

PRZEDMIOT: APLIKACJE INTERNETOWE W DJANGO

Dokumentacja Projektowa

Discord Clone: Aplikacja do komunikacji w czasie rzeczywistym

24 maja 2026

Spis treści

1	Wstęp i cel projektu	2
1.1	Kluczowe funkcjonalności	2
2	Architektura systemu i wykorzystane technologie	3
2.1	Warstwa Prezentacji (Klient / Frontend)	3
2.2	Warstwa Aplikacji (Backend)	3
2.3	Warstwa Danych	3
3	Środowisko i instalacja	4
3.1	Wymagania systemowe	4
3.2	Kroki instalacji	4
4	Struktura repozytorium	5
5	Kluczowe fragmenty implementacji	6
5.1	Consumer obsługujący WebSocket (chat/consumers.py)	6
5.2	Generowanie tokenów TURN (chat/views.py)	7
5.3	Sygnalizacja WebRTC (chat/consumers.py)	7
6	Prezentacja interfejsu graficznego	9
6.1	Dostęp i Autoryzacja	9
6.2	Główny panel komunikacyjny	10
7	Podsumowanie	11

1 Wstęp i cel projektu

Prezentowany projekt „Discord Clone” to nowoczesna aplikacja webowa stworzona w celu dostarczenia funkcjonalności umożliwiających wieloosobową komunikację tekstową w czasie rzeczywistym, analogiczną do popularnej platformy Discord.

Głównym celem edukacyjnym i technologicznym projektu jest zintegrowanie klasycznego frameworka opartego na protokole HTTP z zaawansowanymi mechanizmami asynchronicznymi i protokołem **WebSocket**, z zachowaniem optymalnej wydajności i uproszczonej infrastruktury.

1.1 Kluczowe funkcjonalności

- **Autoryzacja:** Rejestracja i logowanie. Dostęp do głównej funkcjonalności jest chroniony.
- **Zarządzanie Kanałami:** Użytkownicy z odpowiednimi uprawnieniami mogą tworzyć tematyczne kanały grupowe (tekstowe i głosowe).
- **Role i uprawnienia:** Podział na Administratorów, Moderatorów oraz Użytkowników z dedykowanymi przywilejami (np. usuwanie obcych wiadomości przez moderatora).
- **Czat Real-Time:** Natychmiastowe przesyłanie wiadomości bez konieczności odświeżania okna przeglądarki.
- **Kanały Głosowe (Voice Channels):** Wieloużytkownikowe rozmowy audio w czasie rzeczywistym realizowane w architekturze peer-to-peer (P2P) z wykorzystaniem WebRTC, wspierane przez globalną usługę Cloudflare Realtime TURN do NAT-traversalu.

2 Architektura systemu i wykorzystane technologie

Aplikacja została zaprojektowana w oparciu o architekturę trójwarstwową, w której eliminowano konieczność stosowania osobnego serwera kolejkowego (np. Redis) na rzecz wbudowanych mechanizmów bazy PostgreSQL.

2.1 Warstwa Prezentacji (Klient / Frontend)

Użytkownik komunikuje się z systemem za pośrednictwem przeglądarki internetowej. Warstwa ta renderowana jest przy pomocy **szablonów HTML/CSS (Bootstrap)**. Lekki skrypt **JavaScript** nawiązuje i utrzymuje ciągłe połączenie WebSocket z serwerem, asynchronicznie odbierając powiadomienia o nowych wiadomościach. W przypadku połączeń głosowych, frontend odpowiada za przechwytywanie dźwięku z mikrofonu (Web Audio API), realizację nawiązania połączeń WebRTC P2P (Full Mesh) oraz dynamiczną obsługę odtwarzania strumieni audio od innych użytkowników bez przerywania nawigacji dzięki technologii AJAX/Fetch (SPA Shell).

