

# WHAT just happened?

Reacting to Events on the Server

# Administrative Notes

- @pilif on twitter
- pilif on github
- working at Sensational AG

- @pilif on twitter
- pilif on github
- working at Sensational AG
- strongly dislike shirts

"".length() is still 2

There will be some 

Demo first™

# Server-Side Events

- Inform users about *stuff* happening while they are using the site
- Edits made to the current resource by other people
- Chat Messages
- Mobile Devices interacting with the Account

# Additional Constraints

- Must not lose events
- Events must be unique
- Must work with shared sessions
- Separate channels per user
- Must work\* even when hand-written daemons are down
- Must work\* in development without massaging daemons

# Not losing events

- Race condition between event happening and infrastructure coming up on page load
- Need to persist events
- Using a database
- Using a sequence (auto increment ID) to identify last sent event
- Falling back to timestamps if not available (initial page load)

But back to the topic

# TIMTWWTDI\*

- Short Polling
- Long Polling
- EventSource
- Web Sockets

\* yes. I'm that old

# Short Polling

- Are we there yet?
- Are we there yet?
- Are we there yet?
- And now?



# Long Polling

- Send a Query to the Server
- Have the server only\* reply when an event is available
- Keep the connection open otherwise
- Response means: event has happened
- Have the client reconnect immediately

# Server-Sent Events

- <http://www.w3.org/TR/eventsource/>
- Keeping a connection open to the server
- Server is sending data as text/event-stream
- Colon-separated key-value data.
- Empty line separates events.

# WebSockets

- «TCP/IP in your browser»\*
- Full Duplex
- Message passing
- Persistent connection between Browser and Server

Let's try them

Surprise Demo™

also Demo first™

# index.html

```
<script>
  (function(){
    var channel = new EventChannel();
    var log_ws = $('#websockets');
    $(channel.bind('cheese_created', function(e){
      log_ws.prepend($('- ').text(
        e.pieces + ' pieces of ' + e.cheese_type
      ));
    }));
  })();
</script>

```

# index.html

```
<script>
  (function(){
    var channel = new EventChannel();
    var log_ws = $('#websockets');
    $(channel.bind('cheese_created', function(e){
      log_ws.prepend($('<li>').text(
        e.pieces + ' pieces of ' + e.cheese_type
      ));
    }));
  })();
</script>
```

# index.html

```
<script>
  (function(){
    var channel = new EventChannel();
    var log_ws = $('#websockets');
    $(channel.bind('cheese_created', function(e){
      log_ws.prepend($('- ').text(
        e.pieces + ' pieces of ' + e.cheese_type
      ));
    }));
  })();
</script>

```

# index.html

```
<script>
  (function(){
    var channel = new EventChannel();
    var log_ws = $('#websockets');
    $(channel.bind('cheese_created', function(e){
      log_ws.prepend($('- ').text(
        e.pieces + ' pieces of ' + e.cheese_type
      ));
    }));
  })();
</script>

```

# publish.js

- Creates between 5 and 120 pieces of random Swiss cheese
- Publishes an event about this
- We're using redis as our Pub/Sub mechanism, but you could use other solutions too
- Sorry for the indentation, but code had to fit the slide

```
var cheese_types = ['Emmentaler',  
    'Appenzeller', 'Gruyère',  
    'Vacherin', 'Sprinz'  
];  
  
function create_cheese(){  
    return {  
        pieces: Math.floor(Math.random()  
            * 115) + 5,  
        cheese_type:  
            cheese_types[Math.floor(  
                Math.random()  
                *cheese_types.length  
            )]  
    }  
}  
  
var cheese_delivery =  
    create_cheese();  
publish(cheese_delivery);
```

# Web Sockets

# Server

- Do not try this at home
- Use a library. You might know of socket.io – Me personally, I used ws.
- Our code: only 32 lines.

