ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Факультет Информационных технологий

Кафедра Информатики и информационных технологий

направление подготовки

09.03.02 «Информационные системы и технологии»

Профиль «Программное обеспечение игровой компьютерной индустрии»

КУРСОВОЙ ПРОЕКТ

Тема: Серверная часть трекера коллекции видеоигр.

Дисциплина: BackEnd-разработка

Выполнил: студент группы 231-339

Иргит Ян Валерьевич

(Фамили И.О.)

Дата, подпись: 29.12.2025 

(дата) (подпись)

Проверил:

(Фамилия И. О., степень, звание) (оценка)

Дата, подпись:

(дата) (подпись)

Москва

2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Рост объёма цифрового контента и распространение цифровой дистрибуции привели к тому, что у одного пользователя может накапливаться значительное количество приобретённых и пройденных игр. В отсутствие систематизации данные о платформе, статусе прохождения, оценке и заметках быстро теряются, что осложняет планирование игрового времени и ведение персонального «бэклога».

Актуальность выбранной темы обусловлена тем, что даже при наличии крупных экосистем (магазины, лаунчеры, сервисы статистики) пользователю часто требуется лёгкий автономный инструмент, который предоставляет единообразный интерфейс работы с коллекцией и может быть развернут локально либо в облачной инфраструктуре.

Целью курсового проекта является разработка серверной системы учёта коллекции видеоигр на основе микросервисной архитектуры с использованием REST API и асинхронного обмена сообщениями. В рамках проекта реализуются подсистемы регистрации и аутентификации, управления элементами коллекции и регистрации событий, формируемых при изменении коллекции.

Для достижения поставленной цели решаются следующие задачи:

- выполнить анализ предметной области и сформировать требования к системе;
- спроектировать структуру хранения данных и интерфейсы взаимодействия компонентов;
- реализовать сервис аутентификации с выпуском JWT и хранением учётных данных в СУБД;
- реализовать сервис управления коллекцией с минимальным CRUD и проверкой прав доступа;
- реализовать асинхронную доставку событий через брокер сообщений RabbitMQ и сервис статистики, сохраняющий полученные события;
- обеспечить контейнеризацию компонентов и запуск системы через Docker Compose;

- обеспечить журналирование и унифицированную обработку ошибок.

Структура пояснительной записки включает две главы: аналитическую и технологическую, заключение, список использованных источников и приложения с дополнительными материалами.

1 АНАЛИТИЧЕСКАЯ ЧАСТЬ

1.1 Анализ предметной области

Предметной областью проекта является ведение персональной коллекции видеоигр. Для каждого пользователя требуется хранить перечень игр, их платформу (например, PC, PlayStation, Xbox, Switch), текущий статус (запланировано, в процессе, завершено, брошено) и дополнительные атрибуты: субъективную оценку и текстовую заметку.

Типовой сценарий использования включает регистрацию пользователя, авторизацию и последующую работу с коллекцией: добавление новой игры, редактирование статуса и оценки, просмотр списка, а также удаление элементов. Важно обеспечить изоляцию данных: пользователь должен видеть и изменять только собственные записи.

Дополнительно в проекте рассматривается регистрация событий изменения коллекции. При добавлении, обновлении или удалении элемента формируется событие, которое доставляется асинхронно в отдельный сервис и сохраняется как журнал действий. Такой подход позволяет отделить пользовательский контур (HTTP-запросы) от фоновой обработки событий и облегчает дальнейшее расширение (например, расчёт статистики по платформам или построение отчётов).

1.2 Обоснование необходимости создания проекта

Существующие коммерческие платформы зачастую ориентированы на собственную экосистему и не всегда предоставляют единый способ учёта игр с разных источников. Кроме того, часть сервисов требует передачи персональных данных внешним провайдерам или не поддерживает локальное развёртывание.

Рассматриваемая система предназначена для демонстрации подходов проектирования и реализации серверного программного обеспечения: выделения функциональных зон, организации обмена данными, обеспечения безопасности и наблюдаемости. Использование микросервисной архитектуры позволяет

показать практики декомпозиции и интеграции, которые востребованы в современной разработке.

