


---

# Responsive Image Distilled

Abhishek Tiwari 

Citation: *A. Tiwari*, "Responsive Image Distilled", Abhishek Tiwari, 2013.  
[doi:10.59350/7xbst-est95](https://doi.org/10.59350/7xbst-est95)

Published on: December 15, 2013

A lot has been written about current state of responsive images. Things like why we need responsive images<sup>1,2</sup>, use-cases<sup>3</sup>, requirements<sup>4</sup> and how to choose a responsive image solution<sup>5,6,7,8</sup> are well documented elsewhere.

This article is my attempt to distill signal from noise. In this article, I will discuss anatomy of responsive image solutions and what you need to know before implementing or choosing a responsive image solution. I will discuss the current state of the art and cover emerging standards to make working with responsive images easier.

## What is a responsive image?

A responsive image is an image that adapts in response to user agent's environmental conditions such as pixel-density, orientation, max-width, max-height, network connectivity, device type , etc.

### Environmental conditions

Environmental conditions are mainly expressed as CSS media features<sup>9</sup> such as aspect-ratio, viewport (height, and width), orientation, resolution, device-pixel-ratio, monochrome , etc. To some extent environmental conditions are also dictated by media types. In addition, there are environmental conditions not specific to CSS such as network connectivity, browser support for different image formats and protocols.

### Adaptations

Adaptations are generally applied to a source or master image. Adaptations can include, but are not limited to,

- changing the dimensions (*in response to viewport, aspect ratio*)
- cropping and/or zoom (*in response to orientation, art direction*)
- changing file format (*in response to browser support to WebP etc.*)
- changing the quality (*in response to network connectivity, viewport*)

---

<sup>1</sup>[Why We Need Responsive Images](#)

<sup>2</sup>[Why We Need Responsive Images: Part Deux](#)

<sup>3</sup>[Use Cases and Requirements for Standardizing Responsive Images](#)

<sup>4</sup>[Use Cases and Requirements for Standardizing Responsive Images](#)

<sup>5</sup>[Choosing A Responsive Image Solution](#)

<sup>6</sup>[Which responsive images solution should you use?](#)

<sup>7</sup>[RESPONDING TO THE UNKNOWN - The Choose Your Own Adventure approach to selecting a responsive images technique.](#)

<sup>8</sup>[Infographic: Responsive Images Problems and Solutions](#)

<sup>9</sup>[Media features](#)

- changing the resolution (*in response to pixel ratio, resolution, print*)
- changing source image (*in response to art direction*)
- changing to monochrome (*in response to e-ink displays, print*)

## Common Misconception

A very common misconception - responsive images are required solely by responsive web design (RWD) which is a frontend problem hence it requires a frontend solution (basically a special markup solution). First let me clarify, responsive images are not specific to RWD, they can be used with

- any type of application (mobile, web),
- any media type (screen, print),
- any type of web design (RWD, no-RWD).

For instance, responsive images can be used to match media features and media types — what you see on screen you get in print.

## Key considerations

When implementing a responsive image solution it is worth looking at following key challenges beforehand.

### Detecting environmental conditions

Environmental conditions are generally detected using media queries which consist of a media type, contain one or more expressions involving media features. A media query normally resolve to true or false. In addition, using `window.matchMedia` Javascript code can react when a media query condition is true or false or changed. This can be also used to set appropriate cookies and to send environment hints to server-side.

### Detecting network connectivity

To detect network connectivity there is no standard solution. Client side libraries like Foresight.js<sup>1011</sup> allows to detect the network connectivity by running image speed test. Speed tests run periodically and results are set or updated in cookies. Please note that speed testing is just additional overhead, basically you have to download one more file.

<sup>10</sup>Estimates network connection speed prior to requesting an image

<sup>11</sup>Mobile first image loading with bandwidth detectionn

## WebP support detection

Browser detection using the user agent string or property can be performed both client-side as well as server-side. Please note detection using the user agent string is unreliable and it should be used in conjunction with other details such as version number, rendering engine, etc. If the sole purpose of the browser detection is to detect support for WebP then this can be also accomplished by reading `Accept` header<sup>12</sup>. User agents with support for WebP normally advertise it in the `Accept` header as `image/webp`<sup>13</sup>.

