МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ "МИРЭА — РОССИЙСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ"
(РТУ МИРЭА)

ОТЧЁТ
ПО ПРАКТИЧЕСКИМ РАБОТАМ №5-8
по теме:
МИКРОСЕРВИСНАЯ АРХИТЕКТУРА

Руководитель НИР,
Преподаватель _____ Запорожских А.И.
подпись, дата

Москва 2025

СОДЕРЖАНИЕ

1 Практическая работа №5

1.1 Цель работы

Изучение работы с очередями сообщений RabbitMQ и реализация различных схем взаимодействия сервисов с использованием различных типов обменников.

1.2 Ход работы

1.2.1 Задание 1: Создание очередей

В рамках первого задания были созданы три типа очередей с различными характеристиками:

1.2.1.1 Эксклюзивная очередь

Эксклюзивная очередь доступна только для текущего соединения и автоматически удаляется при его закрытии. Такая очередь используется для временных операций, например, для поиска игры в каталоге.

Команда для создания:

```
rabbitmqadmin declare queue name=game_search_temporary exclusive=true
```

Характеристики:

- Название: `game_search_temporary`
- Exclusive: `true`
- Использование: временные операции поиска

1.2.1.2 Durable очередь

Durable очередь сохраняется на диске и переживает перезапуск сервера RabbitMQ. Такая очередь используется для критически важных событий, которые не должны быть потеряны, например, для события «Пользователь зарегистрирован».

Команда для создания:

```
rabbitmqadmin declare queue name=user_registered durable=true
```

Характеристики:

– Название: `user_registered`

– Durable: `true`

– Использование: события регистрации пользователей

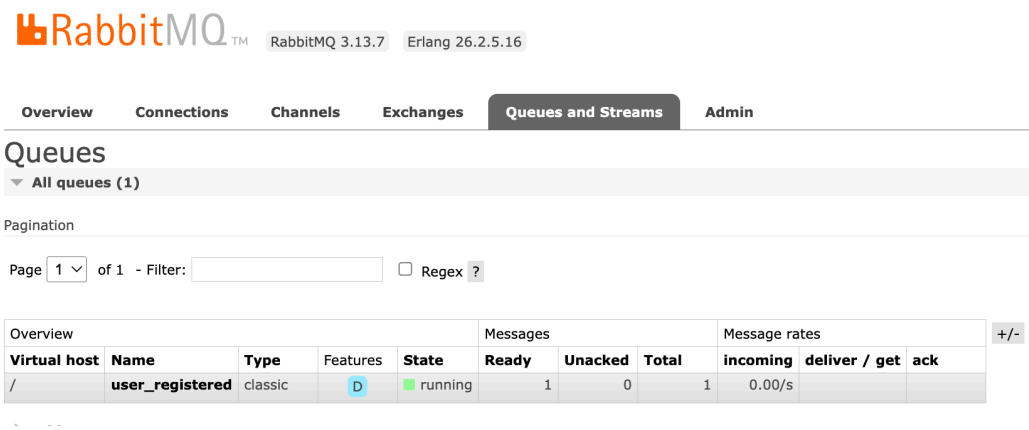

Рисунок 1 — Durable очередь user_registered в RabbitMQ Management UI

1.2.1.3 Автоудаляемая очередь

Автоудаляемая очередь автоматически удаляется, когда на ней не остается активных потребителей. Такая очередь используется для временных уведомлений, которые не требуют долгосрочного хранения.

Команда для создания:

```
rabbitmqadmin declare queue name=notification_temporary
auto_delete=true
```

Характеристики:

– Название: `notification_temporary`

– Auto-delete: `true`

– Использование: временные уведомления

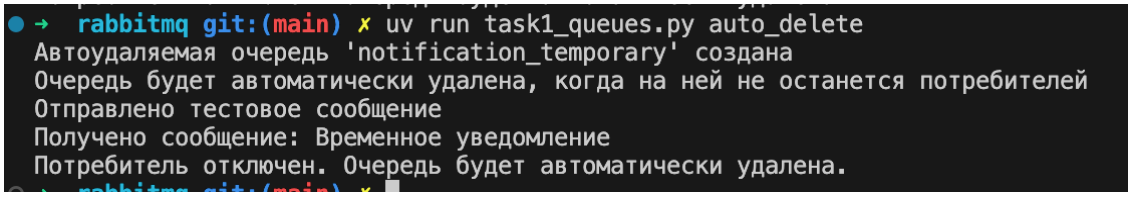

Рисунок 2 — Выполнение скрипта для создания автоудаляемой очереди

1.2.2 Задание 2: Реализация схемы взаимодействия сервисов с нагрузкой

Во втором задании были реализованы три схемы взаимодействия сервисов с использованием различных типов обменников RabbitMQ. Каждая схема имеет свой символ для обозначения времени сна и различные требования к сохранению сообщений.

1.2.2.1 Fanout Exchange

Тип обменника: fanout **Durable:** Да (сообщения сохраняются при выключении RabbitMQ) **Символ сна: # Время сна:** 2 секунды

Используется для широковещательной рассылки событий каталога игр всем подписанным сервисам.

Команды для создания:

```
rabbitmqadmin declare exchange name=game_catalog_events type=fanout
durable=true
rabbitmqadmin declare queue name=game_catalog_listener durable=true
rabbitmqadmin declare binding source=game_catalog_events
destination=game_catalog_listener
```

События:

– Игра добавлена в каталог

– Информация об игре обновлена

– Количество доступных к аренде игр обновлено

– Игра помечена как недоступная

– Фотографии игры загружены

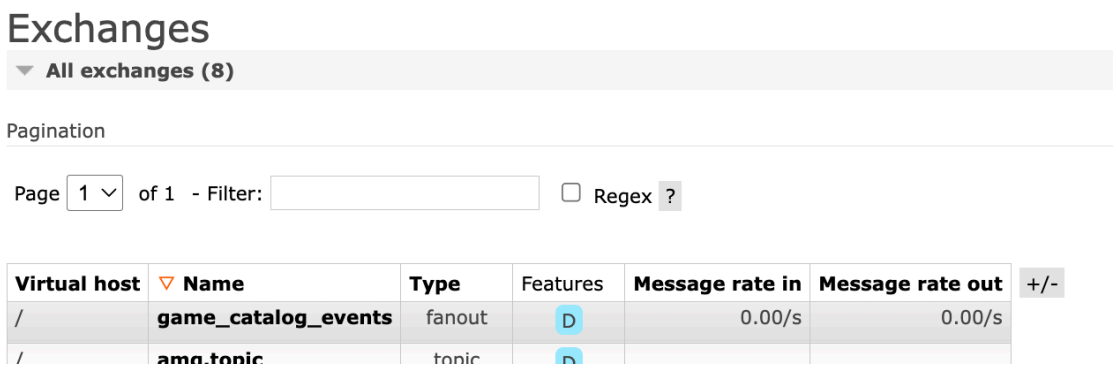

Рисунок 3 — Fanout exchange game_catalog_events в RabbitMQ Management UI

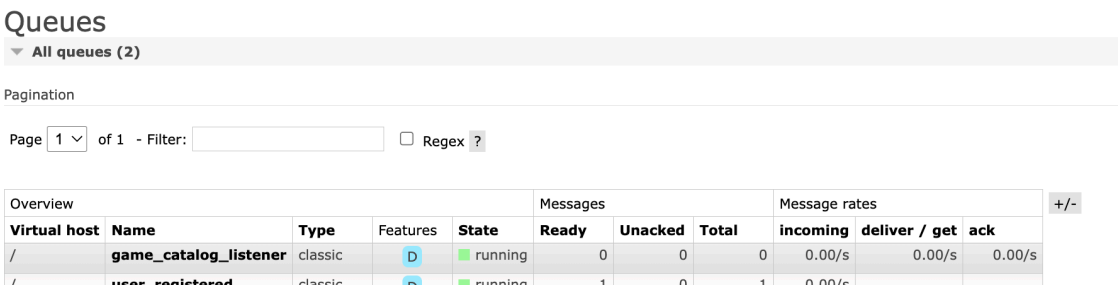

Рисунок 4 — Очередь game_catalog_listener, привязанная к fanout exchange

1.2.2.2 Direct Exchange

Тип обменника: direct **Durable:** Нет (сообщения могут не храниться при выключении RabbitMQ) **Символ сна:** * **Время сна:** 1 секунда

Используется для маршрутизации событий бронирования с использованием routing key.

Команды для создания:

```
rabbitmqadmin declare exchange name=booking_events type=direct
durable=false
rabbitmqadmin declare queue name=booking_processor durable=false
auto_delete=true
rabbitmqadmin declare binding source=booking_events
destination=booking_processor routing_key=game.booked
```

События:

