



WebSockets with PHP

By Caetano Burjack

Introduction	3
Requirements	3
Process Flow	3
Hands-on	4
Goal	4
Installation	4
The Chat class	5
The WebChat	7
Pusher	8
Visual Interface	10
Script.js	11
Trying On	16
Conclusion	17
Quiz	17
Extra Mile	18

Introduction

▶ What are WebSockets? How is it different from HTTP?

WebSockets are a bi-directional, full-duplex, persistent connection from a web browser to a server. Once a WebSocket connection is established the connection stays open until the client or server decides to close this connection. With this open connection, the client or server can send a message at any given time to the other. This makes web programming entirely event-driven, not (just) user-initiated. It is stateful. As well, at this time, a single running server application is aware of all connections, allowing you to communicate with any number of open connections at any given time.

Requirements

In this course we will use the Ratchet library that allows us to create an asynchronous WebSocket server, and for that, it uses the ReactPHP event loop. Another option would be to use Swoole's WebSocket server. But the good thing about using ReactPHP is that we don't need to install any specific extensions in our PHP installation.

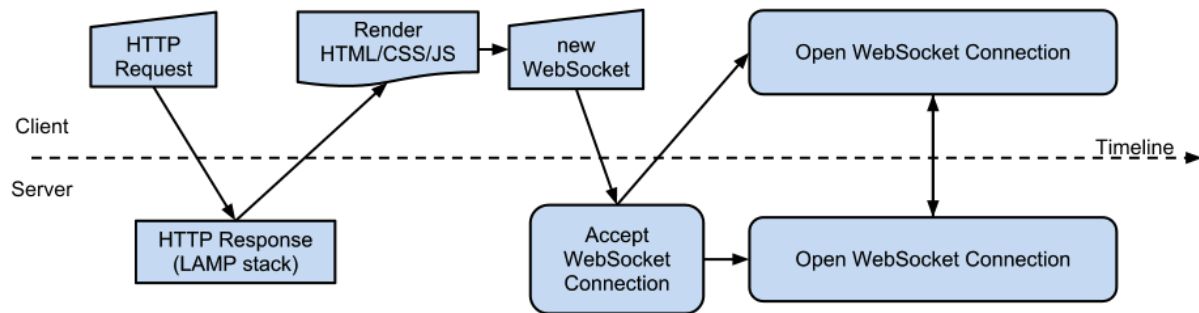
The only requirement at this point will be Composer.

Process Flow

▶ Realtime PHP Using Websockets - Jeff Kolesnikowicz @jkolez

If you've been working with PHP for any length of time you're probably used to writing short-lived scripts. Each web page loaded on the client's side will, on the server side, launch a new PHP script, loads resources (such as a database connection), execute your code, close the resources, and send the output (HTML) back to the client, and close the connection. This is the tried and true nature of the HTTP protocol over the past several decades.

WebSockets using PHP in Ratchet are a little different. Only one script is executed and connections remain open. You may need to do things a little differently; you can't use any global variables as the context of the running process is not confined to a single connection. Below, the diagram illustrates the order of execution beginning with a client requesting a web page on your site:



As you can see, once the web page has been loaded a WebSocket connection is made back to your Ratchet application, where if everything goes correctly, a connection remains open where either the server or client can send data to the other one at any given time.

Hands-on

Goal

The goal of this application is to write a simple Chat application. Chats in event-driven programming are the "Hello World!" of applications. The chat will accept all incoming messages and deliver that messages to all other connections.

Considering our project will be done using Composer and PSR-4, I hope you be familiar with these concepts. But don't worry, if you are not familiar, see the installation on the official site, available at <https://getcomposer.org/download/>.