1.3 Обзор существующих решений

В качестве ориентиров можно выделить следующие типы решений: игровые магазины и лаунчеры, которые учитывают покупки в рамках одной площадки; сайты ведения «бэклога» и каталогов, где пользователь вручную отмечает статус прохождения; агрегаторы метаданных и времени прохождения. Несмотря на функциональную насыщенность, такие решения нередко имеют ограничения по интеграции и автономности.

С учётом учебных целей целесообразно реализовать минимальный, но полнофункциональный серверный контур, в котором демонстрируются ключевые практики: авторизация, доступ к данным по пользователю, обмен сообщениями, контейнеризация и журналирование. В качестве технологической базы выбран стек Python + FastAPI + SQLAlchemy, обеспечивающий быстрый выпуск корректно типизированного API, а также удобство сопровождения и расширения [1] [2].

1.4 Выявление проблем и ограничений

Основным ограничением предметной области является неоднородность источников данных о видеоиграх и отсутствие единого идентификатора для всех платформ. В рамках курсового проекта для упрощения не выполняется интеграция с внешними каталогами, а запись игры создаётся пользователем вручную.

Также следует учитывать, что микросервисная архитектура повышает сложность развёртывания и диагностики по сравнению с монолитом. Для компенсации этого фактора требуется централизованный запуск (Docker Compose) и детализированное логирование.

С точки зрения безопасности в проекте необходимо предотвратить утечки токенов и обеспечить корректную проверку подписи JWT. В качестве

криптографического хеширования паролей применяется алгоритм bcrypt, рассчитанный на противодействие офлайн-подбору паролей за счёт настраиваемой вычислительной сложности.

2 ТЕХНОЛОГИЧЕСКАЯ ЧАСТЬ

2.1 Архитектура системы

Система реализована в виде набора контейнеризованных компонентов: трёх прикладных микросервисов, брокера сообщений RabbitMQ и СУБД PostgreSQL. Взаимодействие с пользователем осуществляется по HTTP через REST API, а обмен событиями между сервисами — по протоколу AMQP через топиковый обменник[3] .

Разделение по микросервисам выполнено следующим образом: Auth Service отвечает за управление учётными данными и выпуск токенов; Collection Service обеспечивает операции CRUD над коллекцией и публикует события изменений; Stats Service подписывается на события и сохраняет их в журнале для последующего просмотра.

Выбор FastAPI обусловлен поддержкой типизации и автоматической генерацией OpenAPI, что повышает качество интерфейсов и упрощает тестирование. Для доступа к БД используется SQLAlchemy, позволяющая описывать модели декларативным способом и формировать таблицы при старте сервисов.