– Игра забронирована

– Бронирование отменено

– Бронирование подтверждено

Routing key: `game.booked`

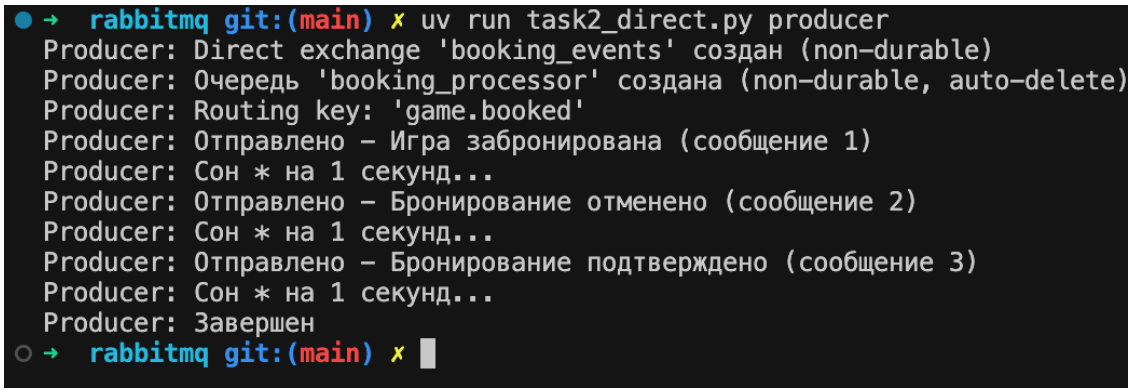

Рисунок 5 — Выполнение producer для direct exchange с символом сна *

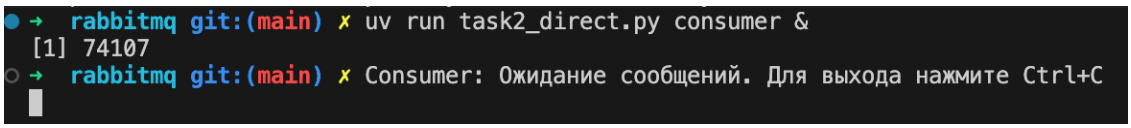

Рисунок 6 — Запуск consumer для direct exchange

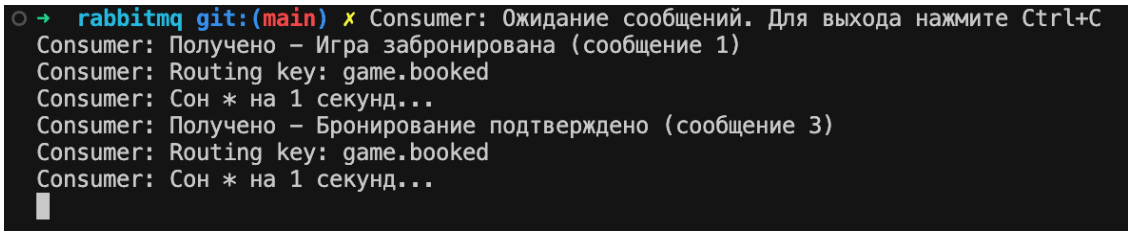

Рисунок 7 — Вывод consumer при получении сообщений из direct exchange

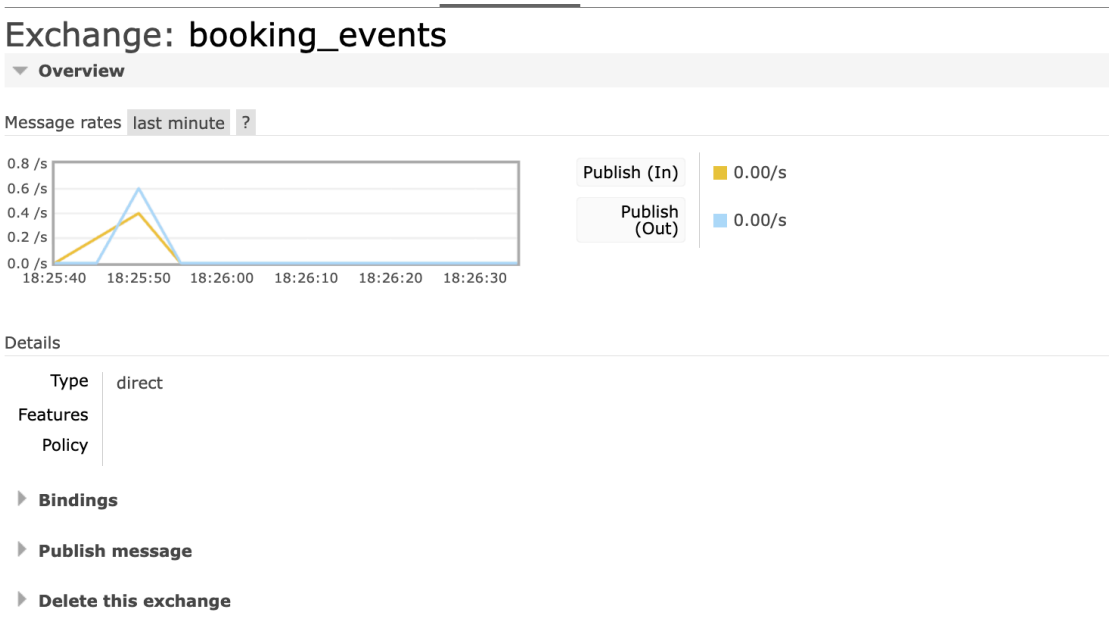

Рисунок 8 — Direct exchange booking_events в RabbitMQ Management UI

1.2.2.3 Topic Exchange

Тип обменника: topic **Durable:** Да (сообщения сохраняются при выключении RabbitMQ) **Символ сна:** - **Время сна:** 1.5 секунды

Используется для маршрутизации событий аренды с использованием паттернов routing key.

Команды для создания:

```
rabbitmqadmin declare exchange name=rent_events type=topic
durable=true
rabbitmqadmin declare queue name=rent_processor durable=true
```

```
rabbitmqadmin declare binding source=rent_events
destination=rent_processor routing_key=rent.order.created
```

События:

– Заказ создан (`rent.order.created`)

– Получение игры подтверждено (`rent.game.received`)

– Срок аренды продлён (`rent.period.extended`)

– Игра возвращена (`rent.game.returned`)

– Штраф начислен (`rent.penalty.charged`)

Routing key pattern: `rent.order.created`

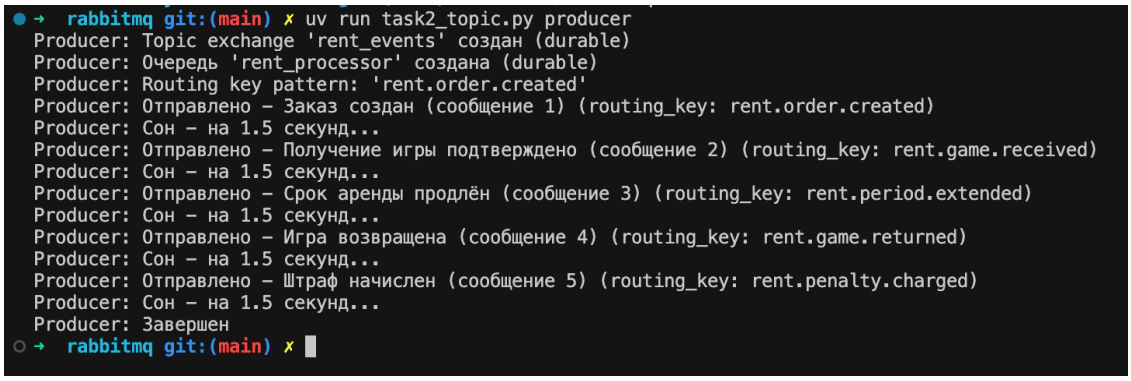

Рисунок 9 — Выполнение producer для topic exchange с символом сна -

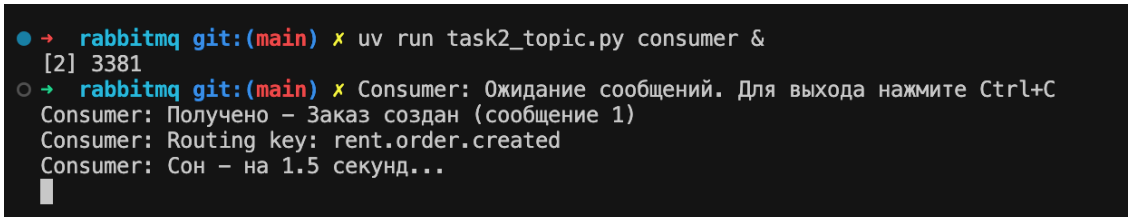

Рисунок 10 — Вывод consumer при получении сообщений из topic exchange

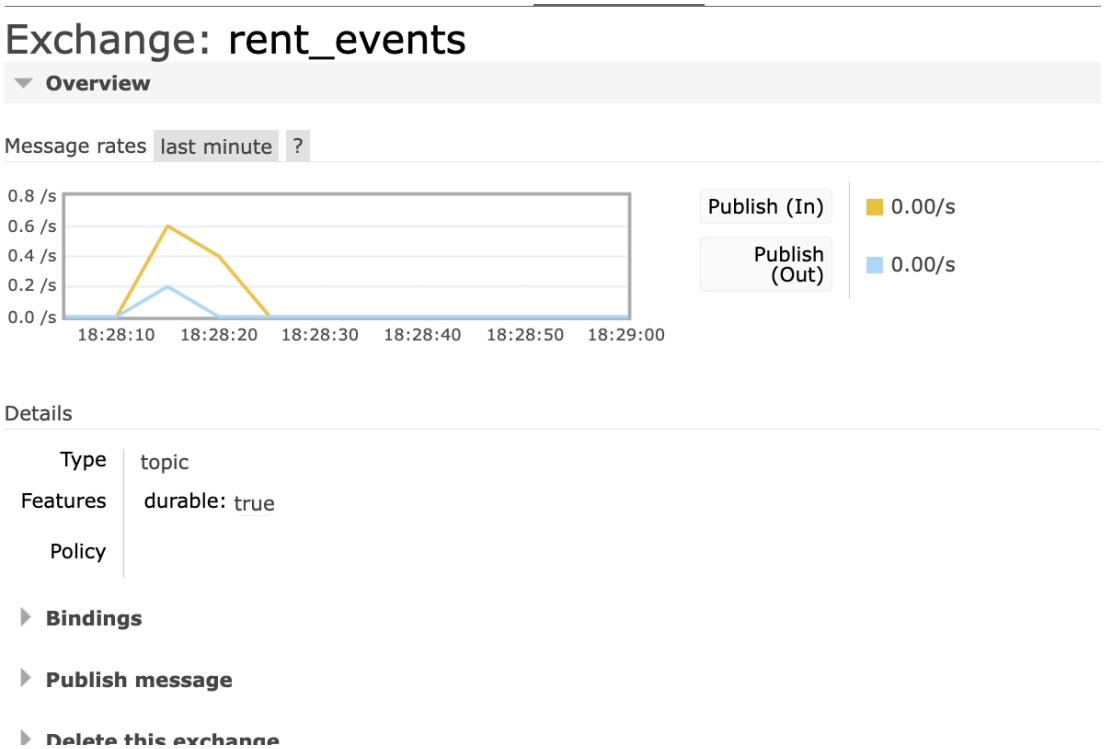

Рисунок 11 — Topic exchange rent_events в RabbitMQ Management UI

1.2.3 Реализация

Все реализации выполнены на языке Python с использованием асинхронной библиотеки `aio-pika` для работы с RabbitMQ. Код находится в директории `./code/rabbitmq`:

– `task1_queues.py` — реализация создания очередей из задания 1

– `task2_fanout.py` — реализация fanout exchange

– `task2_direct.py` — реализация direct exchange

– `task2_topic.py` — реализация topic exchange

Каждый скрипт поддерживает режимы `producer` и `consumer` для демонстрации работы с очередями и обменниками.

1.3 Вывод

В ходе выполнения практической работы были изучены различные типы очередей RabbitMQ (эксклюзивные, durable, автоудаляемые) и типы обменников (fanout, direct, topic). Были реализованы схемы взаимодействия сервисов с различными требованиями к сохранению сообщений и временными

задержками. Получены практические навыки работы с асинхронными библиотеками для работы с очередями сообщений.

2 Практическая работа №6

2.1 Цель работы

Проанализировать функции из персонального варианта, спроектировать API, построенное с помощью HTTP-запросов, и подробно описать API, указав назначение всех переменных.

2.2 Ход работы

2.2.1 Задание 1: Анализ функций из персонального варианта

На основе диаграммы из практической работы №4 были проанализированы все команды и доменные события для каждого агрегата системы аренды настольных игр. Система состоит из шести основных агрегатов:

- **Каталог игр** — управление информацией об играх, их доступностью и поиском;
- **Бронирование** — управление резервированием игр на определенные даты;
- **Оценка** — управление оценками и комментариями к играм;
- **Аккаунт пользователя** — управление регистрацией, авторизацией и профилями пользователей;
- **Аренда** — управление жизненным циклом аренды игр;
- **Оплата** — управление платежами и возвратами средств.

Для каждого агрегата были выделены команды (команды, которые изменяют состояние системы) и запросы (запросы, которые только читают данные).

2.2.2 Задание 2: Проектирование HTTP API

Был спроектирован RESTful API на основе FastAPI, который служит единой точкой входа (API Gateway) для всех микросервисов. API организован по агрегатам и использует стандартные HTTP методы:

- **POST** — для создания новых ресурсов;
- **GET** — для получения информации о ресурсах;

– **PUT** — для полного обновления ресурсов;
– **PATCH** — для частичного обновления ресурсов;
– **DELETE** — для удаления ресурсов (если требуется).

API структурирован по версионированию (`/api/v1/`) и группировке по доменным агрегатам.

