Learn WebSockets

Hot Potato

In this project, you will have the opportunity to practice using WebSockets to create a four-player hot potato game. The logic of the game is as follows:

- When a new player joins the server, they will be assigned an avatar indicated by the star with the text "You".
- The game requires exactly 4 players to begin. No more, no less. When the 4th player joins, the game starts
- When the game starts, one player will be chosen randomly to hold the potato and the clock will begin counting down from 30.
- When the clock strikes 0, the player holding the potato loses, so pass it quickly!
- If your avatar is holding the potato, click on another player's avatar to pass the potato.

To get you started, we've provided some starter code with the basic application structure:

- **public/index.html**, where you will write code to run in the browser
- **server.js**, where you will write code to run on the server
- **utils/constants.js**, where constants for each message type have been defined for you

Because the focus of this project is on WebSockets (and not on software engineering more broadly), we've encapsulated the majority of the game logic and DOM manipulation in various helper functions.

Your task is to implement the WebSocket logic to pass messages between the server and clients and to call the provided game logic helper functions in response to these messages. In addition to reinforcing the mechanics of connecting a WebSocket server and client, this project will help demonstrate the process of designing a system of messages and payloads sent between clients and the server.

Setup Instructions

This project should be completed on your own computer instead of on Codecademy. You can download what you'll need by clicking the "Download" button below. If you need help setting up your computer, read our article about setting up a text editor for HTML/CSS development.

Once you've downloaded the project, open up the project folder in your text editor. Then, use the command line to navigate to the root of the starter code directory, and run the following commands:

```
npm install
node server.js
```

If these commands are successful, you should see the message Listening on: http://localhost:8080 displayed in your terminal.

Navigate to http://localhost:8080 in your browser to view the game and then click on "Start"!

Download

Tasks

26/26 complete

Mark the tasks as complete by checking them off

Getting Started

1.

Before diving in, spend some time familiarizing yourself with the code we've provided. First, take a look at **public/index.html** where we've defined the variables and helper functions below for you:

VARIABLES:

- wsClient will store an instance of the browser's WebSocket client.
- clientPlayerIndex will store a number assigned to a player when the WebSocket server accepts that player into a game (0 through 3).
- potatoHolderIndex will store the index of the player currently holding the potato (eg. the player with clientPlayerIndex = 1 is holding the potato if (and only if) potatoHolderIndex = 1.

WS LOGIC:

- init() contains all logic to initialize the WebSocket client and define its responses to WebSocket events.
- passThePotatoTo() updates the current potatoHolderIndex and sends the data to the server.

DOM HELPER FUNCTIONS (you will not need to modify these functions):

- updateDisplay() updates the text shown in the center of the screen. Used by a number of the helper functions below.
- setPlayerIndex() assigns each client's clientPlayerIndex and renders the star next to their avatar.
- updateCurrentPotatoHolder() updates the potatoHolderIndex and updates the image of the potato holder.
- countDown() renders the current time with a different color as the clock counts down.

- endGame() displays a game over message.

2.

Next, take a look at **server.js** where we've defined the variables and helper functions below for you:

VARIABLES:

- nextPlayerIndex is used to provide the clientPlayerIndex for the next player to join and can be used to keep track of the number of players in the game.

HELPER FUNCTIONS:

- handleNewUser() determines what to do when a new player joins the server. Until there are 4 players in the game, the server will accept a new client into the game by sending them a clientPlayerIndex value. Once there are 4 players in the game, a random player will be assigned to hold the potato and the game will start. If there are 4 or more players in the game, the server will let them know that the game is full.
- passPotatoTo() should broadcast to all clients the newPotatoHolderIndex when a player passes the potato.
- startTimer() starts a timer interval that "ticks" every 1 second from 30 down to 0. It should broadcast the current time to each client connected to the server and notify all players when the game is over.

3.

Finally, take a look at **utils/constants.js** where the various message types used in this application are defined:

CLIENT

- .MESSAGE.NEW_USER is sent from the client to the server when the client joins the server
- .MESSAGE.PASS_POTATO is sent from the client to the server when the client passes the potato to another player.

SERVER

- .MESSAGE.PLAYER_ASSIGNMENT is sent from the server to a single client when the client joins the server
- .MESSAGE.GAME_FULL is sent from the server to a single client when they attempt to join a full game
- .BROADCAST.COUNTDOWN is broadcast to all clients each time the timer ticks

- .BROADCAST.NEW_POTATO_HOLDER is broadcast to all clients when the potato is passed
- .BROADCAST.GAME_OVER is broadcast to all clients when the game ends (the timer reaches 0)

Create a WebSocket Server

4.

