorithm = algorithm
    self.issuer = issuer
    self.token_lifespan_seconds = token_lifespan_seconds
    self.claims_count = 0

  def jwt(self, subject: str) -> str:
    return jwt.encode(
      Claims(
        iss=self.issuer,
        sub=subject,
        aud=subject,
        exp=
```

```
        int(datetime.now(UTC).timestamp()) +
        self.token_lifespan_seconds,
      iat=int(datetime.now(UTC).timestamp()),
      jti=self._next_jti()
    ).to_dict(),
    key=self.key,
    algorithm=self.algorithm
  )

  def validate(self, token: str, owner: str | None = None) -> str
| Claims:
    try:
       return Claims(**jwt.decode(
         token,
         self.key,
         [self.algorithm],
         audience=owner,
         options={'verify_aud': owner is not None}
        ))
    except ExpiredSignatureError:
      return 'Timeout'
    except InvalidAudienceError as e:
      return 'Impersonation is bad'
    except InvalidTokenError:
      return 'Invalid authentication token'

  def _next_jti(self) -> str:
    self.claims_count = self.claims_count + 1
    return str(hash(str(self.claims_count)))
```

db_schema.sql

```
create table if not exists users (
     user_id   integer not null unique primary key autoincrement,
     uuid blob not null unique
)
```

Листинг А.2 – Код сервиса Hosts

requrements.txt

```
tinydb~=4.8.2
uvicorn~=0.34.0
attrs
fastapi~=0.115.7
pydantic~=2.10.6
```

config.py

```
protocol = 'http'
host = '0.0.0.0'
port = 8001

run_url = f'{protocol}://{host}:{port}'
```

```
db_path = 'hosts.json'
```

main.py

```
from typing import Callable

import uvicorn
from fastapi import FastAPI, Body
from fastapi.responses import Response, JSONResponse
from pydantic import create_model

import hosts_repository as hosts


Host = create_model(
    'Host',
    uuid=(str, 'some-uuid'),
    only_friends=(bool, True),
    allow_nonames=(bool, False),
    friends=(list[str], ['this-gui-uuid', 'other-guy-uuid']),
    banlist=(list[str], ['that-guy-uuid'])
)

app = FastAPI()


def run(host: str | None = None, port: int | None = None, db_path:
str | None = None):
    import config

    hosts.init(db_path or config.db_path)
    uvicorn.run(app, host=host or config.host, port=port or
config.port, log_level='trace')


@app.post('/{host}')
async def create(host: str) -> Host:
    return _err_to_response(hosts.create, host)


@app.get('/{host}')
async def get(host: str) -> Host:
    return _err_to_response(hosts.get, host)


@app.put('/{host}/only_friends')
async def set_only_friends(host: str, only_friends: bool =
Body(embed=True)) -> Response:
    return _err_to_response(hosts.set_only_friends, host,
only_friends)
```

```
@app.put('/{host}/allow_nonames')
async def set_allow_nonames(host: str, allow_nonames: bool =
Body(embed=True)) -> Response:
    return _err_to_response(hosts.set_allow_nonames, host,
allow_nonames)


@app.post('/{host}/friends/{friend}')
async def befriend(host: str, friend: str) -> Response:
    return _err_to_response(hosts.befriend, host, friend)


@app.delete('/{host}/friends/{former_friend}')
async def unfriend(host: str, former_friend: str) -> Response:
    return _err_to_response(hosts.unfriend, host, former_friend)


@app.post('/{host}/banlist/{banned}')
async def ban(host: str, banned: str) -> Response:
    return _err_to_response(hosts.ban, host, banned)


@app.delete('/{host}/banlist/{banned}')
async def unban(host: str, banned: str) -> Response:
    return _err_to_response(hosts.unban, host, banned)


@app.get('/{host}/welcomes/{guest}')
async def welcomes(host: str, guest: str | None = None):
    if guest == 'NONAME':
        guest = None

    return _err_to_response(
        hosts.welcomes, host, guest or None,
        transform=lambda is_welcomes: Response(status_code=(200 if
is_welcomes else 404))
    )


def _err_to_response(action: Callable, *args, transform: Callable
= None, **kwargs):
    try:
        args = [arg.upper() if arg is str else arg for arg in
args]
        got = action(*args, **kwargs)
    except IndexError as e:
        return JSONResponse(status_code=404, content={'message':
'not found this'})
    except LookupError as e:
        return Response(status_code=409)
```

```
    except AssertionError as e:
        return JSONResponse(status_code=404, content={'message':
'not found other'})
    except ValueError as e:
        return Response(status_code=400)

    return transform and transform(got) or got or
Response(status_code=200)


if __name__ == '__main__':
    run()
```

hosts_repository.py

```
from tinydb import TinyDB, Query
import tinydb.operations as op
from tinydb.table import Table


_db: TinyDB
_hosts: Table
_Host: Query


def init(db_path: str | None = None):
    global _db, _hosts, _Host
    import config

    db_path = db_path or config.db_path

    _db = TinyDB(db_path)
    _hosts = _db.table('hosts')
    _Host = Query()


def create(uuid: str):
    uuid = uuid.upper()

    if _hosts.contains(_Host.uuid == uuid):
        raise LookupError()

    _hosts.insert({
        'uuid': uuid,
        'only_friends': True,
        'allow_nonames': False,
        'friends': [],
        'banlist': []
    })
    return get(uuid)
```

```
def get(uuid: str):
    uuid = uuid.upper()

    assert _hosts.contains(_Host.uuid == uuid)
    if got := _hosts.get(_Host.uuid == uuid):
        print(f'{uuid=} {got=}')
        return got


def set_only_friends(uuid: str, only_friends: bool):
    uuid = uuid.upper()

    assert _hosts.contains(_Host.uuid == uuid)
    if not _hosts.update({'only_friends': only_friends},
_Host.uuid == uuid):
        raise IndexError()


def set_allow_nonames(uuid: str, allow_nonames: bool):
    uuid = uuid.upper()

    assert _hosts.contains(_Host.uuid == uuid)
    if not _hosts.update({'allow_nonames': allow_nonames},
_Host.uuid == uuid):
        raise IndexError()


def ban(uuid: str, banned_uuid: str):
    uuid = uuid.upper()
    banned_uuid = banned_uuid.upper()

    if uuid == banned_uuid:
        raise ValueError()
    assert _hosts.contains(_Host.uuid == banned_uuid) and not
_hosts.contains(_Host.whitelist.any([banned_uuid]))
    if not _hosts.update(op.add('banlist', [banned_uuid]),
_Host.uuid == uuid):
        raise IndexError()


def befriend(uuid: str, friend_uuid: str):
    uuid = uuid.upper()
    friend_uuid = friend_uuid.upper()

    if uuid == friend_uuid:
        raise ValueError()
    assert _hosts.contains(_Host.uuid == friend_uuid)
    if not _hosts.update(op.add('friends', [friend_uuid]),
_Host.uuid == uuid):
        raise IndexError()
```

