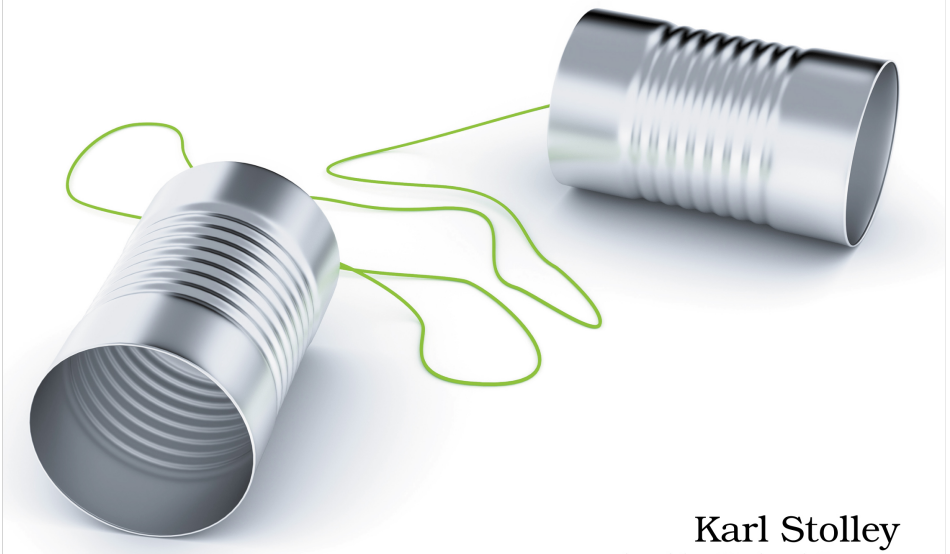
The Pragmatic Programmers

Programming WebRTC

Build Real-Time Streaming Applications for the Web



Karl Stolley
edited by Michael Swaine

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Using a Lightweight Signaling Channel

Browsers lack a signaling channel that enables a connection to be negotiated. The editors of the WebRTC specification have gone so far as to avoid requiring any specific signaling technology whatsoever: you have to bring your own. So to prepare the way to establish a WebRTC connection, we need first to select and set up a signaling channel.

A server-based signaling channel is the most convenient option. Even a very basic peer-to-peer application, like the one you're writing right now, requires both browsers to visit a URL pointing at some server on the web to download the HTML, CSS, and JavaScript necessary to establish and support the call. With a web server already in the mix, any server-side setup for passing a message from one browser to another would suffice.

Technically speaking, a server-based signaling channel isn't strictly necessary. Two peers connecting over WebRTC could, in theory, email or even handwrite and make no-contact delivery of their browsers' offers and answers to set up the peer connection—just like the two friends above could have set their coffee date using semaphore, tin cans on a string, a dead drop, or a whole bunch of other improbable signaling channels. But a server is going to provide the greatest flexibility and much lower latency. It would be possible, for example, to write your own signaling channel using WebSockets, in any server-side language you choose: PHP, Python, Ruby, and so on.

To prevent you from getting sidetracked by all of that work, I have written a very small server in ExpressJS[6] that includes a signaling channel based on Socket.IO.[7] It's included in the book's sample code you already downloaded. The signaling channel works out of the box. While you might find value in experimenting with it, you won't be writing much server-side code in this book. Our focus is in the browser, because that's where the real action is.

But before we get to work in the browser, you will better grasp the code you're about to write for using the signaling channel if we take a quick walking tour of the signaling channel itself. The channel is very little and very dumb: all it does is manage a few events and shuttle messages back and forth. It doesn't know about WebRTC or anything else that we might be doing. To the greatest extent possible, it's better to keep knowledge of WebRTC in and between browsers. The less your signaling channel does, the easier it will be to move your app to a different signaling channel in the future.

