howest
hogeschool

Realtime Communication

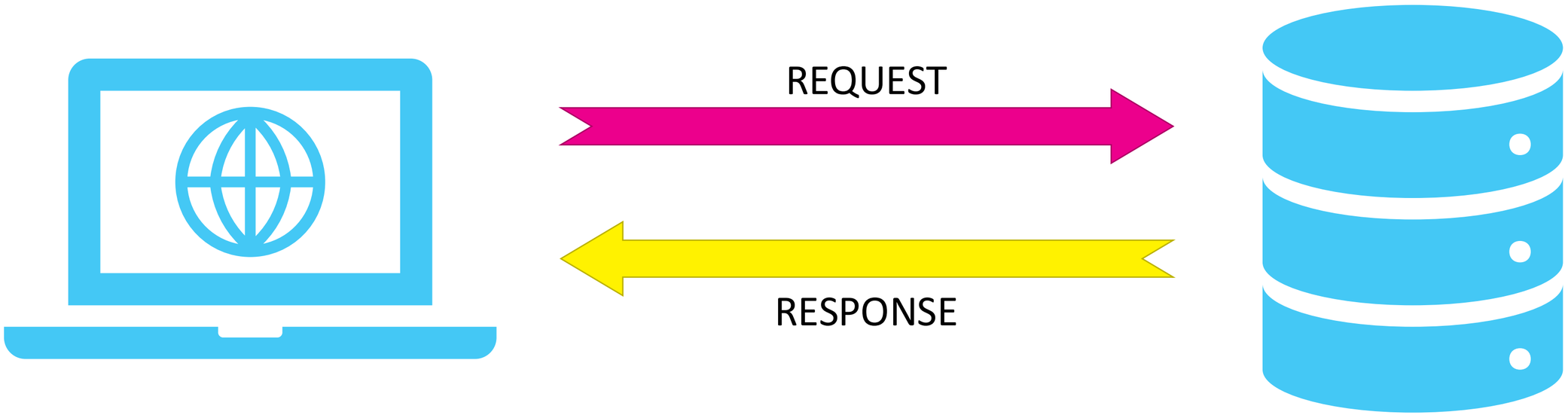
Websockets
SocketIO
Sockets + FastAPI

howest
hogeschool

Websockets

HTTP Connection

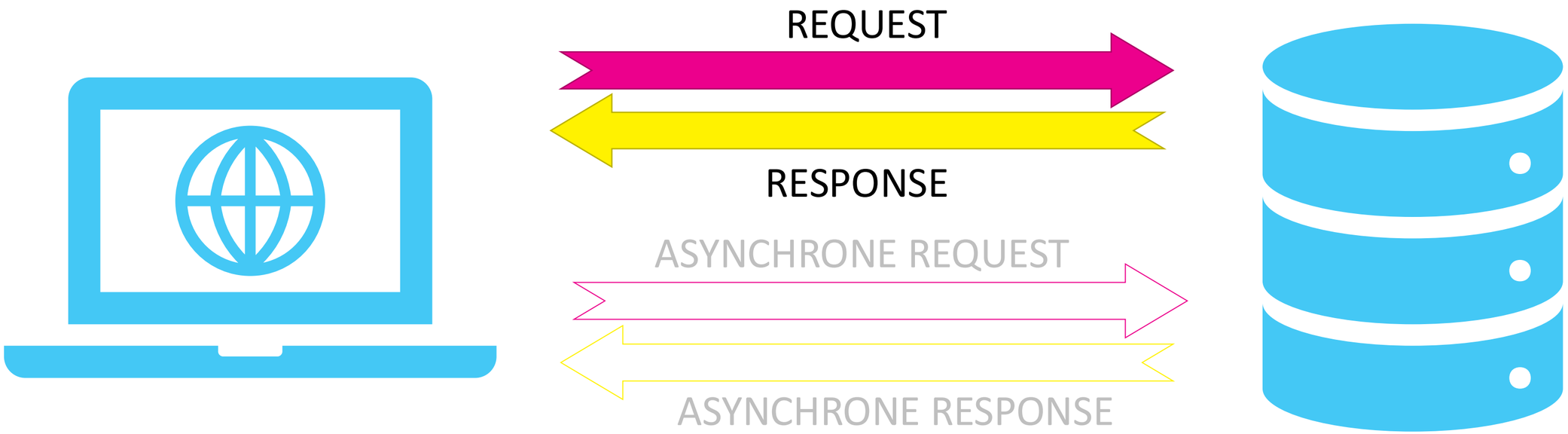
Fetch a page using HTTP:



- A HTTP **request** (GET, POST) followed by a **response**
- **One** connection for each pair. Connections are always stopped

HTTP Connection – Async calls

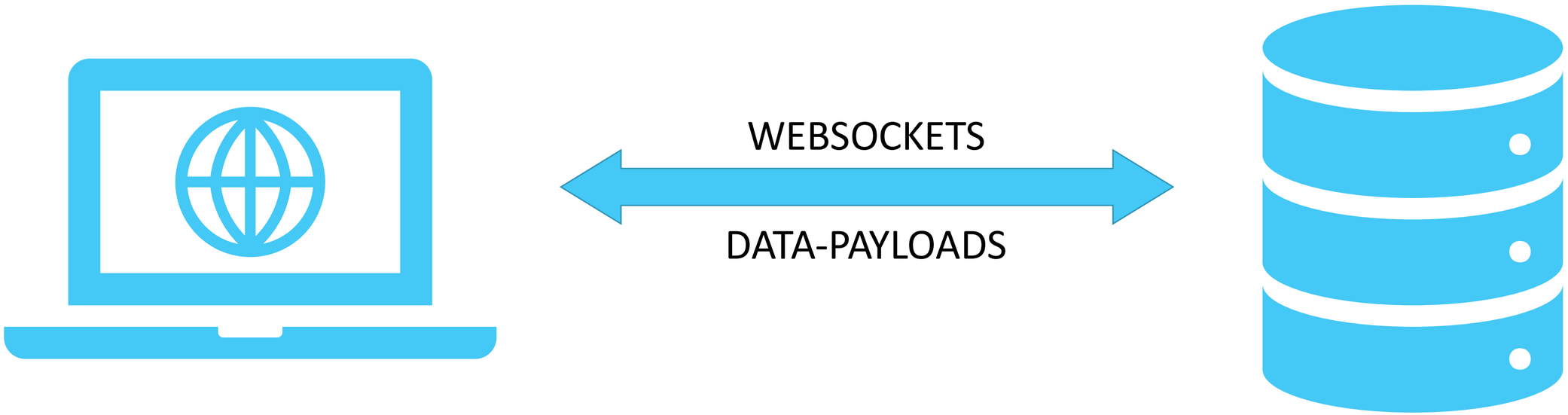
Fetch a page using HTTP but with extra sync calls



- A HTTP **request** (GET, POST) followed by a **response**
- **One** connection for each pair. Connections are always stopped

Websockets

Sometimes we need **realtime** data, which we want to see without fetching and refreshing it.



- Using websockets, there is one connection that keeps on being opened, so we can share data without always sending a request and returning a response.

What are Websockets?

- Websocket is a **networkprotocol** that is a **full-duplex** communication over a TCP-connection. It allows for bidirectional communication.
- A connection to ws:// or wss:// (instead of http://)
- To open a websocket, a special HTTP-request is sent with the `connection: upgrade`-request.
- The server sends a response with code `101 Switching Protocols`
- From now on, bidirectional communication can be sent without the need for http-requests or responses, but using **messages**.
- The messages can contain text data, and thus also **JSON** and Binary (images)

Real-world use cases

- Chat applications
- Multiplayer games
- Live notifications
- Collaborative tools

Websocket vs SocketIO

- SocketIO abstracts the WebSocket complexity
  - Is **NOT** a WebSocket implementation
- Low-latency, bidirectional, event-based
- Room-option for easily sharing data to multiple clients
  - Scalable
  - Broadcasting
- Reliable fallback option
  - Will fallback to HTTP long-polling if a connection is lost
  - Automatic reconnection

Using websockets

- NodeJS packages: **ws, Socket.IO**
  - Useful in Backend systems in NodeJS
  - Javascript doesn’t need a specific package for websockets
    - But for SocketIO it does!
- Python packages: websockets, python-socketio
  - Integrations with Flask & FastAPI

Socket IO Basic Example

```
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");
```

