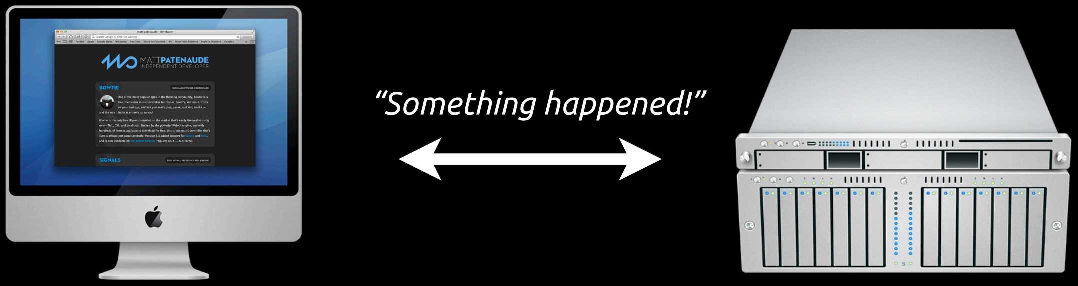
Socket.io

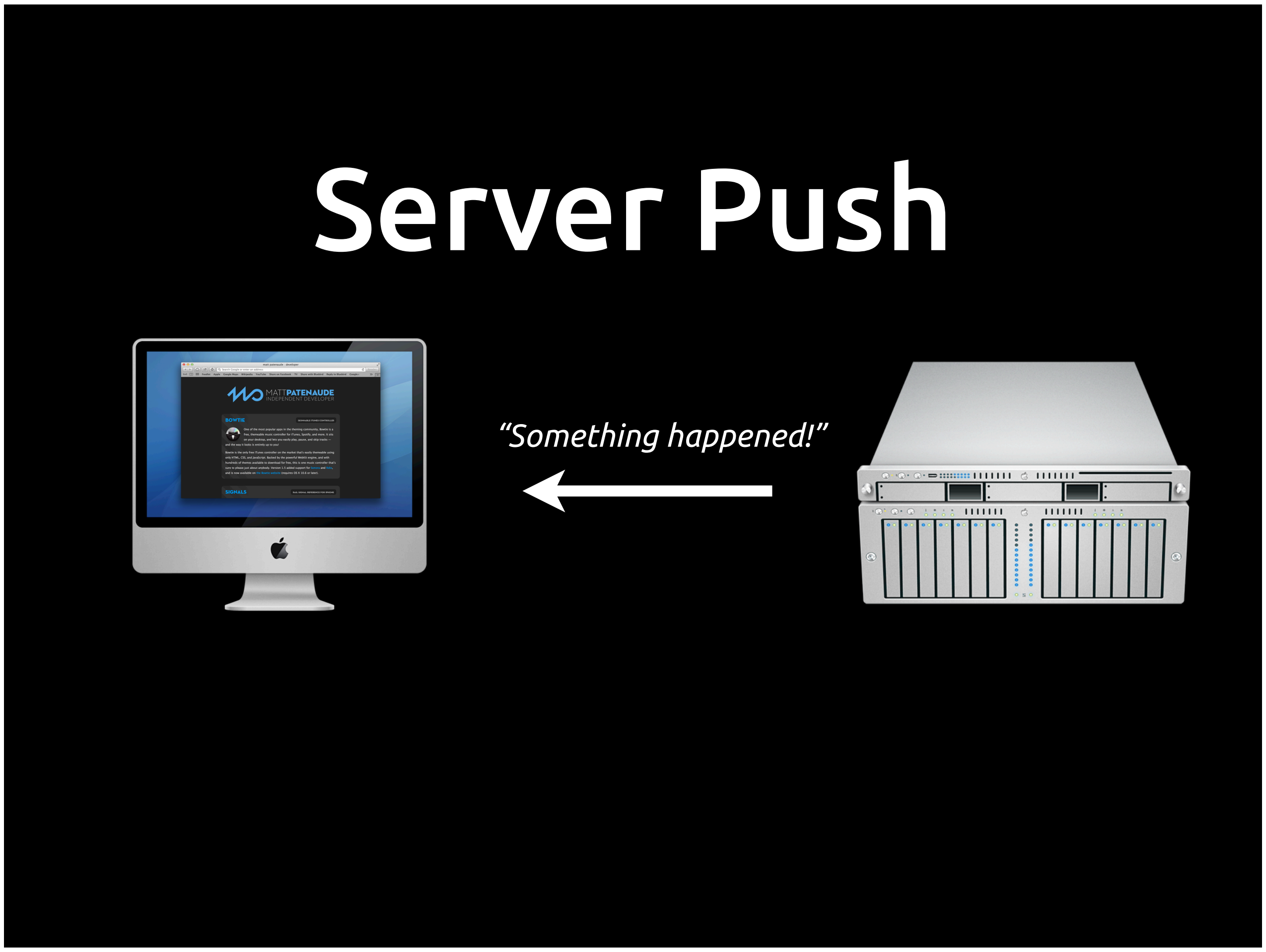
WebSockets for Mere Mortals

A Lecture by
Matt Patenaude

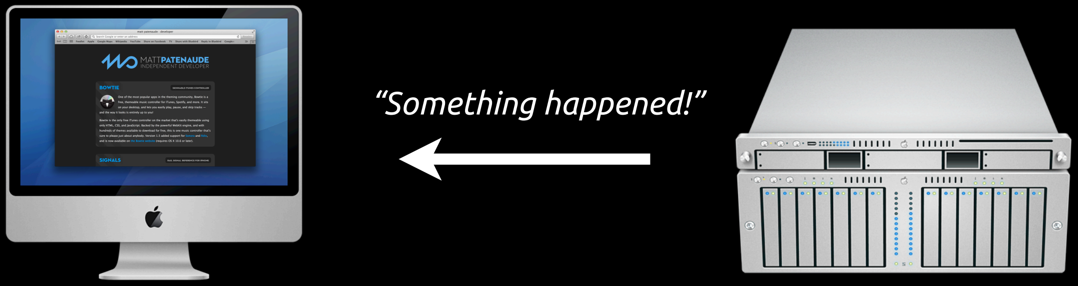
@mattpat



Currently, it’s easy for a website to inform the server that some event happened — we use Ajax for that. But what happens if the server wants to inform the client that something happened?

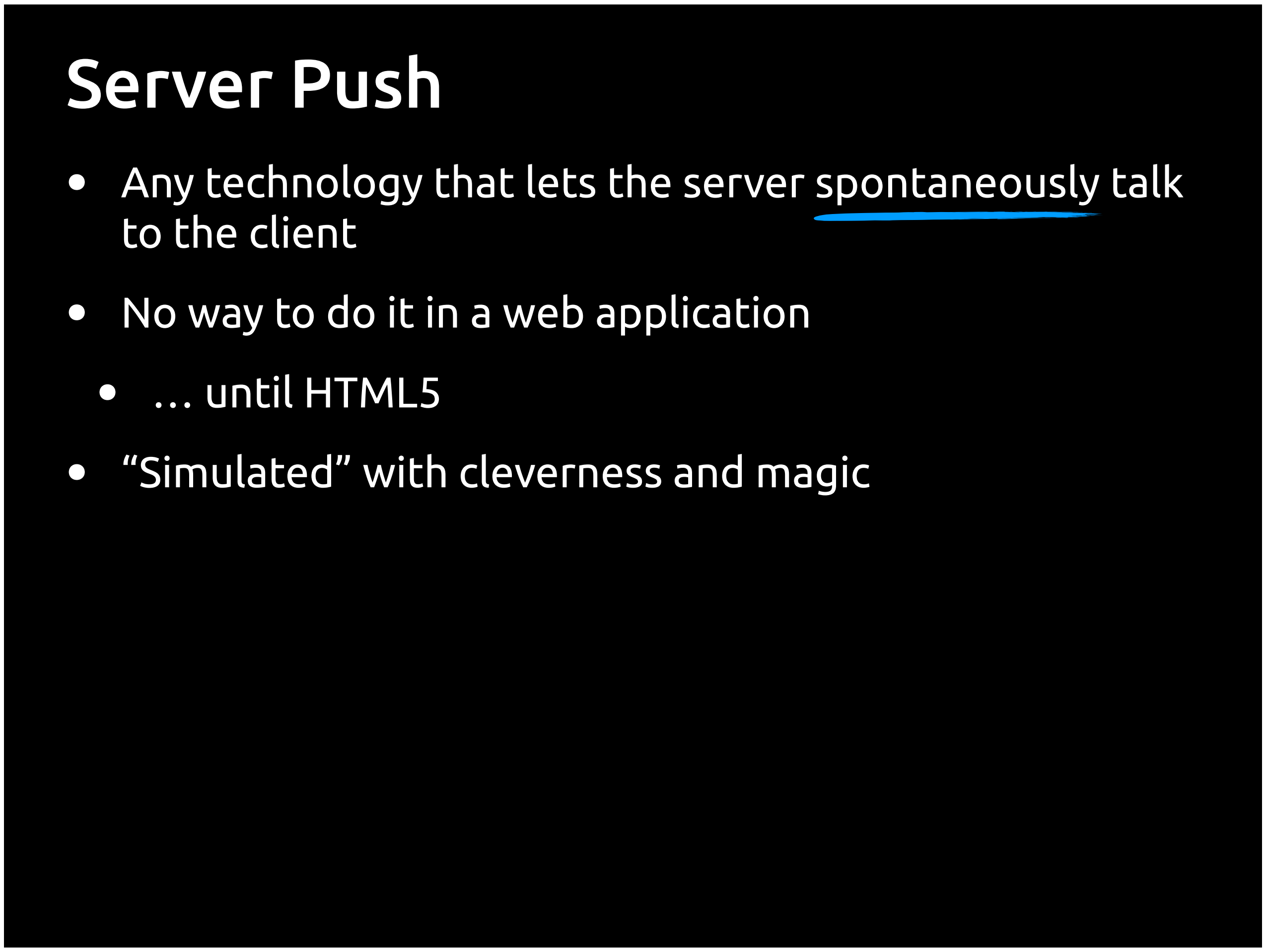


Server Push



The technologies that allow the server to send events directly to the client are collectively known as "Server Push" techniques.

