That’s a beautiful mobile site. But beauty is only skin deep.

Under the covers, it’s a different thing entirely. It may look like a mobile site, but it’s still a desktop site in mobile clothing. If we want this site to be greased lightning on mobile, we need to start with mobile first. We’ll begin by dissecting the current site to find the desktop bones hiding in its mobile closet. We’ll clean house and start fresh with progressive enhancement, building from the basic content all the way to a desktop view. When we’re done, you’ll have a page that is optimized regardless of the screen size.
Just when you thought it was time to celebrate...

Mike called in a panic. As a reformed web developer, he normally resists the urge to tinker with his site, but he fell off the wagon and decided to make a few tweaks. He thinks he broke the Splendid Walrus site and needs help.

Mike added pictures for all of his new brews to the On Tap Now page. He didn’t modify the code other than to add pictures, but now the page is loading very slowly on mobile phones. It’s so slow that customers have started complaining.

Is there really a problem? How do we know?

Jim: Poor Mike. He knows just enough to get in trouble.

Frank: Exactly. But in this case, I’m not so sure he did anything wrong. Mobile phones have slower networks and processors. Of course the page loads slowly.

Jim: That makes sense, but it still seems slower than expected. Mike says that even on a WiFi network, it is unbearable. And he has a brand-new smartphone.

Joe: Hmm…it sounds like we should at least look into it to see if there is something obvious slowing it down.

Frank: How will we know what’s going on? It could just be the network or any number of things between the phone and the server.

Joe: I’ve been using a plug-in for Firefox that gives you a grade on your page performance. We could use something like that.

Jim: That sounds awesome.

Frank: Can we install plug-ins on mobile browsers?

Joe: Ugh, you’re right. There’s no way to install plug-ins on my phone. Some of my favorite developer tools are browser plug-ins. Without them, how do we know what’s really going on?

Jim: The other day, Kim was showing me how you can watch every request that is made on our WiFi network through the network router’s log page. Could we look at something like that and watch what the phone does?

Frank: That’s a great idea. But instead, let’s use a proxy server. It is very similar to what Kim showed you, but it is designed for exactly this purpose. If we hook up the phone to a proxy server, we can see all of the web requests that the phone makes.

The YSlow plug-in gives you a grade for performance based on how a web page is constructed. Get YSlow at this URL: yhoo.it/yslow. Performance is something every web developer—especially mobile web developers—should care about.

Speaking of which, we’re going to spend a little time looking at performance. Don’t worry. We’ll get to mobile-first Responsive Web Design soon, and we promise all of this performance stuff is related.
A famous athlete is stricken with food poisoning the night before a big match against his archrival. The police suspect foul play and have assigned a detective. The detective quickly starts questioning the best witness: the waitress.

The waitress took the food order, wrote down the request, and handed it to the cook. After the food was ready, the waitress brought the food out. The waitress saw everything.

Most of the time when you’re on the Web, you’re talking directly to the cook. Nothing is between you and the web server for the page you’re visiting.

But if we have a proxy server act like a waitress, it will record both what was ordered and delivered by the browser. Now we can be our own detectives and dig into what actually happened.

If you do a lot of mobile work, you may find it worth your while to learn how to set up a proxy server. It is the best way to see what’s going on between the phone and server.

Unfortunately, setting up a proxy server can be a tad difficult. Thankfully, some kind souls at Blaze, a mobile performance company, have set up a free service that is the next best thing to installing your own tool.
What to do when things aren’t blazing fast

Blaze provides Mobitest—a free mobile performance test using real iPhone and Android phones. Mobitest is located at www.blaze.io/mobile.

Blaze’s Mobitest works like a proxy server. You tell it what web page URL you want to test and what device you want to test with. Mobitest then puts your test request in a queue for that device.

When the phone you requested is available, Blaze tracks all of the communication between its test phone and the web server so you can see what happened.

There is even a fun feature that records a video of the page loading so you can see what someone using that phone would see.

---

Test Drive

Ready for some detective work? It’s time to figure out why the On Tap Now page is slow.

1. **Test the On Tap Now page at** [www.blaze.io/mobile](http://www.blaze.io/mobile).

2. **Look at the load time and page size.**
   The load time tells you how long the page took to load on this phone during this test. The page size is the total size of all resources associated with the page including HTML, CSS, JavaScript, images, fonts, etc.

3. **Try two different phones and compare speeds.**
   Not only will the speed of networks vary, but the phones themselves may also vary in the speed at which they process and display pages.
Don’t let its looks fool you, that’s a BIG page

Yikes! The test says the page is approximately 3 megabytes in size. That would be a pretty big page for a desktop browser. It’s a slow-moving elephant on a mobile phone.

No wonder Mike’s customers complained about the page. It takes over 10 seconds to load on a test iPhone.

What is a waterfall chart? I looked at it, and it doesn’t tell me what’s making the page so big. Is the chart even useful?

Waterfall charts are a common web performance report.

The chart shows the files that the browser requested from the server to build the web page. The bars represent the length of time spent downloading a resource. The resources are listed in the order in which the browser requested them from the server.

But don’t go chasing the waterfall on the Blaze report page. It doesn’t have the details we need for our detective work. We’re going to show you how to find a waterfall that is more useful.
There’s gold in ‘em HAR hills

There is a nugget buried in the Blaze Mobitest results page behind the tiny View HAR file link.

Click that link, and you will go to a new site called the HTTP Archive Viewer and see a more detailed waterfall than the test results page.

This waterfall chart shows us every resource that the browser downloaded in much more detail than the picture of the waterfall on the Blaze report page.

The preview tab contains a waterfall chart for the page we tested. It was generated from a HAR file.

HTTP Archive (HAR) Viewer

The HAR file can be used to make waterfall charts. Read more about HAR at httparchive.org.

HAR stands for HTTP Archive. It is a file specification that provides a standard way to record what happens when a browser requests a web page from a server.

Do you see any suspicious files being downloaded in the waterfall chart?
bloated page pie charts

10,000-feet view: Show statistics

The HAR waterfall charts show you the files that were downloaded, how the server responded, and the time it took to download. But before we dig into the waterfall chart itself, let’s take a look at the high-level statistics.

Click the Show Statistics link on the HAR Viewer page to see a series of four pie charts. The chart that matters most to us is the second one, which breaks down the page by type of file. Hover your mouse over each file type to see its total file size.

Eight JavaScript files for this page. Total size of all JS is 351 KB.

So that’s why the page is so slow. There’s nothing on this page that uses JavaScript, but it’s downloading eight JavaScript files. Plus the images are almost 2.4 MB in size. We need to figure out why the images and JavaScript are so big.
Find the drags on page speed

Now it’s time to dig into that waterfall chart to find where the big images and JavaScript are coming from. Here’s a key to reading the chart.

The amount of communication between the browser and the server can be overwhelming, but don’t worry. You just need to look for two things: which resources are the largest, and where is the JavaScript coming from?

Review the waterfall chart for the On Tap Now page. Find the five largest files and examine them. For each file, answer:

1. **What type of file is it?**
2. **What domain is the file coming from?**
3. **If the file is an image, what is its height and width?**
   Hint: You may need to copy the image URL and open it in a new tab or download the image to find the dimensions.