# This is it

```
var WebSocketServer = require('ws').Server;
var redis = require('redis');

var wss = new WebSocketServer({port: 8080});
wss.on('connection', function(ws) {
  var client = redis.createClient(6379, 'localhost');
  ws.on('close', function(){
    client.end();
  });
  client.select(2, function(err, result){
    if (err) {
      console.log("Failed to set redis database");
      return;
    }
    client.subscribe('channels:cheese');
    client.on('message', function(chn, message){
      ws.send(message);
    });
  });
});
```

# Actually, this is the meat

```
client.subscribe('channels:cheese');  
client.on('message', function(chn, message){  
  ws.send(message);  
});
```

# And here's the client

```
(function(window){  
    window.EventChannelWs = function(){  
        var socket = new WebSocket("ws://localhost:8080/");  
        var self = this;  
        socket.onmessage = function(evt){  
            var event_info = JSON.parse(evt.data);  
            var evt = jQuery.Event(event_info.type, event_info.data);  
            $(self).trigger(evt);  
        }  
    }  
})(window);
```

# And here's the client

```
(function(window){  
    window.EventChannelWs = function(){  
        var socket = new WebSocket("ws://localhost:8080/");  
        var self = this;  
        socket.onmessage = function(evt){  
            var event_info = JSON.parse(evt.data);  
            var evt = jQuery.Event(event_info.type, event_info.data);  
            $(self).trigger(evt);  
        }  
    }  
})(window);
```

# And here's the client

```
(function(window){  
    window.EventChannelWs = function(){  
        var socket = new WebSocket("ws://localhost:8080/");  
        var self = this;  
        socket.onmessage = function(evt){  
            var event_info = JSON.parse(evt.data);  
            var evt = jQuery.Event(event_info.type, event_info.data);  
            $(self).trigger(evt);  
        }  
    }  
})(window);
```

# And here's the client

```
(function(window){  
    window.EventChannelWs = function(){  
        var socket = new WebSocket("ws://localhost:8080/");  
        var self = this;  
        socket.onmessage = function(evt){  
            var event_info = JSON.parse(evt.data);  
            var evt = jQuery.Event(event_info.type, event_info.data);  
            $(self).trigger(evt);  
        }  
    }  
})(window);
```

# Sample was very simple

- No synchronisation with server for initial event
- No fallback when the web socket server is down
- No reverse proxy involved
- No channel separation

Flip-Side

```
pilif@fang: ~/Development/server-side-events/sledgehammer/backend (zsh)
⇒ find node_modules -name '*.js' | grep -Ev '(examples|test|bench)' | xargs wc
-l | tail -n 1
    5908 total
pilif@fang:~/Development/server-side-events/sledgehammer/backend|master
⇒ find node_modules -name '*.c' | grep -Ev '(examples|test|bench)' | xargs wc -
l | tail -n 1
    3368 total
pilif@fang:~/Development/server-side-events/sledgehammer/backend|master
⇒ find node_modules -name '*.cc' | grep -Ev '(examples|test|bench)' | xargs wc
-l | tail -n 1
    508 total
pilif@fang:~/Development/server-side-events/sledgehammer/backend|master
⇒
```

# Powering Your 39 Lines

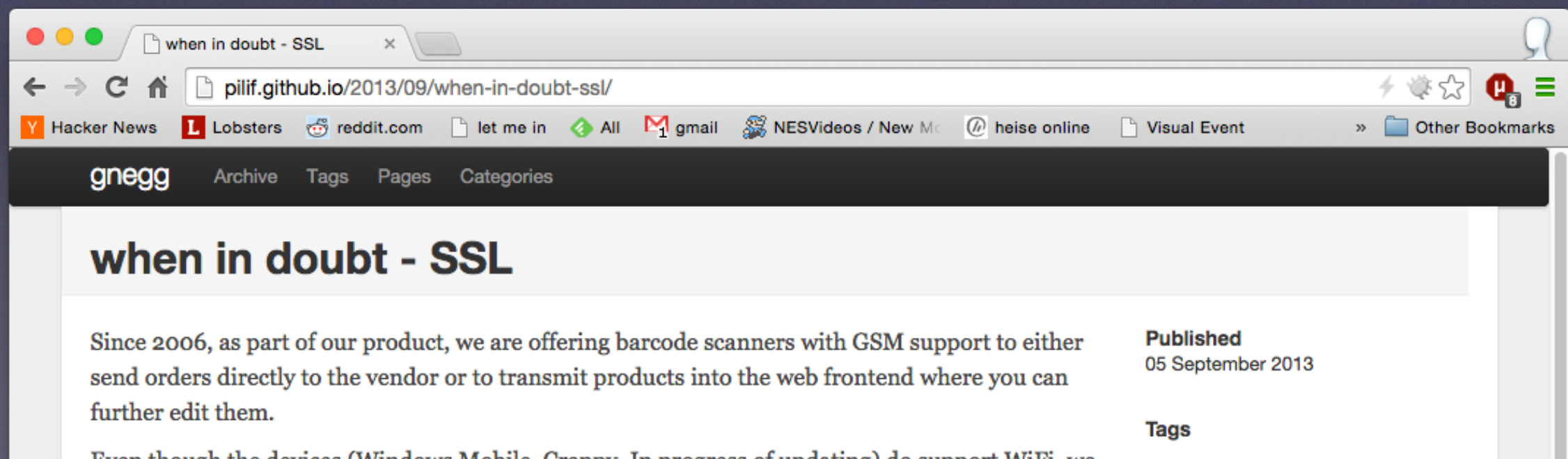
- 6K lines of JavaScript code
- Plus 3.3K lines of C code
- Plus 508 lines of C++ code
- Which is the body that you actually *run* (excluding tests and benchmarks)
- Some of which redundant because NPM