2.2 Warstwa Aplikacji (Backend)

Główny trzon projektu oparty jest o język **Python** oraz framework **Django 4.2**. Ruch sieciowy jest zarządzany przez serwer **Daphne (ASGI)**, który:

- **Zwykłe żądania HTTP** (np. logowanie, ładowanie widoków) przekazuje do synchronicznych widoków Django.
- **Żądania WebSocket** routuje do asynchronicznych consumerów zdefiniowanych w **Django Channels**.

W kontekście kanałów głosowych backend pełni funkcję serwera sygnalizacyjnego (signaling server) za pośrednictwem protokołu WebSocket, pośrednicząc w wymianie ofert SDP (Session Description Protocol) i kandydatów ICE między klientami. Dodatkowo, backend komunikuje się z interfejsem API Cloudflare, pobierając krótkotrwałe, bezpieczne dane uwierzytelniające dla serwerów TURN (ważne przez 24 godziny).

2.3 Warstwa Danych

Zastosowano zunifikowaną bazę danych **PostgreSQL**. Standardowe dane (użytkownicy, historia wiadomości) są obsługiwane przez mechanizm Django ORM. Dodatkowo, dzięki zastosowaniu pluginu **channels-postgres**, architektura w czasie rzeczywistym (pub/sub) wykorzystuje natywne zapytania LISTEN/NOTIFY z PostgreSQL, co pozwala rozgłaszać eventy WebSockets pomiędzy klientami. W bazie danych PostgreSQL przechowywany jest również stan aktywności użytkowników w kanałach głosowych za pomocą pola **current_voice_channel** w profilu użytkownika (**UserProfile**). Umożliwia to natychmiastowe renderowanie listy uczestników kanału głosowego bezpośrednio pod jego nazwą na liście kanałów podczas ładowania strony.

3 Środowisko i instalacja

Do uruchomienia projektu w celach deweloperskich i testowych zastosowano konteneryzację, co eliminuje problem zależności systemowych.

3.1 Wymagania systemowe

- **Docker** (Docker Engine)
- **Docker Compose**
- Połączenie z internetem (do pobrania obrazów oraz weryfikacji API Cloudflare)

3.2 Kroki instalacji

1. **Pobranie kodu:** Sklonuj lub pobierz i rozpakuj repozytorium projektu w wybranym folderze.
2. **Konfiguracja zmiennych środowiskowych:** Utwórz plik `.env` w katalogu głównym projektu na wzór udostępnionych szablonów. Wprowadź tam wygenerowane w panelu Cloudflare dane uwierzytelniające dla usługi Cloudflare Realtime TURN: `CLOUDFLARE_ACCOUNT_ID`, `CLOUDFLARE_API_TOKEN` oraz `CLOUDFLARE_TURN_KEY_ID`. Umożliwi to serwerowi dynamiczne generowanie tokenów ICE dla przeglądarek.
3. **Budowanie kontenerów:** Uruchom terminal w głównym katalogu (tam, gdzie znajduje się plik `docker-compose.yml`) i wykonaj polecenie:

```
docker-compose up --build
```
4. **Uruchomienie bazy:** Kontenery wykonają automatycznie proces migracji struktury danych (`manage.py migrate`). Zostanie także wygenerowane konto administratora.
5. Aplikacja docelowo jest dostępna pod adresem: <http://localhost:8000>.

Domyślne dane dostępowe administratora:

- Login: admin
- Hasło: admin

4 Struktura repozytorium

Projekt zorganizowany jest wokół aplikacji modułowych typowych dla frameworka Django. Poniższe drzewo prezentuje kluczowe pliki z punktu widzenia architektonicznego.

```
/ (Katalog główny projektu)
├── core/
│   ├── settings.py .....Konfiguracja główna (baza
│   │                       danych, channels)
│   ├── asgi.py .....Punkt wejścia dla serwera
│   │                       Daphne i routing Channels
│   └── urls.py .....Główny routing HTTP
├── chat/
│   ├── models.py .....Definicje tabel DB
│   │                       (Użytkownicy, Kanały,
│   │                       Wiadomości)
│   ├── consumers.py .....Asynchroniczna obsługa
│   │                       WebSocketów (czat,
│   │                       sygnalizacja WebRTC)
│   ├── routing.py .....Definicje ścieżek URL dla
│   │                       połączeń ws://
│   └── views.py .....Widoki synchroniczne oraz
│   │                       endpoint generowania
│   │                       tokenów TURN
├── docker-compose.yml .....Orkiestracja kontenerów
│   │                       (PostgreSQL + aplikacja)
├── Dockerfile .....Instrukcja budowania
│   │                       środowiska bazowego Python
└── requirements.txt .....Zależności (Django,
    channels-postgres)
```

