Responsive Media in HTML5

Rapid growth in the use of mobile devices to access the Internet has forced designers to adapt to creating content that is easily viewable on a wide range of different devices. The key is to make images and videos responsive to the platform used, enabling them to be resized automatically when displayed on mobile devices or desktops.

This book is a fast-paced, hands-on guide that shows you how to apply some simple techniques to add images and video content to your site, which may be a simple, one-page portfolio, or a complex content management system. The book starts with showing you how to display images. You will learn about browser support and different platforms, and alternatives for high- and low-resolution images. Moving on, the book covers techniques to add responsive video content. You will get to know about adding images and videos and test the media using online sites and tools. The book also explains the use of plugins and responsive frameworks.

Who this book is written for
If you are a web designer with a good understanding of CSS, jQuery, and HTML, but new to creating responsive sites, then this book is for you. The prerequisite is a good understanding of CSS and HTML; the demos will suit those who have some prior knowledge of Less CSS, WordPress, or Bootstrap.

What you will learn from this book
- Add responsive images and video content to your site
- Understand some of the pitfalls of mixing responsive images and videos, and how to alter your development to ensure both co-operate together
- Incorporate responsive media in popular frameworks such as WordPress, Less CSS, and Bootstrap
- Test responsive sites to ensure content displays correctly on different iOS and Android devices
- Discover how to make allowances for mobile devices, and use tools to improve speed of access
- Utilize tools such as YSlow and other online tools to determine speed of the sites
- Employ preloaders to counter delays in loading

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Mixing Content'
- A synopsis of the book’s content
- More information on Responsive Media in HTML5

About the Author

Alex Libby has a background in IT support. He has been involved in supporting end users for the last 18 years in a variety of different environments and currently works as a technical analyst, supporting a medium-sized SharePoint estate for an international parts distributor based in the UK. Although he gets to play with different technologies in his day job, his first true love has always been the open source movement, in particular, experimenting with CSS/CSS3 and HTML5. So far, he has written several books for Packt Publishing, including one on HTML5 video and another on the jQuery UI. This is his eighth book.

I'd like to thank family and friends for their help and encouragement and the reviewers for providing lots of constructive comments—without them, I am sure I wouldn't have been able to produce this book!
Responsive Media in HTML5

Web design is responsive design. Responsive web design is web design done right.

– Andy Clarke, Stuff and Nonsense

Take a straw poll of one hundred users and it is likely you will find that a good proportion have viewed the Internet from a mobile device at some point in the past. Use of these devices to access the Internet has exploded massively over the last few years. This has highlighted the need to design sites that work well on a variety of different devices and platforms.

First coined as a term by Ethan Marcotte back in 2010, responsive web design is the art of designing such sites. A key element of responsive web design is the addition of images and videos. Throughout this book, we're going to take a look at the tips and tricks you need to get ahead in adding responsive media to your sites. We'll see how the lack of a common W3C standard means the field is wide open for different solutions and that catering for different platforms means understanding their limits and making suitable allowances for each type of device.

We’ll work through a number of practical examples, with both images and videos, and see the importance of testing both to ensure your content displays as expected on any device. We'll then make use of some of the tips and tricks in several real-world examples, using popular frameworks such as WordPress or Less CSS—you’ll see that adding responsive media isn't actually that complicated!

Question is though—are you ready to make a start?

What This Book Covers

Chapter 1, Working with Responsive Images, starts our journey with responsive media, where we take a look at working with responsive media. We'll cover the basics with a look at a wide variety of topics, including retina images, providing fallback support, different image formats to use, and catering to the differences between mobile and desktop platforms.

Chapter 2, Adding Responsive Video Content, continues on from where we left off in Chapter 1, Working with Responsive Images, with a look at adding responsive video to our sites. We'll cover the different formats available to us and the methods to be used to determine which one to use, how to determine the size of the video to use depending on viewport space, providing fallback support, and making allowances for different platforms.
Chapter 3, Mixing Content, helps us bring it all together with a look at some of the considerations or pitfalls of mixing responsive content and how we can reduce delays by using preloader to control when content is loaded and rendered on screen. We'll work through an example of mixing both responsive images and video on the same page, so we can see how it works in practice and consider what allowances may need to be made for different platforms.

