Introduction to WebSocket

Real-time Updates

Really interactive, real-time web applications feature ***real-time updates***. For instance, real-time chat applications update with messages contributed by any user in the chat, or a real-time train map needs to be updated which each train's current position.

We can implement these real-time applications using the technologies we already know through the use of ***polling***, in which we schedule an AJAX request to be sent to the server every so often (every 5 seconds, every minute, etc). However this has the disadvantage that if we want frequent updates, we have to either send a message to the server very frequently, potentially overloading the server (and this would be wasteful if the data is not changing) or send it less frequently, meaning updates would not truly be real-time.

Push Technology

What we need instead is ***push technology***. Normal HTTP requests are ***pull*** requests: they ***pull*** data from the server. ***Push*** requests, however, involve the server sending messages to the client ***without receiving a request***. ***WebSocket*** is a protocol which allows servers to send push requests to clients.

How does WebSocket work?

WebSocket has a detailed specification available here which explains it in detail; this Mozilla article also goes into some detail and is easier to understand than the full specification. But to summarise:

- A ***WebSocket server*** runs as an independent process, on an independent port (typically 8080), to Apache.
- A ***WebSocket client***, running in the browser and typically written in JavaScript, listens to requests coming in from the WebSocket server and can send messages back. Similarly, the WebSocket server can listen to, and respond to, messages coming from the client.

Writing a WebSocket client

Writing a WebSocket client is quite easy because JavaScript provides native support. Here is an example:

```
function init() {
    var ws = new WebSocket('ws://example.com:8080');

    ws.onopen = e => {
        //Handle the web socket being opened...
        alert('Websocket was opened!');
        ws.send('ephp001');
    }

    ws.onmessage = e => {
        console.log('Websocket sent back... ' + e.data)
    }

    ws.onerror = e => {
        alert('Error with websocket communication: is t
    }
}
```

Note how we create a ***WebSocket*** object and specify the server and port (example.com and 8080 respectively, here). We then specify three different event handlers:

- ***onopen*** - this event handler will run when the connection to the web socket is made. In this example we use the ***send()*** method of the WebSocket object to send a username back to the server (so that, as soon as we have connected to the server, we let the server know who we are). JSON is typically used as a format to transmit data between client and server.
- ***onmessage*** - this event handler will run when the client receives a push message from the server. The ***data*** property of the event object will contain the data sent back. Here, we just display it on the console.
- ***onerror*** - this will run if an error of some kind occurs. Typical errors include the server not running, or the port being blocked by a firewall.

Writing a WebSocket server

WebSocket servers are a bit more difficult as they do not have native support in most server-side languages. However plenty of ***libraries*** exist, saving the developer from having to write their own server, and a good one for PHP is ***Ratchet***. This is available here. Writing a Ratchet-based WebSocket server uses a similar style of code to writing a JavaScript client: again, you write functions to handle opening a connection, receiving a request and closing a connection, and you can send messages with a ***send()*** method. Here is a simple example:

```
<?php

require('vendor/autoload.php');

use Ratchet\MessageComponentInterface;
use Ratchet\ConnectionInterface;
use Ratchet\Server\IoServer;
use Ratchet\Http\HttpServer;
use Ratchet\WebSocket\WsServer;

class SockServer implements MessageComponentInterface {

    public function __construct() {
        $this->clients = [];
    }

    public function onOpen(ConnectionInterface $conn) {
        $this->clients[$conn->resourceId] = $conn;
    }

    public function onMessage(ConnectionInterface $from
        $from->send("RECEIVED : $msg");
    }

    public function onClose(ConnectionInterface $conn)
        unset($this->clients[$conn->resourceId]);
    }

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

$server = IoServer::factory(new HttpServer(new WsServer
            8080);
$server->run();

?>
```

Note how we create a PHP class which implements ***MessageComponentInterface*** (for those who have done Java programming, this is an example of an ***interface***). We then create these methods:

- ***onOpen()*** - runs when a client makes a connection. The ***$conn*** object represents the current client. Note that each connection has a unique ***resourceId*** which identifies that particular client. Here, we add the client to an array of clients (so the server can keep track of all clients currently connected) and use the resource ID as the index.
- ***onMessage()*** - runs when a message is sent from the client. Here, we just send the message back to the same client. To send a message back to all clients, we could use a foreach loop to loop through $this->clients.
- ***onClose()*** - runs when a client closes the connection (e.g. the browser closes). Here we remove that client from the array.
- ***onError()*** - if there is an error of some kind.