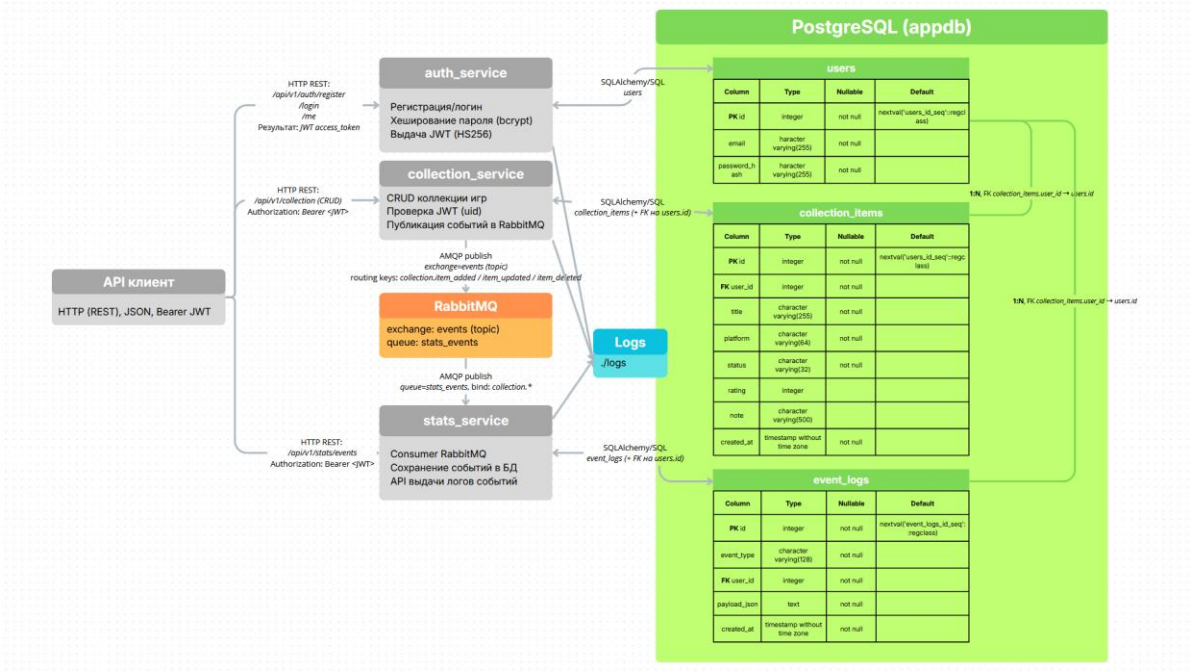

Рисунок 1 — Компонентная схема системы и ER-диаграмма БД

Компоненты развёртываются совместно через docker-compose.yml. Доступ к сервисам осуществляется по портам, проброшенным из контейнеров во внешнюю среду. Взаимодействие с БД производится через внутреннюю сеть Docker по имени сервиса postgres

2.2 Диаграмма компонентов и сценарии взаимодействия

Асинхронная схема обмена данными применяется для регистрации событий. При изменении коллекции HTTP-ответ формируется после записи в БД. Публикация события выполняется в том же обработчике, но ошибка публикации не приводит к отказу в обслуживании, так как исключения перехватываются. Потребитель сообщений обрабатывает события независимо от пользовательских запросов.

2.3 Модель данных и диаграмма базы данных

В качестве хранилища используется PostgreSQL, поддерживающая транзакционность, целостность данных и развитый инструментарий администрирования[4] . Таблицы создаются каждым микросервисом только в пределах собственной зоны ответственности. Для корректного формирования внешних ключей на users.id в Collection и Stats применяется «заглушка» таблицы users в metadata ORM.

Основные сущности представлены тремя таблицами: users, collection_items и event_logs. В таблице event_logs поле payload_json хранит сериализованную структуру события и позволяет сохранять произвольный набор полей без изменения схемы при развитии проекта.

2.4 Описание функциональных модулей

2.4.1 Auth Service

Auth Service реализует регистрацию и аутентификацию. При регистрации выполняется проверка уникальности email, хеширование пароля и сохранение

пользователя. При входе выполняется проверка пароля по хешу и формирование токена доступа.

JWT соответствует стандарту RFC 7519 и содержит идентификатор пользователя (uid) и email (sub). Подпись выполняется алгоритмом HS256 с секретным ключом JWT_SECRET[5] . Рекомендации по безопасному применению токенов и проверке алгоритма подписи приведены в материалах OWASP[6] .

В целях упрощения демонстрации используется единый секрет в переменных окружения всех сервисов. В дальнейшем возможен переход на асимметричную подпись и отдельный сервис авторизации.

Ключевой фрагмент формирования токена представлен в Листинг 1.

```
def create_access_token(subject: str, user_id: int) -> str:
    now = datetime.now(timezone.utc)
    exp = now + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    payload = {
        "sub": subject,
        "uid": int(user_id),
        "iat": int(now.timestamp()),
        "exp": int(exp.timestamp()),
    }
    return jwt.encode(payload, JWT_SECRET, algorithm=JWT_ALG)
```

Листинг 1 — Формирование JWT в Auth Service

2.4.2 Collection Service

Collection Service реализует операции работы с коллекцией. Валидация входных данных и форматирование ответов выполняются через Pydantic-схемы, что снижает риск передачи некорректных значений в БД. Для идентификации пользователя используется зависимость get_current_user_id, извлекающая uid из токена.

Проверка принадлежности элемента коллекции выполняется на уровне запросов к БД: операции чтения/обновления/удаления фильтруются одновременно по item_id и user_id. Такой подход исключает доступ к чужим данным даже при подборе идентификаторов.

Публикация событий производится в RabbitMQ с параметром delivery_mode=2 (персистентные сообщения). Это повышает вероятность доставки при перезапуске брокера при условии наличия durable-очереди на стороне потребителя.