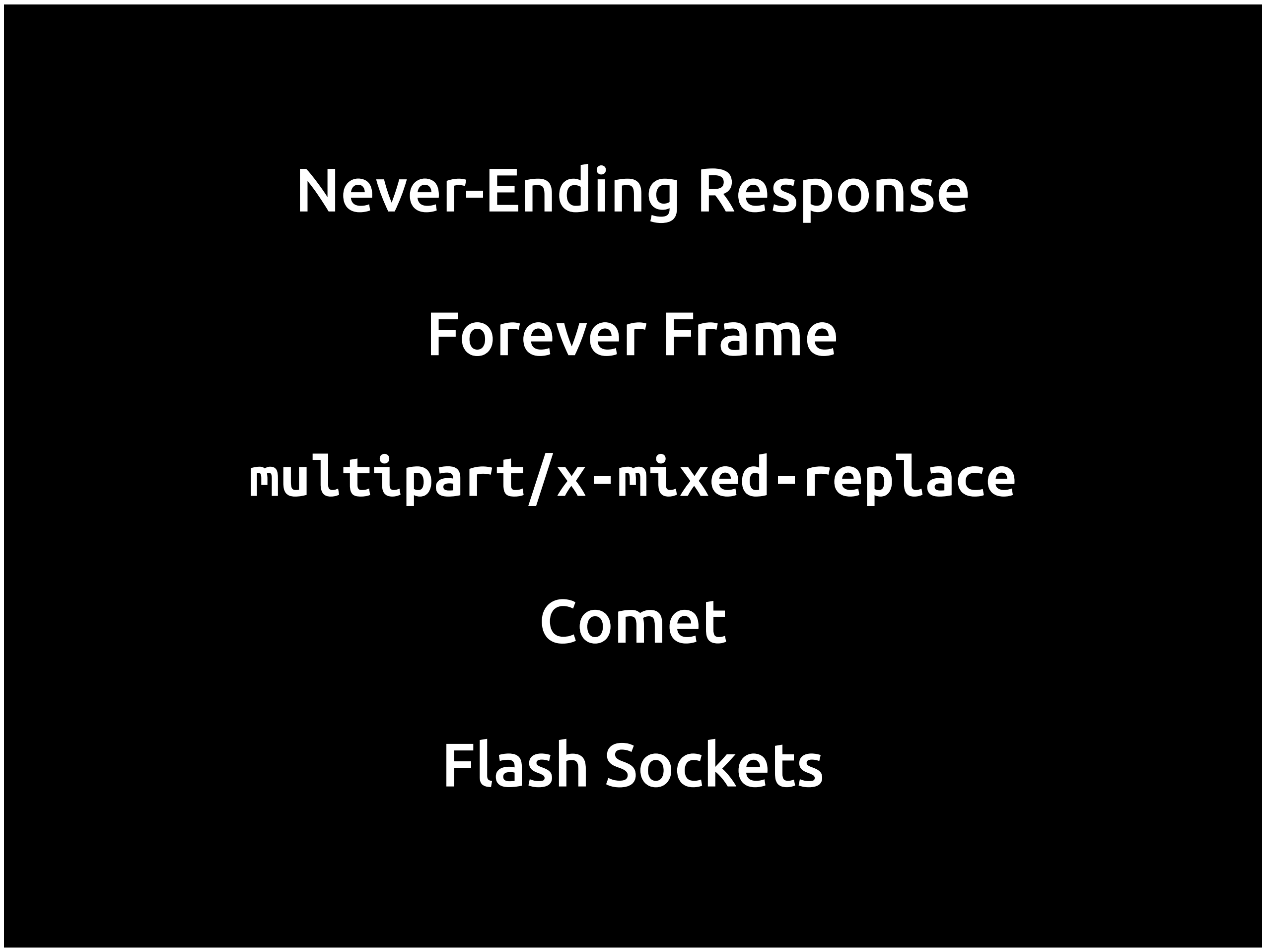
It's worth mentioning that, in the real presentation, the server moved across the screen and pushed the client off the edge. I can assure you, it was adorable.



Server Push

- Any technology that lets the server spontaneously talk to the client
- No way to do it in a web application
    - … until HTML5
- “Simulated” with cleverness and magic

The key point here is “spontaneously.” It’s trivial for the client to open an Ajax request to the server, and for the server to respond with something interesting. What we want here is the ability for the server to reach out and tap us on the shoulder to tell us something happened.



Never-Ending Response

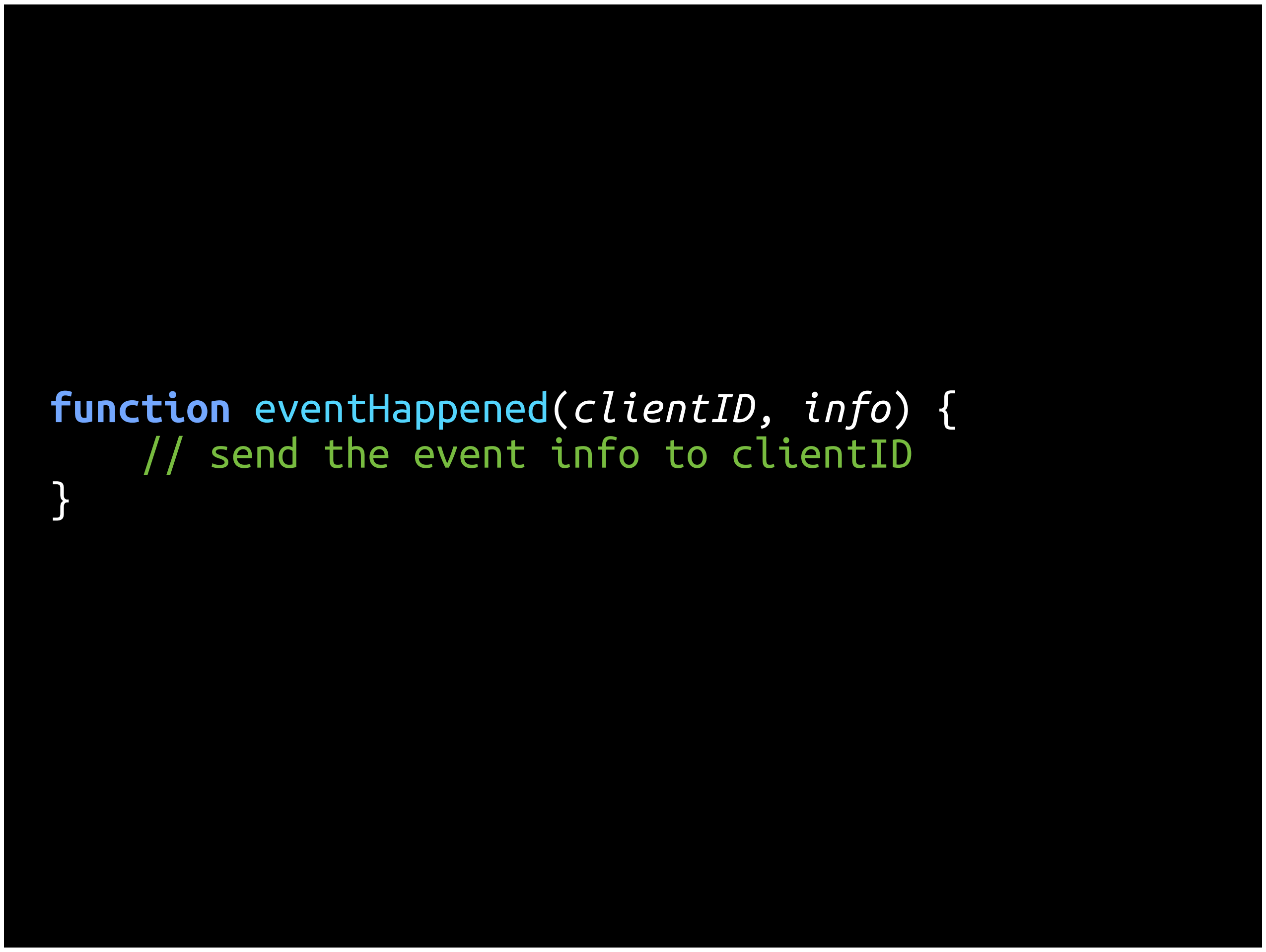
Forever Frame

`multipart/x-mixed-replace`

Comet

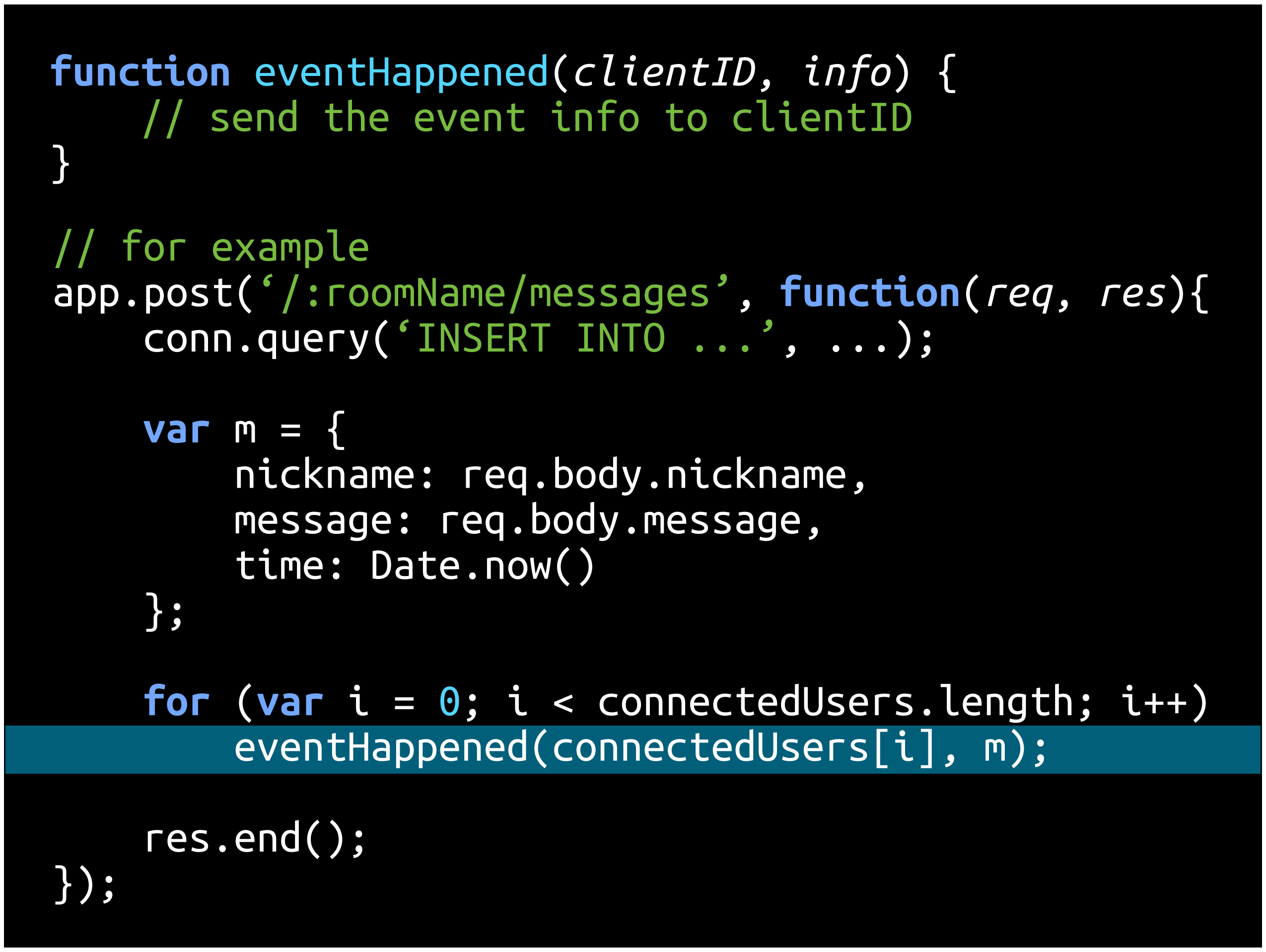
Flash Sockets

There are five basic techniques that are used to simulate server push on the web. For simplicity, in our presentation, we only focus on Forever Frame and Comet. It’s worth noting that this is mostly just a history lesson: you won’t actually be using these techniques, but it’s valuable to know what they are and how they function.