Code snippets from https://socket.io/

howest hogeschool

Debugging tools / Testing

- Chrome DevTools
  - Easy to debug and check it out when working with a client in Javascript
- Postman WebSocket
  - Also supports SocketIO connections

howest
hogeschool

Websockets in practice

With FastAPI

Installing

FastAPI serverside

- Simply install the `websocket` package together with `fastapi` to get started

Clientside Gradio

- Also use the `websocket` package with `gradio` as your Python frontend

For other Clients and Services, search for the packages you need

To test: Use Postman as a Client if needed

FastAPI setup

from starlette.websockets import WebSocket

```
from fastapi import FastAPI, WebSocket


app = FastAPI()


@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    while True:
        data = await websocket.receive_text()
        await websocket.send_text(f"Message text was: {data}")
```


howest hogeschool

FastAPI handling multiple connection

- Create a **manager** Class
- Keep track of **all the connections** in that manager Class
- Broadcast
  - Send a text to everyone
- Response message
  - Respond (Send) a text to the one that sent the first request

FastAPI handling multiple connection -- Code

```python
class ConnectionManager:
    def __init__(self):
        self.active_connections: list[WebSocket] = []

    async def connect(self, websocket: WebSocket):
        await websocket.accept()
        self.active_connections.append(websocket)

    def disconnect(self, websocket: WebSocket):
        self.active_connections.remove(websocket)

    async def send_personal_message(self, message: str, websocket: WebSocket):
        await websocket.send_text(message)

    async def broadcast(self, message: str):
        for connection in self.active_connections:
            await connection.send_text(message)

manager = ConnectionManager()
```

```python
@app.websocket("/ws/{client_id}")
async def websocket_endpoint(websocket: WebSocket, client_id: int):
    await manager.connect(websocket)
    try:
        while True:
            data = await websocket.receive_text()
            await manager.send_personal_message(f"You wrote: {data}", websocket)
            await manager.broadcast(f"Client #{client_id} says: {data}")
    except WebSocketDisconnect:
        manager.disconnect(websocket)
        await manager.broadcast(f"Client #{client_id} left the chat")
```

