Web Performance
We want speed!

Amazon: 100ms latency = 1% revenue

But too many weak points:

• Design, HTML, CSS, Network,
• Database, Hardware
Design

- Responsive Design
- Mobile-first Design
- Adaptive Design
- Progressive Enhancement – on the server (basic vs enhanced version)

Context (differences) based on

*Mobile, Connectivity, Screen, Browser*
Design Techniques

• Background images -> CSS gradients
• Lazy Image loading
• Multiple image resolutions
• Test browser feature capabilities (Modernizr)
• Render core elements first
HTML

- Large DOM
- Expensive DOM manipulations
- Minimizing Screen reflow
  → Triggered by structural changes

### reflow time by browser

<table>
<thead>
<tr>
<th>DHTML action</th>
<th>Chr1</th>
<th>Chr2</th>
<th>FF2</th>
<th>FF3</th>
<th>IE6,7</th>
<th>IE 8</th>
<th>Op</th>
<th>Saf3</th>
<th>Saf4</th>
</tr>
</thead>
<tbody>
<tr>
<td>className</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td>display none</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>display default</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>2x</td>
<td>1x</td>
<td>1x</td>
<td>-</td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td>visibility hidden</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>-</td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td>visibility visible</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>-</td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td>padding</td>
<td>-</td>
<td>-</td>
<td>1x</td>
<td>2x</td>
<td>4x</td>
<td>4x</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>width length</td>
<td>-</td>
<td>-</td>
<td>1x</td>
<td>2x</td>
<td>1x</td>
<td>1x</td>
<td>-</td>
<td>1x</td>
<td>-</td>
</tr>
<tr>
<td>width percent</td>
<td>-</td>
<td>-</td>
<td>1x</td>
<td>2x</td>
<td>1x</td>
<td>1x</td>
<td>-</td>
<td>1x</td>
<td>-</td>
</tr>
<tr>
<td>width default</td>
<td>1x</td>
<td>-</td>
<td>1x</td>
<td>2x</td>
<td>1x</td>
<td>1x</td>
<td>-</td>
<td>1x</td>
<td>-</td>
</tr>
<tr>
<td>background</td>
<td>-</td>
<td>-</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>font-size</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>2x</td>
<td>1x</td>
<td>1x</td>
<td>-</td>
<td>1x</td>
<td>1x</td>
</tr>
</tbody>
</table>

reflow performance varies by browser and action

"1x" is 1-6 seconds depending on browser (1K rules)
HTML Techniques

- Lazy load content
- Use idle time to pre-load content
- Batch DOM updates
- Set image sizes before loading
- Reduce DOM depth
- Do complex renderings outside of “flow”
HTML Resources

• Optimize images (color & size)
• Minify JS and CSS
• Use CSS Sprites to reduce image requests
• Gzip components
• Add Expires or Cache-Control Header
• Don’t forget every request sends cookies

http://css-tricks.com/css-sprites/
Loading Page...

- Browser: Hey, GET me /index.html
- Server: Ok, let me see if index.html is lying around…
- Server: Found it! Here’s your response code (200 OK) and I’m sending the file.
- Browser: 100KB? Ouch… waiting, waiting… ok, it’s loaded.

HTTP Request and Response

1. Browser Request
   GET /index.html HTTP/1.1

2. Web Server Finds File
   /var/www/.../index.html
   Read File

3. Server Response
   HTTP/1.1 200 OK
   <html>...
   </html>

4. Browser Displays Page

1KB

100KB
Gzip components

- Browser: Hey, can I GET index.html? I’ll take a compressed version if you’ve got it.
- Server: Let me find the file... yep, it’s here. And you’ll take a compressed version? Awesome.
- Server: Ok, I’ve found index.html (200 OK), am zipping it and sending it over.
- Browser: Great! It’s only 10KB. I’ll unzip it and show the user.
• Mind hardware acceleration
• Choose selectors carefully (class vs many ids)
• Manage CSS complexity
• Cache references to elements (JS)
CSS Techniques

• Stylesheets at the Top
• Remove unused CSS rules
• Avoid universal selectors (*)
• Don’t abuse “border-radius” & “transform”
• Prefer selectors with native JS support
  $(‘#’id) --- getDocumentById
  $(‘.class’) --- getElementByClassName
  $(‘tag’) --- getElementByTagName

http://perfectionkills.com
Network

- Each page invokes many resources
- Each resource needs HTTP request
- Finding servers involves DNS
- It takes time to transfer resources
• Make Fewer HTTP requests
• Content Delivery Network (e.g., akamai)
• Split resources across servers - load balance
• But avoid too many DNS lookups
• Cookies can consume bandwidth
• Careful with redirects (301, 302)
Database - Techniques

- Learn to write fast queries
- Query optimization issues
  - “explain”, “show index”, “statistics”
- De-normalize
- Configure Buffers, Cache, Query Cache
• Scale up: faster, better, bigger (quick)
• In this order: memory - hard drives - cores
• I/O is mechanical – avoid or cache everything
• Increase concurrency: 2 hard drives
HTTP persistent connection

HTTP 1.0 uses a single pair of request/response.

As opposed to opening a new connection for every single request/response pair, the idea of using a single TCP connection to send and receive multiple HTTP requests/responses.

HTTP persistent connection, also called HTTP keep-alive, or HTTP connection reuse.
HTTP persistent connection
Advantage

- Lower CPU and memory usage (because fewer connections are open simultaneously).
- Enables HTTP pipelining of requests and responses.
- Reduced network congestion (fewer TCP connections).
- Reduced latency in subsequent requests (no handshaking).
- Errors can be reported without the penalty of closing the TCP connection.
Load Balancing
Node.js Way

By using node-http-proxy, an open source proxy server for Node applications.

npm install http-proxy
Load Balancer: Scaling Node App

var proxyServer = require('http-proxy');
var port = parseInt(process.argv[2]);
var servers = [{
    host: "localhost", port: 8081
}, {
    host: "localhost", port: 8080
}];
proxyServer.createServer(function (req, res, proxy) {
    var target = servers.shift();
    proxy.proxyRequest(req, res, target);
    servers.push(target);
}).listen(port);
Load Balancer: Scaling Node App

NginX Way

By changing NginX built in configuration files.

- round-robin
- least-connected
- ip-hash
Load Balancer: Scaling Node App

Default Configuration

http {
  upstream myapp1 {
    server srv1.example.com;
    server srv2.example.com;
    server srv3.example.com;
  }

  server {
    listen 80;

    location / {
      proxy_pass http://myapp1;
    }
  }
}

Load Balancer: Scaling Node App

Least connected

http {
    upstream myapp1 {
        least_conn;
        server srv1.example.com;
        server srv2.example.com;
        server srv3.example.com;
    }

    server {
        listen 80;
        location / {
            proxy_pass http://myapp1;
        }
    }
}
Load Balancer: Scaling Node App

Session persistence

http {
  upstream myapp1 {
    ip_hash;
    server srv1.example.com;
    server srv2.example.com;
    server srv3.example.com;
  }

  server {
    listen 80;
    location / {
      proxy_pass http://myapp1;
    }
  }
}
Load Balancer: Scaling Node App

Weighted load balancing

http {
  upstream myapp1 {
    server srv1.example.com weight=3;
    server srv2.example.com;
    server srv3.example.com;
  }

  server {
    listen 80;
    location / {
      proxy_pass http://myapp1;
    }
  }
}

References

1. http://cs193h.stevesouders.com/
5. nginx.org/en/docs/http/load_balancing.html