Introduction to Computer Science I
COMP 2406 – Fall 2019

Socket.io

Dave McKenney
david.mckenney@carleton.ca

Learning Outcomes

by the End of this Lecture, Students that have Completed the Reading Assignment and Review Questions should be Able to:

Explain the basics of the web socket protocol

Identify differences between polling/Socket.io

Use Socket.io for real-time communication

Design event-based systems using Socket.io

Introduction to Socket.io

Socket.io is a library "that enables real-time, bidirectional and event-based communication between the browser and the server". – Socket.io

Two components we will use:

A Node.js server

A Javascript client library for the browser

Introduction to Socket.io

Socket.io allows for:

Flexibility in connections (polling, web sockets)

Automatic disconnection/reconnection

Binary data support

Cross-browser support

Namespaces/rooms

Introduction to Socket.io

In other words, Socket.io gives us a powerful tool to communicate events/data in real-time between many people without having to do low-level implementation of any sort

(a bit like the HTTP module did for us)

Note that Socket.io is not a Web Socket implementation

Instead, it uses web sockets when it can

Web Sockets

Web Socket (WS) is another communication protocol (like HTTP)

Allows for full duplex communication

Works alongside HTTP – uses same ports, starts as HTTP, 'upgrades' from HTTP if both sides support

Full Duplex Communication

With full duplex communication, information can flow simultaneously in both directions

The server does not only respond to client reuests

Instead, the server can decide to send the client information without a request (i.e., in response to some other event)

Polling-Based Chat App

Consider the design requirements for a polling-based chat application:

Client loads a page from server (easy)
Client sends a chat message to server (easy)
Server stores message (easy)
Clients poll server to update their messages (hard?)

Issues with the polling approach:

What data should we send?

We don’t want to send ALL messages every time a new message is available

We have to send only new messages to each client

Issues with the polling approach:

So we have to know the time of each message

We also need to know when the last retrieval of the request client was

We need to build the set of new messages

Polling-Based Chat App

Issues with the polling approach:

We also request data even if nothing has changes

And we end up not seeing data immediately when it is available

There is a lot of overhead that comes into this!

Socket.io to the Rescue!

We will see that Socket.io can make this easy

Event-based communication simplifies the architecture:

Client sends message to server
Server broadcasts new message to all other clients
Clients update their state/display

We only send information that we need

Protocol Design

This isn't only a chat/messages solution

The messages/events we send and the data they contain can represent anything

For example: addition of an item, remove of item(s), initialization

Protocol Design

If we can produce a protocol of events/data to drive our application, we can use Socket.io to implement it (e.g., T4, A2)

Server and clients react to events that occur, just like we have been doing the past few weeks

Setting Up Socket.io

To setup Socket.io, we will start from a plain static file server similar to last lecture

We then must install Socket.io with NPM

Setting Up Socket.io

To setup Socket.io, we will start from a plain static file server similar to last lecture

We then must install Socket.io with NPM

Setting Up Socket.io

Within our server, we can require Socket.io and give it our server object

This is necessary as Socket.io USES the HTTP server

```
const io = require('socket.io')(server);
```

io is now a Socket.io object we will use to communicate

Setting Up Socket.io

On the client, we can include the Socket.io client library using a script tag:

```
<script src="/socket.io/socket.io.js"></script>
```

And in our client-side Javascript (chat.js), we can create a socket that connects to the server:

```
let socket = io( );
```

Handling Our First Events

Socket.io has some pre-defined events: ‘connection’ and ‘disconnect’

The connection event lets us set up the server-side socket to handle incoming events

Handling Our First Events

```
io.on('connection', socket => {
  //Event occurs when a client connects
  //Input argument is socket linked to client
  //This is the server-side of the connection
});
```

Add functionality to print when a user joins

Handling Our First Events

Since we have a socket object, we can:

Add additional state/functions to it:
socket.someProperty = someValue;

Define events for the socket to handle:
socket.on('someEvent', someHandlerFunction);

Add a handler to print when a client leaves

Adding More Events

What if we want to store the names of people who are in the chat?

Or in other words, what if we want to associate a name with each socket connection?

Sending Events from the Client

When the connection occurs, we don't get any extra information from the client

But after the client connects, it can send a message:

socket.emit('eventName', data);

Add functionality for the user's name to be sent to and stored on the server when the user connects (we are starting to build a more complex protocol)

Adding More Events

The last slide required the client to send information to the server on its socket

But the other clients would not be aware of somebody joining (or leaving)

The server can also send events…

Sending Events from the Server

The server can send an event to a specific socket:

socket.emit('event', data);

Or to all sockets:

io.emit('event', data);

There are some other methods too, which we will see

Sending Events from the Server

The server receives a new name when a client joins

We can use that event to trigger the server to broadcast that name event to all other clients

When the clients receive the event, they can update their state/display if they handle that event

Adding More Events

Add functionality so the clients update their display for users joining/leaving

Add functionality so the clients can send messages and other clients see them

One More Issue

Now one issue remains – if somebody joins in the chat, they do not see the previous messages

How can we solve this problem?

One More Issue

Need to remember the past messages on the server (or past X messages)

When a new client joins, we can send an 'init' event with the past messages

Expanding on This Protocol

This is a simple protocol – three different types of events and not a lot of complex data

Remember: we can stringify/parse Javascript objects

This allows us to transport complex data easily

Expanding on This Protocol

Some things we could do:

Add user profiles
Attach a timestamp to messages
Allow user to change color of messages
Have the message be removed after X time
Show who is online/offline
‘User is typing…’
Mentions
Private messages
…etc…

Expanding on This Protocol

We will likely talk more about protocols when we get to A2 next week

Other Socket.io Functionality

Within Socket.io you can also broadcast to all OTHER sockets connected to the server:

socket.broadcast.emit('event', data);

Other Socket.io Functionality

Socket.io has support for ‘namespaces’ and ‘rooms’

Namespaces allow you to specify different endpoints within a single system

Rooms further divide each namespace

Socket.io Namespaces

Default namespace is “/”, but we can define others:

```
const nsp = io.of('/my-namespace’);
nsp.on('connection', function(socket){
  console.log('someone joined nsp namespace’);
  nsp.emit(‘newconnect', someData);
});
```

This creates a new namespace “my-namespace” and defines the connection handler for that namespace

Socket.io Namespaces

On the client side, you can join a specific namespace:

```
const socket = io('/my-namespace');
```

Socket.io Rooms

Rooms further subdivide namespaces

You can add a socket to a room on the server:

```
io.on('connection', function(socket){
  socket.join('some room');
});
```

Remember: the code could be anywhere in the server

Socket.io Rooms

To send an event to a room on the server:

io.to('some room').emit('some event');

Note that namespaces/rooms are handled by the Socket.io software – they all rely on only a single connection (why is this good?)

Volatile Messages and Nodes

Socket.io supports ‘volatile’ messages – messages that don’t NEED to be delivered (bad connection, overload, etc.)

Additionally, there is support for ‘nodes’ to distribute Socket.io computation/communication across multiple machines

For a good summary of communication methods:

https://socket.io/docs/emit-cheatsheet/

Summary

So we learned about an alternate protocol (WS) and can use Socket.io to build some real-time communication systems

You will get practice with this over the next tutorial/assignment

Summary

Focus on protocol development and data organization!

Focus on the meaning of events/data (semantics)

Plan before you code!

Summary

Over the next few weeks, we will build more complex and powerful HTTP servers

This will require these types of skills