Politechnika Wrocławska

Wydział Informatyki i Zarządzania

Zaawansowane Technologie Webowe

Laboratorium

Temat:	WebSocket
Opracował:	mgr inż. Piotr Jóźwiak
Data:	marzec 2023
Liczba godzin:	2 godziny

Table of Contents

Wstęp

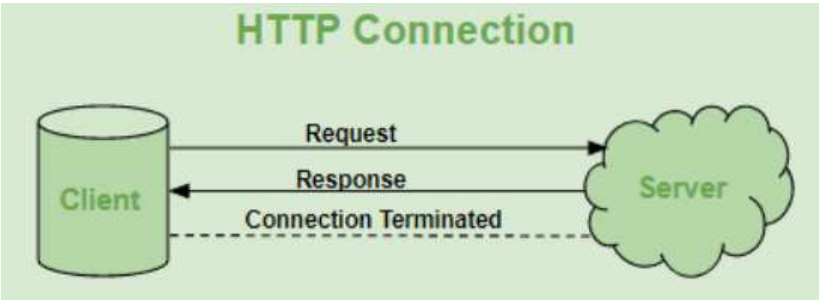
Zanim omówimy sobie czym jest sam WebSocket przypomnijmy sobie jak działa standardowy protokół HTTP.

HTTP oraz jego ograniczenia

HTTP jest protokołem jednokierunkowym, gdzie klient wysyła żądanie, a serwer wysyła odpowiedź. Na przykład, gdy użytkownik wysyła żądanie HTTP/HTTPS do serwera, po otrzymaniu żądania serwer wysyła odpowiedź do klienta. Każde żądanie jest związane z odpowiednią odpowiedzią, po wysłaniu odpowiedzi połączenie zostaje zamknięte, każde żądanie HTTP lub HTTPS nawiązuje nowe połączenie z serwerem za każdym razem i po otrzymaniu odpowiedzi połączenie zostaje zakończone.

HTTP jest protokołem bezstanowym, który działa na szczycie TCP, który jest protokołem zorientowanym na połączenie, gwarantuje dostarczenie pakietów danych przy użyciu trójstronnych metod handshaking i ponowne przesłanie utraconych pakietów. W zasadzie HTTP może działać na dowolnym protokole zorientowanym na połączenie, takim jak TCP czy SCTP. Kiedy klient wysyła żądanie HTTP do serwera, połączenie TCP jest otwierane pomiędzy klientem a serwerem, a po otrzymaniu odpowiedzi połączenie TCP zostaje zakończone. Każde żądanie HTTP otwiera osobne połączenie TCP z serwerem, np. jeśli klient wysyła 10 żądań do serwera, 10 osobnych połączeń TCP zostanie otwartych i zamkniętych po otrzymaniu odpowiedzi.

Reasumując powyższe jednoznacznie widać, jak standardowy protokół HTTP jest oparty o architekturę Klient-Serwer. Klient (przeglądardka) zawsze inicjuje komunikację i w zasadzie komunikacja ta ma charakter prostej rozmowy przypominającej zadawanie pytania i otrzymywania zwięzłej odpowiedzi na nie od serwera zaraz po zadanym pytaniu. Schemat tej komunikacji przedstawia poniższy rysunek:



Należy zwrócić tutaj szczególną uwagę na fakt, że po wysłaniu odpowiedzi z Serwera, serwer zamyka połączenie. Jeśli Klienta (przeglądarkę) interesuje coś dodatkowo, to musi rozpocząć od nowa komunikację. W powyższym schemacie komunikacji nie istnieje możliwość:

- Zadania pytania i oczekiwania na odpowiedź, jak Serwer będzie posiadał wiedzę
- Umożliwienie Serwerowi zainicjowania komunikacji do Klienta, aby poinformować Klienta o jakiś zdarzeniu.

Rozpatrzmy przypadek, w którym Klient zleca przez stronę internetową wykonanie jakiegoś żądania, które po stronie serwera zajmie dłuższą chwilę. Aby przeglądarka mogła monitorować stan realizacji zlecenia musi np. co 1 sekundę odpytywać Serwer o aktualny stan. Przypomina to sytuację, w której użytkownik sam co 1 sekundę odświeża stronę przyciskiem F5, aby sprawdzić czy zadanie jest gotowe. Łatwo w tym zachowaniu, zwanym **HTTP Pooling,** doszukać się jego słabych stron w postaci wysyłania sporej ilości

zapytań, które nie zmieniają nic w naszej sytuacji poza konsumpcją i marnowaniem zasobów w postaci liczby nawiązanych połączeń do serwera jak i zasobów obliczeniowych czy pamięciowych.

