# **Udacity’s Front-End Web Development Nanodegree**

## HTML and CSS Learning

### Lesson 14 – Common Responsive Patterns

The ***column drop layout*** is a common responsive design pattern. Rows are stacked on small screens, but go from 3 rows and 1 column to 2 columns and 2 rows to, finally, 1 row and 3 columns:

Small Medium Large

11111111 1111222222 111222333

22222222 3333333333

33333333

The container is set to display: flex with flex-wrap: wrap and each box in the container is given a width of 100%. Breakpoints are set with media queries as we go from small to medium to large screens. At some point, as the screen expands, the container becomes centered with margins on either side (using margin-left: auto; margin-right: auto;)

The ***mostly fluid layout*** is another common responsive design pattern. It’s similar to, but more complex than the column drop layout.

Small Medium Large

11111111 11111111 11112222

22222222 22223333 33344555

33333333 44444444

44444444

The third layout presented in this lesson is the ***layout shifter layout***. This is the first pattern that makes use of the *order* attribute.

Small Medium Large

11111111 11112222 44222211

22222222 11113333 44333311

33333333 44444444

order: 1

order: -1

44444444

The layout begins in the same the others do, but this time we have 3 boxes, one of which has two nested boxes. So what looks

### Lesson 15 – Optimizations

This lesson has a lot to do with image optimization and using responsive images. Those are really two different topics, but are treated together here.

I took Udacity’s Responsive Images course to fill in some of the missing details from this lesson. Here’s some of what I learned.

**PROBLEMS INVOLVING IMAGES**

* **PROBLEM**: How do I find an image’s file size, format, and dimensions?
* **SOLUTION**: Chrome Dev Tools’ Network tab is a useful tool for inspecting image size and format. The Elements or Console tab will reveal an image’s resolution.
  + In the console, after you click on an element, you can access that element with the *$0* syntax and then access element properties like *naturalWidth*.
* **GOTCHYA**: Less pixels x Better compression = Less bytes. This is somewhat counter intuitive. You might be tempted to use a larger image with higher resolution on a small screen that scales as the screen size increases. Don’t do that.
* **PROBLEM**: I set my image width to 100% and it scales well, but on large screens, images begin to appear pixelated.
* **SOLUTION**: Set max-width: 100%. This will prevent an image from scaling any larger than 100% of its natural width.
* **PROBLEM**: How do you combine a percentage width for an image with a fixed margin?
* **SOLUTION**: The answer is *calc*. E.g in the case where we want to put 2 images side by side and give them each 50% of the width minus some margin, we could use the code below in combination with the *img:last-of-type* selector and *margin-right: 0*; to make sure that the last image is not getting that margin.

img {

 max-width: 426px;

 margin-right: 10px;

 width: calc((100% - 10px)/2);

}

Img:last-of-type {

Margin-right: 0;

