# Building Real-Time Web Applications

Aaron Mulder
CTO, Chariot Solutions

# What are Real-Time Web Applications?

- Something where you want to see the page update *immediately* when a change happens or new data comes from the server
  - Old: Stocks, Chat
  - New: Twitter, Facebook
  - Real: A monitoring console, a whiteboard, a game

# Why Are We Here?

- New "push" technology:
  - Zero-generation: timed page reloads?
  - First-generation: AJAX polling
  - Second-generation: AJAX long polling, Flash, etc.
  - Now: WebSockets plus alternatives: Cometd, Socket.io

# Why Are We Here? con't

- New JavaScript frameworks:
    - Backbone.js, Ember.js, etc.
- Dynamic charts (without Flash, SVG):
    - Smoothie, flot.js, etc.
- Background tasks:
    - Web Workers

# But Why Are We *Here*?

- The good news: lots of libraries, lots of examples
- The bad news: limited documentation, examples tend to be very tightly focused, frameworks can be used many ways
- Will demonstrate these tools based on a real-time monitoring app

# Backbone.js

# Introducing Backbone.js

- MVC for JavaScript
- Holds your data in models
- Treats the Web page as a view that:
  - Fires events that you might care about
  - Should be updated on model changes
- Can use Routers as controllers

# Backbone.js, con’t

- Many ways it can be used
  - With/without Routers/Controllers
  - Based on events or based on methods
  - Various ways to render a view
    - DOM manipulation, templates, etc.
- Which only exacerbates the example issue

# But For Our Purposes

- We need a place to store data
  - *A model (really, a collection of models...)*
- We want to update on model changes
  - *A view per model, with render() function*
- We need a way to gather data updates
  - *Polling works out of the box with fetch()*

# Initial Goal

## Realtime Monitoring Demo

| Machine | # Requests | Response % < 100ms | Response % < 200ms | Response % < 500ms | Response % < 1000ms | Response % > 1000ms | % CPU |
|---|---|---|---|---|---|---|---|
| Node 1 | 413 | 38 | 24 | 16 | 12 | 7 | 23 |
| Node 2 | 409 | 50 | 28 | 13 | 4 | 2 | 10 |
| Node 3 | 480 | 49 | 27 | 15 | 5 | 2 | 17 |
| Node 4 | 381 | 49 | 23 | 15 | 7 | 3 | 1 |

# First, the HTML

```
<html>
<head>
    <title>Realtime Monitoring Demo</title>
    <link href="css/realtime.css" media="all"
          rel="stylesheet" type="text/css"/>
    <script src="javascript/jquery-1.7.1.js" />
    <script src="javascript/underscore-1.3.1.js" />
    <script src="javascript/backbone-0.9.1.js" />
    <script src="javascript/flot-0.7.js" />
    <script src="javascript/realtime.js" />
</head>
...
```

# The HTML, con't

```
<body>
<h1>Realtime Monitoring Demo</h1>
<table border="1">
    <tr>
        <th>Machine</th><th># Requests</th>
        <th>Response % &lt; 100ms</th>
        <th>...</th><th>...</th><th>...</th>
        <th>...</th><th>% CPU</th>
    </tr>
    <tr id="node1">
        <th id="node1_name">Node 1</th>
        <td id="node1_requests">3</td>
        <td id="node1_100ms">33</td>
        <td id="node1_200ms">33</td>
        <td id="node1_500ms">0</td>
        <td id="node1_1000ms">0</td>
        <td id="node1_over1000ms">0</td>
        <td id="node1_cpu">5</td>
    </tr>
```

# The Backbone.js model

```
window.MonitorData = Backbone.Model.extend({
    defaults: function(){
        return {
            requests: 0,
            response100: 0,
            response200: 0,
            response500: 0,
            response1000: 0,
            responseLonger: 0,
            cpu: 0
        };
    }
});
```

# But We Have One of Those For Each Server!

```
window.MonitorDataList = Backbone.Collection.extend({
    model: MonitorData,

    url: 'MonitorData'
});

window.Data = new MonitorDataList;
```

# Now Update The Page

```
window.MonitorItemView = Backbone.View.extend({
    initialize: function() {
        this.model.bind('change', this.render, this);
    },

    render: function() {
        $("#"+this.model.get('id')+"_requests").html(
                    this.model.get('requests'));
        $("#"+this.model.get('id')+"_100ms").html(
                    this.model.get('response100'));
        ...
        $("#"+this.model.get('id')+"_cpu").html(
                    this.model.get('cpu'));
        }
    });
```

# How to Connect That to Server Data?

```
window.MonitorDataView = Backbone.View.extend({
    initialize: function() {
        Data.bind('reset', this.newRows, this);
    },

    newRows: function() {
        Data.each(this.newRow);
    }

    newRow: function(row) {
        new MonitorItemView({model: row}).render();
    },
});
```