```
def unban(uuid: str, banned_uuid: str):
    uuid = uuid.upper()
    banned_uuid = banned_uuid.upper()

    assert _hosts.contains(_Host.uuid == banned_uuid)
    if not _hosts.update(lambda u:
u['banlist'].remove(banned_uuid), _Host.uuid == uuid):
        raise IndexError()


def unfriend(uuid: str, former_friend_uuid: str):
    uuid = uuid.upper()
    former_friend_uuid = former_friend_uuid.upper()

    assert _hosts.contains(_Host.uuid == former_friend_uuid)
    if not _hosts.update(lambda u:
u['friends'].remove(former_friend_uuid), _Host.uuid == uuid):
        raise IndexError()


def welcomes(this_uuid: str, other_uuid: str | None, this: dict |
None = None) -> bool:
    this_uuid = this_uuid.upper()
    other_uuid = other_uuid and other_uuid.upper()

    if this is None:
        this = get(this_uuid)

    if not this['allow_nonames'] and other_uuid is None:
        print('nonames are not allowed')
        return False
    if this['only_friends'] and other_uuid not in this['friends']:
        print('not a friend is not allowed')
        return False
    if other_uuid in this['banlist']:
        print('baddie')
        return False

    return True


def filter_welcomes(this_uuid: str, other_uuids: list[str]) ->
list[str]:
    this_uuid = this_uuid.upper()
    other_uuids = [uuid.upper() for uuid in other_uuids]

    this = get(this_uuid)
    return [
        other_uuid
        for other_uuid in other_uuids
```

```
        if welcomes(this_uuid, other_uuid, this)
    ]
```

Листинг А.3 – Код сервиса Gateway

requirements.txt

```
tinydb
uvicorn
attrs
fastapi
fastapi-decorators
httpx
pydantic
```

config.py

```
from httpx import AsyncClient

protocol = 'http'
host = '0.0.0.0'
port = 8002

run_url = f'{protocol}://{host}:{port}'
auth_url = 'http://localhost:8000'
hosts_url = 'http://localhost:8001'

async_client = AsyncClient()
```

main.py

```
from typing import Literal, Callable

import uvicorn
from fastapi import FastAPI, Body, Depends, Request
from fastapi.exceptions import RequestValidationError
from fastapi.responses import Response, JSONResponse
from fastapi.routing import APIRoute
from fastapi.security import HTTPBearer,
HTTPAuthorizationCredentials
from fastapi.middleware.cors import CORSMiddleware
from httpx import ConnectError
from pydantic import create_model, BaseModel

import config
from config import auth_url, hosts_url, async_client


class Host(BaseModel):
    uuid: str = 'some-host_uuid'
    only_friends: bool = True
    allow_nonames: bool = False
    friends: list[str] = ['this-gui-host_uuid',
'other-guy-host_uuid']
```

```
    banlist: list[str] = ['that-guy-host_uuid']


class Token(BaseModel):
    token: str


class Message(BaseModel):
    message: str


class NotFoundThisOrOther(BaseModel):
    message: Literal['not found this', 'not found other']


class NotFoundOtherOrOtherOrNotFoundIsPredicateResult(BaseModel):
    message: Literal['not found this', 'not found other',
'~none~']


class LoggingMiddleware(APIRoute):
    def get_route_handler(self) -> Callable:
        original_route_handler = super().get_route_handler()

        async def custom_route_handler(request: Request) ->
Response:
            print(f"route: {request.url}")
            print(f"body: {request.body()}")
            return await original_route_handler(request)

        return custom_route_handler


app = FastAPI()
app.add_middleware(
    CORSMiddleware,
    allow_origins=['*'],
    allow_credentials=True,
    allow_methods=['*'],
    allow_headers=['*']
)
security = HTTPBearer()


@app.post('/hosts', summary='register')
async def register() -> Host:
    try:
        auth_response = await async_client.post(f'{auth_url}/')
    except ConnectError:
        return Response(status_code=503, content='auth')
```

```
    host_uuid = auth_response.json()['uuid']

    try:
        host_response = await
async_client.post(f'{hosts_url}/{host_uuid}')
    except ConnectError:
        return Response(status_code=503, content='hosts')

    if host_response.status_code != 200:
        return Response(status_code=host_response.status_code)
    else:
        return Host(**host_response.json())


@app.get(
    '/hosts/{host}/access_token',
    summary='login by host uuid',
    responses={404: {'model': Message}}
)
async def login(host: str) -> create_model('LoginToken',
token=(str, True)):
    try:
        auth_response = await
async_client.get(f'{auth_url}/login/{host}')
    except ConnectError:
        return Response(status_code=503, content='auth')

    return {'token': auth_response.json()['token']}\
        if auth_response.status_code == 200\
        else Response(status_code=auth_response.status_code)


@app.get(
    '/hosts/access_token/renew',
    summary='renew token',
    responses={404: {'model': Message}}
)
async def renew(authorization: HTTPAuthorizationCredentials =
Depends(security)) -> create_model('LoginToken', token=(str,
True)):
    try:
        response = await async_client.get(f'{auth_url}/renew',
headers={'authorisation': authorization.credentials})
    except ConnectError:
        return Response(status_code=503, content='auth')

    return {'token': response.json()}\
        if response.status_code == 200\
        else Response(status_code=response.status_code)
```

```
@app.get(
    '/hosts/{host}',
    summary='get host by uuid',
    responses={200: {'model': Host}, 404: {'model':
NotFoundThisOrOther}, 401: {'model': Message}}
)
async def get(host: str, authorization:
HTTPAuthorizationCredentials = Depends(security)):
    if auth_err_result := await check_access(host, authorization):
        return auth_err_result

    try:
        response = await async_client.get(f'{hosts_url}/{host}')
    except ConnectError:
        return Response(status_code=503, content='hosts')

    return Host(**response.json()) if response.status_code == 200
else JSONResponse(status_code=response.status_code,
content=response.json())


@app.put(
    '/hosts/{host}/only_friends',
    summary='set is {host} welcomes only friends',
    responses={404: {'model': NotFoundThisOrOther}, 401: {'model':
Message}}
)
async def set_only_friends(
        host: str,
        only_friends: bool = Body(embed=True),
        authorization: HTTPAuthorizationCredentials =
Depends(security)
):
    if auth_err_result := await check_access(host, authorization):
        return auth_err_result

    try:
        response = (await
async_client.put(f'{hosts_url}/{host}/only_friends',
json={'only_friends': only_friends}))
    except ConnectError:
        return Response(status_code=503, content='hosts')

    return Response(status_code=response.status_code,
content=response.content)