What does this information tell you about what you might need to do to make the page faster?
Did you find the problems with the page? Let’s review.

1. **What type of file is it?**
   You can determine the type of file by hovering over the filename to see the full URL and extension. You can also add a column containing the file type so you can scan the list quickly.

2. **What domain is the file coming from?**
   Now we’re getting somewhere. Check out where this big, 174.8 KB JavaScript file comes from.

   ![JavaScript file](http://hf-mw.com/ch2/chapter2/ontap.html)

   - GET ontap.html
   - GET taps.css
   - GET maps?f=6&s(source=s_qd)
   - GET bensons_bubbler.png
   - GET chapman_lonsdale.png
   - GET crystal_springs.png

   At 174.8 KB, this is the largest JavaScript file on the page.

   Maps.gstatic.com is a domain for Google Maps. The browser is downloading JavaScript for a map that isn’t displayed on the mobile view. Many of the mysterious files are related to Google Maps.

3. **If the file is an image, what is its height and width?**
   Find the images with the largest file size. Copy the URL and open them in a new window. Without even looking at the height and width, you can tell that these images are far larger than the size they appear on a small screen.

   - GET iVbOwr0KGgoA
   - GET {main,mod_util,fr}
   - GET iVbOwr0KGgoA

   ![Image sizes](images)

   Same file (121 KB) used for both despite the different sizes at which the image appears. *Images are not to scale.*
Where did that Google Maps JavaScript come from?

When you view the On Tap Now page on a mobile phone, the page doesn’t contain a map. Why is that JavaScript downloaded? Let’s open the page in our desktop browser and investigate.

Hey, there’s the map. Mike must have set it up so that it only shows up on wider screens.

Hiding the map on the mobile makes some sense. It’s a bit big for a small screen. Older phones may not be able to handle the map’s complex JavaScript, and we’ve seen that the map has a lot of overhead.

So how did Mike hide the map?

One line to download them all

The map is included in the page via an iframe. The iframe loads all of the components necessary to make the map.

```
<iframe id="map" width="300" height="300" frameborder="0" scrolling="no" marginheight="0" marginwidth="0" src="http://maps.google.com..."></iframe>
```

This single iframe causes 47 files to be downloaded!

Mike hid the map with CSS

Mike figured out how we used media queries to modify the layout for mobile. He added in his own CSS rule inside our media query. The rule Mike added sets the display for the iframe to none.

Unfortunately, while setting the display to none will prevent the map from showing up, it doesn’t prevent it from downloading.
What’s with the big pictures?

The images on this page need to be put on a diet. Let’s look at the waterfall to find the biggest images and see why they’re so big.

<table>
<thead>
<tr>
<th>Image Name</th>
<th>Status</th>
<th>Size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET poweredby.png</td>
<td>200 OK</td>
<td>3.5 KB</td>
<td>670 ms</td>
</tr>
<tr>
<td>GET the_grotto.jpg</td>
<td>200 OK</td>
<td>206.1 KB</td>
<td>590 ms</td>
</tr>
<tr>
<td>GET wells_fargo.jpg</td>
<td>200 OK</td>
<td>156.1 KB</td>
<td>875 ms</td>
</tr>
<tr>
<td>GET taps.jpg</td>
<td>200 OK</td>
<td>440.7 KB</td>
<td>1.24s</td>
</tr>
<tr>
<td>GET transparent.png</td>
<td>200 OK</td>
<td>95 B</td>
<td>52 ms</td>
</tr>
</tbody>
</table>

The taps.jpg file is 440.7 KB, making it the largest file on the page.

But that huge file is the header image, and it isn’t even displayed on mobile.

Right, but that doesn’t mean it’s not downloading.

The taps.jpg image has been hidden from the page in the same way as Google Maps—the display property has been set to none in the CSS. But, as we saw with the map, setting display:none doesn’t stop the content from downloading.

```css
@media screen and (max-width:480px) {
  [Other CSS rules are here]
  .header {display:none;}
}
```

Fluid images are huge images

Another thing to notice from the waterfall is that the brew labels are all large files that range from 93 to 132 KB. They’re desktop images scaled down to fit the screen using the fluid-image technique we learned in Chapter 1.

So this isn’t a new issue, but when we only had one or two images on the page, it wasn’t noticeable. But when you put 16 brew labels on one page, suddenly the fluid images are an anchor slowing the page down.
It looks mobile friendly, but it isn’t

Jim: Well, that’s a bummer. I guess looks can be deceiving, eh?

Frank: At least we can tell Mike he didn’t break the page.

Joe: Yeah, any of us could have made the same mistake. The problems are really bad on this particular page, but I think the same issues exist on every page on the site.

Frank: So what do we do? Build a whole separate site for the mobile version? Ditch Responsive Web Design?

Joe: Let’s not get ahead of ourselves here. There’s got to be a way to make it work. We’re so close right now. Do the image and JavaScript problems have anything in common?

Jim: It seems like all of the problems stem from the fact that we’re starting with desktop-appropriate content—images, maps, etc.—and then hiding that content.

Frank: Exactly. We’ve got big files going to the browser by default and then CSS is being used to try to cover them up. But it seems that, if we’re not careful, the large files will still be downloaded by mobile devices. That’s not the ideal fallback behavior if something goes wrong.

Joe: What if we flipped things around and sent the smallest files by default?

Jim: Oh, interesting. That might work. Start with the mobile templates first and then add on content for desktop.

Frank: What you’re describing sounds a lot like progressive enhancement.

Joe: You’re right. We’ve been using progressive enhancement for years. The only difference now is that we’re starting from mobile and progressively enhancing the document to fit the desktop.

Jim: It seems like it should work. Let’s try it out.

Progressive enhancement promotes building layered web pages. At minimum, everyone can see and use the content. Those with more capable browsers get additional layers of style and interactivity that enhance the experience.
Mobile-first Responsive Web Design

Mobile-first Responsive Web Design (RWD) is exactly what the name suggests: RWD techniques that start from a mobile template. Despite its simplicity, there is a lot of power that comes from this approach.

- Very small screens (feature phones)
  - Basic HTML
  - Simple layout
  - Small images
  - Limited CSS and JS

- Small screens (smartphones)
  - Add newer HTML5 features if supported
  - Simple layout
  - Small images, but bigger than feature-phone size
  - More CSS and JS

- Medium screens (tablets)
  - Because there is more room, we can add optional content like sidebars
  - Multiple column layouts
  - Larger images

- Larger screens (desktops and TVs)
  - Add widescreen layouts
  - Larger images
  - For TVs, optimize navigation for use by people sitting 10 feet away who are using a remote control

*These are just examples of enhancements. What you do depends on the project.*
What is progressive enhancement?

**Progressive enhancement** views web design in a series of layers. The first layer is the content. Combine that with semantic markup to create structured content. If you stop right there, you have a document that nearly every browser in the world can read.