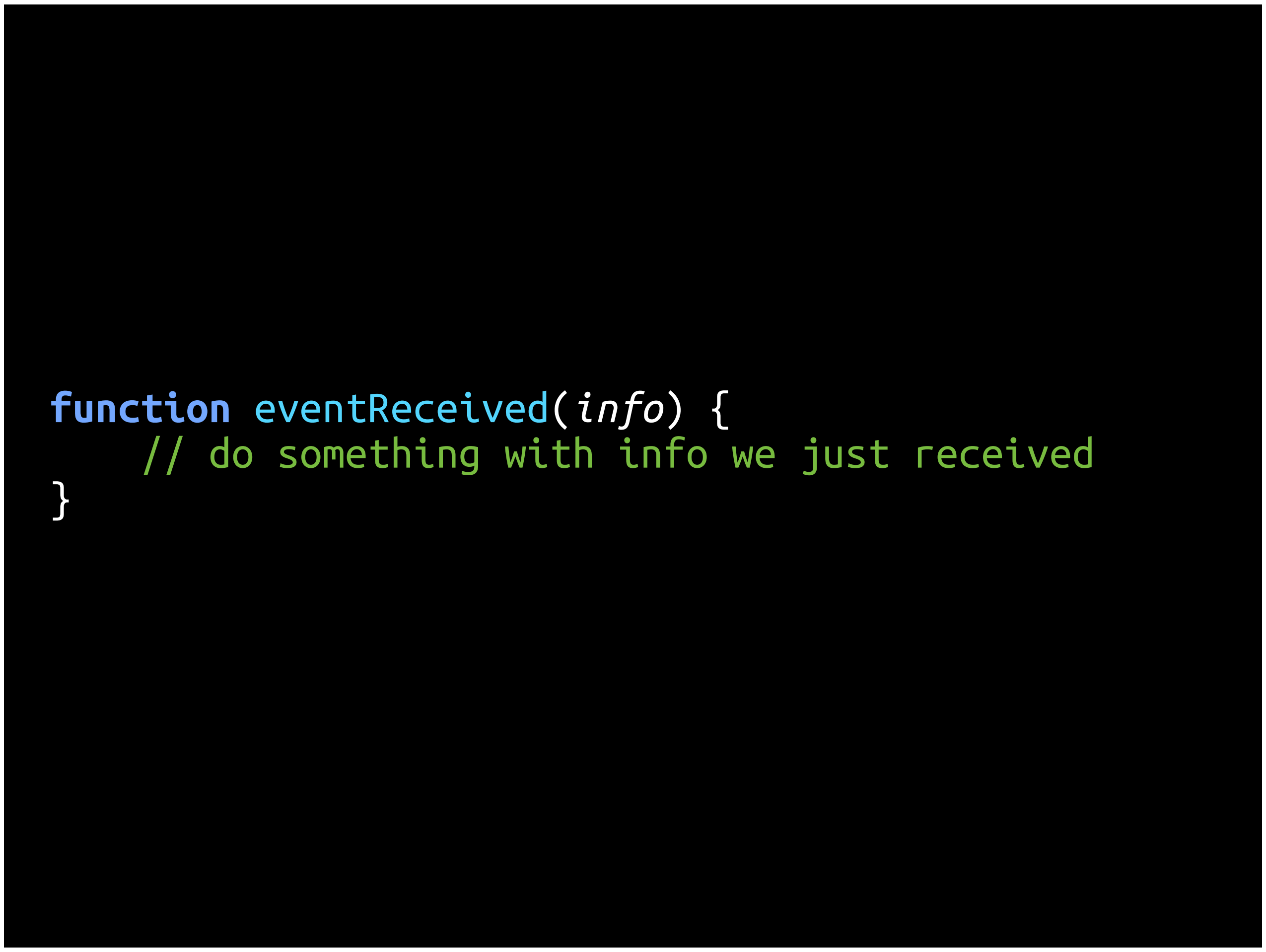
```
function eventHappened(clientID, info) {
    // send the event info to clientID
}
```

For the purposes of explaining how Forever Frame and Comet work, we'll imagine a pair of functions — one on the client, one on the server. The server version is called eventHappened, and the server will call this function any time something interesting happens (e.g., something it wants to notify the client named "clientID" about).



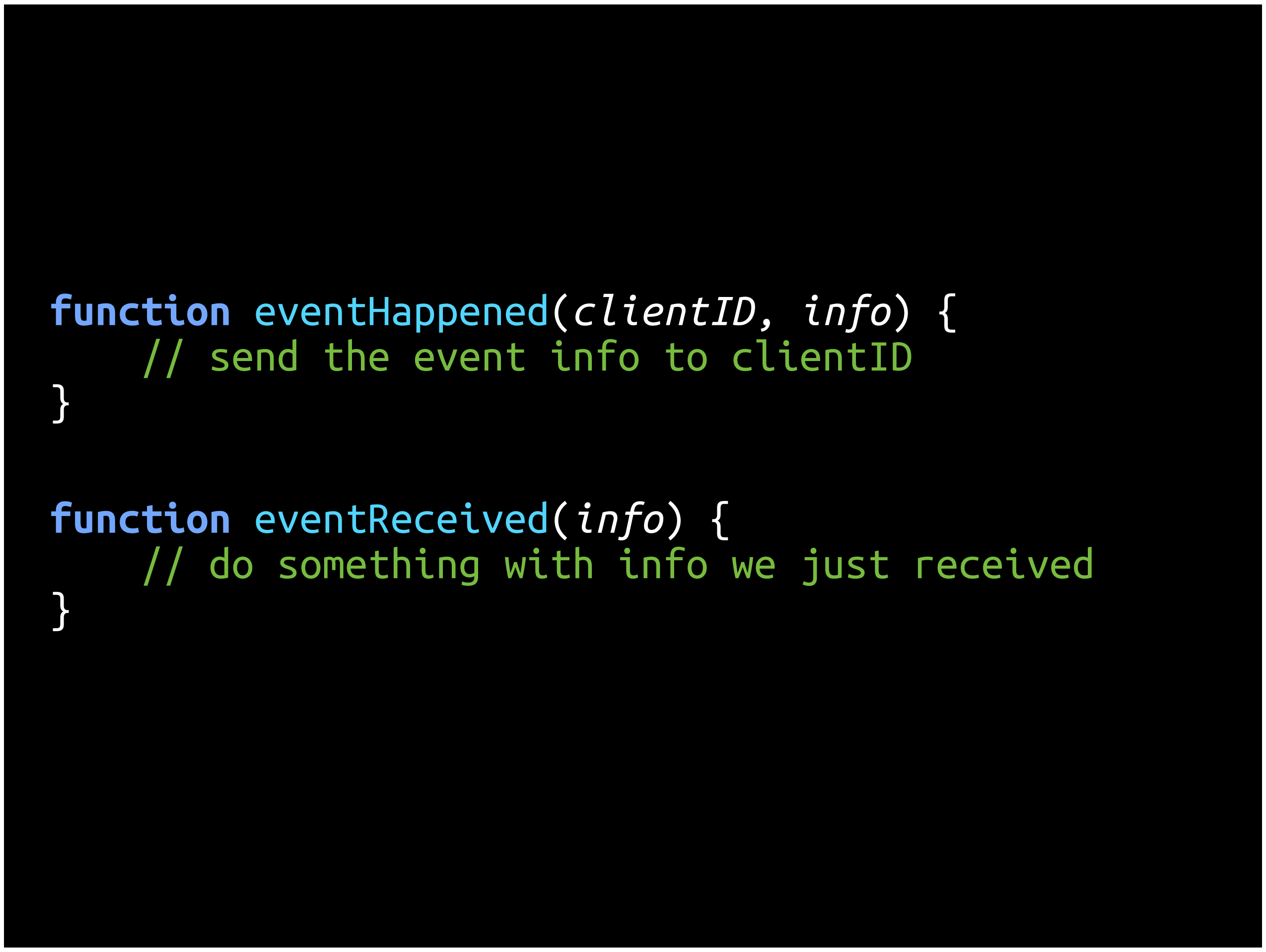
```
function eventHappened(clientID, info) {
    // send the event info to clientID
}

// for example
app.post(‘/:roomName/messages’, function(req, res){
    conn.query(‘INSERT INTO ...’, ...);

    var m = {
        nickname: req.body.nickname,
        message: req.body.message,
        time: Date.now()
    };

    for (var i = 0; i < connectedUsers.length; i++)
        eventHappened(connectedUsers[i], m);

    res.end();
});
```

For example, consider this implementation of the message post handler from chatroom. After we post the message to the database, we iterate through a (fictitious) list of connected users, and call eventHappened with each of their IDs. The goal is to implement eventHappened such that it gets a (nearly) immediate notification to the client in question.



```
function eventReceived(info) {
    // do something with info we just received
}
```

The client counterpart to eventHappened is eventReceived.



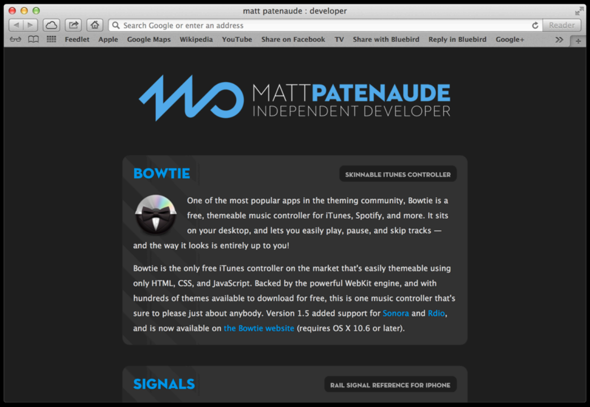
```
function eventHappened(clientID, info) {
    // send the event info to clientID
}

function eventReceived(info) {
    // do something with info we just received
}
```

The basic idea is that, whenever I call eventHappened on the server for a given client ID, the web browser window with that client ID will have eventReceived called with the same info parameter I passed into eventHappened. In other words, there's almost a "pipe" between eventHappened and eventReceived. This is the basis for our server push construction.

Forever Frame

- Browsers interpret HTML sequentially
- `script` tags executed immediately after parsing

```
<script>
  doSomething();
</script>
```


matt patenaude : developer

BOWTIE

SKINNABLE ITUNES CONTROLLER

One of the most popular apps in the theming community, Bowtie is a free, themeable music controller for iTunes, Spotify, and more. It sits on your desktop, and lets you easily play, pause, and skip tracks — and the way it looks is entirely up to you!

