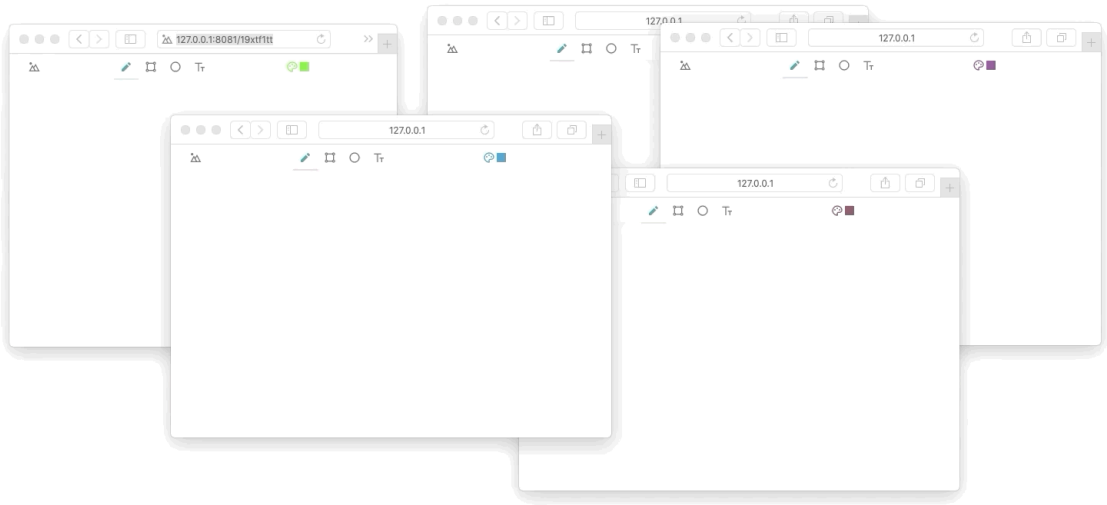
Server Sent Events + WebRTC Mesh Networks

Created Sep 8, 2020 by Tom Holloway and published in Tom's Notes.



- Part 1: Realtime collaborative drawing with canvas and WebRTC
- Part 2: Server Sent Events + WebRTC Mesh Networks
- Part 3: Simulating webkit force, canvas color swatches
- Part 4: Redis PubSub + WebRTC Signaling

In a previous article, we created a realtime collaborative drawing tool with canvas and webrtc. We used simplepeer to get the job done and used WebSocket to communicate to a signaling server.

This works great, but it sort of *glosses over* the underlying implementation with some added abstractions and complexity that may not be necessary. We can simplify things by using SSE. As well, we'll take a closer look at WebRTC by utilizing the RTCPeerConnection directly.

By the end of this article we'll have:

- Randomly generated ids for drawing rooms
- Server Sent Events for our communication signaling channel
- Fully connected mesh network of WebRTC data channels

Simplifying Server Communication with SSE

We were using WebSockets because we needed a way for the server to trigger callbacks on the clients whenever things like *an offer*, *a peer joined*, and any other signaling communication happened. Unfortunately, there are a number of edge cases that need to be handled:

- Losing connection state to our server/client
- WebSockets might not be supported (by default on load balancers)
- Fallbacks to long-polling

This added complexity of WebSockets is such that usually you will just use something like socket.io to handle all these for us. Instead, we can use SSE to handle our communication from the server, and it uses just HTTP to do this.

By using SSE, we can gain the following benefits:

- Data efficient, easily understood protocol over HTTP
- Automatically multiplexed over HTTP/2
- Can use just a single connection
- Connections can be moved to a different server easily
- No need for complicated load balancer configurations or issues with proxies or firewalls

Rooms and Mesh Networks

Our server functionality was mostly just passing messages along but we want to do a little bit more. We need to be able to orchestrate how our peers join the server. We also want to have a sharable room id so that other people can join. Let's take another look at creating our express server now.