After you’ve got the basics out of the way, you add a presentation layer using CSS and a behavior layer using JavaScript. You never assume the browser supports those features, but if it does, visitors get a better experience.

For many years, web developers commonly built things that only worked on the most advanced browsers and tried to make sure the web page degraded gracefully on older browsers. Progressive enhancement flips this practice around.

Benefits of mobile-first design

Mobile-first RWD isn’t that different from progressive enhancement. Recognizing this fact, many call it *content-first design* instead because content is the first layer of progressive enhancement.

Regardless of what you call it, starting from the most basic document not only reaches the most people, it also has beneficial side effects.

Mobile first is like a small-plate diet. Simply by eating on a smaller plate, you’re likely to eat less food.

The desktop home page is an all-you-can-eat buffet. All sorts of junk gets thrown on it.

Mobile is a small plate. You have to choose carefully and prioritize your content.

And once you’ve got a focused mobile site, you’re better prepared to ask the tough questions like whether or not the things that didn’t make the cut for mobile are really important enough to add back in for desktop.
Let's turn this web page around

Because we’re already using RWD, making our page mobile first won’t take too long. Here is a short list of changes we’re going to make.

- Make the HTML as simple as possible and swap the order of the CSS so that the mobile version is first.
- Fix CSS background images so that only one file gets downloaded per image. Make sure `display:none` is being used appropriately.
- Supply different source files for `<img>` tags at different screen resolutions. Make sure the right size image is downloaded.
- Use JavaScript to add Google Maps to the page when the browser can support it and the document is wide enough to accommodate it.

The current structure of the On Tap Now page

Open up the `ontap.html` file for the Splendid Walrus site in the `chapter2` directory. The file looks very similar to the document we built in Chapter 1:

```
<div class="navigation">...</div>
<div class="header">...</div>
<h1>...</h1>
<div id="visit" class="column">...</div>
<div id="ontap" class="column">...</div>
<div class="footer">...</div>
```

Two columns instead of three

Instead of a `<div>` called “main,” Mike has created a new column with the id of “ontap.” That `<div>` contains the list of current beers.

Because we did a good job of creating a template with semantic markup, the document is clean and simple already. It looks like our main task to make the content mobile first will be removing the Google map.

Because we’re going to need to reference the code later, let’s use HTML comments to prevent the iframe from being included in the page.

Comment out the Google Maps iframe by surrounding it with <!-- and -->.

```
<!--
<iframe id="map" width="300" height="300" frameborder="0" scrolling="no" marginheight="0" marginwidth="0" src="http://maps.google.com..."></iframe>
-->
```
Am I on a new page or not?

The On Tap Now page looks so similar to the home page. How can visitors tell they're on a different page?

Good catch. We have a problem with the order of the content.

On the home page, it was fine if the first thing on the mobile view was the Visit Us information. But if the Visit Us content repeats on every page, visitors won't be able to tell that the page has changed without scrolling down.

We need to reorder the content so the On Tap Now info comes before the Visit Us content.

Do this!

Copy everything in the <div> with the visit id and paste it below the ontap <div>.

```html
<div class="navigation">...</div>
<div class="header">..."</div>
<h1>...</h1>
<div id="ontap" class="column">...</div>
<div id="visit" class="column">...</div>
<div class="footer">...</div>
```

Is the Visit Us content essential on this page? Would it be better to move it to a separate page and link to it? Or maybe leave it out of the mobile page and add it using JavaScript if the page is rendered on a larger screen?
Fix the content floats

The change we made to the order of the content broke the layout in a desktop browser. The Visit Us section is at the bottom of the page.

In Chapter 1, we mentioned how putting the right column before the main column was a trick to make it easier to handle floats for layouts. If you want a block to float next to something, you need to put it first in the source order.

Don’t worry, though. There is a simple fix here. We’ve been floating the Visit Us content to the left of the On Tap Now content. Instead, we need to float the On Tap Now content to the right of the Visit Us content.

Open taps.css and make the following two changes.

Before

```css
@media screen and (min-width:481px) {
  .column {
    margin: 10px 1.04166667% 0 0;
  }
  #visit {
    width: 31.25%;
    float: left;
  }
  #points {
    width: 25%;
    float: right;
  }
  #main {
    margin: 10px 27.0833333% 0 26.0416667%;
  }
  #ontap {
    margin: 10px 0 0 32%;
  }
}
```

After

```css
@media screen and (min-width:481px) {
  .column {
    margin: 10px 1.04166667% 0 0;
  }
  #visit {
    margin: 0 68.75% 0 0;
  }
  #points {
    width: 25%;
    float: right;
  }
  #main {
    margin: 10px 27.0833333% 0 26.0416667%;
  }
  #ontap {
    width: 67%;
    float: right;
    margin: 10px 0 0 0;
  }
}
```

Using 67% instead of 68.75% gives us a little wiggle room for the columns.
Mobile-first media queries

Now for a little housekeeping. In Chapter 1, we started with a desktop website and made it mobile. We’re going to turn this around and start from the simplest content and build up to the desktop (and beyond).

But first we have a confession. Mobile first is a little bit of a misnomer when it comes to the CSS. Before we apply any media queries for small screens, we’re going to set all of the basic styles—for color, type, etc.—and then enhance them.

There is a good reason for doing this. Many mobile browsers don’t understand media queries at all. So we need to make sure they at least get the basic style rules.

Put your CSS house in order

CSS files are often like the kitchen junk drawer. It may start out organized and logical, but over time chaos takes over. To put mobile-first media queries in place, you may need to untangle the basic style rules from the layout rules.

Fortunately, the CSS we built in Chapter 1 is already in good shape. Most of the basic style rules are already at the beginning of the file, with the media queries adding the layout and formatting later in the document. All we need to do is put the mobile media query before the desktop query.

```css
/* Wider viewports/higher resolutions (e.g. desktop) */
@media screen and (min-width:481px) {
  [Desktop layout rules here]
}

/* Mobile/lower-resolution devices */
@media screen and (max-width:480px) {
  [Mobile layout rules here]
}
```

Move the mobile media query block above the desktop media query. By doing this, we’re making sure the cascading effect of CSS is consistent with our mobile-first progressive enhancement approach.

Test Drive

We’ve made quite a few changes to the page:  
- Removed Google Maps
- Fixed the floats
- Reordered the markup
- Reordered the media queries

We better check to make sure things still work. Load the page in a few desktop and mobile browsers to see how it looks. Be sure to check Internet Explorer.
Surprise! The page is broken in Internet Explorer

Don’t tell us you didn’t see this coming the moment we hinted you might want to test the page in Internet Explorer (IE). Battling IE is a rite of passage for web developers. You’ve probably been scarred enough from previous battles that you knew there was an IE-sized monkey wrench awaiting us.

So what’s the catch? **IE doesn’t support media queries.**

Now before you toss the book aside and curse us for teaching something that doesn’t work in the world’s most popular browser, take a deep breath and relax. There are ways to work around IE’s (many) shortcomings.

**Internet Explorer’s escape hatch: conditional comments**

Microsoft has provided a nice tool to help web developers target code specifically to Internet Explorer via conditional comments.

