Community Experience Distilled

Responsive Web Design with HTML5 and CSS3 Essentials

Design and deliver an optimal user experience for all devices

Alex Libby Gaurav Gupta

Asoj Talesra

[PACKT] open source* community experience distilled PUBLISHING

Responsive Web Design with HTML5 and CSS3 Essentials

Design and deliver an optimal user experience for all devices

Alex Libby
Gaurav Gupta
Asoj Talesra

BIRMINGHAM - MUMBAI

Responsive Web Design with HTML5 and CSS3 Essentials

First published: August 2016

Production reference: 1240816

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78355-307-5

www.packtpub.com

Credits

Authors

Alex Libby

Gaurav Gupta

Asoj Talesra

Reviewer

Sasan Seydnejad

Commissioning Editor

Kartikey Pandey

Acquisition Editor

Larissa Pinto

Content Development Editor

Sachin Karnani

Technical Editor

Pranav Kukreti

Copy Editor

Charlotte Carneiro

Project Coordinators

Nikhil Nair

Ritika Manoj

Proofreader

Safis Editing

Indexer

Hemangini Bari

Graphics

Abhinash Sahu

Production Coordinator

Arvindkumar Gupta

About the Authors

Alex Libby has a background in IT support. He has been involved in supporting end users for almost 20 years in a variety of different environments; a recent change in role now sees Alex working as an MVT test developer for a global distributor based in the UK. Although Alex gets to play with different technologies in his day job, his first true love has always been with the open source movement, and in particular experimenting with CSS/CSS3, jQuery, and HTML5. To date, Alex has written 11 books on subjects such as jQuery, HTML5 Video, SASS, and CSS for Packt, and has reviewed several more. *Responsive Web Design with HTML5 and CSS3 Essentials* is Alex's twelfth book for Packt, and second completed as a collaboration project.

> *Writing books has always been rewarding. It's a great way to learn new technologies and give back something to others who have yet to experience them. My thanks must go to Packt and my coauthors for letting me review and assist with editing this book. My special thanks must also go to family and friends for their support too—it helps get through the late nights!*

Gaurav Gupta is a young and budding IT professional and has a good amount of experience of working on web and cross-platform application development and testing. He is a versatile developer and a tester and is always keen to learn new technologies to be updated in this domain. Passion about his work makes him stand apart from other developers.

Even at a relatively early stage of his career, he is a published author of two books, named *Mastering HTML5 Forms* and *Mastering Mobile Test Automation* with Packt Publishing.

A graduate in computer science, he currently works for a reputed Fortune 500 company and has developed and tested several web and mobile applications for the internal use.

Gaurav is a native of Chandigarh, India, and currently lives in Pune, India.

> *First of all, I would like to thank the almighty and my family, who have always guided me to walk on the right path in life. My heartfelt gratitude and indebtedness goes to all those people in my life who gave me constructive criticism, as it contributed directly or indirectly in a significant way toward firing up my zeal to achieve my goals. A special thanks to my sister, Anjali, who is a constant support, always.*

Asoj Talesra is an enthusiastic software developer with strong technical background. As a hybrid mobile app developer, he is responsible for crafting and developing intuitive, responsive web pages, and mobile apps using HTML5, CSS3, JavaScript, AngularJS, jQuery, jQuery Mobile, Xamarin, and Appcelerator Titanium. He works with a Fortune 500 company, and is well experienced in the areas of banking, quality and compliance, and audit.

At the very first, I'd like to thank Gaurav Gupta for advising me with such an amazing opportunity. The astounding encouragement and support from my family and friends is something I'm really indebted to, and I owe each one of you a part of this. I'd also like to thank Apurva and Poonam especially, for their contributions and feedback that helped me a lot shaping up this book.

We wish to extend our sincere gratitude to the team from Packt Publishing and the technical reviewers for their valuable suggestions, which proved extremely helpful in making this a better book for the readers. Our special thanks to our mentors, colleagues, and friends for sharing their experiences, which have proved very valuable in making this book better oriented toward the real-world challenges faced.

- Gaurav Gupta and Asoj Talesra

About the Reviewer

Sasan Seydnejad has more than a decade of experience in web UI and frontend application development using JavaScript, CSS, and HTML in .NET and ASP.NET environments. He specializes in modular SPA design and implementation, responsive mobile-friendly user interfaces, AJAX, client architecture, and UX design, using HTML5, CSS3, and their related technologies. He implements framework-less and framework-based applications using Node.js, MongoDB, Express.js, and AngularJS. He is the holder of the US patent for a user interface for a multidimensional data store—US Patent 6907428.

www.PacktPub.com

eBooks, discount offers, and more

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Table of Contents

Preface

A question—how many devices do you own that can access the Internet?

As a user, I'll bet the answer is likely to be quite a few; this includes smart TVs, cell phones, and the like. As developers, it is up to us to provide a user experience that works on multiple devices. Welcome to the world of responsive design!

Responsive design is not only all about creating a great user experience, but one that works well on multiple different devices, from a simple online ordering process for tickets, right through to an extensive e-commerce system. Many of the tips you see throughout the course of this book don't require extensive changes to your existing development methodology. In many cases, it's enough to make some simple changes to begin building responsive sites.

Creating responsive sites can open up a real world of opportunity for you; over the course of this book, I'll introduce you to the essential elements that you need to be aware of when designing responsively, and provide you with examples and plenty of tips to help get you started with creating responsive designs.

Are you ready to get started? Here's hoping the answer is yes. If so, let's make a start.

What this book covers

`Chapter 1`, *Introducing Responsive Web Design*, kicks off our journey into the world of responsive design, with an introduction into the basics of the concept; we explore the importance of RWD in today's environment and examine how it works as a concept.

`Chapter 2`, *Creating Fluid Layouts*, takes a look at creating flexible grid layouts as a key element of our design process; we explore the benefits of using them, and take a look at creating some examples using prebuilt styles.

`Chapter 3`, *Adding Responsive Media*, walks us through how to make our media responsive. We cover some of the tips and tricks available for use and examine why, in some cases, it is preferable to host content externally (such as videos)—if only to save on bandwidth costs!

`Chapter 4`, *Exploring Media Queries*, leads us to explore media queries and how we can use them to control what content is displayed at particular screen width settings. We cover the basics of creating breakpoints and examine why these should be based around where content breaks in our design and not simply for specific devices we want to support.

`Chapter 5`, *Testing and Optimizing for Performance*, rounds off our journey through the essentials of responsive web design with a look at how we can test and optimize our code for efficiency. We explore some of the reasons why pages load slowly, how we can measure our performance, and understand why even though the same tricks can be applied to any site. It's even more critical that we incorporate them when designing responsively.

What you need for this book

All you need to work through most of the examples in this book is a simple text or code editor, an Internet browser, and Internet access. Many of the demos use Google Chrome, so it is ideal if you have this installed; other browsers can be used, although there may be instances where you have to change the steps accordingly.

Who this book is for

The book is for frontend developers who are familiar with HTML5 and CSS3, but want to understand the essential elements of responsive web design. To get the most out of this book, you should have a good knowledge of HTML and CSS3; JavaScript or jQuery are not required for the purposes of running the demos in this book or understanding the code.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Go ahead and extract a copy of `coffee.html` and save it to our project area."