Bowtie is the only free iTunes controller on the market that's easily themeable using only HTML, CSS, and JavaScript. Backed by the powerful WebKit engine, and with hundreds of themes available to download for free, this is one music controller that's sure to please just about anybody. Version 1.5 added support for Sonora and Rdio, and is now available on the Bowtie website (requires OS X 10.6 or later).

SIGNALS

RAIL SIGNAL REFERENCE FOR IPHONE

```
<iframe src="/forever-frame"></iframe>
iframe { display: none; }
```

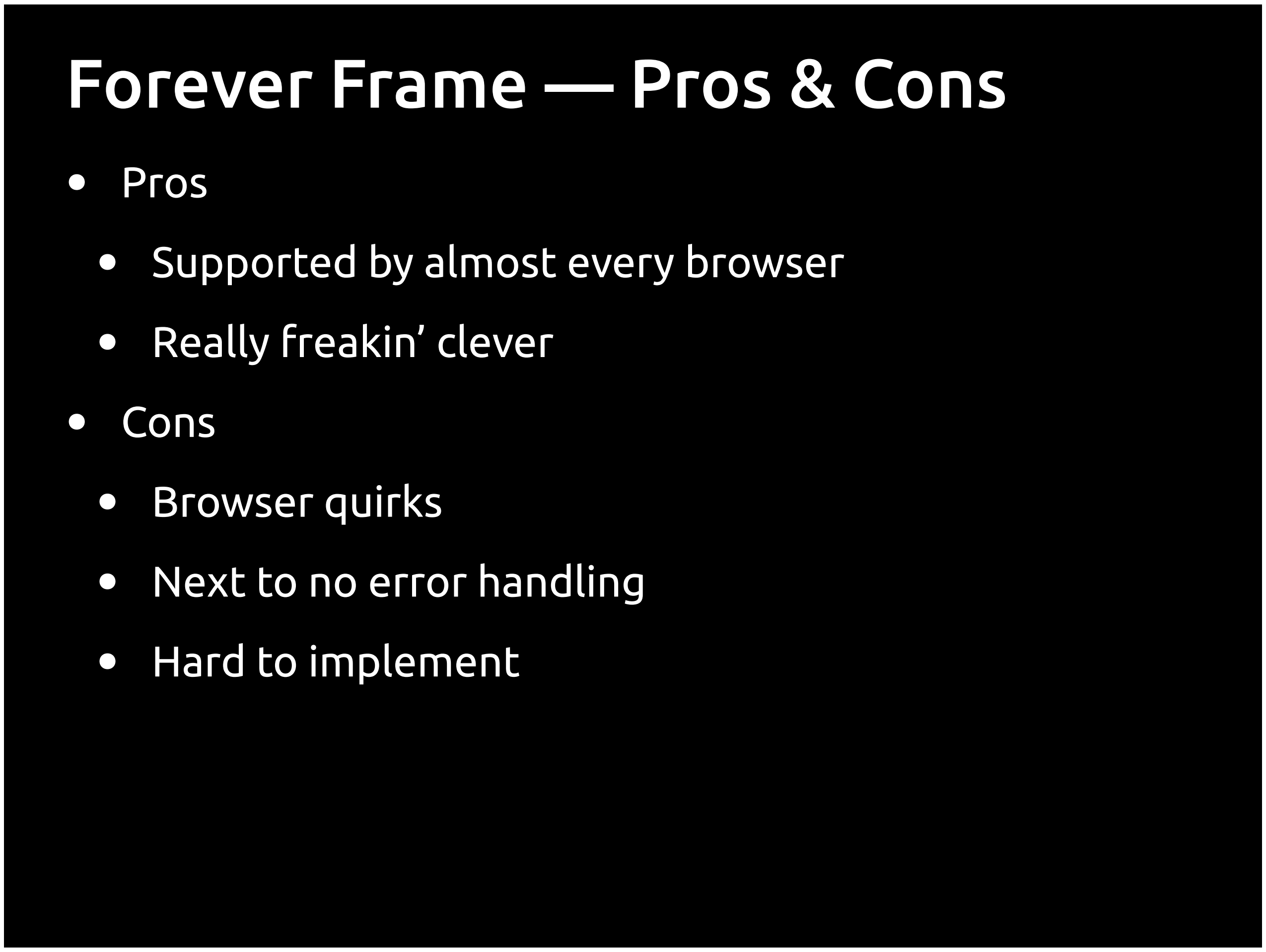
Forever Frame is one way of making it work, and it exploits two characteristics of web browsers: browser interpret HTML as it arrives, and scripts get executed immediately after they're loaded.

The way we make Forever Frame happen is by breaking a cardinal rule of project 3: we never call res.end() (or .send, or .json, or .render, or any of those). What that results in is a page that loads forever. Whenever the server wants to inform the client of something, it sends a script tag that calls some function on the client. Since the browser processes it immediately (rather than waiting for the page to finish loading), it gives us the desired behavior.

Because, though, we don't want the page to appear as though it's never finished loading, we create a separate URL to handle the event stream, stick it in an iframe, and then hide the iframe using CSS.

```
var openResponses = {};

function eventHappened(clientID, info) {
    // send the event info to clientID
    var res = openResponses[clientID];
    res.write('<script>');
    res.write('window.parent.eventReceived(');
    res.write('JSON.parse(\'');
    res.write(JSON.stringify(info));
    res.write('\'));');
    res.write('</script>');
}

app.get('/events/:clientID', function(req, res){
    openResponses[req.params.clientID] = res;
    res.write('<html><head></head><body>');
});
```

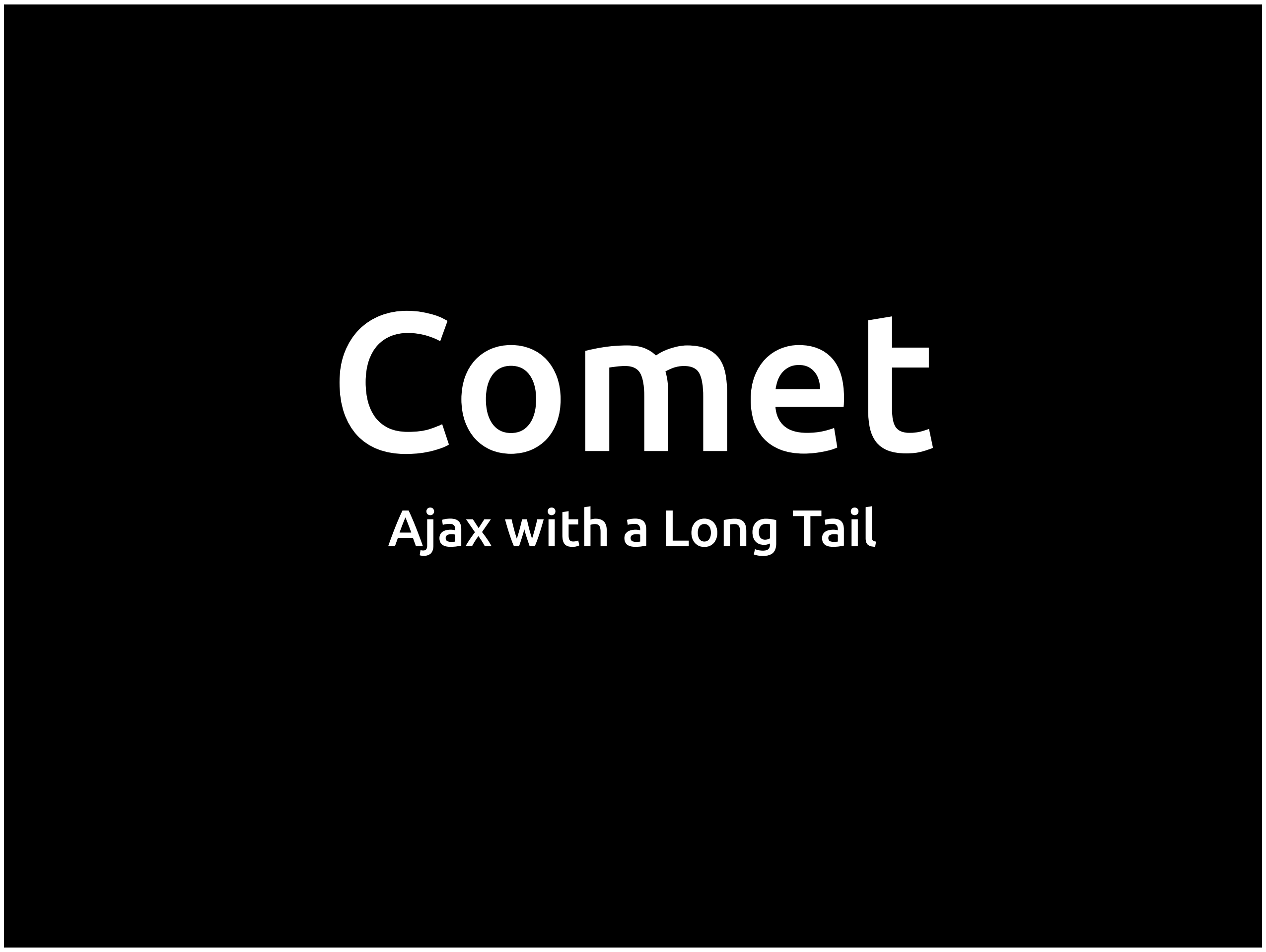
This is a sample server implementation of Forever Frame. Essentially, we expose the /events/clientID URL, and send back only the start of an HTML page (using res.write, not res.send or .end). Then, whenever eventHappened gets called, we look up the (still open) response in a table, and write out a script tag. The script tag contains the text "window.parent.eventReceived(JSON.parse(…));", where the … is replace by the JSON-encoded version of our info parameter. This has the effect of breaking out of the iframe into the parent window (window.parent), and calling its eventReceived function. Thus, whenever eventHappened gets called on the server, eventReceived gets called on the appropriate client.



Forever Frame — Pros & Cons

- Pros
  - Supported by almost every browser
  - Really freakin' clever
- Cons
  - Browser quirks
  - Next to no error handling
  - Hard to implement

This is a great strategy because it works pretty much everywhere, but it's incredibly difficult to recover from errors. There are also a number of "gotchas" to worry about (for example: Safari won't process scripts until its received a thousand bytes of data, so it needs to be padded with blank spaces; IE won't process scripts until a visual element is encountered, so you need to add some empty paragraphs or something similar), which makes reliable implementations more difficult.



