WebSockets

CS-554 – WEB PROGRAMMING

Terms

Socket: An endpoint between processes

WebSocket: A protocol dictating how to perform full-duplex communication over TCP

What packages?

The usual!

- Express, mostly.

Socket.io

- A WebSocket client for NodeJS
- http://socket.io/
- http://socket.io/docs/

Data Transfer Strategies

What's going on in an HTTP Request?

When you make an HTTP request, your browser has to setup a series of handshakes and make a single request.

- The request has a definite start and end
- The request is for a definite resource

This request follows a very structured data format

- Headers
- Body

What’s a socket?

A socket, in general, is a connection between two processes (and often two computers!)

It allows for two processes to connect and pass data back and forth between each other.

What's a WebSocket?

A WebSocket is a full-duplex socket connection that allows both a server and a client to stream data back and forth simultaneously.

This data is fairly lacking in structure; there is no message standardization for communication over WebSockets, such as with HTTP.

Major differences

HTTP REQUESTS

Requests for resource

Idempotent

Follows a format

WEBSOCKETS

Request for action

No standard for idempotency

Schemaless

Common Uses of Websockets

Social media

Chatrooms

News feeds

Online gaming

Using Websockets

The problem

As applications became more interactive, they required a constant stream of information to be passed off.

These actions continued to become more event-based, which REST is not great for

Using a news-based single page application over REST:

- The user navigates to a state in a SPA that represents *BIG BANK CO, LLC* which gets an article ever ~3 seconds.
- Every 10 seconds, an AJAX call comes and downloads articles between time (NOW) and (NOW – 10 seconds); how do you even factor in the time spent computing?
- The page gets updated. The cycle begins to repeat, falling more and more behind because some articles come at the 12th second, meaning it takes an extra 10 seconds to see them + time to do work.

The solution

WebSockets are simply open and ready for messages to come through in either direction, allowing for an event based setup to easily be implement.

Using a news-based single page application over WebSockets:

- The user navigates to a state in a SPA that represents *BIG BANK CO, LLC* which gets an article ever ~3 seconds. They subscribe to a list of updates to *BIG BANK CO, LLC*
- 3 seconds later the server informs them of a new article. They begin the process of updating the page.
- 2 seconds later the server informs them of a new article. They begin the process of updating the page.
- 5 seconds later the server informs them of a new article. They begin the process of updating the page.
- Even if the processing of the page is expensive, these events still get sent down to the user as they occur.
- The user leaves that state, and unsubscribes to that company's changes; it no longer receives events.

When it’s bad to use websockets

Websockets don’t have any sort of request representation, nor do they have any caching strategy.

They are not particularly great for building out APIs due to their relative lack of structure

Popular websocket frameworks

Socket.io:

- Socket.io is a NodeJS server integration and client side solution to Websockets

SignalR

- SignalrR is an ASP.net server integration and client side solution that abstracts the actual usage of WebSockets away and implements fallbacks into other techniques (ie, longpolling, foreverframes, server sent events)

WebSocket Rails

- Naturally, this one is for Ruby on Rails

Jetty

- Allows for Websockets in a java environment

WebSocket Use Cases and when not to use

Event driven products

What are event driven products?

- Event driven products are products where the user is primarily witnessing a stream of events.
- As an analogy, a sports game is an event driven product; the viewer watches for a stream of updates. These updates may not necessarily be constant.

Why are WebSockets useful here?

- WebSockets allow for quickly sending events down as they occur. The user does not have to re-request information to check if there is an update; it will simply be sent to them
- Following the analogy, there's no expectation of **when** events should occur. Polling every 2-minutes won't do anything for the user, as after 2 minutes the score may not have changed; on the other hand, if you polled every 2 minutes, the game could have turned dramatically and you would be playing catch-up.

Constant Updates

How constant are we talking?

- Sometimes, a great deal of events are expected to occur that may not have an end time. For example, a social media aggregator. Do we ever expect Twitter users to stop tweeting? Moreover, if we are aggregating events, aren't we saying the end time is when we close the page?
- The rate of events does not matter so much as the concept of their end time not existing

Why are WebSockets helpful?

- They're untimed, and very good at simply allowing you to accept all those messages.