Now that we have the starter code set up, let's begin implementing the WebSocket logic. First, in the WS LOGIC section of **server.js**, declare a constant variable called wsServer. Then create a new instance of the WebSocket.Server class and store it in wsServer.

You should use the provided server as the HTTP server that the WebSocket connection will be made over.

Hint

You will need to use the WebSocket.Server class exported by the ws package. The constructor for this class accepts an object with a server property whose value should be the server created above.

Your code may look like this:

```
const option = { server: myHttpServer };
const myWebSocketServer = new WebSocket.Server(options);
```

5.

Prepare your newly created server to respond to client 'connection' events by calling the .on() method on your wsServer.

For now, the server should print out the message 'A new client has joined the server' each time the 'connection' event is detected.

Hint

Remember, the .on() method accepts two arguments:

- a string specifying an event (in this case, 'connection') to respond to
- a callback that will run whenever that event occurs. The callback will receive one argument—socket— corresponding to a single server/client connection.

Your code may look like this:

```
myWebSocketServer.on('connection', (socket) => {
  // respond to connection events here
});
```

Connect the Client to the WebSocket Server

6.

Great job! You've created a WebSocket server and prepared it to respond to client connections, and now you're ready to create a WebSocket client that connects to your server.

In the WS LOGIC section of **public/index.html**, find the init() function. You should see that there is some provided logic for handling errors and a URL variable has been defined based on the PORT value imported from **utils/constants.js**.

First, replace the '??://' that is currently being used as the protocol in the URL value with the proper protocol for WebSocket connections.

Hint

Your URL value should use the 'ws://' protocol to establish a WebSocket connection with the server. Your code should look like this:

```
const URL = 'ws://localhost:' + PORT;
```

7.

Now that we have the proper URL value to connect to our WebSocket server, let's form the connection using the browser's native WebSocket API.

Below the URL variable, create a new WebSocket client that connects to the WebSocket server via the URL. Store this WebSocket client object in the provided variable wsClient.

Note: *The wsClient variable has been declared at the top of the file in the VARIABLES section in order to maintain global scope within the application. You should NOT redeclare it using the var, let, or const keywords.*

Hint

You will need to use the browser's native WebSocket class to create the WebSocket client. The constructor for this class accepts a URL value to access the WebSocket server.

Your code may look like this:

```
const URL = 'ws://example.com';
myWebSocketClient = new WebSocket(URL);
```

8.

Finally, below the initialization of your wsClient, define your WebSocket client's onopen property as a function that will be called when the client connects to the server.

For now, it can simply print the message 'Connected to the WebSocket server!' to the console.

Test that your code is working. Restart your server (Ctrl + C followed by node server.js) and then navigate to http://localhost:8080 in the browser. Both your server and your client should log a message when the connection is formed.

Hint

This onopen event handler accepts an event argument, however it is often not needed (as in this case). Your code should look something like this:

```
myWebSocketClient.onopen = (event) => {
  // handle connection events here
}
```

Handle New Users

9.

Well done! You've created a WebSocket server and client and connected them. Your next task will be to write the code that allows players to join a hot potato game and be assigned a player avatar.

Inside the .onopen() event handler, send a custom message object from the client to the server to inform the server that a new user wants to join the game. This custom message object should have the following properties:

- type: CLIENT.MESSAGE.NEW_USER

A .payload property will not be necessary here

Hint

You will need to use JSON.stringify() to serialize your data before sending it to the server. Your code should look something like this:

```
const customMessage = {
  type: 'SOME_MESSAGE_TYPE',
  payload: { someData }
};

myWebSocketClient.send(JSON.stringify(customMessage));
```

10.

Great job! Now that you're sending messages from client to server, you'll need to update your server to respond to those messages.

In **server.js**, inside the callback that runs when a server-client connection is formed, set your server up to respond to client 'message' events using the socket.on() method. The server should print out the message received from the client along the socket connection.

At this point, you may want to test that your code is working properly by restarting your server and refreshing your browser. You should see the custom message object {"type":"NEW_USER"} printed in your server's console.

Hint

The .on() method may be called on the socket, passing in two arguments:

- The string 'message' as the first argument
- a callback function accepting a single argument—data—as the second argument. This callback will run each time the client corresponding to the individual socket emits a message. The data argument will be the message sent by the client (as a string).

```
socket.on('message', (data) => {
  // handle the data here
});
```

11.