@app.put(
    '/hosts/{host}/allow_nonames',
    summary='set is {host} welcomes nonames',
```

```
    responses={404: {'model': NotFoundThisOrOther}, 401: {'model':
Message}}
)
async def set_allow_nonames(
        host: str,
        allow_nonames: bool = Body(embed=True),
        authorization: HTTPAuthorizationCredentials =
Depends(security)
):
    if auth_err_result := await check_access(host, authorization):
        return auth_err_result

    try:
        response = (await
async_client.put(f'{hosts_url}/{host}/allow_nonames',
json={'allow_nonames': allow_nonames}))
    except ConnectError:
        return Response(status_code=503, content='hosts')

    return Response(status_code=response.status_code,
content=response.content)


@app.post(
    '/hosts/{host}/friends',
    summary='befriend',
    responses={404: {'model': NotFoundThisOrOther}, 401: {'model':
Message}, 400: {'model': Message}}
)
async def befriend(host: str, friend: str, authorization:
HTTPAuthorizationCredentials = Depends(security)):
    if auth_err_result := await check_access(host, authorization):
        return auth_err_result

    try:
        response = await
async_client.post(f'{hosts_url}/{host}/friends/{friend}')
    except ConnectError:
        return Response(status_code=503, content='hosts')

    return Response(status_code=response.status_code,
content=response.content)


@app.delete(
    '/hosts/{host}/friends/{former_friend}',
    summary='unfriend',
    responses={404: {'model': NotFoundThisOrOther}, 401: {'model':
Message}}
)
```

```
async def unfriend(host: str, former_friend: str, authorization:
HTTPAuthorizationCredentials = Depends(security)):
    if auth_err_result := await check_access(host, authorization):
        return auth_err_result

    try:
        response = await
async_client.delete(f'{hosts_url}/{host}/friends/{former_friend}')
    except ConnectError:
        return Response(status_code=503, content='hosts')

    return Response(status_code=response.status_code,
content=response.content)


@app.post(
    '/hosts/{host}/banlist',
    summary='ban',
    responses={404: {'model': NotFoundThisOrOther}, 401: {'model':
Message}, 400: {'model': Message}}
)
async def ban(host: str, banned: str, authorization:
HTTPAuthorizationCredentials = Depends(security)):
    if auth_err_result := await check_access(host, authorization):
        return auth_err_result

    try:
        response = await
async_client.post(f'{hosts_url}/{host}/banlist/{banned}')
    except ConnectError:
        return Response(status_code=503, content='hosts')

    return Response(status_code=response.status_code,
content=response.content)


@app.delete(
    '/{host}/banlist/{banned}',
    summary='unban',
    responses={404: {'model': NotFoundThisOrOther}, 401: {'model':
Message}}
)
async def unban(host: str, banned: str, authorization:
HTTPAuthorizationCredentials = Depends(security)):
    if auth_err_result := await check_access(host, authorization):
        return auth_err_result

    try:
        response = await
async_client.delete(f'{hosts_url}/{host}/banlist/{banned}')
    except ConnectError:
```

```
        return Response(status_code=503, content='hosts')

    return Response(status_code=response.status_code,
content=response.content)


async def check_access(host: str, authorization:
HTTPAuthorizationCredentials) -> Response | None:
    token = authorization.credentials

    try:
        response = await async_client.head(f'{auth_url}/{host}',
headers={'authorisation': token})
    except ConnectError:
        return Response(status_code=503, content='auth')

    if response.is_error:
        return Response(status_code=response.status_code)


@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request: Request, exc:
RequestValidationError):
    exc_str = f'{exc}'.replace('\n', ' ').replace('   ', ' ')
    content = {'status_code': 10422, 'message': exc_str, 'data':
await request.json()}
    print(422, f'{content=}')
    return JSONResponse(content=content, status_code=422)


if __name__ == '__main__':
    uvicorn.run(app, host=config.host, port=config.port,
log_level='trace')
```

Листинг А.4 – Код сервиса Sync

requirements.txt

```
typing_extensions
uvicorn[standard]
attrs~=24.3.0
fastapi~=0.115.12
httpx
```

main.py

```
import asyncio
import logging
import random
import sys
import traceback

import uvicorn
```

```
from attrs import define
from fastapi import FastAPI, WebSocket, WebSocketDisconnect,
Depends, Header
from fastapi.responses import Response
from fastapi.security import HTTPAuthorizationCredentials,
HTTPBearer
from fastapi.middleware.cors import CORSMiddleware
from httpx import AsyncClient

from other_services_api import OtherServicesApi
from util import merge

DISCONNECT_TIMEOUT_SECONDS = 10
SOURCE_OF_TRUTH_LAUNCHER = 'cmd /k "start ..\\run.cmd'
AUTH_URL = 'http://0.0.0.0:8000'
HOSTS_URL = 'http://0.0.0.0:8001'

HOST = '0.0.0.0'
PORT = 8765

logger = logging.getLogger(__name__)
app = FastAPI()
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
http_security = HTTPBearer(auto_error=False)

salt = hash(random.random())
sessions: dict[str, 'Session'] = {}

services = OtherServicesApi(AsyncClient())

generated_ids: set[str] = set()


class Session:
    """
    Класс для управления игровой сессией

    Отвечает за:
    - Управление игроками в сессии
    - Обработку сообщений
    - Управление подключениями
    - Синхронизацию данных
    """

    def __init__(self, host_id: str):
```

```
        """
        Инициализация новой сессии
        :param host_id: (str): ID хозяина сессии
        """

        self.players: dict[str, 'Session.Player'] = {}
        self.host_id: str = host_id
        self.source_of_truth_id: str | None = None
        self.source_of_truth_key = ''.join(
            chr(random.randint(ord('A'), ord('Z') + 1))
            for _ in range(20)
        )
        self.running = True

    def consume(self, user_id: str, datum: dict):
        """
        Обработка данных от игрока
        :param user_id: (str): ID игрока
        :param datum: (dict): Данные для обработки
        :raises: ValueError: Если данные некорректны
        """

        source_of_truth_pretends_to_be_someone_else =\
            user_id == self.source_of_truth_id and\
            'player_id' in datum and\
            datum['player_id'] != user_id

        if source_of_truth_pretends_to_be_someone_else:
            self.consume(datum['player_id'], datum)
            return