`Accept` header for Opera,

```
Accept: text/html, application/xml;q=0.9, application/xhtml+xml, image/png, image/webp, image/jpeg, image/gif, image/x-xbitmap, */*;q=0.1
```

`Accept` header for Chrome,

```
Accept: image/webp, */*;q=0.8
```

To serve images in WebP format, servers, proxies and CDN will need to respect and respond `Accept` header<sup>14</sup>. For CDN, origin server must specify `Vary: Accept` in generated responses<sup>15</sup>.

## Device detection

Device detection is mainly required by server-side responsive image solutions. Server-side device detection relies on a device-detection frameworks such as WURFL<sup>16</sup> and DeviceAtlas<sup>17</sup>. Normally these frameworks maintain a database of device information and an API to help mapping a HTTP request to a device profile.

## Preloading

Resource preloading is implemented by most of modern browsers to make pages load faster<sup>18</sup>. In some cases preloading brings upto 20% speed improvement<sup>19</sup>.

---

<sup>12</sup>[Client-Side vs Server-Side Detection for WebP](#)

<sup>13</sup>[Deploying WebP via Accept Content Negotiation](#)

<sup>14</sup>[Deploying WebP via Accept Content Negotiation](#)

<sup>15</sup>[Deploying WebP via Accept Content Negotiation](#)

<sup>16</sup>[WURFL Device Description Repository](#)

<sup>17</sup>[DeviceAtlas Device data solution](#)

<sup>18</sup>[How the Browser Pre-loader Makes Pages Load Faster](#)

<sup>19</sup>[Chrome's preloader delivers a ~20% speed improvement!](#)

## Preloading explained

When parsing source markup while a browser is blocked on a script (downloading and executing), a second lightweight lookahead preload scanner starts parsing rest of the source markup for any other downloadable resources such as stylesheets, scripts, images, video etc for speculative preloading. A preloader will start downloading these resources in background so that on next resource main parser thread is not blocking for download. Some preparer also runs the HTML tree construction algorithm speculatively.

## Issues around preloading

Resources preloading creates interesting challenges for many responsive image solutions. Two key challenges are,

1. Double requests per image
  - Once for placeholder or default version of image using preloader and then another better version of image using JavaScript DOM Manipulation.
2. Unable to take advantage of preloader
  - Preload scanner cannot speculatively execute JavaScript hence resources managed by JavaScript will miss the benefit of preloader.

## Lazyload and Postpone

Currently there is limited support for lazy (and postpone) preloading. According to [W3C Resources Priorities Proposal](#) two attributes `lazyload` and `postpone` can be used set priority order in which the user agent should or will download the resource<sup>20</sup>. Both attributes are boolean.

In following example, user-agent loads image only after all higher priority resources or all default priority resources (without `lazyload` attribute).

```

```

In following example, User Agent must not start downloading image until either image bounding box is already in viewport or about to come in viewport due to panning or scrolling, ~~~.html ~~~

---

<sup>20</sup>[lazyload and postpone attributes](#)

## Art Direction

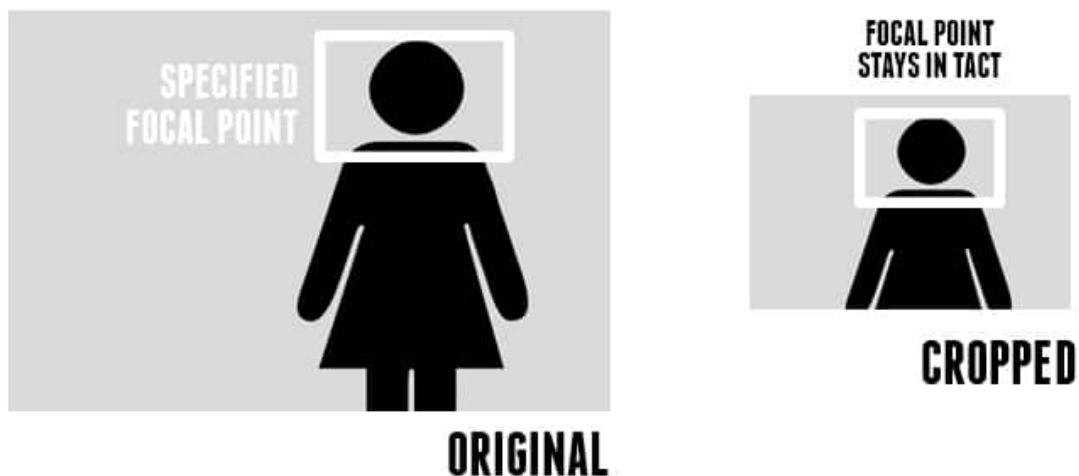
In terms of responsive images, art direction means creative control of individual images. Typically art direction requires different aspect ratios or focal points at different responsive breakpoints<sup>21</sup>. Art direction also comes in play when orientation of device changes (landscape to portrait or vice-versa).