2.2.2.1 Каталог игр (Game Catalog)

API для управления каталогом игр включает следующие endpoints:

– `POST /api/v1/games` — добавление новой игры в каталог;
– `PUT /api/v1/games/{game_id}` — обновление информации об игре;
– `POST /api/v1/games/{game_id}/photos` — загрузка фотографий игры;
– `PATCH /api/v1/games/{game_id}/availability` — обновление количества доступных игр;
– `POST /api/v1/games/{game_id}/unavailable` — пометка игры как недоступной;
– `POST /api/v1/games/sort` — сортировка списка игр;
– `POST /api/v1/games/search` — поиск игр по критериям;
– `GET /api/v1/games/{game_id}` — получение информации об игре.

2.2.2.2 Бронирование (Booking)

API для управления бронированиями:

– `POST /api/v1/bookings` — создание нового бронирования;
– `POST /api/v1/bookings/{booking_id}/cancel` — отмена бронирования;
– `POST /api/v1/bookings/{booking_id}/confirm` — подтверждение бронирования;
– `GET /api/v1/bookings/{booking_id}` — получение информации о бронировании.

2.2.2.3 Оценка (Rating)

API для управления оценками и комментариями:

– `POST /api/v1/ratings` — оставление оценки игре;
– `POST /api/v1/comments` — оставление комментария к игре;
– `PUT /api/v1/games/{game_id}/rating` — обновление рейтинга игры (системная команда).

2.2.2.4 Аккаунт пользователя (User Account)

API для управления пользователями:

– `POST /api/v1/users/register` — регистрация нового пользователя;
– `POST /api/v1/users/authorize` — авторизация пользователя;
– `POST /api/v1/users/{user_id}/block` — блокировка пользователя;
– `POST /api/v1/users/{user_id}/unblock` — разблокировка пользователя;
– `PUT /api/v1/users/{user_id}/profile` — обновление профиля пользователя;
– `GET /api/v1/users/{user_id}` — получение информации о пользователе.

2.2.2.5 Аренда (Rent)

API для управления арендой игр:

– `POST /api/v1/orders` — создание заказа на аренду;
– `POST /api/v1/orders/{order_id}/pickup-notification` — отправка уведомления о самовывозе;
– `POST /api/v1/orders/{order_id}/confirm-receipt` — подтверждение получения игры;
– `POST /api/v1/orders/{order_id}/return-reminder` — отправка напоминания о возврате;
– `POST /api/v1/orders/{order_id}/extend` — продление срока аренды;
– `POST /api/v1/orders/{order_id}/end` — завершение срока аренды (системная команда);
– `POST /api/v1/orders/{order_id}/return` — возврат игры;
– `POST /api/v1/orders/{order_id}/penalty` — начисление штрафа;

– `POST /api/v1/orders/{order_id}/confirm-return` — подтверждение возврата игры;
– `GET /api/v1/orders/{order_id}` — получение информации о заказе.

2.2.2.6 Оплата (Payment)

API для управления платежами:

– `POST /api/v1/payments` — инициирование платежа;
– `POST /api/v1/payments/{payment_id}/process` — обработка платежа;
– `POST /api/v1/refunds` — запрос на возврат средств;
– `POST /api/v1/refunds/{refund_id}/process` — обработка возврата средств;
– `POST /api/v1/refunds/{refund_id}/decline` — отклонение возврата средств;
– `GET /api/v1/payments/{payment_id}` — получение информации о платеже;
– `GET /api/v1/refunds/{refund_id}` — получение информации о возврате.

2.2.3 Задание 3: Тестирование API endpoints

Все endpoints были протестированы с использованием curl. Ниже приведены примеры запросов и ответов для каждого endpoint.

2.2.3.1 Endpoints каталога игр

POST /api/v1/games — добавление игры в каталог

Запрос:

```
curl -X POST http://localhost:8000/api/v1/games \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Каркассон",
    "description": "Стратегическая настольная игра",
    "min_players": 2,
    "max_players": 5,
    "play_time_minutes": 45,
    "age_rating": 7,
```

```
    "category": "Стратегия",
    "price_per_day": 150.0,
    "total_copies": 10
  }'
```

Ответ (201 Created):

```
{
  "game_id": "game-123",
  "name": "Каркассон",
  "status": "available",
  "available_count": 10,
  "total_copies": 10
}
```

GET /api/v1/games/{game_id} — получение информации об игре

Запрос:

```
curl -X GET http://localhost:8000/api/v1/games/game-123
```

Ответ (200 OK):

```
{
  "game_id": "game-123",
  "name": "Каркассон",
  "description": "Стратегическая настольная игра",
  "status": "available",
  "available_count": 8,
  "total_copies": 10,
  "photo_urls": ["https://example.com/photo1.jpg"],
  "rating": 4.5,
  "created_at": "2024-01-15T10:00:00",
  "updated_at": "2024-01-15T10:00:00"
}
```

PUT /api/v1/games/{game_id} — обновление информации об игре

Запрос:

```
curl -X PUT http://localhost:8000/api/v1/games/game-123 \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Каркассон (обновленное издание)",
    "price_per_day": 175.0
  }'
```

Ответ (200 OK):

```
{
  "game_id": "game-123",
  "name": "Каркассон (обновленное издание)",
  "description": "Стратегическая настольная игра",
  "status": "available",
  "available_count": 8,
  "total_copies": 10,
  "price_per_day": 175.0,
  "updated_at": "2024-01-20T11:00:00"
}
```

POST /api/v1/games/{game_id}/photos — загрузка фотографий

Запрос:

```
curl -X POST http://localhost:8000/api/v1/games/game-123/photos \
  -H "Content-Type: application/json" \
  -d '{
    "photo_urls": [
      "https://example.com/photo1.jpg",
      "https://example.com/photo2.jpg"
    ]
  }'
```

Ответ (200 OK):

```
{
  "success": true,
  "message": "Фотографии игры загружены"
}
```

PATCH /api/v1/games/{game_id}/availability — обновление доступности

Запрос:

```
curl -X PATCH http://localhost:8000/api/v1/games/game-123/
availability \
  -H "Content-Type: application/json" \
  -d '{"available_count": 5}'
```

Ответ (200 OK):

```
{
  "game_id": "game-123",
```

```
  "name": "Каркассон",
  "available_count": 5,
  "total_copies": 10,
  "status": "available",
  "updated_at": "2024-01-20T11:30:00"
}
```

POST /api/v1/games/{game_id}/unavailable — пометка как недоступной

Запрос:

```
curl -X POST http://localhost:8000/api/v1/games/game-123/unavailable
```

Ответ (200 OK):

```
{
  "success": true,
  "message": "Игра помечена как недоступная"
}
```

POST /api/v1/games/sort — сортировка игр

Запрос:

```
curl -X POST http://localhost:8000/api/v1/games/sort \
  -H "Content-Type: application/json" \
  -d '{
    "sort_field": "rating",
    "sort_order": "desc",
    "limit": 20,
    "offset": 0
  }'
```

Ответ (200 OK):

```
[
  {
    "game_id": "game-123",
    "name": "Каркассон",
    "rating": 4.8,
    "status": "available"
  },
  {
    "game_id": "game-456",
    "name": "Монополия",
    "rating": 4.5,
    "status": "available"
```

```
  }
]
```

POST /api/v1/games/search — поиск игр

Запрос:

```
curl -X POST http://localhost:8000/api/v1/games/search \
  -H "Content-Type: application/json" \
  -d '{
    "query": "стратегия",
    "category": "Стратегия",
    "min_players": 2,
    "max_players": 4,
    "max_price": 200.0
  }'
```

Ответ (200 OK):

```
[
  {
    "game_id": "game-123",
    "name": "Каркассон",
    "category": "Стратегия",
    "min_players": 2,
    "max_players": 5,
    "price_per_day": 150.0,
    "status": "available"
  }
]
```

2.2.3.2 Endpoints бронирования

POST /api/v1/bookings — создание бронирования

Запрос:

```
curl -X POST http://localhost:8000/api/v1/bookings \
  -H "Content-Type: application/json" \
  -d '{
    "game_id": "game-123",
    "user_id": "user-456",
    "booking_date": "2024-01-20T10:00:00",
    "pickup_date": "2024-01-21T10:00:00"
  }'
```

Ответ (201 Created):

```
{
  "booking_id": "booking-789",
  "game_id": "game-123",
  "user_id": "user-456",
  "status": "pending",
  "booking_date": "2024-01-20T10:00:00",
  "pickup_date": "2024-01-21T10:00:00",
  "created_at": "2024-01-20T10:00:00"
}
```

POST /api/v1/bookings/{booking_id}/confirm — подтверждение бронирования

Запрос:

```
curl -X POST http://localhost:8000/api/v1/bookings/booking-789/
confirm \
  -H "Content-Type: application/json" \
  -d '{
    "booking_id": "booking-789",
    "user_id": "user-456"
  }'
```

Ответ (200 OK):

```
{
  "booking_id": "booking-789",
  "game_id": "game-123",
  "user_id": "user-456",
  "status": "confirmed",
  "booking_date": "2024-01-20T10:00:00",
  "pickup_date": "2024-01-21T10:00:00",
  "created_at": "2024-01-20T10:00:00"
}
```

POST /api/v1/bookings/{booking_id}/cancel — отмена бронирования

Запрос:

```
curl -X POST http://localhost:8000/api/v1/bookings/booking-789/cancel
\
  -H "Content-Type: application/json" \
  -d '{
    "booking_id": "booking-789",
    "user_id": "user-456",
```

```
    "reason": "Изменение планов"
  }'
```

Ответ (200 OK):

```
{
  "success": true,
  "message": "Бронирование отменено"
}
```

GET /api/v1/bookings/{booking_id} — получение информации о бронировании

Запрос:

```
curl -X GET http://localhost:8000/api/v1/bookings/booking-789
```

Ответ (200 OK):

```
{
  "booking_id": "booking-789",
  "game_id": "game-123",
  "user_id": "user-456",
  "status": "confirmed",
  "booking_date": "2024-01-20T10:00:00",
  "pickup_date": "2024-01-21T10:00:00",
  "created_at": "2024-01-20T10:00:00"
}
```

2.2.3.3 Endpoints оценки

POST /api/v1/ratings — оставление оценки

Запрос:

```
curl -X POST http://localhost:8000/api/v1/ratings \
  -H "Content-Type: application/json" \
  -d '{
    "game_id": "game-123",
    "user_id": "user-456",
    "rating": 5
  }'
```

Ответ (201 Created):

```
{
  "rating_id": "rating-111",
  "game_id": "game-123",
  "user_id": "user-456",
  "rating": 5,
  "created_at": "2024-01-20T12:00:00"
}
```

POST /api/v1/comments — оставление комментария

Запрос:

```
curl -X POST http://localhost:8000/api/v1/comments \
  -H "Content-Type: application/json" \
  -d '{
    "game_id": "game-123",
    "user_id": "user-456",
    "comment_text": "Отличная игра! Очень рекомендую."
  }'
```

Ответ (201 Created):

```
{
  "comment_id": "comment-222",
  "game_id": "game-123",
  "user_id": "user-456",
  "comment_text": "Отличная игра! Очень рекомендую.",
  "is_moderated": true,
  "created_at": "2024-01-20T12:00:00"
}
```

PUT /api/v1/games/{game_id}/rating — обновление рейтинга (системная команда)

Запрос:

```
curl -X PUT http://localhost:8000/api/v1/games/game-123/rating \
  -H "Content-Type: application/json" \
  -d '{
    "game_id": "game-123",
    "new_rating": 4.7
  }'
```

