Web Development and API Design

Lesson 11: WebSockets

Prof. Andrea Arcuri

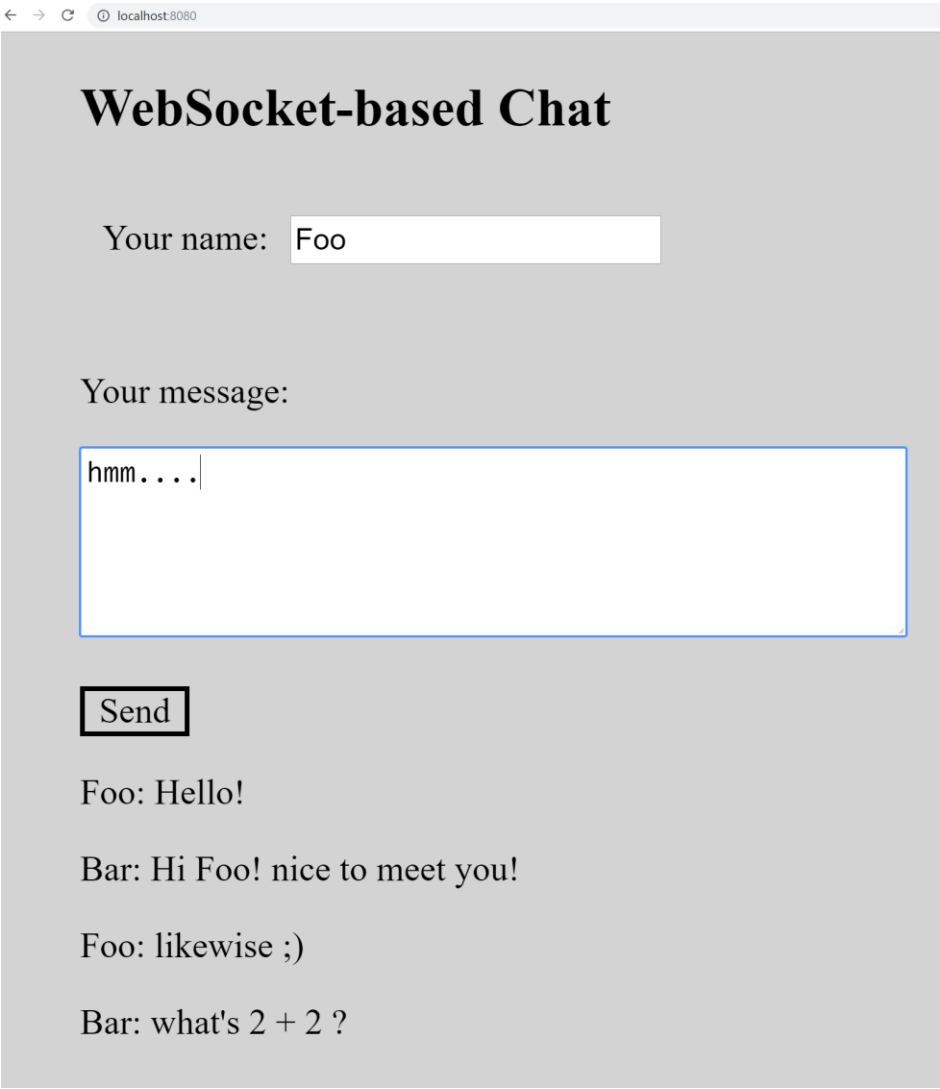
Goals

- Understand what is the problem that *WebSockets* solve
- Learn how to add *WebSocket* support to a *React/NodeJS* app

WebSockets

Chat Application

- How would you implement a chat app in a browser?
- It is not as simple as it sounds…



Option 1: Server-Side Templates

- GET HTML page with current messages
- Create new message with a POST form submission, returning the updated HTML page
- *Issue 0*: download all messages even if only 1 new is created
- *Issue 1*: current user will not see the new messages of other users until s/he interacts with the app
  - eg, reload page or post new message