        if 'update' in datum:
            self.players[user_id].consume(datum['update'])
            for player in self.players.values():
                if player.connection:

asyncio.create_task(player.connection.send_json(datum |
{'player_id': user_id}))

        elif 'command' in datum:
            for player_id, player in self.players.items():
                if player.connection:

asyncio.create_task(player.connection.send_json(datum |
{'player_id': user_id}))

        else:
            raise ValueError('Unknown type of received data')

    def join(self, user_id: str, connection: WebSocket):
        """
```

```
        Добавление нового игрока в сессию
        :param user_id: (str): ID игрока
        :param connection: (WebSocket): WebSocket подключение
        """

        self.players[user_id] = self.Player(connection=connection,
data={})

    def reconnect(self, user_id: str, connection: WebSocket):
        """
        Обработка переподключения игрока
        :param user_id: (str): ID игрока
        :param connection: (WebSocket): Новое WebSocket
подключение
        """

        self.players[user_id].connection = connection

    def disconnect(self, user_id: str):
        """
        Обработка отключения игрока
        :param user_id: (str): ID игрока
        """

        async def timeout():
            await asyncio.sleep(DISCONNECT_TIMEOUT_SECONDS)

            if not self.players[user_id].connection:
                del self.players[user_id]
                await self.reinit()

        self.players[user_id].connection = None

        if user_id == self.host_id:
            self.finish()

        asyncio.create_task(timeout())

    async def reinit(self):
        """
        Реинициализация сессии
        """

        for player_id, player in self.players.items():
            if not player.connection:
                continue

            await player.connection.send_json({
                'player_id': player_id,
                'init': {
                    player_id: player.data
```

```
                    for player_id, player in self.players.items()
                }
            })

    def finish(self):
        """
        Завершение сессии
        """

        self.running = False
        session_id = next(session_id for session_id, session in
sessions.items() if session == self)
        del sessions[session_id]

    @define
    class Player:
        """
        Класс для хранения информации об игроке
        :param connection: (WebSocket): WebSocket подключение
        :param data: (dict): Данные игрока
        """

        connection: WebSocket
        data: dict

        def consume(self, update: dict):
            self.data = merge(self.data, update)


async def provide_services() -> OtherServicesApi:
    """
    Зависимость для внешних сервисов.
    :return: OtherServicesApi: Экземпляр API для внешних сервисов
    """

    return services


@app.get('/')
async def get():
    """
    Получение информации о всех активных сессиях.
    :return: json-dict: Словарь с данными всех сессий
    """

    return {
        session_id: {
            player_id: player.data
            for player_id, player in session.players.items()
        }
        for session_id, session in sessions.items()
```

```
    }


@app.get('/session/{host_id}')
async def session_id(
        host_id: str,
        services: OtherServicesApi = Depends(provide_services),
        authorization: HTTPAuthorizationCredentials =
Depends(http_security)
):
    """
    Создание новой сессии.
    :param host_id: (str): ID хозяина сессии
    :param services: (OtherServicesApi): API для внешних сервисов
    :param authorization: (HTTPAuthorizationCredentials): Токен
аутентификации
    :return: json-dict: Информация о созданной сессии
    :raises: Response 403 если аутентификация не пройдена
    """

    host_id = host_id.upper()

    if not authorization or not await
services.is_authenticated(host_id, authorization.credentials):
        return Response(status_code=401)

    session = Session(host_id)
    session.source_of_truth_id = host_id
    session.players[host_id] = Session.Player(None, {})
    sessions[host_id] = session
    return {'session_id': host_id, 'source_of_truth_key':
session.source_of_truth_key}


@app.websocket('/session/{session_id}')
async def websocket_endpoint(
        player_ws: WebSocket,
        session_id: str,
        player_id: str,
        authorisation: str | None = Header(default=None),
        services: OtherServicesApi = Depends(provide_services)
):
    session_id = session_id.upper()
    player_id = player_id.upper()

    await player_ws.accept()

    if player_id == 'NONAME':
        player_id = None

    if not player_id:
```

```
        pass  # there is nothing to authenticate
    elif not authorisation:
        await player_ws.close(code=3003, reason=f'[http 401],
authentication token is not provided')
        return
    elif not await services.is_authenticated(player_id,
authorisation):
        await player_ws.close(code=3003, reason=f'[http 401], bad
auth')
        return

    session = sessions[session_id]
    if not await services.is_welcome(session.host_id, player_id or
'NONAME'):
        print(f'[http 403], player\'{player_id}\' is not welcome
in session\'{session_id}\'')
        await player_ws.close(
            code=3003,
            reason=f'[http 403], player\'{player_id}\' is not
welcome in session\'{session_id}\''
        )
        return

    if not player_id:
        player_id =
str(hash(f'{salt}{hash(session)}{len(session.players)}'))
        generated_ids.add(player_id)

    session.join(player_id, player_ws)

    await websocket_player_communication(player_ws, session,
player_id)


@app.websocket('/session/{session_id}/player/{player_id}')
async def websocket_endpoint_reconnect(
        player_ws: WebSocket,
        session_id: str,
        player_id: str,
        authentication_info: str | None = Header(default=None),
        authorisation: str | None = Header(default=None),
        services: OtherServicesApi = Depends(provide_services)
):
    source_of_truth_key = authentication_info
    session_id = session_id.upper()
    player_id = player_id.upper()

    await player_ws.accept()

    session = sessions[session_id]
    if player_id not in session.players:
```

```
        return websocket_endpoint(player_ws, session_id,
player_id, authorisation)

    if player_id in generated_ids:
        pass  # this id is not of a registered user
    elif not authorisation:
        await player_ws.close(code=3003, reason=f'[http 401],
authentication token is not provided')
        return
    elif not await services.is_authenticated(player_id,
authorisation):
        await player_ws.close(code=3003, reason=f'[http 401], bad
auth')
        return

    if player_id == session.source_of_truth_id and
source_of_truth_key != session.source_of_truth_key:
        await player_ws.close(
            code=3003,
            reason=f'[http 403], you\'{player_id}\' '
                   'falsely pretending to be a source '
                   'of truth, be ashamed of yourself'
        )
        return

    session.reconnect(player_id, player_ws)

    await websocket_player_communication(player_ws, session,
player_id)


async def websocket_player_communication(
        player_ws: WebSocket,
        session: Session,
        player_id: str,
):
    """
    Обрабатывает коммуникацию с подключенным игроком через
WebSocket.

