How to Quickly Build a Chat App with Ratchet

In this tutorial, we'll be taking a look at Ratchet, a PHP library for working with WebSockets. Let's start by defining what WebSockets are. MDN says:

WebSockets is an advanced technology that makes it possible to open an interactive communication session between the user's browser and a server. With this API, you can send messages to a server and receive event-driven responses without having to poll the server for a reply.

WebSockets allow us to write applications that can pass data from the browser to the server and vice-versa in real-time.

Setup

First, let's install Ratchet using Composer:

```
composer require cboden/ratchet
```

Building the App

Now we're ready to build the app. Create a `Chat.php` file under the `class/ChatApp` directory. This would be a class under the `ChatApp` namespace, and it would use use Ratchet's `MessageComponentInterface` and `ConnectionInterface`.
The `MessageComponentInterface` is used as the basic building block for Ratchet applications, while the `ConnectionInterface` represents the connection to the application.

```
<?php

namespace ChatApp;




use Ratchet\MessageComponentInterface;

use Ratchet\ConnectionInterface;
```

Have the class implement the `MessageComponentInterface`. This contains the blueprint for the methods that we need to implement such as `onOpen`, `onClose` and `onMessage`.

```
class Chat implements MessageComponentInterface {




}
```

Inside the class, we declare a variable called `$clients`. This is where we will store a list of currently connected clients in our chat app later on. From the constructor, you will see that we're using `SplObjectStorage`. This provides a way for us to store objects. In this case, the object that we need to store is the connection object for each client.

```
protected $clients;
```

```
public function __construct() {

    $this->clients = new \SplObjectStorage;

}
```

Next, we implement the `onOpen` method. This method is called every time a new connection is opened in the browser. What this does is store the new connection object using the `attach` method. We also output that someone has connected as a means of testing if the `onOpen` method is working correctly.

```
public function onOpen(ConnectionInterface $conn) {

    //store the new connection

    $this->clients->attach($conn);



    echo "someone connected\n";

}
```

Next is the `onMessage` method. This method is called every time a message is sent by a specific client in the browser. The connection object of the client who sent the message, as well as the actual message, are passed along as an argument every time this method is called. All we do is loop through all the currently connected clients and send the message to them. In the code below, we're checking if the client in the current iteration of the loop is the one who sent the message. We don't want to send the same message to the person who sent it.

```
public function onMessage(ConnectionInterface $from, $msg) {

    //send the message to all the other clients except the one who
sent.

    foreach ($this->clients as $client) {
```

```
        if ($from !== $client) {

            $client->send($msg);

        }

    }

}
```

Next is the `onClose` method. As the name suggests, this method is called every time a client closes the WebSocket connection from the browser. This method is triggered when a user refreshes the browser tab or closes it entirely. All we have to do is call the `detach` method on the list of clients and pass in the connection as an argument. This deletes that specific connection.

```
public function onClose(ConnectionInterface $conn) {

    $this->clients->detach($conn);

    echo "someone has disconnected";

}
```

Lastly, we have the `onError` method which is fired every time there's a connection error. When this happens, we output the error that occurred and then call the `close`method in the connection to close it.

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

    echo "An error has occurred: {$e->getMessage()}\n";

    $conn->close();

}
```

Now we're ready to create the entry file which will utilize the file we've just created. We will be running it from the command line. Call it `cmd.php`, save it in the root of your working directory, then add the following code.

```
<?php

require 'vendor/autoload.php';



use Ratchet\Server\IoServer;

use Ratchet\Http\HttpServer;

use Ratchet\WebSocket\WsServer;

use ChatApp\Chat;



$server = IoServer::factory(

    new HttpServer(

        new WsServer(

            new Chat()

        )

    ),

    8080

);
```

```
$server->run();
```

What this file does is create a new WebSocket server that runs on port 8080. We will connect to this server later on from the client side.

Before we move on, let's break down the file. First we include the autoload file so that we can use the different Ratchet components from our file.

```
require 'vendor/autoload.php';
```

Next, we specify which specific components of Ratchet we need. For this chat application we would need the `IoServer`, `HttpServer` and the `WsServer`.

```
use Ratchet\Server\IoServer;

use Ratchet\Http\HttpServer;

use Ratchet\WebSocket\WsServer;
```

Here's a brief description of each of the components:

- The `IoServer` allows us to receive, read and write, and close connections, as well as handle errors that we might get. This provides basic server functionality as well, so we can use it to create a new server instance.
- The `HttpServer` allows us to parse incoming HTTP requests. This component is used every time a user connects to the server or a user sends a message.
- The `WsServer` is the WebSocket server. This allows us to talk to browsers which implement the WebSocket API. Most modern browsers already implement WebSockets so there won't be a problem if you're not planning to support old browsers. If you do, then you can take a look at adding a Flash Policy to your application.

Going back to the `cmd.php` file, we also use the `Chat` class that we created earlier.