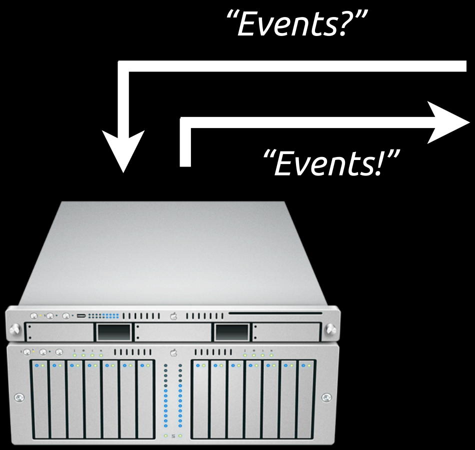
Comet

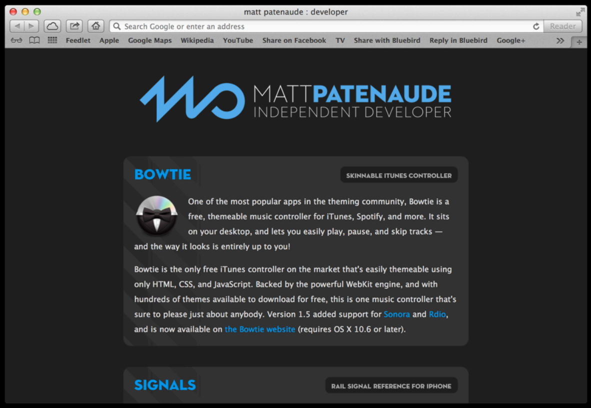
Ajax with a Long Tail

Comet provides an answer to some of the issues. This slide had a cute comet animation.

Comet

- Uses ordinary Ajax requests
- Server doesn't respond until it has something interesting




matt patenaude : developer

Search Google or enter an address

Reader

MATTPATENAUDE
INDEPENDENT DEVELOPER

BOWTIE

SKINNABLE ITUNES CONTROLLER

One of the most popular apps in the theming community, Bowtie is a free, themeable music controller for iTunes, Spotify, and more. It sits on your desktop, and lets you easily play, pause, and skip tracks — and the way it looks is entirely up to you!

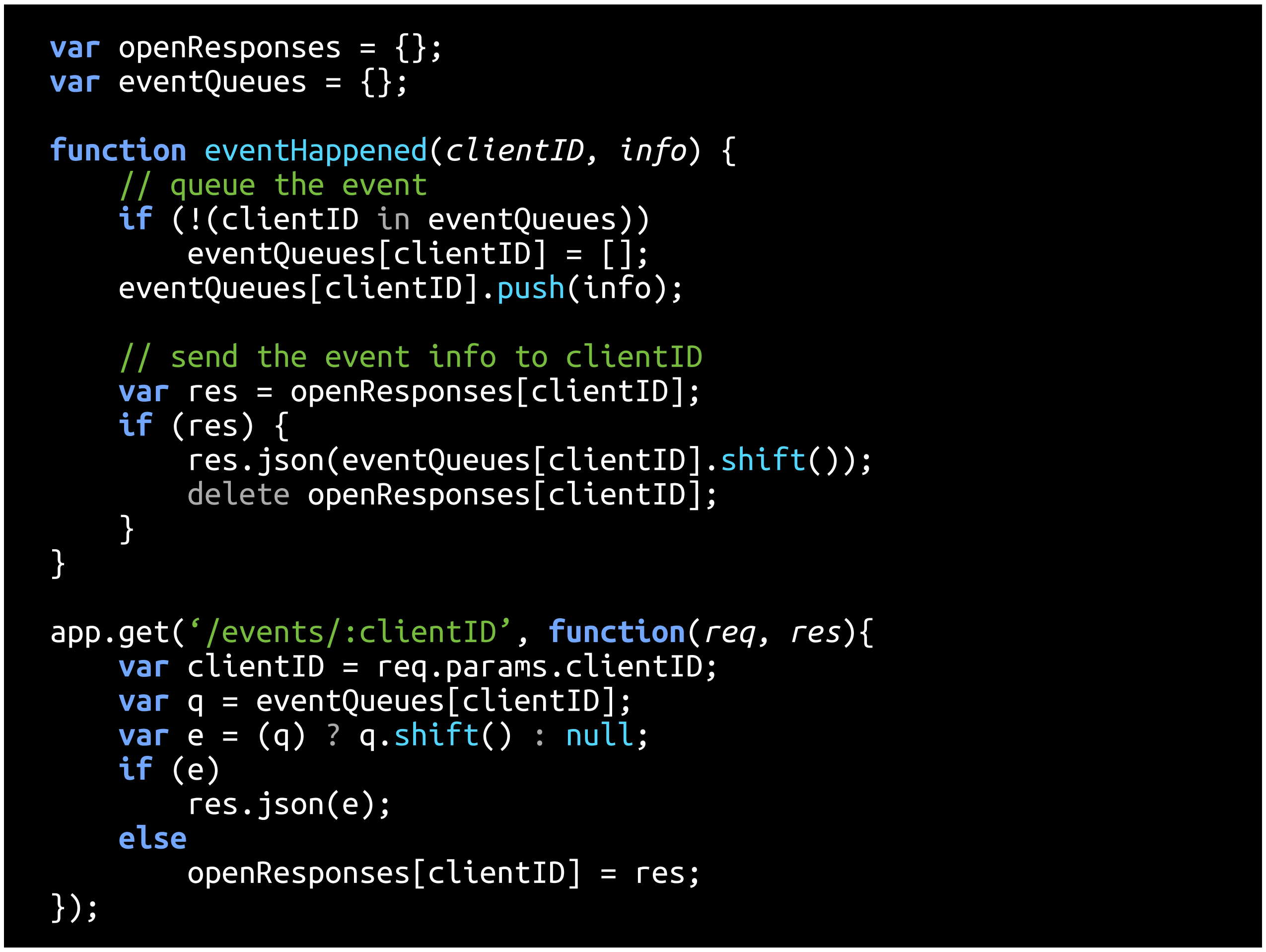
Bowtie is the only free iTunes controller on the market that's easily themeable using only HTML, CSS, and JavaScript. Backed by the powerful WebKit engine, and with hundreds of themes available to download for free, this is one music controller that's sure to please just about anybody. Version 1.5 added support for Sonora and Rdio, and is now available on the Bowtie website (requires OS X 10.6 or later).

SIGNALS

RAIL SIGNAL REFERENCE FOR IPHONE

"Nothing… nothing… nothing… something!"

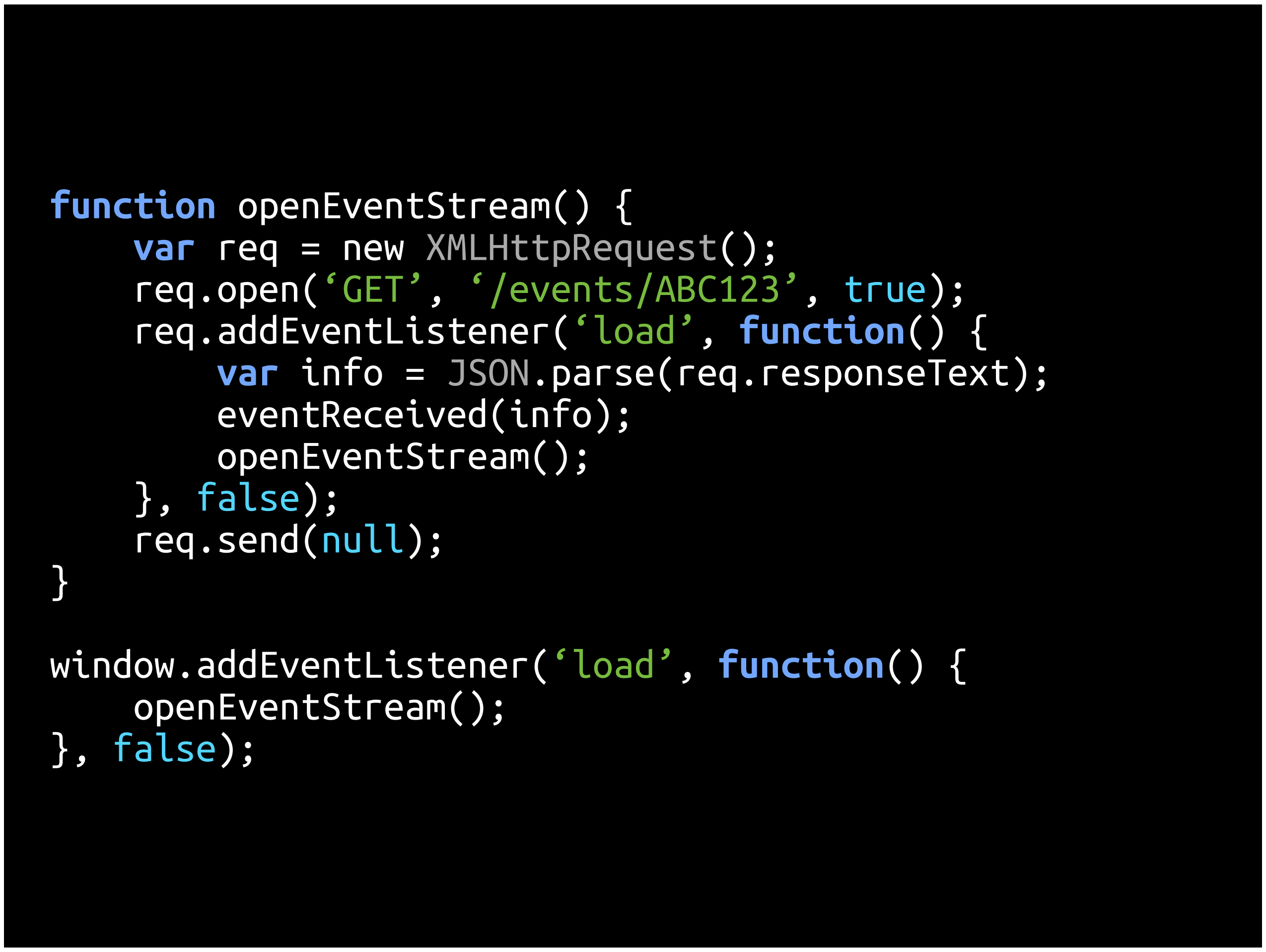
Comet is a clever modification of standard Ajax patterns. The browser makes an Ajax request to the event stream, and if there's something waiting for it, the server sends it back immediately. Otherwise, it just keeps the connection waiting until an event happens.



```
var openResponses = {};
var eventQueues = {};

function eventHappened(clientID, info) {
    // queue the event
    if (!(clientID in eventQueues))
        eventQueues[clientID] = [];
    eventQueues[clientID].push(info);

    // send the event info to clientID
    var res = openResponses[clientID];
    if (res) {
        res.json(eventQueues[clientID].shift());
        delete openResponses[clientID];
    }
}

app.get('/events/:clientID', function(req, res){
    var clientID = req.params.clientID;
    var q = eventQueues[clientID];
    var e = (q) ? q.shift() : null;
    if (e)
        res.json(e);
    else
        openResponses[clientID] = res;
});
```