}

* **PROBLEM**: I want my image to vertically fill the viewport, but my image’s height is constrained by the height of its parent element, so height: 100% isn’t working.
* **SOLUTION**: *vh* and *vw* which stand for viewport height and viewport width are units that are relative to the viewport size, not the parent element’s height. Setting the image height to 100 vh gets around that problem.
* **PROBLEM**: I don’t know what image is going to be served, but I need whatever it is to fill the viewport, regardless of whether it’s in portrait or landscape format.
* **SOLUTION**: The *vmin* unit represents 1% of the height OR width, whichever is smaller. The *vmax* unit represents 1% of the height OR width, whichever is larger. So, setting both height and width to *100vmax* will fill the viewport, regardless of the image dimensions. Conversely, setting height and width to say 10vmin will show the image at 10% of the viewport width or 10% of the viewport height, depending on which length is smaller.
* **PROBLEM**: How can I quickly determine whether I can compress my images any further?
* **SOLUTION**: Use Google’s Page Speed Insights (PSI) to optimize images.
* **PROBLEM**: I have 20 images and for my design I need each image at 3 different size and 2 different pixel densities. How can I automate that?
* **SOLUTION**: Grunt and the Grunt plugin responsive\_images (which uses ImageMagick) can automate bulk editing image sizes, image file sizes, etc. ImageMagick was tricky to install on Ubuntu, and Grunt has a bit of a learning curve. Here’s some background:
  + <https://www.smashingmagazine.com/2015/06/efficient-image-resizing-with-imagemagick/>
  + <https://addyosmani.com/blog/generate-multi-resolution-images-for-srcset-with-grunt/>
* **PROBLEM:** My site is loading slowly. What can I do?
* **SOLUTION:** Cut down on latency and bandwidth use.*Latency* (the time from the source sending a packet to the destination receiving it)is often a bigger issue than is *bandwidth*. For now, the point of diminishing returns for greater bandwidth in terms of load time is around 6 Mb/s, but reducing latency continues to provide faster load times, at least down to 20ms of latency. See: <https://hpbn.co/primer-on-latency-and-bandwidth/>. That said, both latency and bandwidth are important factors to consider when optimizing your site, especially considering the low bandwidth available on some mobile devices.
  + For now, to cut down on latency and bandwidth use, use a combination of techniques like *spriting* - using CSS or responsive sprites to reduce HTTP requests; *file* *concatenation and minification*; and *sharding* - using multiple domains to serve files in order to get around the limit of 2 HTTP requests at a time in older browsers. *HTTP 2* will make some of these and many of our other current hacks unnecessary. Sharding is already largely obsolete, because browser vendors now allow more than 2 HTTP requests at a time. HTTP2 allows multiplexing - delivering many files simultaneously. It has many other benefits, but that’s an oft cited one. HTTP 2 was standardized in 2015 and is quickly gaining market share, although it’s only used with about ¼ of domains as of mid-2018.
* **PROBLEM**: I need to scale a CSS background image as large as possible without either cropping or stretching the image.
* **SOLUTION**: Set the background-size property to *contain*.
* **PROBLEM**: I need to scale a CSS background image as large as possible without stretching the image, but still keep it within the parent element.
* **SOLUTION**: Set the background-size property to *cover*. If the proportions of the image differ from the element, the image is cropped either vertically or horizontally so that no empty space remains.
* **PROBLEM**: I need to use a lot of icon images in a project and I need them to scale well. How can I can keep latency and bandwidth usage down?
* **SOLUTION**: Trying using either Unicode characters or icon fonts.
  + There are literally thousands of unicode characters available, including many basic shapes. Using these characters has all the benefits of using markup and none of the downside of raster images.
  + Font Awesome and We Love Icon Fonts are great sources of icon fonts. Like unicode characters, these characters are scalable and stylable like markup. What’s really cool is that you can use svg icon fonts (<https://fontawesome.com/get-started/svg-with-js>)!
* **PROBLEM**: I have an svg image that I only need to use once. How can I avoid the expense of an additional HTTP request for this one image?
* **SOLUTION**: Inline the image in your HTML or CSS using a *data URI*. CSS Tricks has a good video on this here: <https://css-tricks.com/lodge/svg/09-svg-data-uris/> . The syntax in HTML looks like

<img src=’data:image/svg+xml;utf-8, [the svg code goes here]’ alt=””>

and in CSS looks like

.data-uri {background-image: url(‘data:image/svg+xml;utf-8, [code goes here]’);},

where the .data-uri class is then applied to some element where you want the image to appear. The svg code can also be converted to a base64 encoded string, which is always longer than the svg but is encoded, meaning you don’t have to worry about a stray line break in your CSS breaking the image.