```
use ChatApp\Chat;
```

Once that's done, we can create the WebSockets server. For that, we would need to call the `factory` method on the `IoServer` component and then pass in a new instance of the `HttpServer`. This new `HttpServer` instance then accepts a new `WsServer` instance. Finally, we pass in a new instance of the `Chat` class to the `WsServer`. You can see which specific server component wraps what in the documentation.

```
$server = IoServer::factory(

    new HttpServer(

        new WsServer(

            new Chat()

        )

    ),

    8080

);
```

We run the server by calling the `run` method.

```
$server->run();
```

At this point you can start running the server from the terminal:

```
php cmd.php
```

HTML

Next, we move on to the client side. Create an `index.html` file at the root of your working directory and then add the following code:

```
<!DOCTYPE html>


<html lang="en">
```

```
<head>

    <meta charset="UTF-8">

    <title>chatapp</title>

    <script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.4/jquery.min.js
"></script>

    <script
src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/3.0.3/handle
bars.min.js"></script>

    <script
src="http://cdnjs.cloudflare.com/ajax/libs/moment.js/2.10.2/moment.min
.js"></script>




    <link rel="stylesheet" href="css/style.css">

</head>

<body>

    <div id="wrapper">

        <div id="user-container">

            <label for="user">What's your name?</label>

            <input type="text" id="user" name="user">

            <button type="button" id="join-chat">Join Chat</button>
```

```
        </div>


        <div id="main-container" class="hidden">
            <button type="button" id="leave-room">Leave</button>
            <div id="messages">


            </div>


            <div id="msg-container">
                <input type="text" id="msg" name="msg">
                <button type="button" id="send-msg">Send</button>
            </div>
        </div>


    </div>


    <script id="messages-template" type="text/x-handlebars-template">
```

```
        {{#each messages}}

        <div class="msg">

            <div class="time">{{time}}</div>

            <div class="details">

                <span class="user">{{user}}</span>: <span
class="text">{{text}}</span>

            </div>

        </div>

        {{/each}}

    </script>



    <script src="js/main.js"></script>

</body>

</html>
```

For the client side, we will use jQuery for listening to click events and manipulating the DOM. Handlebars will handle templating and Moment will display the time at which the messages were sent.

CSS

For our stylesheet, we stick with something minimal. Just the bare minimum styling so it looks decent enough. Here is the `css/style.css` file:

```
.hidden {
```

```
    display: none;
}

#wrapper {
    width: 800px;
    margin: 0 auto;
}

#leave-room {
    margin-bottom: 10px;
    float: right;
}

#user-container {
    width: 500px;
    margin: 0 auto;
    text-align: center;
```

```
}



#main-container {

    width: 500px;

    margin: 0 auto;

}



#messages {

    height: 300px;

    width: 500px;

    border: 1px solid #ccc;

    padding: 20px;

    text-align: left;

    overflow-y: scroll;

}



#msg-container {
```

```
    padding: 20px;
}

#msg {
    width: 400px;
}

.user {
    font-weight: bold;
}

.msg {
    margin-bottom: 10px;
    overflow: hidden;
}

.time {
```

```
    float: right;

    color: #939393;

    font-size: 13px;

}



.details {

    margin-top: 20px;

}
```

JS

And for our JavaScript file (`js/main.js`), we have the following:

```
(function(){



    var user;

    var messages = [];



    var messages_template = Handlebars.compile($('#messages-
template').html());



    function updateMessages(msg){
```

```
        messages.push(msg);

        var messages_html = messages_template({'messages': messages});

        $('#messages').html(messages_html);

        $("#messages").animate({ scrollTop:
$('#messages')[0].scrollHeight}, 1000);

    }



    var conn = new WebSocket('ws://localhost:8080');

    conn.onopen = function(e) {

        console.log("Connection established!");

    };



    conn.onmessage = function(e) {

        var msg = JSON.parse(e.data);

        updateMessages(msg);

    };
```

```
    $('#join-chat').click(function(){

        user = $('#user').val();

        $('#user-container').addClass('hidden');

        $('#main-container').removeClass('hidden');



        var msg = {

            'user': user,

            'text': user + ' entered the room',

            'time': moment().format('hh:mm a')

        };



        updateMessages(msg);

        conn.send(JSON.stringify(msg));



        $('#user').val('');