The first thing we need to take care of is routing our users to a unique room. This will ensure that on page load we get our own unique drawing surface and to have others join, we simply need to just share that link.

```
var express = require('express');
var http = require('http');
var path = require('path');

const app = express();
app.use('/static', express.static(`${__dirname}/static`));

const server = http.createServer(app);

// starting index
app.locals.index = 100000000000;

app.get('/', (req, res) => {
    app.locals.index++;
    let id = app.locals.index.toString(36);
    res.redirect(`/${id}`);
});

app.get('/:roomId', (req, res) => {
    res.sendFile(path.join(__dirname, 'static/index.html'));
});

server.listen(process.env.PORT || 8081, () => {
```

```
    console.log(`Started server on port ${server.address().port}`);
});
```

Then in our static directory we have:

- /static/index.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Let's Draw Together</title>
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/remixicon@2.5.0/fonts/remixicon.c
ss">
    <link rel="stylesheet" href="/static/index.css">
    <link rel="alternate icon" type="image/png"
href="/static/logo.png">
    <link rel="icon" type="image/svg+xml" href="/static/logo.png">
</head>
<body>
    <div class="flush vstack">
        <div class="menubar hstack">
            <a class="icon-link center">
                <i class="ri-lg ri-landscape-line"></i>
            </a>
            <div class="spacer"></div>
            <a class="icon-link active center">
                <i class="ri-lg ri-pencil-fill"></i>
            </a>
            <div class="spacer"></div>
            <a class="icon-link center">
                <i class="ri-lg ri-palette-line"></i>
                <i class="ri-lg ri-checkbox-blank-fill"></i>
            </a>
```

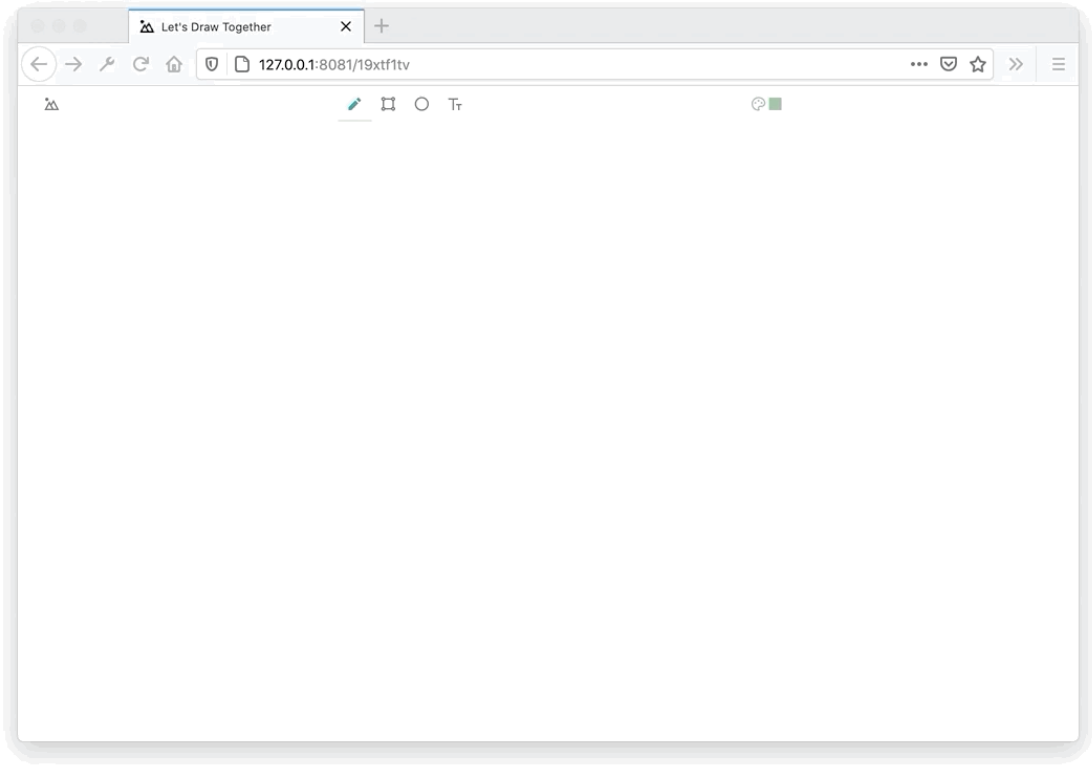
```
            <div class="spacer"></div>
        </div>
        <div class="spacer app">
            <canvas></canvas>
        </div>
    </div>
</body>
</html>
```

- /static/index.css

```
:root {
    --root-font-size: 12px;
    --standard-padding: 16px;

    --bg: #fafafa;
    --fg: #666;
    --menubar-bg: #fdfdfd;
    --active-color: #339999;