Rozwiązaniem tego problemu może być wykorzystanie mechanizmu zwanego **HTTP Long Polling**. Long Polling jest wzorcem dostępu do danych w czasie zbliżonym do rzeczywistego. Klient inicjuje połączenie TCP (zwykle żądanie HTTP) o maksymalnym czasie trwania (np. 20 sekund). Jeśli serwer ma dane do zwrócenia, zwraca je natychmiast, zwykle w partiach do określonego limitu. Jeśli nie, serwer wstrzymuje wątek żądania do czasu, aż dane staną się dostępne, a następnie zwraca dane do klienta. W zasadzie rozwiązanie to sprowadza się do wydłużenia timoutu żądania do np. 20 sekund. To rozwiązanie ma swoje wady, gdyż w tym czasie na serwerze cały czas jest uruchomiony wątek obsługujący to żądanie. Optymalizacji została poddana tylko warstwa sieciowa, w której nie musimy nawiązywać wielu połączeń. Dodatkowo w dalszym kroku nasza komunikacja jest w zasadzie jednokierunkowa. Po ustanowieniu połączenia poprzez wysłanie żądania, to serwer ma jakiś czas (np. 20 sekund) na przygotowanie i wysłanie odpowiedzi. W tym czasie kanał komunikacyjny nie umożliwia wysłania kolejnego zapytania od strony klienta. Zatem tego typu rozwiązanie jest połowiczne.

WebSocket

Mając powyższe ograniczenia na uwadze chcąc zbudować aplikację działającą w czasie rzeczywistym bez nadmiernego marnowania zasobów należy skorzystać w funkcjonalności WebSocket. Protokół WebSocket jest w przeciwieństwie do HTTP protokołem dwukierunkowym, full-duplex w architekturze Klient-Serwer. Aby wskazać chęć wykorzystania tego protokołu musimy wskazać go w adresie żądania w postaci przedrostka ws:// lub dla wersji z szyfrowaniem wss://. Jest to odpowiednik selektora protokołu http:// oraz https://. Samo połączenie WebSocket jest połączeniem stanowym w przeciwieństwie do HTTP. Oznacza to, że połączenie między klientem i serwerem jest utrzymywane do momentu zakończenia go przez jedną ze stron. Utworzone w ten sposób połączenie wykorzystywane jest poprzez obie strony do komunikacji dwukierunkowej.

Samo połączenie nawiązywane jest na początku jako połączenie HTTP. Po wykonaniu handshake'u klient prosi Serwer o przekształcenie połączenia w WebSocket. Jeśli Serwer tylko potrafi to wykonać to zastępuje połączenie HTTP na WS, przez co dalsza komunikacja może swobodnie przepływać pomiędzy obiema stronami. Schematycznie przedstawia to poniższy rysunek:

Kiedy zatem korzystać z protokołu WebSocket:

1. Web aplikacje czasu rzeczywistego – są podstawowym przypadkiem, w którym wykorzystanie WebSocketów jest w pełni uzasadnione.
2. Web aplikacje gamingowe/video – są kolejnym przykładem, w których użycie Web Socketów jest wymagane. Tylko tutaj mamy możliwość utrzymywania cały czas strumienia danych niezbędnego

do przesyłania szybko zmieniających się danych jak w grach czy strumienia video/audio, którego realizacja w oparciu o protokół HTTP nie jest możliwa.

3. Komunikatory (Chaty) – to kolejny przykład, w którym WebSocket wpisuje się idealnie w zastosowanie. Ze swej natury komunikator wymaga utrzymywania dwukierunkowego połączenia, w którym każda ze stron może rozpocząć nadawanie komunikatów.

Kiedy nie korzystać z WebSocketów? W zasadzie w każdej sytuacji, w której praca aplikacji nie traci na funkcjonalności w oparciu o standardowe połączenie HTTP. Czyli w sytuacji, gdzie proste żądanie i natychmiastowa odpowiedź całkowicie zaspokaja potrzeby. Zawsze, kiedy zapytania są rzadko wysyłane, nawet jeśli odpowiedź nie jest dostępna natychmiast, to kolejne żądanie będzie wykonane np. za 5 minut bez utraty na jakości rozwiązania. W tym przypadku przy tak długich przerwach między żądaniami protokół HTTP wydaje się być wystarczający.