Installation

The best (and currently only) way to install Ratchet is by using Composer. Simply add Ratchet by running this:

```
composer require cboden/ratchet
```

We're going to hold everything in the *MyApp* namespace. Your composer file should look something like this:

```
{
  "autoload": {
    "psr-4": {
      "MyApp\\": "src"
```

```

    }
  },
  "require": {
    "cboden/ratchet": "0.4.*"
  }
}

```

The Chat class

We'll start off by creating a class. This class will be our chat "application". This basic application will listen for 4 events:

- *onOpen* - Called when a new client has Connected
- *onMessage* - Called when a message is received by a *Connection*
- *onClose* - Called when a *Connection* is closed
- *onError* - Called when an error occurs on a *Connection*

Given those triggers, the class will implement the `MessageComponentInterface`:

The `Chat.php` class will look like this:

```

<?php
//src/Chat.php
namespace MyApp;

use Ratchet\MessageComponentInterface;
use Ratchet\ConnectionInterface;

class Chat implements MessageComponentInterface
{
    protected $clients;

    public function __construct()
    {
        $this->clients = new \SplObjectStorage;
    }

    public function onOpen(ConnectionInterface $conn)
    {
        // Store the new connection to send messages to later
        $this->clients->attach($conn);

        echo "New connection! ({ $conn->resourceId })\n";
    }

    public function onMessage(ConnectionInterface $from, $msg)
    {
        $numRecv = count($this->clients) - 1;
        echo sprintf('Connection %d sending message "%s" to %d other
connection%s' . "\n"
            , $from->resourceId, $msg, $numRecv, $numRecv == 1 ? '' : 's');
    }
}

```

```

        foreach ($this->clients as $client) {
            if ($from !== $client) {
                // The sender is not the receiver, send to each client connected
                $client->send($msg);
            }
        }
    }

    public function onClose(ConnectionInterface $conn)
    {
        // The connection is closed, remove it, as we can no longer send it
        messages
        $this->clients->detach($conn);

        echo "Connection {$conn->resourceId} has disconnected\n";
    }

    public function onError(ConnectionInterface $conn, \Exception $e)
    {
        echo "An error has occurred: {$e->getMessage()}\n";

        $conn->close();
    }
}

```

Next, we're going to piece together our shell script. This is the script/file we will call from the command line to launch our application.

```

<?php
// bin/chat-server.php
use Ratchet\Server\IoServer;
use MyApp\Chat;

require dirname(__DIR__) . '/vendor/autoload.php';

$server = IoServer::factory(
    new Chat(),
    8080
);

$server->run();

```

Above, you'll see we create an I/O (Input/Output) server class. It stores all the established connections, mediates data sent between each client and our *Chat* application, and catches errors.

The new instance of the *Chat* class then wraps the I/O Server class. Finally, we tell the server to enter an event loop, listening for any incoming requests on port 8080.

Save this script as `/bin/chat-server.php`. Now, we can run it with the following command in your terminal:

```
php bin/chat-server.php
```