Is there room for REST at all?

- You would actually use REST for the initial pull of data, because there is some represented set of data you are requesting that is cacheable; ie, give me my social media feed before xx:yy:zzzz. That data for that request would not change.

High Frequency Messaging

How is that different from constant updates?

- In a constantly updating application, the server is sending down a stream of updates to the client.
- With high frequency message, there could be times that the client has many messages for the server, the server has many messages for the client, or **both simultaneously**.

How does frequency matter?

- Websockets only make expensive HTTP requests on setup.
- The messages sent back and forth are **extremely** small and efficient.
- Since WebSockets are full duplex, both server and client can 'do work' simultaneously.

What's an example there?

- An online game
- Chatrooms

Downsides

No caching

- You're never requesting a resource, so much as you are passing messages; you therefore cannot leverage HTTP caching. You have to manually cache resources server-side / client-side.

Added complexity

- Bi-directional data flow means things are twice as hard to conceptualize now and state issues can crop up

Cannot ensure order of events running

Have to define your own schema, error handling, etc.

- It is essentially standardless

Bad for one-off requests

- The overhead for starting a WebSocket connection is, naturally, higher.

Socket.io

What is Socket.io?

Socket.io is an abstraction over WebSockets that allows bi-directional communication between server and client

- Fancy way of saying it's a WebSocket library that gives you an easy API to use for server and client side

It is extremely, extremely well tested and is arguably the most used WebSocket tool in use currently

Simple API and great documentation found online

- http://socket.io/

Installing Socket.io (server)

Integrating Socket.io into an existing Express server is fairly easy; you are essentially adding another way to communicate with your Node code.

- Install socket.io library
- Hook it into your express app
- Setup listeners for events
- Wait for stuff to happen!

The lecture code contains a clone of the chatroom sample from Socket.io's getting-started guide

Loading Socket.io (client side)

To load Socket.io client side, you simply need to include the script on the client side and connect!

- You'll notice that a socket is then created (with default values, so no config is needed)

Sending data to server

Sending the data to the server plays well with the event-driven nature of the browser and user interaction.

The general strategy for interacting with the server is:

- Wait for user input to occur
- Send data to server

Any JSON serializable data can be sent, or binary data.

Sending data to client

There are two ways that data can be sent to the client:

- As a response to a message the client sent
- As an event that is emitted.

This data is JSON serializable data.

Isolating Messages in Socket.io

What's the problem we're trying to solve?

We often don't want messages to be sent to the entire world.

For example:

- In a chatroom, you don't want private messages to be sent to all connected users, but rather only to one user
- In a chatroom, you want the concept of rooms, where messages are only sent to subsets of people.
- If you are making a sports application, you don't want to get messages about every game that is going on; you'd want to subscribe to certain games to get updates for.

Namespaces

In Socket.io, you can namespace your entire socket. A namespace is the area that your socket has a connection to.

From your client, you can connect to multiple namespaces!

For example:

- You would want to connect to one namespace for chat events
- You would want to connect to another namespace for system wide alerts

Setting up a namespace on your server

Setting up a namespace is simple. In fact, by making a socket.io connection you create a connection that's already setup with a namespace; the default namespace, *io*

We can make a custom namespace using the *io.of* method, as seen here.

```
const app = require('express')();
const http = require('http').Server(app);
const io = require('socket.io')(http);

const chat = io.of("/chat");
const usersToSocket = {};

app.get('/', (req, res) => {
  res.sendFile(__dirname + '/index.html');
});

chat.on('connection', (socket) => {
  socket.emit('request-credentials');

  socket.on('chat message', (msg) => {
    chat.emit('chat message', msg.text);
  });

  socket.on('setup', (connectionInfo) => {
    usersToSocket[connectionInfo.nickname] = socket;
  })
});

http.listen(3000, () => {
  console.log('listening on http://localhost:3000');
});
```

Setting up a namespace on your client

Connecting to a namespace is just as easy on the client as it is on the server.

```
<script>
    const nickname = "Phil the Great";
    const socket = io('http://localhost:3000/chat');

    socket.on('request-credentials', () =>  {
      alert("Test");
      socket.emit('setup',  { nickname: nickname });
    });

    socket.on('chat message', function(msg){
      $('#messages').append($('<li>').text(msg));
    });

    $('form').submit(function(){
      let message = {
        text: $('#m').val()
      };

      socket.emit('chat message', message);
      $('#m').val('');
      return false;
    });


  </script>
```

Rooms

Sometimes, your application needs to support many different areas that each emit events to one or more users.