### Importance of Art Direction

How much weightage one should give to art direction? Importance of art direction depends on the actual application. For instance, Art direction for a news website is quite important. Nonetheless implementing art direction for news website will be challenging because depending on story images can be variable in size and subject(point of interest). On the other hand art direction can be a lower priority for an e-commerce website where images are quite consistent in terms of size and subject.

### Automation and Workflow

Is it possible to automate the art direction? To some extent yes, mainly by cropping and zooming an image around a focal point<sup>22</sup>. But this may not be perfect as some time real focal point may be completely lost.



**Figure 1:** Responsive Images, Art Direction and Focal Point. Image Credits Focal Point Framework.

<sup>21</sup>A breakpoint is one of a series of CSS Media Queries, which can update the styles of a page based on matching of media features

<sup>22</sup>[Focal Point Framework](#)

Art direction can be part of image management workflow. Every-time a new image comes in, focal point or boundary is marked at very beginning which helps to automate the cropping and zooming flawlessly.

## Markup Issues

Most of responsive image solutions rely on special markups (placeholder elements). These markups are neither strictly semantic nor valid. Plus they are verbose and hard to debug. These markups are manipulated at run-time to give `img` element with responsive image. Again these markups cannot be validate against W3C markup service.

Most of special markup is used to avoid browser preloading and hence to avoid double download per image. Once we have more influence on preloading using priority attributes we may not need these special markups.

## Caching issues

As we are talking about responsive images in context of web-based applications, it is quite critical to understand the impact of any kind of application or CDN caching on your responsive image solution (and vice-versa).

If your responsive image solution is relying on a server-side component then any full page caching, baked deployment or any type of template caching will need to be redefined.

Caching issues are one major reason you may want to consider a client-side component. On side note, client-side solutions dependent on cookies are prone to different kind of caching related issues because image URLs are not unique (although cookies are different).

## Integration, Maintenance and Migration

Some key questions must be asked about integration, maintenance and migration before settling on a responsive image solution.

- How we can integrate legacy content or data with a responsive image solution?
  - Ideally any responsive image solution should be backward compatible. This basically applies to markup based solutions.
- How we can integrated responsive image solution with existing content delivery network?

- Many responsive images solutions expect cookies and query string forwarding from CDN service to origin.
- How implemented solution will be maintained?
  - Ongoing maintenance requires regularly updating solution to support new device, new breakpoints and new environmental conditions.
- How easy it will be to migrate from one responsive image solution to another with minimum effort?
  - This basically demands that what ever responsive image solution we choose, actual implementation should be decoupled from data and content so that migration path is not blocked.

## Anatomy of a responsive image solution

Implementing a responsive image solution is typically a 2-step process.