Листинг 2 иллюстрирует публикацию события с гарантированным закрытием соединения.

```
def publish_event(routing_key: str, payload: dict) -> None:
    connection = None
    try:
        params = pika.URLParameters(RABBITMQ_URL)
        connection = pika.BlockingConnection(params)
        channel = connection.channel()

        channel.exchange_declare(exchange=EXCHANGE,
exchange_type=EXCHANGE_TYPE, durable=True)

        body = json.dumps(payload, ensure_ascii=False).encode("utf-8")
        channel.basic_publish(
            exchange=EXCHANGE,
            routing_key=routing_key,
            body=body,
            properties=pika.BasicProperties(content_type="application/json",
delivery_mode=2),
        )

        logging.info("Published event %s", routing_key)
    except Exception:
        logging.exception("Failed to publish event %s", routing_key)
    finally:
        try:
            if connection and connection.is_open:
                connection.close()
        except Exception:
            logging.exception("Failed to close RabbitMQ connection")
```

Листинг 2 — Публикация событий в RabbitMQ (Collection Service)

2.4.3 Stats Service

Stats Service содержит два контура: HTTP API для выдачи данных пользователю и фонового потребителя AMQP. Потребитель объявляет очередь stats_events и привязывает её к обменнику events по шаблону collection.*. Для каждого сообщения выполняется сохранение в таблицу event_logs с указанием user_id и типа события.

Фоновая обработка запускается при старте приложения. Для исключения неконтролируемых остановок потребитель работает в цикле с перезапуском: при исключении выполняется задержка и повторная попытка подключения.

Листинг 3 демонстрирует общую структуру потребителя с повторными попытками подключения.

```
def run_consumer_forever():
    while True:
        conn = None
        try:
            params = pika.URLParameters(RABBITMQ_URL)
```

```
            conn = pika.BlockingConnection(params)
            ch = conn.channel()

            ch.exchange_declare(exchange=EXCHANGE, exchange_type="topic",
durable=True)
            ch.queue_declare(queue=QUEUE, durable=True)
            ch.queue_bind(exchange=EXCHANGE, queue=QUEUE,
routing_key=BIND_KEY)

            ch.basic_qos(prefetch_count=10)
            ch.basic_consume(queue=QUEUE, on_message_callback=_handle_message)

            logging.info("Stats consumer started. Waiting for messages...")
            ch.start_consuming()
        except Exception:
            logging.exception("Consumer crashed, retry in 3s...")
            time.sleep(3)
        finally:
            try:
                if conn and conn.is_open:
                    conn.close()
            except Exception:
                logging.exception("Failed to close RabbitMQ connection")
```

Листинг 3 — Устойчивый потребитель RabbitMQ (Stats Service)

2.5 Интерфейсы взаимодействия

Внешние интерфейсы представлены REST API (JSON) каждого сервиса. Для тестирования доступны автоматически генерируемые страницы документации Swagger UI (/docs) и спецификация OpenAPI (/openapi.json), предоставляемые FastAPI.

Сервис	**Метод**	**Пусть**	**Назначение**
Auth	POST	/api/v1/auth/register	Регистрация (email, пароль)
Auth	POST	/api/v1/auth/login	Вход и получение JWT
Auth	GET	/api/v1/auth/me	Получение email текущего пользователя
Collection	GET	/api/v1/collection	Список игр пользователя
Collection	POST	/api/v1/collection	Добавление игры
Collection	GET	/api/v1/collection/{item_id}	Получени игры по IP
Collection	PATCH	/api/v1/collection/{item_id}	Обновление полей игры
Collection	DELETE	/api/v1/collection/{item_id}	Удаление игры
Stats	GET	api/v1/stats/events	Последние 50 событий пользователя

Таблица 1 — REST API системы

Асинхронный интерфейс обмена событиями построен на топиковом обменнике RabbitMQ, который маршрутизирует сообщения по ключу маршрутизации и шаблону привязки очереди.

2.6 Процесс разработки и развёртывания

Конфигурация сервисов вынесена в переменные окружения. Такой подход позволяет запускать одинаковые контейнеры в разных средах, изменяя только параметры подключения и секреты.

Контейнеризация выполнена с применением Docker. Для запуска всей системы используется Docker Compose, позволяющий описать набор сервисов, сети, тома и зависимости в одном файле[7] . В docker-compose.yml определены: postgres, rabbitmq и три прикладных микросервиса, а также проброс портов и общий том для логов.

При запуске каждый сервис выполняет проверку доступности БД и при необходимости создаёт собственные таблицы. Механизм повторных попыток позволяет избежать ошибки старта в случае, если PostgreSQL ещё не готова принять соединение.