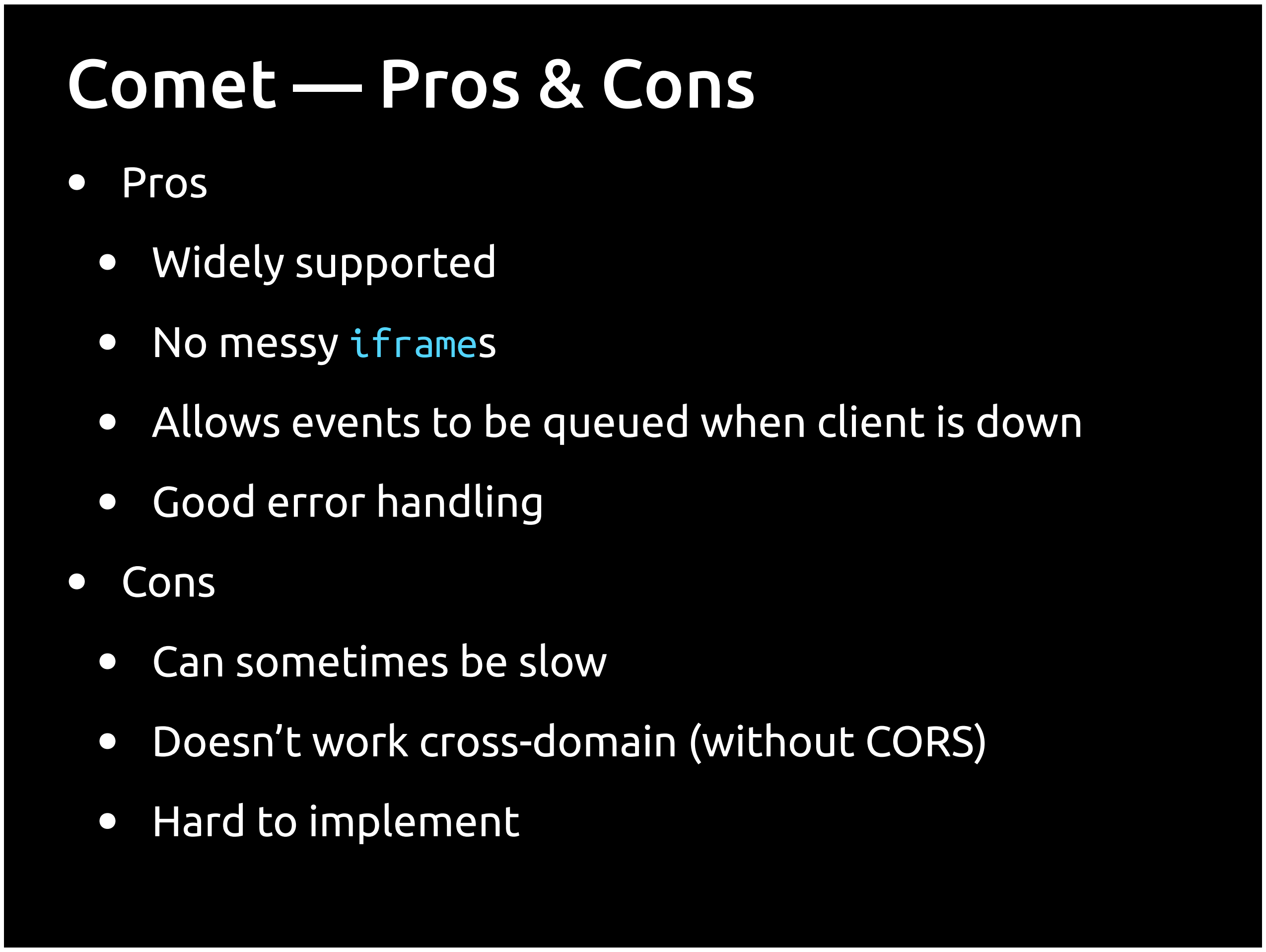
The implementation is a bit more complex, because it now involves an event queue, but it's still somewhat straightforward. When a get request comes in for the event stream, we first check the client's event queue to see if anything's waiting. If it is, we send it back JSON-encoded immediately. Otherwise, we store the response in our openResponses table.

When eventHappened is called, we look up the client's event queue (or create one if necessary), and push "info" onto the queue. We then check to see if the client in question is currently connected, and if so, respond with the first event on the queue.



```
function openEventStream() {
    var req = new XMLHttpRequest();
    req.open(‘GET’, ‘/events/ABC123’, true);
    req.addEventListener(‘load’, function() {
        var info = JSON.parse(req.responseText);
        eventReceived(info);
        openEventStream();
    }, false);
    req.send(null);
}

window.addEventListener(‘load’, function() {
    openEventStream();
}, false);
```

The client implementation looks something like this. We make an Ajax request to the event stream (in this case, we’re pretending our client ID is ‘ABC123’), and upon receiving a response, we call eventReceived, and then immediately re-request the event stream.



Comet — Pros & Cons

- Pros
  - Widely supported
  - No messy `iframes`
  - Allows events to be queued when client is down
  - Good error handling
- Cons
  - Can sometimes be slow
  - Doesn't work cross-domain (without CORS)
  - Hard to implement

This is, in many ways, a better solution than Forever Frame because we can handle errors much more easily, and it introduces the concept of an event queue (which means we won't miss any events if the client goes down for a few seconds between events). However, there is the added overhead of closing and reopening a connection every time an event happens, and it can't be used across different domains unless we configure it with CORS (like we did with BieberFeed).

WebSocket

Client-Server Messaging from the Future

WebSocket provides the answer to all of our problems.

WebSocket

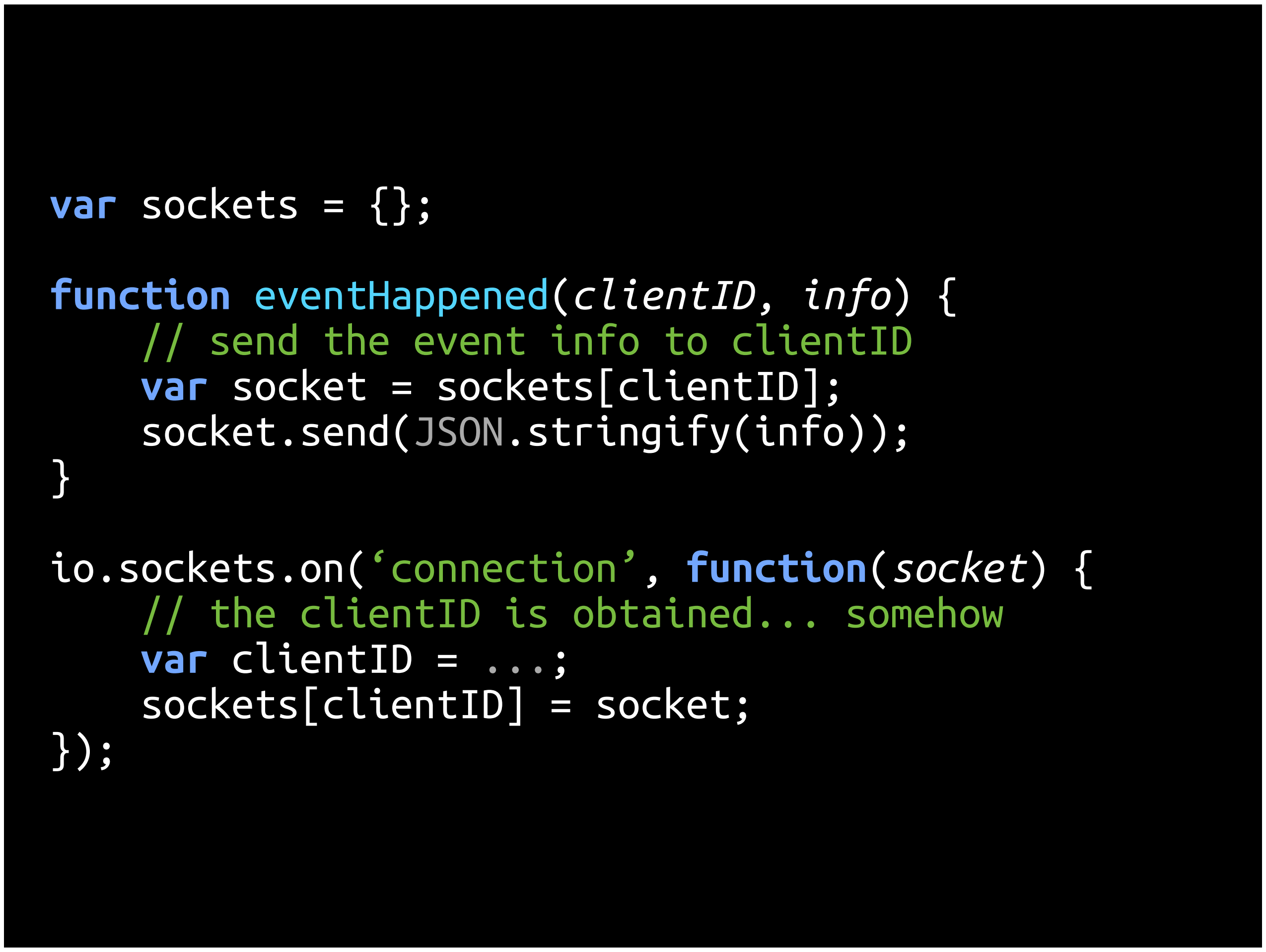
- Two-way messaging with a simple API
- Works across multiple domains
- Incredibly fast

“How’s it hangin’?”

“New message!”

Bowtie is the only free iTunes controller on the market that's easily themeable using only HTML, CSS, and JavaScript. Backed by the powerful WebKit engine, and with hundreds of themes available to download for free, this is one music controller that's sure to please just about anybody. Version 1.5 added support for Sonora and Rdio, and is now available on the Bowtie website (requires OS X 10.6 or later).