# WebSockets are a bloody mess™

- RFC6455 is 71 pages long
- Adding a lot of bit twiddling to intentionally break proxy servers
- Proxies that work might only actually work\*
- Many deployments require a special port to run over

# Use SSL.

# By the Gods. Use SSL



# EventSource

# Client

```
var cheese_channel = new EventSource(url);
var log_source = $('#eventsourcing');
cheese_channel.addEventListener('cheese_created', function(e){
    var data = JSON.parse(e.data);
    log_source.prepend($('- ').text(
        data.pieces + ' pieces of ' + data.cheese_type
    ));
});

```

# Client

```
var cheese_channel = new EventSource(url);  
var log_source = $('#eventsourcing');  
cheese_channel.addEventListener('cheese_created', function(e){  
    var data = JSON.parse(e.data);  
    log_source.prepend($('- ').text(  
        data.pieces + ' pieces of ' + data.cheese_type  
    ));  
});

```

# Client

```
var cheese_channel = new EventSource(url);  
var log_source = $('#eventsources');  
cheese_channel.addEventListener('cheese_created', function(e){  
    var data = JSON.parse(e.data);  
    log_source.prepend($('- ').text(  
        data.pieces + ' pieces of ' + data.cheese_type  
    ));  
});

```

# Client

```
var cheese_channel = new EventSource(url);
var log_source = $('#eventsources');
cheese_channel.addEventListener('cheese_created', function(e){
    var data = JSON.parse(e.data);
    log_source.prepend($('- ').text(
        data.pieces + ' pieces of ' + data.cheese_type
    ));
});

```

# Server

- Keeps the connection open
- Sends blank-line separated groups of key/value pairs as events happen
- Can tell the client how long to wait when reconnecting