# And the Polling Loop

```
window.App = new MonitorDataView;

setInterval(function() {
    Data.fetch();
}, 1000);
```

# Polling Demo

- Started here, because it was the easiest way to make sure the Backbone.js stuff was working
- Scalable? Lots of connection open/close overhead, but few connections per moment

# WebSockets

# First: Long Polling

- Aka Comet, Continuations, etc.
- Not going to show this
- Uses long-running AJAX requests and etc.
- More responsive than polling, but pretty much abusing HTTP to do it
- Now lots of simultaneous connections, plus the overhead of opening & closing them

# With WebSockets

- Only one connection, long-lived and purpose-built
- In this case, just want to receive data pushed from the server
- No reason you can't continue to use AJAX requests for other stuff
    - To the same back end resource, even!

# WebSockets Code

```
var SocketType = window.MozWebSocket ||
                 window.WebSocket;

var socket = new SocketType(
     "ws://localhost:8080/MonitorData", "update");

socket.onmessage = function(msg) {
    window.Data.reset(JSON.parse(msg.data));
};
```

# Back End Code

```java
public class MonitoringServlet extends WebSocketServlet {
    private final Set<ConnectionHandler> sockets = ...
    protected void doGet(...) {...}

    public WebSocket doWebSocketConnect(HttpServletRequest req,
                                        String type) {
        return new ConnectionHandler();
    }
    class ConnectionHandler implements WebSocket {
        private Connection connection;

        public void onOpen(Connection connection) {
            this.connection = connection;
            sockets.add(this);
        }

        public void onClose(int i, String s) {
            sockets.remove(this);
        }
    }
```

# WebSockets Demo

# Gee, That Was Easy...

- Direct way to push data to the browser*
- Some concerns when you dig in deeper
  - Proxies may interfere (esp. non-secure)
  - Connection timeouts & keep-alives
  - No queueing for missed messages
  - No support for multiplexing connections
  - No control channel for administrative packets

# What's the Alternative?

- The Bayeux protocol (Cometd)
- The Socket.io library/protocol
- These are NOT simple abstractions around polling/WebSockets/etc.
- Both require UI *and* back-end support, and define their own message format/protocol
- They can operate *using* WebSockets

# What To Do?

- Is present browser support an issue?
- Do you want a pipe (less overhead) or a messaging protocol (more features)?
- What back-end runtimes are acceptable?
- Is portability/interoperability a concern?

# Charts

# Charting Overview

- Smoothie, flot.js, etc.
- Pretty easy to incorporate
  - All the data lives in the Backbone.js models
  - Rendering the chart goes in the Backbone.js view

# Revised Goal

## Realtime Monitoring Demo

| Machine | # Requests | Response % < 100ms | Response % < 200ms | Response % < 500ms | Response % < 1000ms | Response % > 1000ms | % CPU |
|---|---|---|---|---|---|---|---|
| **Boston** | 657 | 46 | 26 | 14 | 9 | 3 | 17 |
| **NYC** | 733 | 51 | 24 | 15 | 7 | 2 | 21 |
| **Austin** | 785 | 54 | 25 | 14 | 3 | 1 | 15 |
| **San Fran** | 781 | 51 | 27 | 12 | 6 | 2 | 11 |

# Setting Up a Chart

```
initialize: function() {
    ...
    this.chartData = [];
    for(var i=0; i<50; i++) {
        this.chartData[i] = [i,0];
    }
    var options = {
        series: { shadowSize: 0 },
        yaxis: { min: 0, max: 100 },
        xaxis: { show: false }
    };
    this.plot = $.plot($("#chart"),
                       [ this.chartData ], options);
},
```

# Rendering on Every Data Load

```
render: function() { // On App View
    if(Data.series) {
        for(i=0; i<49; i++) {
            this.chartData[i][1] =
                this.chartData[i+1][1];
        }
        this.chartData[49][1] =
                    Data.series.get(Data.field);
        this.plot.setData([this.chartData]);
        this.plot.draw();
    }
}
```

# Desired Behavior

- If you click a cell in the table, it should chart that field
    - Associate a view with a table row
    - Event listeners on the cells
- Not so easy the way the previous demo was built (a static HTML table and a view that just wrote into existing cells)

# Making a Dynamic Table (the HTML)

```
<body>
<h1>Realtime Monitoring Demo</h1>

<div id="chart" style="width:600px;height:300px" />

<table border="1">
    <thead><tr>
        <th>Machine</th><th># Requests</th>
        <th>Response % &lt; 100ms</th>
        <th>...</th><th>...</th><th>...</th>
        <th>...</th><th>% CPU</th>
    </tr></thead>
    <tbody id="DataTable">
    </tbody>
</table>
```