Ogólnie różnice między obydwoma protokołami zostały zebrane w poniższej tabeli:

WebSocket	HTTP
WebSocket jest dwukierunkowym protokołem komunikacyjnym, który może przesyłać dane z klienta do serwera lub z serwera do klienta poprzez ponowne wykorzystanie ustanowionego kanału połączenia. Połączenie jest utrzymywane przy życiu do momentu zakończenia przez klienta lub serwer.	Protokół HTTP jest protokołem jednokierunkowym, działającym na szczycie protokołu TCP. Żądanie zawsze wychodzi od Klienta do Serwera. Serwer musi odpowiedzieć natychmiast na żądanie.
Stosowany głownie do aplikacji czasu rzeczywistego, takich jak (handel, monitoring, powiadomienia). Usługi używają WebSocket do odbierania danych na pojedynczym kanale komunikacyjnym.	Tradycyjne aplikacje webowe, które są bezstanowe (RESTful).
Full-duplex	Half-duplex

Prosty serwer WebSocket w oparciu o NodeJS

Skoro mamy już omówioną teorię stojącą za WebSocketami, przyjrzyjmy się praktycznemu wykorzystaniu WebSocoket. W tym celu napiszemy prosty serwer WS wykorzystując NodeJS. Jeśli jeszcze nie masz zainstalowanego NodeJS to proponuję wykorzystać do tego npm. Po poprawnym zainstalowaniu NodeJS w docelowym folderze zainicjujmy projekt Node poniższym poleceniem:

```
npm init
```

Powyższe polecenie utworzy nowy projekt poprzez zadanie pytania o podstawowe dane. Jeśli spotykasz się z tym pierwszy raz i chcesz poznać więcej szczegółów to odsyłam do tego https://nodesource.com/blog/an-absolute-beginners-guide-to-using-npm/ artykułu. Po poprawnym zainicjowaniu projektu, głównym plikiem serwera będzie plik index.js. Zatem utwórz taki plik wprowadzając w nim poniższą zawartość:

```
//importing http module
const http = require('http');
```

```
//importing ws module
const websocket = require('ws');

//creating a http server
const server = http.createServer((req, res) => {
    res.end("I am connected");
});
//creating websocket server
const wss = new websocket.Server({ server });

//calling a method 'on' which is available on websocket object
wss.on('headers', (headers, req) => {
    //logging the header
    console.log('WebSocket.on headers:\n');
    console.log(headers);
});

console.log('Listening on http://localhost:8000 ...');
//making it listen to port 8000
server.listen(8000);
```

Powyższy przykład jeszcze się nie uruchomi. Zwróć na uwagę, że w pierwszych dwóch poleceniach importujemy dwa moduły: http oraz ws. Pierwszy jest już zainstalowany w systemie razem z NodeJS, jednak drugi trzeba doinstalować samemu. W tym celu wykonaj polecenie:

```
npm install ws
```

Teraz możemy już uruchomić nasz serwer poleceniem:

```
npm start
```

otrzymując w konsoli poniższy komunikat:

```
> lab8-ws@1.0.0 start
> node index.js

Listening on http://localhost:8000 ...
```

Nasz serwer powinien nasłuchiwać pod adresem: http://localhost:8000. Połączenie się przeglądarką do tego adresu nie spowoduje niczego spektakularnego. Przeglądarka wykona tradycyjne żądanie HTTP. Aby nasze żądanie było żądaniem WebSocket musimy napisać stronę frontendową, która będzie takowe połączenie nawiązywać. Zatem utwórz plik client.html z poniższą zawartością:

```
<html>

  <script>
```

```
    //calling the constructor which gives us the websocket object: ws
    let ws = new WebSocket('ws://localhost:8000');
  </script>

  <body>
    <h1>This is a client page</h1>
  </body>
</html>
```

Na tą chwilę nie ma tutaj nic spektakularnego, poza tym, że tworzymy obiekt WebSocket, który łączy się do naszego serwera. Upewniając się, że serwer wciąż pracuje uruchom w przeglądarce plik client.html i zaobserwuj co pojawi się na konsoli serwera NodeJS:

```
[
  'HTTP/1.1 101 Switching Protocols',
  'Upgrade: websocket',
  'Connection: Upgrade',
  'Sec-WebSocket-Accept: qvDFdSlEPckYpWgjOlkZ1BcyK3g='
]
```

Widzimy tutaj nagłówek żądania jaki został wysłany do serwera. Jest to nagłówek, który prosi o konwersję naszego połączenia w WebSocket. Nagłówek ten wyświetlił się ponieważ w kodzie serwera przypięliśmy się do zdarzenia **headers**. Jest to linia `wss.on('headers', (headers, req) => {` .