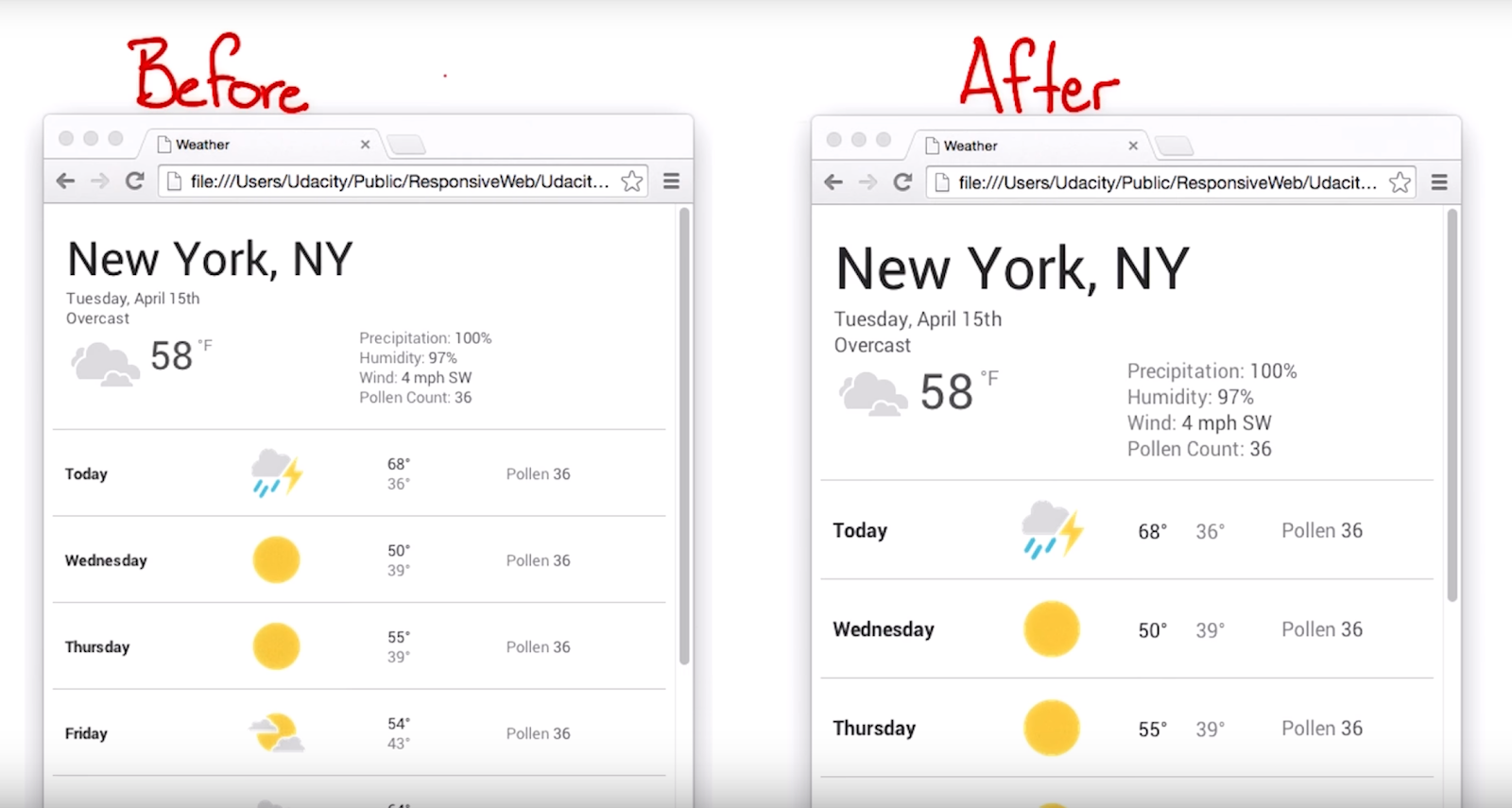
* **PROBLEM**: I need to offer a variety of images for a design that includes multiple breakpoints and that has to be optimized for multiple pixel densities. The math and code required to do that with just media queries is daunting. What can I do?
* **SOLUTION**: The HTML *picture* and *source* elements and the *srcset* and *sizes* attributes are what you need. In short, these elements and attributes tell the server which of several images to serve depending on factors including viewport size, pixel density, and breakpoints.  Those are pieces of information the serve responding to your HTTP request can’t know, unless you explicitly share that information. With that information in hand, the server can send you a small 2x image for retina display with a 500px x 200px screen size or a large 1x image for a 27” monitor. Background:
  + <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/picture>
  + <http://ericportis.com/posts/2014/srcset-sizes/>
  + <https://dev.opera.com/articles/responsive-images/>
  + The picture element wraps around the source and image child elements. The source element can specify media queries dictating when to use various images of various dimensions and even mime types (like webp, with a fall back to jpg). The picture tag’s last child must be an img tag. The picture tag is still being implemented and older browsers will require the *picturefill* polyfill to understand it.
  + Consider an example in which the designer needs the server to serve different sizes and pixel densities in different circumstances and also use some art direction to server entirely different images.
    - For browser windows with a width of 1280 CSS pixels and wider, a full-shot photo with a width of 50% of the viewport width will be used
    - For browser windows with a width of 640-1279 CSS pixels, a photo with a width of 60% of the viewport width will be used
    - For less wide browser windows, a photo with a width that is equal to the full viewport width will be used.
  + In each case, the browser picks the optional image from a selection of images with widths of 200px, 400px, 800px, 1200px, 1600px and 2000px, keeping in mind image width and screen DPI. Take a look at the (abbreviated) code below:

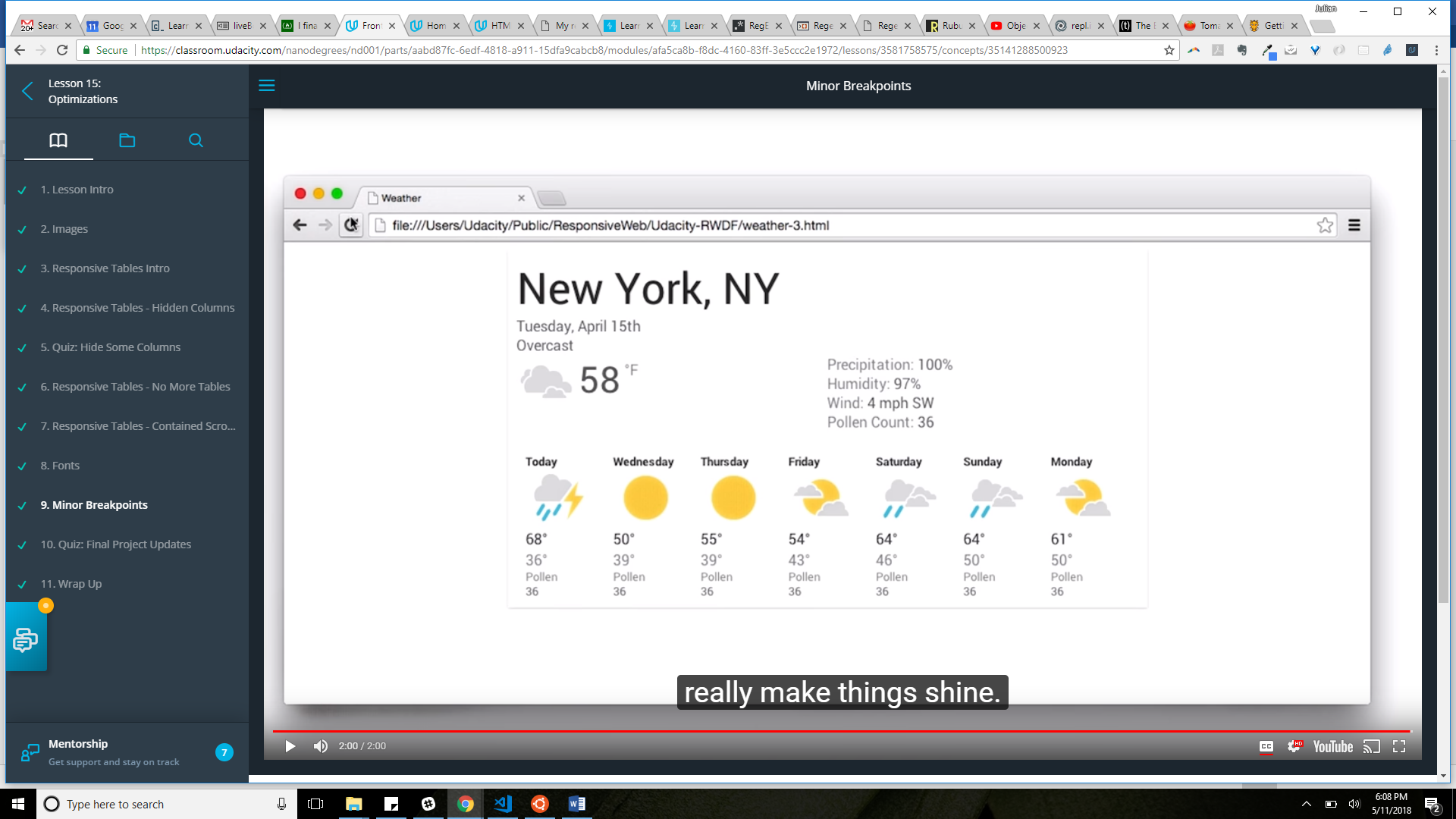
<picture>  
 <source

media="(min-width: 1280px)"  
 sizes="50vw"  
 srcset="opera-fullshot-200.jpg 200w,  
 opera-fullshot-400.jpg 400w,  
 opera-fullshot-800.jpg 800w,  
 ">  
 <img  
 src="opera-closeup-400.jpg" alt="The Oslo Opera House"  
 sizes="(min-width: 640px) 60vw, 100vw"  
 srcset="opera-closeup-200.jpg 200w,  
 opera-closeup-400.jpg 400w,  
 opera-closeup-800.jpg 800w,  
 ">  
</picture>

.

* + Research suggests the ideal “measure,” i.e. line-length, is between 45 and 90 characters per line. 65 cpl is the standard for the web.
  + Choose a base font size of at least 16px with a line height of 1.2 em.
* **PROBLEM**: My layout just looks a little off on larger screens. Any suggestions?
* **SOLUTION**: Use minor breakpoints to overcome problems associated with vertical layouts shifting to a more horizontal layout as screen size increase.
  + For example, make sure that your line-length (known as a “measure”) stays between 45 and 90 characters per line (cpl), ideally staying at around 65 cpl.
  + As the screen-size increases, increase font size as well.
  + Below is an example. On very small screens the “before” image looks great. As screen size increases, the design needs to adjust. Font size and images increase in size and some elements are arranged horizontally rather than vertically. On the larges screens the design gains margins and becomes centered. The vertical layout of the days of forecast is dropped and replaced by a horizontal layout of the days, with highs and lows, etc. shown below each day.