### Step-1 Reference responsive images in HTML/CSS

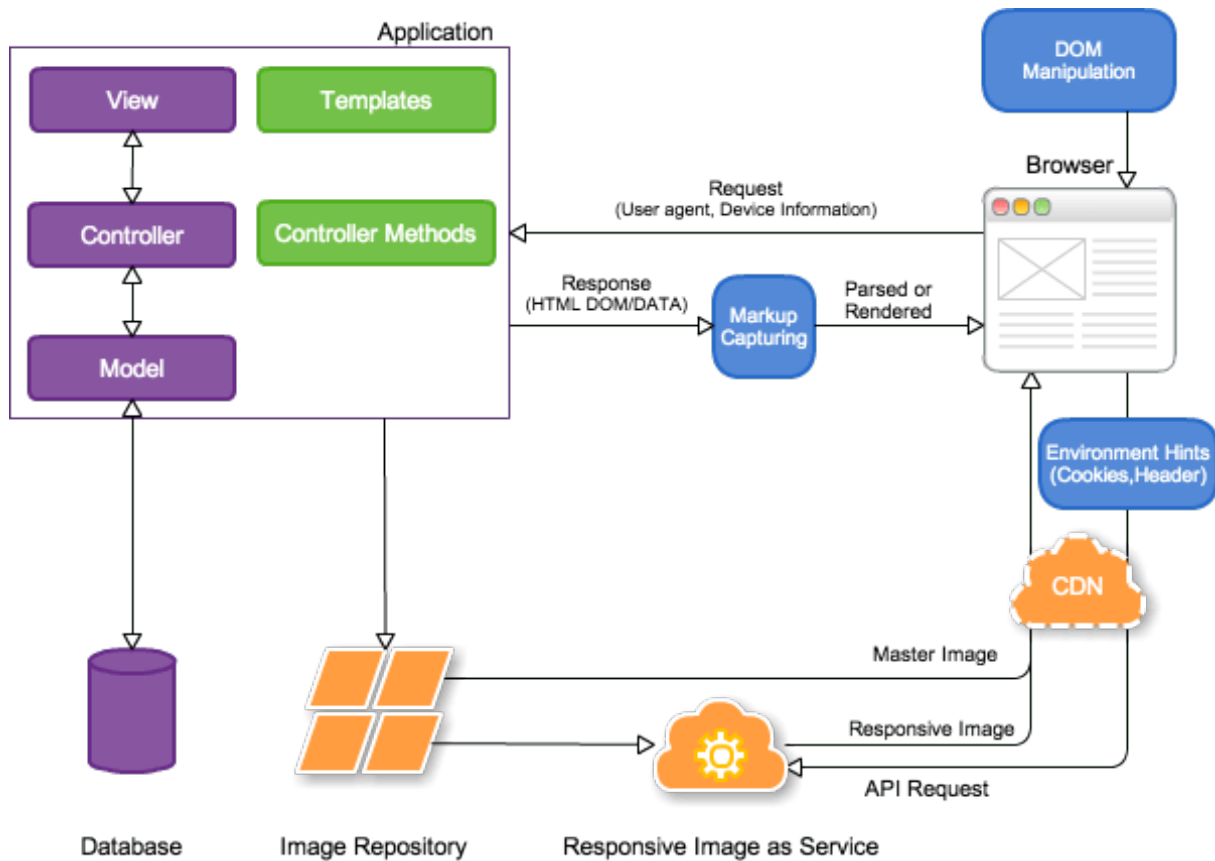
Step-2: Generating and Serving responsive images

Following figure illustrates possible touch points for a responsive image solution. This assumes you are running a web-based application based on MVC or a similar design pattern.

Box labeled as *green* and *blue* are components which we can use to reference or source responsive images in a HTML page (Step-1). **Client-side** components are *blue* and server-side components are *green*.

Box labeled as *orange* are components (image-side) which we can use to generate and serve these responsive images (Step-2).





**Figure 2:** Anatomy of a responsive image solution. Server-side, Client-side and Image-side components.

### Step-1: Reference responsive images in HTML/CSS

#### Reference responsive images in CSS

Referencing responsive images in CSS is quite straightforward (at least with CSS3). Using appropriate media queries we can always change image referenced in CSS to handle environmental conditions.

```

/* Responsive images for background */
body {
  background: url(bg-small.jpg);
}
@media (min-width: 400px) {
  body {
    background: url(bg-medium.jpg);
  }
}
@media (min-width: 800px) {

```

```
body {
  background: url(bg-large.jpg);
}
@media (min-width: 1000px) {
  body {
    background: url(bg-extralarge.jpg);
  }
}
```

Moreover, a CSS4 proposal for `image-set()` enables developers to specify multiple resolutions of an image and serve responsive images accordingly<sup>23</sup>. Although experimental feature, it has been already implemented for background images in Safari 6 and Chrome 21<sup>24</sup>.

```
#selector {
  background-image: url(no-image-set.png);
  background-image: -webkit-image-set( url(image.jpg) 1x, url(image-hires.
  jpg) 2x );
  /* other prefixes for -moz, -o and -ms ... */
}
```

## Reference responsive images in HTML

This requires inserting or referencing appropriate responsive images in a HTML source or document based on environmental conditions. Again there are two approaches to accomplish this.

**Approach-1: Special markup**

**Approach-2: Image src override**

Both **Approach-1** and **Approach-2** require combination of **client-side** and server-side components.

**Approach-1: Special markup** This approach requires some special markup in HTML source. Then it relies on client-side run-time processing of either new DOM elements like `picture`<sup>25</sup> or existing DOM elements with help of new `data-*`<sup>26</sup>, `src-N`<sup>27</sup> `srcset`<sup>28</sup> attributes.