    Эта функция слушает входящие сообщения от игрока и
обрабатывает их. Она также управляет
    отключением и любыми потенциальными ошибками во время
коммуникации.
    :param player_ws: (WebSocket): WebSocket-соединение для
игрока.
    :param session: (Session): Текущий объект игровой сессии.
    :param player_id: (str): Уникальный идентификатор
подключенного игрока.
    """
```

```
    exit_code = 1000
    exit_reason: str | None = None

    # noinspection PyBroadException
    try:
        await session.reinit()

        while True:
            datum = await player_ws.receive_json()
            if 'type' in datum and datum['type'] ==
'websocket.close':
                break
            session.consume(player_id, datum)

    except WebSocketDisconnect:
        pass
    except ValueError as e:
        exit_code = 4000
        exit_reason = '\n'.join(e.args)
    except Exception:
        print(traceback.format_exc(), file=sys.stderr)
    finally:
        session.disconnect(player_id)
        try:
            await player_ws.close(code=exit_code,
reason=exit_reason)
        except RuntimeError:
            # connection can be closed outside of this function
            # , this will lead to this error
            pass


if __name__ == '__main__':
    uvicorn.run(app, host=HOST, port=PORT)
```

other_services_api.py

```
import asyncio
import logging
import random
import sys
import traceback

import uvicorn
from attrs import define
from fastapi import FastAPI, WebSocket, WebSocketDisconnect,
Depends, Header
from fastapi.responses import Response
from fastapi.security import HTTPAuthorizationCredentials,
HTTPBearer
from fastapi.middleware.cors import CORSMiddleware
```

```
from httpx import AsyncClient

from other_services_api import OtherServicesApi
from util import merge

DISCONNECT_TIMEOUT_SECONDS = 10
SOURCE_OF_TRUTH_LAUNCHER = 'cmd /k "start ..\\run.cmd'
AUTH_URL = 'http://0.0.0.0:8000'
HOSTS_URL = 'http://0.0.0.0:8001'

HOST = '0.0.0.0'
PORT = 8765

logger = logging.getLogger(__name__)
app = FastAPI()
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
http_security = HTTPBearer(auto_error=False)

salt = hash(random.random())
sessions: dict[str, 'Session'] = {}

services = OtherServicesApi(AsyncClient())

generated_ids: set[str] = set()


class Session:
    """
    Класс для управления игровой сессией

    Отвечает за:
    - Управление игроками в сессии
    - Обработку сообщений
    - Управление подключениями
    - Синхронизацию данных
    """

    def __init__(self, host_id: str):
        """
        Инициализация новой сессии
        :param host_id: (str): ID хозяина сессии
        """

        self.players: dict[str, 'Session.Player'] = {}
        self.host_id: str = host_id
```

```
        self.source_of_truth_id: str | None = None
        self.source_of_truth_key = ''.join(
            chr(random.randint(ord('A'), ord('Z') + 1))
            for _ in range(20)
        )
        self.running = True

    def consume(self, user_id: str, datum: dict):
        """
        Обработка данных от игрока
        :param user_id: (str): ID игрока
        :param datum: (dict): Данные для обработки
        :raises: ValueError: Если данные некорректны
        """

        source_of_truth_pretends_to_be_someone_else =\
            user_id == self.source_of_truth_id and\
            'player_id' in datum and\
            datum['player_id'] != user_id

        if source_of_truth_pretends_to_be_someone_else:
            self.consume(datum['player_id'], datum)
            return

        if 'update' in datum:
            self.players[user_id].consume(datum['update'])
            for player in self.players.values():
                if player.connection:

asyncio.create_task(player.connection.send_json(datum |
{'player_id': user_id}))

        elif 'command' in datum:
            for player_id, player in self.players.items():
                if player.connection:

asyncio.create_task(player.connection.send_json(datum |
{'player_id': user_id}))

        else:
            raise ValueError('Unknown type of received data')

    def join(self, user_id: str, connection: WebSocket):
        """
        Добавление нового игрока в сессию
        :param user_id: (str): ID игрока
        :param connection: (WebSocket): WebSocket подключение
        """

        self.players[user_id] = self.Player(connection=connection,
data={})
```

```
    def reconnect(self, user_id: str, connection: WebSocket):
        """
        Обработка переподключения игрока
        :param user_id: (str): ID игрока
        :param connection: (WebSocket): Новое WebSocket
подключение
        """

        self.players[user_id].connection = connection

    def disconnect(self, user_id: str):
        """
        Обработка отключения игрока
        :param user_id: (str): ID игрока
        """

        async def timeout():
            await asyncio.sleep(DISCONNECT_TIMEOUT_SECONDS)

            if not self.players[user_id].connection:
                del self.players[user_id]
                await self.reinit()

        self.players[user_id].connection = None

        if user_id == self.host_id:
            self.finish()

        asyncio.create_task(timeout())

    async def reinit(self):
        """
        Реинициализация сессии
        """

        for player_id, player in self.players.items():
            if not player.connection:
                continue

            await player.connection.send_json({
                'player_id': player_id,
                'init': {
                    player_id: player.data
                    for player_id, player in self.players.items()
                }
            })

    def finish(self):
        """
        Завершение сессии
```

```
        """

        self.running = False
        session_id = next(session_id for session_id, session in
sessions.items() if session == self)
        del sessions[session_id]

    @define
    class Player:
        """
        Класс для хранения информации об игроке
        :param connection: (WebSocket): WebSocket подключение
        :param data: (dict): Данные игрока
        """

        connection: WebSocket
        data: dict

        def consume(self, update: dict):
            self.data = merge(self.data, update)


async def provide_services() -> OtherServicesApi:
    """
    Зависимость для внешних сервисов.
    :return: OtherServicesApi: Экземпляр API для внешних сервисов
    """

    return services


@app.get('/')
async def get():
    """
    Получение информации о всех активных сессиях.
    :return: json-dict: Словарь с данными всех сессий
    """

    return {
        session_id: {
            player_id: player.data
            for player_id, player in session.players.items()
        }
        for session_id, session in sessions.items()
    }


@app.get('/session/{host_id}')
async def session_id(
        host_id: str,
        services: OtherServicesApi = Depends(provide_services),
```

```
        authorization: HTTPAuthorizationCredentials =
Depends(http_security)
):
    """
    Создание новой сессии.
    :param host_id: (str): ID хозяина сессии
    :param services: (OtherServicesApi): API для внешних сервисов
    :param authorization: (HTTPAuthorizationCredentials): Токен
аутентификации
    :return: json-dict: Информация о созданной сессии
    :raises: Response 403 если аутентификация не пройдена
    """

    host_id = host_id.upper()

    if not authorization or not await
services.is_authenticated(host_id, authorization.credentials):
        return Response(status_code=401)

    session = Session(host_id)
    session.source_of_truth_id = host_id
    session.players[host_id] = Session.Player(None, {})
    sessions[host_id] = session
    return {'session_id': host_id, 'source_of_truth_key':
session.source_of_truth_key}