Option 2: AJAX Polling

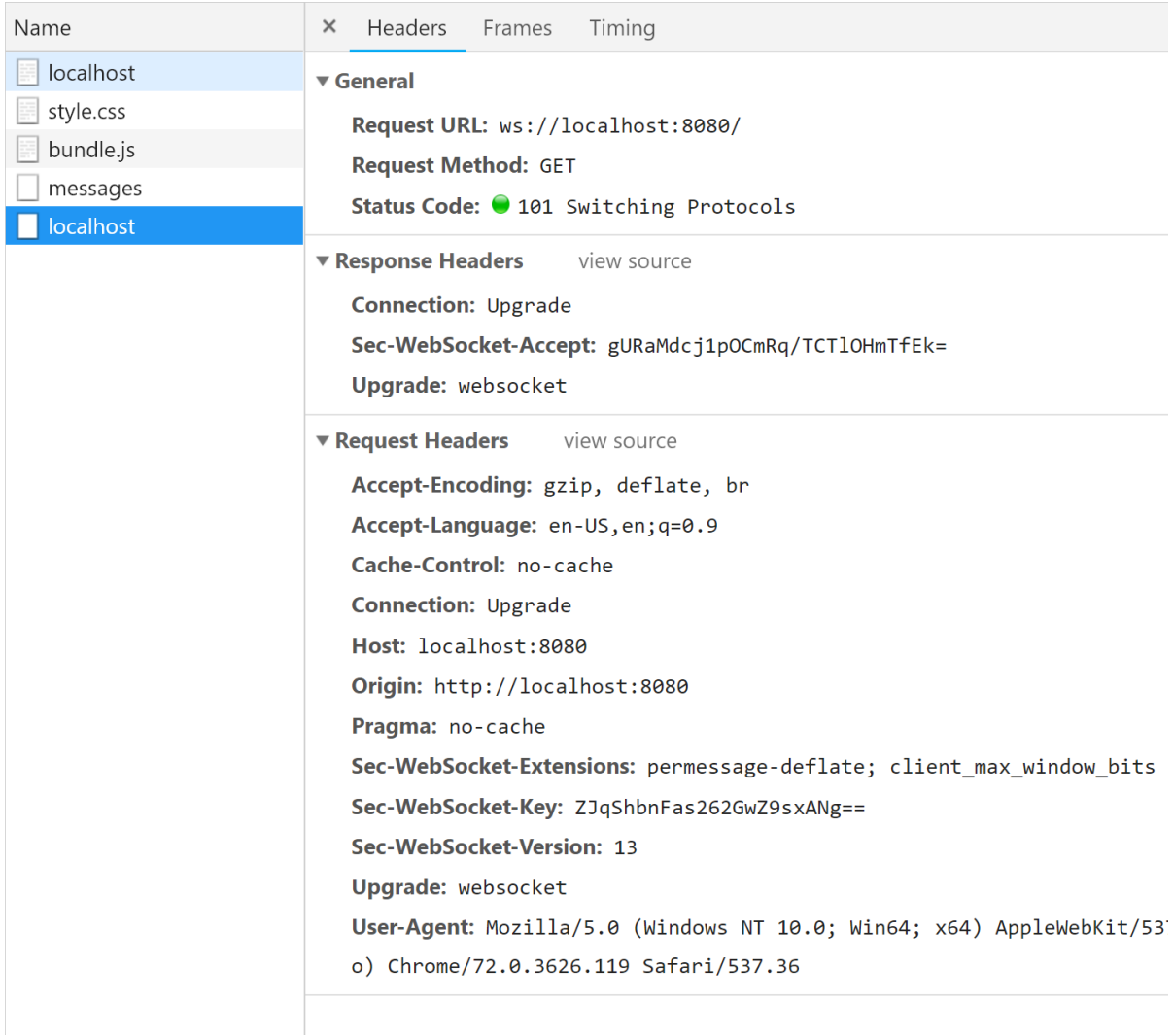
- Use AJAX to fetch list of only the new messages to display
- Repeat AJAX calls in a loop, eg every X milliseconds
- *Issue 0*: might have to wait up to X ms before seeing the new messages from other users
- *Issue 1*: if no new messages, all these AJAX requests are a huge waste of bandwidth
- Choosing X is a tradeoff between Issue 0 and 1
  - eg, small X improves usability, but at a huge bandwidth waste cost

