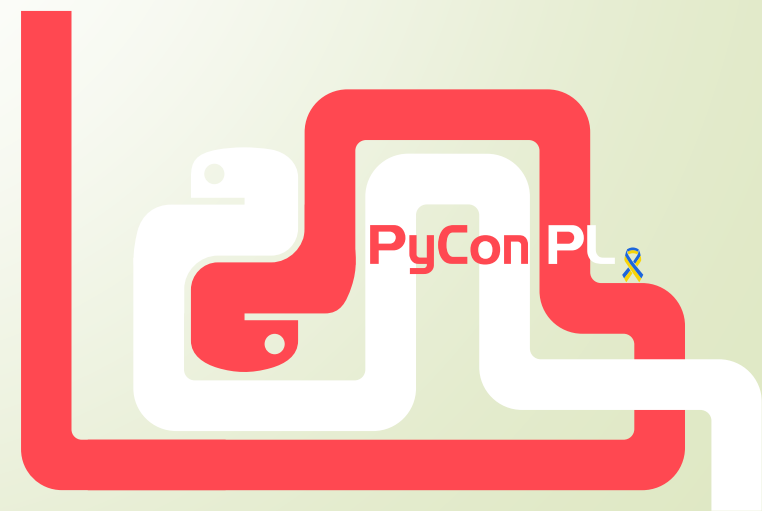




# Nie tylko WebSocket...



Artur Lew,  
PyCon PL 2025  
Gliwice 29.08





# Dzisiejsza tematyka



1. Komunikacja w aplikacjach web
2. WebSocket vs SSE
3. Push w HTTP/1.1 i HTTP/2
4. Konfiguracja serwera (Nginx)
5. Implementacja SSE w przeglądarce
6. FastAPI + SSE
7. Rozwiązania typowych problemów związanych z SSE
8. Przykładowe zastosowania SSE
9. Podsumowanie



# Komunikacja w aplikacjach web



- HTTP żądanie – odpowiedź (request-response)
- Polling
- Long Polling
- WebSocket
- Server-Sent Events (SSE)
- HTTP/2 Push



# HTTP request-response



klasyczny  
model

klient pyta -  
serwer  
odpowiada

brak inicjatywy  
serwera



# Polling

- Klient cyklicznie pyta serwer o nowe dane.
- Plusy: prostota implementacji.
- Minusy: nieefektywne, opóźnienia, obciążenie serwera.





# Long Polling



- Klient utrzymuje żądanie otwarte, serwer odpowiada, gdy pojawi się zdarzenie.
- Efektywniejsze niż polling, ale wciąż kosztowne (wiele requestów).
- Problemy z timeoutami.



# WebSockets



- Pełny duplex: klient ↔ serwer
- Protokół binarny poza HTTP
- Dodatkowa konfiguracja na serwerze (np. nagłówki dla handshake WebSocket)
- Trudniejsza obsługa autoryzacji i skalowania
- Złożoność zarządzania połączeniami (konieczność obsługi ponownego połączenia)
- Konieczność zdefiniowania albo użycia dedykowanego protokołu wyższego rzędu jak STOMP
- Biblioteki wprowadzają swoje abstrakcje które nie do końca są standardowe – np. Socket.IO (np. fallback)




# Server-Sent Events (SSE)



- Jednokierunkowa komunikacja: serwer 👉 klient
- Wbudowana implementacja dostępna w każdej przeglądarce
- Protokół tekstowy
- Kompatybilne z większością serwerów proxy i firewall
- Można wysyłać różne zdarzenia w tym samym kanale (event:)
- Prostota i niezawodność





# HTTP/2 Push

- Serwer może wysłać dodatkowe zasoby klientowi
- Wycofane ze specyfikacji



404

# WebSockets i SSE – Podobieństwa i różnice

## Podobieństwa:

- Umożliwiają komunikację w czasie rzeczywistym.
- Używają jednego otwartego połączenia TCP.

## Różnice:

- WebSocket: full-duplex, protokół niezależny od HTTP po ustanowieniu połączenia (ws://, wss://).
- SSE: jednokierunkowy (tylko serwer ➡ klient), działa na standardowym HTTP/1.1.
- SSE ma natywne wsparcie przez EventSource.

# WebSockets i SSE - Zastosowania

- WebSocket 👉 chaty, gry multiplayer, dwukierunkowa synchronizacja
- SSE 👉 powiadomienia, aktualizacje statusów, monitoring
- Zużycie zasobów:
  - SSE 👉 znikome
  - WS 👉 narzut bibliotek i dodatkowego kodu

# Rozwiązania "push" w HTTP1.1, HTTP/2 i HTTP/3

- HTTP/1.1: brak natywnego push, jedynie „triki” typu long polling czy SSE.
- HTTP/2: wprowadził Server Push (np. serwer może wysłać CSS/JS zanim przeglądarka poprosi).
- HTTP/3 (QUIC): focus na multiplexing i niższe opóźnienia, ale bez Server Push.