Листинг 4 демонстрирует фрагмент инициализации таблицы с повторными попытками подключения.

```
@app.on_event("startup")
def on_startup():
    from .db import Base

    for attempt in range(1, 31):
        try:
            Base.metadata.create_all(bind=engine,
tables=[CollectionItem.__table__])
            logger.info("DB schema ensured (attempt %s)", attempt)
            break
        except Exception:
            logger.exception("DB init failed (attempt %s/30). Retrying in
2s...", attempt)
            time.sleep(2)
    else:
        raise RuntimeError("DB init failed after retries")
```

Листинг 4 — Инициализация схемы с повторными попытками (Collection Service)

2.7 Обработка ошибок и логирование

Во всех сервисах реализованы обработчики исключений FastAPI и Starlette. Ошибки валидации входных данных преобразуются в ответ с кодом 400 и подробным описанием. Штатные HTTP-ошибки логируются и возвращаются клиенту.

Для непредвиденных исключений используется глобальный обработчик, который возвращает унифицированное сообщение об ошибке и фиксирует стек

вызова в логе. Такой подход повышает безопасность (не раскрываются детали внутренней реализации) и упрощает сопровождение.

Журналирование выполняется через модуль logging. Для записи в файл применяется RotatingFileHandler с кодировкой UTF-8 и ограничением размера файла. Одновременно логи выводятся в консоль контейнера.

Листинг 5 показывает настройку файлового логирования с ротацией.

```
file_handler = RotatingFileHandler(
log_path,
maxBytes=2_000_000,
backupCount=3,
encoding="utf-8",
)
```

Листинг 5 — Ротация логов (общий подход)

2.8 Тестирование и демонстрация работоспособности

Проверка корректности функционирования выполнялась посредством ручного тестирования через Swagger UI и утилиту curl. Сценарий включает регистрацию, получение токена, операции CRUD над коллекцией и проверку журнала событий.

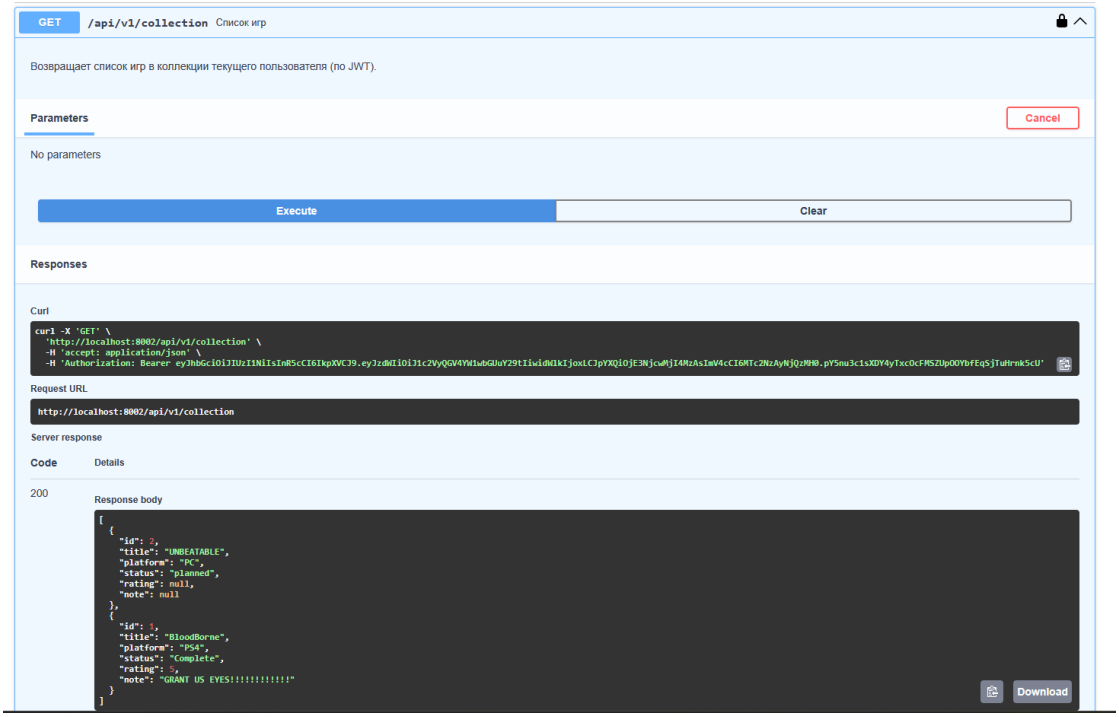

Рисунок 2 — Результат операции получения коллекции видеоигр в Swagger UI

Для демонстрации разграничения доступа выполнялись обращения к Collection Service без токена и с некорректным токеном. В обоих случаях возвращался код 401, а факт ошибки фиксировался в логах. Таким образом подтверждается выполнение требования N1.