Now that we have confirmed that we are receiving the custom message object, parse the incoming data object from the callback so that we may use it's values:
Pass data to JSON.parse() to deserialize the data string Extract the message .type and .payload properties into their own variables

Note: *Even though the 'NEW_USER' message type doesn't have a .payload property, other message types that we plan on receiving will so we might as well set up that logic now.*

12.

With the .type extracted, write a switch statement that switches on the message type. Then, create a case for the 'CLIENT.MESSAGE.NEW_USER' message type. This case should:

- Call the handleNewUser() function defined for you in the HELPER FUNCTION section.
- Pass the socket value along to handleNewUser() as an argument

13.

Scroll down to the handleNewUser() helper function in the HELPER FUNCTIONS section.

Currently, this function has some basic structural logic implemented for handling new users:

- Until there are 4 players in the game, we want to assign the new player a player index and increment the index

- If there are 4 players in the game, start the game (which we will handle later)
- If there are more than 4 players in the game, tell the client that the game is full

As you can see, the logic for sending the socket information is missing! Let’s fix that.
Update handleNewUser() to:

- If there are less than 4 players: send the socket a custom message with the following properties:
    - type: SERVER.MESSAGE.PLAYER_ASSIGNMENT
    - 'payload: { clientPlayerIndex: nextPlayerIndex }'
- If there are more than 4 players: send the socket a custom message with the following properties:
    - type: SERVER.MESSAGE.GAME_FULL

Hint

Each socket will have a send() method that will send data to the client corresponding to that socket connection object. Remember you will need to serialize the data using JSON.stringify() before sending an object along the WebSocket connection!

14.

Now that our server is sending our clients messages, we need to update our client-side code to respond to these server messages. First, set the wsClient‘s .onmessage() handler to a function that receives one argument, messageEvent.

Then, parse the messageEvent.data property to extract the message’s type and payload, storing them in their own variables.

Hint

Remember, the data sent by the server will be stored in the messageEvent.data property. You will also need to parse this data using JSON.parse(). Your code should look something like this:

```
myWebSocketClient.onmessage = (messageEvent) => {
  const { type, payload } = JSON.parse(messageEvent.data);
}
```

15.

Finally, write a switch statement (much like the one you write in **server.js**) that switches on the message type. Write a case statement to handle each of the messages the server might send when a player attempts to join a game:

- If the client receives a SERVER.MESSAGE.GAME_FULL message, call updateDisplay() with a helpful message letting the player know that there's no room in the current round.
- If, on the other hand, the client receives a SERVER.MESSAGE.PLAYER_ASSIGNMENT message, call the setPlayerIndex() helper function, passing in the .clientPlayerIndex from the payload.

Test that your code is working properly by restarting your server and refreshing your browser. For the first 4 clients that join the server, they will see a star appear next to their assigned avatar with the text "You". Any clients that join afterwards should see a message informing them that the game is full.

Hint

Your switch statement might look like this:

```
switch(type) {
  case SERVER.MESSAGE.GAME_FULL:
    updateDisplay('The game is full :(')
    break;
  case SERVER.MESSAGE.PLAYER_ASSIGNMENT:
    setPlayerIndex(payload.clientPlayerIndex);
    break;
  default:
    break;
}
```

Implement the Broadcasting Pattern

16.

Great job! Before we can move on to the next set of features, we will need to implement the broadcasting pattern

At the top of the HELPER FUNCTIONS section In **server.js**, declare a function called broadcast() that accepts two arguments – data and socketToOmit.

Note: *In this application, we will be broadcasting our messages to ALL connected clients. Accordingly, the socketToOmit parameter will largely not be used however, it is a good practice to include it in your implementation to reinforce the pattern.*

17.

Now, implement the broadcast pattern. This function should:

- Iterate through the complete list of its connected clients.

- For each connectedClient:
    - Check if the connectedClient still has an open ready state
    - Check if the connectedClient is not the same socket as socketToOmit:
        - If both of these checks pass, send the data to the connectedClient.

Make sure to serialize all data sent in this function using JSON.stringify().

Hint

The broadcasting pattern will require you to use a few key pieces of data:

- The wsServer.clients array holds all socket connection objects
- Each connectedClient has a .readyState property. This property will equal WebSocket.OPEN if the connection is still open.
- Each connectedClient will have a send() method to send data along the connection.

Your function should look something like this:

```
function broadcast(data, socketToOmit) {
  wsServer.clients.forEach((connectedClient) => {
    if (connectedClient.readyState === WebSocket.OPEN && connectedClient !== socketToOmit) {
      connectedClient.send(JSON.stringify(data));
    }
  });
}
```

Pass the Potato

18.