A block of code is set as follows:

```
img {
   max-width: 100%;
   height: auto;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
@media only screen and (min-device-width: 768px) and (max-device-width :
1024px) and (orientation : landscape)
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Click on the cog, then select **Share and embed** map"

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Responsive-Web-Design-with-HTML5-and-CSS3-Essentials`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/ResponsiveWebDesignwithHTML5andCSS3Essentials_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Introducing Responsive Web Design

The concept of web design used to be simple—designers would develop content for a popular desktop screen, using a layout which works for most devices to produce well laid-out and cohesive pages.

With changes in technologies and the introduction of mobile devices, the whole experience changed—gone are the days when a static layout would suffice. In its place came a need for content that responded to the available screen real estate, with elements automatically resized or hidden, according to the device being used. This forms the basis for a technique popularized by Ethan Marcotte, which we now know as **responsive web design** (**RWD**). Throughout the course of this chapter, we will begin to explore what this means, and understand some of the principles that are key to this concept.

In this chapter, we will cover the following topics:

- Getting started with RWD
- Understanding the elements of RWD
- Understanding the importance of RWD
- Exploring how RWD works
- Setting up a development workflow
- Exploring best practice and common errors

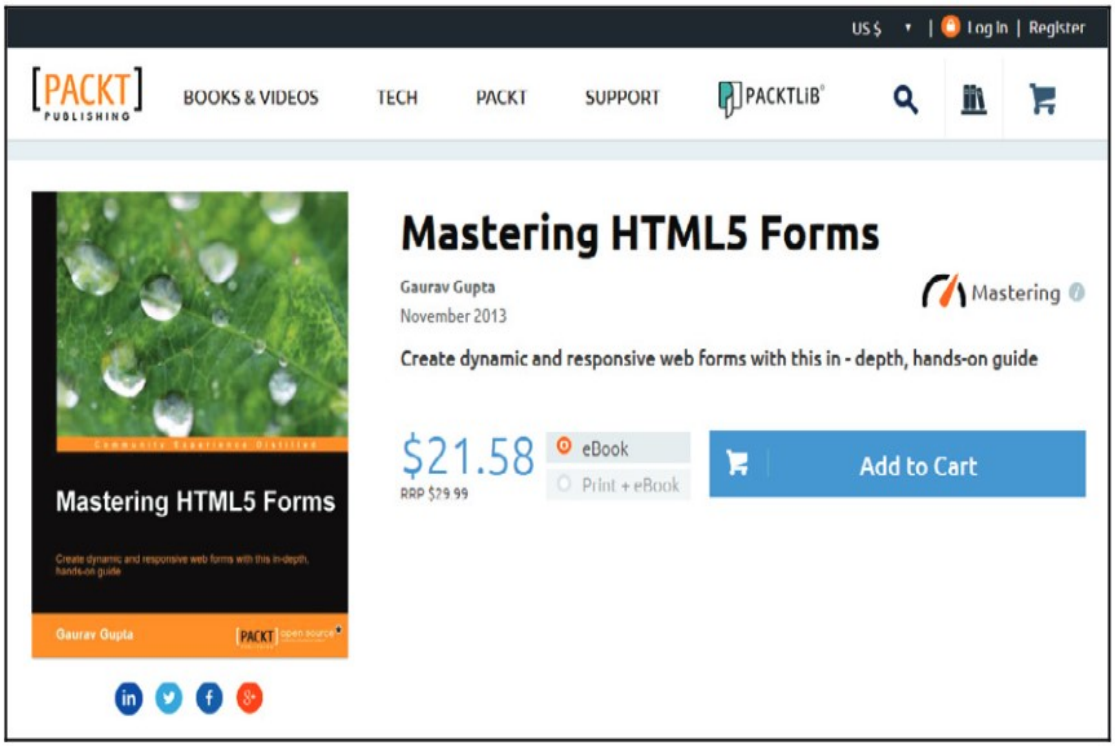
Curious? Let's get started!

Getting started with RWD

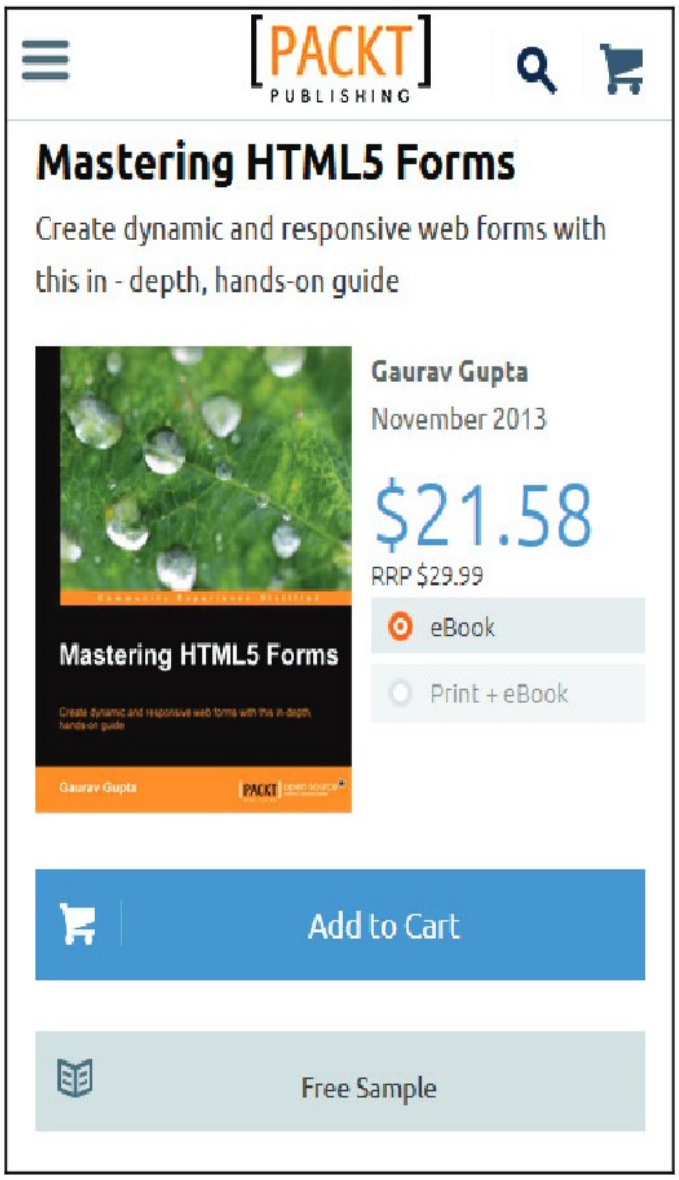
If one had to describe RWD in a sentence, then responsive design describes how the content is displayed across various screens and devices such as mobiles, tablets, phablets, or desktops. To understand what this means, let's use water as an example. The property of water is that it takes the shape of the container in which it is poured. It is an approach in which a website or a webpage adjusts its layout according to the size or resolution of the screen dynamically. This ensures that users get the best experience while using the website.

We develop a single website that uses a single code base. This will contain fluid, flexible images, proportion-based grids, fluid images, or videos and CSS3 media queries to work across multiple devices and device resolutions. The key to making them work is the use of percentage values, in place of fixed units such as pixels or ems-based sizes.

The best part of this is that we can use this technique without the knowledge or need of server based/backend solutions; to see it in action, we can use Packt's website as an example. Go ahead and browse `https://www.packtpub.com/web-development/mastering-html5-forms`; this is what we will see as a desktop view:

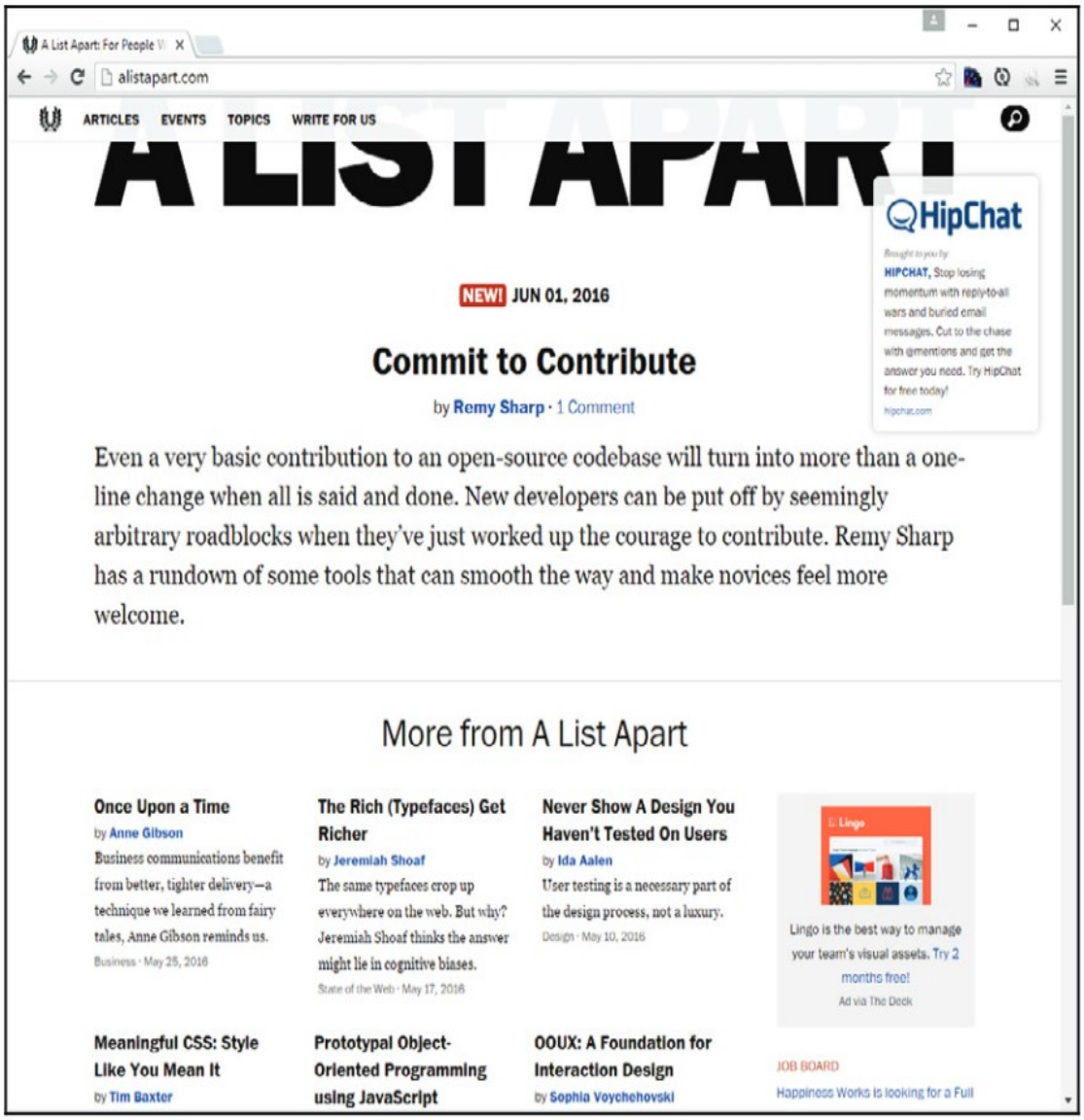


The mobile view for the same website shows this, if viewed on a smaller device:



We can clearly see the same core content is being displayed (that is, an image of the book, the buy button, pricing details, and information about the book), but elements such as the menu have been transformed into a single drop-down located in the top-left corner. This is what RWD is all about—producing a flexible design that adapts according to which device we choose to use, in a format that suits the device being used.

Let's try it with another technology site; I am sure some of you are familiar with the A List Apart site (hosted at `http://alistapart.com` and founded by the well-known Jeffery Zeldman):



Try resizing your browser window. This is a perfect example of how a simple text site can reflow content with minimal changes; in place of the text menu, we now have the hamburger icon (which we will cover later in this chapter):

While the text in this site realigns with minimal difficulty, it seems that the top image doesn't work so well—it hasn't resized as part of the change, so will appear cut off on smaller devices.

Although this may seem a complex task to achieve, in reality it boils down to following some simple principles; how these are implemented will ultimately determine the success of our site. We've seen some in use as part of viewing Packt's and A List Apart's sites at various sizes—let's take a moment to explore the principles of how responsive design works, and why it is an important part of creating a site for the modern Internet.

Exploring how RWD works

For some, the creation of what we now know as RWD is often associated with Ethan Marcotte, although its true origins date from earlier when the site Audi.com was first created, which had an adaptive viewport area as a result of limitations within IE at the time.

If one had to describe what RWD is, then Ethan sums it up perfectly:

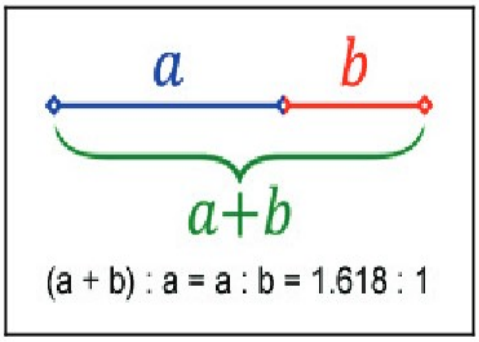
Rather than tailoring disconnected designs to each of an ever-increasing number of web devices, we can treat them as facets of the same experience. We can design for an optimal viewing experience, but embed standards-based technologies into our designs to make them not only more flexible, but more adaptive to the media that renders them. In short, we need to practice responsive web design.

In a nutshell, RWD is about presenting an experience for customers on different devices that allows them to interact with your business. It is important to note though that the experience does not have to use the same process; it is more important that the customer can achieve the same result, even though they may arrive using a different route. So, how does RWD work?

RWD is a set of principles we should follow, but the overriding philosophy is about making content fluid. Gone are fixed values, at least for elements on the page; in their place, we use percentage values or em/rem units. Our page layout will use a fluid grid, which resizes automatically depending on the available viewport space.

One key concept that will help determine the success of our site is not one we might automatically associate with responsive design, but nevertheless will help: divine proportion.

Divine proportion, or the Golden Ratio as it is often known, is a way of defining proportions that are aesthetically pleasing—it is perfect for setting the right proportions for a responsive layout. The trick behind it is to use this formula:



Imagine we have a layout that is 960px wide, which we want to split into two parts, called **a** and **b**. Divine proportion states that the size of **a** must be **1.618** times the size of **b**.

To arrive at our column widths, we must complete the following calculations:

1. Divide the width of our site (960px) by 1.618. This gives 593px (rounded to the nearest integer).
2. Subtract 593px from 960px. This gives 367px.

It's a simple formula, but one that will help improve the communication of content for your site—we're not forced to have to use it though; sites are available on the Internet that don't follow this principle. It does mean that they must ensure that the content is still equally balanced, to give that pleasing effect—this isn't so easy!

The important point here though is that we shouldn't be using fixed pixel values for a responsive site; instead, we can use rem units (which resize better) or ideally percentage values.

To translate this into something more meaningful, we can simply work out the resulting column widths as percentages of the original size. In this instance, 593px equates to 62% and 367px is 38%. That would give us something like this:

```
#wrapper { width: 60rem;}
#sidebar { width: 32%; }
#main { width: 68%; }
```

Okay, a little theoretical, but hopefully you get the idea! It's a simple formula, but a great way to ensure that we arrive at a layout which is properly balanced; using percentage values (or at the very least rem units), will help make our site responsive at the same time.

Understanding the elements of RWD

Now that we've been introduced to RWD, it's important to understand some of the elements that make up the philosophy of what we know as flexible design. A key part of this is understanding the viewport or visible screen estate available to us; in addition to viewports, we must also consider flexible media, responsive text and grids, and media queries. We will cover each in more detail later in the book, but to start with, let's take a quick overview of the elements that make up RWD.

Controlling the viewport

A key part of RWD is working with the viewport or visible content area on a device. If we're working with desktops, then it is usually the resolution; this is not the case for mobile devices.

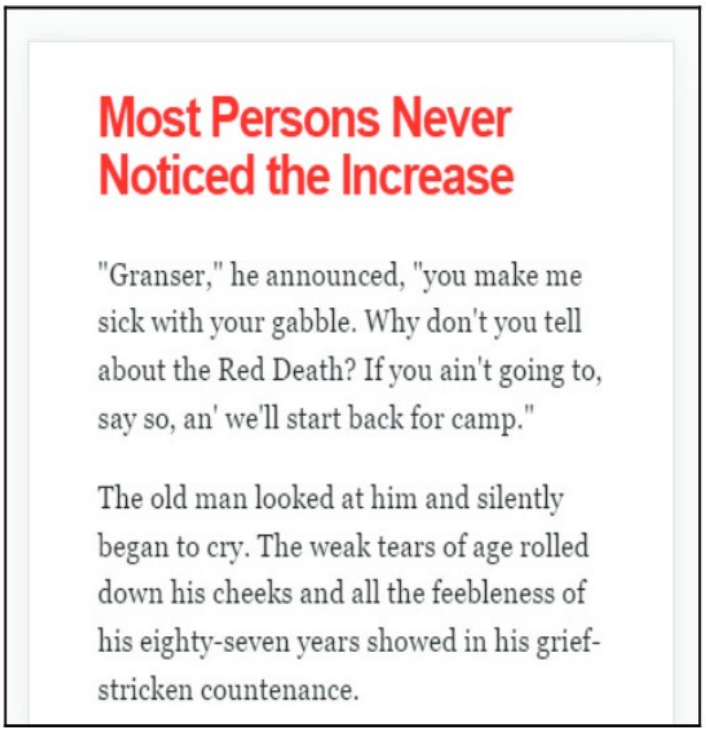
There is a temptation to reach for JavaScript (or a library such as jQuery) to set values such as viewport width or height, but there is no need, as we can do this using CSS:

```
<meta name="viewport" content="width=device-width">
```

or using this directive:

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

This means that the browser should render the width of the page to the same width as the browser window—if for example the latter is 480px, then the width of the page will be 480px. To see what a difference that not setting a viewport can have, take a look at this example screenshot:



This example was created from displaying some text in Chrome, in iPhone 6 Plus emulation mode, but without a viewport. Now, let's view the same text, but this time with a viewport directive set:

Even though this is a simple example, do you notice any difference? Yes, the title color has changed, but more importantly the width of our display has increased. This is all part of setting a viewport—browsers frequently assume we want to view content as if we're on a desktop PC. If we don't tell it that the viewport area has been shrunk in size (and that we have not set the scaling correctly), it will try to shoehorn all of the content into a smaller size. This will result in a page that will appear to be zoomed out, which doesn't work very well!

It's critical, therefore, that we set the right viewport for our design, and that we allow it to scale up or down in size, irrespective of the device. We will explore this in more detail, in `Chapter 2`, *Creating Fluid Layouts*.

Creating flexible grids

When designing responsive sites, we can either create our own layout or use a grid system already created for use, such as Bootstrap. The key here though is ensuring that the mechanics of our layout sizes and spacing are set according to the content we want to display for our users, and that when the browser is resized in width, it realigns itself correctly.

For many developers, the standard unit of measure has been pixel values; a key part of responsive design is to make the switch to using percentage and em (or preferably rem) units. The latter scales better than standard pixels, although there is a certain leap of faith needed to get accustomed to working with the replacements!

Making media responsive

Two key parts of our layout are of course images and text—the former though can give designers a bit of a headache, as it is not enough to simply use large images and set overflow—hidden to hide the parts that are not visible!

Images in a responsive site must be as flexible as the grid used to host them—for some, this may be a big issue if the site is very content heavy; now is a good time to consider if some of that content is no longer needed, and can be removed from the site. We can of course simply apply `display: none` to any image which shouldn't be displayed, according to the viewport set. However, this isn't a good idea though, as content still has to be downloaded before styles can be applied; it means we're downloading more than is necessary! Instead, we should assess the level of content, make sure it is fully optimized and then apply percentage values so it can be resized automatically to a suitable size when the browser viewport changes.

Constructing suitable breakpoints

With content and media in place, we must turn our attention to media queries—there is a temptation to create queries that suit specific devices, but this can become a maintenance headache.

We can avoid the headache by designing queries based on where the content breaks rather than for specific devices—the trick to this is to start small and gradually enhance the experience, with the use of media queries:

```
<link rel="stylesheet" media="(max-device-width: 320px)" href="mobile.css"
/>
<link rel="stylesheet" media="(min-width: 1600px)" href="widescreen.css" />
```

We should aim for around 75 characters per line, to maintain an optimal length for our content.

Appreciating the importance of RWD

So – we've explored how RWD works, and some of the key elements that make up this philosophy; question is, why is it so important to consider using it? There are several benefits to incorporating a responsive capability to our sites, which include the following:

- It is easier for users to interact with your site, if it is designed to work with multiple devices.
- Creating a single site that caters to multiple devices may in itself require more effort, but the flip side of this is that we're only developing one site, not multiple versions.
- Constructing a site that works based on how much can be displayed in a specific viewport is a more effective way to render content on screen than relying on browser agent strings, which can be falsified and error prone.
- RWD is able to cater to future changes. If we plan our site carefully, we can construct media queries that cater for devices already in use and ones yet to be released for sale.

Making our site accessible and intuitive

Accessibility plays a key role in responsive design. Our content should be as intuitive as possible, with every single piece of information easy to access. Responsive design places great emphasis on making our design self-explanatory; there shouldn't be any doubt as to how to access information on the site.

In this context, accessibility refers to making our site available on a wide variety of devices; this should not be confused with the need to cater for individuals with disabilities. Making sites accessible for them is equally important, but is not a primary role in RWD.

Even though our mobile version may not contain the same information (which is perfectly accessible), it nonetheless must be engaging, with appealing colors, legible text (at all sizes), and a design that retains visual harmony and balance with our chosen color scheme.

Organizing our content

The success of our site is determined by a host of factors, of which one of these will be how our content is organized in the layout. The content should be organized in such a way that its layout makes it easy to process, is simple and free of clutter, and that we're making full use of the available viewport space for the device we're targeting.

We must also ensure our content is concise—we should aim to get our point across in as few words as possible so that mobile users are not wasting time with downloading redundant content. Keeping our options simple is essential – if we make it too complicated, with lots of links or categories, then this will increase the time it takes for visitors to make decisions, and ultimately only serve to confuse them!

At this point, it is worth pointing out something – over time, you may come across the phrase **adaptive design**, when talking about responsive design. There is a subtle but key difference between the two, and either can be used as a principle when constructing our site. Let's take a moment to explore what each means, and the differences that might affect how we go about constructing our sites.

Comparing RWD to adaptive design

So, what is adaptive design, and how does it differ to responsive design?

Responsive design is about making one design fit many devices—it requires us to create the optimal viewing solution for a site, no matter which device we care to use. This means that we should not have to resize, scroll, or pan more than is absolutely necessary; if for example our page width doesn't fit the screen we're using, then our design isn't right! Ultimately though, we can view responsive design as a ball that grows or shrinks in size automatically, to fit several sizes of hoops.

Staying with the hoop analogy, adaptive design works on the principle that we have multiple layouts that are available for use; the browser will select the right one to use, based on detecting which type of device is in use. In this instance, we would be putting several different balls through different sized hoops, depending on the size of hoop in use. The key difference though is that responsive design focuses on client-side development; adaptive design in the main uses server-side detection to display the best-fitting page, based on the device being used to browse the site.

For the purpose of this book, we will work on using responsive design throughout all of the examples used within the text.

Now that we understand the importance of using RWD and how it differs from adaptive design, let's really begin on our journey; our first step is to get our development environment prepared for use. At this point, you might be expecting to download lots of different plugins or be using libraries such as jQuery. You might be in for a little surprise!

Preparing our development environment

Okay, we've covered enough general background; time to get more practical!

There are many tools available to help us, when constructing responsive sites; this of course includes tools such as JavaScript or jQuery, but also plugins such as FitVids (to manage videos, responsively) or ResponsiveSlides for creating responsive carousels.

However, we're not going to use any of them. All we need is a text editor and a browser, and nothing more! We're not going to download anything as part of completing the exercises in this book.

Yes, I hear those exclamations of incredulity. I must have lost my marbles, I hear you say. There is a very good reason for this approach though; let me explain:

On too many occasions, I see instances where we simply reach for the latest plugin to help us achieve a result. Ordinarily, there is nothing wrong with this; after all, time pressures frequently mean that we can't afford the time to take a more considered approach.

However, I believe we've become lazy. There is no need for many of the tools that are available, when building responsive sites. It's time to go back to basics; throughout the course of this book, we're going to prove that we can build the basics of responsive functionality, with nothing more than a simple text editor and a browser for viewing.

There are some caveats to this approach though, that we should bear in mind:

- Much of what we construct won't work in some older browsers—IE9 or below is a good case in point. The question you must ask yourself is: how many people use this for your site? If the percentage is very low, then you can consider dropping them; if not, then you will need to seek a different approach.
- Concentrating on using just HTML and CSS does not mean that we're rejecting other tools outright; if we need to use them, then we need to use them. The question we must ask ourselves though is this: do we really need to use them? Or are we just too lazy to go *old school* and create things from the ground up?

With that aside, there are a couple of administration tasks we need to complete first; we need a project area to store our content. I would recommend creating a folder somewhere on your PC or Mac to store files; I will for the purposes of this book assume you've called it `B03568`, and that it is stored on the `C:` drive. If you've stored it somewhere else, then adjust accordingly.

Next up, we will need a copy of the code download that accompanies this book—there will be instances where we won't cover some of the more mundane content, but focus on the really important content; we can get those less critical files from the code download.

Finally, do you have a text editor that you like using? If not, then you might like to look at Sublime Text 3; it is our preferred editor of choice. The real benefit of using it means we can add plugins, such as the REM-PX (available from `https://packagecontrol.io/packages/REM%2PX`), which is perfect for converting from pixel to rem-based values! (We will cover this more in later chapters).

Okay, onwards we go; the next stage in our journey is to consider a suitable strategy for creating our responsive sites. There is nothing outrageously complex about this, it is more about making some sensible choices as to what approach we should use which bests fit our solution. Let's take a moment to explore what this means in practice.

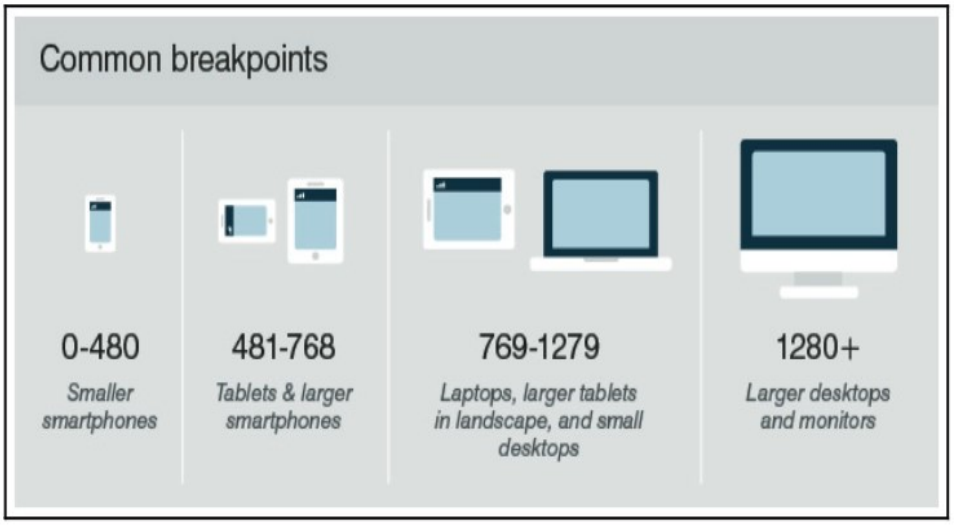
Considering a suitable strategy

As a developer, I am sure you will be wanting to get stuck into creating your next masterpiece, right? Sounds familiar; after all, that is what helps to pay the bills…

Before we can get into writing code, we must develop some form of strategy, this can be as simple or as complex as befits our requirements. Although there is something to be said for defining ground rules, I've never been one for rigid principles; instead, developing a set of guidelines and principles to follow means that we can be flexible about our approach, while still retaining a sense of consistency.

There are a few guidelines that we should try to incorporate into our development process, these are more about helping to reduce any lack of clarity when developing responsive sites for the first time:

- **Choosing mobile over desktop**: This is not a new idea; it has been around since before responsive design took off as a serious approach to creating sites. The mobile-first concept means that we should develop our content for smaller mobile devices first, before working up to tablets and desktops. At this stage, it is important to be aware of what visitors may be using for your current site; this will help determine which devices we should concentrate on as part of our design.
- **Breakpoints**: Getting these right will be essential to the success of any site we build; not only must we choose the right thresholds for the devices we want to support, but also ensure that the cut-off points do not overlap each other. A common misconception is to develop for standard breakpoints such as desktops or tablets; instead, we should set our breakpoints to kick in when our content breaks within our layout. This screenshot shows the type of thresholds we have to consider in our designs:



- **The flow of content**: When we start designing for smaller screens, content will naturally be squeezed and begin to flow downwards. This may be tricky to grasp as a concept, if you're used to pixels and points when designing, but it will make sense once you get used to how content will flow downwards in responsive design.
- **Relative units**: This is an important concept in responsive design. It's a case of learning how to make the switch from static values to relative units. Calculating values in percentages means that content will always fit, no matter what the viewport size; for example, the size of a `<div>` set to 50% means that it will only ever fill 50% of its parent container, no matter what the size of the viewport. It may be a case of building the initial design with static values, but we should try to get into the mindset of converting to use percentages as part of our design process.
- **Max and min values**: A part of using relative values means that our browser won't understand what the lower and upper size limits of each element will be. To work around this, we must set `min-width` or `max-width` values; these will ensure that no matter what width our elements are at, they won't go past the limits set in code.
