COEN 168/268

Mobile Web Application Development

Optimizing Apps For Production

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University

Optimizing Apps For Production

Deploying an app in production is relatively easy

The challenge is doing it the right way to provide the best user experience possible

At a Bare Minimum…

If it is a simple web app:

- Get a domain
- Get a publicly web server
- Upload your code and assets to the server

If is a complex app, you might also have do a build:

- Like if you are using `Ember-CLI`

That's it

Kind of

However, in the real world, users expect

- That your app loads quickly
- That your app is fast when using it
- That your app won't crash
- And many, many other things…

Let's Talk About Performance

- The biggest challenge is loading performance:
  - The time it takes from when the user hits "Go" in their browser to the page is loaded and useable
- The secondary, but often overlooked challenge is run time performance:
  - For example, clicking on a button takes more than **100ms** to get a response which users will notice as slowness

Performance is Even More of a Concern on Mobile

- Slower connections
- Slower devices
- More latency

Optimizing Loading Performance

What happens when you load a page?

1. User hits "Go"
2. DNS resolves URL
3. Browser connect to site
4. Browser starts to download resources:
    - HTML, CSS, JavaScript, images, etc
5. Once the browser has enough to start executing the page it will

Some Challenges

- The browser reads in the web page
- When it finds resources (CSS, JavaScript, images to load), it will start doing so
- However, the page isn't really finished loading until it's loaded the whole HTML page
- Chrome is limited to loading ***6* resources at once per domain**
- Therefore, you need to be careful to load things in the order you want in order for your page to load quickly

The Bottom Line

- Due to latency, you want to minimize the number of individual requests
  - Try to bundle CSS, JavaScript, and images resources together
  - Reduce latency by using Content Delivery Networks (CDNs)
- Due to bandwidth, you want to minimize the size of your assets so that they take less time to transfer
  - Minify your resources so that they are smaller in size

Why Bundle Assets Together?

Ex: Let's say that you have 50ms latency and 100 JS source files to transfer at 50ms each:

Non-Bundled:

```
(50ms latency + 50ms xfer time) x 100 = 25 s
```

Bundled:

```
50ms latency + (50ms xfer time) x 100 = 5.5 s
```

But Wait, You Can Load 6 Things Per Domain!

Non-Bundled:

```
(50ms latency + 50ms xfer time) x 100 / 6  = 4.2 s
```

Bundled:

```
50ms latency + (50ms xfer time) x 100 / 1 = 5.5 s
```

BUT, in the bundled case, you have **5 more connections** to load images, CSS, etc that you would otherwise have to wait on

Therefore, For Huge Success You Should

- Concatenate and minify JavaScript source files together
- Concatenate and minify CSS source files
- Use image sprites instead of individual images
- Load assets from multiple domains or CDNs (cross domain is OK for resources)
- Use server tricks to compress assets, such as gzipping

JavaScript Concatenation and Minification

- Combines various JavaScript files in ORDER into one larger file
    - Loading order is very important. For example, Ember-CLI helps you do that with module dependency tracking
- This reduces the number of individual fetches of files form the server
- Using minification removes comments, whitespace, and mangles variable names to be smaller

JavaScript Concatenation and Minification

There are lots of packages you can use, but these are popular and well-proven:

- UglifyJS
  - `npm install uglify-js`
- YUI Compressor
  - `npm install yuicompressor`

And many more, some good, some bad.

Let's Check Out the jQuery Source Code

- Take the contents of the `src` directory
- There are 81 JavaScript files there for a total of 251 KB
- That is a realistic amount of code with whitespace and comments

Let's make it smaller!

Test 1: Combine all into one file, `jquery.js`

Just use `cat` command:

```
cat [files] > jquery.js
```

Resulting size:
201 KB (20% smaller)

Note: This does not take in account file dependencies, which can be problematic

Test 1, What Happened?

- Some efficiencies since it is in one file
- This does not take in account file dependencies, which can be problematic
- In real life, you might need to arrange the files in the right order to ensure that dependencies are met
- Ember-CLI does this for you

Test 2: Run `jquery.js` compress

Removal of comments and whitespace

```
uglifyjs jquery.js -o jquery-uglified.js
```

Resulting size:
110 KB (56% smaller)

Test 2: What Happened?