Wniosek:

Server Push w HTTP/2 nie przyjął się 🙅 SSE i WebSocket nadal rządzą 😁

# Konfiguracja serwera (Nginx)

- Problem: Nginx może buforować odpowiedzi 📡 SSE przestaje działać.

```
location /events {  
    proxy_pass http://backend;  
    proxy_set_header Host $host;  
    proxy_set_header X-Real-IP $remote_addr;  
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;  
    proxy_set_header X-Forwarded-Proto $scheme;  
  
    # Kluczowe dla SSE  
    proxy_buffering off;  
    proxy_cache off;  
    proxy_set_header Connection '';  
  
    # Nagłówki dla SSE  
    add_header Cache-Control no-cache;  
    add_header X-Accel-Buffering no;  
  
    # Timeouty  
    proxy_read_timeout 24h;  
    proxy_send_timeout 24h;  
}
```



## SSE w przeglądarce

```
const eventSource = new EventSource('http://localhost:8000/stream')
const output = document.getElementById('output')

eventSource.onopen = (event) => {
  output.textContent += '📄 Podłączono do serwera\n'
}

eventSource.addEventListener('push', (event) => {
  const data = JSON.parse(event.data);
  output.textContent += `💻 ${data.ip} 👉 📄 ${data.message}\n`
})

eventSource.onerror = (err) => {
  output.textContent += '⚠️ Wystąpił błąd w komunikacji\n'
}
```

# FastAPI i SSE


```
import asyncio
import json

from fastapi import FastAPI
from fastapi.responses import StreamingResponse

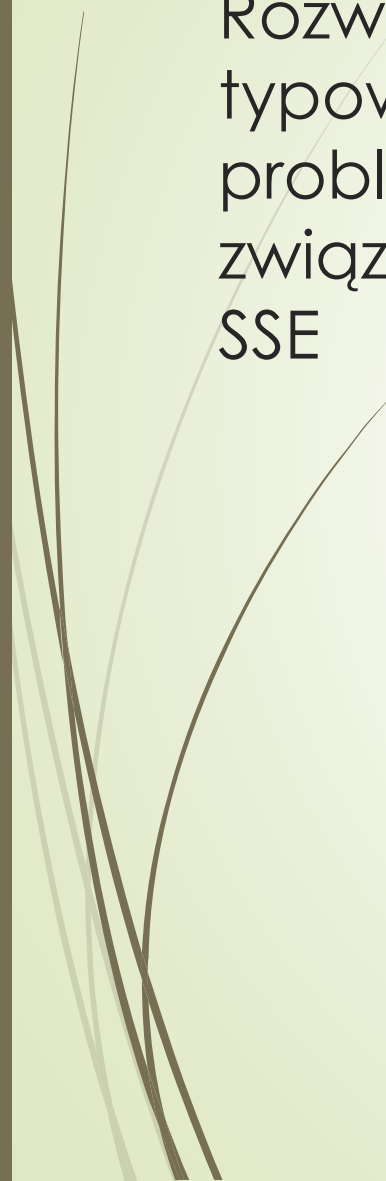
app = FastAPI()


@app.get("/events")
async def stream_events():
    async def event_generator():
        counter = 0
        while True:
            data = dict(message=f"Hello from server! Count: {counter}")
            yield f"data: {json.dumps(data)}\n\n"
            counter += 1
            await asyncio.sleep(1)

    return StreamingResponse(
        event_generator(),
        media_type="text/event-stream",
        headers={
            "Cache-Control": "no-cache",
            "Connection": "keep-alive"
        }
    )
```