The solution to this problem is to use *rooms*, which are isolated areas that route messages only to users subscribed to particular rooms.

For example:

- In a chatroom, users could be a part of multiple rooms that each get messages sent only to that room.
- In a real-time online research tool used to see news on companies, you would enter a room to track updates to that company. As more information comes in, each user would get the most recent news.

Trivia info – in some WebSocket frameworks, *Rooms* are referred to as *Groups*

Setting up a room on your server

Joining a room is simple and has to be done on the server.

You can join a room simply by using the *socket.join* method.

```
chat.on('connection', (socket) => {
  socket.join('general'); // join the general room.

  socket.on('join-room', (roomId) => {
    socket.join(roomId);
  });
```

Sending a direct message

Sometimes, we want to directly message a certain user. Rather than isolating a user, however, we can view this problem simpler: isolating a socket, and sending a message to that socket.

One of the benefits of using JavaScript is that it is heavily built around functional scope. This allows us to access an object in an event handler that stores a reference to a particular socket.

When wanting to directly message a user, you simply message their socket directly.

Sending a direct message on your server

Using simple events, we can setup a reference to a socket that other users can then directly emit messages to.

```
const chat = io.of("/chat");
const usersToSocket = {};

app.get('/', (req, res) => {
  res.sendFile(__dirname + '/index.html');
});

chat.on('connection', (socket) => {
  socket.on('direct message', (msg) => {
    usersToSocket[msg.userName].emit('private message', {
      from: msg.fromUserName,
      text: msg.text
    });
  });

  socket.on('setup', (connectionInfo) => {
    usersToSocket[connectionInfo.nickname] = socket;
  });

  socket.on('chat message', (msg) => {
    chat.emit('chat message', msg.text);
  });

  socket.emit('request-credentials');
});
```

Authenticating in Socket.io

Just as we would want to authenticate users in REST calls, we'd also want to authenticate users over WebSockets.

We can plug in an authentication layer very simply by using a node package, such as *socket-io* auth

- https://www.npmjs.com/package/socketio-auth

Setting up socketio-auth on the client

Immediately after connecting, simply issue an authentication event and provide whatever data you need to authenticate.

```
var socket = io.connect('http://localhost');
socket.on('connect', function(){
  socket.emit('authentication', {username: "John", password: "secret"});
  socket.on('authenticated', function() {
    // use the socket as usual
  });
});
```

Setting up socketio-auth on the server

There are three methods that can be passed as arguments to setup the auth, as well as a timeout value in milliseconds.

The first method, authenticate, is required.

postAuthenticate is a method that runs after successful authentication.

Disconnect is a method that runs when the client is disconnected.

```
var io = require('socket.io').listen(app);

require('socketio-auth')(io, {
  authenticate: authenticate,
  postAuthenticate: postAuthenticate,
  disconnect: disconnect,
  timeout: 1000
});
```

Setting up socketio-auth on the server

Setting up socketio-auth is relatively simple and just requires you to pass your socket.io module to the authentication layer.

socketio-auth does tie you to a particular implementation manner for your authentication; you can easily drop in token based authentication.

```
var io = require('socket.io').listen(app);

require('socketio-auth')(io, {
  authenticate: function (socket, data, callback) {
    //get credentials sent by the client
    var username = data.username;
    var password = data.password;

    db.findUser('User', {username:username}, function(err, user) {

      //inform the callback of auth success/failure
      if (err || !user) return callback(new Error("User not found"));
      return callback(null, user.password == password);
    });
  }
});
```

Setting up socketio-auth on the server

postAuthenticate can be used with the same data provided in order to associate a user object to a client object.

```
function postAuthenticate(socket, data) {
  var username = data.username;

  db.findUser('User', {username:username}, function(err, user) {
    socket.client.user = user;
  });
}
```