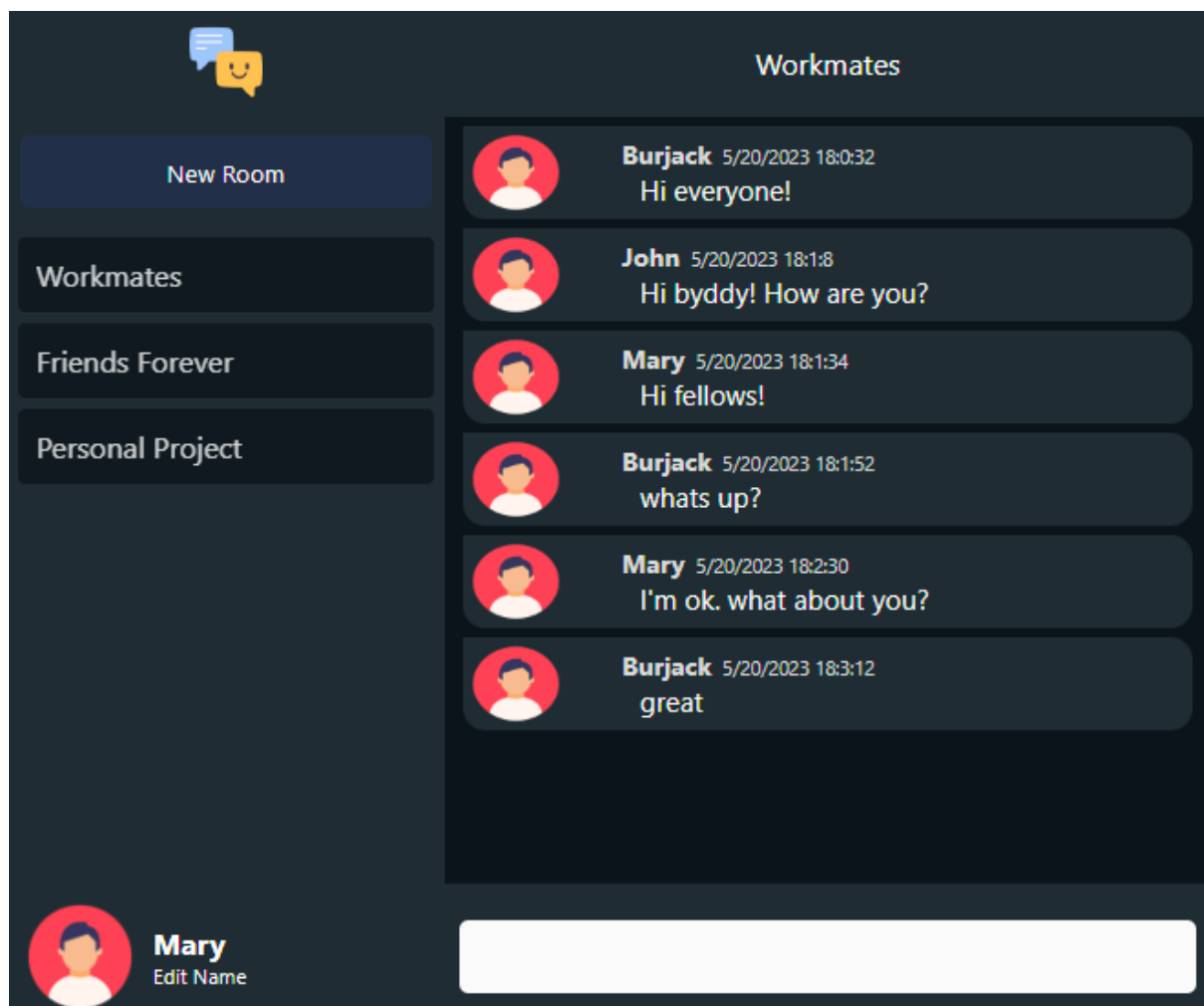
The script should now execute, taking possession of your terminal. You can cancel the script, as we're not quite finished yet.

You can open two new terminals and in each of the telnet windows, type a message ("Hello World!") and see it appear in the other!

As you could see, the focus of our code was creating a long-lived application where users interacted with each other entirely over WebSockets with no persistence.

Now, we are going to see that in action in the context of a web application. Considering that the goal of this course is teaching web sockets, we are going to integrate this code into a previously created web chat.

The WebChat



From now on, we are going to make some changes so that our web socket application integrates with our chat page.

The first big change is replacing the Chat.php class we created earlier with a new one, which will do the same things, but now in the context we need.