- Comments are gone
- Newlines and other whitespace is removed

The Compression Step

From:

```
1 function func(title) {
2    var titleString = 'Title: '   + title;
3    var heading = $('h1')[0];
4    heading.html(titleString);
5 }
```

To:

```
function func(title){var titleString="Title: "+title;var heading=$("h1")[0];heading.html(titleString)}
```

Test 3: Run `jquery.js` mangle and compress

With variable name mangling and removal of comments and whitespace

```
uglifyjs jquery.js -o jquery-uglified-mangled.js
-c -m
```

Resulting size:
74 KB (70% smaller)

Test 3: What Happened?

- Comments are gone
- Newlines and other whitespace is removed
- Variable names are mangled

Mangled, what is it?

(white space added to show mangling)

```
1 function func(title) {
2   var titleString = 'Title: '  + title;
3   var heading = $('h1')[0];
4   heading.html(titleString);
5 }
```

```
1 function func(n) {
2   var t = "Title: "+ n,
3       c = $("h1")[0];
4   c.html(t)6
5 }
```

However, Not Everything Can Be Mangled

Calls into objects cause problems because it can't be mangled:

```
1 var updateTitles = function() {
2   this.titles = [];
3   for(var i=0; i < this.content.length; i++) {
4     this.content[i].title = "Title" + i;
5     this.titles.push(this.content[i].title);
6   }
7 }
```

Also, things have to be inside of function scope to be mangled. Furthermore, all these lookups can affect performance.

Instead, Do This

```
1 var updateTitles = function() {
2   var titles  = [],
3       content = this.content;
4  for(var i=0, iLen=content.length; i < iLen; i++) {
5    var contentAtIndex = content[i];
6    contentAtIndex.title = "Title" + i;
7    titles.push(contentAtIndex.title);
8  }
9  this.titles = titles;
10 }
```

Your code will also run faster because it is more optimized and there are less lookups.

And Mangle Into This:

```
1 var updateTitles = function(){
2   for(var t=[], i=this.content, e=0, l=i.length; l > e; e++) {
3     var n = i[e];
4     n.title = "Title" + e, t.push(n.title);
5  }
6   this.titles = t
7 };
```

Test 4: Run through `gzip`

Web servers can gzip assets, let's gzip:

- `jquery.js` from test 1: 59 KB (79% smaller)
- `jquery-uglified.js` from test 2: 31 KB (88% smaller)
- `jquery-uglified-mangled.js` from test 3: 25 KB (90% smaller)

So, You Notice That Using `gzip` Is the Biggest Bang For Your Buck

- Even with the non-compressed files, gzipping saves the highest percentage
- However, it is still **vital** that you concatenate files because it saves on round trips to the server
- You should configured `gzip` for your `.html`, `.js`, `.css`, etc files
- It will increase CPU load on the server, but it is worth it

How to use `gzip` in Apache `.htaccess`

```
<ifModule mod_gzip.c>
  mod_gzip_on Yes
  mod_gzip_dechunk Yes
  mod_gzip_item_include file .(html?|txt|css|js|php|pl)$
  mod_gzip_item_include handler ^cgi-script$
  mod_gzip_item_include mime ^text/.*
  mod_gzip_item_include mime ^application/x-javascript.*
  mod_gzip_item_exclude mime ^image/.*
  mod_gzip_item_exclude rspheader ^Content-Encoding:.*gzip.*
</ifModule>
```

Example from: http://www.feedthebot.com/pagespeed/enable-compression.html

Using Ember-CLI to build

- As part of `ember build` for production, it will use UglifyJS to:
  - combine, compress, and mangle

This is great, because you do not have to worry about it! Also, it ensures that all files are loaded in order when they are combined.

