Performance on the Yahoo! Homepage

Nicholas C. Zakas
Principal Front End Engineer, Yahoo!
Velocity, June 24 2010
HELLO
my name is
Nicholas
Principal Front End Engineer

HELLO
my name is Nicholas
Contributor

YUI Library

YUI Library is a set of utilities and controls, written with JavaScript and CSS, for building richly interactive web applications using techniques such as DOM scripting, DHTML and AJAX. YUI is available under a BSD license and is free for all users.

YUI is proven, scalable, fast, and robust. Built by frontend engineers at Yahoo and contributors from around the world, it's an industrial-strength JavaScript library for professionals who love JavaScript.

GET STARTED WITH YUI 2 OR YUI 3

YUI 2 — ROBUST AND PROVEN
- Launched February 2006
- Used on thousands of sites around the Web — including Yahoo!
- One of the most popular, best-documented JavaScript/CSS libraries available
- Comprehensive suite of utilities and widgets

Get Started with YUI 2

YUI 3 — YUI EVOLVED
- Launched September 2009
- YUI’s next-generation framework — used on the new Yahoo home page
- Rebuilt from the ground up to be incredibly fast, powerful, and secure
- Full suite of utilities, widgets coming in 2010
- Gallery of community-contributed components

Get Started with YUI 3

HELLO

my name is

Nicholas
The Challenge:
Create a new Yahoo! homepage*

*add a ton of new functionality
**without sacrificing performance
By the Numbers

345 million
unique users per month worldwide

110 million
unique users per month in United States

(no pressure)
The Legacy
1997
1999
2006
We strapped ourselves in, believing we could make the fastest Yahoo! homepage yet.
Performance is hard
The best features for users aren't always the fastest
Content Optimization Engine determines which stories to display at request time.
Sites can be completely customized by the user.
Popular search topics are determined at request time to display up-to-date info.
Random information about other parts of the Yahoo! network
Apps provide more info on-demand
The Cost of Customization

• Spriting is difficult
  – Hard to know which images will be on the page together

• Limited image caching
  – With content constantly changing, getting images into cache doesn't help much

• A lot more JavaScript/CSS
  – And very different, depending on how the user has customized the page
Performance reboot

Many of the optimizations made on the previous homepage won't work
Coming to peace with reality

We can't optimize everything – so let's just focus on the parts we can.
Areas of Focus

• Time to interactivity
• Ajax Responsiveness
• Perceived performance
The time to interactivity is the time between the initial page request and when the user can complete an action.
Time to Interactivity

• For most pages, happens between DOMContentLoaded and onload
  – Can actually happen earlier
• Links work, forms can be submitted even while the page is loading
  – As long as JavaScript isn't running
• Difficult to measure
Net tab reveals some information
Where DOMContentLoaded and onload occur
YSlow reports onload time
Useful, but doesn't really determine time to interactivity
Goal:
Ensure interactivity by DOMContentLoaded
Simple User Actions

• Clicking a headline to read the story
• Performing a search
• Clicking on a favorite

Wait a second!
You don't need JavaScript for any of that!
Progressive Enhancement FTW!
The more tasks that don't require JavaScript, the faster the user can complete an action.
The page is very functional even without JavaScript.
Not relying on JavaScript for everything allows us an opportunity to deliver what appears to be a faster experience.
JavaScript
Loading onto the page without pain
Best Practices for Speeding Up Your Web Site

The Exceptional Performance team has identified a number of best practices for making web pages fast. The list includes 35 best practices divided into 7 categories.

Put Scripts at the Bottom

tag: javascript

The problem caused by scripts is that they block parallel downloads. The HTTP/1.1 specification suggests that browsers download no more than two components in parallel per hostname. If you serve your images from multiple hostnames, you can get more than two downloads to occur in parallel. While a script is downloading, however, the browser won't start any other downloads, even on different hostnames.

In some situations it's not easy to move scripts to the bottom. If, for example, the script uses document.write to insert part of the page's content, it can't be moved lower in the page. There might also be scoping issues. In many cases, there are ways to workaround these situations.

