

Extracted from:

Programming WebRTC

Build Real-Time Streaming Applications for the Web

This PDF file contains pages extracted from *Programming WebRTC*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2023 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

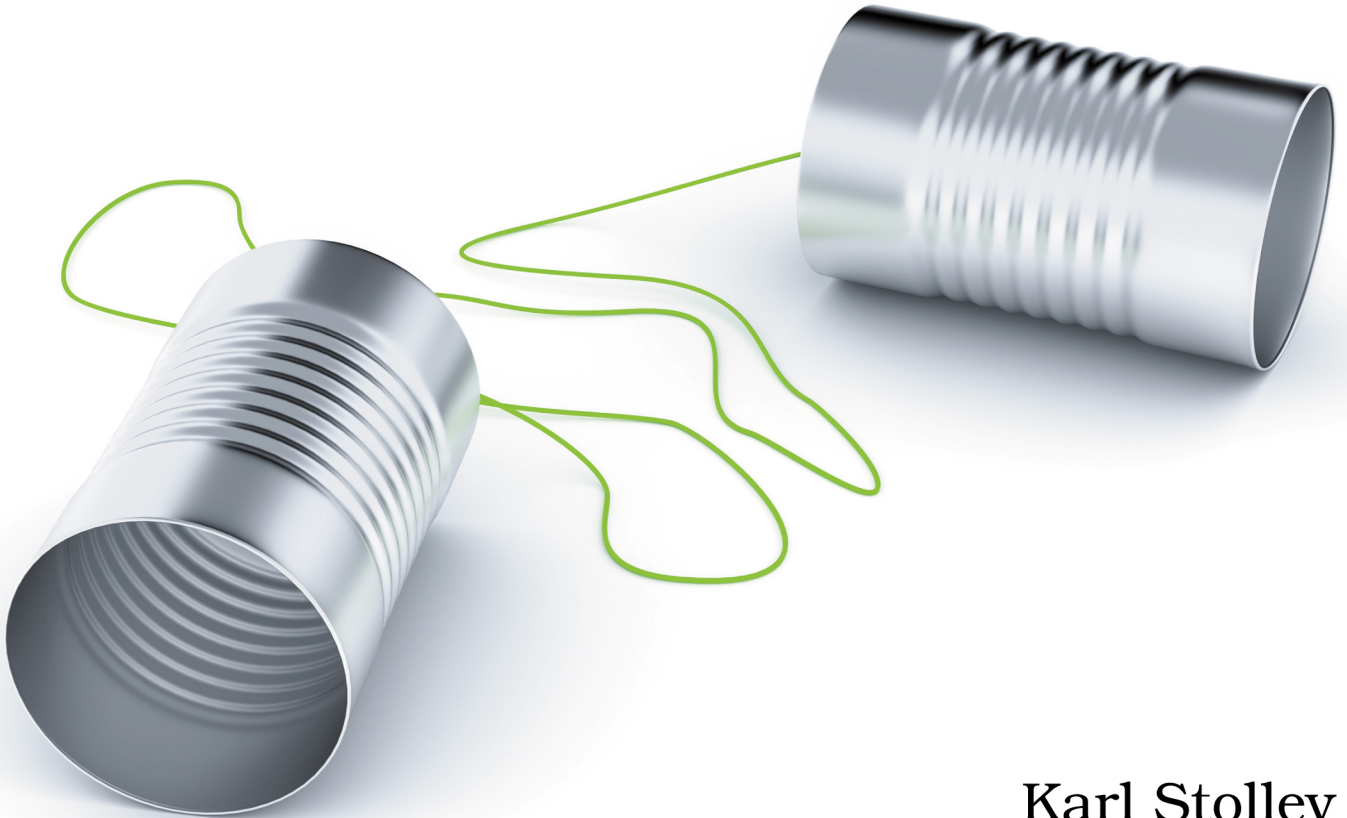
The Pragmatic Bookshelf

Raleigh, North Carolina

The
Pragmatic
Programmers

Programming WebRTC

Build Real-Time Streaming
Applications for the Web



Karl Stolley
edited by Michael Swaine

Programming WebRTC

Build Real-Time Streaming Applications for the Web

Karl Stolley

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit <https://pragprog.com>.

For sales, volume licensing, and support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Copyright © 2023 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 978-1-68050-903-8

Encoded using the finest acid-free high-entropy binary digits.