    --menubar-shadow: 0 8px 6px -6px #f4f4f4;
}

/** Reset */
html, body, nav, ul, h1, h2, h3, h4, a, canvas {
    margin: 0px;
    padding: 0px;
    color: var(--fg);
}
html, body {
    font-family: Roboto, -apple-system, BlinkMacSystemFont, 'Segoe
UI', Oxygen, Ubuntu, Cantarell, 'Open Sans', 'Helvetica Neue', sans-
serif;
    font-size: var(--root-font-size);
    background: var(--bg);
```

```
    height: 100%;
    width: 100%;
    overflow: hidden;
}
*, body, button, input, select, textarea, canvas {
    text-rendering: optimizeLegibility;
    -webkit-font-smoothing: antialiased;
    -moz-osx-font-smoothing: grayscale;
    outline: 0;
}

/** Utilities */
.hstack {
    display: flex;
    flex-direction: row;
}
.vstack {
    display: flex;
    flex-direction: column;
}
.center {
    display: flex;
    align-items: center;
}
.spacer {
    flex: 1;
}
.flush {
    height: 100%;
}
.icon-link {
    padding: calc(var(--standard-padding) / 2);
    margin: calc(var(--standard-padding) * -1) 0px;
    font-size: 1rem;
    position: relative;
    border-bottom: 2px solid transparent;
    top: 2px;
```

```
    cursor: pointer;
}
.icon-link:hover {
    color: var(--active-color);
}
.icon-link.active {
    color: var(--active-color);
    border-bottom: 2px solid var(--active-color);
}

/** Sections */
.menubar {
    padding: var(--standard-padding);
    box-shadow: var(--menubar-shadow);
    background: var(--menubar-bg);
}
.app {
    width: 100%;
}
```



Connecting to the Server Event Stream

A Server Sent Event Stream, in terms of HTTP, is not much different than a download that never finishes. We simply want to open up a connection to the server and establish this connection as a client that can be written to anywhere in the application. Let's add the code for that portion:

```
// store the connections from clients here
var clients = {};

function disconnected(client) {
    let index = app.locals.clients.indexOf(client);
    if (index > -1) {
        app.locals.clients.splice(index, 1);
```

```
    }
}

app.get('/connect', (req, res) => {
    if (req.headers.accept !== 'text/event-stream') {
        return res.sendStatus(404);
    }

    // write the event stream headers
    res.setHeader('Cache-Control', 'no-cache');
    res.setHeader('Content-Type', 'text/event-stream');
    res.setHeader("Access-Control-Allow-Origin", "*");
    res.flushHeaders();

    // setup a client
    let client = {
        id: uuid.v4(),
        emit: (event, data) => {
            res.write(`id: ${uuid.v4()}`);
            res.write(`event: ${event}`);
            res.write(`data: ${JSON.stringify(data)}\n\n`);
        }
    };

    clients[client.id] = client;

    // emit the connected state
    client.emit('connected', { user: req.user });

    req.on('close', () => {
        disconnected(client);
    });
});
```

In the above implementation, all we are doing is keeping the response connection around in the clients in order to respond to other messages

and relay information from one client to another. To do this, all we have to do is write the headers as a response that we are going to use a `text/event-stream` and all subsequent writes can take the simple format described below.

Server Sent Events Format

> The event stream is a simple stream of text data which must be encoded using UTF-8. Messages in the event stream are separated by a pair of newline characters. A colon as the first character of a line is in essence a comment, and is ignored.

> Each message consists of one or more lines of text listing the fields for that message. Each field is represented by the field name, followed by a colon, followed by the text data for that field's value.

Server Sent Events consist of 4 available fields (one per line) separated by a colon. These include:

- event

> A string identifying the type of event described. If this is specified, an event will be dispatched on the browser to the listener for the specified event name; the website source code should use addEventListener() to listen for named events. The onmessage handler is called if no event name is specified for a message.

- data

> The data field for the message. When the EventSource receives multiple consecutive lines that begin with data:, it concatenates them, inserting a newline character between each one. Trailing newlines are removed.

- id

> The event ID to set the EventSource object's last event ID value

- retry

> The reconnection time to use when attempting to send the event. This must be an integer, specifying the reconnection time in milliseconds. If a non-integer value is specified, the field is ignored.