An alternative suggestion that often comes up is to use deferred scripts. The defer attribute indicates that the script does not contain document.write, and is a clue to browsers that they can continue rendering. Unfortunately, Firefox doesn't support the defer attribute. In Internet Explorer, the script may be deferred, but not as much as desired. If a script can be deferred, it can also be moved to the bottom of the page. That will make your web pages load faster.

top | discuss this rule

Traditional thinking was put scripts at the bottom
<html>
<head>
    <!-- head contents -->
</head>
<body>
    <!-- body contents -->
    <script type="text/javascript" src="yourfile.js">  
    </script>
    <script type="text/javascript" src="yourfile2.js">  
    </script>
</body>
</html>
Our results were upsetting
Putting scripts at the bottom actually caused other problems
Results

• Page would fully render, but be frozen
  – User can't interact while JavaScript is being fetched/parsed/executed

• Delayed onload to after 5s on fast connection

• Time to interactivity tied to onload

• Experience was especially bad over slower connections
  – Page was unresponsive for 30s or more
In order to fix things, we had to get lazy
Loading Scripts Without Blocking
April 27, 2009 10:49 PM

This post is based on a chapter from Even Faster Web Sites, the follow-up to High Performance Web Sites. Posts in this series include: chapters and contributing authors, Splitting the Initial Payload, Loading Scripts Without Blocking, Coupling Asynchronous Scripts, Positioning Inline Scripts, Sharing Dominant Domains, Flashing the Document Early, Using iframes Sparingly, and Simplifying CSS Selectors.

As more and more sites evolve into “Web 2.0” apps, the amount of JavaScript increases. This is a performance concern because scripts have a negative impact on page performance. Mainstream browsers (i.e., IE 6 and 7) block in two ways:

- Resources in the page are blocked from downloading if they are below the script.
- Elements are blocked from rendering if they are below the script.

The Scripts Block Downloads example demonstrates this. It contains two external scripts followed by an image, a stylesheet, and an iframe. The HTTP waterfall chart from loading this example in IE7 shows that the first script blocks all downloads, then the second script blocks all downloads, and finally the image, stylesheet, and iframe all download in parallel. Watching the page render, you’ll notice that the paragraph of text above the script renders immediately. However, the rest of the text in the HTML document is blocked from rendering until all the scripts are done loading.

Browser are single threaded, so it’s understandable that while a script is executing the browser is unable to start other downloads. But there’s no reason that while the script is downloading the browser can’t start downloading other resources. And that’s exactly what newer browsers, including Internet Explorer 8, Safari 4, and Chrome 2, have done. The HTTP waterfall chart for the Scripts Block Downloads example in IE8 shows the scripts do indeed download in parallel, and the stylesheet is included in that parallel download. But the image and iframe are still blocked. Safari 4 and Chrome 2 behave in a similar way. Parallel downloading improves, but is still not as much as it could be.
The best way to load external JavaScript

Posted at July 28, 2009 03:00 am by Nicolas C. Zakas
Tags: Blogging, JavaScript, Performance

Not too long ago, I wrote about "Loading JavaScript without blocking" by creating a dynamic `<script>` tag. When `<script>` tags are in the flow of an HTML document, the browser must stop rendering and wait for the script file to download and execute before continuing (example). Creating a new `<script>` tag via JavaScript avoids this issue because it's out of the flow of the document, so the script file is downloaded and executed without waiting. The result: dynamically loading JavaScript files allows your page to render faster and therefore improve perceived performance.

The best technique

Steve Souders has explored several different ways to load JavaScript without blocking both on his blog and in his books. After thinking about it and experimenting, I've come to the conclusion that there's just one best practice for loading JavaScript without blocking:

1. Create two JavaScript files. The first contains just the code necessary to load JavaScript dynamically, the second contains everything else that's necessary for the initial level of interactivity on the page.
2. Include the first JavaScript file with a `<script>` tag at the bottom of the page, just inside the `</body>`.
3. Create a second `<script>` tag that calls the function to load the second JavaScript file and contains any additional initialization code.

That's it! There's really no need to do anything else. The key takeaway is to have only two JavaScript and make the first one as small as possible. For example, the first file may just contain this function:

```javascript
function loadScript(url, callback) {
  var script = document.createElement("script");
  script.type = "text/javascript";
  if (script.readyState) { // IE
    script.onreadystatechange = function() {
      if (script.readyState == "loaded" || script.readyState == "complete") {
        script.onreadystatechange = null;
        callback();
      }
    };
  } else { // Others
    script.onload = function() {
      callback();
    };
  }

  script.src = url;
  document.getElementsByTagName("head")[0].appendChild(script);
}
```