# Making a Dynamic Table (app view)

```
window.MonitorDataView = Backbone.View.extend({
    el: $("#DataTable"),

    rows: [],

    initialize: function() {
        Data.bind('reset', this.newData, this);
        ...
    },
```

# Making a Dynamic Table (event handlers)

```
newData: function() {
    if(this.rows.length > 0) { // Rows Present
        for(var i=0; i<Data.length; i++)
            this.rows[i].model.set(
                                Data.at(i).toJSON());
    } else { // Need to create table rows
        for(var j=0; j<Data.length; j++)
            this.newRow(Data.at(j), this.rows);
    }
}
newRow: function(row, rows) {
    var view = new MonitorItemView({model: row});
    $(this.el).append(view.render().el);
    rows.push(view);
},
```

# Making a Dynamic Table (row view)

```
window.MonitorItemView = Backbone.View.extend({
    tagName: 'tr',
    render: function() {
        $(this.el).html(
          "<th>"+this.model.get('id')+"</th>"+
          "<td class=\"requests\">0</td>"+
          "<td class=\"under100\">0</td>"+
          "<td class=\"under200\">0</td>"+
          "<td class=\"under500\">0</td>"+
          "<td class=\"under1000\">0</td>"+
          "<td class=\"over1000\">0</td>"+
          "<td class=\"cpu\">0</td>");
        this.updateRow();
        return this;
    },
```

# Making a Dynamic Table (row updates)

```
initialize: function() {
    this.model.bind('change', this.updateRow, this);
},

updateRow: function() {
    this.$('.requests').html(
                  this.model.get('requests'));
    this.$('.under100').html(
                  this.model.get('response100'));
    this.$('.under200').html(this.model.get(...));
    this.$('.under500').html(this.model.get(...));
    this.$('.under1000').html(this.model.get(...));
    this.$('.over1000').html(this.model.get(...));
    this.$('.cpu').html(this.model.get(...));
},
```

# Making a Dynamic Table (click events)

```
events: {
    "click .requests" : "clickRequest",
    "click .cpu" : "clickCPU"
},

clickCPU: function() {
    Data.series = this.model;
    Data.cleared = true;
    Data.field = 'cpu';
}
```