Chapter 4, Testing Responsive Media, delves into the world of testing our creations to ensure they work properly. We'll see that there is no need for complicated tools as most of our work can be done in a browser; we will also cover some tips to troubleshoot slow performance issues and how we can make quick and easy changes to help improve speed.

Chapter 5, Using Frameworks, finishes up the book with a look at three real-world examples of using responsive media in popular frameworks; our examples include a look at WordPress, Less CSS, and Bootstrap. We'll take a look at some of the tips and tricks (and plugins) available for use in WordPress, adding responsive media to a Bootstrap-enabled webpage, and how we can use Less CSS to help better manage our CSS styles.
Load Times For 69% Of Responsive Design Mobile Sites Deemed "Unacceptable"…

A staggering fact, but absolutely true: a survey carried out by the mobile developer company Tribilis in April 2014 found that for 155 sites surveyed, only 21 percent loaded in four seconds or less on smartphones. Moreover, those that took longer had an average page weight of 1.7 MB.

Sobering thoughts, but ones that perfectly illustrate the pitfalls of mixing videos and images on pages in responsive design! We need to strike a balance between displaying the right media content that is not too large or too small while keeping our page load times small. This chapter works through some examples and details some of the pitfalls associated with mixing content, which can lead to heavy pages and long download times.

We’ll cover a number of topics in this chapter, which will include:

- Mixing video and image content on the same page
- Considering limitations of mixing content
- Using preloaders to reduce delays
- Adding lazy loading support to our pages
- Making allowances for mobile devices

Intrigued? Let’s make a start!
Mixing Content

Mixing video and image content on the same page

Throughout this book, we've seen some of the tips and tricks we need to use to make our media content responsive and worked through some examples of the styles we need to implement to make this happen.

However, it is unlikely that every responsive site will have either images or videos; it is more likely that there will be a mix of both on at least some sites! This can present its own issues, so to see what can happen, let's run through a simple demo.

From the code download, extract the code folder for this chapter and save it to your project folder; this contains a simple demo that shows the issues we face with mixing videos and images together in a responsive design. If we run the mix-responsive.html demo as it stands, it would appear to look okay from the start until we scroll down and view the video. The layout is not ideal, but there are a few changes we can make:

1. The banner image at the top of the browser window is too large. While this may be okay on a desktop, it will choke smartphones; it needs to be resized.
2. The embedded video isn't respecting the boundaries of its container, with the result that it is spilling out over the whole page, making the text difficult to view.
3. We can improve on the use of the banner. Once we've changed it to a thinner design, we can then add a media query to switch in a smaller version of this thinner design.

Fortunately, these are easy fixes to make. Let's see what is involved:

1. Save a copy of mix-responsive.html and mix-responsive.css as mix-responsiveV2.html and mix-responsiveV2.css respectively (in the same folder). Don't forget to change the link to the CSS file from within our HTML markup to point to the new file!
2. Look for the banner div on or around line 28 and remove the <img> tags, so the HTML markup is as shown in the following line of code:

   ```html
   <div id="banner"></div>
   ```

3. Further down, look for <video controls> on or around line 34; we need to encompass it within a wrapper <div>:

   ```html
   <div id="video-wrapper">
     <video controls>
       <source src="video/bigbuckbunny-480px.mp4" type="video/mp4">
       <source src="video/bigbuckbunny-480px.webm" type="video/webm">
     </video>
   </div>
   ```
4. Open up mix-responsiveV2.css and add the following styles:

```css
#video-wrapper video { max-width: 100%; }
#banner { background-image: url('../img/abstract-banner-large.jpg'); height: 15.31rem; width: 45.5rem; max-width: 100%; }

@media screen and (max-width: 30rem) {
  #banner { margin-top: 150px; background-image: url('../img/abstract-banner-medium.jpg'); height: 4.85rem; width: 45.5rem; max-width: 100%; }
}
```

5. Save the files. If we preview the results of our work in a browser, we can see a significant improvement in the appearance of our page.
Try resizing the page now; you should see that the video and image elements will resize perfectly, without any spillage. In a nutshell, our changes have been very simple. We've added a container to the video to which we've added the `max-width` style attribute and set this to 100%. We then switched out the bulky banner to a thinner version, setting this to retain the full width of its container when resized. It is switched to a smaller version once we go below a screen width of 30rem.