```
event: userconnect
data: {"username": "bobby", "time": "02:33:48"}

event: usermessage
data: {"username": "bobby", "time": "02:34:11", "text": "Hi
everyone."}

event: userdisconnect
data: {"username": "bobby", "time": "02:34:23"}
```

```
event: usermessage
data: {"username": "sean", "time": "02:34:36", "text": "Bye,
bobby."}
```

JWT (Json Web Tokens)

We need a quick way to identify which requests belong to which user in the website. For this, we're just going to use jwt. It's a quick pass at letting us make sure we have the right user and that subsequent peer requests can be correctly associated with that user.

First, make sure to add it as a dependency to our *package.json*. You should already have express at this point. Additionally, we're going to setup a `.env` file to configure a `TOKEN_SECRET` environment variable. To take advantage of this we will use dotenv.

```
npm install --save express jsonwebtoken dotenv
```

In a `.env` I created a `TOKEN_SECRET` using the following (you can use any method you like, the below is for the sake of simplicity):

```
require('crypto').randomBytes(64).toString('hex')
```

Then paste the result in the `.env` file

```
TOKEN_SECRET=569e3cd22e2ff68ef02688c2100204cd29d7ad2520971ad9eea6db1
c2be576a666734a4531787448811001a76d63fd5394e1fc8f7083bab7793abead60b
```

```
a1392
```

Next, add the following code to make sure we can generate tokens and authenticate them on incoming requests.

```
var jwt = require('jwt');
var dotenv = require('dotenv');

dotenv.config();

function auth(req, res, next) {
    let token;
    if (req.headers.authorization) {
        token = req.headers.authorization.split(' ')[1];
    } else if (req.query.token) {
        token = req.query.token;
    }
    if (typeof token !== 'string') {
        return res.sendStatus(401);
    }

    jwt.verify(token, process.env.TOKEN_SECRET, (err, user) => {
        if (err) {
            return res.sendStatus(403);
        }
        req.user = user;
        next();
    });
}

app.post('/access', (req, res) => {
    if (!req.body.username) {
        return res.sendStatus(403);
    }
    const user = {
```

```
        id: uuid.v4(),
        username: req.body.username
    };

    const token = jwt.sign(user, process.env.TOKEN_SECRET, {
expiresIn: '3600s' });
    return res.json(token);
});
```

Now we have a way to generate auth tokens. In a more realistic scenario, we might decide to see if this authentication method can generate unique tokens according to the logged in user. However, for the time being this is just going to be based on anonymous users. We also have an auth method to verify the incoming token. Let's go ahead and update our /connect function to use our local `req.user` and make sure it passes through auth function.

```
app.get('/connect', auth, (req,res) => {
    if (req.headers.accept !== 'text/event-stream') {
        return res.sendStatus(404);
    }

    // write the event stream headers
    res.setHeader('Cache-Control', 'no-cache');
    res.setHeader('Content-Type', 'text/event-stream');
    res.setHeader("Access-Control-Allow-Origin", "*");
    res.flushHeaders();

    // setup a client
    let client = {
        id: req.user.id,
        user: req.user,
        emit: (event, data) => {
            res.write(`id: ${uuid.v4()}`);
            res.write(`event: ${event}`);
```

```
            res.write(`data: ${JSON.stringify(data)}`);
        }
    };

    clients[client.id] = client;

    req.on('close', () => {
        disconnected(client);
    });
});
```

Now all the peer ids will line up with the auth token generated user ids. We'll use this whenever our users actually join a room below.

Joining a room, relaying messages and disconnecting

There are essentially 3 main functions that we care about in this application as far as the server is concerned.

When a user wants to join a *room*

When we join a room we need to be able to tell all the current clients in that room that a new peer has joined. Additionally, the currently associated client connection needs to communicate with all these existing clients to setup a peer connection by generating an offer.

```
var channels = {};

app.post('/:roomId/join', auth, (req, res) => {
    let roomId = req.params.roomId;
    if (channels[roomId] && channels[roomId][req.user.id]) {
```

```
            return res.sendStatus(200);
        }
        if (!channels[roomId]) {
            channels[roomId] = {};
        }

        for (let peerId in channel) {
            if (clients[peerId] && clients[req.user.id]) {
                clients[peerId].emit('add-peer', { peer: req.user,
    roomId, offer: false });
                clients[req.user.id].emit('add-peer', { peer:
    clients[peerId].user, roomId, offer: true });
            }
        }

        channels[roomId][req.user.id] = true;
        return res.sendStatus(200);
    });
```

When a user needs to relay messages to another peer

When a peer-to-peer connection is being established, WebRTC must be able to pass along SDP messages for things like the WebRTC Session, WebRTC Offers, and WebRTC Answers.

This relay information needs to be passed through a signaling server. We're going to simply pass along these messages to whichever intended peer (or peers) the user is requesting to send to.