Ответ (200 OK):

```
{
  "success": true,
```

```
  "message": "Рейтинг игры обновлён"
}
```

2.2.3.4 Endpoints аккаунта пользователя

POST /api/v1/users/register — регистрация пользователя

Запрос:

```
curl -X POST http://localhost:8000/api/v1/users/register \
  -H "Content-Type: application/json" \
  -d '{
    "email": "user@example.com",
    "password": "securepassword123",
    "first_name": "Иван",
    "last_name": "Иванов",
    "phone": "+79991234567"
  }'
```

Ответ (201 Created):

```
{
  "user_id": "user-456",
  "email": "user@example.com",
  "first_name": "Иван",
  "last_name": "Иванов",
  "phone": "+79991234567",
  "is_blocked": false,
  "created_at": "2024-01-20T10:00:00",
  "updated_at": "2024-01-20T10:00:00"
}
```

POST /api/v1/users/authorize — авторизация пользователя

Запрос:

```
curl -X POST http://localhost:8000/api/v1/users/authorize \
  -H "Content-Type: application/json" \
  -d '{
    "email": "user@example.com",
    "password": "securepassword123"
  }'
```

Ответ (200 OK):

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "token_type": "bearer",
  "user": {
    "user_id": "user-456",
    "email": "user@example.com",
    "first_name": "Иван",
    "last_name": "Иванов"
  }
}
```

PUT /api/v1/users/{user_id}/profile — обновление профиля

Запрос:

```
curl -X PUT http://localhost:8000/api/v1/users/user-456/profile \
  -H "Content-Type: application/json" \
  -d '{
    "user_id": "user-456",
    "first_name": "Иван",
    "last_name": "Петров",
    "phone": "+79991234568"
  }'
```

Ответ (200 OK):

```
{
  "user_id": "user-456",
  "email": "user@example.com",
  "first_name": "Иван",
  "last_name": "Петров",
  "phone": "+79991234568",
  "is_blocked": false,
  "updated_at": "2024-01-20T13:00:00"
}
```

POST /api/v1/users/{user_id}/block — блокировка пользователя

Запрос:

```
curl -X POST http://localhost:8000/api/v1/users/user-456/block \
  -H "Content-Type: application/json" \
  -d '{
    "user_id": "user-456",
    "reason": "Нарушение правил"
  }'
```

Ответ (200 OK):

```
{
  "success": true,
  "message": "Пользователь заблокирован"
}
```

POST /api/v1/users/{user_id}/unblock — разблокировка пользователя

Запрос:

```
curl -X POST http://localhost:8000/api/v1/users/user-456/unblock \
  -H "Content-Type: application/json" \
  -d '{"user_id": "user-456"}'
```

Ответ (200 OK):

```
{
  "success": true,
  "message": "Пользователь разблокирован"
}
```

GET /api/v1/users/{user_id} — получение информации о пользователе

Запрос:

```
curl -X GET http://localhost:8000/api/v1/users/user-456
```

Ответ (200 OK):

```
{
  "user_id": "user-456",
  "email": "user@example.com",
  "first_name": "Иван",
  "last_name": "Петров",
  "phone": "+79991234567",
  "is_blocked": false,
  "created_at": "2024-01-20T10:00:00",
  "updated_at": "2024-01-20T10:00:00"
}
```

2.2.3.5 Endpoints аренды

POST /api/v1/orders — создание заказа на аренду

Запрос:

```
curl -X POST http://localhost:8000/api/v1/orders \
  -H "Content-Type: application/json" \
  -d '{
    "booking_id": "booking-789",
    "user_id": "user-456",
    "pickup_location": "Москва, ул. Примерная, д. 1",
    "rental_days": 7
  }'
```

Ответ (201 Created):

```
{
  "order_id": "order-333",
  "booking_id": "booking-789",
  "game_id": "game-123",
  "user_id": "user-456",
  "status": "created",
  "pickup_date": "2024-01-21T10:00:00",
  "pickup_location": "Москва, ул. Примерная, д. 1",
  "rental_days": 7,
  "total_amount": 1050.0,
  "created_at": "2024-01-20T15:00:00"
}
```

POST /api/v1/orders/{order_id}/confirm-receipt — подтверждение получения

Запрос:

```
curl -X POST http://localhost:8000/api/v1/orders/order-333/confirm-
receipt \
  -H "Content-Type: application/json" \
  -d '{
    "order_id": "order-333",
    "user_id": "user-456"
  }'
```

Ответ (200 OK):

```
{
  "success": true,
  "message": "Получение игры подтверждено"
}
```

POST /api/v1/orders/{order_id}/extend — продление аренды

Запрос:

```
curl -X POST http://localhost:8000/api/v1/orders/order-333/extend \
  -H "Content-Type: application/json" \
  -d '{
    "order_id": "order-333",
    "user_id": "user-456",
    "additional_days": 3
  }'
```

Ответ (200 OK):

```
{
  "order_id": "order-333",
  "rental_days": 10,
  "return_date": "2024-01-31T10:00:00",
  "total_amount": 1500.0,
  "status": "extended"
}
```

POST /api/v1/orders/{order_id}/return — возврат игры

Запрос:

```
curl -X POST http://localhost:8000/api/v1/orders/order-333/return \
  -H "Content-Type: application/json" \
  -d '{
    "order_id": "order-333",
    "user_id": "user-456",
    "return_location": "Москва, ул. Примерная, д. 1"
  }'
```

Ответ (200 OK):

```
{
  "success": true,
  "message": "Игра возвращена"
}
```

POST /api/v1/orders/{order_id}/pickup-notification — отправка уведомления о самовывозе

Запрос:

```
curl -X POST http://localhost:8000/api/v1/orders/order-333/pickup-
notification \
  -H "Content-Type: application/json" \
  -d '{
    "order_id": "order-333",
```

```
    "pickup_date": "2024-01-21T10:00:00",
    "pickup_location": "Москва, ул. Примерная, д. 1"
  }'
```

Ответ (200 OK):

```
{
  "success": true,
  "message": "Уведомление о дате и месте самовывоза отправлено"
}
```

POST /api/v1/orders/{order_id}/return-reminder — отправка напоминания о возврате

Запрос:

```
curl -X POST http://localhost:8000/api/v1/orders/order-333/return-
reminder \
  -H "Content-Type: application/json" \
  -d '{
    "order_id": "order-333",
    "return_date": "2024-01-28T10:00:00"
  }'
```

Ответ (200 OK):

```
{
  "success": true,
  "message": "Напоминание о возврате отправлено"
}
```

POST /api/v1/orders/{order_id}/end — завершение срока аренды (системная команда)

Запрос:

```
curl -X POST http://localhost:8000/api/v1/orders/order-333/end \
  -H "Content-Type: application/json" \
  -d '{"order_id": "order-333"}'
```

Ответ (200 OK):

```
{
  "success": true,
  "message": "Срок аренды завершён"
}
```

POST /api/v1/orders/{order_id}/penalty — начисление штрафа

Запрос:

```
curl -X POST http://localhost:8000/api/v1/orders/order-333/penalty \
  -H "Content-Type: application/json" \
  -d '{
    "order_id": "order-333",
    "penalty_amount": 200.0,
    "reason": "Просрочка возврата на 2 дня"
  }'
```

Ответ (200 OK):

```
{
  "success": true,
  "message": "Штраф начислен"
}
```

POST /api/v1/orders/{order_id}/confirm-return — подтверждение возврата игры

Запрос:

```
curl -X POST http://localhost:8000/api/v1/orders/order-333/confirm-
return \
  -H "Content-Type: application/json" \
  -d '{
    "order_id": "order-333",
    "game_condition": "Отличное состояние"
  }'
```

Ответ (200 OK):

```
{
  "success": true,
  "message": "Возврат игры подтвержден"
}
```

GET /api/v1/orders/{order_id} — получение информации о заказе

Запрос:

```
curl -X GET http://localhost:8000/api/v1/orders/order-333
```

Ответ (200 OK):

```
{
  "order_id": "order-333",
  "booking_id": "booking-789",
  "game_id": "game-123",
  "user_id": "user-456",
  "status": "completed",
  "pickup_date": "2024-01-21T10:00:00",
  "pickup_location": "Москва, ул. Примерная, д. 1",
  "return_date": "2024-01-28T10:00:00",
  "rental_days": 7,
  "total_amount": 1050.0,
  "penalty_amount": null,
  "created_at": "2024-01-20T15:00:00"
}
```

2.2.3.6 Endpoints оплаты

POST /api/v1/payments — инициирование платежа

Запрос:

```
curl -X POST http://localhost:8000/api/v1/payments \
  -H "Content-Type: application/json" \
  -d '{
    "order_id": "order-333",
    "user_id": "user-456",
    "amount": 1050.0,
    "payment_method": "card"
  }'
```

Ответ (201 Created):

```
{
  "payment_id": "payment-444",
  "order_id": "order-333",
  "user_id": "user-456",
  "amount": 1050.0,
  "status": "initiated",
  "payment_method": "card",
  "created_at": "2024-01-20T16:00:00"
}
```

POST /api/v1/payments/{payment_id}/process — обработка платежа

Запрос:

```
curl -X POST http://localhost:8000/api/v1/payments/payment-444/
process \
  -H "Content-Type: application/json" \
  -d '{
    "payment_id": "payment-444",
    "transaction_id": "txn-123456789"
  }'
```

Ответ (200 OK):

```
{
  "payment_id": "payment-444",
  "order_id": "order-333",
  "user_id": "user-456",
  "amount": 1050.0,
  "status": "completed",
  "payment_method": "card",
  "transaction_id": "txn-123456789",
  "completed_at": "2024-01-20T16:05:00"
}
```

POST /api/v1/refunds — запрос на возврат средств

Запрос:

```
curl -X POST http://localhost:8000/api/v1/refunds \
  -H "Content-Type: application/json" \
  -d '{
    "payment_id": "payment-444",
    "user_id": "user-456",
    "reason": "Отмена заказа",
    "amount": 1050.0
  }'
```

Ответ (201 Created):

```
{
  "refund_id": "refund-555",
  "payment_id": "payment-444",
  "user_id": "user-456",
  "amount": 1050.0,
  "status": "requested",
  "reason": "Отмена заказа",
  "created_at": "2024-01-21T10:00:00"
}
```

POST /api/v1/refunds/{refund_id}/process — обработка возврата

Запрос:

```
curl -X POST http://localhost:8000/api/v1/refunds/refund-555/process
\
  -H "Content-Type: application/json" \
  -d '{
    "refund_id": "refund-555",
    "transaction_id": "refund-txn-987654321"
  }'
```

Ответ (200 OK):

```
{
  "refund_id": "refund-555",
  "payment_id": "payment-444",
  "user_id": "user-456",
  "amount": 1050.0,
  "status": "completed",
  "reason": "Отмена заказа",
  "transaction_id": "refund-txn-987654321",
  "completed_at": "2024-01-21T10:30:00"
}
```

POST /api/v1/refunds/{refund_id}/decline — отклонение возврата средств

Запрос:

```
curl -X POST http://localhost:8000/api/v1/refunds/refund-555/decline
\
  -H "Content-Type: application/json" \
  -d '{
    "refund_id": "refund-555",
    "reason": "Игра была возвращена с повреждениями"
  }'
```

Ответ (200 OK):

```
{
  "success": true,
  "message": "Возврат средств отклонен"
}
```

GET /api/v1/payments/{payment_id} — получение информации о платеже

Запрос:

```
curl -X GET http://localhost:8000/api/v1/payments/payment-444
```

Ответ (200 OK):

```
{
  "payment_id": "payment-444",
  "order_id": "order-333",
  "user_id": "user-456",
  "amount": 1050.0,
  "status": "completed",
  "payment_method": "card",
  "transaction_id": "txn-123456789",
  "created_at": "2024-01-20T16:00:00",
  "completed_at": "2024-01-20T16:05:00"
}
```

GET /api/v1/refunds/{refund_id} — получение информации о возврате

Запрос:

```
curl -X GET http://localhost:8000/api/v1/refunds/refund-555
```

Ответ (200 OK):

```
{
  "refund_id": "refund-555",
  "payment_id": "payment-444",
  "user_id": "user-456",
  "amount": 1050.0,
  "status": "completed",
  "reason": "Отмена заказа",
  "transaction_id": "refund-txn-987654321",
  "created_at": "2024-01-21T10:00:00",
  "completed_at": "2024-01-21T10:30:00"
}
```

2.2.3.7 Служебные endpoints

GET / — корневой endpoint

Запрос:

```
curl -X GET http://localhost:8000/
```

Ответ (200 OK):

```
{
  "service": "Game Rental System API Gateway",
  "version": "1.0.0",
  "status": "running",
  "docs": "/docs",
  "openapi": "/openapi.json"
}
```

GET /health — проверка здоровья сервиса

Запрос:

```
curl -X GET http://localhost:8000/health
```

Ответ (200 OK):

```
{
  "status": "healthy",
  "service": "gateway"
}
```

2.2.4 Задание 4: Описание API и назначение переменных

Все модели данных определены с использованием Pydantic и содержат подробные описания всех полей. Ниже приведены основные модели для каждого агрегата.

2.2.4.1 Модели каталога игр

AddGameRequest — запрос на добавление игры:

- `name: str` — название игры (обязательное, 1-200 символов);
- `description: Optional[str]` — описание игры (до 2000 символов);
- `min_players: int` — минимальное количество игроков (1-20);
- `max_players: int` — максимальное количество игроков (1-50);
- `play_time_minutes: Optional[int]` — продолжительность игры в минутах;
- `age_rating: Optional[int]` — возрастное ограничение (0-18);
- `category: Optional[str]` — категория игры (до 100 символов);
- `price_per_day: float` — стоимость аренды за день (≥ 0);
- `total_copies: int` — общее количество экземпляров игры (≥ 1).

GameResponse — ответ с информацией об игре:

- `game_id: str` — уникальный идентификатор игры;
- `name: str` — название игры;
- `description: Optional[str]` — описание игры;
- `status: GameStatus` — статус доступности (available, unavailable, reserved, rented, inspection, repair);
- `available_count: int` — количество доступных экземпляров;
- `total_copies: int` — общее количество экземпляров;
- `photo_urls: List[str]` — список URL фотографий;
- `rating: Optional[float]` — средний рейтинг игры (0-5);
- `created_at: datetime` — дата создания записи;
- `updated_at: datetime` — дата последнего обновления.

2.2.4.2 Модели бронирования

BookGameRequest — запрос на бронирование:

- `game_id: str` — идентификатор игры для бронирования;
- `user_id: str` — идентификатор пользователя;
- `booking_date: datetime` — дата бронирования;
- `pickup_date: datetime` — желаемая дата самовывоза.

BookingResponse — ответ с информацией о бронировании:

- `booking_id: str` — уникальный идентификатор бронирования;
- `game_id: str` — идентификатор игры;
- `user_id: str` — идентификатор пользователя;
- `status: str` — статус бронирования (pending, confirmed, canceled);
- `booking_date: datetime` — дата бронирования;
- `pickup_date: datetime` — дата самовывоза;
- `created_at: datetime` — дата создания бронирования.

2.2.4.3 Модели оценки

LeaveRatingRequest — запрос на оставление оценки:

– `game_id: str` — идентификатор игры;
– `user_id: str` — идентификатор пользователя;
– `rating: int` — оценка от 1 до 5.

LeaveCommentRequest — запрос на оставление комментария:

– `game_id: str` — идентификатор игры;
– `user_id: str` — идентификатор пользователя;
– `comment_text: str` — текст комментария (1-2000 символов).

2.2.4.4 Модели аккаунта пользователя

RegisterUserRequest — запрос на регистрацию:

– `email: EmailStr` — email адрес пользователя;
– `password: str` — пароль пользователя (8-100 символов);
– `first_name: str` — имя пользователя (1-100 символов);
– `last_name: str` — фамилия пользователя (1-100 символов);
– `phone: Optional[str]` — номер телефона (до 20 символов).

AuthorizeUserRequest — запрос на авторизацию:

– `email: EmailStr` — email адрес пользователя;
– `password: str` — пароль пользователя.

AuthResponse — ответ с результатом авторизации:

– `access_token: str` — JWT токен доступа;
– `token_type: str` — тип токена (обычно «bearer»);
– `user: UserResponse` — информация о пользователе.

2.2.4.5 Модели аренды

CreateOrderRequest — запрос на создание заказа:

– `booking_id: str` — идентификатор подтвержденного бронирования;
– `user_id: str` — идентификатор пользователя;
– `pickup_location: str` — адрес места самовывоза (до 200 символов);
– `rental_days: int` — количество дней аренды (1-30).

OrderResponse — ответ с информацией о заказе:

- `order_id: str` — уникальный идентификатор заказа;
- `booking_id: str` — идентификатор бронирования;
- `game_id: str` — идентификатор игры;
- `user_id: str` — идентификатор пользователя;
- `status: str` — статус заказа;
- `pickup_date: datetime` — дата самовывоза;
- `pickup_location: str` — адрес места самовывоза;
- `return_date: Optional[datetime]` — дата возврата;
- `rental_days: int` — количество дней аренды;
- `total_amount: float` — общая сумма заказа;
- `penalty_amount: Optional[float]` — сумма штрафа (если есть);
- `created_at: datetime` — дата создания заказа.

2.2.4.6 Модели оплаты

InitiatePaymentRequest — запрос на инициирование платежа:

- `order_id: str` — идентификатор заказа;
- `user_id: str` — идентификатор пользователя;
- `amount: float` — сумма платежа (≥ 0);
- `payment_method: str` — способ оплаты (до 50 символов).

PaymentResponse — ответ с информацией о платеже:

- `payment_id: str` — уникальный идентификатор платежа;
- `order_id: str` — идентификатор заказа;
- `user_id: str` — идентификатор пользователя;
- `amount: float` — сумма платежа;
- `status: str` — статус платежа (initiated, processing, completed, declined, refunded);
- `payment_method: str` — способ оплаты;
- `transaction_id: Optional[str]` — идентификатор транзакции;
- `created_at: datetime` — дата создания платежа;

– `completed_at: Optional[datetime]` — дата завершения платежа.

RequestRefundRequest — запрос на возврат средств:

– `payment_id: str` — идентификатор платежа;
– `user_id: str` — идентификатор пользователя;
– `reason: str` — причина возврата (до 500 символов);
– `amount: Optional[float]` — сумма возврата (если частичный возврат, ≥ 0).

2.2.5 Реализация

API Gateway реализован на FastAPI и находится в директории `./code/services/gateway`. Структура проекта:

– `app.py` — основное приложение FastAPI со всеми endpoints;
– `models.py` — Pydantic модели для всех запросов и ответов;
– `main.py` — точка входа для запуска сервиса;
– `pyproject.toml` — конфигурация проекта с зависимостями.

Все микросервисы организованы в отдельные директории `./code/services/<service_name>`, каждый с собственным `pyproject.toml`, что позволяет им быть независимыми проектами в рамках UV workspace.

Скрипты для работы с RabbitMQ перемещены в отдельную директорию `./code/rabbitmq`.

2.2.6 OpenAPI документация

FastAPI автоматически генерирует OpenAPI спецификацию, доступную по следующим адресам:

– Swagger UI: `http://localhost:8000/docs` — интерактивная документация API;
– ReDoc: `http://localhost:8000/redoc` — альтернативная документация;
– OpenAPI JSON: `http://localhost:8000/openapi.json` — JSON спецификация для экспорта.

Для экспорта OpenAPI спецификации можно использовать команду:

```
curl http://localhost:8000/openapi.json > openapi.json
```

2.3 Вывод

В ходе выполнения практической работы были проанализированы функции всех агрегатов системы аренды настольных игр, спроектирован RESTful API на основе HTTP-запросов с использованием FastAPI, и подробно описаны все модели данных с указанием назначения каждой переменной. API Gateway служит единой точкой входа для всех микросервисов и предоставляет полную OpenAPI документацию для внешних клиентов.

3 Практическая работа №7

3.1 Цель работы

Реализация микросервисной системы аренды настольных игр с использованием Docker, Docker Compose, gRPC и RabbitMQ. Получение практических навыков контейнеризации микросервисов и настройки межсервисного взаимодействия.

3.2 Ход работы

3.2.1 Задание 1: Реализация микросервисов

В рамках практической работы была реализована полнофункциональная микросервисная система, состоящая из шести основных сервисов и API Gateway. Каждый микросервис реализует функциональность своего агрегата и имеет от 2 до 5 эндпоинтов.

3.2.1.1 Сервис аккаунта пользователя (User Account)

Порт: 8001 **База данных:** PostgreSQL **Количество эндпоинтов:** 5

Реализованные эндпоинты:

– `POST /api/v1/users/register` — регистрация нового пользователя;
– `POST /api/v1/users/authorize` — авторизация пользователя с выдачей JWT токена;
– `GET /api/v1/users/{user_id}` — получение информации о пользователе;
– `PUT /api/v1/users/{user_id}/profile` — обновление профиля пользователя;
– `POST /api/v1/users/{user_id}/block` — блокировка пользователя.

Особенности реализации:

– Использование SQLAlchemy для работы с базой данных PostgreSQL
– Хеширование паролей с помощью bcrypt
– Генерация JWT токенов для аутентификации
– Публикация доменных событий в RabbitMQ

3.2.1.2 Сервис каталога игр (Game Catalog)

Порт: 8002 **База данных:** PostgreSQL **Количество эндпоинтов:** 5

Реализованные эндпоинты:

- `POST /api/v1/games` — добавление новой игры в каталог;
- `GET /api/v1/games/{game_id}` — получение информации об игре;
- `PUT /api/v1/games/{game_id}` — обновление информации об игре;
- `POST /api/v1/games/search` — поиск игр по различным критериям;
- `PATCH /api/v1/games/{game_id}/availability` — обновление количества доступных игр.

Особенности реализации:

- Хранение информации об играх в PostgreSQL
- Поддержка поиска по названию, категории, количеству игроков и цене
- Отслеживание доступности игр

3.2.1.3 Сервис бронирования (Booking)

Порт: 8003 **Количество эндпоинтов:** 3

Реализованные эндпоинты:

- `POST /api/v1/bookings` — создание бронирования игры;
- `POST /api/v1/bookings/{booking_id}/cancel` — отмена бронирования;
- `GET /api/v1/bookings/{booking_id}` — получение информации о бронировании.

Особенности реализации:

- Вложенный вызов функции: валидация пользователя через HTTP-запрос к сервису User Account
- Публикация доменных событий в RabbitMQ при создании и отмене бронирования
- Хранение данных в памяти (для демонстрации, в production используется БД)

3.2.1.4 Сервис оплаты (Payment)

Порт: 8004 (HTTP), 50051 (gRPC) **Количество эндпоинтов:** 6

Реализованные эндпоинты:

- `POST /api/v1/payments` — инициирование платежа;
- `POST /api/v1/payments/{payment_id}/process` — обработка платежа;
- `POST /api/v1/refunds` — запрос на возврат средств;
- `POST /api/v1/refunds/{refund_id}/process` — обработка возврата;
- `POST /api/v1/refunds/{refund_id}/decline` — отклонение возврата;
- `GET /api/v1/payments/{payment_id}` — получение информации о платеже.

Особенности реализации:

- gRPC сервер для синхронного взаимодействия с сервисом Rent
- Мокированная платежная система (Эквайринг) с случайным успехом/отказом
- Мокированная система ОФД (фискальный оператор) для генерации чеков
- Публикация доменных событий в RabbitMQ

3.2.1.5 Сервис аренды (Rent)

Порт: 8005 **База данных:** PostgreSQL **Количество эндпоинтов:** 6

Реализованные эндпоинты:

- `POST /api/v1/orders` — создание заказа на аренду;
- `POST /api/v1/orders/{order_id}/pickup-notification` — отправка уведомления о самовывозе;
- `POST /api/v1/orders/{order_id}/confirm-receipt` — подтверждение получения игры;
- `POST /api/v1/orders/{order_id}/return` — возврат игры;
- `POST /api/v1/orders/{order_id}/penalty` — начисление штрафа;
- `GET /api/v1/orders/{order_id}` — получение информации о заказе.

Особенности реализации:

– gRPC клиент для синхронного вызова сервиса Payment при создании заказа
– Мокированные сервисы уведомлений: OneSignal (push-уведомления) и SendGrid (email)
– Хранение заказов в PostgreSQL
– Публикация доменных событий в RabbitMQ

3.2.1.6 Сервис оценки (Rating)

Порт: 8006 **Количество эндпоинтов:** 4

Реализованные эндпоинты:

– `POST /api/v1/ratings` — оставление оценки игре;
– `POST /api/v1/comments` — оставление комментария с модерацией;
– `PUT /api/v1/games/{game_id}/rating` — обновление рейтинга игры (системная команда);
– `GET /api/v1/ratings/{rating_id}` — получение информации об оценке.

Особенности реализации:

– Мокированный API Perspective для модерации комментариев
– Автоматическое обновление рейтинга игры в каталоге
– Публикация доменных событий в RabbitMQ

3.2.2 Задание 2: Контейнеризация с Docker

Каждый микросервис упакован в отдельный Docker-контейнер. Для каждого сервиса создан Dockerfile на основе образа `python:3.12-slim`.

Пример Dockerfile для сервиса User Account:

```
FROM python:3.12-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    gcc \
    postgresql-client \
```

```
    curl \
    && rm -rf /var/lib/apt/lists/*

# Copy dependency files
COPY pyproject.toml ./

# Install Python dependencies
RUN pip install --no-cache-dir -e .

# Copy application code
COPY . .

# Expose port
EXPOSE 8001

# Run the application
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8001"]
```

Особенности Dockerfile:

- Минимальный базовый образ для уменьшения размера
- Установка только необходимых системных зависимостей
- Копирование зависимостей перед кодом для кэширования слоев
- Открытие соответствующего порта для каждого сервиса

3.2.3 Задание 3: Оркестрация с Docker Compose

Все сервисы, базы данных и инфраструктурные компоненты оркестрируются с помощью Docker Compose. Файл `docker-compose.yml` включает:

3.2.3.1 Базы данных

- **user-account-db** — PostgreSQL для сервиса аккаунта пользователя (порт 5433);
- **game-catalog-db** — PostgreSQL для сервиса каталога игр (порт 5434);
- **rent-db** — PostgreSQL для сервиса аренды (порт 5435).

Все базы данных используют образ `postgres:15-alpine` и настроены с health checks для обеспечения готовности перед запуском зависимых сервисов.

3.2.3.2 Инфраструктурные сервисы

– **rabbitmq** — брокер сообщений RabbitMQ с веб-интерфейсом управления (порты 5672, 15672);

RabbitMQ используется для асинхронной публикации доменных событий между сервисами.

3.2.3.3 Микросервисы

Все шесть микросервисов и API Gateway настроены с:

– Зависимостями от соответствующих баз данных и RabbitMQ;

– Health checks для мониторинга состояния;

– Переменными окружения для конфигурации;

– Правильной последовательностью запуска через `depends_on`.

Пример конфигурации сервиса:

```
user-account:
  build:
    context: ./code/services/user-account
    dockerfile: Dockerfile
  container_name: user-account-service
  ports:
    - "8001:8001"
  environment:
    DATABASE_URL: postgresql://postgres:postgres@user-account-db:
5432/user_account
    SECRET_KEY: your-secret-key-change-in-production
    RABBITMQ_URL: amqp://guest:guest@rabbitmq:5672/
  depends_on:
    user-account-db:
      condition: service_healthy
    rabbitmq:
      condition: service_healthy
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:8001/health"]
    interval: 30s
    timeout: 10s
    retries: 3
```

3.2.4 Задание 4: Реализация gRPC коммуникации

Для синхронного взаимодействия между сервисами Rent и Payment реализована gRPC коммуникация согласно диаграмме из практической работы №4.

3.2.4.1 gRPC сервер (Payment Service)

В сервисе Payment реализован gRPC сервер на порту 50051. Определен proto-файл `payment.proto`:

```
syntax = "proto3";

package payment;

service PaymentService {
  rpc InitiatePayment (InitiatePaymentRequest) returns
(InitiatePaymentResponse);
  rpc ProcessPayment (ProcessPaymentRequest) returns
(ProcessPaymentResponse);
}

message InitiatePaymentRequest {
  string order_id = 1;
  string user_id = 2;
  double amount = 3;
  string payment_method = 4;
}

message InitiatePaymentResponse {
  string payment_id = 1;
  string status = 2;
  string message = 3;
}
```

3.2.4.2 gRPC клиент (Rent Service)

Сервис Rent использует gRPC клиент для вызова метода `InitiatePayment` при создании заказа:

```
async def initiate_payment_grpc(
    order_id: str,
    user_id: str,
    amount: float,
```

```
    payment_method: str = "card"
) -> Optional[Dict]:
    """Call Payment service via gRPC to initiate payment."""
    channel = grpc.aio.insecure_channel(PAYMENT_SERVICE_GRPC_URL)
    # ... gRPC call implementation
```

Это обеспечивает синхронное взаимодействие между сервисами, как указано в архитектурной диаграмме.

3.2.5 Задание 5: Интеграция RabbitMQ

Все сервисы интегрированы с RabbitMQ для публикации доменных событий. Используется Topic Exchange для маршрутизации событий по routing keys, что позволяет гибко настраивать подписки на различные типы событий.

3.2.5.1 Архитектура событий

Каждый сервис публикует доменные события в свой собственный exchange с типом Topic:

- **payment_events** — события от сервиса Payment;
- **booking_events** — события от сервиса Booking;
- **rent_events** — события от сервиса Rent;
- **rating_events** — события от сервиса Rating.

3.2.5.2 Реализация публикации событий

Пример реализации функции публикации события:

```
async def publish_event(event_type: str, event_data: Dict[str, Any]):
    """Publish a domain event to RabbitMQ."""
    try:
        connection = await aio_pika.connect_robust(RABBITMQ_URL)
        channel = await connection.channel()

        # Declare exchange
        exchange = await channel.declare_exchange(
            "payment_events",
            aio_pika.ExchangeType.TOPIC
        )

        # Publish event
```

```
        message_body = json.dumps(event_data).encode()
        await exchange.publish(
            aio_pika.Message(
                message_body,
                content_type="application/json",
                headers={"event_type": event_type}
            ),
            routing_key=event_type
        )

        await connection.close()
        print(f"Published event: {event_type}")
    except Exception as e:
        print(f"Error publishing event: {e}")
```

3.2.5.3 Публикуемые доменные события

Сервис Payment:

- `payment.initiated` — платеж инициирован
- `payment.successful` — платеж успешно проведен
- `payment.declined` — платеж отклонен
- `refund.requested` — запрос на возврат средств создан
- `refund.processed` — возврат средств выполнен
- `refund.declined` — возврат средств отклонен

Сервис Booking:

- `booking.created` — бронирование создано
- `booking.canceled` — бронирование отменено

Сервис Rent:

- `rent.order.created` — заказ создан
- `rent.pickup_notification.sent` — уведомление о самовывозе отправлено
- `rent.game_receipt.confirmed` — получение игры подтверждено
- `rent.game.returned` — игра возвращена
- `rent.penalty.charged` — штраф начислен

Сервис Rating:

- `rating.left` — оценка оставлена

– `comment.left` — комментарий оставлен

– `rating.updated` — рейтинг игры обновлён

3.2.6 Задание 6: Мокирование внешних систем

Для демонстрации работы системы без реальных внешних сервисов реализованы моки всех внешних систем, указанных в архитектурной диаграмме.

3.2.6.1 Платежная система (Эквайринг)

Мокированная платежная система имитирует обработку платежей с вероятностью успеха 90% и случайной генерацией идентификаторов транзакций:

```
class MockPaymentGateway:
    async def process_payment(
        self,
        payment_id: str,
        amount: float,
        payment_method: str
    ) -> Dict[str, str]:
        # Simulate network delay
        await asyncio.sleep(0.5)

        # Randomly succeed or fail (90% success rate)
        success = random.random() > 0.1

        transaction_id = f"TXN_{payment_id[:8]}
_{random.randint(100000, 999999)}"

        if success:
            status = "completed"
        else:
            status = "declined"

        return {
            "transaction_id": transaction_id,
            "status": status,
        }
```

Особенности:

– Имитация сетевой задержки (0.5 секунды)

– Вероятность успешной обработки: 90%

– Генерация уникальных идентификаторов транзакций

3.2.6.2 Система ОФД (Оператор фискальных данных)

Мокированная система ОФД имитирует генерацию фискальных чеков:

```
# Mock OFD (fiscal data operator) - just log
print(f"[OFD] Fiscal receipt generated for payment {payment_id}")
```

При успешной обработке платежа система логирует создание фискального чека.

3.2.6.3 Система уведомлений OneSignal

Мокированный сервис push-уведомлений OneSignal:

```
class MockNotificationService:
    async def send_push_notification(
        self,
        user_id: str,
        title: str,
        message: str
    ) -> Dict[str, bool]:
        await asyncio.sleep(0.2)  # Simulate network delay
        success = random.random() > 0.05  # 95% success rate
        print(f"[OneSignal] Push notification to user {user_id}:
{title} - {message}")
        return {"success": success}
```

Использование: отправка push-уведомлений о дате самовывоза и напоминаний о возврате игр.

3.2.6.4 Система уведомлений SendGrid

Мокированный сервис email-уведомлений SendGrid:

```
async def send_email(
    self,
    email: str,
    subject: str,
    body: str
) -> Dict[str, bool]:
    await asyncio.sleep(0.3)  # Simulate network delay
```

```
    success = random.random() > 0.05  # 95% success rate
    print(f"[SendGrid] Email to {email}: {subject}")
    return {"success": success}
```

Использование: отправка email-уведомлений о самовывозе и возврате игр.

3.2.6.5 Perspective API

Мокированный API для модерации комментариев с анализом токсичности:

```
class MockPerspectiveAPI:
    async def analyze_comment(
        self,
        comment_text: str
    ) -> Tuple[bool, Dict[str, float]]:
        await asyncio.sleep(0.3)  # Simulate network delay

        # Simple mock: check for some basic toxic words
        toxic_words = ["bad", "hate", "stupid", "terrible"]
        is_toxic = any(word in comment_text.lower() for word in
toxic_words)

        # Generate random scores
        toxicity_score = random.uniform(0.7, 0.95) if is_toxic else
random.uniform(0.1, 0.4)
        spam_score = random.uniform(0.1, 0.3)
        profanity_score = random.uniform(0.0, 0.5) if is_toxic else
random.uniform(0.0, 0.2)

        scores = {
            "toxicity": toxicity_score,
            "spam": spam_score,
            "profanity": profanity_score,
        }

        # Comment is moderated if toxicity > 0.5
        is_moderated = toxicity_score > 0.5

        return is_moderated, scores
```

Особенности:

– Проверка на наличие токсичных слов

– Генерация оценок токсичности, спама и ненормативной лексики
– Комментарий помечается как модерируемый при токсичности > 0.5

3.2.7 Задание 7: Управление зависимостями

Каждый микросервис имеет собственный `pyproject.toml` с объявленными зависимостями, что обеспечивает независимость и изолированность каждого сервиса.

3.2.7.1 Структура зависимостей

Все микросервисы организованы как отдельные проекты в рамках UV workspace. Корневой `pyproject.toml` определяет workspace:

```
[project]
name = "code"
version = "0.1.0"
requires-python = ">=3.12"

[tool.uv.workspace]
members = [
    "services/gateway",
    "services/game-catalog",
    "services/booking",
    "services/rating",
    "services/user-account",
    "services/rent",
    "services/payment",
]
```

3.2.7.2 Зависимости по сервисам

Сервис User Account:

```
dependencies = [
    "fastapi>=0.104.0",
    "uvicorn[standard]>=0.24.0",
    "pydantic>=2.5.0",
    "pydantic[email]>=2.5.0",
    "sqlalchemy>=2.0.0",
    "psycopg2-binary>=2.9.0",
    "python-jose[cryptography]>=3.3.0",
    "passlib[bcrypt]>=1.7.4",
```

```
    "python-multipart>=0.0.6",
]
```

Сервис Game Catalog:

```
dependencies = [
    "fastapi>=0.104.0",
    "uvicorn[standard]>=0.24.0",
    "pydantic>=2.5.0",
    "sqlalchemy>=2.0.0",
    "psycopg2-binary>=2.9.0",
    "python-multipart>=0.0.6",
]
```

Сервис Booking:

```
dependencies = [
    "fastapi>=0.104.0",
    "uvicorn[standard]>=0.24.0",
    "pydantic>=2.5.0",
    "httpx>=0.25.0",
    "aio-pika>=9.0.0",
    "python-multipart>=0.0.6",
]
```

Сервис Payment:

```
dependencies = [
    "fastapi>=0.104.0",
    "uvicorn[standard]>=0.24.0",
    "pydantic>=2.5.0",
    "grpcio>=1.60.0",
    "grpcio-tools>=1.60.0",
    "aio-pika>=9.0.0",
    "python-multipart>=0.0.6",
]
```

Сервис Rent:

```
dependencies = [
    "fastapi>=0.104.0",
    "uvicorn[standard]>=0.24.0",
    "pydantic>=2.5.0",
    "sqlalchemy>=2.0.0",
    "psycopg2-binary>=2.9.0",
    "grpcio>=1.60.0",
    "grpcio-tools>=1.60.0",
```

```
    "aio-pika>=9.0.0",
    "httpx>=0.25.0",
    "python-multipart>=0.0.6",
]
```

Сервис Rating:

```
dependencies = [
    "fastapi>=0.104.0",
    "uvicorn[standard]>=0.24.0",
    "pydantic>=2.5.0",
    "aio-pika>=9.0.0",
    "httpx>=0.25.0",
    "python-multipart>=0.0.6",
]
```

Особенности:

– Каждый сервис содержит только необходимые зависимости

– Сервисы с базами данных используют SQLAlchemy и psycopg2-binary

– Сервисы с gRPC используют grpcio и grpcio-tools

– Все сервисы используют aio-pika для работы с RabbitMQ

3.2.8 Реализация

3.2.8.1 Структура проекта

Полная структура проекта микросервисной системы:

```
code/
├── services/
│   ├── user-account/          # Сервис аккаунта пользователя
│   │   ├── main.py            # FastAPI приложение
│   │   ├── schemas.py         # Pydantic модели
│   │   ├── models.py          # SQLAlchemy модели
│   │   ├── database.py       # Конфигурация БД
│   │   ├── auth.py            # Утилиты аутентификации
│   │   ├── rabbitmq_client.py # Клиент RabbitMQ
│   │   ├── Dockerfile         # Docker образ
│   │   └── pyproject.toml    # Зависимости
│   ├── game-catalog/          # Сервис каталога игр
│   │   ├── main.py
│   │   ├── schemas.py
│   │   ├── models.py
│   │   ├── database.py
```

```
│   │   ├── Dockerfile
│   │   └── pyproject.toml
│   ├── booking/                # Сервис бронирования
│   │   ├── main.py
│   │   ├── schemas.py
│   │   ├── user_service.py     # HTTP клиент для User Account
│   │   ├── rabbitmq_client.py
│   │   ├── Dockerfile
│   │   └── pyproject.toml
│   ├── payment/                # Сервис оплаты (с gRPC)
│   │   ├── main.py
│   │   ├── schemas.py
│   │   ├── payment_gateway.py # Мок платежной системы
│   │   ├── grpc_server.py     # gRPC сервер
│   │   ├── rabbitmq_client.py
│   │   ├── proto/
│   │   │   └── payment.proto  # Proto файл для gRPC
│   │   ├── Dockerfile
│   │   └── pyproject.toml
│   ├── rent/                   # Сервис аренды (с gRPC клиентом)
│   │   ├── main.py
│   │   ├── schemas.py
│   │   ├── models.py
│   │   ├── database.py
│   │   ├── grpc_client.py     # gRPC клиент для Payment
│   │   ├── notification_service.py # Моки OneSignal и SendGrid
│   │   ├── rabbitmq_client.py
│   │   ├── Dockerfile
│   │   └── pyproject.toml
│   ├── rating/                 # Сервис оценки
│   │   ├── main.py
│   │   ├── schemas.py
│   │   ├── perspective_api.py  # Мок Perspective API
│   │   ├── rabbitmq_client.py
│   │   ├── Dockerfile
│   │   └── pyproject.toml
│   └── gateway/                # API Gateway
│       ├── app.py              # FastAPI приложение
│       ├── models.py           # Pydantic модели
│       ├── main.py             # Точка входа
│       ├── Dockerfile
│       └── pyproject.toml
├── rabbitmq/                   # Скрипты для работы с RabbitMQ
│   ├── task1_queues.py
│   ├── task2_fanout.py
│   ├── task2_direct.py
```

```
│   └── task2_topic.py
├── pyproject.toml              # Workspace конфигурация
└── docker-compose.yml          # Оркестрация всех сервисов
```

3.2.8.2 Компоненты сервисов

Общие компоненты для всех сервисов:

- `main.py` — основное приложение FastAPI с эндпоинтами
- `schemas.py` — Pydantic модели для валидации запросов и ответов
- `Dockerfile` — конфигурация Docker-образа
- `pyproject.toml` — зависимости проекта

Дополнительные компоненты:

- `database.py` и `models.py` — для сервисов с PostgreSQL (user-account, game-catalog, rent)
- `rabbitmq_client.py` — для всех сервисов, публикующих события
- `grpc_server.py` и `proto/` — для сервиса Payment (gRPC сервер)
- `grpc_client.py` — для сервиса Rent (gRPC клиент)
- `auth.py` — для сервиса User Account (JWT и хеширование паролей)
- Моки внешних систем: `payment_gateway.py`, `notification_service.py`, `perspective_api.py`

3.2.9 Запуск системы

3.2.9.1 Команды для запуска

Для запуска всей системы используется команда:

```
docker compose up --build
```

Для запуска в фоновом режиме:

```
docker compose up --build -d
```

Для просмотра логов:

```
docker compose logs -f
```

Для остановки системы:

```
docker compose down
```

3.2.9.2 Последовательность запуска

Система автоматически выполняет следующие шаги:

- Создание Docker сети для изоляции контейнеров;
- Создание и запуск баз данных PostgreSQL (user-account-db, game-catalog-db, rent-db);
- Запуск RabbitMQ с веб-интерфейсом управления;
- Ожидание готовности баз данных и RabbitMQ через health checks;
- Сборка Docker-образов для всех микросервисов;
- Запуск микросервисов в правильном порядке с учетом зависимостей;
- Проверка здоровья сервисов через health checks каждые 30 секунд.

3.2.9.3 Доступные сервисы

После успешного запуска доступны следующие сервисы:

HTTP API:

- API Gateway: `http://localhost:8000` (Swagger UI: `http://localhost:8000/docs`)
- User Account Service: `http://localhost:8001`
- Game Catalog Service: `http://localhost:8002`
- Booking Service: `http://localhost:8003`
- Payment Service: `http://localhost:8004`
- Rent Service: `http://localhost:8005`
- Rating Service: `http://localhost:8006`

gRPC:

- Payment Service gRPC: `localhost:50051`

Инфраструктура:

- RabbitMQ Management UI: `http://localhost:15672` (логин: `guest`, пароль: `guest`)
- RabbitMQ AMQP: `localhost:5672`

Базы данных:

– User Account DB: `localhost:5433`

– Game Catalog DB: `localhost:5434`

– Rent DB: `localhost:5435`

3.2.9.4 Проверка работоспособности

Для проверки работоспособности всех сервисов можно использовать health check endpoints:

```
curl http://localhost:8000/health  # Gateway
curl http://localhost:8001/health  # User Account
curl http://localhost:8002/health  # Game Catalog
curl http://localhost:8003/health  # Booking
curl http://localhost:8004/health  # Payment
curl http://localhost:8005/health  # Rent
curl http://localhost:8006/health  # Rating
```

Все сервисы должны вернуть ответ вида:

```
{"status": "healthy", "service": "<service_name>"}
```

3.2.9.5 Мониторинг через Docker Compose

Проверка статуса всех контейнеров:

```
docker compose ps
```

Просмотр логов конкретного сервиса:

```
docker compose logs user-account
docker compose logs payment
```

Проверка использования ресурсов:

```
docker stats
```

3.3 Вывод

В ходе выполнения практической работы была реализована полнофункциональная микросервисная система аренды настольных игр. Все микросервисы упакованы в Docker-контейнеры и оркестрированы с помощью Docker Compose. Реализована синхронная коммуникация

через gRPC между сервисами Rent и Payment, а также асинхронная коммуникация через RabbitMQ для публикации доменных событий. Система включает три базы данных PostgreSQL, мокированные внешние сервисы и полностью функциональные эндпоинты для всех агрегатов. Получены практические навыки контейнеризации микросервисов, настройки межсервисного взаимодействия и оркестрации распределенных систем.

4 Практическая работа №8

4.1 Цель работы

Разработка и настройка тестирования микросервисной системы. Реализация интеграционных и компонентных тестов для всех сервисов. Настройка CI/CD конвейера для автоматического выполнения тестов.

4.2 Ход работы

4.2.1 Задание 1: Выбор типов тестирования

Из трех доступных типов тестирования (unit, интеграционное, компонентное) были выбраны два:

– **Интеграционное тестирование** — для сервисов с базами данных (User Account, Game Catalog, Rent);
– **Компонентное тестирование** — для сервисов с внешними зависимостями (Booking, Payment, Rating).

Обоснование выбора:

– Интеграционные тесты необходимы для проверки взаимодействия сервисов с базами данных
– Компонентные тесты позволяют изолировать сервисы и мокировать внешние зависимости
– Unit-тесты не были выбраны, так как основная логика уже покрыта интеграционными и компонентными тестами

4.2.2 Задание 2: Реализация интеграционных тестов

Интеграционные тесты реализованы для сервисов, работающих с базами данных PostgreSQL.

4.2.2.1 Сервис User Account

Реализованы интеграционные тесты для всех основных операций:

– Регистрация пользователя;
– Авторизация пользователя;

– Получение информации о пользователе;
– Обновление профиля;
– Блокировка пользователя;
– Проверка дублирования email;
– Проверка авторизации заблокированного пользователя.

Особенности реализации:

– Использование SQLite in-memory для изоляции тестов
– Переопределение database engine в тестовом окружении
– Автоматическая настройка и очистка базы данных для каждого теста

Пример теста:

```
def test_register_user_success(self, client):
    """Test successful user registration."""
    response = client.post(
        "/api/v1/users/register",
        json={
            "email": "test@example.com",
            "password": "securepassword123",
            "first_name": "John",
            "last_name": "Doe",
            "phone": "+79991234567"
        }
    )
    assert response.status_code == status.HTTP_201_CREATED
    data = response.json()
    assert data["email"] == "test@example.com"
    assert data["first_name"] == "John"
    assert "user_id" in data
```

4.2.2.2 Сервис Game Catalog

Реализованы интеграционные тесты для операций с каталогом игр:

– Добавление игры в каталог;
– Получение информации об игре;
– Обновление информации об игре;
– Поиск игр по различным критериям;
– Обновление доступности игр.

Пример теста:

```
def test_add_game(self, client):
    """Test adding a new game to catalog."""
    response = client.post(
        "/api/v1/games",
        json={
            "name": "Каркассон",
            "description": "Стратегическая настольная игра",
            "min_players": 2,
            "max_players": 5,
            "price_per_day": 150.0,
            "total_copies": 10
        }
    )
    assert response.status_code == status.HTTP_201_CREATED
    data = response.json()
    assert data["name"] == "Каркассон"
    assert data["status"] == "available"
```

4.2.2.3 Сервис Rent

Реализованы интеграционные тесты для операций аренды:

– Создание заказа на аренду;

– Подтверждение получения игры;

– Возврат игры;

– Начисление штрафа;

– Получение информации о заказе.

Особенности:

– Мокирование gRPC вызовов к сервису Payment

– Мокирование внешних сервисов уведомлений

– Проверка изменения статусов заказов

4.2.3 Задание 3: Реализация компонентных тестов

Компонентные тесты реализованы для сервисов с внешними зависимостями, где используются моки.

4.2.3.1 Сервис Booking

Компонентные тесты проверяют логику бронирования с мокированным сервисом User Account:

– Создание бронирования с валидным пользователем;

– Отказ в создании бронирования для несуществующего пользователя;

– Отмена бронирования;

– Получение информации о бронировании;

– Проверка прав доступа при отмене.

Пример теста:

```
def test_book_game_success(self, client, mock_user_service):
    """Test successful game booking with mocked user validation."""
    # Mock user validation to return True
    mock_user_service.return_value = True

    response = client.post(
        "/api/v1/bookings",
        json={
            "game_id": "game-123",
            "user_id": "user-456",
            "booking_date": "2024-01-20T10:00:00",
            "pickup_date": "2024-01-21T10:00:00"
        }
    )
    assert response.status_code == 201
    data = response.json()
    assert data["game_id"] == "game-123"
    assert data["status"] == "pending"
```

4.2.3.2 Сервис Payment

Компонентные тесты проверяют обработку платежей с мокированным платежным шлюзом:

– Инициирование платежа;

– Успешная обработка платежа;

– Отклонение платежа;

– Запрос на возврат средств;

– Обработка возврата;

– Получение информации о платеже.

Особенности:

– Мокирование платежного шлюза (Эквайринг)

– Проверка различных сценариев обработки платежей

– Тестирование логики возврата средств

4.2.3.3 Сервис Rating

Компонентные тесты проверяют функциональность оценок и комментариев:

– Оставление оценки игре;

– Оставление комментария с модерацией;

– Обработка токсичных комментариев;

– Получение информации об оценке;

– Получение информации о комментарии.

Особенности:

– Мокирование Perspective API для модерации

– Проверка логики модерации комментариев

– Тестирование обновления рейтинга игр

4.2.4 Задание 4: Настройка зависимостей для тестирования

Все тестовые зависимости вынесены в отдельную группу `test` в `pyproject.toml` каждого сервиса с использованием UV dependency groups.

Пример конфигурации:

```
[project]
name = "user-account"
version = "0.1.0"
requires-python = ">=3.12"
dependencies = [
    "fastapi>=0.104.0",
    "sqlalchemy>=2.0.0",
    # ... основные зависимости
]
```

```
[dependency-groups]
test = [
    "pytest>=7.4.0",
    "pytest-asyncio>=0.21.0",
    "httpx>=0.25.0",
    "pytest-cov>=4.1.0",
]
```

Запуск тестов:

```
uv sync --group test
uv run --group test pytest tests/ -v
```

4.2.5 Задание 5: Настройка CI/CD конвейера

Реализован CI/CD конвейер на GitHub Actions для автоматического выполнения тестов при каждом push и pull request.

4.2.5.1 Конфигурация GitHub Actions

Файл `.github/workflows/ci.yml` содержит:

- Триггеры на push и pull request в ветки main и develop;
- Матричную стратегию для параллельного тестирования всех сервисов;
- Настройку сервисов PostgreSQL и RabbitMQ для тестирования;
- Установку зависимостей через UV;
- Запуск тестов с покрытием кода;
- Загрузку отчетов о покрытии в Codecov.

Структура workflow:

```
name: CI/CD Pipeline

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main, develop ]

jobs:
```

```yaml
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        service:
          - user-account
          - game-catalog
          - booking
          - payment
          - rent
          - rating

    services:
      postgres:
        image: postgres:15-alpine
        # ... конфигурация
      rabbitmq:
        image: rabbitmq:3-management-alpine
        # ... конфигурация

    steps:
      - uses: actions/checkout@v4
      - name: Set up Python
        uses: actions/setup-python@v5
      - name: Install UV
        run: curl -LsSf https://astral.sh/uv/install.sh | sh
      - name: Install dependencies
        run: uv sync --group test
      - name: Run tests
        run: uv run --group test pytest tests/ -v --cov=. --cov-
report=xml
      - name: Upload coverage
        uses: codecov/codecov-action@v3
```

4.2.5.2 Особенности CI/CD конвейера

– Параллельное выполнение тестов для всех сервисов;

– Использование сервисов GitHub Actions для PostgreSQL и RabbitMQ;

– Автоматическая установка зависимостей через UV;

– Генерация отчетов о покрытии кода;

– Интеграция с Codecov для отслеживания покрытия.

4.2.6 Реализация

4.2.6.1 Структура тестов

Тесты организованы в директории `tests/` для каждого сервиса:

```
code/services/
├── user-account/
│   ├── tests/
│   │   ├── __init__.py
│   │   ├── conftest.py          # Pytest конфигурация и fixtures
│   │   └── test_integration.py  # Интеграционные тесты
│   └── pyproject.toml
├── game-catalog/
│   ├── tests/
│   │   ├── __init__.py
│   │   ├── conftest.py
│   │   └── test_integration.py
│   └── pyproject.toml
├── booking/
│   ├── tests/
│   │   ├── __init__.py
│   │   └── test_component.py   # Компонентные тесты
│   └── pyproject.toml
├── payment/
│   ├── tests/
│   │   ├── __init__.py
│   │   └── test_component.py
│   └── pyproject.toml
├── rent/
│   ├── tests/
│   │   ├── __init__.py
│   │   ├── conftest.py
│   │   └── test_integration.py
│   └── pyproject.toml
└── rating/
    ├── tests/
    │   ├── __init__.py
    │   └── test_component.py
    └── pyproject.toml
```

4.2.6.2 Pytest конфигурация

Для сервисов с базами данных используется `conftest.py` с настройкой тестовой базы данных:

```
# Test database URL (SQLite in-memory for tests)
TEST_DATABASE_URL = "sqlite:///:memory:"

# Override database before importing main
test_engine = create_engine(
    TEST_DATABASE_URL,
    connect_args={"check_same_thread": False},
    poolclass=StaticPool,
)

# Override the engine in database module
import database
database.engine = test_engine

@pytest.fixture(scope="function", autouse=True)
def setup_test_db():
    """Set up and tear down test database for each test."""
    Base.metadata.create_all(bind=test_engine)
    yield
    Base.metadata.drop_all(bind=test_engine)
```

4.2.6.3 Мокирование зависимостей

Для компонентных тестов используются моки:

- **pytest-mock** — для создания моков функций и методов;
- **unittest.mock** — для мокирования внешних сервисов;
- Переопределение зависимостей через `app.dependency_overrides` в FastAPI.

4.2.7 Запуск тестов

4.2.7.1 Локальный запуск

Для запуска тестов локально:

```
# Установка тестовых зависимостей
uv sync --group test

# Запуск всех тестов
uv run --group test pytest tests/ -v

# Запуск с покрытием кода
```

```
uv run --group test pytest tests/ -v --cov=. --cov-report=term

# Запуск конкретного теста
uv run --group test pytest tests/
test_integration.py::TestUserRegistration::test_register_user_success
-v
```

4.2.7.2 Запуск в CI/CD

Тесты автоматически выполняются при каждом push и pull request через GitHub Actions. Результаты тестирования и покрытие кода отображаются в интерфейсе GitHub и Codecov.

4.3 Вывод

В ходе выполнения практической работы были реализованы интеграционные и компонентные тесты для всех микросервисов системы. Интеграционные тесты проверяют взаимодействие сервисов с базами данных, а компонентные тесты изолируют сервисы и проверяют их логику с мокированными зависимостями. Настроен CI/CD конвейер на GitHub Actions для автоматического выполнения тестов при каждом изменении кода. Получены практические навыки написания тестов для микросервисных систем, настройки тестового окружения и автоматизации процесса тестирования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Камышев, М. Моделирование микросервисов с помощью Event storming / М. Камышев
2. Кидяшев, А. Event Storming: что будет, если запереть 10 человек в одной комнате / А. Кидяшев
3. Эванс, Э. Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем / Э. Эванс ; пер. с англ. — Москва : Вильямс, 2011. — 448 с.
4. Brandolini, A. Introducing EventStorming: An Act of Deliberate Collective Learning / A. Brandolini. — Text : electronic
5. Richardson, C. Microservices Patterns: With Examples in Java / C. Richardson. — Manning Publications, 2018. — 520 p.
6. Docker Documentation. — URL: https://docs.docker.com/
7. RabbitMQ Documentation. — URL: https://www.rabbitmq.com/documentation.html
8. gRPC Documentation. — URL: https://grpc.io/docs/