Before we move on and take a look at how we can make allowances for mobile devices when mixing content, I want to cover off a couple of tips that might help give you a little inspiration:

- We've embedded the video directly in the page. It works perfectly well, but if we wanted to give it a little extra sparkle, we could consider using an overlay. There are plenty available, but one good (responsive) example is FrescoJS, available at [http://www.frescojs.com/](http://www.frescojs.com/).

- There may be a need to resort to jQuery to provide the responsive functionality for an element if CSS3 is not supported. Older browsers such as IE8 or below might fall into this category. We can do this using something such as Embedly jQuery, available at [http://embedly.github.io/embedly-jquery/](http://embedly.github.io/embedly-jquery/).

- If you need to add responsive code for third-party embedded videos (such as YouTube), you can use the service at [http://embedresponsively.com/](http://embedresponsively.com/); simply enter the URL of any video or image to embed and click on Embed to get the code.

Okay! On we go! We've talked a lot about mixing content, but as we've seen from the demo, there are some aspects where there is room for improvement. The alterations we've made in our demo are just some of the pointers we can use to remove some of the pitfalls of mixing content, so let's take a look at a few in more detail.
Considering the pitfalls of mixing content

Responsive design has become hugely popular over the last few years, with visitors browsing content on the Internet using a variety of devices. This increases the pressure on developers to ensure that their sites work well on a range of different devices and platforms. A recent study by eMarketeer illustrates that four out of every five people will access content via a mobile device and that this figure is projected to increase over the next three years.

<table>
<thead>
<tr>
<th>Region</th>
<th>2014</th>
<th>2015</th>
<th>2016</th>
<th>2017</th>
</tr>
</thead>
<tbody>
<tr>
<td>Middle East and Africa</td>
<td>94.0</td>
<td>94.0</td>
<td>95.0</td>
<td>96.0</td>
</tr>
<tr>
<td>Asia Pacific</td>
<td>87.4</td>
<td>90.0</td>
<td>92.6</td>
<td>93.9</td>
</tr>
<tr>
<td>Central and Eastern Europe</td>
<td>72.4</td>
<td>80.6</td>
<td>86.6</td>
<td>91.7</td>
</tr>
<tr>
<td>North America</td>
<td>64.3</td>
<td>69.8</td>
<td>74.4</td>
<td>79.2</td>
</tr>
<tr>
<td>Western Europe</td>
<td>67.8</td>
<td>78.2</td>
<td>85.0</td>
<td>90.6</td>
</tr>
<tr>
<td>Latin America</td>
<td>58.5</td>
<td>65.0</td>
<td>70.1</td>
<td>75.2</td>
</tr>
<tr>
<td>Worldwide</td>
<td>79.1</td>
<td>83.6</td>
<td>87.3</td>
<td>90.1</td>
</tr>
</tbody>
</table>

The figures in the preceding table are percentage values taken from eMarketeer in December 2013.

So that we don’t fall into the trap of producing a responsive site that offers a poor experience, let’s take a look at some of the pitfalls we need to consider when mixing media sources in responsive design:

- The size and number of images on a page will directly affect how quickly your visitors can view the page. To reduce the impact, we can use several techniques:
  - Image compression is the first simple technique widely used. For a perfect example of the need to compress images, an article by Tribilis makes for interesting reading, available at http://blog.trilibis.com/trilibis-web-performance-survey-finds-69-percent-of-responsive-design-websites-fail-to-deliver-acceptable-load-times-on-mobile-devices-02759. It shows some significant gains when images are compressed!
  - The next important technique is the conversion of small images into data URIs or as part of an image sprite.
  - Another technique is the use of lazy loading to only show those images that are visible within the viewport.
• We need to be aware of the breakpoints in our design and alter our media queries to suit. What may have worked well for images may not work equally as well for videos, particularly if they are of different sizes.