howest
hogeschool

Socket.io in practice

With FastAPI

Socket.io – How it works

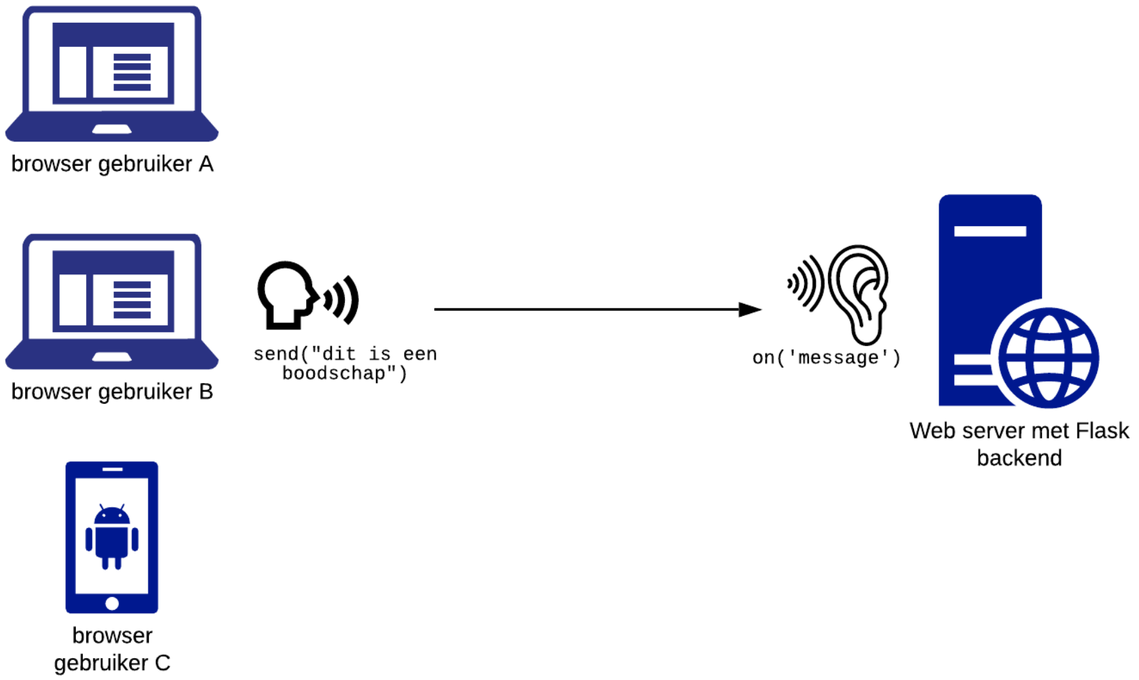
socket.io

Send: Sending a String

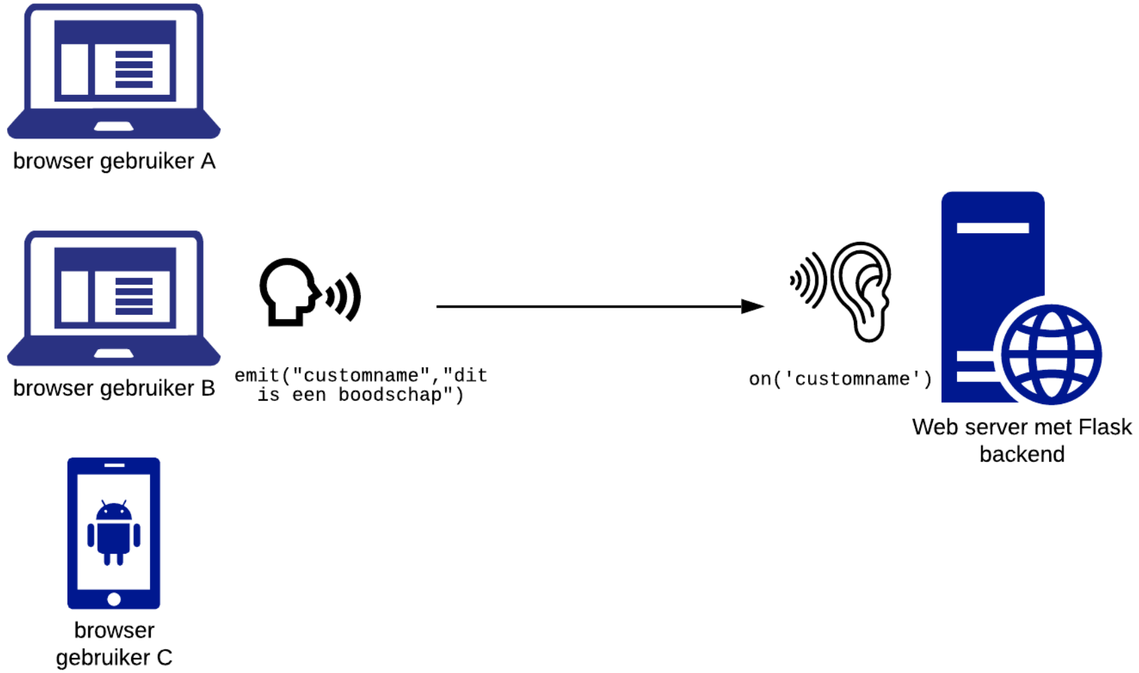
→ Easy and default to the `message` listener

Emit: Send an object (payload) with a custom **eventname**

→ More variations of objects, such as Images etc.


browser gebruiker A
browser gebruiker B
send("dit is een boodschap")
on('message')
Web server met Flask backend
browser gebruiker C

of


browser gebruiker A
browser gebruiker B
emit("customname","dit is een boodschap")
on('customname')
Web server met Flask backend
browser gebruiker C

Socket.io – How it works

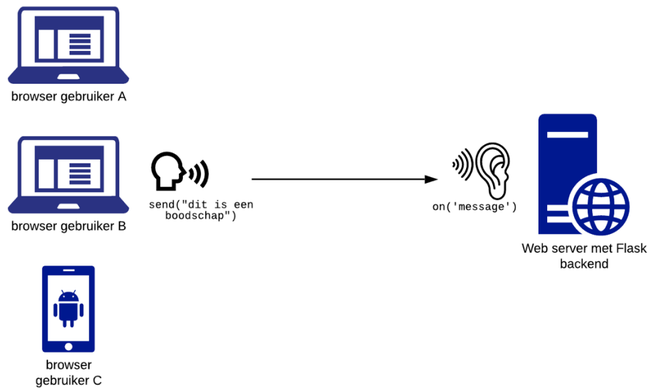
socket.io

Sending from Python to the default message listener

```
sio.send(message)
```

Catching in Python

```
@sio.on("message")
async def message(sid, data):
        print(f"Message from client: {data}")
```