Pusher

This class is responsible to trigger all the events, like subscribing, unsubscribing, opening, closing, publishing, etc.

```
<?php

namespace MyApp;

use Ratchet\ConnectionInterface;
use Ratchet\Wamp\WampServerInterface;

class Pusher implements WampServerInterface
{
    /**
     * A lookup of all the topics clients have subscribed to
     */
    protected array $subscribedTopics = array();

    public function onSubscribe(ConnectionInterface $conn, $topic)
    {
        $this->subscribedTopics[$topic->getId()] = $topic;
        if (isset($this->messages[$topic->getId()])) {
            //print_r($this->messages);
            $json = '[' . $this->messages[$topic->getId()] . ']';

            //send the history only for the user who just connected/subscribed
            $conn->event($topic, $json);
        }
    }

    public function onUnSubscribe(ConnectionInterface $conn, $topic)
    {
    }

    public function onOpen(ConnectionInterface $conn)
    {
    }

    public function onClose(ConnectionInterface $conn)
    {
    }

    public function onCall(ConnectionInterface $conn, $id, $topic, array $params)
    {
        // In this application if clients send data it's because the user hacked
    }
}
```



```

around in console
    $conn->callError($id, $topic, 'You are not allowed to make
calls')->close();
}

    public function onPublish(ConnectionInterface $conn, $topic, $event, array
$exclude, array $eligible)
    {
        if (!isset($this->messages[$topic->getId()])) {
            $this->messages[$topic->getId()] = json_encode($event);
        } else {
            $this->messages[$topic->getId()] .= ', ' . json_encode($event);
        }

        //trigger a message for everyone on the same topic
        $topic->broadcast($event);
    }

    public function onError(ConnectionInterface $conn, \Exception $e)
    {
    }
}

```

We are creating the concept of rooms, in order to have private rooms. So we created the array `subscribedTopics` to store our rooms.

Now we need to update our chat-server.php file in order to use our class Pusher.

```

<?php
// bin/chat-server.php
require dirname(__DIR__) . '/vendor/autoload.php';

$loop = React\EventLoop\Factory::create();
$pusher = new MyApp\Pusher;

// Set up our WebSocket server for clients wanting real-time updates
$webSock = new React\Socket\Server('0.0.0.0:8080', $loop);
// Binding to 0.0.0.0 means remotes can connect
$webServer = new Ratchet\Server\IoServer(
    new Ratchet\Http\HttpServer(
        new Ratchet\WebSocket\WsServer(
            new Ratchet\Wamp\WampServer(
                $pusher
            )
        )
    ),
    $webSock
);

$loop->run();

```

Visual Interface

Now, let's see the code of the frontend of our webchat.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset='utf-8'>
  <title>Chat with WebSocket</title>
  <meta name='viewport' content='width=device-width, initial-scale=1'>
  <link rel="stylesheet" type="text/css" href="assets/css/style.css?v=1">
</head>
<div id="container">
  <div id="leftColumn">
    <div id="logoDiv">
      
    </div>
    <form id="newRoomForm" style="display: none">
      <input type="text" id="roomName" name="roomName" required="required"
placeholder="Room Name">
      <input type="text" id="userNameOnRoomForm" name="userName"
placeholder="User Name" style="display: none">
      <button class="defaultBtn" style="display: none"
id="btnCreateRoom">Create Room</button>
    </form>
    <button class="defaultBtn" onclick="showNewRoomForm()" id="newRoom">New
Room</button>
    <div id="roomsDiv">

  </div>
    <div id="userNameDiv">
      
      <div id="userName">
        <form id="changeNameForm" style="display: none">
          <input type="text" id="userNameInput" name="userName"
placeholder="User Name">
          <button class="defaultBtn" id="btnCreateRoom" style="display:
none">Save User Name</button>
        </form>
        <span id="userNameSpan"></span>
        <span id="editUserName" onclick="showChangeNameForm()">Edit
Name</span>
      </div>
    </div>
  </div>

  <div id="rightColumn">
    <div id="roomTitleDiv">
    </div>

    <div id="talkDiv">
    </div>

    <div id="messageInputDiv">
```

```

        <form id="messageSendingForm">
            <input type="text" class="messageInput" autocomplete="false"
id="message"/>
        </form>
    </div>
</div>