Just run:

```
ember build --environment production
```

Minifying CSS

- Basically the same as JavaScript
- You can use `lessc` for this:

```
# compile app.less to app.css
$ lessc app.less app.css

# compile app.less to app.css and minify (compress) the result
$ lessc -x app.less app.css
```

CSS Unminified

```
footer {
  z-index: 2;
  position: fixed;
  bottom: 0px;
  width: 100%;
  height: 44px;
  border-top: 1px solid #cccccc;
  background: #efefef;
}
footer ul {
  margin-top: 2px;
}
footer ul li {
  display: inline;
  float: left;
  width: 33%;
  text-align: center;
}
footer ul li a {
  color: #aaa;
  text-decoration: none;
  font-size: 12px;
}
footer ul li a.active {
  color: red;
}
footer ul li a span {
  font-size: 18px;
  line-height: 0px;
}
```

CSS Minified (wrapped from one line)

```
footer{z-index:2;position:fixed;bottom:0;width:1
00%;height:44px;border-top:1px solid #ccc;backgr
ound:#efefef}footer ul{margin-top:2px}footer ul
li{display:inline;float:left;width:33%;text-alig
n:center}footer ul li a{color:#aaa;text-decorati
on:none;font-size:12px}footer ul li a.active{col
or:#f00}footer ul li a span{font-size:18px;line-
height:0}
```

Images

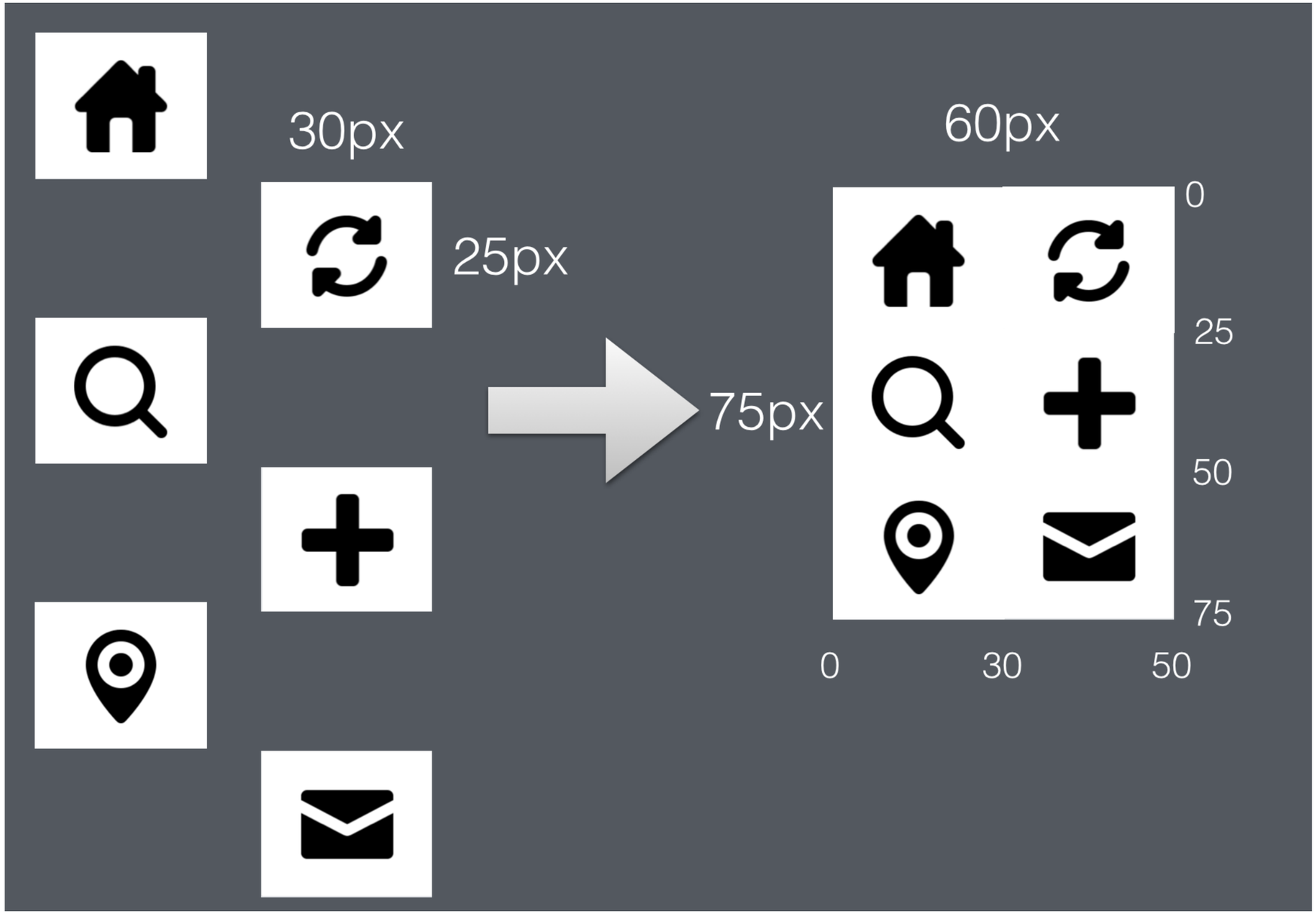
- Images are already compressed
- No `gzip` tricks here can help you
- However, there are things you can do such as **image spriting**
- For photos, use JPGs with lower compression
- For icons and things with solid colors, use PNGs since they compress well

