ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ   
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ   
ВЫСШЕГО ОБРАЗОВАНИЯ  
МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Отчёт по лабораторной работе №2

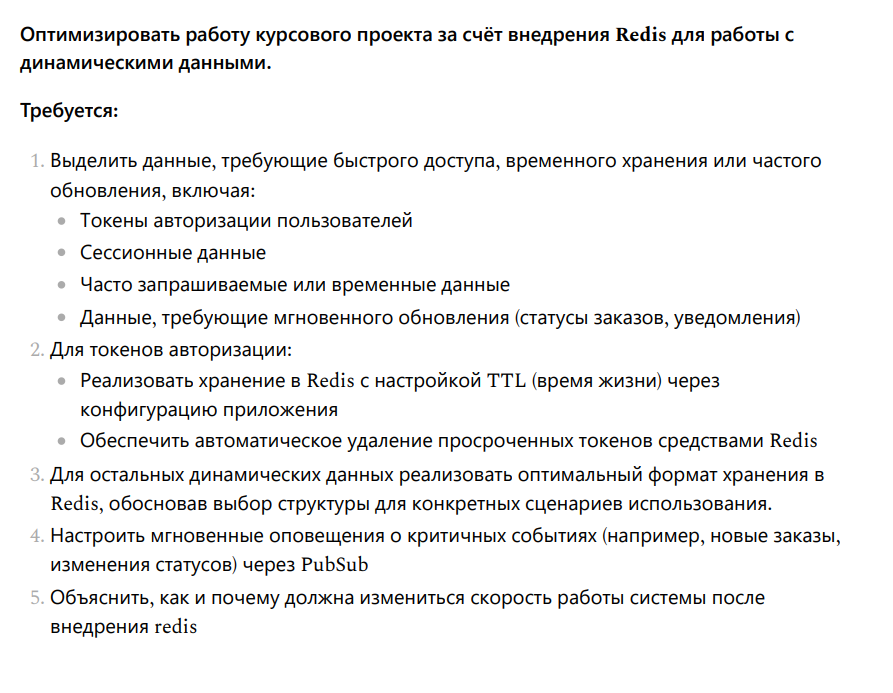
по курсу «Базы данных»

Студент: Андрюшин Л.Д.

Группа: М8О-312Б-21

Преподаватель: Сеченых П.А.

Москва 2025



**Теория**

**Redis**

Redis — это высокопроизводительная система управления базами данных, работающая в памяти и поддерживающая различные структуры данных, такие как строки, списки, множества, хэши и другие. Она предназначена для очень быстрого доступа к данным и широко используется как кеш, брокер сообщений или временное хранилище данных в высоконагруженных веб-приложениях. В контексте нашей системы Redis был внедрён прежде всего как средство кэширования часто используемых данных и как инструмент для передачи уведомлений между различными частями приложения — например, для оповещения администратора о регистрации нового клиента.

Прежде чем ввести Redis, система взаимодействовала с постоянным хранилищем данных (обычно это реляционная база данных), что приводило к относительно медленному отклику при повторных запросах одних и тех же данных. Каждый такой запрос требовал установления соединения с базой, выполнения SQL-запроса, обработки ответа и его передачи в интерфейс пользователя. Особенно критично это становилось при работе с данными, которые не изменяются часто, но запрашиваются регулярно — например, списки игр, количество доступных копий, информация о зарегистрированных клиентах. В таких случаях каждый повторный запрос создавал избыточную нагрузку на сервер базы данных, увеличивал время отклика и снижал масштабируемость системы.

С внедрением Redis стало возможным сохранять результаты таких часто повторяющихся операций в памяти и извлекать их практически мгновенно. Поскольку Redis работает в оперативной памяти, чтение из него происходит на порядки быстрее, чем обращение к дисковым БД. Это дало нам прирост скорости и отзывчивости системы при минимальной доработке кода. Кроме того, мы использовали Redis как механизм временного хранения и управления сессионными данными, включая авторизационные токены. Это позволило нам быстро проверять подлинность пользователей без необходимости каждый раз обращаться к основной БД.

Также Redis стал ключевым элементом системы уведомлений. При регистрации нового пользователя система публикует уведомление в Redis (в виде строки или элемента списка), а административная панель считывает эти данные и отображает их в интерфейсе администратора. Это дало возможность реализовать простой и эффективный механизм оповещений между изолированными компонентами системы без необходимости создания отдельного сервера очередей сообщений, вроде RabbitMQ или Kafka. Redis выступил здесь как лёгкий и быстрый брокер сообщений.