</div>
<script src="assets/js/autobahn.js"></script>
<script src="assets/js/script.js"></script>
</body>
</html>

```

As you can see in the code above, our page is calling one CSS file (style.css) and two JS files (autobahn.js and script.js).

Since the focus here is not CSS, we are not going to talk about it. You can get it in the repository of this project, available at: https://github.com/caetanoburjack/chat_websockets

The autobahn.js is a recommendation of the ratchet documentation but it is not required. However, we are going to use it in our project.

In our project, we will use it locally. For this, I will leave the content of this file in the repository, which can be accessed through the aforementioned link.

Now, the most important of those three files, the script.js is the point of our project, responsible to keep the communication between the front end and the WebSocket on the back end.

Script.js

The content of this file should look like this:

```

// assets/js/script.js
const messageSendingForm =
document.getElementById('messageSendingForm');
const newRoomForm = document.getElementById('newRoomForm');
const changeNameForm = document.getElementById('changeNameForm');
const talkDiv = document.getElementById('talkDiv');
const roomsDiv = document.getElementById('roomsDiv');
const roomTitleDiv = document.getElementById('roomTitleDiv');
const userNameSpan = document.getElementById('userNameSpan');
let currentRoom = localStorage.getItem('chat_current_room');
let conn;
let conn_status = false;

```

```

let storedRooms = localStorage.getItem('chat_rooms');
let storedUserName = localStorage.getItem('chat_username');

window.onload = initizile;

function initizile() {
  showRooms();
  showUserName();
  if (currentRoom) {
    activateRoom(currentRoom);
  } else if (storedRooms.length > 0) {
    storedRooms = JSON.parse(storedRooms);
    activateRoom(storedRooms[0]);
  } else {
    activateRoom('room');
  }
}

function showUserName(userName = null) {
  userNameSpan.innerHTML = userName ?? storedUserName;
}

newRoomForm.addEventListener('submit', function (e) {
  e.preventDefault();
  let roomName = document.getElementById('roomName').value;
  let userName = document.getElementById('userNameOnRoomForm').value;
  createRoom(roomName);
  if (userName) {
    saveUserName(userName);
  }
  document.getElementById('roomName').value = '';
  newRoomForm.style.display = 'none';
  document.getElementById('newRoom').style.display = 'inline-block';
  localStorage.setItem('chat_current_room', roomName);
  connect();
});

changeNameForm.addEventListener('submit', async function (e) {
  e.preventDefault();

  let userName = document.getElementById('userNameInput').value;
  await saveUserName(userName);
  document.getElementById('userNameInput').value = '';
  changeNameForm.style.display = 'none';
  document.getElementById('userNameSpan').style.display = 'block';
  document.getElementById('editUserName').style.display = 'block';
});

messageSendingForm.addEventListener('submit', function (e) {

```

```

e.preventDefault();

let name = localStorage.getItem('chat_username');
let room = localStorage.getItem('chat_current_room');
let message = document.getElementById('message').value;

let $msgTime;
if (name && message) {

    $msgTime = getRightTime();
    let data = {
        'name': name, 'message': message, 'msgTime': $msgTime
    };

    conn.publish(room, data);
}
document.getElementById('message').value = '';
document.getElementById('message').focus();
});

function getRightTime() {
    let msgTime = new Date();
    let day = msgTime.getDate();           // 1-31
    let mes = msgTime.getMonth();          // 0-11 (zero=january)
    let year4 = msgTime.getFullYear();     // 4 digits
    let hour = msgTime.getHours();         // 0-23
    let min = msgTime.getMinutes();        // 0-59
    let sec = msgTime.getSeconds();        // 0-59

    let str_data = (mes + 1) + '/' + day + '/' + year4;
    let str_hour = hour + ':' + min + ':' + sec;

    return str_data + ' ' + str_hour;
}

function createRoom(room) {
    let rooms = localStorage.getItem('chat_rooms');
    if (rooms) {
        let roomsArray = JSON.parse(rooms);
        if (roomsArray.indexOf(room) === -1) {
            roomsArray.push(room);
            let roomsArrayJson = JSON.stringify(roomsArray);
            localStorage.setItem('chat_rooms', roomsArrayJson);
        }
    } else {
        let roomsArray = [room];
        let roomsArrayJson = JSON.stringify(roomsArray);
        localStorage.setItem('chat_rooms', roomsArrayJson);
    }
}

```