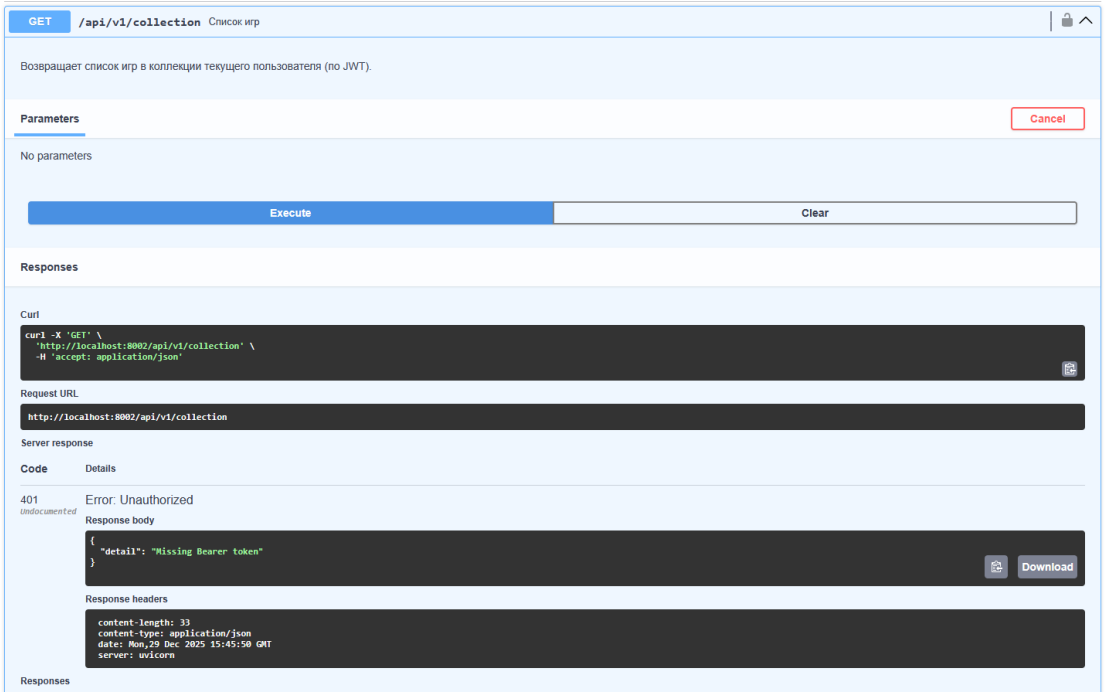

Рисунок 3 — доступа к методу нет для неавторизированных пользователей

Корректность асинхронного обмена подтверждается появлением записи в event_logs после изменения коллекции. При этом запись в журнал может появляться с небольшой задержкой, обусловленной асинхронной доставкой сообщения и обработкой потребителем.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