Wcześniej natomiast utworzyliśmy stałą `wss`, która jest zainicjowana obiektem `new websocket.Server`. Obiekt ten dostarcza nam niezbędne API do tworzenia i zarządzania połączeniem WebSocket. W tym przypadku obiekt `wss` pomoże nam nasłuchiwać zdarzeń emitowanych, gdy coś się wydarzy. Np. gdy połączenie zostanie ustanowione lub zakończone itp.

Wracając do naszego nagłówka. Zwróć uwagę na pierwszą linię zawierającą polecenie *Switching Protocols*. Rozłóżmy na czynniki co się wydarzyło:

- Pierwszą rzeczą, którą zauważysz, jest to, że otrzymaliśmy kod statusu **101**. Być może widziałeś już wcześniej kody **200**, **201**, **404**. **101** jest statusem HTTP informującym o zmianie protokołu (101 Switching Protocols). Mówi „Hej, potrzebuję uaktualnienia”.
- Drugi wiersz zawiera informacje o aktualizacji. Określa, że chce uaktualnienia do protokołu WebSocket.
- To właśnie dzieje się podczas handshake’a. Przeglądarka używa połączenia HTTP do ustanowienia połączenia przy użyciu protokołu HTTP/1.1, a następnie uaktualnia go do protokołu WebSocket.

Zdarzenie “headers” jest emitowane, zanim nagłówki odpowiedzi zostaną zapisane do gniazda w ramach uzgadniania. Pozwala to na sprawdzenie/modyfikację nagłówków przed ich wysłaniem. Innymi słowy daje nam to możliwość podjęcia decyzji o akceptacji połączenia lub jego odrzuceniu.

Dodajmy kolejne zdarzenia o nazwie **connection** oraz **message**, które jest emitowane po zakończeniu nawiązywania połączenia przez klienta. W tym miejscu możemy wysłać pierwszą wiadomość do klienta. Przykładowy kod może wyglądać następująco (dodajemy do index.js):

```
//Event: 'connection'
wss.on('connection', (ws, req) => {
    ws.send('Welcome, your connection is ready');
    //receive the message from client on Event: 'message'
    ws.on('message', (msg, isBinary) => {
        console.log('Received message from client:');
        const message = isBinary ? msg : msg.toString();
        console.log(message);
    });
});
```

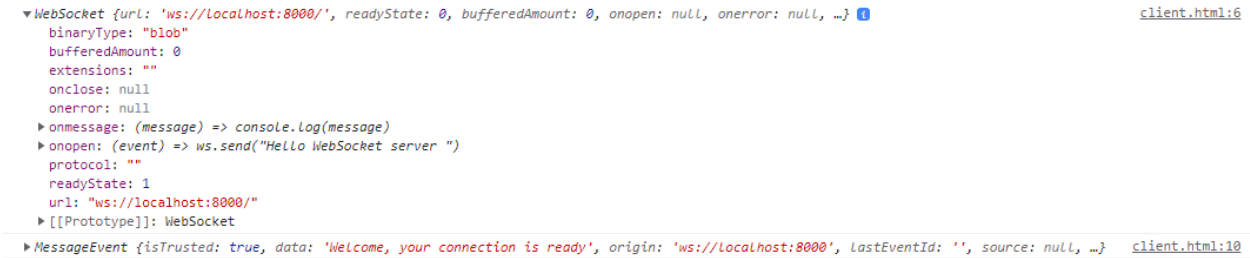
Dodatkowo modyfikujemy także kod klienta (plik client.html):

```
<script>
    //calling the constructor which gives us the websocket object: ws
    let ws = new WebSocket('ws://localhost:8000');
    //logging the websocket property properties
    console.log(ws);
    //sending a message when connection opens
    ws.onopen = (event) => ws.send("Hello WebSocket server ");
    //receiving the message from server
    ws.onmessage = (message) => console.log(message);
</script>
```