Option 3: WebSockets

- Besides HTTP, establish a WS connection
  - most browsers do support WS
- WS enables duplex communications
  - server can decide to send data to browser, which will listens to updates
- Server will keep an active TCP connection for each client
- When new message, server can *broadcast* it to all clients
- Browser just waits for notifications, and update HTML when it receives incoming messages from server
- Server *pushes* data only when available
  - no bandwidth waste

WebSocket Protocol

- Usually over TCP
- It is **NOT** HTTP, but *first message* has same syntax as HTTP
- Note the different protocol in the URL, eg ***ws://localhost:8080***
  - **wss** is for encrypted, like HTTPS



Request ws://localhost:8080

- When making a request using WS protocol, browser will craft a message with same syntax as HTTP, with following headers
- **Sec-WebSocket-Extensions**
  - specify some WS extensions to use during the communications, like how to compress the messages, eg, *permessage-deflate* tells to use the "*deflate*" compression algorithm
- **Sec-WebSocket-Key**
  - needed to tell the server that this is indeed a WS connection, and not a HTTP one
  - using a random key
- **Sec-WebSocket-Version**
  - tell the server which version of WS protocol the browser is using
- **Upgrade: websocket**
  - standard HTTP header, telling that, although this request was handled like HTTP, the client (ie browser) wants to switch to a different protocol (WS in this case)

Server Response

- If server supports and accepts the WS connection, it will answer with a HTTP message having the following
- **Connection: Upgrade**
  - tell browser to update the connection from current HTTP to something else
- **Upgrade: websocket**
  - the protocol to use for all following requests
- **Sec-WebSocket-Accept**
  - used to confirm that server is willing to use WS protocol for all following requests
  - it contains the hashed key sent by the browser. Useful to prevent caches to resend previous WS conversations
- HTTP status code **101**
  - it represents "*Switching Protocols*"

Established WS Connection

- Once WS is established, can send blocks of byte data or strings over TCP
- Can wait for receiving messages
  - duplex communications between browser and server
  - data split and sent as "*frames*" of bytes, with special codes to specify sequences of frames belonging to the same message
- How to structure messages is up to you
  - eg, could use protocols like STOMP
- Typically, *we will just send JSON objects, serialized as strings*

Why First Message in HTTP?

- It allows server to have a single listening TCP socket
  - eg, either 80 or 443, serving both HTTP(S) and WS(S)
- Easy to integrate in current web infrastructures, including **reverse-proxies**
  - often you do not speak directly with a server, but rather with proxies and gateways in front of them… but this is not something we will see in this course
- WS is younger than HTTP
  - first version in Chrome in 2009
- Needed an easy way to integrate the new WS protocol in the existing web infrastructures tailored for HTTP

WebSocket in the Browser

- In JavaScript, can use the *WebSocket* class from global scope
  - Most browsers nowadays support WS
- **WebSocket(url)**
  - create a WS object, trying to connect to the given URL of the server
  - recall to use either "*ws*" or "*wss*" as protocol, and not "*http*"
- **WebSocket.send(payload)**
  - send the given payload (e.g., a string) to the server
- **WebSocket.onmessage**
  - callback used to handle messages from server
- **WebSocket.close()**
  - to close the connection

WebSocket in the Server

- Backend support for WS depends on the programming language and libraries we use
- In this course, we will use the library "*ws*", and "*express-ws*" to integrate it with Express
- In Express, we will have an endpoint dealing with the "*ws://*" protocol
- When called, a WS object will be created, on which we can register callbacks for incoming messages, open/close events, send messages to browser, broadcast to all users, etc.