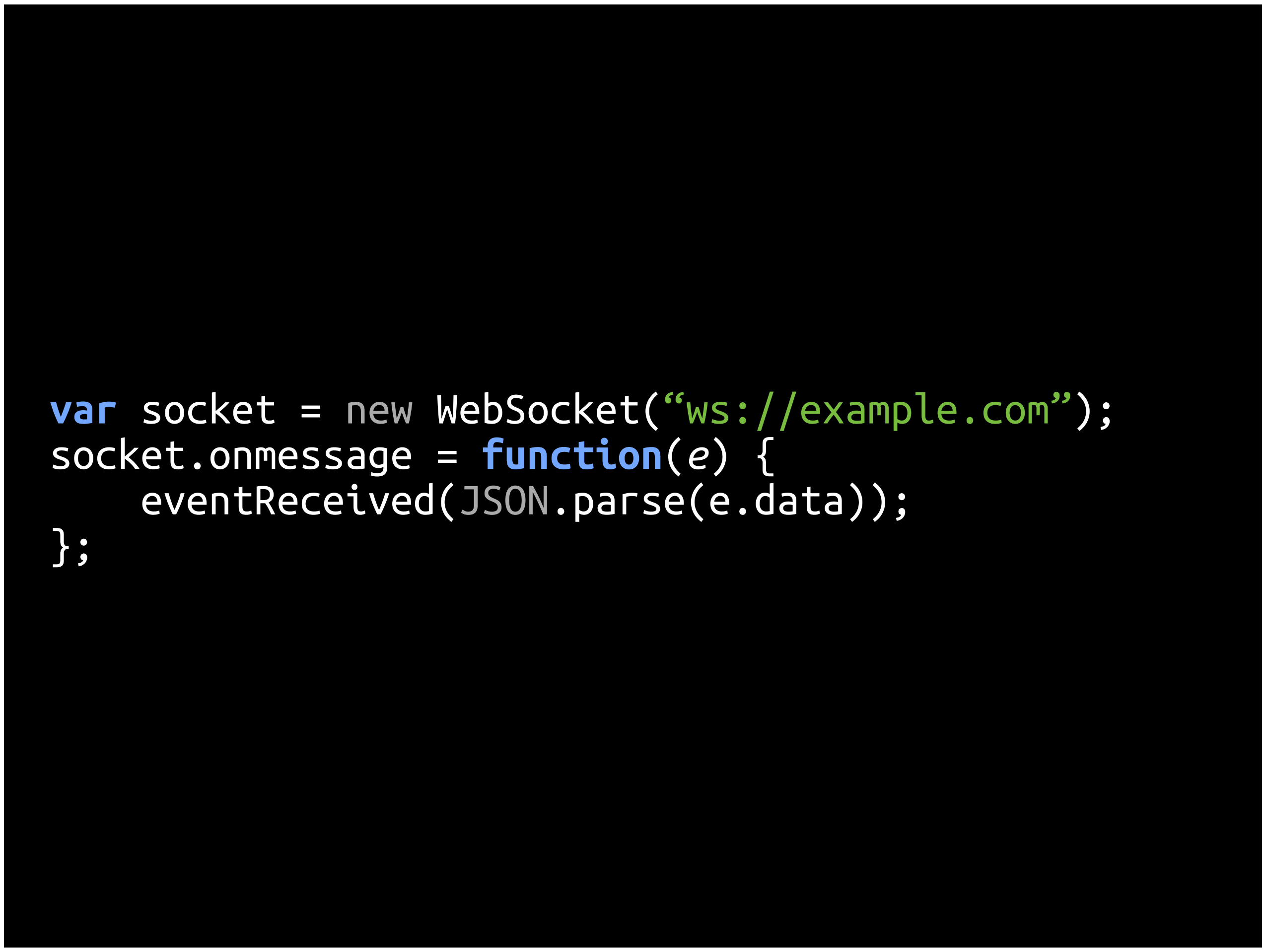
Really, it’s quite awesome. WebSocket is a true push technology — we’re not faking it anymore. It gives us a way to send messages in both directions between the client and server. More cute animations happened here.



```
var sockets = {};

function eventHappened(clientID, info) {
    // send the event info to clientID
    var socket = sockets[clientID];
    socket.send(JSON.stringify(info));
}

io.sockets.on('connection', function(socket) {
    // the clientID is obtained... somehow
    var clientID = ...;
    sockets[clientID] = socket;
});
```

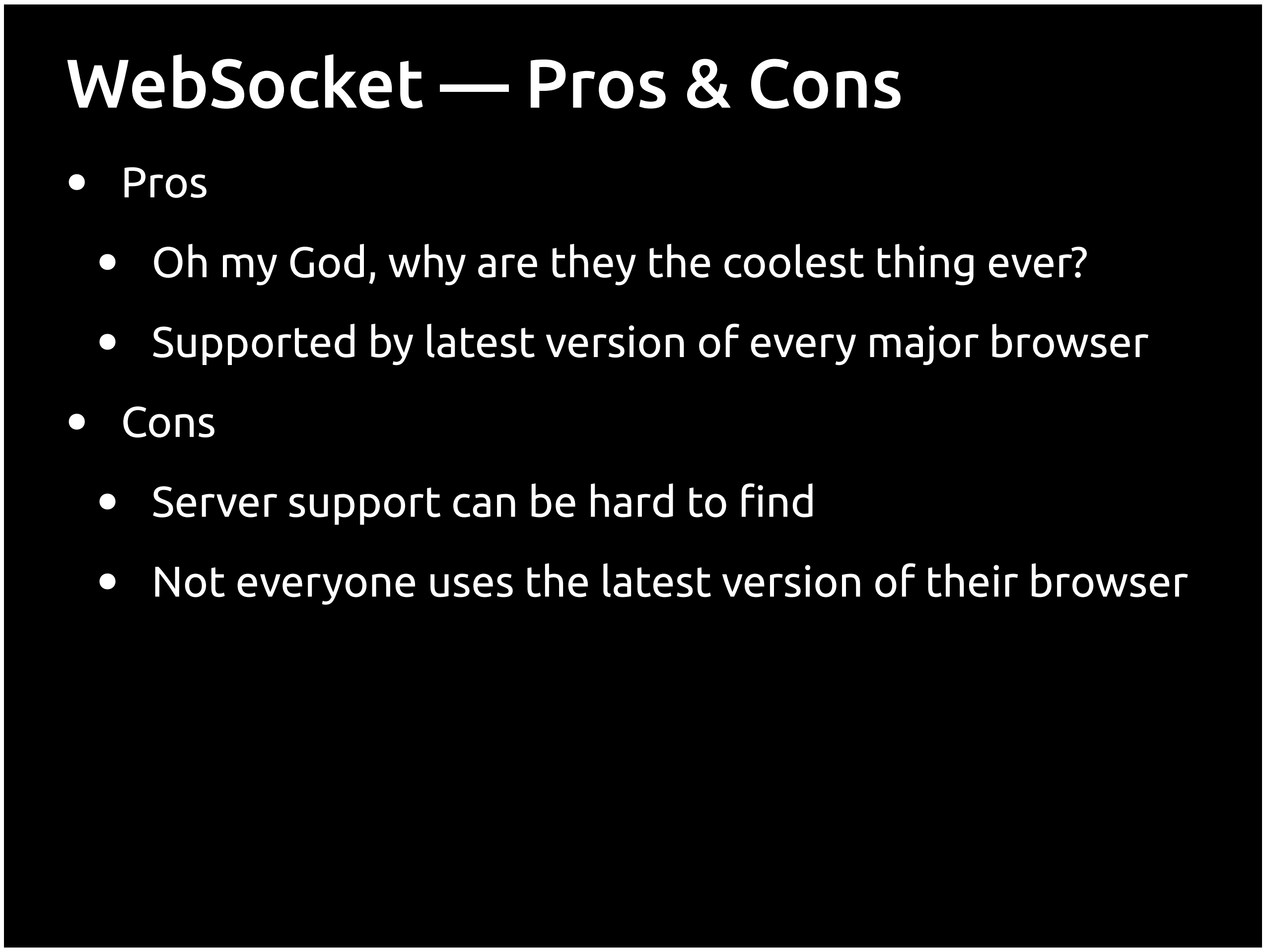
Our server implementation (at least on Node) is vastly simpler. Here, we're using socket.io, but only for its WebSocket functionality (it does much more, which we'll talk about later). Instead of using an app.get(…), we use io.sockets.on('connection', …), which gets called whenever a new WebSocket connection is opened.

Basically, all we do is store our socket in a table, and whenever an event happens, we call the socket's send(…) method to send the JSON version of "info" back to the client.



```
var socket = new WebSocket("ws://example.com");
socket.onmessage = function(e) {
    eventReceived(JSON.parse(e.data));
};
```

The client code is even simpler. We create a new WebSocket object pointed at our server (note we use "ws://" instead of "http://"), and then we just associate a "message" event handler (we could also have done addEventListener('message', …, false), but I wanted to make the code look even simpler). The data from the server is included in the "data" property of the event object.



WebSocket — Pros & Cons

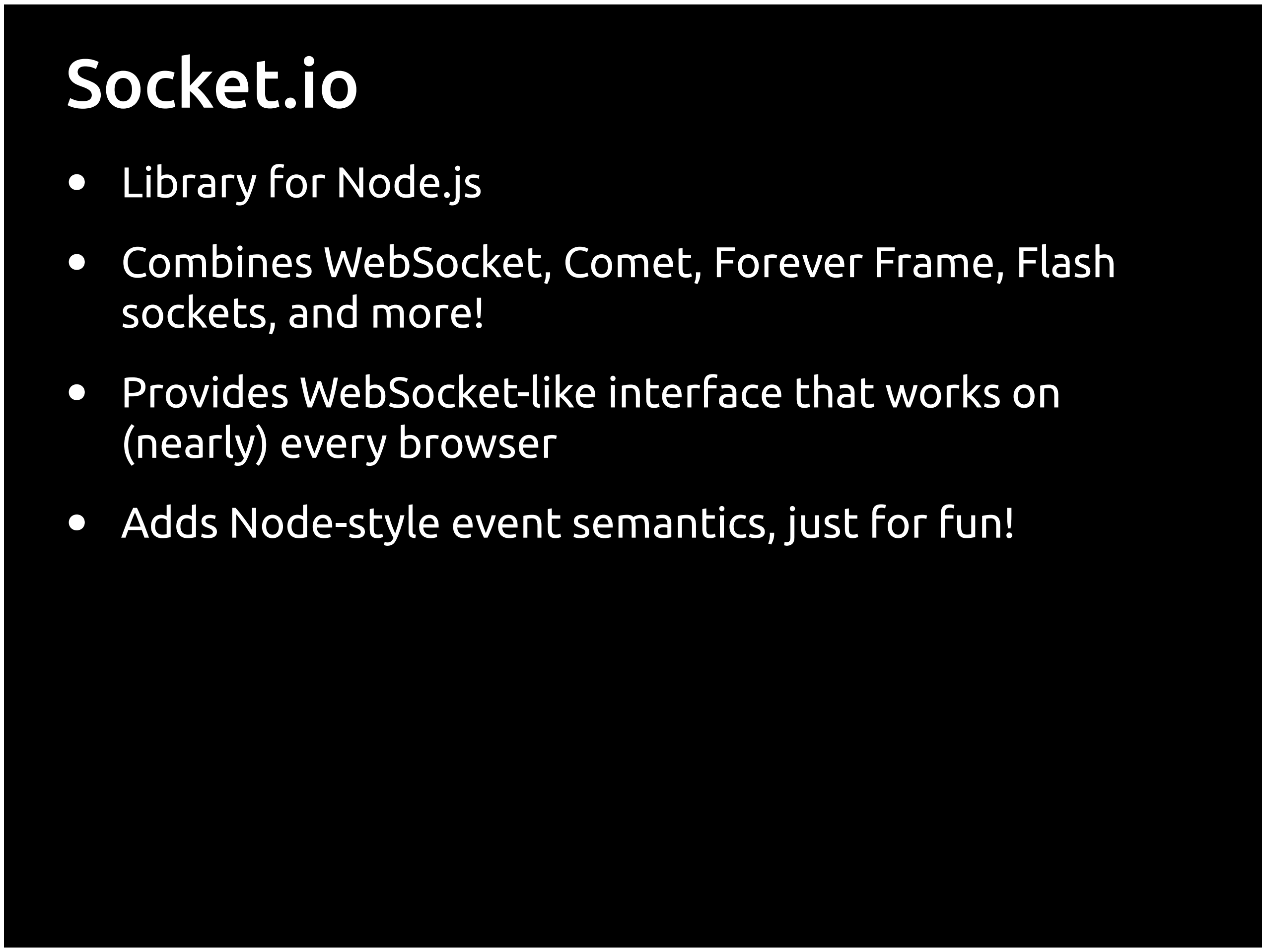
- Pros
  - Oh my God, why are they the coolest thing ever?
  - Supported by latest version of every major browser
- Cons
  - Server support can be hard to find
  - Not everyone uses the latest version of their browser

