Server-Sent Events for Push-Notifications on FastAPI

Alex Zhydyk · Follow

Published in Python in Plain English · 5 min read · Aug 14, 2023

51 2

Image by Ryan McGuire from Pixabay

We frequently encounter the need to update client data triggered by events on the server, in a one-way manner.

In most projects involving customer interaction, it’s crucial to promptly notify users about events such as successful order processes or receiving likes on a post.

If we conduct brief investigation into the optimal approach for implementing this feature, we will find several potential methods to implement this feature:

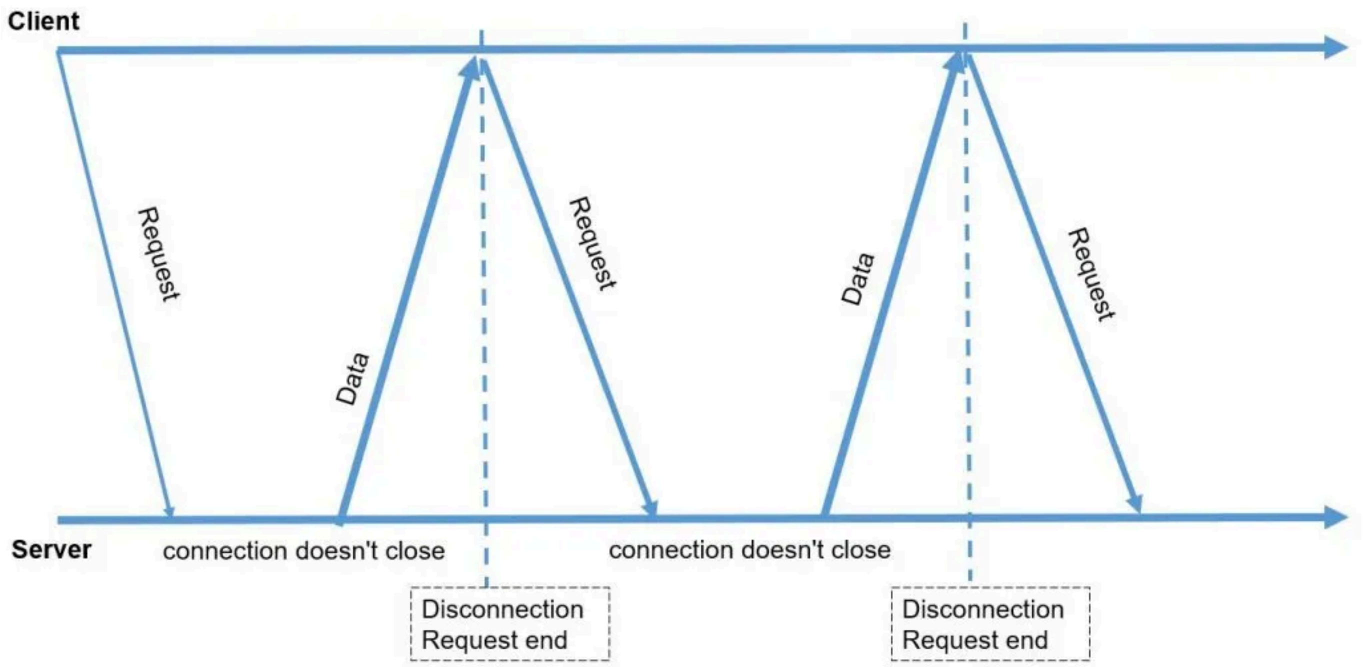
1. Web Sockets:

WebSocket is a communication protocol facilitating full-duplex, two-way communication channels through a persistent TCP connection. It enables real-time interaction between a web browser and a server.

WebSocket is suitable for bidirectional communication; however, it necessitates the implementation of a push-notification feature. Moreover, managing a multitude of WebSocket connections can consume extra server resources, posing potential challenges in resource-intensive scenarios.