• Scaling smaller images to a larger size will result in the loss of quality; it is recommended to start with large size images and dynamically resize them for smaller devices. Once we get to a particular breakpoint, then we can switch to using a smaller image by default to avoid images becoming illegible if they have not been sized for a particular breakpoint / viewport width.

• Scaling images can cause issues when they are reduced to a very small size. The dimensions are likely to mean that we can't view the image properly, its meaning will be lost, the file size won't be any smaller, and it may be better to simply not display it instead.

• Consider converting vector images to the SVG format. Conversion to standard image formats such as JPG, GIF, or PNG will likely lead to the same loss of quality as scaling small images to a larger size. SVG images scale well without any loss of quality.

• If your site uses video, then one pitfall that can trip you up is the format used. Although we only need to encode for MP4 or WebM support, not every device will use both. A careful analysis of web metrics should help us understand which operating systems are used to access the site and therefore help determine which format of video to use.

For an up-to-date check on format support, take a look at the articles available at http://www.jwplayer.com/html5.

• Do you use images for small elements such as buttons? If so, consider converting them to CSS3 equivalents, at least for the browsers that can support them. It will mean a reduction in requests to the server (we're not calling the images) and the CSS style sheet would have been cached by the browser, so the response will be quicker. There are plenty of examples available online; you can try Chris Coyier's Button creator at http://css-tricks.com/examples/ButtonMaker/ or CSS Shape Generator at http://html-generator.weebly.com/css-shape-generator.html.

• Specifying only the width of images may cause a doubling or tripling of the cycles that many browsers must process to layout the new resized page. While each of these cycles typically take less than a millisecond, they stack up, especially if there are multiple scalable elements on the page. Addressing the height in the same declaration can reduce this issue:

```css
img, video { max-width: 100%; height: auto; }
```
• If your site needs to display videos in 4:3 or 16:9 ratio format, then these may not resize properly. We can get around this using a wrapper in HTML that is styled to the proper dimensions, then stretch the video to fit the following container:

```html
<div class="wrapper-with-intrinsic-ratio">
  <div class="element-to-stretch"></div>
</div>
```

The CSS styling would look something like this:

```css
.wrapper-with-intrinsic-ration {
  position: relative; height: 0;
  padding-top: 25px; /* to allow for the player chrome */
  padding-bottom: 56.25% /* 9:16 = 0.5625 */
}
.element-to-stretch { position: absolute; top: 0; left: 0;
  width: 100%; height: 100%; background: teal; }
```

Phew! There are certainly plenty of places where we can be tripped up if we're not careful! Let's move on and take a look at a couple of tricks we can use to help manage media content in our pages, beginning with a look at preloading images.

## Using preloaders to reduce delays

Preloading content is not a new concept in the world of web design. The idea is to reduce the amount of time it takes to view the page in a browser, as some (or all) of the content has been fetched ahead of viewing the page and cached in the browser.

There are plenty of examples available using JavaScript (such as PreloadJS at [http://www.createjs.com/#!/PreloadJS](http://www.createjs.com/#!/PreloadJS)). But this adds an extra overhead to our pages that we can avoid, particularly in a responsive design.

Instead, we can use CSS to achieve the same effect, provided the styles and images are called at the right time. This way, they will be ready for us when we render the images on screen.

For the purposes of this demo, I've simplified the code to concentrate on the preloading process only.
Let's take a look at preloading in action with a simple demo:

1. From the code download that accompanies this book, extract a copy of preloading.html and then save it in our project folder.

2. Next, extract img01.png to img04.png from the code download. The images should go into a subfolder marked img in our project folder.

3. In a new file, add the following styles to a new CSS style sheet. Save this as preloading.css, in a css subfolder within our project folder:

```html
html { font-family: Verdana, Arial, Helvetica, sans-serif; }
body { margin: 60px auto 0; padding: 0; background-color: #606061; color: #ffffff; }
img { border: none; padding: 10px; max-width: 100%;}
p { font-size: 1em; margin: 0 0 1em 0; }
#container { position: absolute; top: 0; left: 0;
    background-color: #18191d; width: 100%; height: 40px; }
#container p { color: #ffffff; margin: 10px auto 0; text-align: center; width: 310px; }
#preload { margin-left: auto; margin-right: auto; width: 900px; }
body:before {
    content: url(../img/img01.png) no-repeat -9999px -9999px,
    url(../img/img02.png) no-repeat -9999px -9999px,
    url(../img/img03.png) no-repeat -9999px -9999px,
    url(../img/img04.png) no-repeat -9999px -9999px;
    display: none;
}
```