5 Kluczowe fragmenty implementacji

W tej sekcji przedstawiono wybrane fragmenty kodu źródłowego odpowiadające za najważniejsze mechanizmy realizujące komunikację w czasie rzeczywistym.

5.1 Consumer obsługujący WebSocket (chat/consumers.py)

Poniższy kod ukazuje w jaki sposób framework Channels obsługuje podłączanie się użytkowników do specyficznej dla kanału „grupy” oraz metodykę odbierania wiadomości tekstowych z przeglądarki i rozsyłania ich w czasie rzeczywistym.

```
1 import json
2 from channels.generic.websocket import AsyncWebsocketConsumer
3
4 class ChatConsumer(AsyncWebsocketConsumer):
5     async def connect(self):
6         self.room_name = self.scope['url_route']['kwargs']['room_name']
7         self.room_group_name = f"chat_{self.room_name}"
8
9         # Dodanie klienta do grupy nasluchujacej
10        await self.channel_layer.group_add(
11            self.room_group_name,
12            self.channel_name
13        )
14        await self.accept()
15
16    async def disconnect(self, close_code):
17        # Odłączenie od grupy przy zamknięciu strony
18        await self.channel_layer.group_discard(
19            self.room_group_name,
20            self.channel_name
21        )
22
23    async def receive(self, text_data):
24        text_data_json = json.loads(text_data)
25        message = text_data_json['message']
26
27        # Rozesłanie otrzymanej wiadomości do całej grupy
28        await self.channel_layer.group_send(
29            self.room_group_name,
30            {
31                'type': 'chat_message',
32                'message': message
33            }
34        )
35
36    # Funkcja wywoływana, gdy warstwa kanałów wysyła typ 'chat_message'
37    async def chat_message(self, event):
38        message = event['message']
39        await self.send(text_data=json.dumps({
40            'message': message
41        })))
```

Listing 1: Przykład implementacji klasy AsyncWebsocketConsumer

5.2 Generowanie tokenów TURN (chat/views.py)

Poniższy kod prezentuje bezpieczne pobieranie danych uwierzytelniających (tokenów ICE) z API Cloudflare TURN za pomocą synchronicznego widoku Django. W przypadku braku konfiguracji kluczy Cloudflare zastosowano automatyczną ścieżkę awaryjną (fallback) do publicznych serwerów STUN.

```
1 import urllib.request
2 import urllib.error
3 import json
4 from django.conf import settings
5 from django.http import JsonResponse
6 from django.contrib.auth.decorators import login_required
7
8 @login_required
9 def voice_credentials_view(request):
10     api_token = getattr(settings, 'CLOUDFLARE_API_TOKEN', '')
11     turn_key_id = getattr(settings, 'CLOUDFLARE_TURN_KEY_ID', '')
12
13     if not api_token or not turn_key_id:
14         return JsonResponse({
15             "iceServers": [{"urls": ["stun:stun.cloudflare.com:3478"]}]}
16         )
17
18     url = (
19         f"https://rtc.live.cloudflare.com/v1/turn/keys/"
20         f"{turn_key_id}/credentials/generate-ice-servers"
21     )
22     headers = {
23         "Authorization": f"Bearer {api_token}",
24         "Content-Type": "application/json"
25     }
26     body = json.dumps({"ttl": 86400}).encode("utf-8")
27     req = urllib.request.Request(url, data=body, headers=headers, method="POST")
28     try:
29         with urllib.request.urlopen(req, timeout=5) as response:
30             return JsonResponse(json.loads(response.read().decode()))
31     except urllib.error.URLError:
32         return JsonResponse({
33             "iceServers": [{"urls": ["stun:stun.cloudflare.com:3478"]}]}
34         )
```