Socket.io – How it works

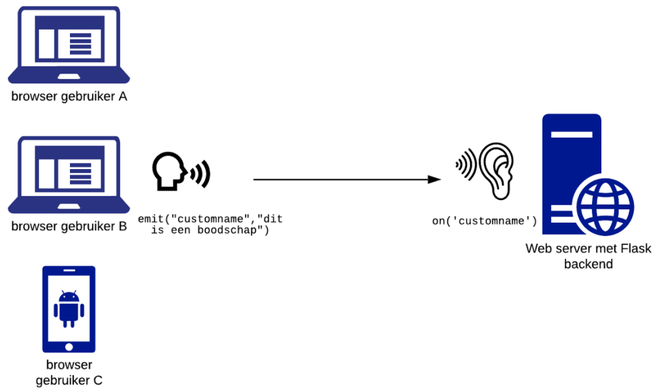
socket.io

Sending from Python to a custom event listener

```
sio.emit("custom", message)
```

Catching in Python

```
@sio.on("custom")
async def custom(sid, data):
        print(f"Custom from client: {data}")
```



Socket.io – How it works

socket.io

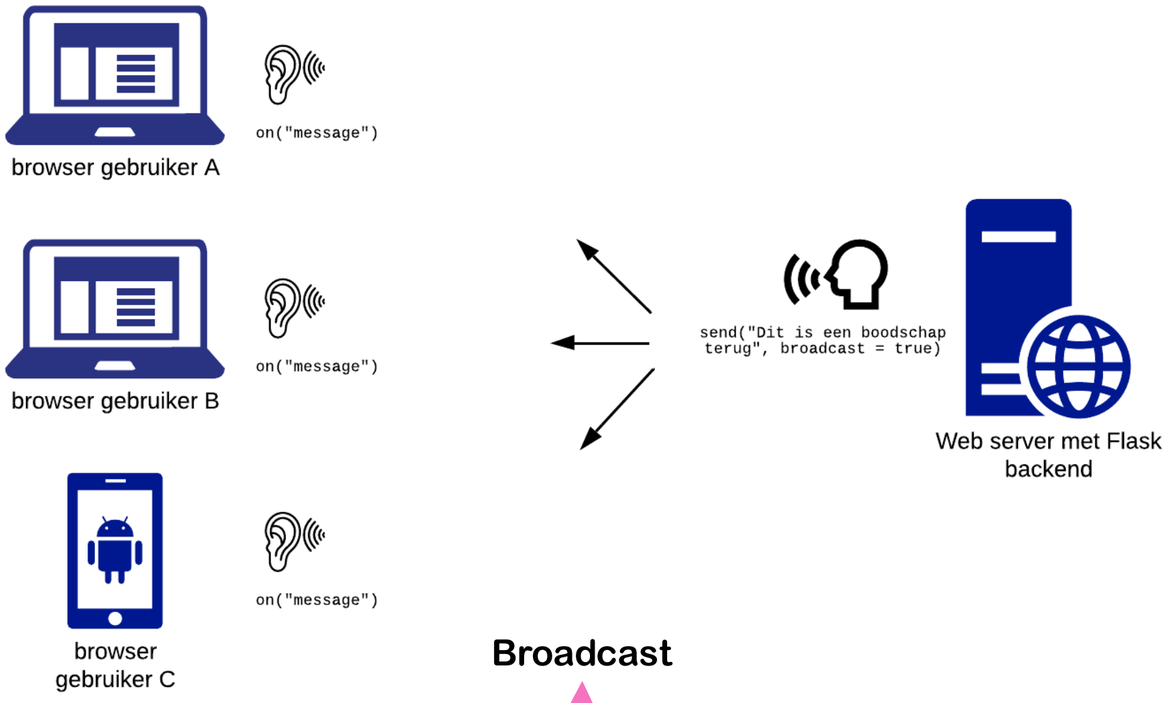
From now on, we can send messages from **client** to **server**

Cliënt → server is always a 1-on-1 connection.