1. Add a set of responsive images in HTML source along with environmental condition specific to each responsive image. Normally images are added
  - either in `div`, `span` or `img` element with `data-*` attribute,

---

<sup>23</sup>[Image Set Notation for CSS4](#)

<sup>24</sup>[Safari, Chrome Ship Proposed High-Resolution Image Solution](#)

<sup>25</sup>[The picture element and srcset](#)

<sup>26</sup>[Embedding custom non-visible data with the data-\\* attributes](#)

<sup>27</sup>[Proposal for Resplmg Syntax using src-N](#)

<sup>28</sup>[The srcset attribute](#)

- or in `picture` element.
2. Add a fallback or default image in `img` element. If you are using a Javascript solution on client side, then place `img` element in `noscript` element. Some solution put a low quality image as default (LQIP<sup>2930</sup> or mobile-first<sup>31</sup>)
  3. Then -
    - either capture the source markup before it has a chance to be parsed by the browser and process the element to produce final markup according to the media queries or environmental conditions<sup>323334</sup>,
    - or manipulate the DOM after it is parsed to get right responsive image according to the media queries or environmental conditions<sup>35363738</sup>.

Elements	Used Attributes
<code>picture</code>	<code>srcset, data-*</code>
<code>div, span, img</code>	<code>data-*</code>
<code>img</code>	<code>src-N, srcset</code>

**Figure 3:** DOM Elements and Attributes used or proposed for Responsive Images.

**Approach-2: Image `src` override** Unlike previous method this approach does not require any special markup. Standard `img` element is used and on run-time depending on environmental conditions `src` attribute is overridden.

Run-time `src` value override or rewrite can take place both server-side as well as client side. Of-course server-side override depends on device detection and environment hints via cookies. In addition, override can also happen via source capturing (described in next section).

<sup>29</sup>[LQIP - Low Quality Image Placeholders](#)

<sup>30</sup>[Imadaem — JS Daemon for Responsive Images](#)

<sup>31</sup>[Mobile-First Images that Scale Responsively & Responsibly](#)

<sup>32</sup>[Capturing – Improving Performance of the Adaptive Web](#)

<sup>33</sup>[Capturing and Mobify.js](#)

<sup>34</sup>[Automate Your Responsive Images With Mobify.js](#)

<sup>35</sup>[Estimates network connection speed prior to requesting an image](#)

<sup>36</sup>[BBC-News / Imager.js](#)

<sup>37</sup>[Picturefill](#)

<sup>38</sup>[responsive-images.js](#)

**Image src as REST URL** Typically values of `src` attribute follow REST URL pattern where parameters reflect environmental conditions (either directly or by proxy). For instance image quality is just proxy for environment conditions like device type, network connectivity, etc. In the `src` URL you can use Path parameters, Query parameters or both.

Because each REST URL is a unique endpoint for a given set of environmental conditions, there are no cacheability issues when used with a content delivery network (CDN).

Here is one example to serve responsive version of a master image `bedroom.tif` (Path parameter) using quality and width as Query parameter.

```



```

Following example highlight a more sophisticated use of REST `src` URL pattern. As you can see below this approach has ability to handle art direction flawlessly.

```




```

## Client-side components

**Markup Capturing** Markup Capturing on client-side is a unique way to serve not just responsive images but also complete responsive version of a website.

In this approach source markup is captured by a script way before source markup is parsed by the browser. Then source markup is modified to inject responsive images depending on environmental conditions detected by media queries. Key here is the delayed preloading. Once finished, actual parsing starts and preloading will commence.

Currently [Mobify.js](#) provides excellent support for Markup Capturing<sup>394041</sup>. Using Capturing API<sup>42</sup> one

<sup>39</sup>[Capturing – Improving Performance of the Adaptive Web](#)

<sup>40</sup>[Capturing and Mobify.js](#)

<sup>41</sup>[Automate Your Responsive Images With Mobify.js](#)

<sup>42</sup>[Automate Your Responsive Images With Mobify.js](#)

can

Capture and modify the DOM before any resources have loaded by delay the lookahead preparer.

Please not that Capturing technique delay preloading, but it does not prevent parallel downloads.