* **PROBLEM**: May tables look great on smaller screens, but start to look like garbage as the screen size increases.
* **SOLUTION**: There are 3 techniques that can help: hidden columns, no more tables, and contained tables.
  + “Hidden columns” refers to simply hiding certain columns of a table using display:none when on smaller screens. The “hidden” class be removed with a scrip and re-applied as the viewport changes.
  + “No more tables” involves forcing all table related tags to display:block, thereby causing each row and td to appear on its own line. The table header row must be positioned off screen using position:absolute. The td’s must be position:relative with padding-left:50%, this will give room for the row labels. The row labels can be inserted using td:before and then setting position:absolute; left: 6px; content: attr(data-th); That last bit is the most important. The data-th will be grabbed from the data-th attribute of the td in question, which you’ll need to include in your HTML.
  + In the “contained tables” technique you simply wrap your table in a div with width:100% and overflow-x:auto. This will create a scrollable table.

PROBLEM: I have some CSS background images that appear pixelated on some screens. What can I do?

SOLUTION: CSS background images have a new property called image-set. The image set property is added to the background-image property like this:

.selector {

background-image: url('../img/image-1x.jpg';

background-image: -webkit-image-set(url('../img/image-1x.jpg') 1x,

url('../img/image-2x.jpg') 2x);

}

The first appearance of background-image is a fallback when image-set is not supported.