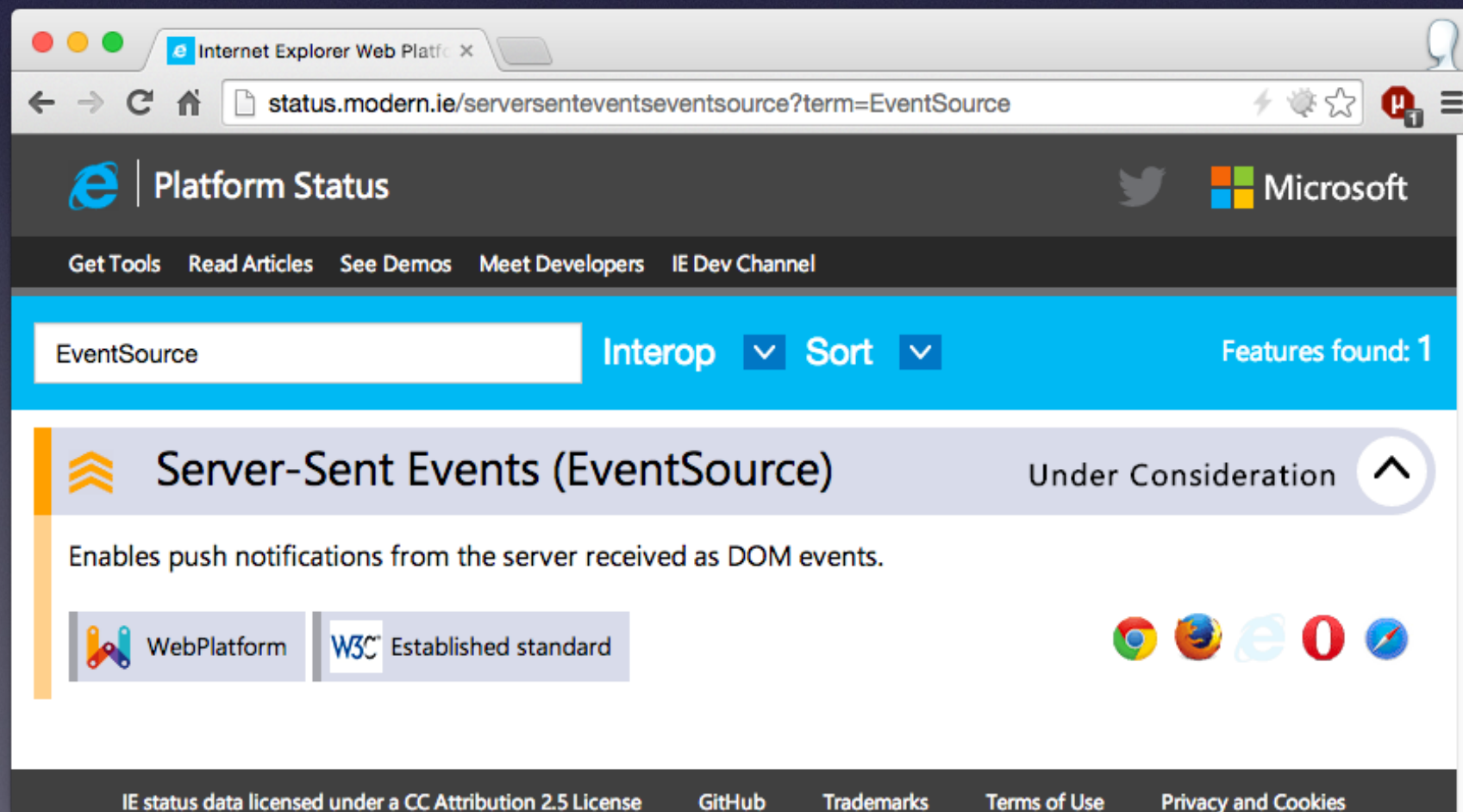
```
pilif@fang: ~ (zsh)
pilif@fang:~| => curl -i $url
HTTP/1.1 200 OK
Content-Type: text/event-stream
Date: Wed, 23 Jul 2014 14:25:22 GMT
Connection: keep-alive
Transfer-Encoding: chunked

event: cheese_created
data: {"pieces":92,"cheese_type":"Gruyère"}
id: 434
retry:0

^C
pilif@fang:~| => _
```