    });
```

```
$('#send-msg').click(function(){

    var text = $('#msg').val();

    var msg = {

        'user': user,

        'text': text,

        'time': moment().format('hh:mm a')

    };

    updateMessages(msg);

    conn.send(JSON.stringify(msg));



    $('#msg').val('');

});




$('#leave-room').click(function(){
```

```
        var msg = {
            'user': user,
            'text': user + ' has left the room',
            'time': moment().format('hh:mm a')
        };
        updateMessages(msg);
        conn.send(JSON.stringify(msg));

        $('#messages').html('');
        messages = [];

        $('#main-container').addClass('hidden');
        $('#user-container').removeClass('hidden');

        conn.close();
    });
```

```
})();
```

Breaking the JavaScript file down. First, we wrap everything in an immediately-invoked function expression so we can be sure that the code we include wouldn't interact with any other JavaScript that we might include later on.

```
(function(){




})();
```

Initialize the user, messages, and the messages template. We will use the `user` variable to store the name of the user who entered the chat room. `messages` is for storing the current messages that were sent, and the `messages_template` is the handlebars template which we will use to build the HTML that shows all the messages.

```
var user;

var messages = [];




var messages_template = Handlebars.compile($('#messages-
template').html());
```

We then create the method that will be executed every time a message is sent or received by a specific user. What this does is push the message supplied as the argument into the messages array. It then builds the HTML for the messages using the messages array and the `messages_template`. Then it updates the HTML of the messages container. Lastly, we scroll down to the bottom of the messages container so the user sees the latest message.

```
function updateMessages(msg){

    messages.push(msg);

    var messages_html = messages_template({'messages': messages});
```

```
    $('#messages').html(messages_html);

    $("#messages").animate({ scrollTop:
$('#messages')[0].scrollHeight}, 1000);

}
```

Then we create a new WebSocket connection. This takes the URL of the WebSocket server as its argument. In this case, we're connecting to `ws://localhost:8080` which is the server that we ran earlier from the terminal.

```
var conn = new WebSocket('ws://localhost:8080');
```

After that, we listen for the `onopen` event in the WebSocket connection that we created. This doesn't really do anything relevant, we're merely using it to check if we have successfully connected to the server.

```
conn.onopen = function(e) {

    console.log("Connection established!");

};
```

Next, we listen for the `onmessage` event. This is triggered every time a new message is sent from any of the connected clients. When this event happens, we use the `JSON.parse` method to convert the JSON string that we got from the server into a JavaScript object. We then call the `updateMessages` method and pass in the result.

```
conn.onmessage = function(e) {

    var msg = JSON.parse(e.data);

    updateMessages(msg);

};
```

Joining the Chat Room

Joining the chat room happens when the user clicks on the 'join chat' button. What this does is assign the value entered by the user in the username field into the user variable. It then hides the user container which contains the username field and shows the main container which contains the messages and the field for entering a new message. After that, we construct a new message object. A message object has the `user`, `text` and `time` properties. We then call the `updateMessages` method so the message is appended into the messages box. Next, we call the `send` method on the WebSocket connection and pass in the JSON string representation of the new message. Finally, we empty the username field.

```
$('#join-chat').click(function(){

    user = $('#user').val();

    $('#user-container').addClass('hidden');

    $('#main-container').removeClass('hidden');



    var msg = {

        'user': user,

        'text': user + ' entered the room',

        'time': moment().format('hh:mm a')

    };



    updateMessages(msg);

    conn.send(JSON.stringify(msg));
```

```
    $('#user').val('');

});
```

Sending a New Message

Here we're doing the same thing we did when a new user joined the chat room, only this time the text would be the text entered by the user.

```
$('#send-msg').click(function(){

    var text = $('#msg').val();

    var msg = {

        'user': user,

        'text': text,

        'time': moment().format('hh:mm a')

    };

    updateMessages(msg);

    conn.send(JSON.stringify(msg));



    $('#msg').val('');

});
```

Leaving the Chat Room

What this does is send a message to all the connected clients that a specific user has left the room. Then it cleans up the messages HTML and array. We also hide the main container and show the user container so that a new user can join.

```
$('#leave-room').click(function(){



    var msg = {

        'user': user,

        'text': user + ' has left the room',

        'time': moment().format('hh:mm a')

    };

    updateMessages(msg);

    conn.send(JSON.stringify(msg));



    $('#messages').html('');

    messages = [];



    $('#main-container').addClass('hidden');

    $('#user-container').removeClass('hidden');



});
```

Conclusion

In this tutorial, we've created a simple chat application harnessing the real-time capabilities provided by WebSockets through Ratchet. The files used for this tutorial are available on this Github repo.