Sending messages from **server** to **client:**

- Send it to the client that just send the message
- Send a message to **all** connected clients → Broadcast
  Can be done with both a **send** and **emit** option.

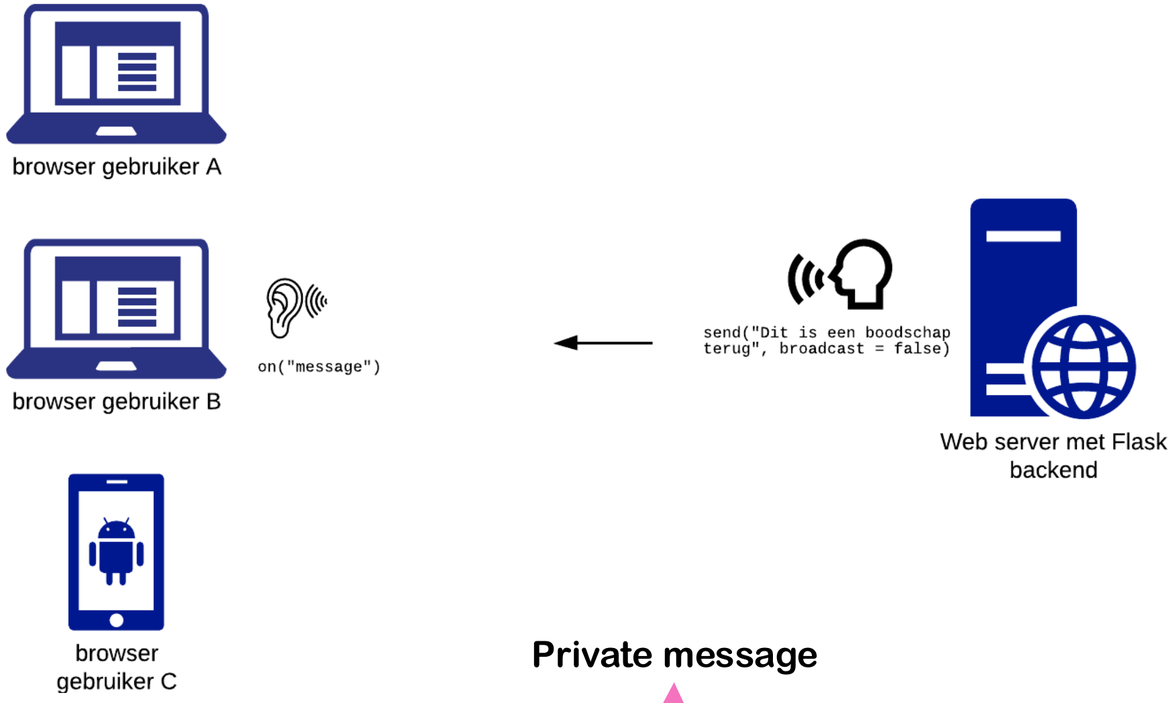
Server → cliënt is either 1-on-1 connection or 1-to-many connection.



Broadcast

sio.send(message, to=None)

of



Private message

sio.send(message, to="room1")

sio.send(message, to="client_id")

Socket.io – How it works

socket.io

We now send a message from **server** to **client(s)** whenever there is an incoming message.

```
@sio.on("message")
async def message(sid, data):
    await sio.send(data, skip_sid=sid)
    print(f"Message from client: {data}")
```

All connected clients, <u>except for the client that sent it</u>, will now get the ”data” message.

They will be able to catch this using ”sio.on(‘message’)”

Socket.io – Good2Know

socket.io

When you create a connection from a client to a server, a `connect` event gets triggered. You can catch this on your server using code like this.

```
@sio.on("connect")
async def connect(sid, env):
        print(f"New Client connected with this ID : {sid}")
        await sio.send(f"Client with ID {sid} connected")
```

As soon as the connection is created, the message will be sent to the connected client.

If you wish to, you could also perform a broadcast to all other clients.