### Image src override using Mobify.js

Using following Mobify.js script we can override or rewrite image src for responsive images.

```
<script>!function(a,b,c,d,e){function g(a,c,d,e){var f=b.
  getElementsByTagName("script")[0];a.src=e,a.id=c,a.setAttribute("class"
  ,d),f.parentNode.insertBefore(a,f)}a.Mobify={points:[+new Date]};var f
  =/(;( )|#|&|^)mobify=(\d)/.exec(location.hash+"; "+b.cookie);if(f&&f
  [3]){if(!+f[3])return}else if(!c())return;b.write('<plaintext style="
  display:none">'),setTimeout(function(){var c=a.Mobify=a.Mobify||{};c.
  capturing=!0;var f=b.createElement("script"),h="mobify",i=function(){
  var c=new Date;c.setTime(c.getTime()+3e5),b.cookie="mobify=0; expires="
  +c.toGMTString()+"; path=/" ,a.location=a.location.href};f.onload=
  function(){if(e)if("string"==typeof e){var c=b.createElement("script");
  c.onerror=i,g(c,"main-executable",h,mainUrl)}else a.Mobify.
  mainExecutable=e.toString(),e(),f.onerror=i,g(f,"mobify-js",h,d)}}(
  window,document,function(){a=/webkit|(firefox)[\\s](\d+)|(opera)[\\s\
  ]*version[\\s](\d+)|(trident)[\\s](\d+)|3ds/i.exec(navigator.
  userAgent);return!a||a[1]&&4>a[2]||a[3]&&11>a[4]||a[5]&&6>a
  [6]?!1:!0},

  // path to mobify.js
  "///cdn.mobify.com/mobifyjs/build/mobify-2.0.5.min.js",

  // calls to APIs go here
  function() {
    var capturing = window.Mobify && window.Mobify.capturing || false;

    if (capturing) {
      Mobify.Capture.init(function(capture){
        var capturedDoc = capture.capturedDoc;
        // call Capturing API, get all images from source
        var images = capturedDoc.querySelectorAll("img, picture");
        // call Image API, resize all images and override/rewrite `src`
        Mobify.ResizeImages.resize(images, {
          // Pass additional option like format, device-pixel-ratio, quality
          , cache-control
          cacheHours: "240", // Cache in CDN for 10 days
          quality: "95", // Quality control for JPEG/WebP
        });

        // Render source DOM to document
        capture.renderCapturedDoc();
      });
    }
  }
};
```

```
}
});</script>
```

Above script is using two Mobify.js APIs - Capturing API and Image API. Using above script following markup:

```


```

is dynamically modified into this:

```


```

`Mobify.ResizeImages.resize` is using Mobify image manipulation backend `ir0.mobify.com`. `Mobify.ResizeImages.resize` rewrites `src` according to following convention,

```
http://ir0.mobify.com/<format><quality>/<maximum width>/<maximum height>/<
url>
http://ir0.mobify.com/c<hours>/<format><quality>/<maximum width>/<maximum
height>/<url>
```

You can always access Mobify image manipulation directly using this URL convention.

### Picture src override using Mobify.js

In addition, you can also use Capturing with an alternate and simplified version of `picture` markup. With this alternate `picture` markup instead of specifying a different image for each breakpoint, in source you just specify one (or two for art direction).

```
<picture data-src="http://example.com/extralarge.jpg">
  <source src="http://example.com/alternate_art.png" media="(min-width:
    320px)" data-width="320">
  <source media="(min-width: 800px)" data-width="400">
  <source media="(min-width: 1000px)" data-width="500">
  
</picture>
```

is modified into this on the fly. ~~~ .html ~~~

Please note three changes in above source.

1. Capturing added missing `src` attribute on `source` element using `data-src` (`picture` element).
2. Capturing rewrote `src` for `source` element with help of `data-width`.
3. Capturing rewrote default `img` element by changing `src` attribute to `data-orig-src` to avoid preload.

After Capturing changes the markup, the `picture` polyfill will run and select the appropriate image based on environmental conditions (DOM Manipulation).

```
<picture data-src="http://example.com/extralarge.jpg">
  <source src="//ir0.mobify.com/webp/320/1418/http://example.com/
    alternate_art.png" media="(min-width: 320px)" data-width="320">
  <source media="(min-width: 800px)" data-width="400" src="//ir0.mobify.
    com/webp/400/1418/http://example.com/extralarge.jpg">
  <source media="(min-width: 1000px)" data-width="500" src="//ir0.mobify
    .com/webp/500/1418/http://example.com/extralarge.jpg">
  
</picture>
```

**DOM Manipulation** Currently DOM Manipulation using JavaScript `onload` remains most popular way to reference responsive images in an HTML page.