This tests to see if the browser is less than (lt) IE 9 and that it isn’t IE Mobile (!IEMobile). We exclude IE Mobile because it should get the mobile layout. IE9 and above understand media queries, so they don’t need the extra help.

We’re being a little too harsh. IE9 and above do support media queries, so help is coming.

Because IE 8 and below don’t support media queries, IE isn’t getting the CSS rules that create columns.

Internet Explorer’s escape hatch: conditional comments

Microsoft has provided a nice tool to help web developers target code specifically to Internet Explorer via conditional comments.

Look carefully. The HTML comment opens on the first line, but doesn’t close until “—>” is included on the final line. Other browsers will see this as a comment and ignore its content.


If the conditions are met, IE will do whatever is in between the opening [if] statement and the closing [endif]. The example shows a link to a CSS file, but it could be anything you would find in an HTML document.
Use conditional comments with a media query

You probably noticed that the conditional comment points to layout.css. Time to create that file.

We’re going to grab some of the rules from the current stylesheet. We’ve called the new file layout.css because it will only be used for browsers that have enough screen real estate that multicolumn layouts make sense.

1. **Create a blank text file called layout.css and copy the desktop rules into it.**
   Make sure you copy everything between the beginning and end of the media query, but not the @media rule itself.

```css
/* Wider viewports/higher resolutions (e.g. desktop) */
@media screen and (min-width: 481px) {
  .column {
    margin: 10px 1.04166667% 0 0;
  }
  #visit {
    margin: 0 68.75% 0 0;
  }
  #points {
    width: 25%;
    float: right;
  }
  #main {
    margin: 10px 27.0833333% 0 26.0416667%;
  }
  #ontap {
    width: 67%;
    float: right;
    margin: 10px 0 0 0;
  }
}
```

Copy these rules to your new file.

After you copy them, remove the rules and the surrounding media query from taps.css. We'll reapply the rules to the HTML document next.
2 Add a link to the new stylesheet.
For browsers that support media queries, we’re going to add a link to the new layout.css file if the screen size is wide enough.

```html
<link rel="stylesheet" type="text/css" href="taps.css" />
<link rel="stylesheet" type="text/css" href="layout.css" media="all and (min-width: 481px)" />
```

Add this link tag to ontap.html. The 481px value for min-width was copied from the media query we removed from taps.css.

3 Add the IE conditional comment.
We’ve got it working for most desktop browsers. Now we just need to add the conditional comment we created earlier to finish up.

```html
<link rel="stylesheet" type="text/css" href="taps.css" />
<link rel="stylesheet" type="text/css" href="layout.css" media="all and (min-width: 481px)">
<!--[if (lt IE 9)&(!IEMobile)]>
<link rel="stylesheet" type="text/css" href="layout.css" media="all" />
<![endif]-->
```

The conditional comment repeats the line above it, ensuring that desktop IE sees our layout.css file.

4 Time to test again.
Check the page in a browser that supports media queries and different versions of Internet Explorer. Looks good, huh?

Even our persnickety old friend IE is showing the layout properly now.
Q: With super-fast 4G phones on the horizon, is performance really that big of a deal?
A: Absolutely. Even 4G phones end up on the EDGE network occasionally (EDGE is an older, slower network). Studies show that slow sites decrease usage and directly affect the bottom line.

Q: Why am I getting different results from the Blaze Mobitest?
A: There are many reasons why this can occur. Page download time will change with every test depending on network traffic. Google Maps code is different for each operating system and may change over time. The behavior of the phones will also change as new versions of the operating systems are released. For the book, we tested using Blaze’s iOS 4.3, Android 2.2, and Android 2.3 test devices.

Don’t worry too much about the variations in test results. What matters is the code and images being downloaded unnecessarily.

Q: By separating the stylesheet into two files, aren’t you making the site load more slowly?
A: It is true that the number of HTTP requests makes a big difference in the download speed. So we shouldn’t recklessly add requests. In this case, we thought it made more sense to separate them so IE could use the same file.

Q: You mentioned that setting up a proxy server might make sense. What do you recommend?
A: There are many proxy servers, including some fantastic open source ones. We happen to be fans of a commercial product called Charles Proxy.

Q: The lack of plug-ins seems like a big deal. How do you get anything done without Firebug and Web Inspector?
A: It isn’t easy. First, a lot of your debugging work can be done in a desktop browser so long as you are careful to test on real devices at some point in the process. There are also a lot of new tools that attempt to get around the plug-in limitations. The Mobile Perf Bookmarklet (http://bit.ly/mw-perf) includes many performance tools. weinre (http://bit.ly/mweinre) and Opera Dragonfly (http://opera.com/dragonfly) let you run Web Inspector on your desktop and examine what is going on in the phone browser.

Q: It doesn’t seem like much changed when we switched to mobile-first media queries. Why bother?
A: For this page, there wasn’t a big difference between a desktop-first CSS file and a mobile-first one. In our experience, however, this is the exception. With more complex styles, you often want the wider rules to override some, but not all, of the styles set for smaller screens. Reordering the media queries ensures that the CSS cascading behavior is consistent with the goal of progressively enhancing the page as the screen gets wider.

Q: It seems like the order of content may often be different between desktop and mobile. How do you handle this in more complex pages?
A: Ah, you caught that, huh? Yes, this is one of the common challenges for Responsive Web Design. In the long run, the Flexible Box Module (Flexbox) in CSS3 promises an easy way to reorder content in stylesheets. Combine Flexbox with media queries, and you can completely reorder pages as needed. Unfortunately, Flexbox is still young and isn’t fully supported. So developers resort to JavaScript to reorder content or combine RWD with device detection (see Chapter 5). Frankly, content ordering and image handling remain two of the biggest challenges for RWD.

Q: Will the versions of IE that don’t support media queries see the responsive design? Aren’t media queries necessary?
A: Internet Explorer will display the desktop version. It will still have the fluid grids and flexible images. But it won’t change based on any of the media query instructions. If media query support is critical, there is an open source library called Respond.js that fills in support for media queries for older IE versions. This is a fairly intensive script, so be sure to test extensively if you decide to implement it.
How are we doing?

We’ve got the basics in place and our CSS in order. What’s next on our list?

- ✓ Make the HTML as simple as possible and swap the order of the CSS so that the mobile version is first.
- □ Fix CSS background images so that only one file gets downloaded per image. Make sure `display:none` is being used appropriately.
- □ Supply different source files for `<img>` tags at different screen resolutions. Make sure the right size image is downloaded.
- □ Use JavaScript to add Google Maps to the page when the browser can support it and the document is wide enough to accommodate it.

Play taps for the header image

Our waterfall chart showed us that we had one large CSS background image that was being hidden with `display:none`. Despite the fact that the image never shows up on the page, the browser still downloads the image.

So let’s make sure the image is only downloaded when it is needed. How do we do that? By putting it in a media query so it only gets downloaded if the screen is wider than 480 pixels.