2. Long Polling:

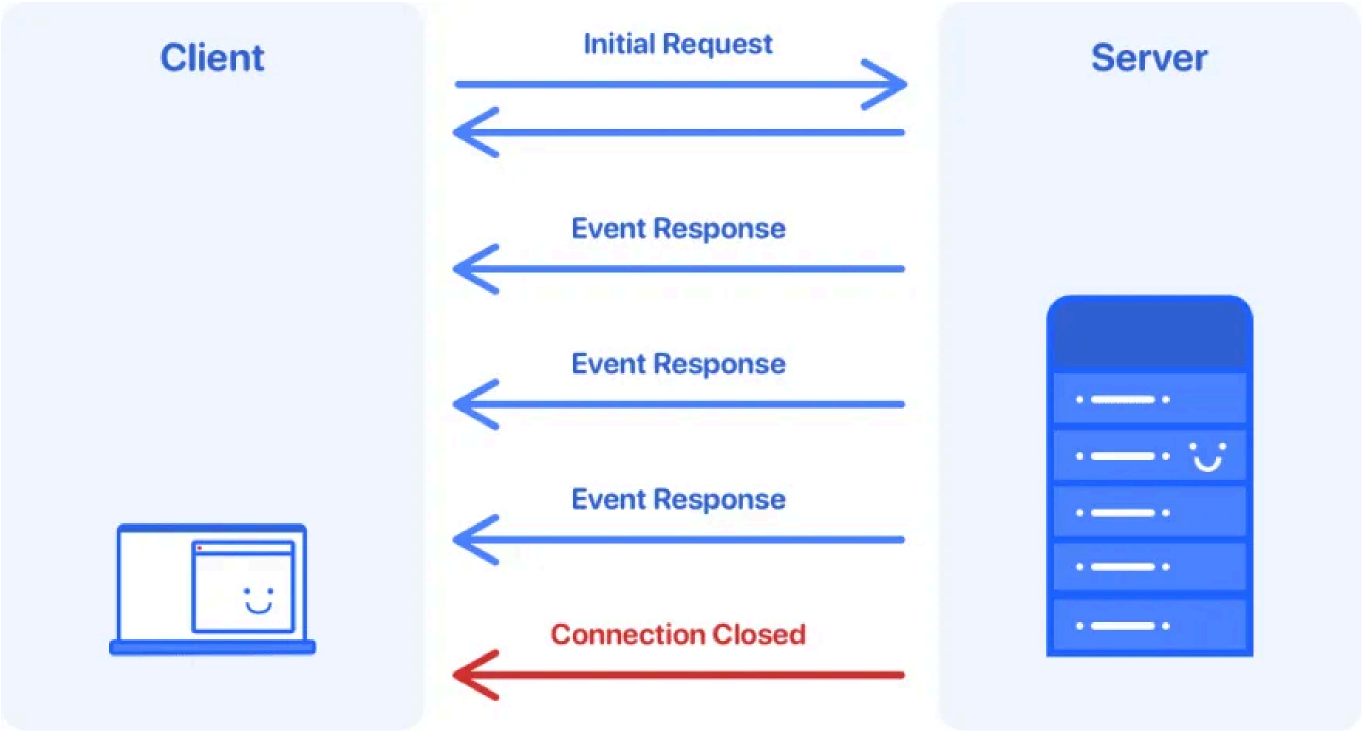
Long Polling involves the client periodically querying the server for new data. This type of connection is better suited for computationally intensive tasks where event generation requires substantial time.



3. Server-Sent Events (SSE):

SSE, a web development technology, establishes a one-way communication channel between a web server and a browser. It facilitates server-initiated real-time updates over a single HTTP connection. SSE is commonly employed for delivering live notifications, continuous updates, or data streams to a web page without requiring constant polling from the client.

SSE provides unidirectional real-time communication and consumes fewer resources compared to WebSockets. It appears that SSE fits well with all our requirements and criteria.



Hence, I have opted for SSE and intend to demonstrate how to set up SSE in FastAPI.

We will develop a basic FastAPI service to implement push-notification functionality.

The objectives of this service are as follows:

1. Authenticate users for SSE requests: User authentication will be carried out by sending requests to an authentication server.
2. Capture events from a source: The service will capture events from a source using Redis publish/subscribe functionality.
3. Dispatch notifications based on recipients: Event forwarding to recipients will be determined by recipient_id information within the

event.