6. https://expressjs.com/
7. https://socket.io/

The server's signaling channel component is fewer than a dozen lines, all of which can be reproduced here. As you've seen, sometimes I walk through setting up a piece of code, line by line. Other times, I will do a dramatic reveal of the whole thing before talking through it with you. Like here:

server.js

```
const namespaces = io.of(/^\/[0-9]{7}$/);

namespaces.on('connect', function(socket) {

  const namespace = socket.nsp;

  socket.broadcast.emit('connected peer');

  socket.on('signal', function(data) {
    socket.broadcast.emit('signal', data);
  });

  socket.on('disconnect', function() {
    namespace.emit('disconnected peer');
  });

});
```

Two features of the signaling channel set the stage for the code we'll write in the browser: a set of events and a precise seven-digit namespace. Let's look at the events first.

Exploring the Signaling Channel's Events

The signaling channel handles seven events: three that it listens for—connect, signal, and disconnect—and four that it emits—connected peer, signal, and disconnected peer, plus a connect event that Socket.IO emits automatically.

When a client connects, the signaling channel will broadcast the connected peer event to other connected clients. When a client sends a signal, the signaling channel will rebroadcast the signal event and its data, essentially acting as a repeater. And finally, when a client disconnects, the signaling channel will emit a disconnected peer event.

That captures everything a basic signaling channel needs to do: listen for connections, listen for and repeat signals, and listen for disconnections. That's it. The code we write in the browser to establish a peer connection will trigger or respond to each of those events.

Namespacing the Signaling Channel

The video-call app you're building is meant to scale: it will allow multiple different independent pairs of peers to connect to each other simultaneously. That is no different from how Zoom or Google Meet works. When you use

Zoom, you and the person you want to talk to must share (over some *other* signaling channel!) a unique URL like https://fake-example.zoom.us/j/72072139453. You'll also be building unique URLs similar to that.

The opening lines of the signaling channel in server.js use a regular expression to verify the structure of a namespace, which is similar to a meeting code in Zoom. With a shared namespace, one pair of peers can negotiate their connection over the namespace /0000001, while another pair can connect simultaneously over /0000002. Their signals will never cross.

Those are very easy to guess patterns, of course, so we'll write a function in the browser to test or generate a random number that matches the namespace's expected seven-digit pattern on the server. We'll make the namespace easy for users to share by attaching it as a hash on the URL, something like https://localhost/basic-p2p/#1234567. (Later in the book, we'll use the server to generate namespaces using URL paths rather than hashes.)

At the very bottom of the main.js file, you will find a section for utility functions. Here I've written a function called prepareNamespace() that takes two arguments. The hash argument will work with an existing hash, likely as reported by the browser from window.location.hash. The second argument, set_location, is a Boolean value (true or false) for setting the prepared namespace on window.location.hash. Look through the whole thing, and then let's walk through it together.

demos/basic-p2p/js/main.js

```
/**
 *  Utility Functions
 */
function prepareNamespace(hash, set_location) {
  let ns = hash.replace(/^#/, ''); // remove # from the hash
  if (/^[0-9]{7}$/.test(ns)) {
    console.log('Checked existing namespace', ns);
    return ns;
  }
```

```
  ns = Math.random().toString().substring(2, 9);
  console.log('Created new namespace', ns);
  if (set_location) window.location.hash = ns;
  return ns;
}
```

The local variable `ns` uses the `replace()` string method with a regular expression and empty string to remove the # (octothorp) that `window.location.hash` always returns. With the octothorp removed, the function can check the namespace's value against another regular expression pattern: `/^[0-9]{7}$/`. That is almost identical to the signaling channel's pattern you already saw in `server.js` (its pattern must also check for the slash that Socket.IO prepends to namespaces).

Demystifying Regular Expressions

If you've never worked with regular expressions before, or if the phrase *regular expressions* makes your palms sweat and your chest tighten, please try to relax. Their basic purpose is pretty easy to summarize: regular expressions describe patterns.

Let's examine the regular-expression pattern `/^[0-9]{7}$/` from the inside out. A pattern that matches a sequence of seven digits, 0 through 9, looks like this: `[0-9]{7}`. `[0-9]` represents all numbers in the range zero to nine. The seven in curly braces, `{7}`, means that we want the numbers in the range to appear seven times in a row. Easy enough, right? You'd think. The problem is that any string of text that includes seven digits in a row—no matter what other text the string includes—would still satisfy a regular expression pattern specifying seven sequential digits.