@app.websocket('/session/{session_id}')
async def websocket_endpoint(
        player_ws: WebSocket,
        session_id: str,
        player_id: str,
        authorisation: str | None = Header(default=None),
        services: OtherServicesApi = Depends(provide_services)
):
    session_id = session_id.upper()
    player_id = player_id.upper()

    await player_ws.accept()

    if player_id == 'NONAME':
        player_id = None

    if not player_id:
        pass  # there is nothing to authenticate
    elif not authorisation:
        await player_ws.close(code=3003, reason=f'[http 401],
authentication token is not provided')
        return
    elif not await services.is_authenticated(player_id,
authorisation):
```

```
        await player_ws.close(code=3003, reason=f'[http 401], bad
auth')
        return

    session = sessions[session_id]
    if not await services.is_welcome(session.host_id, player_id or
'NONAME'):
        print(f'[http 403], player\'{player_id}\' is not welcome
in session\'{session_id}\'')
        await player_ws.close(
            code=3003,
            reason=f'[http 403], player\'{player_id}\' is not
welcome in session\'{session_id}\''
        )
        return

    if not player_id:
        player_id =
str(hash(f'{salt}{hash(session)}{len(session.players)}'))
        generated_ids.add(player_id)

    session.join(player_id, player_ws)

    await websocket_player_communication(player_ws, session,
player_id)


@app.websocket('/session/{session_id}/player/{player_id}')
async def websocket_endpoint_reconnect(
        player_ws: WebSocket,
        session_id: str,
        player_id: str,
        authentication_info: str | None = Header(default=None),
        authorisation: str | None = Header(default=None),
        services: OtherServicesApi = Depends(provide_services)
):
    source_of_truth_key = authentication_info
    session_id = session_id.upper()
    player_id = player_id.upper()

    await player_ws.accept()

    session = sessions[session_id]
    if player_id not in session.players:
        return websocket_endpoint(player_ws, session_id,
player_id, authorisation)

    if player_id in generated_ids:
        pass  # this id is not of a registered user
    elif not authorisation:
```

```
        await player_ws.close(code=3003, reason=f'[http 401],
authentication token is not provided')
        return
    elif not await services.is_authenticated(player_id,
authorisation):
        await player_ws.close(code=3003, reason=f'[http 401], bad
auth')
        return

    if player_id == session.source_of_truth_id and
source_of_truth_key != session.source_of_truth_key:
        await player_ws.close(
            code=3003,
            reason=f'[http 403], you\'{player_id}\' '
                   'falsely pretending to be a source '
                   'of truth, be ashamed of yourself'
        )
        return

    session.reconnect(player_id, player_ws)

    await websocket_player_communication(player_ws, session,
player_id)


async def websocket_player_communication(
        player_ws: WebSocket,
        session: Session,
        player_id: str,
):
    """
    Обрабатывает коммуникацию с подключенным игроком через
WebSocket.

    Эта функция слушает входящие сообщения от игрока и
обрабатывает их. Она также управляет
    отключением и любыми потенциальными ошибками во время
коммуникации.
    :param player_ws: (WebSocket): WebSocket-соединение для
игрока.
    :param session: (Session): Текущий объект игровой сессии.
    :param player_id: (str): Уникальный идентификатор
подключенного игрока.
    """

    exit_code = 1000
    exit_reason: str | None = None

    # noinspection PyBroadException
    try:
        await session.reinit()
```

```
        while True:
            datum = await player_ws.receive_json()
            if 'type' in datum and datum['type'] ==
'websocket.close':
                break
            session.consume(player_id, datum)

    except WebSocketDisconnect:
        pass
    except ValueError as e:
        exit_code = 4000
        exit_reason = '\n'.join(e.args)
    except Exception:
        print(traceback.format_exc(), file=sys.stderr)
    finally:
        session.disconnect(player_id)
        try:
            await player_ws.close(code=exit_code,
reason=exit_reason)
        except RuntimeError:
            # connection can be closed outside of this function
            # , this will lead to this error
            pass


if __name__ == '__main__':
    uvicorn.run(app, host=HOST, port=PORT)
```

util.py

```
import logging
from typing import Sequence


def merge(data, update):
    if data is None:
        return update
    if update is None:
        return data
    if isinstance(data, dict) and isinstance(update, dict):
        snapshot = {
            k: (
                merge(data.get(k, update.get(k, None)),
update.get(k, data.get(k, None)))
                if is_collection(update.get(k, None))
                else
                update.get(k, data.get(k, None))
            )
            for k in data | update
        }
```

```
            logging.getLogger(None).debug(f'{data=} {snapshot=}')
            return data | snapshot
        if isinstance(data, list) and isinstance(update, list):
            return sorted(data + update)
        return update