GitHub source code for this article

Dependencies

pyproject.toml

```
python = "^3.10"
redis = "^4.6.0"
fastapi = {extras = ["all"], version = "^0.100.1"}
sse-starlette = "^1.6.1"
sqlalchemy = "^2.0.19"
aiohttp = "^3.8.5"
```

Authenticate users for SSE requests

In order to authenticate the user, we must validate its token. To accomplish this, we extract the token from the request header and send a request to the authentication server to verify the token's validity.

auth_service.py

```
import aiohttp
from starlette import status
from starlette.datastructures import Headers

from src.config import Config
from src.exceptions import HeadersValidationError, AuthenticationError
```

```
# Retrieving the authorization token from the incoming request header
def get_headers_token(headers: Headers) -> str | None:
    authorization_header = headers.get("authorization")
    if authorization_header is None:
        raise HeadersValidationError(msg="Credentials are not provided.", status
    if authorization_header.startswith("Bearer "):
        return authorization_header
    raise HeadersValidationError(msg="Incorrect token type.", status=status.HTTP


# Check authorization token's validity
async def is_user_authenticated(token: str) -> str | None:
    async with aiohttp.ClientSession() as session:
        try:
            response = await session.get(
                    f"{Config.MAIN_HOST}/api/v1/is_user_auth/",
                    headers={"Authorization": token}
            )
        except aiohttp.ClientError:
            raise AuthenticationError(msg="Authentication API connection error."
        if response.ok:
            return res.get("user_id")
        raise AuthenticationError(msg=res, status=status.HTTP_401_UNAUTHORIZED)
```

Capture events from a source/Dispatch notifications based on recipients

For event capture we will use Redis `Publish/Subscribe` mechanism, which allows for real-time message broadcasting and communication between different parts of an application.
Our event source server will be a publisher and FastAPI app will be subscriber. Publishers send messages to channels, while subscribers listen to specific channels for messages.

`redis_service.py`

```
import asyncio
import json
import redis

from typing import Callable
from redis import asyncio as aioredis
from redis.asyncio.client import Redis

from src.config import Config

# Create Redis conection client
async def redis_client():
    try:
        return await aioredis.from_url(
            f"redis://{Config.REDIS_HOST}:{Config.REDIS_PORT}",
            encoding="utf8", decode_responses=True
        )
    except redis.exceptions.ConnectionError as e:
        print("Connection error:", e)
    except Exception as e:
        print("An unexpected error occurred:", e)
```

SSE connection has media type `text/event-stream` therefore we should use a `generator` to generate the data by “chunks”.

```
async def listen_to_channel(filter_func: Callable, user_id: str, redis: Redis):
    # Create message listener and subscribe on the event source channel
    async with redis.pubsub() as listener:
        await listener.subscribe(Config.PUSH_NOTIFICATIONS_CHANNEL)
        # Create a generator that will 'yield' our data into opened TLS connecti
        while True:
            message = await listener.get_message()
            if message is None:
                continue
            if message.get("type") == "message":
                message = json.loads(message["data"])
            # Checking, if the user that opened this SSE conection
            # is recipient of the message or not.
            # The message obj has field recipient_id to compare.
```

```
                if filter_func(user_id, message):
                    yield {"data": message}
```

Connect all together

`main.py`

```
import uvicorn
from fastapi import FastAPI
from fastapi.params import Depends
from redis.asyncio.client import Redis
from sse_starlette.sse import EventSourceResponse
from starlette.middleware.cors import CORSMiddleware
from starlette.requests import Request

from src.auth_service import get_headers_token, is_user_authenticated
from src.exceptions import create_response_for_exception, HeadersValidationError
from src.redis_service import listen_to_channel, redis_client
from src.utils import is_user_recipient

app = FastAPI(title="notification_service")

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)


# SSE implementation
@app.get("/notify")
async def notification(request: Request, redis: Redis = Depends(redis_client)):

    # Get and check request headers
    try:
        authorization_header = get_headers_token(request.headers)
    except HeadersValidationError as e:
        return create_response_for_exception(msg=e.msg, status=e.status)
```

```
    try:
        user_id = await is_user_authenticated(authorization_header)

    # EventSourceResponse function allows to send python generators as server-se
        return EventSourceResponse(listen_to_channel(is_user_recipient, user_id,

    except AuthenticationError as e:
        return create_response_for_exception(msg=e.msg, status=e.status)


if __name__ == "__main__":
    uvicorn.run("main:app", host="0.0.0.0", port=8001, reload=True)
```

Summary

SSE is often overshadowed but it is a good tool for certain tasks and requirements. For example: updating dynamic content, sending push notifications, and streaming data in real-time and more.
For more example of SSE on Python I would recommend to visit:

- GitHub source code for this article
- Server-Sent Events in FastAPI using Redis Pub/Sub
- Realtime Log Streaming with FastAPI and Server-Sent Events

Recommended from Medium

Aayush Dip Giri

How to integrate SSE(Server Sent Events)in FastAPI using RabbitM...