After defining the class, we then create our server object - note how we have to wrap several Ratchet objects for full implementation of the WebSocket protocol. We also specify the port (8080 here). Finally we start the server running with the ***run()*** method.

Note the ***use*** statements at the top. These statements import classes from PHP ***namespaces***. A PHP namespace is a similar concept to a Java package: it is a way of using libraries including multiple classes of the same name to exist in one project. You can import the class from the appropriate namespace. For example here, ***MessageComponentInterface*** belongs to the ***Ratchet*** namespace.

For the meaning of `require('vendor/autoload.php')`, see the discussion on Composer, below.

Installing Ratchet - Introduction to Composer

Ratchet is a third-party library; it is not part of the core PHP distribution. However it is quite easy to install thanks to ***Composer***. Composer (see here) is a dependency management system for PHP projects, allowing you to import specified third-party libraries (***dependencies*** - so called because your project depends on them). You create a ***composer.json*** file specifying which libraries are needed, and then run an install command, and all necessary dependencies will be installed. Composer itself can be downloaded as an application from the Composer site: the Composer application is a file named ***composer.phar*** which is typically saved to your home directory (~ on Linux systems) and run with PHP, e.g:

```
php ~/composer.phar (Composer command)
```

Here is an example of a composer.json file for Ratchet:

```
{
    "require": {
        "cboden/ratchet" : ">=0.3.0"
    }
}
```

This means, as you might expect, the Ratchet package is a dependency of your project, and it needs to be at least version 0.3.0. You can add multiple libraries to the "require" block. Once you have setup a composer.json, you can then install the dependencies with:

```
php ~/composer.phar install
```

See here for full documentation on Composer.

However you do not have to write the composer.json file yourself. You can ***require*** a package by running Composer's require command, e.g. by running this in your project directory:

```
php ~/composer.phar require cboden/ratchet
```

This will generate a composer.json automatically, and will also load all the dependencies.

The dependencies will be downloaded to a ***vendor*** directory within your project directory. Also, an ***autoload.php*** will be generated in this project directory, allowing you to automatically include all the necessary PHP files for all dependencies. So in your PHP code, you can simply add:

```
<?php
require("vendor/autoload.php");
...
```

Exercise

Write a simple chat application, similar to the one demonstrated in class.

1. Create a web page with a text field to add a message, and a div to show all messages so far. The message should be sent to the web socket server when a button is clicked.
2. Write a corresponding Ratchet server and upload this to your home directory on Neptune (the directory before public_html). Like in the likve example, the div should update each time ***anyone*** sends a new message. To do this, use a foreach loop on $this->clients to loop through each client in onMessage(), e.g:

   ```
   foreach($this->clients as $client) {
       // .. send message to this client
   }
   ```

3. Run your Ratchet server by logging into the Linux shell prompt on Neptune. Use PuTTY and use:

   ```
   
   ```

```
Host: ephp.solent.ac.uk
Username: your Edward2 username
Password: your Edward2 password
```

To run your server:

```
php ~nick/composer.phar require cboden/ratchet
```

to install the Ratchet dependency (only necessary first time), and then:

```
php chatserver.php
```

(or whatever you called your Ratchet server).

If it produces no error messages, all is OK: it will then sit there waiting for connections from clients. To test, open up two Chrome windows; each is treated as a separate client. ***Use your allocated port number*** to ensure everyone is running on a separate port.

4. Enhance your application so that it works like a real chat system; each time a new message is sent back from the server, the user who wrote that message should be displayed. To do this you will need a user ID field in the front end and send the user ID with the chat message. Send a JSON message with two fields, one for the user ID and one for the chat message; you can use ***JSON.stringify()*** to encode a JavaScript object as JSON.
5. **ADVANCED:** you can avoid having to send the user ID each time using this technique.
    - Users should have to login first, by specifying a user ID and clicking a button. The user ID will then be sent to the server. Note that you should send it as a piece of JSON data including a message type field, so that the server can tell that this is a user ID and not a chat message.
    - When the user sends a chat message, this should be again sent as JSON so that the server can tell that it is a chat message and not a user ID.
    - When the server receives the ***user*** message containing the user ID, use the ***resourceId*** property of the ***$from*** object to identify the client, i.e.

      ```
      $from->resourceId
      ```

      Each connecting client has a unique resource ID. Store the user ID in an array, using the resource ID as the index. The array will need to be a property of your server object.
    - Then, when the server receives a ***chat message***, use the resource ID to look up the corresponding user ID in the array, and add the user ID to the response.