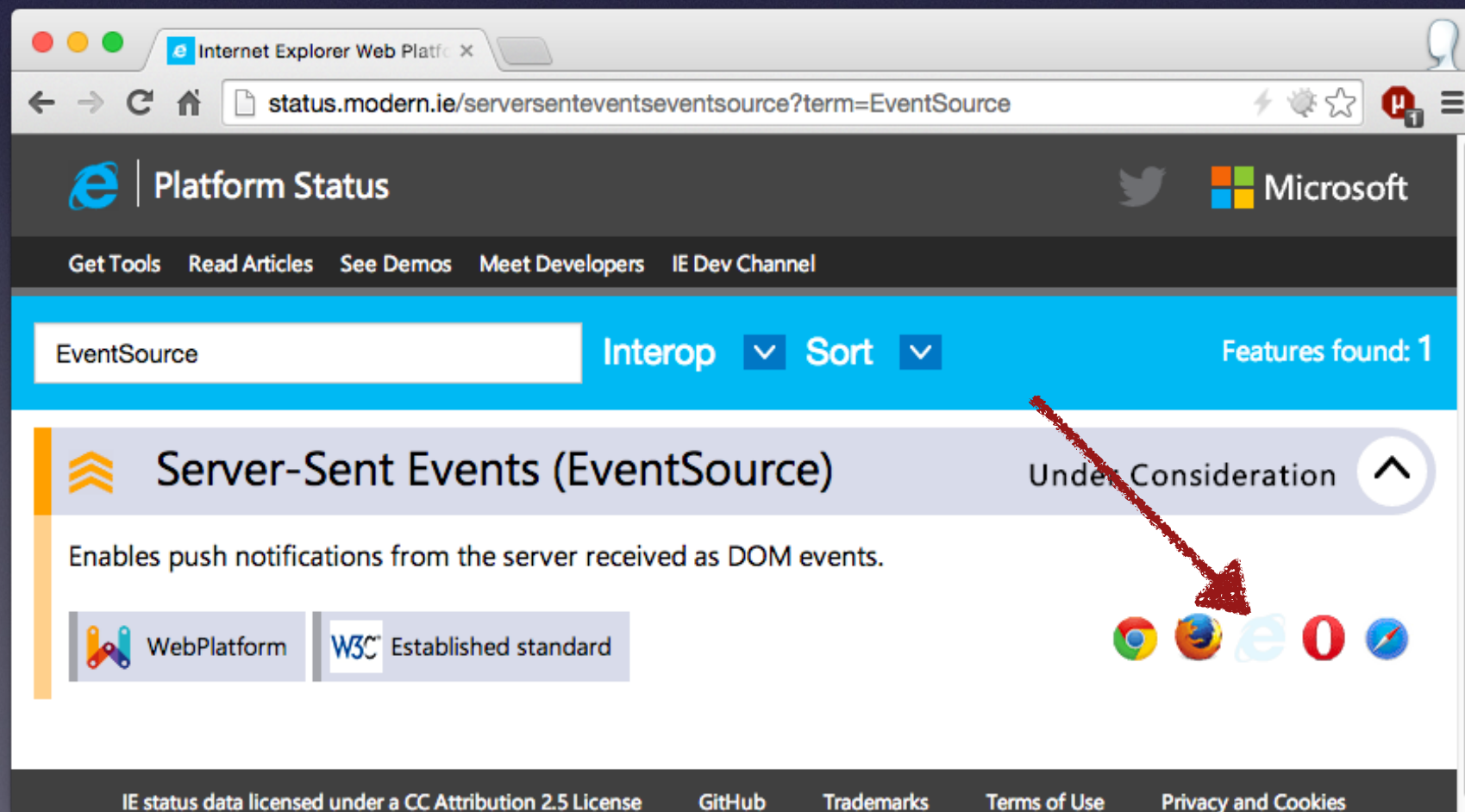
# Disillusioning

- Bound to the 6-connections per host rule
- Still needs manual synchronising if you don't want to lose events
- Browser support is as always



# Disillusioning

- Bound to the 6-connections per host rule
- Still needs manual synchronising if you don't want to lose events
- Browser support is as always



# Long Polling

# I like it

- Works even with IE6 (god forbid you have to do this)
- Works fine with proxies
  - On both ends
- Works fine over HTTP
- Needs some help due to the connection limit
- Works even when your infrastructure is down\*

# Production code

- The following code samples form the basis of the initial demo
- It's production code
- No support issue caused by this.
- Runs fine in a developer-hostile environment

Caution:  ahead

# Synchronising using the database

```
events_since_id = (channel, id, cb)->
    q = """
        select * from events
        where channel_id = $1 and id > $2
        order by id asc
        """
    query q, [channel, id], cb

events_since_time = (channel, ts, cb)->
    q = """
        select * from events o
        where channel_id = $1
        and ts > (SELECT TIMESTAMP WITH TIME ZONE 'epoch'
                  + $2 * INTERVAL '1 second'
                  )
        order by id asc
        """
    query q, [channel, ts], cb
```

# The meat

```
handle_subscription = (c, message)->
    fetch_events channel, last_event_id, (err, evts)->
        return http_error 500, 'Failed to get event data' if err
        abort_processing = write res, evts, true
        last_event_id = evts[evts.length-1].id if (evts and evts.length > 0)
        if abort_processing
            unsubscribe channel, handle_subscription
            clear_waiting()
            res.end()

fetch_events channel, last_event_id, (err, evts)->
    return http_error res, 500, 'Failed to get event data: ' + err if err

    last_event_id = evts[evts.length-1].id if (evts and evts.length > 0)
    if waiting() or (evts and evts.length > 0)
        abort_processing = write(res, evts, not waiting());
        if waiting() or abort_processing
            unsubscribe channel, handle_subscription
            res.end()

    set_waiting()
    subscribe channel, handle_subscription
```

- If events are pending
- Or if there's already a connection waiting for the same channel
- Then return the event data immediately
- And tell the client when to reconnect
- The abort\_processing mess is because of support for both EventSource and long-polling

```
if waiting() or (evts and evts.length > 0)
  abort_processing = write(res, evts, not waiting());
if waiting() or abort_processing
  unsubscribe channel, handle_subscription
res.end()
```