Integrating Server Sent Events (SSE) into a FastAPI application with RabbitMQ and...

6 min read · Oct 31, 2023

13

Rajan Sahu

Implementing Background Job Scheduling in FastAPI with...

Enable your background job scheduling in a FastAPI application using APScheduler.

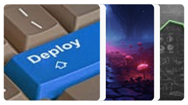
3 min read · Mar 22, 2024

61 1

Lists

Coding & Development

11 stories · 543 saves



Predictive Modeling w/ Python

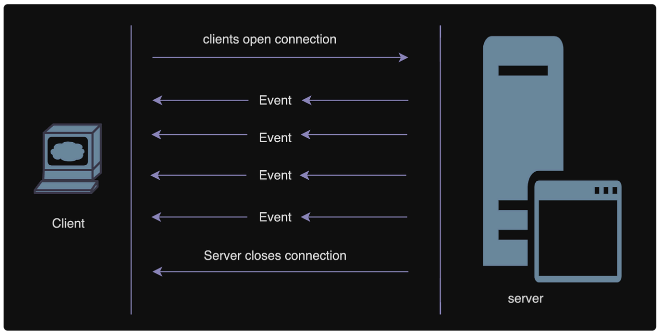
20 stories · 1066 saves

Practical Guides to Machine Learning

10 stories · 1276 saves

ChatGPT

21 stories · 552 saves



Nanda Gopal Pattanayak

Server-Sent Events with Python FastAPI

Many times when we have a requirement to send continuous stream of data from sever t...

5 min read · Mar 8, 2024

👏 180 💬

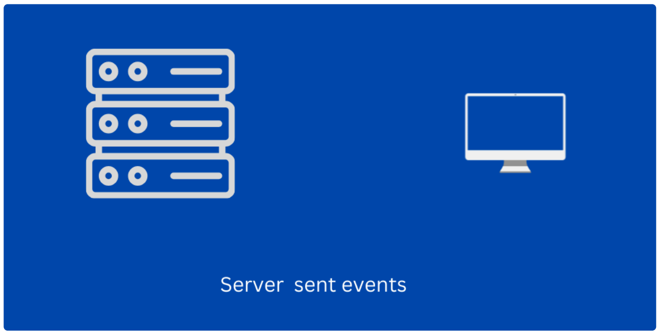
Philip Okiokio

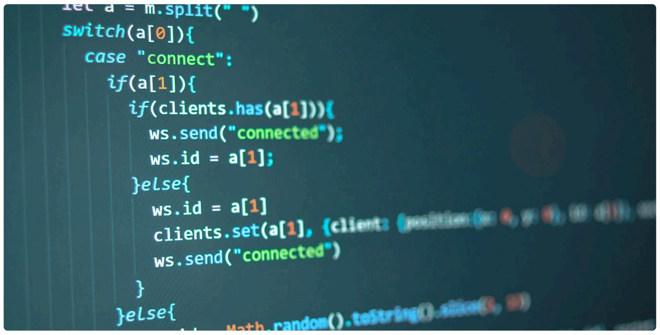
Broadcasting WebSockets Messages, across instances and...

The last Websocket-related article I wrote received a comment on LinkedIn. Intially it...

6 min read · Dec 14, 2023

👏 46 💬 1





```
switch(a[0]){
  case "connect":
    if(a[1]){
      if(clients.has(a[1])){
        ws.send("connected");
        ws.id = a[1];
      }else{
        ws.id = a[1]
        clients.set(a[1], {client:
        ws.send("connected")
      }
    }else{
```

Christopher Gathuthi

Real-time Notification Streaming using SSE and Htmx

In this article, I will demonstrate how to stream notifications using server-sent event...

9 min read · Jan 14, 2024

👏 10 💬 1

SarahDev

🚀 Supercharging WebSockets with PUB/SUB in Python, Redis, and...

Introduction:

✦ · 3 min read · Nov 8, 2023

👏 1 💬

See more recommendations