Таким образом, после внедрения Redis скорость работы системы должна была существенно возрасти, особенно в части доступа к неизменяющимся или редко изменяющимся данным. Уменьшилась нагрузка на базу данных, повысилась отзывчивость пользовательского интерфейса, появилась возможность быстро реагировать на события, происходящие в других частях приложения. Redis позволил реализовать более гибкую и масштабируемую архитектуру, сохранив простоту разработки и минимизировав изменения в существующем коде. В результате система стала быстрее, надёжнее и удобнее в сопровождении.

**Pub/Sub**

Pub/Sub (Publish/Subscribe — «публикация/подписка») — это модель обмена сообщениями, при которой отправители сообщений (паблишеры) не взаимодействуют напрямую с получателями (сабскрайберами). Вместо этого они публикуют сообщения в некие каналы, а подписчики получают сообщения, которые приходят в те каналы, на которые они подписаны. Такая модель идеально подходит для систем, где важна асинхронность, расширяемость и слабая связанность компонентов. Redis предоставляет встроенный механизм Pub/Sub, который позволяет использовать его как простой, но эффективный брокер сообщений.

В контексте нашей системы Pub/Sub через Redis использовался для реализации функциональности оповещения администратора о регистрации новых клиентов. Когда пользователь завершает регистрацию, сервер вызывает функцию публикации сообщения в определённый канал Redis (в нашем случае, это канал new\_client\_channel). Это сообщение содержит данные о новом пользователе — например, его имя и ID, сериализованные в формат JSON. Публикация занимает миллисекунды и немедленно доступна для всех подписчиков канала.

Административный интерфейс в Streamlit работает как подписчик: при запуске он подписывается на этот канал Redis и ожидает поступления новых сообщений. Когда такое сообщение приходит, вызывается callback-функция, которая добавляет соответствующее уведомление в состояние интерфейса Streamlit. Благодаря этому администратор может почти в реальном времени увидеть уведомление о новом пользователе, не запрашивая вручную обновления или не перезагружая страницу (при использовании автообновления, как st\_autorefresh, это ощущается как «живой» поток данных).

Преимущество Redis Pub/Sub состоит в его простоте: не требуется ни внешней инфраструктуры, ни сложной настройки. Redis уже работает как in-memory база, и его Pub/Sub — это встроенная возможность, которую можно использовать практически сразу. Это идеально подходит для относительно небольших, но интерактивных приложений, где необходимо быстро передавать сообщения между изолированными процессами или сервисами, например между пользовательским интерфейсом и серверной логикой.

Однако важно отметить, что Redis Pub/Sub имеет и ограничения: сообщения не сохраняются, и если подписчик не подключён в момент публикации, он это сообщение пропускает. Поэтому такая система хороша для «живых» уведомлений, но не подходит, если необходима надёжная доставка или повторное воспроизведение сообщений. В более сложных системах для этого применяются очереди сообщений с гарантией доставки, например RabbitMQ или Kafka. Тем не менее, для нашей задачи, где важна скорость и простота, Pub/Sub в Redis оказался идеальным решением.

Таким образом, Redis Pub/Sub позволил эффективно внедрить механизм событийной передачи информации в нашей системе — быстро, просто и без избыточных зависимостей. Это повысило интерактивность и отзывчивость интерфейса администратора, а также продемонстрировало, как можно строить реактивные системы с помощью минимального набора инструментов.

Ход работы

В начале я написал новый файл, который отвечает за подключение к Redis:

import redis

# Клиент для текстовых данных (авторизация и сессии)

redis\_client\_text = redis.Redis(

    host='localhost',

    port=6379,

    db=0,

    decode\_responses=True

)

# Клиент для бинарных данных (pickle)

redis\_client\_bin = redis.Redis(

    host='localhost',

    port=6379,

    db=0,

    decode\_responses=False

)

После этого я изменил файл, отвечающий за авторизацию. Теперь вход пользователей осуществляется c помощью токенов, которые хранятся в Redis и действуют один час. Для этого я добавил нижепредставленный изменения в функцию login\_user():

import uuid

from redis\_client import redis\_client\_text as redis\_client

...

if result and bcrypt.checkpw(password.encode('utf-8'), result[1].encode('utf-8')):

    st.success("Успешный вход!")