Before we preview our work, it's useful to take a quick look to see how this process works. The key to it is the class added to the body to retrieve the images before we use them later in our code:

```html
<body class="preload-images">
    <div id="container">
        ...
    </div>
    <div id="preload">
        <img src="img/img01.png" alt="Image 01" />
        ...
    </div>
</body>
```
If we preview the results of our work, we'll see the following four images appear:

The beauty about using this method is that we've removed the need to have to use yet another library, such as PreloadJS. Instead, we can now choose to call it if we decide to provide support for older browsers that don't support the `body:before` trick we used in our demo.

Let's change tack and look at the reverse side of this coin—lazy loading. No, this is not some form of getting up on a Sunday morning after a particularly good night out (terrible joke!), but a means to only load and display images at the point of viewing them. Intrigued? I will reveal all as part of our next exercise.

**Adding lazy loading support to our pages**

One of the primary aims of responsive design should be the consideration of any tactic we can use to reduce the overall loading time of our pages; this is particularly true for mobile devices.
So far, you've seen how we can preload content. In some instances, this may not be ideal; instead, we can do the opposite and not load content until we are about to view it:

This process is known as lazy loading; it effectively removes the focus of loading the images from the front (that is, on loading of the page) to the point of when we need to see the images. To illustrate the process, we're going to use the bLazy plugin by Bjørn Klinggard, available at http://dinbror.dk/blog/blazy/. The great thing about this library is that it is lightweight and written in pure JavaScript, so has a zero dependency count! Oh and did I forget— it's responsive too.

Let's take a quick look at a demo (based on the original by the author), which we could easily use as a basis for something more involved within any responsive design:

1. For the purposes of this demo, extract a copy of the lazy loading folder within the code download that accompanies this book. It contains the markup files, images, styling, and JavaScript needed for our demo.
2. Run the lazyloading.html file. Notice how only the first few images show? If we scroll down, we can see the green loading image appear with the next image appearing after a short delay.
The magic in this demo centers on this code excerpt from the demo:

```html
<div class="wrapper ratio_big-img loading">
  <img class="b-lazy" src=img/placeholder.png data-src="img/big-bear7.jpg" data-src-small="img/small-bear7.jpg" alt="Lazy load images image7" />
</div>
```

We kick off with a placeholder image. In this instance, we're using a 1 px square PNG file that could be easily converted to its data-URI equivalent with little overhead. We then use two `data-src` tags to flip between either the small or large versions of the image when the window is resized.

We can test to ensure that the pictures are indeed only being loaded when visible. In this instance, using Firefox's `Developer` toolbar, we can see each URL loaded when viewed in the `Console` tab, as indicated in the following screenshot:

![Console - http://localhost/lazyloading/lazyloading.html](image)

A small point to remember—in this instance, the preceding screenshot shows the use of a local webserver; the principle works the same whether the images are loaded from the filesystem or through a web server.

This demo is perfect as a basis to display a gallery effect on a site, particularly when viewed on a mobile device. With a bit of reconfiguration, we could even set it to display retina-based images; although that is something I will leave for you as a challenge! Let's move on and cover a key topic when designing responsively: the need to make allowances for mobile devices.
Mixing Content

Making allowances for mobile devices

Adding visual content to our pages will begin to bring the site alive, but we should bear in mind the increase in latency times. Latency is a major killer on mobile devices; if we take a 1 MB page with 80 plus requests, for example, loading the page can take over 4 seconds!