That's a tiny amount of code to get your bootstraped so it will load incredibly fast (especially when gipped). The actual code on your page ends up looking like this:

```html
<script type="text/javascript" src="http://your.cdn.com/file.js"></script>
```
function loadScript(url, callback) {

    var script = document.createElement("script");
    script.type = "text/javascript";

    if (script.readyState) { //IE
        script.onreadystatechange = function () {
            if (script.readyState == "loaded" ||
                script.readyState == "complete") {
                script.onreadystatechange = null;
                callback();
            }
        }
    } else { //Others
        script.onload = function () {
            callback();
        }
    }

    script.src = url;
    document.getElementsByTagName("head")[0].appendChild(script);
}
<html>
<head>
  <!-- head contents -->
</head>
<body>
  <!-- body contents -->
  <script type="text/javascript" src="smallfile.js"></script>
  <script type="text/javascript">
    loadScript(filename, function(){
      //initialization
    });
  </script>
</body>
</html>
Y.Get.script(YUI.presentation.lazyScriptList,
    { onSuccess: function()
    {
        Y.use("*");
        Y.ModulePlatform.init(Y.dali.config, true);
    }});
Results

• Page is interactive as soon as each section is rendered
• Reduced onload time to ~2.5s on fast connections
• Slow connection experience vastly improved
JavaScript Loads

• Small amount on page load
• Larger amount loaded in non-blocking manner
  – Everything necessary for core JavaScript interactivity
• Ajax responses can specify more JavaScript is needed to handle the response
  – True, on-demand loading
Page Flushing
Getting data out to the browser fast
Flush the Buffer Early

tag: server

When users request a page, it can take anywhere from 200 to 500ms for the backend server to stitch together the HTML page. During this time, the browser is idle as it waits for the data to arrive. In PHP you have the function \texttt{flush()}. It allows you to send your partially ready HTML response to the browser so that the browser can start fetching components while your backend is busy with the rest of the HTML page. The benefit is mainly seen on busy backends or light frontends.

A good place to consider flushing is right after the \texttt{HEAD} because the HTML for the head is usually easier to produce and it allows you to include any CSS and JavaScript files for the browser to start fetching in parallel while the backend is still processing.

Example:

```html
<body>
... <!-- content -->
</body>
<?php flush(); ?>
</head>
... <!-- css, js -->

Yahoo! search pioneered research and real user testing to prove the benefits of using this technique.
Flushing after `<head>` ensures CSS starts to download as quickly as possible

But the user still sees a blank screen until the rest of the HTML is rendered to the browser

**Solution:** flush after major sections of the page have been output
The browser won't render a block-level element inside of `<body>` until the closing tag has been received
Removing page-level wrapper `<div>` allows the head to render as quickly as possible
Twist in the LeBron James sweepstakes

One team jumps ahead in the race to land the NBA's biggest free agent, a columnist says. >> "Down payment"

TRENDING NOW
1. British Airways
2. Robin Soderling
3. Balmain Stakes
4. Spice Movie
5. GPS Devices
6. Jason Bateman
7. iPhone
8. Jupiter Impact
9. Pizza Delivery
10. Jake Knotts
 Twist in the LeBron James sweepstakes

One team jumps ahead in the race to land the NBA's biggest free agent, a columnist says.  

**Today's Special Offer**

Get HD free for life - Display Ad - Ad Feedback

**Trending Now**

1. British Airways  
2. Robin Soderling  
3. Balmain Stakes  
4. Spice Movie  
5. GPS Devices

**DAILY OFFERS**

Netflix - Try for free.

Want to Build Muscle? Try Force Factor Today to See Results! Limited per household.

**Now on Yahoo!**

Jacko times America: Michael Jackson is to dominate the latest American music charts...

Snap, Crackle and Pop: Today's tabloid little-tits in one healthy crunch

Michael's Lost Music: Additional copy goes here

**Yahoo! Sites**

Web  Images  Video  Local  Shopping  More

Sign In  New here? Sign Up  Have something to share?  Page Options

More Yahoo! Sites

More Yahoo! Sites

**Yahoo! For Your Business**

Advertise with Us  Yahoo Developer Network

About Yahoo!

Company Info  Careers  Help/Contact Us  Yahoo Blog - Yodel

Copyright © 2010 Yahoo Inc. All rights reserved.

Privacy Policy  About Our Ads  Safety  Terms of Service  Copyright & Policy
(Actually, we flush a bunch of times as the page is output)

Even when the browser can't render yet, it can still start to download other resources such as images
Areas of Focus

• Time to interactivity
• Ajax Responsiveness
• Perceived performance
The biggest area of concern regarding Ajax performance was around the apps. For our very first test, it sometimes took as long as 7 seconds to load a single app.
What exactly is taking 7 seconds?
The measurement itself was a huge black box – before doing anything, we had to figure out exactly what was happening in that time.
Roundtrip Time

The first step is the amount of time between when the browser sends the request and the time it receives the response.
Parse Time

Next, the JSON response returned from the server has to be parsed
JavaScript/CSS Download Time

Each response can indicate it needs more JavaScript/CSS before the content can be used
Render Time

The amount of time it takes to actually change the display via innerHTML
Where We Started
Fixing Roundtrip Time
What's taking so damn long?
The right-side ads were a roundtrip issue. The server-side ad call took extra time plus the ad markup represented 50-60% of the returned markup.
“Fixing” the ad

Entire right column now renders in an iframe. This defers the ad call until after the app has been loaded in the browser, saving both server time for app rendering and bytes in the response.
Fixing Parse Time
What's taking so freakin' long??
JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language. Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

In JSON, they take on these forms:

An object is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma).

An array is an ordered collection of values. An array begins with [ (left bracket) and ends with ] (right bracket). Values are separated by , (comma).
{  
    "msg": "Hello world!",  
    "day": 6,  
    "found": true,  
}

JSON is super-efficient for transporting numbers, Booleans, simple strings, objects, and arrays
Very inefficient for large HTML strings
Escapement adds a lot of extra bytes
The larger the JSON string, the longer it took to parse

Keep in mind there was no native browser JSON parsing when we began testing the new page

The regular expression validation in the YUI JSON utility (based on json2.js) could take up to 40% of the parse time
Shrinking the size of the HTML by deferring the ad helped

But we still wanted to see if we could eek out better gains
We experimented with an alternate response format where meta data was in JSON form but HTML was included afterward in plain text.
Results were good
But then native JSON parsing hit and a lot of problems went away
Fixing Download Time

What's taking so (*&$Q@! long???
On-demand JavaScript/CSS downloading hurt app loading time

Intended to decrease page load time, and did – but left us with this side effect
Waiting until the user takes an action ensures paying the cost of download.

What if you knew which apps the user was going to use?

**Solution:** predictive fetch of JavaScript/CSS *before* you need it.
After page load, we start to download JavaScript/CSS for the apps on the page.
After onload
Fixing Render Time

WTF is taking so (*&#$Q@! long?!?!?
Actually, render time was okay
Results