    # Генерация токена

    token = str(uuid.uuid4())

    # Храним токен в Redis с TTL 1 час

    redis\_client.redis\_client.setex(f"token:{token}", 3600, result[0])

    redis\_client.redis\_client.hset(f"user:{result[0]}", mapping={

        "user\_type": result[2],

        "username": username

    })

    # Сохраняем токен в session\_state

    st.session\_state.authenticated = True

    st.session\_state.token = token

    st.session\_state.user\_id = result[0]

    st.session\_state.username = username

    st.session\_state.user\_type = result[2]

    st.rerun()

Так же пришлось изменить главный запускающий файл app.py:

# app.py

import streamlit as st

from authentication import login\_user, register\_user

from client\_dashboard import display\_client\_dashboard

from admin\_dashboard import display\_admin\_dashboard

from redis\_client import redis\_client\_text as redis\_client

def main():

    st.title("Библиотека настольных игр")

    if "authenticated" not in st.session\_state:

        st.session\_state.authenticated = False

        st.session\_state.token = None

        st.session\_state.user\_id = None

        st.session\_state.username = None

        st.session\_state.user\_type = None

    # Проверка токена в Redis

    if st.session\_state.token:

        user\_id = redis\_client.get(f"token:{st.session\_state.token}")

        if user\_id:

            # decode\_responses=True, значит user\_id уже строка, декодировать не надо

            user\_data = redis\_client.hgetall(f"user:{user\_id}")

            # user\_data — словарь строк, поэтому просто берем ключи

            st.session\_state.authenticated = True

            st.session\_state.user\_id = user\_id

            st.session\_state.username = user\_data.get("username")

            st.session\_state.user\_type = user\_data.get("user\_type")

        else:

            st.session\_state.authenticated = False

            st.session\_state.token = None

    if not st.session\_state.authenticated:

        st.subheader("Добро пожаловать!")

        choice = st.selectbox("Выберите действие", ["Вход", "Регистрация"])

        if choice == "Вход":

            login\_user()

        elif choice == "Регистрация":

            register\_user()

    else:

        if st.session\_state.user\_type == "client":

            display\_client\_dashboard(st.session\_state.username)

        elif st.session\_state.user\_type == "admin":

            display\_admin\_dashboard()

        else:

            st.warning("Неизвестный тип пользователя")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

Теперь авторизация работает так:

 Пользователь вводит логин и пароль

 Мы проверяем их по базе данных

 Генерируем token и:

* сохраняем token:{uuid} → user\_id в Redis
* сохраняем user:{user\_id} → информация о пользователе (в Redis как хэш)
* ставим TTL 3600 секунд (1 час)

 Сохраняем token в st.session\_state

 При каждом запуске/обновлении app.py:

* мы проверяем токен в Redis
* получаем оттуда user\_id и user info
* восстанавливаем сессию

После этого я так же реализовал удаление токена при выходе из приложения в файлах admin\_dashboard.py и client\_dashboard.py:

elif menu == "Выход":

    if st.session\_state.token:

        redis\_client.delete(f"token:{st.session\_state.token}")  # Удаляем токен из Redis

    st.session\_state.authenticated = False

    st.session\_state.user\_type = None

    st.session\_state.username = None

    st.session\_state.user\_id = None

    st.session\_state.token = None

    st.rerun()

После этого я реализовал кеширование с помощью Redis в различных файлах. Идея в том, что к некоторым данным мы часто обращаемся, но при этом они редко изменяются, поэтому в подобным случае нам гораздо выгоднее хранить их в оперативной памяти

Руководствуясь этой логикой я реализовал кеширование в нескольких файлах своего приложения. Вот пример измененного game\_list.py:

import streamlit as st

from database import get\_connection

from redis\_client import redis\_client\_bin as redis\_client

import pickle

def display\_game\_list():

    try:

        cache\_key = "games\_list"

        cached\_data = redis\_client.get(cache\_key)

        if cached\_data is not None:

            try:

                games\_data = pickle.loads(cached\_data)

            except pickle.UnpicklingError:

                games\_data = None

        else:

            games\_data = None

        if games\_data is None:

            with get\_connection() as conn:

                with conn.cursor() as cur:

                    cur.execute("""