In addition whenever a browser is resized or device orientation changes DOM Manipulation will update `img` attribute with right image.

#### Low Quality Image Placeholders

Low Quality Image Placeholders (LQIP)<sup>434445</sup> or Mobile First<sup>4647</sup> approach is used by Apple, Twitter, Facebook, Etsy and many others.

This happens as 2 parts process,

1. initially a low quality image or mobile version of image is downloaded on page load, then
2. depending on environmental conditions using `onload` event low quality image is replaced or swapped by a high quality image.

For swapping, high quality image can be loaded using a hidden IMG tag, once download is complete images are swapped. This will prevent the low quality image from disappearing before the full quality image is fully downloaded.

As you can see this approach requires double download of same image but if use intelligently it can delivery great user experience. For instance, if device is mobile or if device network connection is slow, then we can disable part 2 of the process.

---

<sup>43</sup>[LQIP - Low Quality Image Placeholders](#)

<sup>44</sup>[Imadaem — JS Daemon for Responsive Images](#)

<sup>45</sup>[Slimmage - sane & simple responsive images](#)

<sup>46</sup>[Mobile-First Images that Scale Responsively & Responsibly](#)

<sup>47</sup>[Mobile first image loading with bandwidth detectionn](#)

**Environment Hints** Client-side environment hints are quite important for scaling the images on-the-fly. Environment hints can be passed as cookies to a Responsive Image as Service (discussed later) to dynamically scale images based on viewport, device-pixel-ratio, breakpoints , etc.

This JavaScript snippet sets viewport, breakpoint and device-pixel-ratio in cookie. ~~~ .html

```
#### Server-side components

##### Templates
Templates normally render data passed by MVC controller. Templates are
pretty much like plain HTML markup but template variables, template
inheritance and template tags make them more re-usable and modular for
our purpose.

Templates are good place to implement special markup based solutions.
Templates can also be used for server-side image `src` override based
on environment hints received from controller.

##### Controller
MVC Controller can be mainly used for,

* detecting the environment conditions such as (device class, user-agent,
screen size, pixel ratio etc) and pass this information to templates,
and
* in some cases to modify or format the data used for rendering of `img`
element in the templates (this can easily accomplished in template)

Currently there is very limited support to detect environment conditions
on server side. This can be accomplished using environment hints via
cookies.

## Step-2: Generating and Serving responsive images
This requires to generate responsive images either on-demand or in-advance
typically using a master image, and serve them.

### On-demand generation
On-demand responsive image generation requires an automated service
approach - Responsive Image as Service.

### In-advance creation
In-advance responsive image creation can be done either automatically in
batch for predefined environmental conditions or manually one-by-one.
This approach is not recommended due to maintenance overhead created by
change in breakpoints like new device viewports, resolution , etc. In
addition, a manual process will not scale well.

With this approach responsive images can follow a file path and file name
convention to emulate a REST like behaviour. For instance if master
image is `static/images/mymaster.jpg` then corresponding responsive
```



```

    image can be

1. `static/images/type/mymaster.jpg`
2. `static/images/mymaster-type.jpg`
3. `static/images/type/pr/mymaster-type.jpg`
4. `static/images/type/mymaster-type-pr.jpg`

Where `type` can be either `low`, `medium`, `high` (representing quality)
or `mobile`, `tablet` `desktop` (representing device). Similarly `pr`
represents pixel ratio and valid options for `pr` are `1`, `2`, `1.3` ,
etc., (depends on device, retina display or not).

### Image-side components

#### Image Repository
Source or master images used to generate responsive images are normally
stored in a repository such as,

1. Digital Asset Management (DAM), typically part of WCM or ECommerce
system.
2. Blob storage systems like Amazon S3, Rackspace CloudFiles or Azure Blob
Storage.

#### Responsive Image as Service
A [Responsive Image as Service (RIaS)](http://abhishek-tiwari.com/post/responsive-image-as-service-rias/) offers on-the-fly responsive image
generation and delivery using REST APIs or REST like image paths.

**Cookies**

Some RIaS requires environment hint via cookies. For instance, Sencha.io
Src[^35] uses a JavaScript measurement library to detect the browser's
screen dimensions and set them in a cookie. Once cookie is set,
subsequent API calls to Sencha.io Src can use values set in cookie.

Following `img` markup will make Sencha.io Src API call using `sw` (`
screen.width`) property which is set in a cookie scoped to the src.
sencha.io domain.