- **Web fonts or system fonts**: If you maintain a desktop version of your site already, and it uses one or more web fonts, then you have an important decision to make: should the mobile site use the same fonts? The reason for asking is because this requires additional content to be downloaded; it will consume extra bandwidth, which will be a concern for devices where this is limited. Bear in mind that anyone using a mobile device is likely to be under time pressure, they will want to do something quickly and with the minimum of fuss, so selecting a non-standard font won't be a useful way forward.
- **Bitmaps or vector images**: When working with responsive designs, a key principle is scalability; we must have images that can easily be resized or swapped for larger or smaller as appropriate. Taking the latter approach on a mobile device isn't wise; a better route would be to consider using vector images. These not only resize without loss of quality, but can also be manipulated using CSS if desired, which reduces the need for separate versions of the same image (provided the vector image has been optimized prior to editing with CSS).
- **Content strategy**: The aim of responsive design is to create the best possible experience, irrespective of the device used to view the site. Part of this centers around content; as a part of developing a strategy, content should always come first. We need to add enough to make our design a useful foundation; we can then add or develop this at a later date, when we add support for larger screens and resolutions.

- **Sketch and prototype**: Once we have worked out our content and a strategy for managing it, it's time to develop the layout. A key part of this should incorporate sketching or wireframing; it will help turn the rough beginnings of a plan into something more solid. Many designers will use PhotoShop, but this is at the risk of wasting lots of billable hours that must be accounted for with the client. Sketching on paper is portable, and has a low visual and content fidelity, which means we can focus on how content interacts, rather than how it looks.
- **Frameworks**: Using a framework in our development can bring several benefits to our development process; it's tried and tested code means that we can cut down on development time, and make use of the grid approach to build and refine our layout around our content. Frameworks will have already been tested across multiple devices, so the debugging time is reduced; we should concentrate on choosing the right framework, based on criteria such as the learning curve required, support, and documentation availability.

A caveat with using frameworks though is their size; if we go down this route, we should choose carefully which one to use, as many are code heavy and adding extra unnecessary code will make our sites slower.

- **Scalable media**: Images and videos are essential for any site; not only do they help add color, they can also convey more in a smaller space. All of our images must be scalable; it doesn't matter if we swap images out as our resolution increases, or we use a scalable format such as SVG. We need to give consideration to sourcing both standard and hi-resolution versions of our images, to cater for those devices that support the latter; these can either be individual or in the form of image sprites. Our media strategy should also consider using icon fonts, which are perfect for small, repeatable elements such as social media logos.
- **Minification**: If we're building sites to work on devices where bandwidth constraints and limited resources may be a factor, then we must consider minifying our CSS and JavaScript content. Minifying should be a standard part of any development workflow. We can reduce the number of HTTP requests to the server and improve response times. When designing responsively, minification becomes more critical, as we are adding more CSS styles (such as media queries) to cater for different viewports. This will inflate the size of our style sheets even further so anything we can do to help reduce the size, will help encourage takeup of our newly developed responsive site on mobile devices.

As developers, we should have some form of strategy in place when creating our sites; it should not be so rigid as to prevent us changing direction if our existing plans are not working. Whichever way we decide to go, we should always consider an element of best practice in our development workflow; there are a few tips we could use right about now, so let's take a look in more detail.

Exploring best practices

Best practices…ugh…what a phrase!

This is a phrase that is used and abused to death; the irony is that when it is used, it isn't always used correctly either! This said, there are some pointers we can use to help with designing our responsive sites; let's take a look at a few:

- **Use a minimalistic approach**: This is very popular at the moment and perfect for responsive design. It reduces the number of elements we have to realign when screen dimensions change; it also makes editing content and elements easier, if our page content is at a minimum.
- **Always go for a mobile-first strategy**: It is essential to consider what the user experience will be like on a mobile device. The proportion of those who own a mobile device or phone is higher than those who own their own PC; we must make sure that content is both viewable and legible on a smaller screen. Once we've nailed this, we can then extend our design for larger screens.
- **Understand the need for speed**: A slow site is a real turn off for customers; a good guideline to aim for when measuring load times is for around 4–5 seconds. There can be many reasons why a site is slow to load, from slow site feeds to poorly optimized hardware, but one of the easier to rectify is large images. Make sure that you've been through all of the media that is loaded, and checked to ensure it has been fully optimized for your site.
- **Try to keep the site design as clean as possible**: Eliminate anything that is not needed to convey your message. It goes without saying, but why use 20 words, when we can get our message across in 10?
- **Make use of the hamburger icon:** No, I'm not referring to food, irrespective of what size it is (sorry, bad joke!). Instead, make use of it as a consistent point of access to your site. Be mindful though that users may not want to have to keep tapping on it to access everything, so if you have menu items that are frequently used more often, then consider exposing these in your site, and hide anything less important under the hamburger icon.

- **Don't use small button sizes for anything**: Bear in mind that many users will use fingers to tap on icons, so any clickable icons should be large enough to allow accurate tapping and reduce any frustration with accidentally tapping the wrong link.
- **Familiarize yourself with media queries**: As we'll see later in `Chapter 4`, *Exploring Media Queries*, we can use media queries to control how content is displayed under certain conditions on different devices. These play a key role in responsive design; it's up to us to get the right mix of queries based on what we need to support!

There are plenty more best practices we can follow, but we shouldn't simply follow them blindly; apart from the risk of our sites losing any *stand-out* appeal (that is, all being the same), not all best practice necessarily applies or is even accurate.

Instead, it is worth researching what others have done online; over time, you will begin to see common threads—these are the threads that should form the basis for any discussions you may have with regards to the design of your site.

There are some practices we should follow, not necessarily because they are best practices, but borne more out of common sense; a great example is touch. When we are designing for touch, there are some pointers which we should use that will influence our design, so let's take a look at these in more detail.

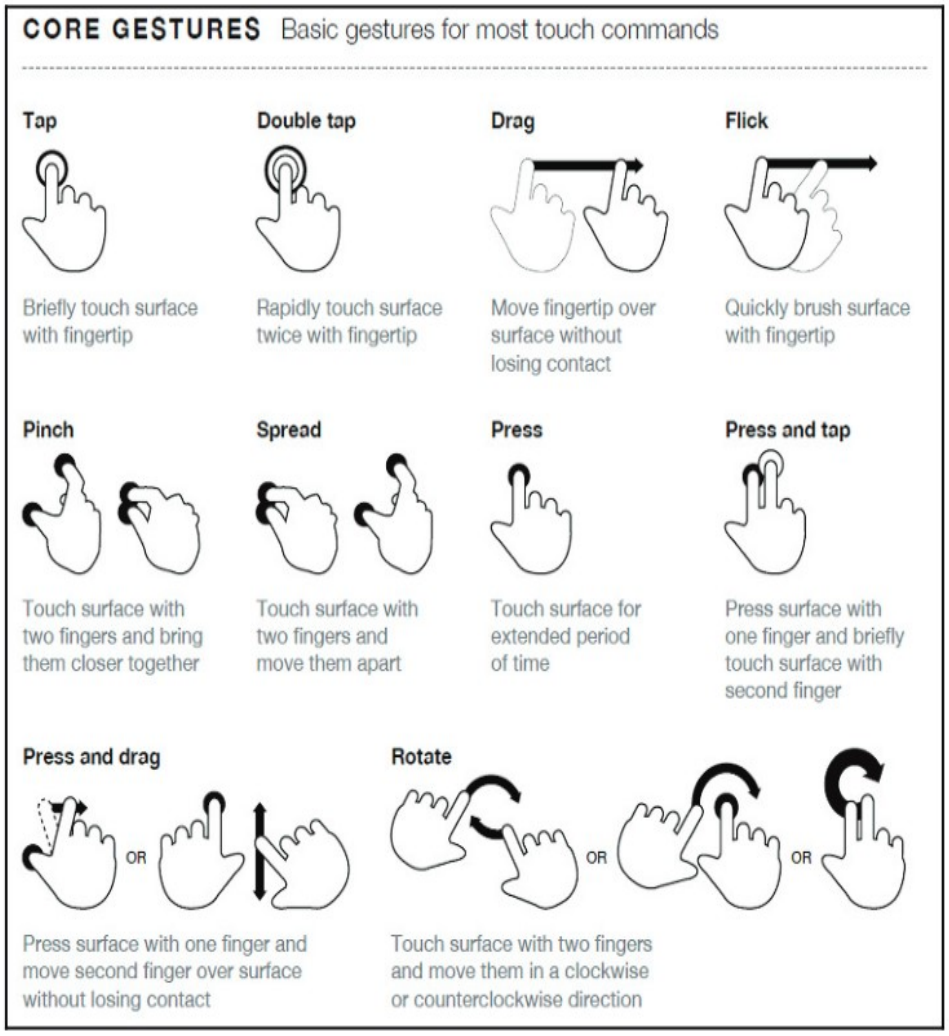
Designing for touch

Although there are lots of tips and pointers we can use to help improve our responsive development, there is one subject that is worth exploring in more detail—touch.

Why? The answer is simple—working responsively is not just about writing code; anyone can write code. The difference though, and that which separates great developers from the crowd, is the thought given to how that site looks and works. Care paid at this point will separate exceptional sites from the also-rans; to get a feel for the kind of decisions and thoughts one might go through, let's take a look at what we might have to consider when designing for touch:

- Aim to use the corners. On small screens, the lower left corner is frequently the easiest to access; for tablets, the top corners work better. We must consider putting any call to action buttons in these locations, and adapt our designs to realign automatically if different devices are used.
- Don't make the buttons too small. A good guideline for tap targets (including buttons) is usually around 44 points (or 3.68rem).

- Avoid placing items too closely together to prevent someone accessing the wrong item by mistake. A good starting point for spacing to avoid interface errors is a minimum of 23pt (or 1.92rem).
- Use natural interactions and create navigation that works well with common hand gestures, such as swiping. This screenshot shows some of the example hand gestures we can use, and that we must allow for when constructing our sites:



Source: The Next Web (`http://www.thenextweb.com`)

Hover stages are a no-no in responsive design—these do not exist. Any calls to action should be based around tapping, so make sure your design takes this factor into consideration.

Phew, there are a few things to think of there! The key thing though is that while writing code is easy, creating an effective responsive design takes time and resources, and should always be a continuous affair, so our designs can stay fresh and up to date.

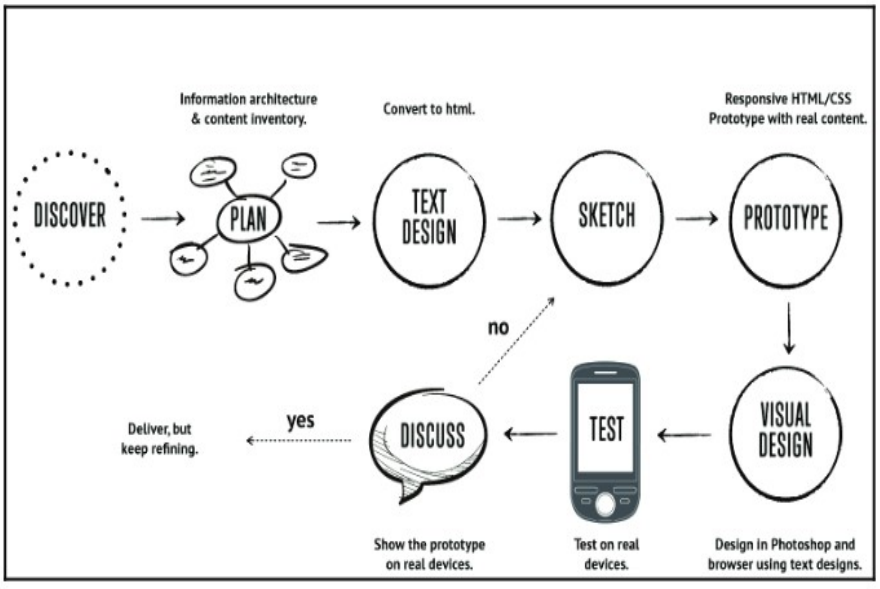
It's time for us to move on to a different subject, now that we've explored some of the guidelines and tips we can use to help with responsive development, it's time for us to consider our workflow. We may already have an established one for producing standard sites, making a shift to incorporate responsive design may require us making some changes to our processes, so let's go and explore what a typical workflow could look like when working with responsive design.

Setting up a development workflow

Before we start on our workflow, let me ask you a question:

Does your workflow process consist of steps such as planning, creating static wireframes, building static designs, adding code then testing, and finally launching your site? Sounds familiar?

If so, then be prepared for a change; using the waterfall process (to which these steps align), doesn't work so well when creating responsive sites; working responsively is all about being agile, as illustrated by the developer Viljami Salminen from 2012:



Source: Viljami Salminen, 2012

Although his design dates from 2012, it is still perfectly valid today; before you all put your hands up in horror, let me explain why a waterfall process doesn't work for responsive design:

- We cannot hope to get support for lots of different devices built into our site in one go. We must focus on a small viewport first, before gradually enlarging it to include support for larger devices and finally desktops.