```plaintext
fpm-www-prod : pa_perf : module/Mail-14782488

<table>
<thead>
<tr>
<th>Metric</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Last</th>
</tr>
</thead>
<tbody>
<tr>
<td>90th percentile</td>
<td>1512.6</td>
<td>1512.6</td>
<td>1512.6</td>
<td>1512.6</td>
</tr>
<tr>
<td>roundtrip</td>
<td>838.39</td>
<td>677.93</td>
<td>6162.5</td>
<td>701.70</td>
</tr>
<tr>
<td>download js/css</td>
<td>232.96</td>
<td>184.93</td>
<td>1179.0</td>
<td>197.70</td>
</tr>
<tr>
<td>parse response</td>
<td>8.4278</td>
<td>7.6011</td>
<td>15.190</td>
<td>9.4649</td>
</tr>
<tr>
<td>render view</td>
<td>154.79</td>
<td>135.94</td>
<td>792.12</td>
<td>152.33</td>
</tr>
<tr>
<td>requests</td>
<td>79.920</td>
<td>37.914</td>
<td>108.89</td>
<td>83.461</td>
</tr>
</tbody>
</table>
```
Areas of Focus

- Time to interactivity
- Ajax Responsiveness
- Perceived performance
Don't underestimate the power of perception
Initially, the new page was actually slower to load than the previous

To be expected – a lot more JavaScript and CSS
Blank space is bad
Makes it seem like nothing is happening
Adjusting Perception

• Frequent page flushing
  – Progressive rendering avoids a lot of blank space
• JavaScript at the bottom
  – Ensure it doesn't block rendering
• Lazy load JavaScript
  – Decrease time to interactivity
Initially, apps were completely blank while loading (seemed slow)
We changed it to a pseudo-loaded state, which made loading seem faster
In the end, user testing showed that perceived performance of the new page was the same as on the old page
Wrap Up
Lessons Learned

• Time to interactivity is a big deal
• Progressive enhancement creates a better experience
  – Allows for delayed load of JavaScript
• Load as much JavaScript as possible in a non-blocking manner
• Ajax performance is a macro measurement
  – Get more insight by looking at the parts
• Perceived performance is important
Achievements

- Reduced time to onload from ~5s to ~2.5s
  - Actually better than previous version
- Very short time to interactivity
- Reduced time to open apps from ~7s to ~2s
- Maintained perception of speed from previous version
Questions?
Etcetera

- My blog: www.nczonline.net
- My email: nzakas@yahoo-inc.com
- Twitter: @slicknet