def is_collection(obj):
    return isinstance(obj, dict) or isinstance(obj, list)
```

Приложение Б. Скриншоты результатов тестирования

Рисунок Б.1 – Вывод сообщения об ошибке при авторизации по несуществующему UUID

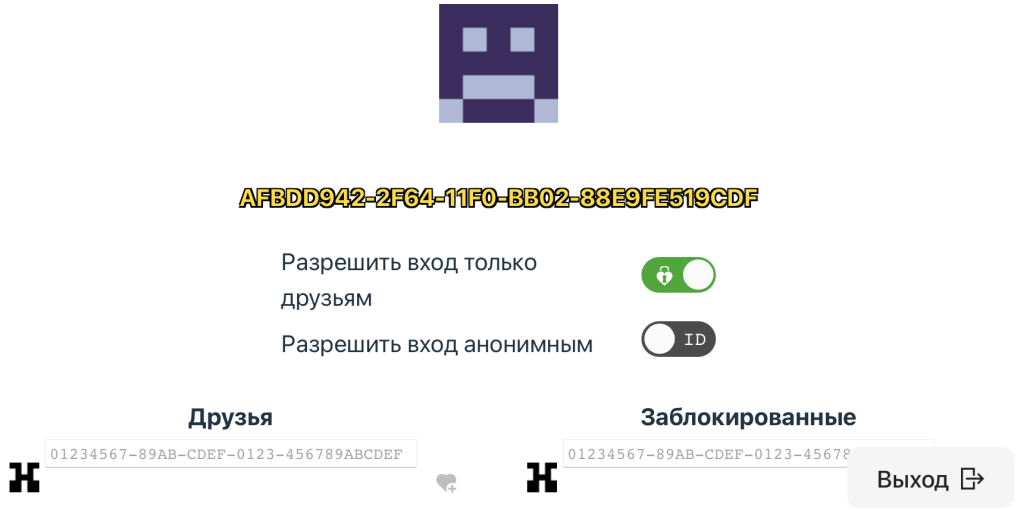

Рисунок Б.2 – Результат регистрации

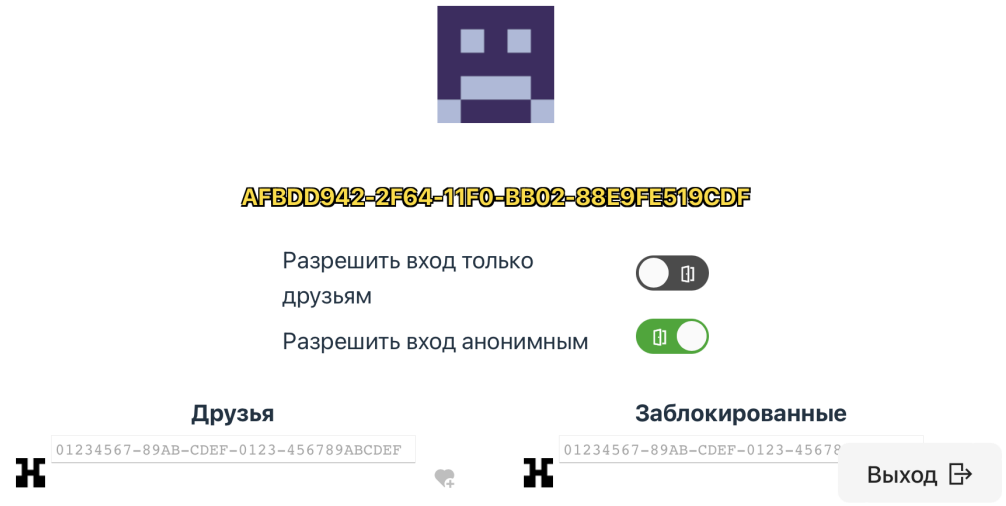

Рисунок Б.3 – Изменённая конфигурация хоста

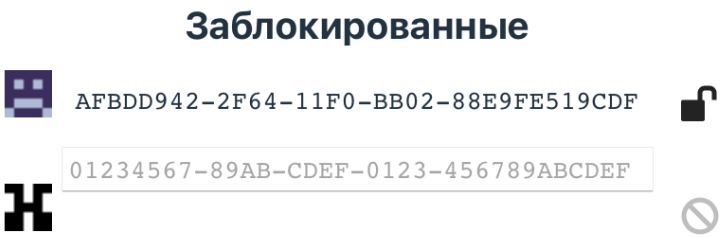

Рисунок Б.4 – Фрагмент Dashboard со вставленным заблокированным пользователем