Listing 2: Pobieranie tokenów ICE Cloudflare Realtime TURN

5.3 Sygnalizacja WebRTC (chat/consumers.py)

Aby nawiązać połączenie P2P, klienci muszą wymienić oferty SDP i kandydatów ICE. Poniższy fragment klasy `VoiceSignalingConsumer` realizuje to zadanie, automatycznie aktualizując status obecności użytkownika w kanale głosowym w bazie danych oraz rozsyłając sygnały do właściwych odbiorców docelowych.

```
1 from channels.generic.websocket import AsyncWebsocketConsumer
2
3 class VoiceSignalingConsumer(AsyncWebsocketConsumer):
4     async def connect(self):
5         self.channel_id = self.scope['url_route']['kwargs']['channel_id']
6         self.group_name = f'voice_{self.channel_id}'
```



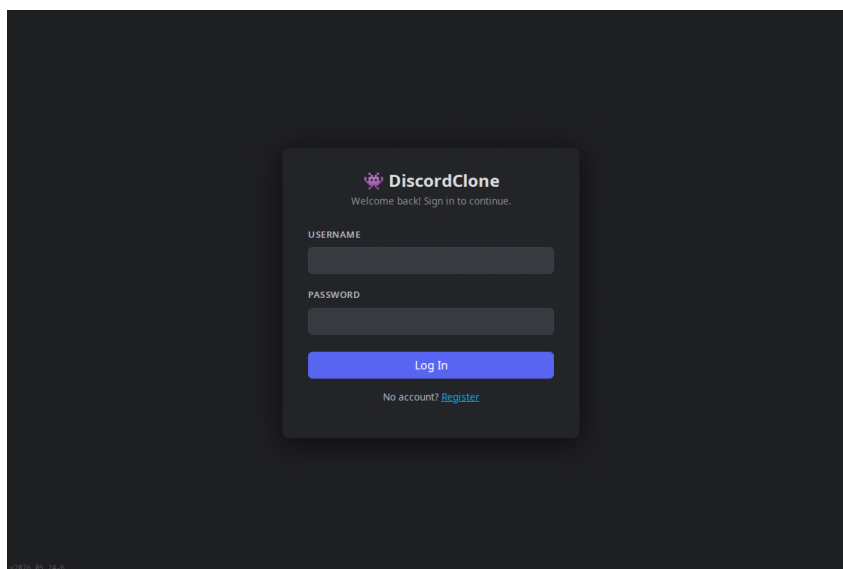
```
7     self.user = self.scope['user']
8
9     await self.channel_layer.group_add(self.group_name, self.channel_name)
10    await self.accept()
11
12    await self.set_voice_channel(self.channel_id)
13    await self.channel_layer.group_send(
14        'global_updates',
15        {
16            'type': 'voice_presence_change',
17            'user_id': self.user.id,
18            'username': self.user.username,
19            'channel_id': int(self.channel_id),
20            'action': 'joined',
21        }
22    )
23
24    async def receive(self, text_data):
25        data = json.loads(text_data)
26        if data.get('action') == 'signal':
27            await self.channel_layer.group_send(
28                self.group_name,
29                {
30                    'type': 'voice_signal_relay',
31                    'sender_id': self.user.id,
32                    'target_id': data.get('target_id'),
33                    'data': data.get('data'),
34                }
35            )
36
37    async def voice_signal_relay(self, event):
38        if event['target_id'] == self.user.id:
39            await self.send(text_data=json.dumps(event))
```