id	event_type	user_id	payload_json	created_at
1	collection.item_added	1	{"user_id": 1, "item_id": 1, "title": "BloodBorne", "platform": "PS4"}	2025-12-28 12:04:47.88831
2	collection.item_updated	1	{"user_id": 1, "item_id": 1, "status": "Complete", "rating": 5}	2025-12-28 12:06:56.522831
3	collection.item_updated	1	{"user_id": 1, "item_id": 1, "status": "Complete", "rating": 5}	2025-12-28 12:06:57.551276
4	collection.item_updated	1	{"user_id": 1, "item_id": 1, "status": "Complete", "rating": 5}	2025-12-28 12:06:59.249206
5	collection.item_updated	1	{"user_id": 1, "item_id": 1, "status": "Complete", "rating": 5}	2025-12-28 12:06:59.446498
6	collection.item_updated	1	{"user_id": 1, "item_id": 1, "status": "Complete", "rating": 5}	2025-12-28 12:06:59.60746
7	collection.item_updated	1	{"user_id": 1, "item_id": 1, "status": "Complete", "rating": 5}	2025-12-28 12:06:59.760301
8	collection.item_updated	1	{"user_id": 1, "item_id": 1, "status": "Complete", "rating": 5}	2025-12-28 12:06:59.914166
9	collection.item_updated	1	{"user_id": 1, "item_id": 1, "status": "Complete", "rating": 5}	2025-12-28 12:07:00.054989
10	collection.item_updated	1	{"user_id": 1, "item_id": 1, "status": "Complete", "rating": 5}	2025-12-28 12:07:00.224339
11	collection.item_updated	1	{"user_id": 1, "item_id": 1, "status": "Complete", "rating": 5}	2025-12-28 12:07:00.380615
12	collection.item_updated	1	{"user_id": 1, "item_id": 1, "status": "Complete", "rating": 5}	2025-12-28 12:07:00.517133
13	collection.item_updated	1	{"user_id": 1, "item_id": 1, "status": "Complete", "rating": 5}	2025-12-28 12:07:00.641084
14	collection.item_updated	1	{"user_id": 1, "item_id": 1, "status": "Complete", "rating": 5}	2025-12-28 12:07:00.792531
15	collection.item_updated	1	{"user_id": 1, "item_id": 1, "status": "Complete", "rating": 5}	2025-12-28 12:07:00.948185
16	collection.item_updated	1	{"user_id": 1, "item_id": 1, "status": "Complete", "rating": 5}	2025-12-28 12:07:01.089928
17	collection.item_updated	1	{"user_id": 1, "item_id": 1, "status": "Complete", "rating": 5}	2025-12-28 12:07:01.234153
18	collection.item_added	1	{"user_id": 1, "item_id": 2, "title": "UNBEATABLE", "platform": "PC"}	2025-12-28 12:16:46.98055
19	collection.item_updated	1	{"user_id": 1, "item_id": 2, "status": "Playing", "rating": 6}	2025-12-29 15:50:15.951771

(19 rows)

Рисунок 4 — event_logs

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсового проекта разработана микросервисная система учёта коллекции видеоигр, включающая сервис аутентификации, сервис управления коллекцией и сервис регистрации событий. Реализованы защищённые REST API, хранение данных в PostgreSQL и асинхронная доставка событий через RabbitMQ.

Поставленные задачи выполнены: сформированы требования, спроектирована модель данных, реализованы ключевые функциональные модули и обеспечен контейнеризованный запуск через Docker Compose. Реализованы логирование и обработка ошибок, обеспечивающие предсказуемое поведение системы и удобство диагностики.

Дальнейшее развитие может включать: интеграцию с внешними каталогами игр, расширение модели коллекции (жанры, время прохождения, теги), построение статистических отчётов, добавление API Gateway и централизованной авторизации, а также внедрение автоматизированного тестирования и CI/CD.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] FastAPI Documentation. — Электронный ресурс. — Режим доступа: https://fastapi.tiangolo.com/ (дата обращения: 29.12.2025).

[2] SQLAlchemy Documentation: Declarative mapping / mapped_column(). — Электронный ресурс. — Режим доступа: https://docs.sqlalchemy.org/ (дата обращения: 29.12.2025).

[3] RabbitMQ Documentation: Exchanges; AMQP concepts. — Электронный ресурс. — Режим доступа: https://www.rabbitmq.com/docs (дата обращения: 29.12.2025).

[4] PostgreSQL Documentation. — Электронный ресурс. — Режим доступа: https://www.postgresql.org/docs/ (дата обращения: 29.12.2025).

[5] RFC 7519: JSON Web Token (JWT). — IETF. — Электронный ресурс. — Режим доступа: https://datatracker.ietf.org/doc/html/rfc7519 (дата обращения: 29.12.2025).

[6] OWASP Cheat Sheet Series: JSON Web Token Cheat Sheet. — Электронный ресурс. — Режим доступа: https://cheatsheetseries.owasp.org/ (дата обращения: 29.12.2025).

[7] Docker Documentation: Compose file reference. — Электронный ресурс. — Режим доступа: https://docs.docker.com/reference/compose-file/ (дата обращения: 29.12.2025).