```
    app.post('/relay/:peerId/:event', auth, (req, res) => {
        let peerId = req.params.peerId;
        if (clients[peerId]) {
            clients[peerId].emit(req.params.event, { peer: req.user,
    data: req.body });
```

```
        }
        return res.sendStatus(200);
    });
```

When a user disconnects entirely from the server

Finally, when a user disconnects from the server we need to clean-up the channels that this user was in. To do this, we're going to update the `disconnected` function.

```
    function disconnected(client) {
        delete clients[client.id];
        for (let roomId in channels) {
            let channel = channels[roomId];
            if (channel[client.id]) {
                for (let peerId in channel) {
                    channel[peerId].emit('remove-peer', { peer:
    client.user, roomId });
                }
                delete channel[client.id];
            }
            if (Object.keys(channel).length === 0) {
                delete channels[roomId];
            }
        }
    }
```

Setting up the client connection

Now that we have a server that can properly handle communication from our clients, let's go ahead and write our WebRTC library to perform all this

communication. In the previous article, we were using simplepeer, however in this article we're going to use the WebRTC api directly instead. This will let us get a bit better handle on what is precisely going on and how you could even do some of this communication manually.

Before we do this, we need to setup a few things like getting the `/access` token, and setting up the EventSource to stream messages to us.

Add the following to the bottom of our index.html

```
<script type="text/javascript" src="/static/load.js"></script>
```

Then, in a new file /static/load.js we need to add the following code to setup the event stream and access token.

```
var context = {
    username: 'user' + parseInt(Math.random() * 100000),
    roomId: window.location.pathname.substr(1),
    token: null,
    eventSource: null
};

async function getToken() {
    let res = await fetch('/access', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
            username: context.username
        })
    });
    let data = await res.json();
```

```
    context.token = data.token;
}

async function join() {
    return fetch(`/${context.roomId}/join`, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
            'Authorization': `Bearer ${context.token}`
        }
    });
}

async function connect() {
    await getToken();
    context.eventSource = new EventSource(`/connect?
token=${context.token}`);
    context.eventSource.addEventListener('add-peer', addPeer,
false);
    context.eventSource.addEventListener('remove-peer', removePeer,
false);
    context.eventSource.addEventListener('session-description',
sessionDescription, false);
    context.eventSource.addEventListener('ice-candidate',
iceCandidate, false);
    context.eventSource.addEventListener('connected', () => {
        join();
    });
}

function addPeer(data) {}
function removePeer(data) {}
function sessionDescription(data) {}
function iceCandidate(data) {}

connect();
```

This is nearly all the communication we need to get started! In the above code, we are using the fetch api to make a request to get an access token by providing username in our *context*.

Once we are done setting up the event source, we can call `join` which will use the fetch api to POST that we would like to *join* the current room. If you recall, `/:roomId/join` will iterate over the clients in a given channel and call `add-peer` with the newly joined user.id, it will also call `add-peer` to this client with `offer: true` in the data.

WebRTC - Setting up a Mesh Network

WebRTC is built using a number of protocols and APIs that work together to achieve the capabilities of capturing and streaming audio/media/data between browsers without an intermediary.

> For a full-length guide and overview on how WebRTC works, I suggest looking into WebRTC for the curious as it is actually written by some of the authors of WebRTC

In WebRTC, specifically we are interested in setting up RTCPeerConnection in order to communicate with other members of the network. We will be setting up a peer connection whenever we receive the `add-peer` message.