To make sure the namespace is only ever exactly seven digits and therefore prevent potentially malicious characters from ever reaching the signaling channel, the pattern opens with a caret: `^`. The caret is a regular-expression symbol that marks the very beginning of a one-line string. Similarly, a dollar sign `$` marks the very end of the string. Without the dollar sign, the namespace `0011222-EVIL-BUSINESS-HERE!` would still match, because it opens with seven digits.

That kind of thing is probably why most people find themselves so confused and angered by regular expressions: they are incredibly, infuriatingly literal. *Hey, you said you wanted seven digits–I found you seven digits!* Ask regular expressions for a haircut, and they'll cut exactly one of your hairs. They are the dad jokes of computer programming.

Back to the pattern at hand: bookending the regular expression are slashes, `/`, which demarcate regular expressions in JavaScript, just like quotation marks or backticks demarcate strings. Put it all together, and this little creep

should look a tad less imposing: /^[0-9]{7}$/. Seven digits in a row, and seven digits only. No more, no less—and nothing else. That's the pattern for our namespace.

Making Use of the Namespace

And now back to the prepareNamespace() function definition itself: With the regular expression in place, we call the test() method on it and pass along the namespace we have on hand, ns. If the test passes, we return the octothorp-free namespace, ns.

But if the test fails because either the hash doesn't match our pattern or there's no hash at all, the function generates a new random hash in a complex one-liner:

demos/basic-p2p/js/main.js

```
ns = Math.random().toString().substring(2, 9);
```

That generates a random number and converts it to a string. The substring() method gets called to remove the first two characters: numbers generated by Math.random() always begin with 0. The second argument to substring(), 9, ensures that we get a seven-digit number, thanks to the first two characters being discarded by the first argument, 2.

If set_location is true, the new namespace gets set on the URL by assigning it to window.location.hash.

Finally, the function returns the value of ns, which is the brand-new, randomly generated hash.

With the prepareNamespace() function definition in place, you can put it to work at the top of your file and assign its output–the returned namespace–to a namespace variable:

demos/basic-p2p/js/main.js

```
const namespace = prepareNamespace(window.location.hash, true);
```

Let's do something user-facing with the namespace variable. In the user-interface area of the JavaScript file, add another one-liner that sets the h1 text to welcome users to a correctly namespaced room:

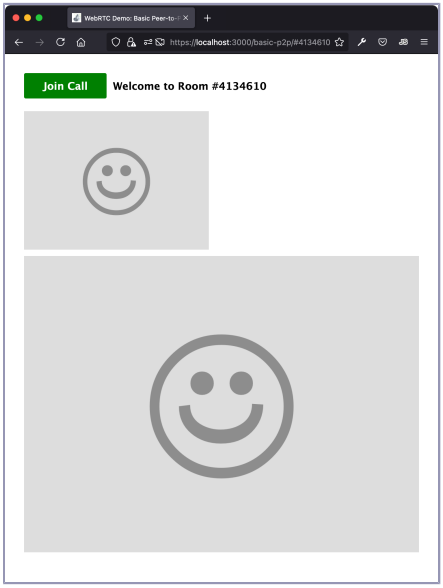
demos/basic-p2p/js/main.js

```
/**
 *  User-Interface Setup
 */

document.querySelector('#header h1')
  .innerText = 'Welcome to Room #' + namespace;

document.querySelector('#call-button')
  .addEventListener('click', handleCallButton);
```

Reload the page at https://localhost:3000/basic-p2p/. You should see two things: a random, seven-digit hash appended to the URL, something like https://localhost:3000/basic-p2p/#4134610, and that same hash repeated in the text of the page's first-level heading, like this:



Those adjustments are basically cosmetic, changing only the appearance of the URL and page. Now let's use the namespace variable to connect to the signaling channel.