But instead of creating a whole new media query, put the CSS rules in `layout.css`, which is already being included in the page via a media query in the `<link>` tag.

```css
.header {
    background: URL('images/taps.jpg') repeat-x;
    height: 300px;
}
```

Check the On Tap Now page using the Blaze Mobitest to make sure `taps.jpg` is no longer being downloaded. Try both iPhone and Android devices. You can use `http://hf-mw.com/ch2/ex/3/ontap.html` if your copy of the page is not on a public server.
It works on iPhone, but the image is still downloading on Android.

Blaze’s Mobitest says Android is still downloading the image, but it is a false report.
Blaze had to modify its phones to make them work for remote testing. This causes some occasional odd behavior.

When you use a stock Android phone, the taps.jpg image will not be downloaded.

Going old school with image optimization

Back in the early days of the Web, web developers spent a lot time worrying about image optimization. As bandwidth has increased, web developers stopped worrying about eking out every bit of performance from images. But mobile devices make image optimization paramount once again.

It looks like we can make the taps.jpg image smaller with some basic web image optimization. Changing the JPEG quality from 80 to 45 makes the image 78 KB instead of 440 KB, and you have to look very closely to see any difference in image quality.

For more on web image optimization, see Chapter 5 of Head First HTML with CSS & XHTML.


Copy the optimized version of taps.jpg from the extras folder into the images folder to replace the original, large file with a smaller version.
CSS images are just the beginning of our image woes. The `<img>` tag presents problems for every responsive design because there can only be one value for the `src` attribute regardless of screen size. So how do we deliver the right size image?

There are 16 beer labels on the On Tap Now page that use an `<img>` tag like this one for Benson’s Bubbler.

Despite the need for multiple versions of this image depending on the screen size, HTML only allows one value for the `src` attribute.

It’s tempting to replace the value of the `src` attribute using JavaScript. Unfortunately, most browsers look ahead at the HTML document and preload images before the JavaScript has been fully evaluated. This often means one size file downloads before the JavaScript changes the `src`, resulting in duplicate downloads and causing the browser to reflow the page layout.

**A responsive image server to the rescue**

If the browser can’t ask the server for the right image, the server will just have to figure it out for itself. That’s what Sencha.io Src attempts to do.

We can use Sencha.io Src to deliver the best-sized image for every device.

CSS can’t be used to override the value of the `src` attribute, either.

The image resizing service formerly known as TinySRC

Set the first part of the `src` to `http://src.sencha.io/`. Replace with your domain and path to the images.

After the slash, add the full URL of the image you want to have resized.

Sencha.io Src only shrinks images. It doesn’t make them bigger. Enlarging images results in poor quality. It is better to find a higher-quality source image.

Sencha.io Src will resize the image to fit the size of the device screen. For example, if an iPhone visits the site, the image will be constrained to its screen size of 320 by 480 pixels.

Do this!

Update all of the brew label images in the `ontap.html` document to use Sencha.io Src.
How Sencha.io Src works

Sencha.io Src works like magic. You request an image from an iPhone, it gives you an iPhone-sized image. Using a feature phone? No problem—Sencha.io Src will give you a tiny image. Sitting at your desktop? Here’s the full image.

How does Sencha.io Src know what size image to deliver? It uses the browser’s user-agent string—an identifier that every browser provides—to look up the device in a big database. The database contains information about thousands of devices. One of the things these device databases track is the size of the screen.

Once Sencha.io Src knows the screen size, it goes to work scaling the image to the maximum width of the device. It stores the image it created in a cache for 30 minutes so subsequent requests for the image at that size will be even faster.

No great solutions for <img> tags

Using Sencha.io Src has drawbacks. It relies on device detection, which can occasionally get things wrong (as we will discuss in more detail in Chapter 5). It also requires you to route all your images through a third party.

The reality is that there are currently no great solutions for how to handle different image sources for different screen sizes. But watch this space closely, because a lot of people are trying to find a better solution.

One final tweak: optimized beer label images

As with the taps.jpg image, we can reduce the file size of the beer label images by saving them at a slightly lower JPEG quality level. Don’t worry, we’ve optimized them for you. Find them in the extras/labels-optimized directory.

Copy the optimized images into the brew_images folder, replacing any existing image files.

Test Drive

We should have an efficient, fast, mobile-optimized web page now. Test it using www.blaze.io/mobile to see how we did. Select the option to have Blaze run three tests on the phone to get an average. Compare the total file size and download time of the new page to the original page.

If your pages are not publicly accessible, you can test using http://hf-mw.com/ch2/ex/4/ontap.html.
That’s a blazing-fast mobile web page

Our diet plan worked! The On Tap Now page is 87% slimmer than it was before. iPhone download time has gone from almost 12 seconds to under 3 seconds.

Before

<table>
<thead>
<tr>
<th>LOAD TIME</th>
<th>PAGESIZE</th>
<th>WATERFALL CHART</th>
<th>Screenshot</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.04s</td>
<td>3027.52kb</td>
<td><img src="image1.png" alt="Waterfall Chart" /></td>
<td><img src="image2.png" alt="Before Screenshot" /></td>
</tr>
</tbody>
</table>

Can you say “Bloated, S-L-O-W loading page”? 

After

<table>
<thead>
<tr>
<th>LOAD TIME</th>
<th>PAGESIZE</th>
<th>WATERFALL CHART</th>
<th>Screenshot</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.56s</td>
<td>389.94kb</td>
<td><img src="image3.png" alt="Waterfall Chart" /></td>
<td><img src="image4.png" alt="After Screenshot" /></td>
</tr>
</tbody>
</table>

The On Tap Now page’s post-optimization results. Much better.

Mike is going to love how fast the site is on a mobile phone now.

Web performance optimization is a growing field with many more ways to make pages faster. What other performance improvements could we make to this page?
Q: Why do browsers download CSS images that are never used?

A: The browser usually can’t know for certain that an image isn’t going to be used. It could be an image that shows up when some JavaScript or CSS activity triggers it. The browser downloads the images in advance so people don’t have to wait if an activity suddenly triggers an image to be displayed.

Q: OK, so why don’t they download images that are within media queries?

A: Originally, browsers downloaded them as well. Browser makers have seen how developers are using media queries and are adjusting browser behavior accordingly. All of this is fairly new, which is why some browsers still download resources inside a media query that doesn’t apply.

Q: Is it safe to route our images through Sencha.io Src? It makes me nervous.

A: Being cautious is reasonable. Any time you integrate a third-party service into a critical part of your site, you’re going to be impacted if that service goes down.

Sencha has said it is committed to providing this service and that it will remain free. At the same time, you can be sure that if a tremendously large site started using it, Sencha would need to be compensated or it wouldn’t be able to run the service.

If you don’t like Sencha.io Src, you could build a similar service using the device detection tools we teach in Chapter 5.

Q: What about other media? Do video and audio suffer from the same problems?

A: In a word: yes. The HTML5 video and audio formats are a little better because they allow you to define fallback versions of the media in different file formats. If your browser doesn’t support the first option provided, it will look at the second one.

But while better, this approach does nothing to address network speed or resolution. Someone using a mobile phone on a wireless network probably doesn’t need an HD-quality movie. By contrast, Apple’s QuickTime video offers a movie reference format that delivers movies based on Internet connection speed.