Listing 3: Klasa VoiceSignalingConsumer obsługująca sygnalizację WebRTC

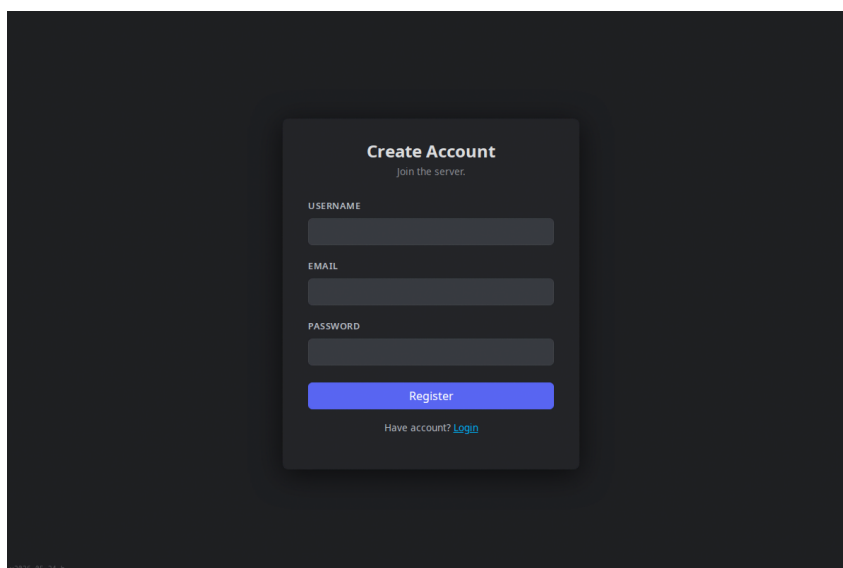
6 Prezentacja interfejsu graficznego

Kolejne ekrany prezentują wygląd aplikacji z perspektywy klienta webowego. Widoki zostały przygotowane w oparciu o framework Bootstrap, co zapewnia responsywność interfejsu na różnych urządzeniach.

6.1 Dostęp i Autoryzacja

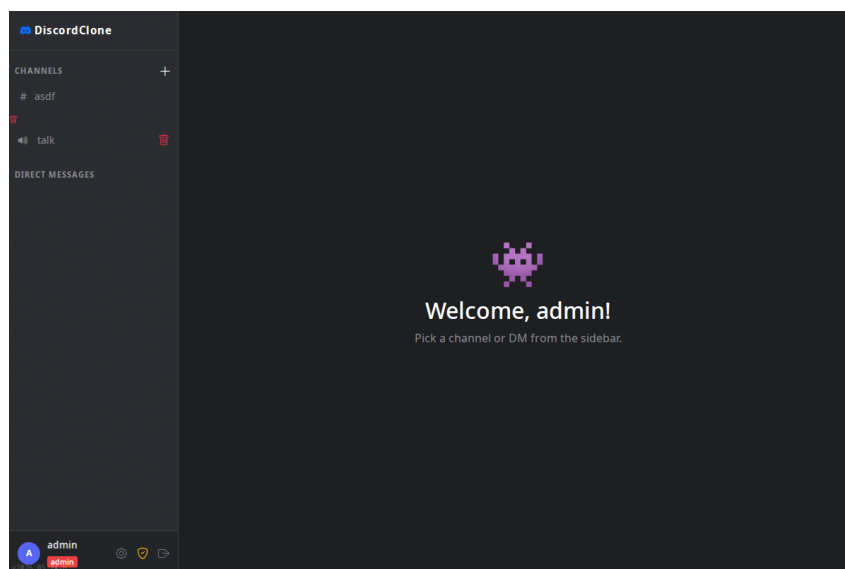


Rysunek 1: Ekran logowania – Wymagany dla wszystkich użytkowników przed przystąpieniem do konwersacji.