~~~ .html
<img
  src='http://src.sencha.io/sw-16/http://sencha.com/files/u.jpg'
  alt='Client-measurement, reduced'
/>

```

Similarly Adaptive Images<sup>48</sup> completely relies on environment hints via cookies. On client-side following JavaScript code will set cookies required by Adaptive Images server-side.

```
<script>document.cookie='resolution='+Math.max(screen.width,screen.height)
```

<sup>48</sup>Adaptive Images

```
+("devicePixelRatio" in window ? ","+devicePixelRatio : ",1")+'; path=/'></script>
```

When an image `/static/example.jpg` is requested to Adaptive Images service, server side will check for cookies describing viewport and device-pixel-ratio. Using these values server will generate responsive image and return.

### Header

To serve WebP using `Accept` header received either from downstream CDN server or directly from user agent, RIaS sever must be configured properly<sup>49</sup>.

**Content Delivery Network** Although optional, a content delivery network can be used to speed up the delivery by caching the responsive images.

Some CDN can perform WebP content negotiations using `Accept` header, but this depends on fetching of WebP version from proxy RIaS server<sup>50</sup>.

## Worth Watching

### Responsive image container

Proposed Responsive image container will store image data in layered format<sup>5152</sup>.

- lower layer will represent the lowest resolution image, and layers above it will represent a specific resolution and contain required data diff.
- in addition lower layer will contain focal point and using layers above it one can construct the art-direction.

Container will support multiple image formats using decoding of master file format (ISO base media file format or something similar).

In addition, it has been demonstrated that Progressive JPEG can be used to delivery responsive images when resolution switching is required<sup>53</sup>.

---

<sup>49</sup>[Deploying WebP via Accept Content Negotiation](#)

<sup>50</sup>[Deploying WebP via Accept Content Negotiation](#)

<sup>51</sup>[Responsive Image Container](#)

<sup>52</sup>[Responsive Image Container Prototype](#)

<sup>53</sup>[Responsive image format: Progressive JPEG for the resolution-switching](#)

## Content negotiation headers

Server-side content negotiation is one really promising area. One example is [Accept](#) header for WebP which is already supported by selected browsers (Chrome and Opera) and CDN services (Akamai, MaxCDN).

There are few more header proposals in progress. These headers will require implementation from both browser vendors, CDN and proxy services.

## HTTP client Hints

Like [Accept](#) header for WebP, proposed HTTP client hints<sup>54</sup> allows client to advertise its device pixel ratio (DPR) via [CH-DPR](#) header, and the resource display width via [CH-RW](#) (in DIPs) of the requested resource.

```
GET /img.jpg HTTP/1.1
User-Agent: Awesome Browser
Accept: image/webp, image/jpg
CH-DPR: 2.0
CH-RW: 160
```

Again servers, proxies and CDN will need to use these headers to serve appropriate assets.

## Responsive image protocol

Another interesting proposal to implement responsive image using a new HTTP header [Image-Resolution-Patch](#)<sup>55</sup>. Basic idea is to allow user agent to download a low-resolution version first then allow to download and apply high-resolution patch sequentially depending on environmental conditions.

## Final words

In this article I discussed anatomy of responsive image solutions. Let's face it — implementing a responsive image solution is hard and problematic<sup>56</sup>. Although I am quite convinced that new [picture](#) element is going to improve the situation but by no mean it will solve all our problems<sup>5758</sup>. I think

---

<sup>54</sup>[HTTP Client-Hints Draft Proposal](#)

<sup>55</sup>[Responsive Image Protocol proposal](#)

<sup>56</sup>[Responsive images: what's the problem, and how do we fix it?](#)

<sup>57</sup>[Responsive Image Protocol proposal](#)

<sup>58</sup>[Why Responsive Images Is So Hard](#)

browser vendors can make preload (and prefetch) more smarter by implementing resource priorities . Also content negations headers are definitely game changer as they are future proof and backward compatible. Last but not least success of any responsive image solution will require a mature RIaS because batch generation of responsive images for all predefined environmental conditions is not realistic.