Image Spriting

- Instead of loading each image asset separately you combine them all into a larger image
- This means that the image is larger, but it isn't as large all the images combined
- But you save on the number of calls to load assets
- As a result, you take much less of latency hit when loading assets
- Image spriting is done via CSS background position

How Do You Make A Sprite?

- Let's say that you have 6 icons on your website that are 30px wide and 25 px tall
- Each are PNGs and about 4KB each.
- Combined in a sprite, they are all 5KB together
- Savings are made because PNGs compress solid colors, for example


30px
25px
75px
60px
0
25
50
75
0
30
50

Sprite CSS

```
.icon {
  width: 30px;
  height: 25px;
  background: url(images/sprite.png);
}
.icon.home-icon {
  background-position: 0px 0px;
}
.icon.search-icon {
  background-position: 0px -25px;
}
.icon.search-icon {
  background-position: 0px -50px;
}
.icon.reload-icon {
  background-position: -30px 0px;
}
.icon.add-icon {
  background-position: -30px -25px;
}
.icon.mail-icon {
  background-position: -30px -50px;
}
```

Use CSS Instead of Images

- CSS is text which compresses well, images do not compress
- Whenever possible, try to achieve the effect in CSS
- Resort to an image only when it is strictly needed
- Most things are achievable in CSS if they are not photos

Load Assets From Several Domains

- Loading assets such as JavaScript, CSS, and images are not restricted to one domain
- If you load assets from other domains, then you can get around the browser connection limits
- For example, if you have an image heavy site, you can alternate loading from different domains:
  - `img1.mydomain.com`, `img2.mydomain.com`, etc

Load Assets From a Content Delivery Network

- These are expensive but allow you to cache assets closer to the user
- Instead of having to go to your server, it can hit a cache layer that is much closer
- Closer means that there is less latency
- You can use this for jQuery, for example by loading from `ajax.googleapis.com`:
  - See: https://developers.google.com/speed/libraries/devguide

Load JavaScript Last

- Put `<script>` tags at the bottom of the page
- This allows the HTML page to be loaded faster so that the user sees your initial page instead of a white screen
- Can drastically improve the user's perception of loading if you have a loading screen baked into the page

Take advantage of the async nature of AJAX

- If you have required data that you need to load, try to send off the AJAX call as early as possible in a "bootstrap"
- When you're waiting, on the AJAX call, it won't block a connection
- Even if need to do an AJAX call, you might just do it with raw JavaScript at the top of the page
- While jQuery and other libraries are loading, you can also be waiting on your initial JSON data in parallel

A JavaScript Performance (and Minification) Tip

Cache properties that are accessed using object notation or in functions, to make your code faster and also compress better

```
for(var i=0; i < this.get('content').length; i++) {
  this.get('content').objectAt(i).set('localTime', new Date());
}

var content = this.get('content'),
    date    = new Date();
for(var i=0, iLen=content.length; i < iLen; i++) {
  content.objectAt(i).set('localTime', date);
}
```

Use things such as the Webkit Debugger or YSlow

- Use the Webkit Debugger to analyze loading and runtime speed
- Audit your website using YSlow
- Check out: https://developers.google.com/speed/

Let's try this now!

COEN 168/268

Mobile Web Application Development

Optimizing Apps For Production

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University