- If events are pending
- Or if there's already a connection waiting for the same channel
- Then return the event data immediately
- And tell the client when to reconnect
- The abort\_processing mess is because of support for both EventSource and long-polling

```
if waiting() or (evts and evts.length > 0)
  abort_processing = write(res, evts, not waiting());
if waiting() or abort_processing
  unsubscribe channel, handle_subscription
res.end()
```

- If events are pending
- Or if there's already a connection waiting for the same channel
- Then return the event data immediately
- And tell the client when to reconnect
- The abort\_processing mess is because of support for both EventSource and long-polling

```
if waiting() or (evts and evts.length > 0)
  abort_processing = write(res, evts, not waiting());
if waiting() or abort_processing
  unsubscribe channel, handle_subscription
res.end()
```

- If events are pending
- Or if there's already a connection waiting for the same channel
- Then return the event data immediately
- And tell the client when to reconnect
- The abort\_processing mess is because of support for both EventSource and long-polling

```
if waiting() or (evts and evts.length > 0)
  abort_processing = write(res, evts, not waiting());
if waiting() or abort_processing
  unsubscribe channel, handle_subscription
res.end()
```

- If events are pending
- Or if there's already a connection waiting for the same channel
- Then return the event data immediately
- And tell the client when to reconnect
- The abort\_processing mess is because of support for both EventSource and long-polling

```
if waiting() or (evts and evts.length > 0)
  abort_processing = write(res, evts, not waiting());
  if waiting() or abort_processing
    unsubscribe channel, handle_subscription
    res.end()
```

# Waiting

```
handle_subscription = (c, message)->
  fetch_events channel, last_event_id, (err, evts)->
    return http_error 500, 'Failed to get event data' if err
    abort_processing = write res, evts, true
    last_event_id = evts[evts.length-1].id if (evts and evts.length > 0)
    if abort_processing
      unsubscribe channel, handle_subscription
      clear_waiting()
      res.end()

set_waiting()
subscribe channel, handle_subscription
```

# Waiting

```
handle_subscription = (c, message)->
  fetch_events channel, last_event_id, (err, evts)->
    return http_error 500, 'Failed to get event data' if err
    abort_processing = write res, evts, true
    last_event_id = evts[evts.length-1].id if (evts and evts.length > 0)
    if abort_processing
      unsubscribe channel, handle_subscription
      clear_waiting()
      res.end()

set_waiting()
subscribe channel, handle_subscription
```

# Waiting

```
handle_subscription = (c, message)->
  fetch_events channel, last_event_id, (err, evts)->
    return http_error 500, 'Failed to get event data' if err
    abort_processing = write res, evts, true
    last_event_id = evts[evts.length-1].id if (evts and evts.length > 0)
    if abort_processing
      unsubscribe channel, handle_subscription
      clear_waiting()
      res.end()

set_waiting()
subscribe channel, handle_subscription
```

# Waiting

```
handle_subscription = (c, message)->
    fetch_events channel, last_event_id, (err, evts)->
        return http_error 500, 'Failed to get event data' if err
        abort_processing = write res, evts, true
        last_event_id = evts[evts.length-1].id if (evts and evts.length > 0)
        if abort_processing
            unsubscribe channel, handle_subscription
            clear_waiting()
            res.end()

set_waiting()
subscribe channel, handle_subscription
```

# Waiting

```
handle_subscription = (c, message)->
  fetch_events channel, last_event_id, (err, evts)->
    return http_error 500, 'Failed to get event data' if err
    abort_processing = write res, evts, true
    last_event_id = evts[evts.length-1].id if (evts and evts.length > 0)
    if abort_processing
      unsubscribe channel, handle_subscription
      clear_waiting()
      res.end()

set_waiting()
subscribe channel, handle_subscription
```

LOL - Boolean  
parameter!!!

# Waiting

```
handle_subscription = (c, message)->
  fetch_events channel, last_event_id, (err, evts)->
    return http_error 500, 'Failed to get event data' if err
    abort_processing = write res, evts, true
    last_event_id = evts[evts.length-1].id if (evts and evts.length > 0)
    if abort_processing
      unsubscribe channel, handle_subscription
      clear_waiting()
      res.end()

set_waiting()
subscribe channel, handle_subscription
```

# Fallback

- Our fronted code connects to /e.php
- Our reverse proxy redirects that to the node daemon
- If that daemon is down or no reverse proxy is there, there's an actual honest-to god /e.php ...
- ...which follows the exact same interface but is always\* short-polling