```
const rtcConfig = {
    iceServers: [{
        urls: [
            'stun:stun.l.google.com:19302',
            'stun:global.stun.twilio.com:3478'
        ]
```

```
    }]
};

function addPeer(data) {
    let message = JSON.parse(data.data);
    if (context.peers[message.peer.id]) {
        return;
    }

    // setup peer connection
    let peer = new RTCPeerConnection(rtcConfig);
    context.peers[message.peer.id] = peer;

    // handle ice candidate
    peer.onicecandidate = function (event) {
        if (event.candidate) {
            relay(message.peer.id, 'ice-candidate',
event.candidate);
        }
    };

    // generate offer if required (on join, this peer will create an
offer
    // to every other peer in the network, thus forming a mesh)
    if (message.offer) {
        // create the data channel, map peer updates
        let channel = peer.createDataChannel('updates');
        channel.onmessage = function (event) {
            onPeerData(message.peer.id, event.data);
        };
        context.channels[message.peer.id] = channel;
        createOffer(message.peer.id, peer);
    } else {
        peer.ondatachannel = function (event) {
            context.channels[message.peer.id] = event.channel;
            event.channel.onmessage = function (evt) {
                onPeerData(message.peer.id, evt.data);
```

```
            };
        };
    }
}

function broadcast(data) {
    for (let peerId in context.channels) {
        context.channels[peerId].send(data);
    }
}

async function relay(peerId, event, data) {
    await fetch(`/relay/${peerId}/${event}`, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
            'Authorization': `Bearer ${context.token}`
        },
        body: JSON.stringify(data)
    });
}

async function createOffer(peerId, peer) {
    let offer = await peer.createOffer();
    await peer.setLocalDescription(offer);
    await relay(peerId, 'session-description', offer);
}
```

This is doing a few things, first we have to actually create an RTCPeerConnection. We are passing along some default ICE/STUN servers to ensure that the ICE/STUN portion of the protocol works successfully as we pass things through the signaling server (our express app). Next, when onicecandidate is called due to the negotiation phase of an offer, it will relay that information along to the peer.

We also are creating the datachannel and subscribing to the messages whenever this happens. We only want to create a new data channel and generate an offer when we are supposed to initiate this portion of the negotiation. Otherwise, we will simply listen for the ondatachannel.

`remove-peer`, `ice-candidate` and `session-description` have to be handled whenever a peer is removed, is initiating candidate or session information. We need to handle `ice-candidate` and `session-description` to create answers so that the remote peer can set the `localDescription` or `remoteDescription` appropriately.

```
async function sessionDescription(data) {
    let message = JSON.parse(data.data);
    let peer = context.peers[message.peer.id];

    let remoteDescription = new RTCSessionDescription(message.data);
    await peer.setRemoteDescription(remoteDescription);
    if (remoteDescription.type === 'offer') {
        let answer = await peer.createAnswer();
        await peer.setLocalDescription(answer);
        await relay(message.peer.id, 'session-description', answer);
    }
}

function iceCandidate(data) {
    let message = JSON.parse(data.data);
    let peer = context.peers[message.peer.id];
    peer.addIceCandidate(new RTCIceCandidate(message.data));
}

function removePeer(data) {
    let message = JSON.parse(data.data);
    if (context.peers[message.peer.id]) {
        context.peers[message.peer.id].close();
    }
```

```
        delete context.peers[message.peer.id];
    }
```

Notice that in the `session-description` function we are setting the remoteDescription according to the information provided and we proceed to generate an answer to an offer (if one was provided) before setting our localDescription and relaying that information along. Both the `offer` and `answer` provide information in the form of SDP.

Awesome! 🚀 It may not seem like it at first, but we've just create a system for communicating data over UDP using WebRTC Data Channels! If you start up the server with `node .` and load up the same room id in two different browser windows you should be able to inspect the `context.channels`.

Realtime Collaborative Drawing

Let's copy over the code from our last article and create a file called /static/draw.js.

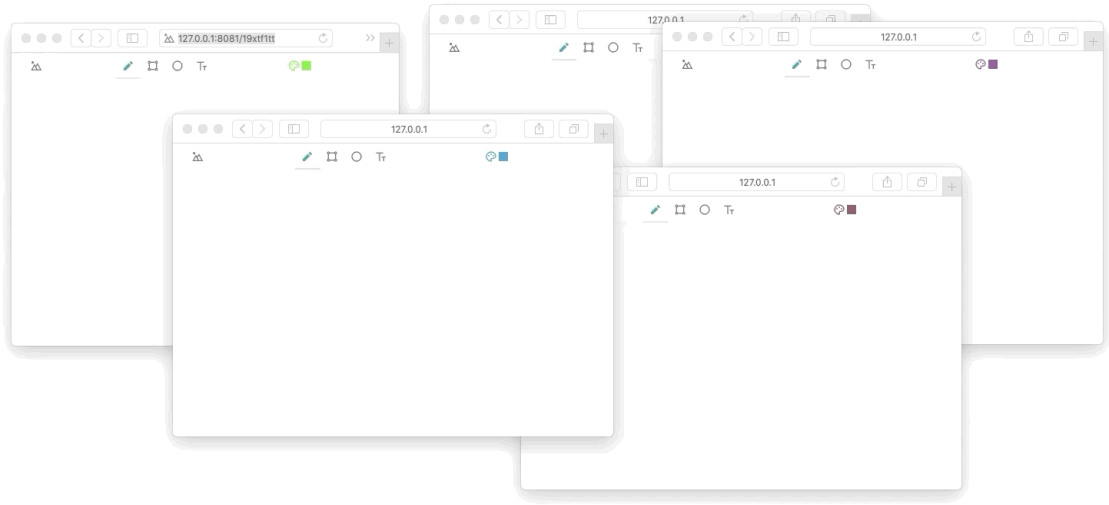
```
const canvas = document.querySelector('canvas');
const ctx = canvas.getContext('2d');