Q: Is it just me, or are there a lot of unknowns and problems related to Responsive Web Design?

A: There are definitely challenges. As with any new technique, people are still trying to figure out what works and what doesn’t. RWD is bleeding edge. That’s why we’re covering a lot of techniques in this book. It’s likely you’ll need to combine techniques to deliver the best experience for your project.

Despite the challenges, the promise of RWD inspires many people to strive to build more complete solutions. Things are moving quickly when it comes to RWD.

Q: Are there alternatives to Sencha.io Src? Are there solutions to the <img> tag problem that are client side only?

A: There are many different ways to handle <img> tags in responsive designs. A lot of work is currently underway to find a solution that doesn’t require device detection. There are compromises with every solution, including the one we’re using for this project. You can find an extensive review of the techniques at http://bit.ly/rwdimgs2.
to zoom or not to zoom?

Zoom, zoom, pow...

Sorry, guys! I hate to spring a new requirement on you in the middle of your work, but one of my best customers has trouble seeing small text and is complaining that she can’t zoom the page. Can you fix it?

Remember that viewport <meta> tag from Chapter 1? Time to look at it more closely.

The viewport <meta> tag tells the browser the intended dimensions and scaling (aka zoom level) for a page. It also contains controls that can prevent users from being able to change the size of the page.

Zoom in on the viewport <meta> tag

You’ll find the viewport <meta> tag in the <head> of the ontap.html document. The syntax is pretty simple.

```html
<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1" />
```

Width of the viewport. Can be set in pixels or can be set to “device-width,” which tells the browser to match the viewport to the device resolution.

What type of <meta> tag is this?


Sets the initial scale (or zoom level) of the page. Setting it to 1 means that the document should be displayed at its normal scale.

Declares a limit on how much the page can be scaled up. There is also a similar minimum-scale setting.

The maximum-scale is what is preventing the users from zooming the page.
The right to zoom?

It seems like being able to zoom is important for accessibility. Why would anyone ever turn it off?

It can make a difference for accessibility. Some web developers have gone so far as to declare that zooming on mobile is a fundamental human right.

We wouldn’t necessarily go that far, but zooming is important, and it should be considered carefully before it is disabled.

As for why designers disable scaling, there are a few reasons. If the page is using complex touch gestures, disabling zoom makes it easier for people to swipe successfully.

There is also a bug in iOS that causes the zoom level to change when the device is rotated into landscape mode. The bug zooms the page in, causing the right side of the page to get cut off.

Turn zooming back on

To turn zooming back on, we need to remove the maximum-scale setting from the viewport <meta> tag.

Do this!

Edit the viewport <meta> tag and remove “maximum-scale=1.”

Make sure you remove the extra comma after “initial-scale=1.”

After you turn zooming back on, rotate an iPhone or iPod Touch to see the iOS zooming bug in action.
Back to our regularly scheduled project

With our emergency viewport adventure out of the way, let’s take a look at our progress. Our fast mobile page puts us very close to a mobile-first RWD. All that’s left to do is add the map back in if the screen is big enough.

- Make the HTML as simple as possible and swap the order of the CSS so that the mobile version is first.
- Fix CSS background images so that only one file gets downloaded per image. Make sure `display:none` is being used appropriately.
- Supply different source files for `<img>` tags at different screen resolutions. Make sure the right size image is downloaded.
- Use JavaScript to add Google Maps to the page when the browser can support it and the document is wide enough to accommodate it.

Add the map back using JavaScript

The only remaining item is to add the map back if the browser window is wide enough. We’ve already seen that, if we hide and show the map using CSS, the resources for the map will still get downloaded.

So we’re going to need to use JavaScript to add the map when appropriate. Think of it as a JavaScript version of the media queries we know and love.

Grab that Google Maps iframe code that we set aside earlier. We’re going to need to put that back into the page in order to show the map. Let’s take a closer look at the iframe code.

```html
<!--
<iframe id="map" width="300" height="300" frameborder="0" scrolling="no" marginheight="0" marginwidth="0" src="http://maps.google.com..."></iframe>
-->
```
On second thought, a map would be useful

Location. Location. Location.

The old saying takes on new meaning when it comes to mobile phones. Because many phones can tell where you are via GPS and other forms of triangulation, using location to provide more relevant content is common.

Mike hid the map on mobile because it was too big. Now that we’ve seen how many files it downloads, it makes sense to keep the map hidden.

But that doesn’t mean a map wouldn’t be nice. So instead of embedding a map on narrow screens, let’s link to the map.

Add a link to the map

To link to the map, we’ll need a `<div>` that our JavaScript can reference. The `<div>` will contain a `<p>` tag with a link to the map. Why do you think we need to order it that way?

```html
<div id="mapcontainer">
  <p id="maplink">
    <a href="http://g.co/maps/jm2pw">View Google Map</a>
  </p>
</div>
```

We’re going to need a container for the JavaScript to reference, so we’ll add a `<div>` here. The id on the `<p>` tag will allow us to insert the iframe above the link for wider screens. More on this soon.

Add the new `<div>` above the commented-out iframe code.
**Build a pseudo-media query in JavaScript**

Let’s take a look at the JavaScript code we’re going to use to insert the iframe into the page. The code acts like a very simple media query.

```javascript
var breakpoint = 481,
    id = 'mapcontainer',
    viewportWidth = window.innerWidth;

if (viewportWidth > breakpoint) {
    var mapElement = document.createElement('iframe');
    mapElement.id = 'map';
    mapElement.width = '300';
    mapElement.height = '300';
    mapElement.frameborder = '0';
    mapElement.scrolling = 'no';
    mapElement.marginheight = '0';
    mapElement.marginwidth = '0';
    mapElement.src = 'http://maps.google.com/maps?f=q&source=s_q&hl=en&geocode=&q=334+NW+11th+Ave,+Portland,+Oregon+97209&aq=&sll=37.0625,-95.677068&sspn=58.164117,80.32031&vpsrc=0&ie=UTF8&hq=&hnear=334+NW+11th+Ave,+Portland,+Oregon+97209&t=m&ll=45.525472,-122.68218&spn=0.01804,0.025749&z=14&output=embed';
    document.getElementById(id).insertBefore(mapElement, maplink);
}
</script>
```

Sets the breakpoint variable to 481 pixels. The breakpoint is the width at which the map will be added to the page.

Checks to see if the window viewport is larger than the breakpoint.

These lines add all of the attributes to our new iframe element. The attributes and their values were copied from the Google Maps iframe snippet.

This final step adds the iframe (mapElement) into the mapcontainer <div> (id) before the paragraph containing the link (maplink).

**Remove the commented-out iframe code**

We no longer need the original iframe code, so delete it from the HTML document.

```html
<iframe id="map" width="300" height="300" frameborder="0" scrolling="no" marginheight="0" marginwidth="0" src="http://maps.google.com..."></iframe>
```
Add the JavaScript to the On Tap Now page

Now we need to add the JavaScript to the page. Because the map is a nice-to-have feature and not essential, we’re going to make it one of the last things the browser adds to the page.