```

    makeCurrentRoom(room);
}

function makeCurrentRoom(room) {
    localStorage.setItem('chat_current_room', room);
}

function saveUserName(userName) {
    localStorage.setItem('chat_username', userName);
    showUserName(userName);
}

function connect() {
    let room = localStorage.getItem('chat_current_room') ?
localStorage.getItem('chat_current_room') : 'room';
    if (conn_status) {
        conn.close();
        conn_status = false;
        talkDiv.innerHTML = '';
    }
    conn = new ab.Session('ws://localhost:8080', function () { //Using
autobahn in order to start a websocket session.
        conn_status = true;
        makeCurrentRoom(room);
        conn.subscribe(room, function (topic, data) {
            if (typeof data === 'string') {
                data = JSON.parse(data);

                for (let i = 0; i < data.length; i++) {
                    showMessages(data[i]);
                }
            } else {
                showMessages(data);
            }
        });
    }, function () {
        console.warn('WebSocket connection closed');
    }, {'skipSubprotocolCheck': true});
    showRooms();
    showRoomName(room);
}

function showRooms() {
    let storedRooms = localStorage.getItem('chat_rooms');
    if (storedRooms) {
        storedRooms = JSON.parse(storedRooms);
        roomsDiv.innerHTML = '';
        for (let i = 0; i < storedRooms.length; i++) {
            let p = document.createElement('p');

```

```

        p.setAttribute('class', 'each_room');
        p.setAttribute('onclick', 'activateRoom("' + storedRooms[i]
+ '")');
        p.textContent = storedRooms[i];
        roomsDiv.appendChild(p);
    }
}

function showRoomName(room) {
    roomTitleDiv.innerHTML = room;
}

//Printar Mensagens na Tela
function showMessages(data) {
    let img_src = "assets/img/user.png";

    let div = document.createElement('div');
    div.setAttribute('class', 'me');

    let img = document.createElement('img');
    img.setAttribute('src', img_src);

    let div_txt = document.createElement('div');
    div_txt.setAttribute('class', 'text');

    let span = document.createElement('span');
    span.setAttribute('class', 'msgTime');
    span.textContent = data.msgTime;

    let h5 = document.createElement('h5');
    h5.textContent = data.name;
    h5.appendChild(span);

    let p = document.createElement('p');
    p.textContent = data.message;

    div_txt.appendChild(h5);
    div_txt.appendChild(p);

    div.appendChild(img);
    div.appendChild(div_txt);

    talkDiv.appendChild(div);
    talkDiv.scrollTop = talkDiv.scrollHeight;
}

function showNewRoomForm() {
    if (!localStorage.getItem('chat_username')) {

```

```

        document.getElementById('userNameOnRoomForm').style.display =
        'inline-block'
    }
    document.getElementById('newRoom').style.display = 'none';
    newRoomForm.style.display = 'flex';
    document.getElementById('roomName').focus();
}

function showChangeNameForm() {
    changeNameForm.style.display = 'block';
    document.getElementById('userNameSpan').style.display = 'none';
    document.getElementById('editUserName').style.display = 'none';
    document.getElementById('userNameInput').focus();
}