Nice work! With the broadcast pattern implemented, you can now start passing the potato.

In **server.js**, we've provided an outline for passThePotatoTo, the function that will be called each time the potato changes hands. This will occur in two different situations:

1. Once when the game begins, the potato is "passed" to a random player to start
2. While the game is running, players will be passing the potato to each other

Update this passThePotatoTo() function to broadcast a custom message informing all clients of the newPotatoHolderIndex value. This custom message object should include the following properties:

- type: SERVER.BROADCAST.NEW_POTATO_HOLDER
- `payload: { newPotatoHolderIndex }

You should not include a socketToOmit argument when calling the broadcast() function.

Hint

Since your broadcast function has taken care of serializing all data it sends, you do not need to use JSON.stringify() before calling the broadcast() function. Your code should look like this:

```
const data = {
  type: SOME_MESSAGE,
  payload: { someData }
}
broadcast(data);
```

19.

Currently, the passThePotatoTo() function is called when the 4th player joins the server and a random potato holder is chosen. Now, let’s respond to these 'NEW_POTATO_HOLDER' messages on the client-side to properly update the DOM.

In **index.html** and inside the .onmessage() event handler, add a case statement to respond when the server sends the client a SERVER.BROADCAST.NEW_POTATO_HOLDER message:

- Upon receiving this message type, call the updateCurrentPotatoHolder() helper function (defined for you), passing in the .newPotatoHolderIndex value from the payload.

Test out your code by restarting the server, and refreshing your browser. Connect four clients to start the game and you should see one of the four avatars will be holding the hot potato!

20.

Nicely done! You can make the potato change hands from the server side, but players still can’t pass the potato on their own.

In the WS LOGIC section of **index.html**, we’ve started writing a function also called passThePotatoTo() that is called whenever the current potato holder clicks on another player’s avatar (see the .onclick() handlers defined in the DOM SETUP section).

Currently this function just updates the potatoHolderIndex based on the playerIndex argument. However, this update only occurs within the one client that passes the potato. This data needs to be sent to the server so that it may be broadcast out to all other clients.

Finish implementing this function by sending a custom message to the server informing it of the new potato holder. This custom message object should have the following properties:

- type: CLIENT.MESSAGE.PASS_POTATO
- payload: { newPotatoHolderIndex }

Hint

Remember to serialize your data before sending it to the server using the JSON.stringify() method! Your code should look something like this:

```
const messageObj = {
  type: SOME_TYPE,
  payload: { someData }
}
myWebSocketClient.send(JSON.stringify(messageObj));
```

21.

Lastly, set your server up to respond when the client messages that the potato has hands by adding a case statement for the CLIENT.MESSAGE.PASS_POTATO message type.

This case should call the passThePotatoTo() function with the .newPotatoHolderIndex value from the payload as an argument.

Test out your code by restarting your server and refreshing your browsers. At this point, once 4 players have joined the game and a random potato-holder has been assigned, you should be able to pass the potato between the players in the game!

Hint

Notice how we are able to use the passThePotatoTo() helper function to send the same message type (SERVER.BROADCAST.NEW_POTATO_HOLDER) in multiple locations.

When planning out your own WebSocket applications, it is important to consider all of the ways that each message type may be used within your application. Asa result, you may find ways to reduce repetition by creating helper functions.

Start the Timer

22.

Nice job – we are almost finished! At this point you have a functional, albeit relaxed game of hot potato. To finish the game, let’s add time pressure by broadcasting the remaining time to all players and ending the game when the clock strikes 0.

In **server.js**, we’ve defined a function, startTimer() that is called when the 4th player joins the game and contains the countdown clock logic. This function calls setInterval() in order to run a callback every second that does the following:

1. If the timer is still running (it is above 0):
   - it decrements the timer
   - it should broadcast the new time to all players
2. If time has run out (it has reached 0):
   - stops the timer by calling clearInterval()

    - resets the nexPlayerIndex to 0 so that players can join a new game
    - it should broadcast to all players that the game is over

Your task is to integrate WebSocket messaging so that all the clients are aware of the remaining time, and are informed when the game is over.

First, for every clock tick in which there is time remaining, call broadcast() with the current clockValue time by sending out a custom message with the following properties:

- type: SERVER.BROADCAST.COUNTDOWN
- payload: { clockValue }

Hint

Remember, you will not need to serialize this data before broadcasting since that has been taken care of within the broadcast helper function.

23.