- When working responsively, our focus must be on content first, and not the layout; the reason for this is that our media queries should be built around where content breaks if we change the width of our available viewport. This removes the dependency on building for specific devices, and means that we can build queries that can serve multiple devices.

Our development process needs to be iterative, and focus chiefly on text and media before constructing the layout. Throughout the process, we should keep in constant contact with our client, as part of each iteration; gone are the days of working towards the big reveal! With all of this in mind, let's go through Viljami's illustration in more detail.

Discovering the requirements

This initial stage is the same for any sites, but is particularly important for responsive design, given the different viewport sizes we have to support in our site. It's all about getting to know the client and their requirements. We must get an understanding of our client's business, their competitors, and what they are trying to achieve.

It's at this point we should be asking questions such as, “Why would people come to your site?”, “What is the main goal you are trying to achieve?”, and “Who are your main competitors?” The more we ask, the more we can understand and therefore better advise our clients on the best solution to fit their requirements.

Planning our site

When we've gathered all of the knowledge needed to construct our site, we now need to plan how it will look. We can start with the initial concept, but rather than work on the layout as we might have done before, we should concentrate on the content elements, and creating user stories and the information architecture. We can then put this together in a rudimentary HTML wireframe. At this stage, layout is less critical; it's key that we get the right view of information, before focusing on where it sits on the page.

Designing the text

At this point, we now need to focus on writing our content in textual form. This often under-rated step is probably the most important of the whole process; without it, people won't come to see the site, and there is no point in designing a layout if we don't know what will fill it! A useful tip is to keep styling to a minimum and to concentrate on the content; at least initially, we can see how it will display in a long, continuous format, and if it works for those who use screen readers. Don't worry though, we can always refine the text during the prototyping stage; at this point, we need something to start with, but it will be unlikely we get it right first time.

Sketching our design

For our next step, forget using PhotoShop; it takes forever and is a real pain to make corrections quickly! The agile process is about making quick and easy changes and there is no better medium than traditional pen and paper. We can even print off the plain text content and sketch around, it if that helps, it will save us hours of development time, and can even help reduce those occasions when you hit the developer's block of…what now?

Prototyping and creating our design

With the design sketched, it's time to create a prototype. This is when we can see how the layout will respond to different viewport sizes and allow us to highlight any potential challenges, or react to any issues that are reported by our client. It's good to be aware of the various breakpoints, but ultimately we should let our content determine where these breakpoints should be set in our design.

We might be used to prototyping our design using PhotoShop, but a better alternative is to switch to using a browser. We can use a service such as Proto.io(`https://proto.io/`) or Froont (`http://froont.com/`). This gives us extra time to get the harder elements right, such as typography; it also helps to remove any constraints that we might have with tools such as PhotoShop.

Testing our design

Although we may still be at a prototype stage, it's important to introduce testing early on. The number of breakpoints we may end up having in our design means that testing on multiple devices will take time! As an alternative to running a big test and reveal, we can instead perform multiple tests and reveals. This has the benefit of reducing the impact of any rollbacks (if something doesn't fit requirements), but also helps to keep the client involved with the project, as they can see progress take place during development.

The one thing we absolutely must do is to test over multiple devices. It's an expensive proposition to maintain a test suite with these devices, so it's worth asking colleagues, friends, and family to see if they can help test for you. They can at least browse the site and help pinpoint where things *don't look right* (to use customer parlance). It's up to us to work out where the root cause is, and implement a fix to help improve the user experience.

Exploring mistakes

With the best will in the world, we are only human; there will be times when we make a mistake! As the playwright Oscar Wilde once said, "*...to err once is human, to err twice is careless.*" Well, in the hope that we don't go that far, there are some common errors that are regularly made when working responsively; let's take a look at the top five candidates:

- **Failing to allow for touch input**: It might seem odd, but failing to allow for touch is surprisingly common! Some users expect a site to *simply work*, and to have a consistent approach across the board both for desktops and mobiles. This will include any mechanism for activating links (such as buttons). What might work on a desktop client will very likely fail on a mobile device. It is absolutely key that we include something to allow mobile users to navigate around a site using touch. This can be achieved (in the main) with CSS3 styling, so there is no excuse!
- **Insisting on a consistent navigation**: A part of creating a successful site will be to have some form of navigation that retains a consistent look and feel across all pages; it does not mean to say that we have to carry this over to mobile devices though! Navigation on a mobile device will of course act differently; we have the added extra of touch input to cater for, as part of our design. At an absolute minimum, links and buttons, along with our choice of typeface and colors should remain consistent; the size of buttons, our visual layout, and how we click on buttons can change.

- **Building in links to non-mobile friendly content**: How many times have you accessed content via a mobile device, only to find it is a huge image or substantial document that has to be downloaded? I'll bet that this irked you. Make sure your content is suitable for downloading on different devices. On a broadband connection for a desktop, we might think nothing of using 100Kb images; try loading these over a 3G connection, and it is easy to see why we need to reconsider what we've previously used on standard broadband connections.
- **Ignoring performance**: In an age of modern broadband connections, it is easy to be complacent about what is made available for download. If we think desktop first, then we will be building ourselves a real problem when it comes to designing our mobile site! We can't compress a desktop environment into a mobile experience—it won't be efficient, will offer poor user experience, and ultimately lead to a drop in conversions in sales. To avoid issues with performance, we should aim to use a minimalistic or *mobile-first* approach, as the basis for our site designs.

- **Testing**: A common error to make is not sufficiently testing our solutions on multiple devices, running them prior to release will uncover any UX issues that need to be resolved before making our solution available for general use. A sticking point is likely to be the question of how many devices we test for. If we have access to analytics for an existing desktop version of the site, then this should give us a starting point we can use to double check our design is working as expected. Failing this, we can make use of device emulators in browsers to run some basic checks. We can also use online testing services, such as MobileTest.me (`http://mobiletest.me`), to ensure our design is working sufficiently to release for wider use.

These common issues can easily be solved with some simple changes to our development workflow process, building at least some of these steps to avoid the grief we may get from these errors, early on, will save a lot of heartache later during development!

Summary

The philosophy that is RWD opens up lots of opportunities for us as designers. With the advent of mobile and other internet-capable devices, it is important to not only make the switch, but also understand how to get it right. We've covered a number of useful topics around RWD, so let's take a moment to reflect on what you've learned in this chapter.

We kicked off with a gentle introduction to RWD, before exploring the basic principles behind making our sites responsive and understanding some of the key elements that make up RWD.

We then moved on to explore the importance of RWD as a set of guiding principles we can follow; we explored how this compares to adaptive design, and that while responsive design can be harder to implement, it is worth the effort over time.

Next up came a look at strategy—we covered the importance of getting this right, along with the different elements that should be considered when making the move toward working responsively. We took a look at some of the best practices that we can follow and called out designing for touch as a key part of these guidelines, to illustrate some of the decisions we need to make during development.

We then rounded out the chapter with an extensive look at creating a development workflow. We explored how we may have to make changes to our existing processes, and some of the topics that have to be incorporated into our development, before discussing some of the points where we might trip us up, if we don't take care over our designs!

Phew, there's a lot of detail there! The great thing though is that we've covered a lot of the strategic considerations we need to make; it's time to put some of this into practice and start building content and layouts. Let's move on and start looking at how we can build flexible grid layouts. This will be the subject of the next chapter in our journey.

2

Creating Fluid Layouts

A key part of our journey through the essentials of responsive design is laying out content on the page—in the early days of the Internet, this was a simple process!

With the advent of mobile devices (and those non-PC devices) that can access the Internet, content layout has become ever more critical; for example, how many images do we have, or do we include content X, or show a summary instead? These are just some of the questions we might ask ourselves. It goes to show that it can open a real can of worms!

To simplify the process, we can use grid or fluid-based layouts. Throughout the course of this chapter, we'll take a look at using them in more detail; we'll start with setting the available viewport, and take it right through to future grid-based layouts.

In this chapter, we will cover the following topics:

- Introducing grid layouts and understanding different types
- Setting the available viewport for use
- Exploring the benefits and mechanics of using grid layouts
- Implementing a prebuilt grid layout
- Exploring the future of grid-based template layouts

Curious? Let's get started!

Note that the exercises have been designed for the Windows platform, as this is the authors' platform of choice; alter as appropriate if you use a different platform.

