Server Sent Events

27 May, 2020

If you are looking for *realtime* features in WEB applciations you often end with Websockets.

Just to be complete I also mention 'Long/Short polling' as a client-side pull option. I regard it as mere workaround to a *realtime* communication implementation.

As always, requirements are key:

- If you need bi-directional realtime features with high volume and low latency, Websocket is your choice.
- If you need to deliver *realtime* information to many clients, but you can live with the HTTP Request/Response model to trigger actions, Server Sent Events (SSE) are a powerful contender.

> *SSE is a mechanism that allows the server to asynchronously push the data to the client once the client-server connection is established. The client subscribes to the server data via an Event Source.*

What is SSE?

Server sent events are a part of the HTML standard, not HTTP[1]. They define a protocol that is invisible to the HTTP layer, and does not disrupt any of the lower layers.

At it's core, SSE is just a `Content-Type` header that informs the client that the response will be delivered in pieces. It also alerts the browser that it should expose each piece to the code as it arrives, and not wait for the full request, much like WebSocket's frames[2].

On the browser this is implemented with the easy-to-use `EventSource` interface in client side code:

```
var source = new EventSource('updates.cgi');
source.onmessage = function (event) {
  alert(event.data);
}
```

Why SSE?

1. SSE is based on HTTP, so it has a natural fit with HTTP/2. This establishes an efficient transport layer based on multiplexed streams, out of the box. The client only needs one connection to get various data feeds (no magic limit of 6!).
2. The client and server are informed when the connection drops, there is no special protocol needed.
3. Reliability: Maintaining an unique Id with messages the server can see that the client missed a number of messages and send the backlog of missed messages on reconnect.
4. Proxy issues are much easier to handle than with Websockets, after all you are talking plain HTTP.
5. Load Balancing also just works as with regular HTTP sessions. Nothing special. Load Balancing of websockets can become very complicated and you have to roll your own solution.
6. SSL: Tick in the box.

Features like small messages size and persistent open connections from client to server are nowadays provided by HTTP/2 via header compression and de-duplication.

WebSocket is basically a different protocol for delivering data[3]. It is cannot be multiplexed over HTTP/2 connections (it doesn't really run on top of HTTP at all). Implementing custom multiplexing both on the server and the client is complicated.

Don't do it!

Make it work

Server Side code is not complicated to implement. One pitfall to look out for:
Proxy servers are known sometimes to drop HTTP connections after a short timeout. To protect against that, you can include a comment line (one starting with a ':' character) every 15 seconds or so.

This is taken care of for you when using Starlette:

Python Implementation

Using Starlette as a modern high performance Python webserver you can use just use the Response type: EventSourceResponse.

It implements the SSE protocol for Starlette and takes care of the proxy thing:

```
import asyncio
import uvicorn
from starlette.applications import Starlette
from starlette.routing import Route
from sse_starlette.sse import EventSourceResponse

async def numbers(minimum, maximum):
    for i in range(minimum, maximum + 1):
        await asyncio.sleep(0.9)
        yield dict(data=i)

async def sse(request):
    generator = numbers(1, 5)
    return EventSourceResponse(generator)

routes = [
    Route("/", endpoint=sse)
]

app = Starlette(debug=True, routes=routes)

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000, log_level='info')
```

All you have to do, is to implement a generator which provides the data to be streamed.

Summary

With Websocket you need to take care of a lot of problems that are already solved in plain HTTP.

If you just need to stream *realtime* data to many clients via an uncontrolled network, a.k.a. Internet, SSE are your friend. It simplifies your life and is around for long enough to be proven and tested.

---

1. HTML ↩
2. WebSockets, HTTP/2, and SSE ↩
3. Using SSE Instead Of WebSockets For Unidirectional Data Flow Over HTTP/2 ↩

#python #async