                        SELECT g.game\_id, g.game\_name, g.release\_year, g.play\_time,

                               g.min\_players, g.max\_players, g.age\_rating,

                               array\_agg(DISTINCT t.name) AS tags,

                               array\_agg(DISTINCT t.description) AS tag\_descriptions

                        FROM games g

                        JOIN games\_to\_tags gt ON g.game\_id = gt.game\_id

                        JOIN tags t ON gt.tag\_id = t.tag\_id

                        GROUP BY g.game\_id

                    """)

                    games\_data = cur.fetchall()

                    redis\_client.setex(cache\_key, 300, pickle.dumps(games\_data))

        if games\_data:

            col1, col2 = st.columns([3, 1])

            with col1:

                st.write("### Список игр")

                for game in games\_data:

                    game\_id, game\_name, release\_year, play\_time, min\_players, max\_players, age\_rating, tags, tag\_descriptions = game

                    st.write(f"\*\*{game\_name}\*\* ({release\_year})")

                    st.write(f"- \*\*Игроки:\*\* от {min\_players} до {max\_players}")

                    st.write(f"- \*\*Время игры:\*\* {play\_time} мин")

                    st.write(f"- \*\*Возрастной рейтинг:\*\* {age\_rating}+")

                    st.write(f"- \*\*Теги:\*\* {', '.join(tags)}")

                    st.write("---")

            with col2:

                st.write("### Описание тегов")

                displayed\_tags = set()

                for game in games\_data:

                    for tag\_name, tag\_description in zip(game[7], game[8]):

                        if tag\_name not in displayed\_tags:

                            st.write(f"\*\*{tag\_name}\*\*: {tag\_description}")

                            displayed\_tags.add(tag\_name)

        else:

            st.write("Нет доступных игр.")

    except Exception as e:

        st.error(f"Ошибка при получении списка игр: {e}")

А вот измененный rented\_games.py:

import streamlit as st

from database import get\_connection

from redis\_client import redis\_client\_bin as redis\_client

import pickle

def display\_rented\_games(username):

    try:

        cache\_key = f"rented\_games:{username}"

        cached\_data = redis\_client.get(cache\_key)

        if cached\_data is not None:

            try:

                rented\_games = pickle.loads(cached\_data)

            except pickle.UnpicklingError:

                rented\_games = None

        else:

            rented\_games = None

        if rented\_games is None:

            with get\_connection() as conn:

                with conn.cursor() as cur:

                    cur.execute("""

                        SELECT cl.client\_id

                        FROM clients cl

                        JOIN users u ON cl.user\_id = u.user\_id

                        WHERE u.username = %s

                    """, (username,))

                    client\_data = cur.fetchone()

                    if not client\_data:

                        st.error("Не удалось найти клиента, связанного с этим пользователем.")

                        return

                    client\_id = client\_data[0]

                    cur.execute("""

                        SELECT g.game\_name, r.start\_date, r.end\_date, l.location\_name, l.location\_address

                        FROM rentals r

                        JOIN copies c ON r.copy\_id = c.copy\_id

                        JOIN games g ON c.game\_id = g.game\_id

                        JOIN locations l ON c.location\_id = l.location\_id

                        WHERE r.client\_id = %s

                        ORDER BY r.start\_date DESC

                    """, (client\_id,))

                    rented\_games = cur.fetchall()

                    redis\_client.setex(cache\_key, 120, pickle.dumps(rented\_games))

        if rented\_games:

            st.write("### Ваши арендованные игры:")

            for game in rented\_games:

                game\_name, start\_date, end\_date, location\_name, location\_address = game

                st.write(f"\*\*{game\_name}\*\*")

                st.write(f"- \*\*Дата начала аренды:\*\* {start\_date.strftime('%d-%m-%Y')}")

                st.write(f"- \*\*Дата окончания аренды:\*\* {end\_date.strftime('%d-%m-%Y')}")

                st.write(f"- \*\*Локация:\*\* {location\_name}")

                st.write(f"- \*\*Адрес локации:\*\* {location\_address}")

                st.write("---")

        else:

            st.write("У вас нет арендованных игр.")

    except Exception as e:

        st.error(f"Ошибка при получении списка арендованных игр: {e}")

Так же кеширование было добавлено в нескольких других файлах.