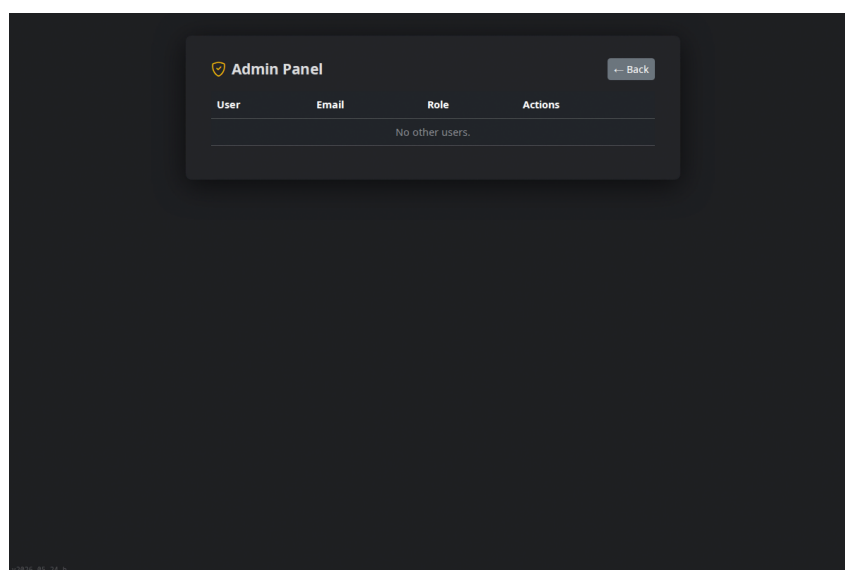


Rysunek 2: Ekran rejestracji – Umożliwia utworzenie nowego profilu.

6.2 Główny panel komunikacyjny



Rysunek 3: Główne okno aplikacji przypominające interfejs platformy Discord. Po lewej stronie widoczna jest lista dostępnych kanałów tematycznych, z prawej okno wybranego czatu z historią wiadomości.



Rysunek 4: Panel Administracyjny Upewnnień – Służy do moderowania i edycji uprawnień użytkowników.

7 Podsumowanie

Projekt Discord Clone stanowi kompletną, chociaż z konieczności uproszczoną na potrzeby ćwiczeniowe, wersję nowoczesnego komunikatora grupowego. Użycie architektury ASGI (Daphne, Django Channels) przełamuje ograniczenia tradycyjnego, synchronicznego modelu HTTP/WSGI i pozwala na obsługę setek równoczesnych połączeń w czasie rzeczywistym.

Najważniejsze osiągnięcia technologiczne zastosowane w projekcie:

1. **Integracja channels-postgres:** Rezygnacja z klasycznego wykorzystywania zewnętrznych rozwiązań cachingowych (takich jak np. Redis) pozwala znacznie uprościć infrastrukturę serwerową oraz obniżyć zasoby wykorzystywane przez kontenery Dockera. Komunikaty WebSocket skutecznie wykorzystują wewnętrzne wyzwalacze PostgreSQL.
2. **Podział uprawnień:** Prawidłowe zdefiniowanie ról wewnątrz systemu uniemożliwia użytkownikom ze standardowymi uprawnieniami modyfikowanie i kasowanie wiadomości innych użytkowników, chroniąc system przed nadużyciami.
3. **Prostota środowiska deweloperskiego:** Plik konfiguracyjny `docker-compose.yml` sprowadza proces instalacji środowiska uruchomieniowego do jednej komendy terminala, rozwiązując odwieczny problem niezgodności wersji paczek pomiędzy maszynami programistów.
4. **Komunikacja głosowa WebRTC z Cloudflare Realtime TURN:** Zaimplementowanie kanałów głosowych w modelu P2P Full Mesh, przy jednoczesnym wyeliminowaniu kosztów stałych dzięki darmowemu pakietowi Cloudflare Realtime (do 1000 GB egressu/miesiąc) i dynamicznemu generowaniu krótkoterminowych tokenów sesyjnych.

Projekt daje doskonałą podstawę do wdrażania kolejnych funkcji w przyszłości, takich jak udostępnianie ekranu (screen sharing), grupowe rozmowy wideo czy natywne powiadomienia typu Push Notifications.