# Dynamic Table & Click-to-Chart Demo

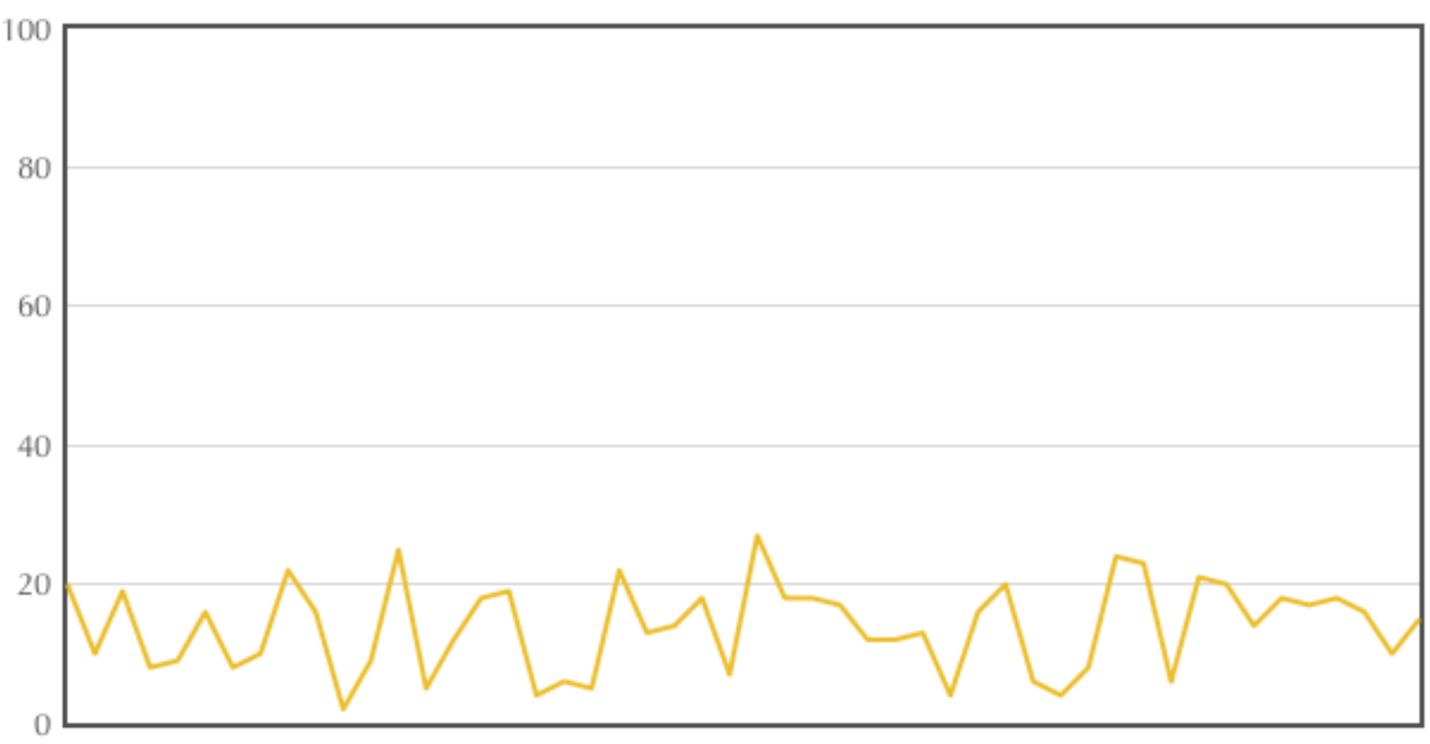
# Web Workers

# Desired Behavior

- CPU field should be colored depending on whether it's over or under the average
- Calculating that should happen on a background thread
  - So it doesn't lock up the UI thread
  - And uses more cores on a modern chip
- *Though of course, this is a pretty simple case...*

# Revised Goal

## Realtime Monitoring Demo




| Machine | # Requests | Response % < 100ms | Response % < 200ms | Response % < 500ms | Response % < 1000ms | Response % > 1000ms | % CPU |
|---|---|---|---|---|---|---|---|
| **Boston** | 762 | 46 | 25 | 15 | 9 | 3 | 15 |
| **NYC** | 731 | 47 | 26 | 14 | 8 | 3 | 20 |
| **Austin** | 769 | 46 | 27 | 13 | 9 | 3 | 9 |
| **San Fran** | 816 | 46 | 26 | 12 | 9 | 4 | 6 |

# Web Workers

- Portable!  *(across very modern browsers...)*
- Scripts that run in a separate thread
- Unable to manipulate the DOM
- Restricted to communicating with the main thread using events
  - Further, data passed back and forth is serialized

# Calling a Web Worker

```
var worker = new Worker("javascript/worker.js");
worker.onmessage = function(e) { // data from worker
    Data.reset(e.data);
};

var socket = ...
socket.onmessage = function(e) { // data from server
    worker.postMessage(e.data);
};          // pushed JSON parsing to the worker


// in MonitorItemView updateRow:
this.$('.cpu').removeClass('goodCPU badCPU')
              .addClass(this.model.get('cpuClass'));
```

# Web Worker Code

```
var averageCPU = {};

onmessage = function(e) { // data from main thread
    var allData = JSON.parse(e.data);
    for(var i=0; i<allData.length; i++) {
        var stats = averageCPU[allData[i].id];
        if(!stats) {...}
        stats.count = stats.count+1;
        stats.total = stats.total+allData[i].cpu;
        allData[i].cpuClass =
          allData[i].cpu > stats.total / stats.count
            ? 'badCPU' : 'goodCPU';
    }
    postMessage(allData);
};
```

# On Web Workers

- The global object is no longer 'window'
  - You still get functions like setTimeout, WebSocket, XMLHttpRequest, etc.
  - At least, you are *supposed to*...
- Can get fancy, and have workers spawn other workers (well, you're *supposed to*...)
- A bit restricted by the serialization in/out

# So Let's Go Further...

```
var worker = new Worker("javascript/worker.js");
worker.onmessage = function(e) { // data from worker
    Data.reset(e.data);
};

// No more socket defined in main thread
```

# ...When IE 10 is Legacy

```
var averageCPU = {};  // Next line fails in FF 11
var socket = new WebSocket("ws://...", "update");
socket.onmessage = function(e) {
    var allData = JSON.parse(e.data);
    for(var i=0; i<allData.length; i++) {
        var stats = averageCPU[allData[i].id];
        if(!stats) {...}
        stats.count = stats.count+1;
        stats.total = stats.total+allData[i].cpu;
        allData[i].cpuClass =
          allData[i].cpu > stats.total / stats.count
            ? 'badCPU' : 'goodCPU';
    }
    postMessage(allData);
};
```

# Web Worker & CPU Coloring Demo

# Final Code Review

# Discussion: When To Use These Techniques

- Games, Chat, Real-time feeds, Whiteboard / collaboration, ...
- But what's the threshold for not polling?
  - In number of users?
  - In frequency of updates?
  - In connect overhead vs. data sent?

# Q&A

ammulder@chariotsolutions.com

http://chariotsolutions.com/presentations