Реализацию pubsub я решил продемонстрировать через показ уведомлений админу при регистрации нового пользователя. Для начала я написал файл notifier.py, который «централизует» всю логику pubsub, чтобы использовать ее в разных частях проекта. Вот непосредственно сам код данного файла:

import threading

import redis

import json

from redis\_client import redis\_client\_text as redis\_client

CHANNEL\_NEW\_CLIENT = "new\_client\_channel"

def publish\_new\_client(username, user\_id):

    message = json.dumps({"username": username, "user\_id": user\_id})

    redis\_client.publish(CHANNEL\_NEW\_CLIENT, message)

def subscribe\_to\_new\_clients(callback):

    pubsub = redis\_client.pubsub()

    pubsub.subscribe(CHANNEL\_NEW\_CLIENT)

    def listen():

        for message in pubsub.listen():

            if message['type'] == 'message':

                data = json.loads(message['data'])

                callback(data)

    thread = threading.Thread(target=listen, daemon=True)

    thread.start()

После этого вставляем в функции, отвечающий за авторизацию следующий код:

from notifier import publish\_new\_client

st.success("Регистрация завершена! Теперь вы можете войти.")

# Вставляем данный код, ниже строчки об успешной регистрации

publish\_new\_client(username, user\_id)

После этого добавляем изменения в amdin\_dashboard.py:

import streamlit as st

from add\_game import display\_add\_game\_form

from delete\_game import delete\_game

from add\_copies import add\_game\_copies

from game\_list import display\_game\_list

from admin\_rented\_games import display\_admin\_rented\_games

from show\_copies\_count import display\_copies\_count

from display\_clients import display\_clients\_list

from redis\_client import redis\_client\_text as redis\_client

from notifier import subscribe\_to\_new\_clients

def display\_admin\_dashboard():

    st.sidebar.title("Панель администратора")

    # Инициализация списка уведомлений в сессии

    if "new\_client\_notifications" not in st.session\_state:

        st.session\_state.new\_client\_notifications = []

    # Колбек для добавления уведомления о новом клиенте

    def new\_client\_callback(data):

        notification = f"Новый клиент зарегистрирован: {data['username']} (ID: {data['user\_id']})"

        if notification not in st.session\_state.new\_client\_notifications:

            st.session\_state.new\_client\_notifications.append(notification)

            # Автообновление страницы из другого потока не сработает,

            # но уведомления покажутся при следующем обновлении страницы

    # Запускаем подписку на уведомления один раз

    if "subscriber\_started" not in st.session\_state:

        subscribe\_to\_new\_clients(new\_client\_callback)

        st.session\_state.subscriber\_started = True

    # Выводим уведомления в сайдбар

    if st.session\_state.new\_client\_notifications:

        st.sidebar.markdown("### Новые регистрации клиентов:")

        for note in st.session\_state.new\_client\_notifications:

            st.sidebar.info(note)

    menu = st.sidebar.radio("Выберите действие:",

                             ["Главная", "Добавить копии игры", "Добавить новую игру", "Показать количество копий",

                              "Удалить игру", "Список игр", "Список пользователей", "Список арендованных настольных игр",

                               "Выход"], index=0)

    if menu == "Главная":

        st.write("Добро пожаловать в панель администратора! Выберите действие в меню слева.")

    elif menu == "Добавить копии игры":

        add\_game\_copies()

    elif menu == "Добавить новую игру":

        display\_add\_game\_form()

    elif menu == "Удалить игру":

        delete\_game()

    elif menu == "Список игр":

        display\_game\_list()

    elif menu == "Список арендованных настольных игр":

        display\_admin\_rented\_games()

    elif menu == "Показать количество копий":

        display\_copies\_count()

    elif menu == "Список пользователей":

        display\_clients\_list()

    elif menu == "Выход":

        # Удаляем токен из Redis

        if st.session\_state.token:

            redis\_client.delete(f"token:{st.session\_state.token}")