Open ontap.html and find the bottom of the HTML document.

We’re going to add our JavaScript as the very last thing on the page before the closing </body> tag.

Putting nonessential JavaScript at the bottom of the page is a great way to make a page load faster. The browser will parse all of the HTML and CSS before it gets to the JavaScript. Our visitors will have a usable page more quickly and won’t be stuck waiting for the map code to load.

Time to put our work to the test. Grab ontap.html and answer the following:

1. Does the JavaScript get downloaded on mobile phones?
   Load the page on iPhone and Android using www.blaze.io/mobile/. Check the waterfall chart to see if the Google Maps code is downloading. If your web page isn’t on a public network, you can use http://hf-mw.com/ch2/ex/5/ontap.html to test.

2. Does the map show up on larger screens?
   Open the On Tap Now page in your favorite desktop browser. Does the map show up when the window is wider than 480 pixels?

3. How does the map fit into the responsive design?
   Try adjusting the size of your browser window. Does the map scale like the rest of the design? Are there any problems with the map?
is the map responsive?

How’s it looking? Any problems?

1. **Does the JavaScript get downloaded on mobile phones?**
   
   Check the page using an iPhone on the Blaze Mobitest service. Make your way to the detailed waterfall chart by clicking the HAR file link on the results page to see all of the files downloaded.

   The map files are not getting downloaded. Perfect.

   Now what about Android? Blaze says the JavaScript still downloads, but it is another false report.

   We mentioned that Blaze had to modify its phones to make them work for remote testing. One odd by-product of this modification is that JavaScript running on its test phones reports the screen width as much wider (800 pixels!) than what unmodified phones do.

   You’ll have to take our word for it that the JavaScript works and the map code isn’t downloading on Android, either.

2. **Does the map show up on larger screens?**
   
   Yep. The map looks great in our desktop browser.

   The map shows up in Chrome, which means our JavaScript is working.

3. **How does the map fit into the responsive design?**
   
   Uh oh. We’ve got some problems here. The map doesn’t scale like the rest of the responsive design. Not only that, but there are some screen widths where the map overlaps the beer labels.

   Yikes. When the browser window narrows, the map overlaps the beer labels.

   Why isn’t the map scaling like the images on the page?
These widgets aren’t responsive

Responsive Web Design is so new that widgets like Google Maps are unlikely to be fluid by default.

When companies provide widgets to embed in other web pages, they do everything they can to make sure the widget will work regardless of the page layout. That often means hardcoding things like height and width in the HTML itself.

Dealing with poorly built third-party widgets is a problem for nearly every mobile site. Responsive designs have an additional requirement that widgets be fluid.

```
<iframe id="map" width="300" height="300" frameborder="0" scrolling="no" marginheight="0" marginwidth="0" src="http://maps.google.com..."></iframe>
```

Width and height are fixed, which prevents the map from scaling.

Many of the attributes on this iframe could be moved to CSS.

Ideally, our HTML would only contain the content and markup. It wouldn’t contain any presentation information.

Which CSS properties map to the attributes used in the iframe?
**Move iframe attributes to CSS equivalents**

Let’s move as many of the iframe’s attributes to CSS as possible and make them fluid while we’re at it.

First, we need to create a list of attributes we want to move to CSS by identifying which attributes are for presentation and which are content or metadata.

All those presentation attributes belong in the CSS.

**Match styles to attributes**

Some of the attributes share the same name with their CSS comrades. We don’t have to look hard to find the CSS version of width and height. Others, like frameborder, are obscure attributes. Fortunately, the CSS counterparts are still fairly straightforward.

```css
#map {
    width:100%;
    height:100%;
    border:none;
    overflow:hidden;
    margin:0;
}
```

To make the iframe fluid, we’re changing from a set number of pixels to a percentage like we learned in Chapter 1.

In CSS, scrolling is controlled by the overflow property. This says we want to hide any extra content instead of adding scroll bars. It accomplishes the same thing as scrolling = “no”.

Add these rules to layout.css.

Do this!
Remove attributes from the JavaScript

Now that we’ve got CSS doing the heavy lifting, let’s modify our JavaScript so the presentation attributes aren’t set. Remove the lines that add the presentation attributes that we identified.

```javascript
<script type="text/JavaScript">
var breakpoint = 481,
   id = 'mapcontainer',
   viewportWidth = window.innerWidth;
if (viewportWidth > breakpoint) {
   var mapElement = document.createElement('iframe');
   mapElement.id = 'map';
   mapElement.width = '300';
   mapElement.height = '300';
   mapElement.frameborder = '0';
   mapElement.scrolling = 'no';
   mapElement.marginheight = '0';
   mapElement.marginwidth = '0';
   mapElement.src = 'http://maps.google.com/maps?f=q&source=s_q&hl=en&geocode=&q=334+NW+11th+Ave,+Portland,+OR+97209&aq=&sll=37.0625,-95.677068&sspn=58.164117,80.332031&vpsrc=0&ie=UTF8&hq=&hnear=334+NW+11th+Ave,+Portland,+Oregon+97209&t=m&ll=45.525472,-122.68218&spn=0.01804,0.025749&z=14&output=embed';
   document.getElementById(id).insertBefore(mapElement, maplink);
}
</script>
```

Find the JavaScript at the bottom of ontap.html.

Test Drive

No one should have trouble finding the pub now

The map got a little full of itself, didn’t it? It nearly took over the whole left column. Soon it will start singing, “I’m the map, I’m the map” to get our attention—unless we tame it.

If you make the window narrow, you can begin to see why the map might be trying to get our attention. The map gets squeezed until it turns into a thin, tall strip that is completely unusable.

We still want the map to scale, but we need to set some boundaries on how far it scales in each direction.

1. The height of the map is too big.
   Setting the height to 100% makes the map longer than the tallest image on the page. Let’s keep the map a little more under control by setting the height to 400px.

   Add this line to the #map CSS rule:

   ```
   height: 400px;
   ```

2. Hey look, the map is wearing skinny jeans.
   When the window gets narrow, the map gets so thin that most of the information cannot be seen. We need to set a minimum width so that the map doesn’t go beanpole on us.

   After adding the two new lines, your #map rule in layout.css should look like this.

   ```
   #map {
     width:100%;
     height:400px;
     border:none;
     overflow:hidden;
     margin:0;
     min-width: 200px;
   }
   ```

These screenshots are from Safari. Other browsers behave differently. The map isn’t doing what we want in any of them.
The map overlap is back

Moving the iframe presentation attributes into CSS was the first step. Now we need to take a fresh look at our media queries.

In Chapter 1, we used media queries to switch the layout at 480 pixels. We determined that width based on the width of popular smartphones.

What we’re seeing with the map is that we need to look at the content of the page when we make decisions about where to apply media queries.

The map is covering up the beer labels again when the window is narrow.
Let the content be your guide

There’s a problem with using 480 pixels as our breakpoint for the media query. Not every phone has the same width. And even if a majority of them do today, who’s to say that 540 pixels won’t be the most common size in the future?

A better approach is to let your content be the guide on when to make changes to the layout.