The only real problem that WebSocket has is that server support is limited, and not everyone uses a WebSocket–supporting browser. We need a way to work around that somehow, and the answer is…

Socket.io

Cross-Platform WebSockets + Rainbow Magic

The coolest open source library released on the Internet in the past 5 years. #boldstatement #don’tevencare

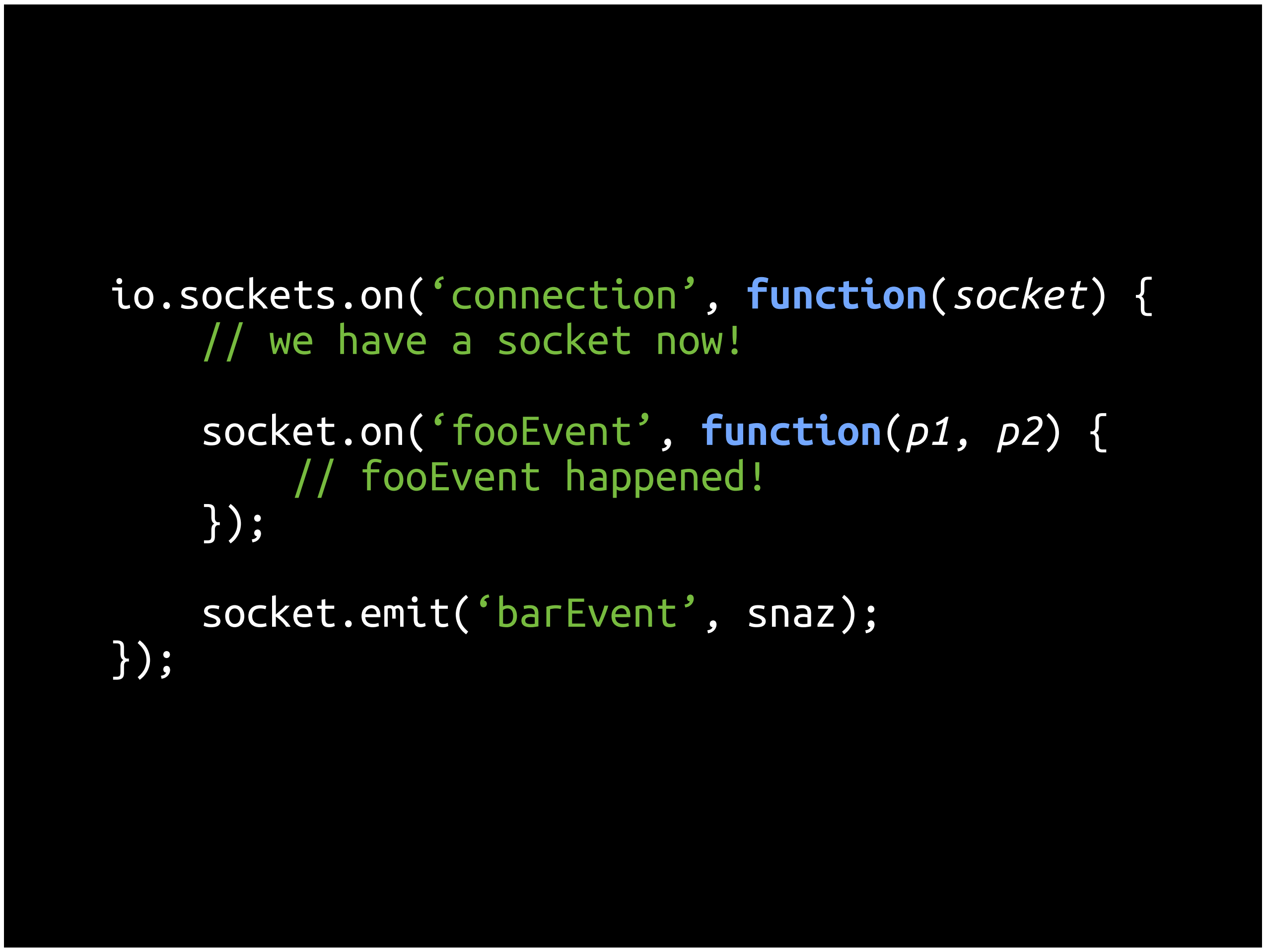


Socket.io

- Library for Node.js
- Combines WebSocket, Comet, Forever Frame, Flash sockets, and more!
- Provides WebSocket-like interface that works on (nearly) every browser
- Adds Node-style event semantics, just for fun!

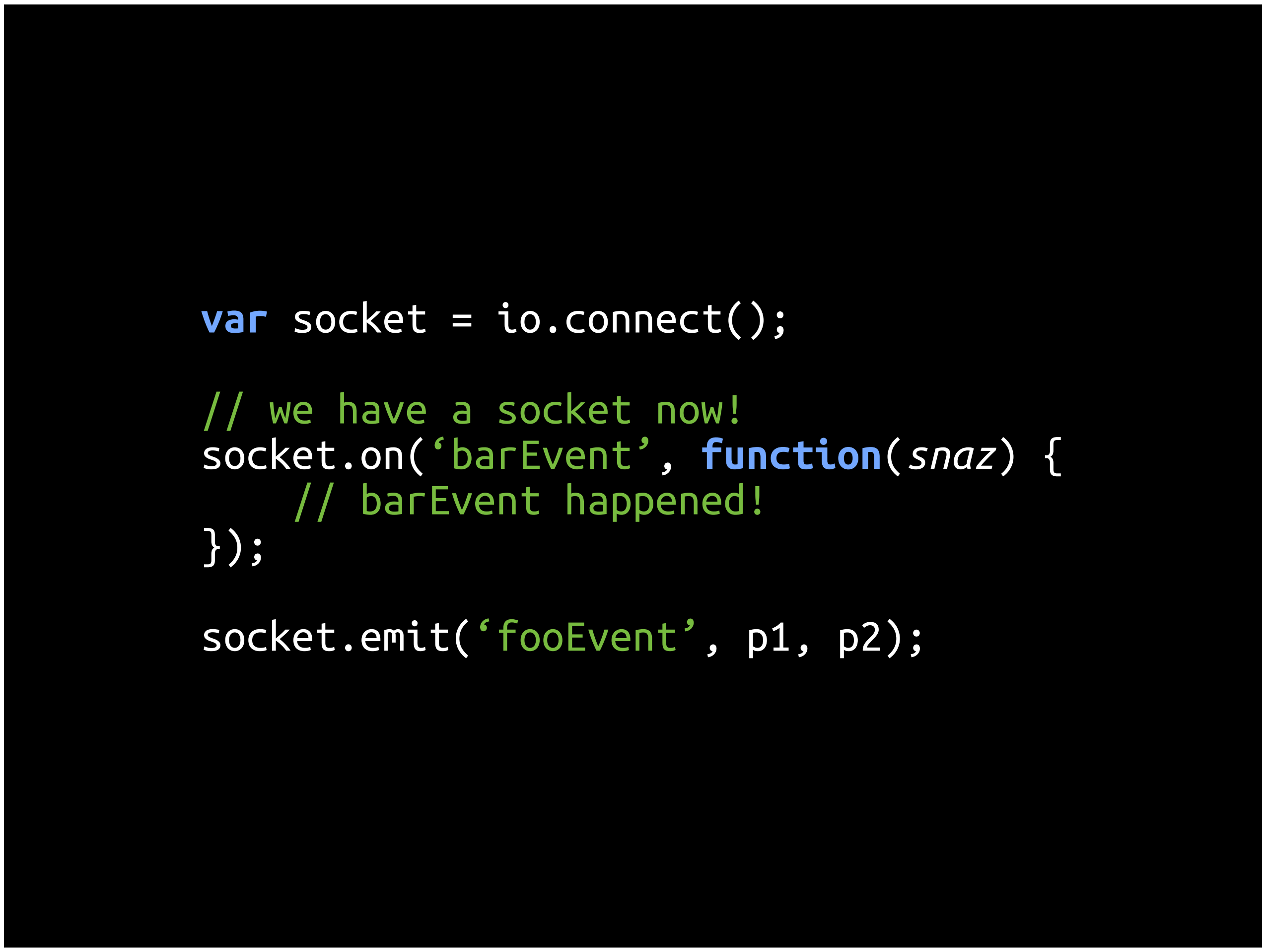
Socket.io combines a bunch of different “simulated” server push techniques, along with real honest-to-goodness WebSockets, and hides them all behind a simple WebSocket-like API. When socket.io loads on a given web page, it queries the browser to see what technique will work best. If the browser supports WebSocket, it uses it! If not, it might fall back to Comet, Forever Frame, or a number of other possibilities. This is all done transparently to us, the developers.

And just for good measure, it adds Node-style event emitter semantics (including the ability to have CALLBACKS OVER THE INTERWEBS WHOMG).



```
io.sockets.on('connection', function(socket) {
    // we have a socket now!

    socket.on('fooEvent', function(p1, p2) {
        // fooEvent happened!
    });

    socket.emit('barEvent', snaz);
});
```

A server implementation looks like this. We pick our own names for events — no one tells us what to do anymore, this is America! — and we associate event handlers using the canonical .on(…) method. Here, we do the setup when a new socket connects to the server, which is the best place to do it. We specify a function that should be called whenever the client emits a “fooEvent”, and then we emit a “barEvent” of our own to the client. Note that we just pass normal JavaScript parameters along with our event — no more need to call JSON.parse(…) and JSON.stringify(…), socket.io does it for us! Nifty.



```
var socket = io.connect();

// we have a socket now!
socket.on('barEvent', function(snaz) {
    // barEvent happened!
});

socket.emit('fooEvent', p1, p2);
```

And here's the client version. At the top of our script we call io.connect(), which opens our connection to the server. We use the exact same syntax as the server for setting up handlers and emitting events.

Demo

This was the bulk of the presentation: we took a working implementation of project 3, and turned it into a working implementation of project 4 over the course of about 45 minutes. While I will not be releasing the project 4 version of the code (that would be unfair), I will be publishing my working version of project 3 so that you can fix any bugs in your implementation if they exist.

Q & A

Saturday, April 20, 13

If you have any questions about the content contained in these slides, you can always ask the TAs during office hours, or else email us at cs132tas@cs.brown.edu. Thanks!