# Client is more complicated

```
poll: =>
  url = "#{@endpoint}/#{channel}/#{wait_id}"
  $.ajax url,
    cache: false,
    dataType: 'json',
    headers:
      'Last-Event-Id': @last_event_id
  success: (data, s, xhr) =>
    return unless @enabled
    @fireAll data

    reconnect_in = parseInt xhr.getResponseHeader('x-ps-reconnect-in'), 10
    reconnect_in = 10 unless reconnect_in >= 0
    setTimeout @poll, reconnect_in*1000 if @enabled
  error: (xhr, textStatus, error) =>
    return unless @enabled
    # 504 means nginx gave up waiting. This is totally to be
    # expected and we can just treat it as an invitation to
    # reconnect immediately. All other cases are likely bad, so
    # we remove a bit of load by waiting a really long time
    # 12002 is the ie proprietary way to report an WinInet timeout
    # if it was registry-hacked to a low ReadTimeout.
    # This isn't a server-error, so we can just reconnect.
    rc = if (xhr.status in [504, 12002]) || (textStatus == 'timeout') then 0 else 10000
    setTimeout @poll, rc if @enabled
```

# Client is more complicated

```
poll: =>
  url = "#{@endpoint}/#{channel}/#{wait_id}"
  $.ajax url,
    cache: false,
    dataType: 'json',
    headers:
      'Last-Event-Id': @last_event_id
    success: (data, s, xhr) =>
      return unless @enabled
      @fireAll data

      reconnect_in = parseInt xhr.getResponseHeader('x-ps-reconnect-in'), 10
      reconnect_in = 10 unless reconnect_in >= 0
      setTimeout @poll, reconnect_in*1000 if @enabled
    error: (xhr, textStatus, error) =>
      return unless @enabled
      # 504 means nginx gave up waiting. This is totally to be
      # expected and we can just treat it as an invitation to
      # reconnect immediately. All other cases are likely bad, so
      # we remove a bit of load by waiting a really long time
      # 12002 is the ie proprietary way to report an WinInet timeout
      # if it was registry-hacked to a low ReadTimeout.
      # This isn't a server-error, so we can just reconnect.
      rc = if (xhr.status in [504, 12002]) || (textStatus == 'timeout') then 0 else 10000
      setTimeout @poll, rc if @enabled
```

# Client is more complicated

```
poll: =>
  url = "#{@endpoint}/#{channel}/#{wait_id}"
  $.ajax url,
    cache: false,
    dataType: 'json',
    headers:
      'Last-Event-Id': @last_event_id
    success: (data, s, xhr) =>
      return unless @enabled
      @fireAll data

      reconnect_in = parseInt xhr.getResponseHeader('x-ps-reconnect-in'), 10
      reconnect_in = 10 unless reconnect_in >= 0
      setTimeout @poll, reconnect_in*1000 if @enabled
    error: (xhr, textStatus, error) =>
      return unless @enabled
      # 504 means nginx gave up waiting. This is totally to be
      # expected and we can just treat it as an invitation to
      # reconnect immediately. All other cases are likely bad, so
      # we remove a bit of load by waiting a really long time
      # 12002 is the ie proprietary way to report an WinInet timeout
      # if it was registry-hacked to a low ReadTimeout.
      # This isn't a server-error, so we can just reconnect.
      rc = if (xhr.status in [504, 12002]) || (textStatus == 'timeout') then 0 else 10000
      setTimeout @poll, rc if @enabled
```

# Client is more complicated

```
poll: =>
  url = "#{@endpoint}/#{channel}/#{wait_id}"
  $.ajax url,
    cache: false,
    dataType: 'json',
    headers:
      'Last-Event-Id': @last_event_id
  success: (data, s, xhr) =>
    return unless @enabled
    @fireAll data

    reconnect_in = parseInt xhr.getResponseHeader('x-ps-reconnect-in'), 10
    reconnect_in = 10 unless reconnect_in >= 0
    setTimeout @poll, reconnect_in*1000 if @enabled
  error: (xhr, textStatus, error) =>
    return unless @enabled
    # 504 means nginx gave up waiting. This is totally to be
    # expected and we can just treat it as an invitation to
    # reconnect immediately. All other cases are likely bad, so
    # we remove a bit of load by waiting a really long time
    # 12002 is the ie proprietary way to report an WinInet timeout
    # if it was registry-hacked to a low ReadTimeout.
    # This isn't a server-error, so we can just reconnect.
    rc = if (xhr.status in [504, 12002]) || (textStatus == 'timeout') then 0 else 10000
    setTimeout @poll, rc if @enabled
```