We’re not asking you to commune with your content until it starts to speak to you. But if you adjust the size of your browser window until things don’t look right, the content will tell you a lot.

Maybe images are getting too small. Maybe the columns are too narrow.

When those things happen, that’s where you need a breakpoint. Then you can craft a media query to change the presentation at that breakpoint.

We shouldn’t pay so much attention to typical mobile and desktop screen sizes. When the content breaks the layout, it is telling us to adjust our media queries and JavaScript.
Time to bend and stretch that browser

We need to put our content through its paces by making the page as big and as small as we can while watching for when the layout breaks.

But before we do that, we need some way of knowing how big the screen is when something looks wrong on the page. The easiest way to do this is to install a *bookmarklet* that will show you the window size.

Install the bookmarklet in your browser

Go to [http://bit.ly/window-resize](http://bit.ly/window-resize) and drag the link labeled Window Size into your bookmarks toolbar to create the bookmarklet. Click the bookmarklet to activate it. Resize your browser and watch the numbers change in the upper-left corner of the browser window.

Optional: Install an extension

There is an extension for Google Chrome that not only will show the window size, but will also resize your window to match common screen resolutions. You can get it at [http://bit.ly/chrome-resizer](http://bit.ly/chrome-resizer).


Sharpen your pencil

Load the On Tap Now page in the browser with the Window Size bookmarklet (or a browser extension). Activate the bookmarklet. Resize the browser.

Write down the width of the browser when the layout breaks or the content looks odd.
Let's review some of the trouble spots that show up when you resize the browser.

**610 pixels: Beer labels touch the map.**

At around 610 pixels, the labels touch the map. If we're going to create a new breakpoint to address this problem, we'll need to do it before they touch. This means instead of using 610, we'll use 640 pixels as the breakpoint.

**1,200 pixels: Huge beer labels.**

As the browser gets wider, the beer labels become ridiculously big. Where they become too big is an aesthetic judgment. For our tastes, they start getting too big when the browser is 1,200 pixels wide.

**BRAIN POWER**

Did you see other problems as you resized the browser? How significant do you think a problem needs to be before it makes sense to address it with an additional media query?
Breakpoints to the rescue

All in all, not too bad. Just a couple of small tweaks to the CSS should do it.

Shrink the humongous beer labels

There are currently three beer labels in each row. When the page gets wider, there is room for four beer labels per row.

Create a media query for windows wider than 1,200 pixels that changes the beer labels to four across the page.

```css
@media screen and (min-width:1201px) {
  .taplist li {
    width: 25%;
  }
}
```

This change only happens if the window is bigger than 1,200 pixels.

Setting the width of the list item (li) containing the beer labels to 25% will put four labels on each row.

Going to one column sooner

Even if the beer labels didn’t overlap with the map, the layout is getting very crowded at 640 pixels. Instead of adding a new breakpoint to address the overlap, we can move our existing media query from 480 pixels to 640 pixels.

Making this change will convert the layout to a single column and hide the map. This has the added benefit of applying the single-column layout to phones bigger than 480 pixels.

```html
<link rel="stylesheet" type="text/css" href="layout.css" media="all and min-width: 641px">
```

```javascript
var breakpoint = 641,
...
</script>
```

It’s common to make images smaller proportionally as screens get wider.

Our HTML, JavaScript, and CSS all reference 480 pixels, so we’ll need to update all three.

```css
@media screen and (max-width:640px) {
  // Mobile/lower-resolution devices
}
```

Set max-width to 640px.

Set breakpoint in JavaScript to 641px.

/* Mobile/lower-resolution devices */
```css
@media screen and (max-width:640px) {
  ...
}```

Set min-width to 641px.

It's common to make images smaller proportionally as screens get wider.

Our HTML, JavaScript, and CSS all reference 480 pixels, so we’ll need to update all three.

```html
<link rel="stylesheet" type="text/css" href="layout.css" media="all and min-width: 641px">
```

```javascript
var breakpoint = 641,
...
</script>
```

Set breakpoint in JavaScript to 641px.

It’s common to make images smaller proportionally as screens get wider.

Our HTML, JavaScript, and CSS all reference 480 pixels, so we’ll need to update all three.
Our mobile-first responsive design is complete

- Make the HTML as simple as possible and swap the order of the CSS so that the mobile version is first.
- Fix CSS background images so that only one file gets downloaded per image. Make sure `display:none` is being used appropriately.
- Supply different source files for `<img>` tags at different screen resolutions. Make sure the right size image is downloaded.
- Use JavaScript to add Google Maps to the page when the browser can support it and the document is wide enough to accommodate it.
Q: What exactly is a viewport?

A: Imagine taking a sheet of cardboard and cutting out a rectangle in the middle of it. Lay that rectangle over your monitor so you can only see the portion of the web page that shows through the rectangle. That’s what a viewport does for web pages.

Q: So the viewport <meta> tag tells the browser what size to make the viewport?

A: Exactly. By default, iOS sets the viewport to 980 pixels. If you’ve optimized your page for smaller screens, setting the <meta-viewport> tag lets the browser know to set the viewport accordingly.

Q: What are breakpoints?

A: Breakpoints are just a fancy way of describing the resolution at which a designer decides to change the layout of a page. This is usually done via media queries checking to see if a page is narrower or wider than a certain number of pixels.

A complex responsive design may have multiple breakpoints, including some that make wholesale changes to the layout as well as some minor breakpoints that only make a few targeted tweaks to fix minor layout issues.

Q: Why does the overlap with the map occur in the first place?

A: Because the map is an element that doesn’t scale with the browser window. When the window is small, the browser can’t scale the map any smaller, so the left column ends up overlapping the right column.

Q: Doesn’t adding a min-width to the map break the responsive design by creating an element that doesn’t scale with the browser window?

A: Technically, yes. It seems like a decent solution here because we’ve modified the media queries to address overlapping content. Another option would have been to use media queries to adjust the dimensions of the map and proportions of the columns.

Q: I don’t want to prevent people from zooming, but that iOS bug is pretty heinous. Is there any way to enable zooming and not have a broken page?

A: You can find a JavaScript workaround at https://gist.github.com/901295.

--

**BULLETPoints**

- Adding media queries to an existing desktop site may make it look good on mobile, but doesn’t mean that it is mobile optimized.

- Because most mobile browsers don’t support plug-ins, there are fewer tools to assist mobile web developers.

- Using a proxy server or a testing solution like Blaze Mobitest can help you see what is actually getting downloaded by a mobile browser.

- HTTP archive files and waterfall charts are essential performance tools.

- Mobile-first Responsive Web Design helps optimize web pages by making sure that smaller resources are downloaded by default.

- Mobile-first RWD is another form of progressive enhancement that uses screen size to determine how to enhance web pages.

- Designing for mobile first forces you to focus on what really matters, thus helping you remove cruft from pages.

- Internet Explorer 8 and below do not support media queries. Conditional comments are a workaround.

- JavaScript can augment media queries by testing for screen size and adding content when appropriate.

- Instead of designing breakpoints based on the typical screen resolutions, let the content dictate the resolutions at which you need to modify the layout.