        # Сбрасываем session\_state

        st.session\_state.authenticated = False

        st.session\_state.token = None

        st.session\_state.user\_type = None

        st.session\_state.username = None

        st.session\_state.user\_id = None

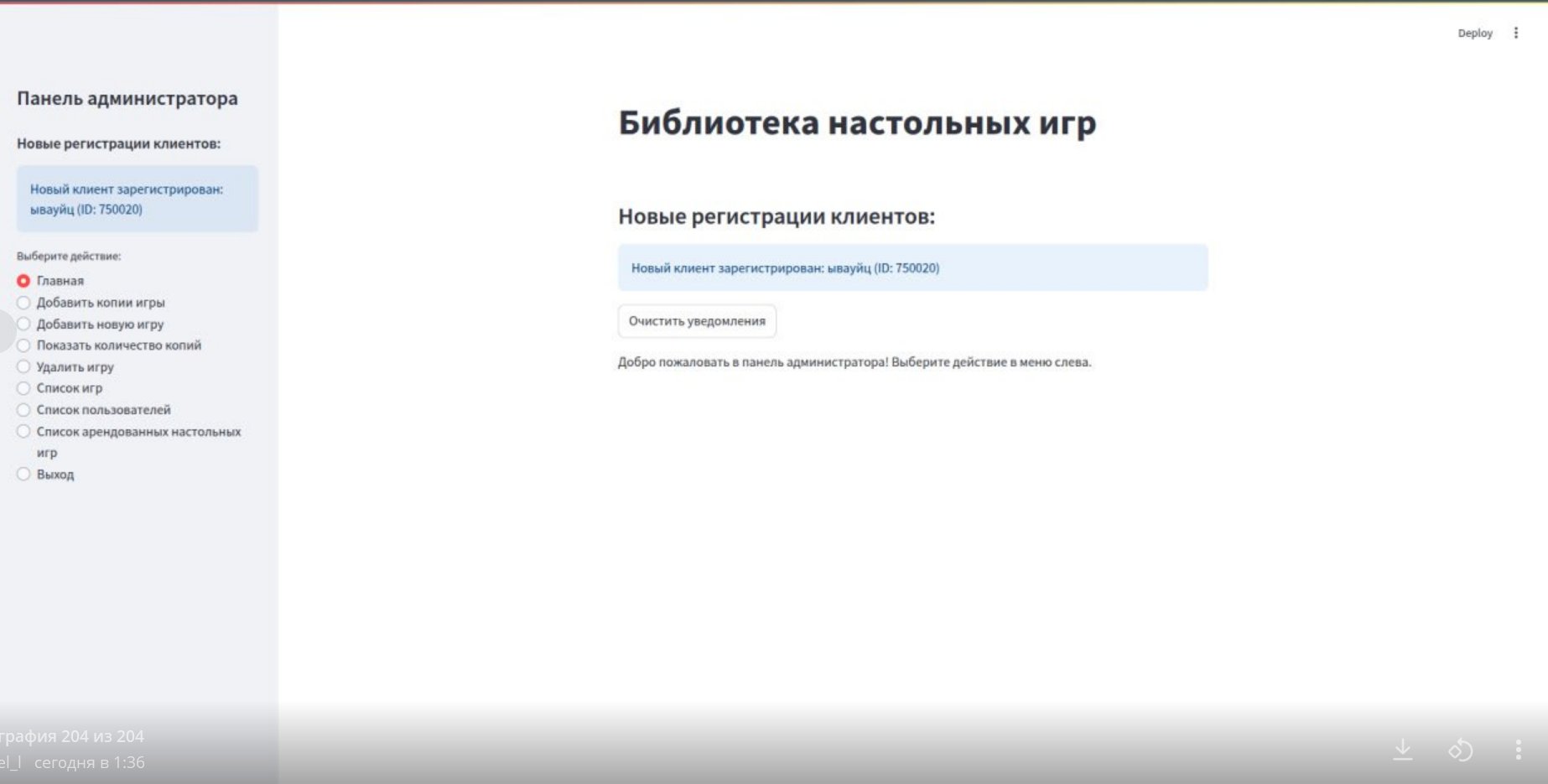
        st.experimental\_rerun()

        for note in st.session\_state.new\_client\_notifications:

            st.sidebar.info(note)

    # Далее остальной мой код

Пример уведомления:



**Вывод**

В ходе выполнения лабораторной работы я освоил использование Redis как средства оптимизации обработки динамических данных в рамках курсового проекта. Основное внимание было уделено интеграции Redis для хранения токенов авторизации, сессионных данных, временных и часто запрашиваемых сведений, а также для организации мгновенных уведомлений о критичных событиях в системе. Я научился применять механизм TTL для установки времени жизни токенов, что позволило автоматически очищать устаревшие записи и повысить безопасность и эффективность работы с авторизацией. Были реализованы схемы хранения данных в Redis, учитывающие характер их использования — от строк и списков до словарей и каналов публикации-подписки. Особенно важным стало освоение технологии Pub/Sub, с помощью которой были настроены оповещения об обновлениях в системе в реальном времени, в частности, уведомления для администратора о регистрации новых клиентов. В результате использования Redis существенно повысилась скорость доступа к критически важной информации, снизилась нагрузка на основной сервер и обеспечена высокая отзывчивость интерфейса. Эта лабораторная работа позволила мне на практике освоить архитектурные решения для масштабируемых и отзывчивых веб-сервисов, применяя Redis как ключевой компонент быстрой обработки данных.