Using JavaScript is expensive in terms of resources. It takes time and can block the rendering of pages. A much better alternative is to only load the resource when it is needed. On desktops, this is less of an issue, but becomes crucial for mobile devices. Take, for example, Google's Gmail service—using this principle reduces the latency for loading JavaScript from 2600 ms to just 240 ms. With this in mind, let's take a look at some useful tips we can consider to help reduce the impact of mixing responsive images and videos on a mobile device:

• If we are designing for mobile devices, it is essential to work out which platform our users are using. This will dictate what content can be displayed. One way is that we can use lazy loading (in the form of media queries) to only show images when required:

```javascript
// equivalent to 720px
@media all and (min-width: 45rem) {
  body:after { content: 'desktop'; display: none; }
}
```

Another method encourages conditional loading to only load those elements and styling that are necessary:

```javascript
var size = window.getComputedStyle(document.body, ':after').getPropertyValue('content');
if (size == 'desktop') {
  // load content here...
}
```

These methods do not require any additional libraries to be imported. This is ideal if our site is already code-heavy and we want to avoid adding more to our site. In reality though, we may look to use something like RequireJS to manage the importing of files when needed; this does come with the overhead of the RequireJS library though!
• A good practice is to carefully consider whether all of the elements on the page are really needed for a usable mobile experience. Check those social media buttons. For example, some may be larger than you think and while they could be replaced with smaller/lighter versions, it may make better sense to not include them at all for mobile devices. Alternatively, consider loading the larger ones using conditional loading, while smaller ones can be loaded automatically. A great example of an easy replacement is to consider using the FontAwesome icons instead of images. This will help reduce the number of files that need to be loaded by the server (see the next point).

• Consider using data URIs (via a generator service, such as http://datauri.net/) or sprites/web fonts (such as Entype or FontAwesome) in place of images. Although data URIs or sprites may not make the code smaller, they will reduce the number of requests to the server. Using web fonts can have the same effect too; we can take it even further with building a custom font using the Fontello service at http://www.fontello.com).

• If you are using videos on your site, consider hosting them on YouTube. This will save space and bandwidth costs to your site. In addition, there will be a consistent format, which reduces the risk of any issues where videos can’t play on mobile devices.

• Be careful about where you use large, high quality images—on mobile devices, your visitors will not thank you when their data usage goes through the roof! There are plenty of options available to reduce the image size. We will look at two such examples later in this chapter that use Node JS to resize and compress images automatically.

Alternatively, if you need to display images on high pixel density displays, doubling the size but increasing the compression will help reduce the file size. To see the effect in action, take a look at the tutorial available at http://greatfridays.com/blog/images-in-responsive-web-development/. Can you see any difference in quality between the two images used in the PayPal demo?

• Always set videos to show a poster image and not to automatically play for mobile devices. There are many options to produce images (including those that can be automated), so there is no excuse. We’ll see one example later in this chapter, which can be easily adapted to produce an extra image to serve as the poster.
A useful tool to get to grips with is the Page Visibility or Network APIs. This appears very simple to implement, but opens up a variety of possible uses. One such use is to shut off any videos that are playing temporarily if a browser window is not being displayed; this will help reduce bandwidth costs.

If your site requires a lot of vector images, you will get better results using SVG-formatted files. These are effectively XML files that can resize with no loss of quality.

Enough theory – let's actually try out two of these tips; we could complete them manually, but instead we can take advantage of a task runner to complete them automatically. Intrigued? I will explain all, beginning with a look at creating responsive versions of our chosen images.

**Creating responsive images automatically**

A key part of providing content for any website is of course images. This becomes all the more important on responsive sites where we've seen that multiple versions of the same image are needed to provide a fluid solution to our responsive needs.

We could of course create each version of the image manually, but this is the 21st century: who wants (or even has the time) to create different sizes manually? Surely there has to be a better way. You'd be right: we can automate the whole process. Let's take a look how using the Node JS task runner as the basis to automate the process.

This exercise is designed to run as standalone; the principles can easily be applied to a larger, more complex grunt process as you get more accustomed to using Grunt.

Let's make a start by installing Node JS:

1. We'll begin by browsing to [http://nodejs.org/download/](http://nodejs.org/download/) to download the latest version for your platform. It's available for Windows, Mac, and Linux platforms. Double-click on the **MSI installer** and run through the wizard, accepting all defaults.