Po ponownym uruchomieniu serwera z wprowadzonymi zmianami oraz wywołaniu kodu klienta otrzymujemy na serwerze poniższy log:

```
Received message from client:
Hello WebSocket server
```

Natomiast po stronie przeglądarki nie odnotowaliśmy zasadniczo nic nowego. Aby zobaczyć co się zmieniło musimy uruchomić w przeglądarce narzędzia deweloperskie. W przypadku przeglądarki Chrome wystarczy wcisnąć przycisk F12. W zakładce Console otrzymujemy poniższe wpisy:



Widać tutaj jak nawiązujemy połączenie do serwera WebSocket. Następnie otrzymujemy MessageEvent z wiadomością od serwera. Nasze połączenie działa poprawnie.

Oczywiście przedstawiony powyżej przykład jest trywialny. Demonstruje jedynie proste połączenie i wysłanie kilku wiadomości. Na upartego podobną rzecz można byłoby uzyskać zwykłym AJAXem czy też

HTTP. Siła tego rozwiązania tkwi gdzie indziej. Mając zestawione połączenie nic nie stoi na przeszkodzie, aby serwer sam mógł w dowolnym momencie inicjować wysłanie wiadomości do klienta. Jednakże tą część pozostawiam do wykonania samodzielnego.

Biblioteka Socket.IO jako nakładka na WebSocket

W poprzednim rozdziale omówiony został przykład wykorzystujący bezpośrednio API WebSocket. Jednakże nie zawsze jesteśmy zmuszeni do wykorzystywania niskopoziomowego API. Czasem łatwiej jest skorzystać z biblioteki, która wiele spraw może przyspieszyć lub ułatwić. Jako przykład chciałbym wskazać bibliotekę Socket.IO dostępną tutaj: https://socket.io/.

Socket.IO jest biblioteką umożliwiającą pisanie połączeń czasu rzeczywistego na bazie protokołu HTTP, dostarczającą dwukierunkową komunikację pomiędzy klientem i serwerem. Jeśli jest to tylko możliwe, to Socket.IO stara się wykorzystywać WebSocket do komunikacji. Jednakże, jeśli któraś ze stron nie jest w stanie takie połączenie zapewnić to biblioteka stara się zapewnić połączenie z wykorzystanie Long Polling HTTP (zazwyczaj AJAX).

Przez długi czas Socket.IO oficjalnie udostępniał jedynie implementację serwera Node.js, oraz klienta JavaScript. Ostatnio twórcy Socket.IO opracowali także kilka innych klientów, w Javie, C++ i Swift.

Poza oficjalnie wspieranymi implementacjami, istnieje wiele innych implementacji klienta i serwera Socket.IO, utrzymywanych przez społeczność, obejmujących języki programowania i platformy takie jak Python, Golang czy .NET.

Trzeba mieć jednak na uwadze, że biblioteka Socket.IO nie jest tak wydajna jak czysty WebSocket. Sam WebSocket jest też mniej pamięci żerny. Wynika to z faktu, że Socket.IO dostarcza znacznie więcej niż czysty WS.

Socket.IO zapewnia takie funkcje jak auto-reconnect, pokoje i fallback do długiego pollingu. W przypadku czystego WebSocket nie korzystasz z takich funkcji po wyjęciu z pudełka; musisz sam zbudować wszystkie te możliwości, jeśli są one istotne dla twojego przypadku użycia.

W wielu miejscach odwołanie do Socket.IO jest bardzo podobne do czystego WebSocket. Zwróć uwagę na poniższy przykład kodu Socket.IO dla serwera i klienta:

```
import { Server } from "socket.io";

const io = new Server(3000);

io.on("connection", (socket) => {
  // send a message to the client
  socket.emit("hello", "world");

  // receive a message from the client
  socket.on("howdy", (arg) => {
    console.log(arg); // prints "stranger"
  });
});
```

```
import { io } from "socket.io-client";

const socket = io("ws://localhost:3000");

// receive a message from the server
socket.on("hello", (arg) => {
  console.log(arg); // prints "world"
});

// send a message to the server
socket.emit("howdy", "stranger");
```

Widać jak bardzo praca z Socket.IO przypomina pracę z czystym WebSocket. Więcej szczegółów proszę doczytać w dokumentacji Socket.IO dostępnej tutaj: https://socket.io/docs/v4/