Book version: B5.0—February 6, 2023

Structuring Chat Messages in JSON

Instead of sending messages as unadorned strings, as we did in the last chapter, let's use JSON to structure some metadata with each message sent over the chat. To start with, the remote peer will use the metadata's content to acknowledge each message received. We can even add a little CSS so that, on the sender's side, there is a visual difference between messages that have been received by the remote peer and those that have not.

Preparing and Sending JSON

Sending JSON isn't much different from sending strings: JSON, or JavaScript Object Notation, is just a fancy kind of string. Let's rework `handleMessageForm()` so that it builds a small object literal on message in place of the message strings we relied on in the previous chapter. We'll set up two properties on the message object: the text of the message, and a timestamp, which uses the `Date.now()` class method to generate a Unix timestamp in milliseconds.¹ The timestamp will uniquely identify each sent message:

demos/dc-chat-json/js/main.js

```
function handleMessageForm(event) {
  event.preventDefault();
  const input = document.querySelector('#chat-msg');
  ➤ const message = {};
  ➤ message.text = input.value;
  ➤ message.timestamp = Date.now();
  ➤ if (message.text === '') return;

  appendMessage('self', '#chat-log', message);

  sendOrQueueMessage($peer, message);

  input.value = '';
}
```

If you're like me, you much prefer working with JavaScript objects directly. Let's set up the `sendOrQueueMessage()` function to make a JSON string out of the message object at the last possible moment. We'll do that by calling `JSON.stringify()` inside the call to the data channel's `send()` method:

demos/dc-chat-json/js/main.js

```
function sendOrQueueMessage(peer, message, push = true) {
  const dc = peer.chatChannel;
  if (!dc || dc.readyState !== 'open') {
    queueMessage(message, push);
    return;
  }
}
```

1. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/now

```

    }
    try {
➤   dc.send(JSON.stringify(message));
    } catch(e) {
      console.error('Error sending message:', e);
      queueMessage(message, push);
    }
  }
}

```

Messages that wind up on the queue will remain as JavaScript objects. That's the benefit of stringifying objects within the `send()` method, even if it makes the method call look a little crowded: an object is only JSON when it's sent. At the same time, you can glance at a call like `dc.send(JSON.stringify(message))` and know that there's JSON involved. Best of all, `JSON.stringify()` relieves you of the error-prone business of constructing a JSON string yourself, by hand.

With `handleMessageForm()` now handling messages and objects, and `sendOrQueueMessage()` properly sending messages as JSON, we need to update how messages are appended to the chat log. Two changes are all we need: referencing `message.text` from the new message object that we're passing in and preserving a reference to the Unix timestamp in a `data-timestamp` attribute.

`demos/dc-chat-json/js/main.js`

```

function appendMessage(sender, log_element, message) {
  const log = document.querySelector(log_element);
  const li = document.createElement('li');
  li.className = sender;
➤   li.innerText = message.text;
➤   li.dataset.timestamp = message.timestamp;
  log.appendChild(li);
  if (log.scrollTo) {
    log.scrollTo({
      top: log.scrollHeight,
      behavior: 'smooth'
    });
  } else {
    log.scrollTop = log.scrollHeight;
  }
}

```

If you've not used `data-` attributes before,² they are a super useful feature for storing out-of-band data in HTML. The `HTMLElement.dataset` property that you referenced as `li.dataset.timestamp` makes it pretty straightforward to read and write `data-` attributes using JavaScript too.³

2. https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/data-

3. <https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/dataset>

Acknowledging Received Messages

Let's set up the logic to send a response that acknowledges each message a peer receives. This will take two steps: first, the receiving peer must parse the JSON message and check the resulting object for an `id` attribute. Only responses will have `id` attributes—messages have only `text` and `timestamp` attributes. When a message comes in, we create a response object whose `id` will take the `timestamp` value of the incoming message. The response object will also include its own `timestamp`, which captures the moment the message was received. Let's also reference a `handleResponse()` function that we'll build next for—you guessed it!—handling incoming responses:

`demos/dc-chat-json/js/main.js`

```
function addChatChannel(peer) {
  peer.chatChannel =
    peer.connection.createDataChannel('text chat',
      { negotiated: true, id: 100 });

  peer.chatChannel.onmessage = function(event) {
    ▶ const message = JSON.parse(event.data);
    ▶ if (!message.id) {
    ▶   // Prepare a response and append an incoming message
    ▶   const response = {
    ▶     id: message.timestamp,
    ▶     timestamp: Date.now()
    ▶   };
    ▶   sendOrQueueMessage(peer, response);
    ▶   appendMessage('peer', '#chat-log', message);
    ▶ } else {
    ▶   // Handle an incoming response
    ▶   handleResponse(message);
    ▶ }
  };

  peer.chatChannel.onclose = function(event) {
    console.log('Chat channel closed.');
```

```
  };

  peer.chatChannel.onopen = function(event) {
    console.log('Chat channel opened.');
```

```
    while ($self.messageQueue.length > 0 &&
      peer.chatChannel.readyState === 'open') {
      console.log('Attempting to send a message from the queue...');
      // get the message at the front of the queue:
      let message = $self.messageQueue.shift();
      sendOrQueueMessage(peer, message, false);
    }
  };
}
```


Should the call or data channel fail in the time it takes to respond, the `sendOrQueueMessage()` function steps in and queues the response. Responses will of course be sent from the queue just like any other message, just as soon as the data channel opens again. Pretty snazzy, right?

Now you can turn your attention to building out the `handleResponse()` function. What it will do is use `document.querySelector()`, backed by CSS attribute selectors,⁴ to hunt down in the DOM the exact list item for the appended message that the remote peer is acknowledging. That message will take a `received` class. If more than a second elapses between the message being composed and the acknowledgement, the list item also takes a `delayed` class:

```
demos/dc-chat-json/js/main.js
function handleResponse(response) {
  const sentItem = document
    .querySelector(`#chat-log *[data-timestamp="${response.id}"]`);
  const classes = ['received'];
  if (response.timestamp - response.id > 1000) {
    classes.push('delayed');
  }
  sentItem.classList.add(...classes);
}
```

If you've been quietly irritated by my use of the old-school `className` property across all the earlier code, I hope you can now breathe easier with the call to the modern `classList` API.⁵ Note also the use of the fancy spread syntax,⁶ `...classes`, to ease passing in the array's values as a series of comma-separated arguments like `.add()` expects. The `delayed` class is only pushed onto to the `classes` array if there is more than 1000 milliseconds' difference between the timestamps for when a message was composed and when it was acknowledged.

Let's reference and style both the `received` and `delayed` classes in CSS:

```
demos/dc-chat-json/css/screen.css
#chat-log .self {
  background: #009;
  color: #EEE;
  ➤ opacity: 0.3;
  ➤ float: right;
}
➤ #chat-log .self.received {
➤   opacity: 1.0;
➤ }
➤ #chat-log .self.received.delayed {
```

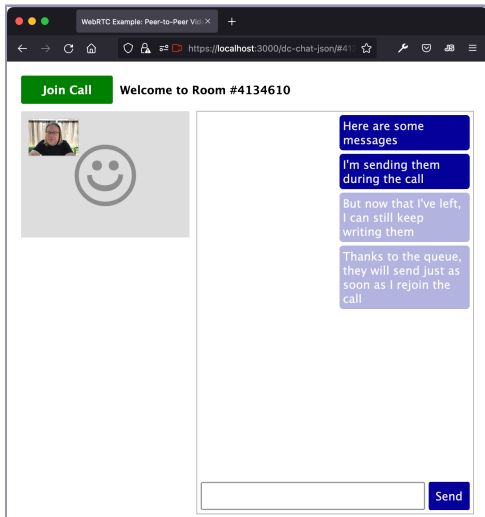
4. https://developer.mozilla.org/en-US/docs/Web/CSS/Attribute_selectors

5. <https://developer.mozilla.org/en-US/docs/Web/API/Element/classList>

6. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

- transition: opacity 0.4s;
- }

A user's sent messages might now appear faded out when appended to the chat log, thanks to a semi-transparent opacity value. The received class restores full opacity (opacity: 1.0). The delayed class, however, let's us add a nice little touch to the interface: delayed messages will appear to fade up from semi-transparency to full opacity over a little less than half a second, thanks to the CSS transition property.⁷ Messages with less than a one-second delay will appear appended at full opacity. That prevents users from being annoyed by messages always fading up during a low-latency call—which would have the effect of making the call feel like molasses. Here you can see acknowledged and queued messages sent after leaving the call:



Excellent. You're now sending and receiving JSON over data channels. You've also implemented a simple method for acknowledging messages as a peer receives them. And thanks to some carefully crafted DOM attributes and CSS, you've seen once more just how tight the connection is between WebRTC and interface design.

Let's apply that same tightly connected approach to a fundamental feature of WebRTC applications: giving users the ability to toggle their cameras and microphones on and off.

7. <https://developer.mozilla.org/en-US/docs/Web/CSS/transition>