2. From the code download, extract the **responseimg** folder and save it on your PC.

3. Bring up a Node JS command prompt, then change to the **responseimg** folder and run this command to install **grunt-responsive-images**: 
   ```
   npm install
   ```
4. This will install grunt-responsive-images, along with the dependencies and Grunt; in this instance, we're using the grunt-simple-watch applet to automatically monitor our folder for any changes.

5. At the command prompt, change to the src folder and then run the following command:
   ```
   grunt autoresize
   ```

6. This kicks off the grunt-simple-watch plugin to begin automatically monitoring for changes in the src folder. In this example, I've already dropped a single JPEG image into the src folder, which is also shown in the following screenshot:

   ![Grunt Command Output](image-url)

   Running "simple-watch" task
   Running "responsive_images:files" <responsive_images> task
   Resized 1 file for small
   Resized 1 file for medium
   Resized 1 file for large
   Running "simple-watch" task
   Simple Watch, Waiting...

7. Any image dropped into the src folder will be used to create three new images, such as the example shown in the following screenshot:

   ![Image Screenshot](image-url)

   ![Image Screenshot](image-url)
The automated process is now ready for use. We can drop any number of JPEG images into the folder; three new images will be produced for each new image dropped into the folder.

We're using the default sizes for each image as specified in the plugin; these can be easily customized in the options for the plugin within our gruntfile.js file.

**Automating the shrinking process**

We can of course develop this automation further. It's definitely worth reading up on the capabilities of Node and checking out some of the wide range of plugins that have been created to help automate otherwise time-consuming processes that do not add any value when done manually.

Let's take a look at one way of extending our automated process. We've set up a process to create the images, but there is a likelihood that some of the images are not compressed as much as they could be. We can fix that by adding in support for compression, using the grunt-contrib-imagemin plugin:

1. Open up a copy of the gruntfile.js file. We first need to add in support for the grunt-contrib-imagemin plugin, so go ahead and modify the line as shown in the following code snippet:

   ```javascript
   files: ['src/*.jpg'],
   tasks: ['responsive_images', 'imagemin'],
   options: {
   
   2. Next, alter the code to add in the block for imagemin task as shown in the following code snippet:

   ```javascript
   dest: '../responseimg/assets'
   },
   imagemin: {
   }
   ```

3. When running the task in Grunt, we need to tell it to load in the grunt-contrib-imagemin plugin, so go ahead and add this line into the gruntfile.js file as indicated in the following lines of code:

   ```javascript
   grunt.loadNpmTasks('grunt-responsive-images');
   grunt.loadNpmTasks('grunt-contrib-imagemin');
   ```
4. We need to make one final change, so go ahead and alter the following line as shown:

   grunt.registerTask('default', ['simple-watch']);

5. Bring up a NodeJS command prompt and run the following command to add the plugin and its dependencies to the package.json file:

   npm install grunt-contrib-imagemin --save-dev

6. We're now ready to test our Grunt package. In the command prompt, change the directory to the project folder, then run this command:

   grunt

7. If all is well, we'll see the Grunt task run and successfully produce three new images, which are then compressed.

![Grunt Task Run](image.png)

The automation process is now ready for use. Although we've only put one image through the process, we could at this point put many images through. I would recommend running some tests to gauge how many you can safely process at the same time, as this will be dependent on resources in your PC or Mac.
Summary

Phew! We've certainly packed a lot into a small space! Let's recap on what you've learned throughout this chapter. We kicked off with a demo of mixing video and image content, initially to see firsthand some of the issues we face when mixing content, but then to use some of the tips and tricks you've learned to get the pages to behave responsively.

Having looked at a demo, we then moved onto considering some of the pitfalls of mixing content; we saw how the large increase in responsive use has increased over the last few years making it crucial that our designs work on a variety of platforms. We covered a number of tips we can use to make allowances for mobile platforms in our designs.

We ended the chapter looking at two demos using NodeJS to automate the creation of different versions of our images and automatically shrinking them; this illustrates how we can automate the process and save us a lot of manual effort.

Now that we have our content, we need to test it thoroughly to ensure it works across a wide range of devices. We'll take a look at the testing process in the next chapter, examining some of the considerations and pitfalls we need to be aware of when working with responsive media.
Where to buy this book
You can buy Responsive Media in HTML5 from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.