function activateRoom(room) {
    localStorage.setItem('chat_current_room', room);
    talkDiv.scrollTop = talkDiv.scrollHeight;
    connect();
}

```

There are some important things you need to pay attention to at this point. As you can see in the code above, some data is being stored in local storage, so, you need to keep it in mind when you are trying your application. That means you will need to open two different browsers to try it.

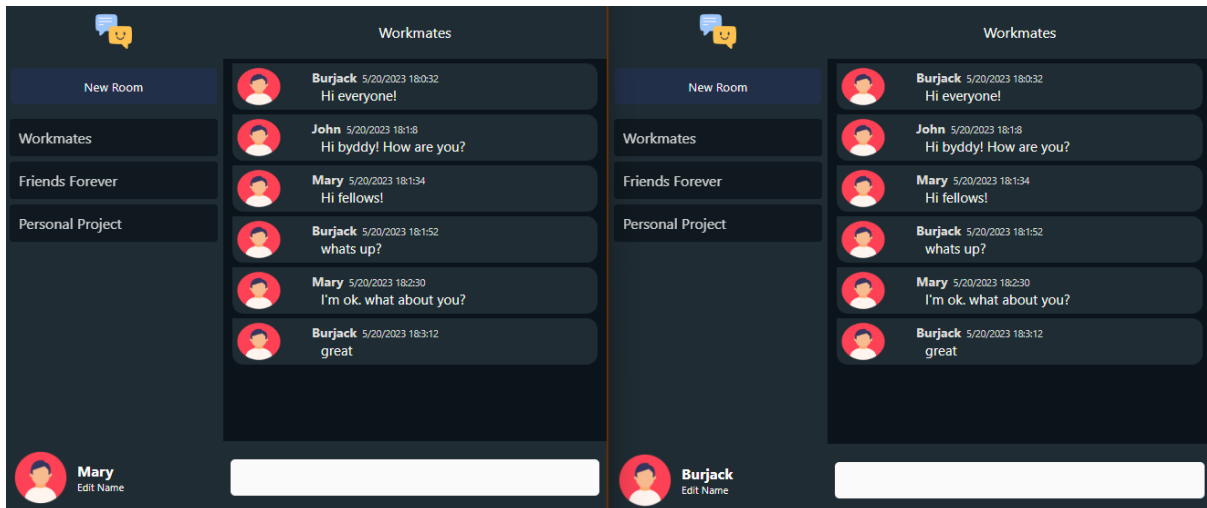
Another important point here is the function connect, responsible to connect our chat interface to the WebSocket. I put a comment in the line responsible to start a WebSocket session using the autobahn.js.

Trying On

At this point, you already have a completely functional webchat using WebSockets.

All you need to do is run the command to start the chat-server.php.

After that, open two different browsers, to try your final project. You will have something like this:



Conclusion

If you've come this far, it's expected that you've learned the fundamentals of WebSockets. The codebase for this project is available at:

https://github.com/caetanoburjack/chat_websockets.

Now you are able to understand the potential of sockets in the context of web applications and have the necessary knowledge to apply it in a practical way in real projects.

Always remember to resort to official documentation, as many points that we do not cover here are described there.

In the end, if you want to go the extra mile, I'll propose a challenge.

See you on the next course.

Quiz

1 - What is the command to add the Ratchet lib in our project?

- a) composer require ratchet
- b) composer install ratchet
- c) composer add ratchet
- d) **composer require cboden/ratchet**

2 - Which of the libs below was used in our project?

- a) **Ratchet.**
- b) Soketi.
- c) Pawl.
- d) OpenSwoole.


3 - Which of the PSRs below was explicitly used in our project?


- a) **PSR-4.**
- b) PSR-7.
- c) PSR-10
- d) PSR-11


Extra Mile


Now that you know the basics of WebSockets, you can go the extra mile trying to use another famous library focused on this type of technology, the OpenSwoole.

To help you with this task, here is a set of interesting videos teaching how to use it.

 16 minutes of OpenSwoole - WebSocket - Part 1

 57 minutes of OpenSwoole - WebSocket - Part 2

 34 minutes of OpenSwoole - WebSocket - Part 3

 15 minutes of OpenSwoole - WebSocket - Part 4