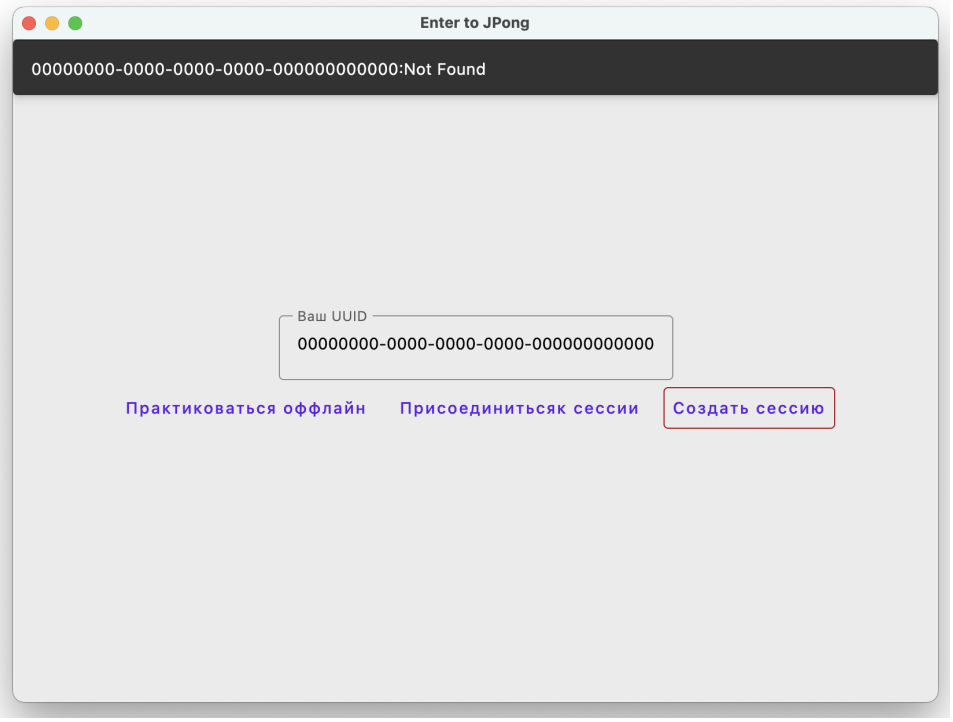

Рисунок Б.4 – Вывод сообщения об ошибке при попытке создания сессии под несуществующим UUID

Рисунок Б.5 – Окно игровой сессии для хоста

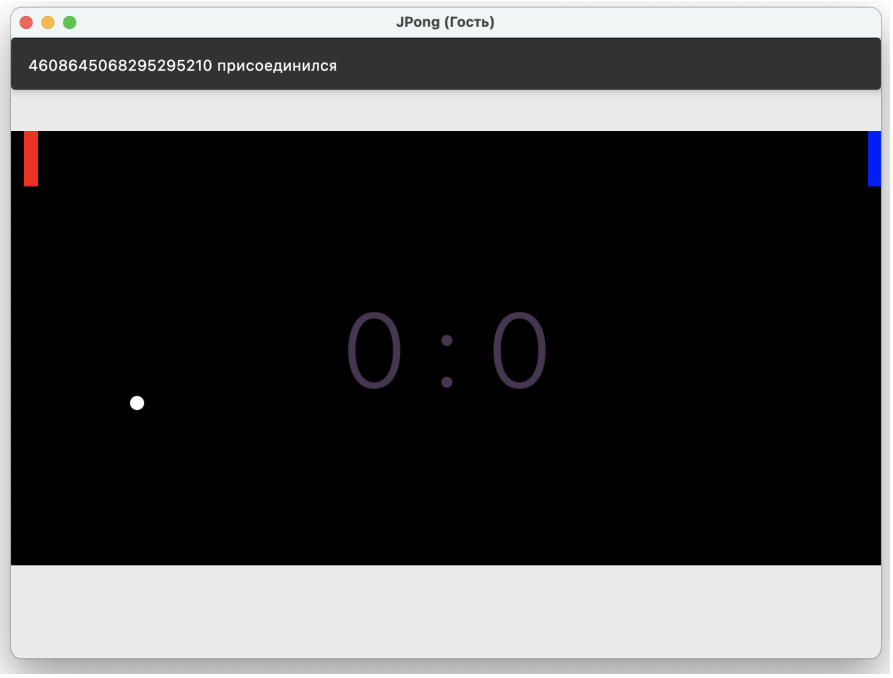

Рисунок Б.6 – Окно игровой сессии для анонимного гостя

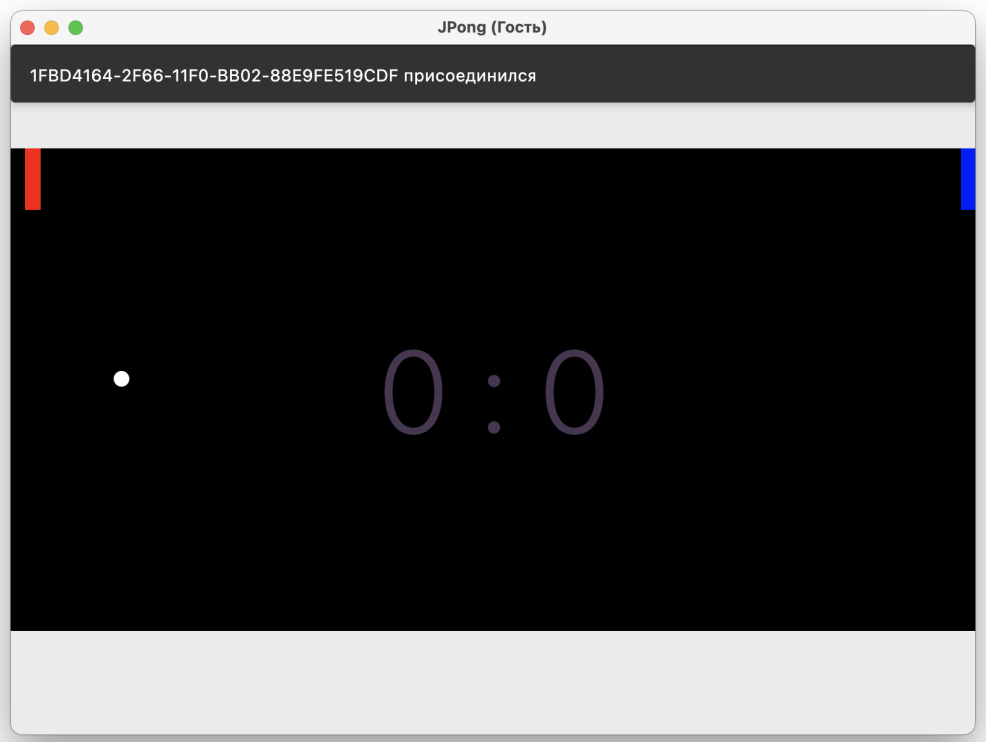

Рисунок Б.7 – Окно игровой сессии для не анонимного гостя

Рисунок Б.8 – Окно игровой сессии при не подключении

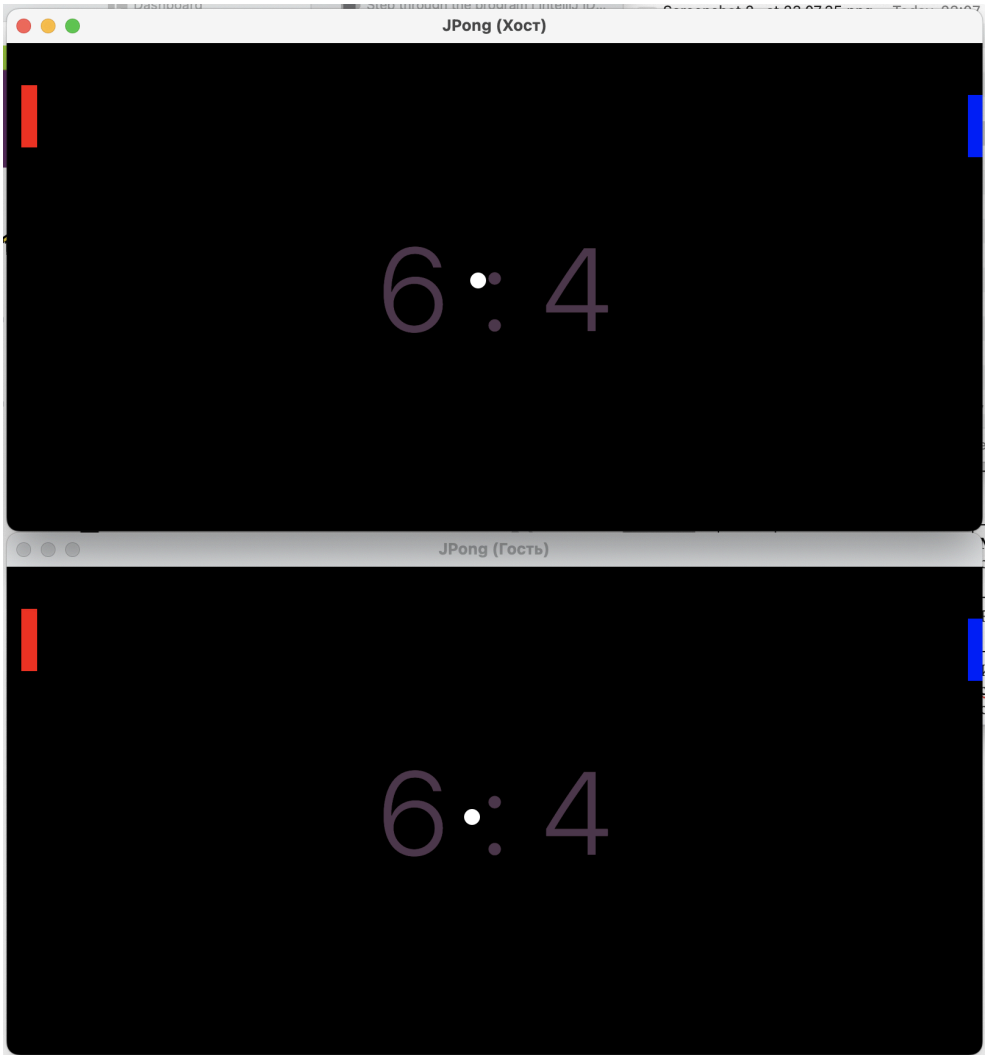

Рисунок Б.9 – Синхронизированность окон хоста и гостя