## Rozwiązania typowych problemów związanych z SSE



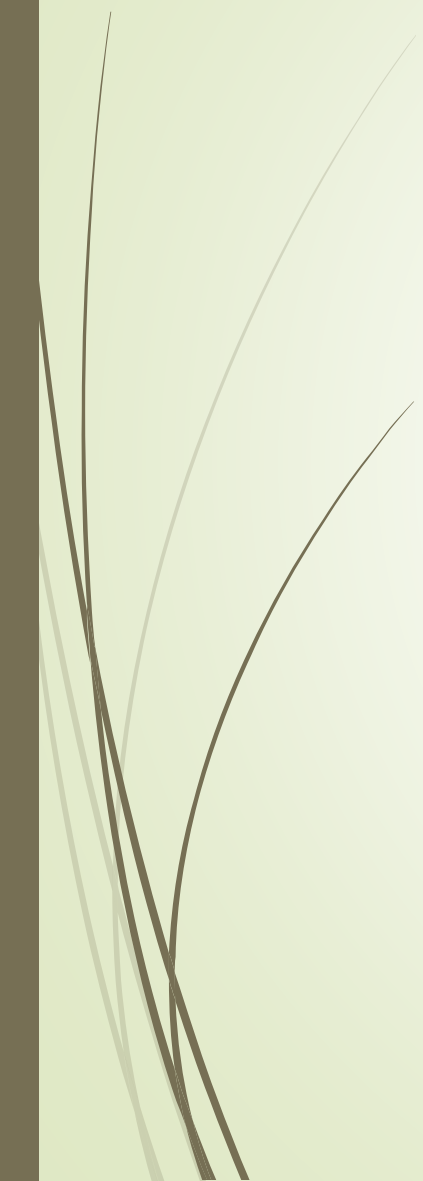
- Limit połączeń do danego adresu
  - Utrzymywanie połączenia oraz problemy z HTTPS
  - Obsługa błędów – ponowne podłączenie i obsługa zgubionych zdarzeń
  - Skalowanie aplikacji w środowiskach wieloinstancyjnych
  - Rozwiązania zabezpieczające przed zalewem zdarzeń (stampede)
- 





## Limit połączeń do danego adresu

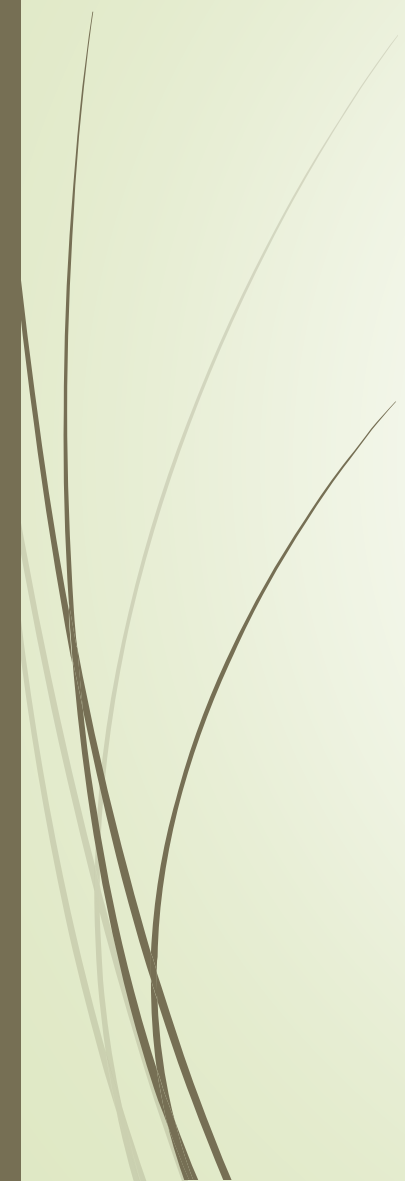


- Użycie certyfikatu SSL / przejście na HTTP/2
  - Ograniczenie ilości danych w payload
- 



## Utrzymywanie połączenia oraz problemy z HTTPS



- SSE wymaga stałego połączenia – firewalle, load balancery mogą zrywać połączenie.
  - Rozwiązanie: keep-alive, reconnect na kliencie (automatyczny), heartbeat messages.
- 

# Obsługa błędów



- Ponowne połączenie i obsługa zgubionych zdarzeń
- EventSource automatycznie ponawia połączenie
- Mechanizm Last-Event-ID pozwala na wznowienie od konkretnego zdarzenia
- Rozwiązanie: serwer musi trzymać bufor zdarzeń i odtwarzać brakujące




# Skalowanie aplikacji w środowiskach wieloinstancyjnych



- Problem: połączenia do różnych instancji 🙌 brak globalnej konsystencji.
- Rozwiązanie: Redis/Kafka jako message broker, sticky sessions albo Pub/Sub.




## Rozwiązania zabezpieczające przed zalewem zdarzeń

- Pojęcie „stampede”
  - Problem: nagła fala zdarzeń przeciąża klienta i serwer.
  - Rozwiązanie: throttling, batchowanie zdarzeń, deduplikacja, filtry per klient.
- 



## Przykładowe zastosowania SSE

- Wzorzec CQRS (Command Query Responsibility Segregation) i Event sourcing
  - Database changelog (CDC - Change Data Capture)
  - Użycie razem z Conflict-free Replicated Data Types
  - Biblioteka `FastStream`
- 



Q&A...





Dziękuję za uwagę