var lastPoint;
var force;

function randomColor() {
    let r = Math.random() * 255;
    let g = Math.random() * 255;
    let b = Math.random() * 255;
    return `rgb(${r}, ${g}, ${b})`;
}
```

```
var color = randomColor();
var colorPicker = document.querySelector('[data-color]');
colorPicker.dataset.color = color;
colorPicker.style.color = color;

function resize() {
    canvas.width = window.innerWidth;
    canvas.height = window.innerHeight;
}

function onPeerData(id, data) {
    draw(JSON.parse(data));
}

function draw(data) {
    ctx.beginPath();
    ctx.moveTo(data.lastPoint.x, data.lastPoint.y);
    ctx.lineTo(data.x, data.y);
    ctx.strokeStyle = data.color;
    ctx.lineWidth = Math.pow(data.force || 1, 4) * 2;
    ctx.lineCap = 'round';
    ctx.stroke();
    ctx.closePath();
}

function move(e) {
    if (e.buttons) {
        if (!lastPoint) {
            lastPoint = { x: e.offsetX, y: e.offsetY };
            return;
        }

        draw({
            lastPoint,
            x: e.offsetX,
            y: e.offsetY,
```

```
            force: force,
            color: color
        });

        broadcast(JSON.stringify({
            lastPoint,
            x: e.offsetX,
            y: e.offsetY,
            force: force,
            color: color
        }));

        lastPoint = { x: e.offsetX, y: e.offsetY };
    }
}

function up() {
    lastPoint = undefined;
}

function key(e) {
    if (e.key === 'Backspace') {
        ctx.clearRect(0, 0, canvas.width, canvas.height);
    }
}

function forceChanged(e) {
    force = e.webkitForce || 1;
}

window.onresize = resize;
window.onmousemove = move;
window.onmouseup = up;
window.onkeydown = key;

window.onwebkitmouseforcechanged = forceChanged;
```

```
resize();
```

Notice that we are using the `onPeerData(id, data)` to draw that information to the screen and we are using the /static/load.js function broadcast to broadcast our current drawing information to the room. By the end of this, we now have a fully functioning P2P Mesh Network that uses Server Sent Events as our realtime signaling server.



Conclusion

Phew! We may have done a bit more in this tutorial. We learned about *Server Sent Events*, we implemented our signaling server and directly used the WebRTC library, we even added support for mesh network peer connections within unique room ids. Now that we have the underlying communication just about squared away, our drawing tool needs a bit of an upgrade in capabilities and tools.

In the next article, we're going to add a few more tools other than the brush and learn a bit about state synchronization. Currently, our drawing state is simply overlayed by executing every operation we receive - but there is nothing that tells us what the state of the drawing surface is when we load the page. We will take a look at how to utilize CRDTs to fix this kind of distributed problem.

In a future article, we will revisit the architecture and add in a PubSub server using Redis. Adding a PubSub server will allow us to create a load balancer and any number of deployed signaling servers with our connections (thus helping us scale).

Check out the next part in the series:

Part 3: Simulating webkit force, canvas color swatches

Thanks for reading. Comment on this post!

Back to index.

RSS feed / Subscribe via email / Follow me on Twitter / GitHub

© Copyright 2021 Tom Holloway / Resume