In **index.html**, respond to the SERVER.BROADCAST.COUNTDOWN message by adding a case statement in the .onmessage() handler. This case should call countDown() with the .clockValue value from the message's payload.

Test out your code by restarting your server and refreshing your browsers. At this point, once 4 players have joined the game, you should see the clock counting down. Notice how the colors change as the time approaches 0!

24.

If you took the time to test out your code, you will notice that the clock stops at 1. This occurs because we only broadcast the time when the clockValue is greater than 0! Let's fix this by implementing the server-side logic for when the clock reaches 0.

In **server.js** back in the startTimer() function, when there is no time life on the clock, broadcast to all clients that the game is over by sending a custom message object with the following properties:

- type: SERVER.BROADCAST.GAME_OVER

25.

Finally, In **index.html**, respond to the SERVER.BROADCAST.GAME_OVER message by adding a case statement in the .onmessage() handler. This case should call the helper function endGame().

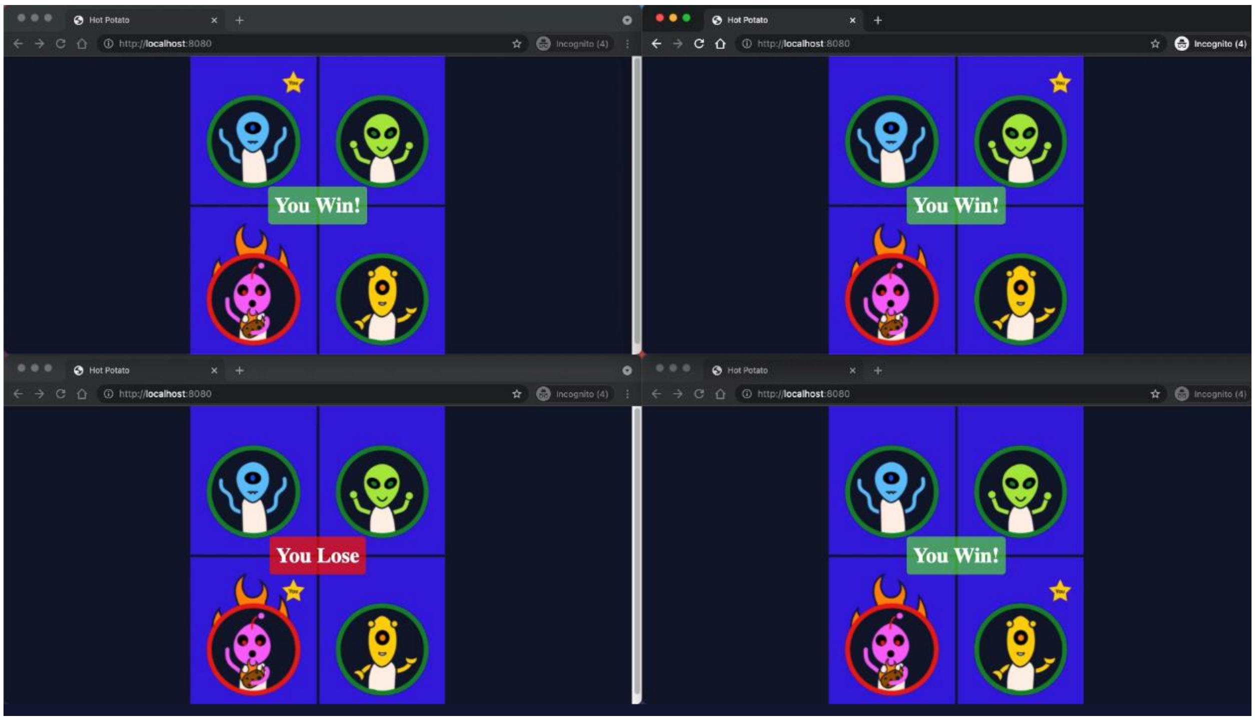
Test out your code by restarting your server and refreshing your browsers. At this point, once 4 players have joined the game and the clock reaches 0, you should see that whomever is holding the potato will see a losing message while all other players will see a winning message!

Celebrate!

26.

Great work! You’ve built a fast-paced multiplayer hot potato game using WebSockets, Now it’s time to share your code with Codecademy learners around the world by visiting our forums.

You can also compare your project to the solution code. Remember, your solution might look different from ours, and that’s okay! There are multiple ways to solve these projects, and you’ll learn more by seeing others’ code.


Hot Potato
http://localhost:8080
Incognito (4)
You Win!
You Win!
You Lose
You Win!