# Client is more complicated

```
poll: =>
  url = "#{@endpoint}/#{channel}/#{wait_id}"
  $.ajax url,
    cache: false,
    dataType: 'json',
    headers:
      'Last-Event-Id': @last_event_id
  success: (data, s, xhr) =>
    return unless @enabled
    @fireAll data

    reconnect_in = parseInt xhr.getResponseHeader('x-ps-reconnect-in'), 10
    reconnect_in = 10 unless reconnect_in >= 0
    setTimeout @poll, reconnect_in*1000 if @enabled
  error: (xhr, textStatus, error) =>
    return unless @enabled
    # 504 means nginx gave up waiting. This is totally to be
    # expected and we can just treat it as an invitation to
    # reconnect immediately. All other cases are likely bad, so
    # we remove a bit of load by waiting a really long time
    # 12002 is the ie proprietary way to report an WinInet timeout
    # if it was registry-hacked to a low ReadTimeout.
    # This isn't a server-error, so we can just reconnect.
    rc = if (xhr.status in [504, 12002]) || (textStatus == 'timeout') then 0 else 10000
    setTimeout @poll, rc if @enabled
```

# Client is more complicated

```
poll: =>
  url = "#{@endpoint}/#{channel}/#{wait_id}"
  $.ajax url,
    cache: false,
    dataType: 'json',
    headers:
      'Last-Event-Id': @last_event_id
  success: (data, s, xhr) =>
    return unless @enabled
    @fireAll data

    reconnect_in = parseInt xhr.getResponseHeader
    reconnect_in = 10 unless reconnect_in >= 0
    setTimeout @poll, reconnect_in*1000 if @enabled
  error: (xhr, textStatus, error) =>
    return unless @enabled
    # 504 means nginx gave up waiting. This is
    # expected and we can just treat it as an invitation to
    # reconnect immediately. All other cases are likely bad
    # we remove a bit of load by waiting a really long time
    # 12002 is the ie proprietary way to report an WinInet timeout
    # if it was registry-hacked to a low ReadTimeout.
    # This isn't a server-error, so we can just reconnect.
    rc = if (xhr.status in [504, 12002]) || (textStatus == 'timeout') then 0 else 10000
    setTimeout @poll, rc if @enabled
```



# So. Why a daemon?

- Evented architecture lends itself well to many open connections never really using CPU
- You do not want to long-poll with forking architectures
- Unless you have unlimited RAM

# In conclusion

also quite different from what I initially meant to say



**Philip Hofstetter**  
@pilif

+ Follow

by now the “simple” solution is so complicated that the “complex” solution starts to feel compelling :-)

↩ Reply ↻ Retweet ★ Favorite ... More

RETWEET

1



6:49 AM - 23 Jul 2014

<https://twitter.com/pilif/status/491943226258239490>

And that was last  
wednesday

pilif@fang: ~/Development/server-side-events/publish (zsh)

pilif@fang:~/Development/server-side-events/publish|master

⇒ psql -c "select max(id) from events" cheese

max

-----

730

(1 row)

pilif@fang:~/Development/server-side-events/publish|master

⇒

\_

So...



- If your clients use browsers (and IE10+)

- If your clients use browsers (and IE10+)
- and if you have a good reverse proxy

- If your clients use browsers (and IE10+)
- and if you have a good reverse proxy
- and if you can use SSL

- If your clients use browsers (and IE10+)
- and if you have a good reverse proxy
- and if you can use SSL
- then use WebSockets

- If your clients use browsers (and IE10+)
- and if you have a good reverse proxy
- and if you can use SSL
- then use WebSockets
- Otherwise use long polling

- If your clients use browsers (and IE10+)
- and if you have a good reverse proxy
- and if you can use SSL
- then use WebSockets
- Otherwise use long polling
- Also, only use one - don't mix - not worth the effort

- If your clients use browsers (and IE10+)
- and if you have a good reverse proxy
- and if you can use SSL
- then use WebSockets
- Otherwise use long polling
- Also, only use one - don't mix - not worth the effort
- EventSource, frankly, sucks

# Thank you!

- @pilif on twitter
- <https://github.com/pilif/server-side-events>

Also: We are looking for a front-end designer with CSS skills and a backend developer. If you are interested or know somebody, come to me