Socket.io – Room management

Rooms can be created so that clients can communicate in the same room

```
@sio.on("join")
async def join(sid, room):
        sio.enter_room(sid, room)
    await sio.send(f"Client {sid} has joined room {room}", room=room)
    print(f"Client {sid} joined room {room}")
```

In order to communicate in a room, simply use the sio.send(message, to="room1") and fill in the room name in the `to` property

↓	message	Client 2cb1yRxKb_rtVEsnAAAB has joined room room1
↑	join	room1

howest hogeschool

Socket.io – Room management

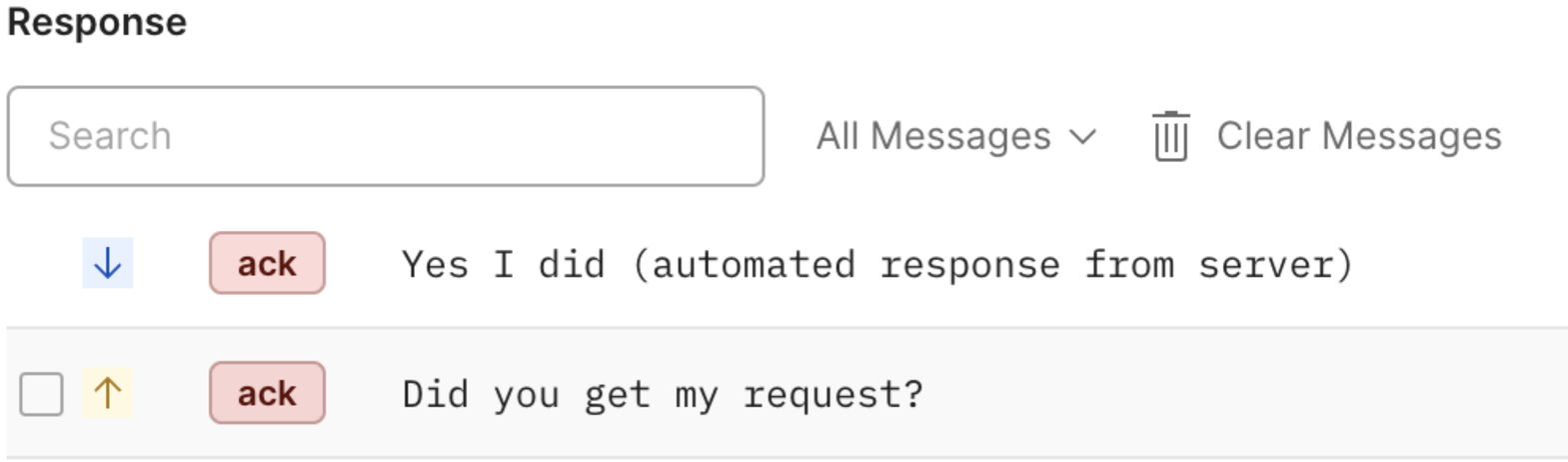
Serverside, Rooms can also be closed close_room()

Clientside, you can leave a room using leave_room()

Socket.io – Acknowledgment

- A client can let the server (and vice-versa) know it has received the message by sending an **ACK** or Acknowledge
- A server can then fire the `callback` method in the `emit` or `send` option.

```
@sio.on("ack")
async def ack(sid, data):
    print(f"Question from Client: {data}")
    return "Yes I did (automated response from server)"
```



Socket.io – Acknowledgment with Callback

When adding a callback method in the server, and sending the ID in the `to`-parameter, we can execute that callback when the Acknowledgement has proceeded from the client.

```
def callback_method(callback_ack: Any):
    print(callback_ack)

# HTTP POST from FastAPI to test
@app.post("/message_with_callback")
async def send_message_with_callback(message: str, id: str):
    await sio.emit("ack", message, callback= callback_method, to=id)
    return "Message sent, awaiting Acknowledgement... Check the logs for that!"
```

```
@sio.on("ack")
async def ack(data):
print('Received ack:', data)
return "Ack received"
```

howest
hogeschool

SocketIO in Gradio

SocketIO in Gradio

- To work with event-listeners in Gradio, we use a Message Queue from Asyncio

```
message_queue = asyncio.Queue()
```

- We can then **put** and **read** items in the queue in a continuous loop to update our Gradio components

```
@sio.on("message")
async def message(data):
    print('Received message:', data)
    # Add message to queue
    await message_queue.put(data)
```

```
async def endless_loop():
    await start_connection()
    while True:
        msg = await message_queue.get()
        yield msg
```

```
btn = gr.Button("start")
text = gr.Textbox(label="Messages", lines=7, interactive=True)
btn.click(endless_loop, [], text)
```