Introducing flexible grid layouts

For many years, designers have built layouts of different types; they may be as simple as a calling card site, right through to a theme for a content management system, such as WordPress or Joomla. The meteoric rise of accessing the Internet through different devices means that we can no longer create layouts that are tied to specific devices or sizes—we must be flexible!

To achieve this flexibility requires us to embrace a number of changes in our design process—the first being the type of layout we should create. A key part of this is the use of percentage values to define our layouts; rather than create something from the ground up, we can make use of a predefined grid system that has been tried and tested, as a basis for future designs.

The irony is that there are lots of grid systems vying for our attention, so without further ado, let's make a start by exploring the different types of layouts, and how they compare to responsive designs.

Understanding the different layout types

A problem that has faced web designers for some years is the type of layout their site should use—should it be fluid, fixed width, have the benefits of being elastic, or a hybrid version that draws on the benefits of a mix of these layouts?

The type of layout we choose to use will of course depend on client requirements—making it a fluid layout means we are effectively one step closer to making it responsive; the difference being that the latter uses media queries to allow resizing of content for different devices, not just normal desktops!

To understand the differences, and how responsive layouts compare, let's take a quick look at each in turn:

- **Fixed width layouts**: These are constrained to a fixed width; a good size is around 960px, as this can be split equally into columns, with no remainder. The downside is fixed width makes assumptions about the available viewport area, and if the screen is too small or large, it results in lots of scrolling which affects the user experience.
- **Fluid layouts**: Instead of using static values, we use percentage-based units; it means that no matter what the size of the browser window, our site will adjust accordingly. This removes the problems that surround fixed layouts at a stroke.

- **Elastic layouts**: They are similar to fluid layouts, but the constraints are measured by type or font size, using em or rem units; these are based on the defined font size, so 16px is 1 rem, 32px is 2 rem, and so on. These layouts allow for decent readability, with lines of 45-70 characters; font sizes are resized automatically. We may still see scrollbars appear in some instances, or experience some odd effects if we zoom our page content.
- **Hybrid layouts**: They combine a mix of two or more of these different layout types; this allows us to choose static widths for some elements while others remain elastic or fluid.

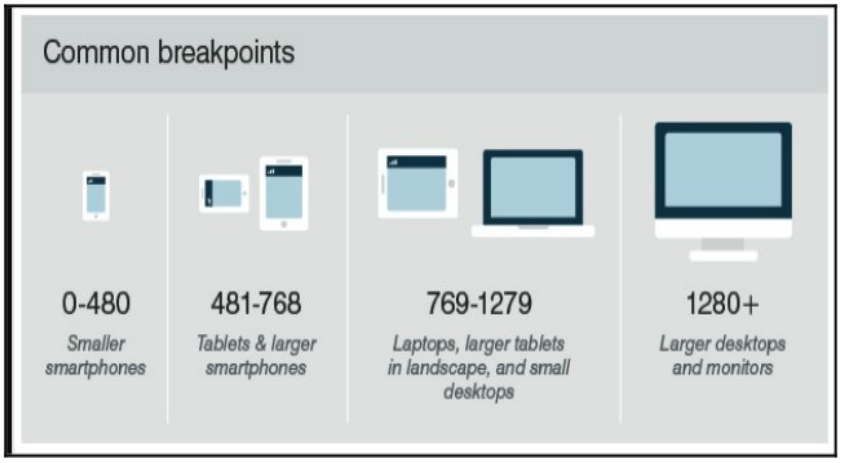
In comparison, responsive layouts take fluid layouts a step further, using media queries to not only make our designs resize automatically, but also present different views of our content on multiple devices.

How do we set the available space though, and be sure that our content will zoom in or out as appropriate? Easy—we can do this by adding the viewport directive to our markup; let's go and explore what is required to allow our viewport to resize as needed.

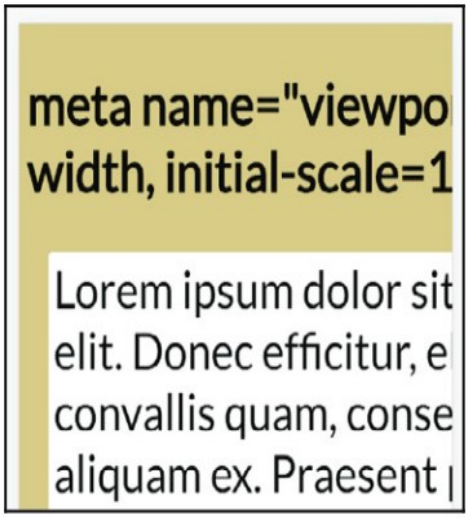
Setting the available viewport for use

When viewing a website on different devices, we of course expect it to resize to the available device width automatically with no loss of *experience*; unfortunately, not every site does this quite the right way or successfully!

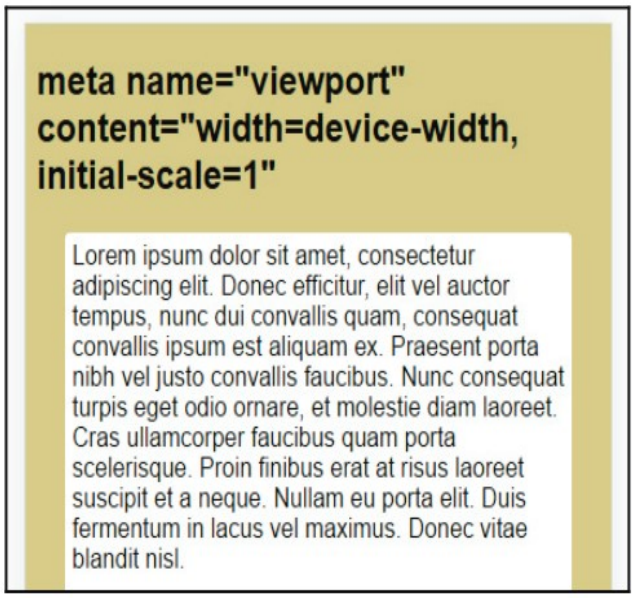
To understand why this is important, let's assume we operate a desktop version of our site (one in the 1280+ group in this screenshot), and a mobile equivalent from the 418-768 group:



The first stage in making our site responsive is to add the viewport directive; without it, we are likely to end up with a similar effect to this when resizing our sites:



See what I mean? It looks awful—text is cut off, we would have to swipe to the right…ugh! In stark contrast, adding one line of code can have a dramatic effect:



Our example uses the Google Chrome set to emulate an iPhone 6 Plus. The code needed to restore sanity to our example can be added to the `<head>` of our code:

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

Once set, we can immediately see the difference. Granted, our demo isn't going to win any style awards, but then it wasn't the aim! It does, however, show that the text has been reduced in size to fit the screen, we have a proper border around the text—it all looks more pleasing as a display.

To see what happens in action, try running the `viewport.html` demo from the code download that accompanies this book; you will need to run it in device/responsive mode for your browser; remove line 5, and re-add it back in to see the difference.

The content property in this directive supports using any one of a number of different values:

Property	Description
`width`	The width of the virtual viewport of the device.
`device-width`	The physical width of the device's screen.
`height`	The height of the virtual viewport of the device.
`device-height`	The physical height of the device's screen.
`initial-scale`	The initial zoom when visiting the page; setting 1.0 does not zoom.
`minimum-scale`	The minimum amount the visitor can zoom on the page; setting 1.0 does not zoom.
`maximum-scale`	The maximum amount the visitor can zoom on the page; setting 1.0 does not zoom.
`user-scalable`	Allows the device to zoom in and out (yes) or remain fixed (no).

Current versions of MS Edge don't play so well with viewport tags; it is worth noting that `@-ms-viewport` needs to be specified in code to ensure our viewport widths behave in the same way as other browsers.

Balancing viewport against experience

You will notice that I italicized the word experience at the start of this section—the key point here is that in responsive design, the experience does not have to be identical across all devices; it must be useful though, and allow our visitors to interact with us as an organization. In other words, if we worked for a theater, we might limit our mobile offer to simply booking tickets, and let the main desktop site manage everything else.

This is perfectly valid; while limiting a site, mobile ticketing might be considered by some as very restrictive. The concept is still technically sound, as long as the user experience is acceptable.

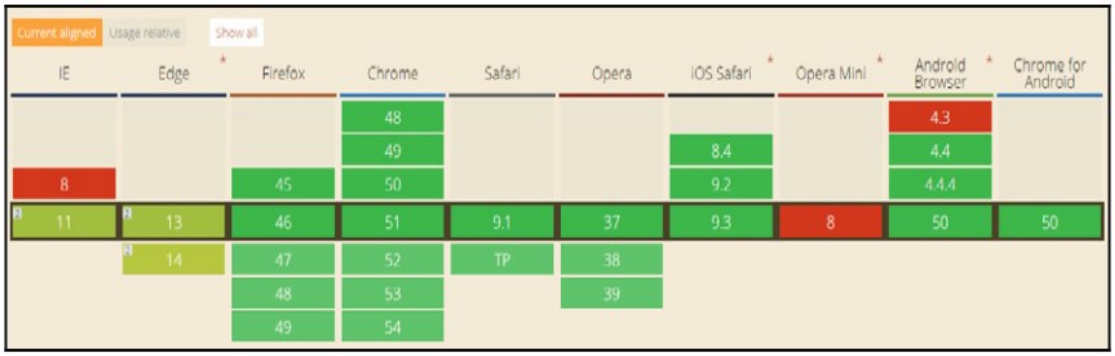
It's worth noting that we could have set a specific width using `width=<value>`. This is great if we need a certain width to display our content; if the orientation changes from portrait (320px) to landscape (360px) for example, then the viewport's content will be automatically scaled up and down to reflect these changes. If, however, we had set a device-width as a maximum, this implies that no scaling is need and that the browser should adjust the content within it to fit.

Considering viewport units

A key part of responsive design is to make the move away from using pixel values to working with em or rem units. In our examples (and the viewport demo from earlier in this chapter), we used both pixel and rem units. Although this works well, we still have a dependency on parent elements. Instead, we should consider using an alternative for working with viewports. They are:

- **vw**: viewport width
- **vh**: viewport height
- **vmax**: maximum of the viewport's height and width
- **vmin**: minimum of the viewport's height and width

As a unit of measure, these equate to 1% of the viewport area that has been set; the beauty though is that they remove any dependency elements, and are calculated based on the current viewport size. Browser support for them is currently very good:



Source: `http://caniuse.com/#search=vh`

Leaving aside the slight quirks with more recent versions of Internet Explorer, this is a useful option that combines the ease of units, with the flexibility of using percentages, in our designs.

Let's move on—we've introduced flexible grids and explored how setting a viewport is critical to displaying content correctly. It's time we moved on and explore some of the benefits of incorporating a grid system into our layout, and dive into the internals of how they work as a principle in responsive design.

Exploring the benefits of flexible grid layouts

Now that we've been introduced to grid layouts as a tenet of responsive design, it's a good opportunity to explore why we should use them. Creating a layout from scratch can be time consuming and needs lots of testing; there are some real benefits from using a grid layout:

- **Grids make for a simpler design**: Instead of trying to develop the proverbial wheel, we can focus on providing the content instead; the infrastructure will have already been tested by the developer and other users.
- **They provide for a visually appealing design**: Many people prefer content to be displayed in columns, so grid layouts make good use of this concept to help organize content on the page.
- **Grids can of course adapt to different size viewports**: The system they use makes it easier to display a single codebase on multiple devices, which reduces the effort required for developers to maintain and webmasters to manage.
- **Grids help with the display of adverts**: Google has been known to favor sites which display genuine content and not those where it believes the sole purpose of the site is for ad generation; we can use the grid to define specific areas for adverts, without getting in the way of natural content.

All in all, it makes sense to familiarize ourselves with grid layouts; the temptation is of course to use an existing library. There is nothing wrong with this, but to really get the benefit out of using them, it's good to understand some of the basics around the mechanics of grid layouts and how this can help with the construction of our site.

Let's take a quick look first at how we would calculate the widths of each element, an important part of creating any grid layout.

Understanding the mechanics of grid layouts

So far, we explored one of the key critical elements of responsive design, in the form of how we would set our available screen estate (or viewport)—as someone once said, *it's time...*

Absolutely—it's time we cracked on and explored how grids operate! The trick behind grids is nothing special; it boils down to the use of a single formula to help define the proportions of each element used in our layouts:

target ÷ context = result

Let's imagine that we have a layout with two columns, and that the container (or context) is 960px wide (I will use pixel values purely to illustrate the maths involved).

To create our layout, we will make use of the Golden Ratio that we touched on in `Chapter 1`, *Introducing Responsive Web Design*; to recap, we use the ratio of 1.618 to every 1 pixel. So, if our layout is 960px wide, we multiply 960 x 0.618 (the difference)—this gives 593px (rounded down to the nearest integer). We then simply subtract 593 from 960, to arrive at 367px for our side column. Easy, when you know how…!

At this stage, we can convert these to percentages; `593px` becomes `61.77%`, and the side bar will be `38.23%`. Let's translate this into some sample CSS, with values rounded to 2 decimal places:

```
section, aside {
   margin: 1.00%;     /*  10px ÷ 960px = 0.010416 */
}

section {
   float: left;
   width: 61.77%;     /* 593px ÷ 960px = 0.617708 */
}

aside {
   float: right;
   width: 38.23%;     /* 367px ÷ 960px = 0.382291 */
}
```

Here, our target is the `aside` (or sub-element), with context as the container; in this case, we've set it to 960px. The section forms a second target; in both cases, we've divided the target by the context to arrive at our result. As our result figures need to be expressed as percentages, we can simply multiply each by 100 to get the figures we need.

The observant among you will note the presence of `margin: 1.00%`. We must allow sufficient space for our margin, so the resulting figures will need to change. We'll keep the section width at `61.77%`, so our margin will need to drop down to `34.23%`, to retain a full width of 100% (this allows for the two margins each side of the two sub-elements).

If we carried this through to its conclusion, we could end up with something akin to this screenshot, as an example layout:

Okay, let's move on. I feel it's time for a demo! Before we get stuck into writing code, there are a few pointers we should take a quick look at:

- Although we've only scraped the surface of how grid layouts work, there is a lot more we can do; it will all depend on how many columns your site needs, whether the columns should be equal in width, or be merged with others, how big the container will be, and so on.
- There are dozens of grid layout frameworks available online. Before getting into designing and creating your own from scratch, take a look at what is available; it will save you a lot of time!
- Keep it simple; don't try to overcomplicate your layout. You may read stories of developers extolling the virtues of flexbox, or that you must use JavaScript or jQuery in some form or other; for a simple layout, it isn't necessary. Yes, we might use properties such as box sizing, but flexbox-based grid systems can become overinflated with CSS.

With this in mind, it's time we got stuck into a demo. Before we do though, there is something we need to cover, as it will become a recurring theme throughout this book:

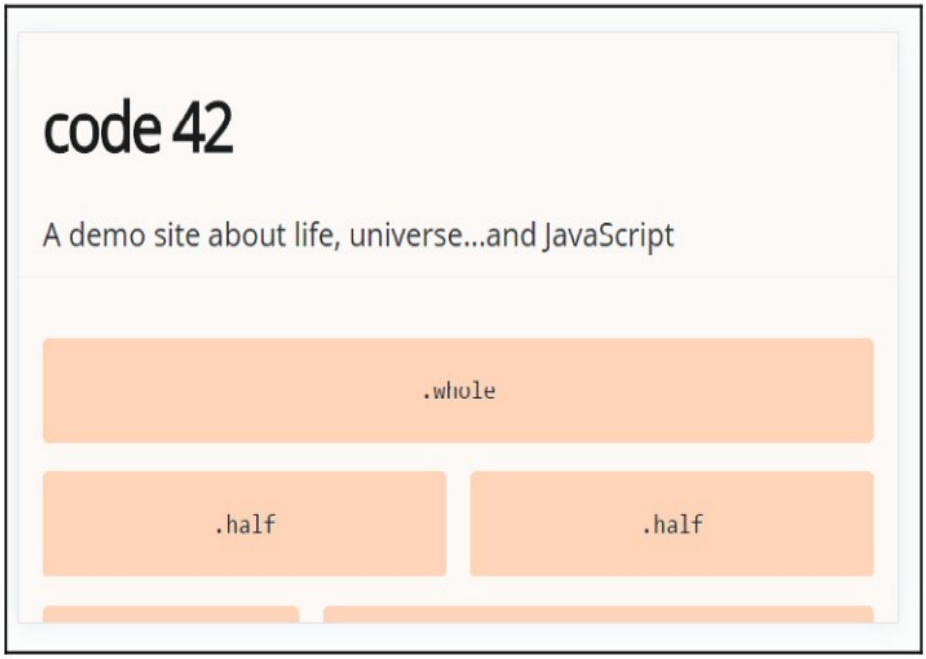
We will avoid the use of JavaScript or downloading libraries in order to create our demos. Yes, you heard right. We're going to attempt to use nothing more than plain HTML5 or CSS3 to construct our responsive elements!

The reason for this is simple—I maintain that we've become lazy as developers, and that sometimes it is good to go back to basics and really appreciate that sometimes simple is better. You may hear of singers who want to get back to their roots or where they started from; we're simply applying the same principle to our responsive development. It does mean that we can't always use the most feature-rich, or latest version, but that isn't always a bad thing, right?

Implementing a prebuilt grid layout

We've touched on the basics of creating grids; these can be really time consuming to create from scratch, so with so many already available online, it makes better sense to use a prebuilt version unless your requirements are such that you can't find one that works for you! It is worth spending time researching what is available, as no two grids are the same.

As an example of what is available and to prove that we don't need all the bells and whistles that grids can offer, let's take a look at an example grid, in the form of Gridism. We can see an example of how our next demo looks like when completed, in this screenshot:



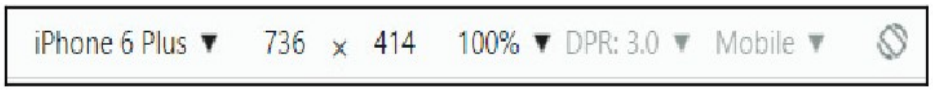
Although this library has been around for two to three years, its simplicity proves that we don't need to implement a complex solution in order to create the basis for a simple layout. The flexbox attribute in CSS is perfect for creating grids, but its flexibility adds a layer of complexity that isn't needed; instead, we'll make use of the `box-sizing` attribute, which will work just as well.

Created by Cody Chapple, it doesn't make use of flexbox (of which more, anon), but does make use of box-sizing as an attribute in the grid. The library can be downloaded from `https://github.com/cobyism/gridism/blob/master/gridism.css` (or installed using Bower), but as it consists of one file only, we can simply copy the contents to a text file and save it that way (and still keep to our earlier aim of not downloading content).

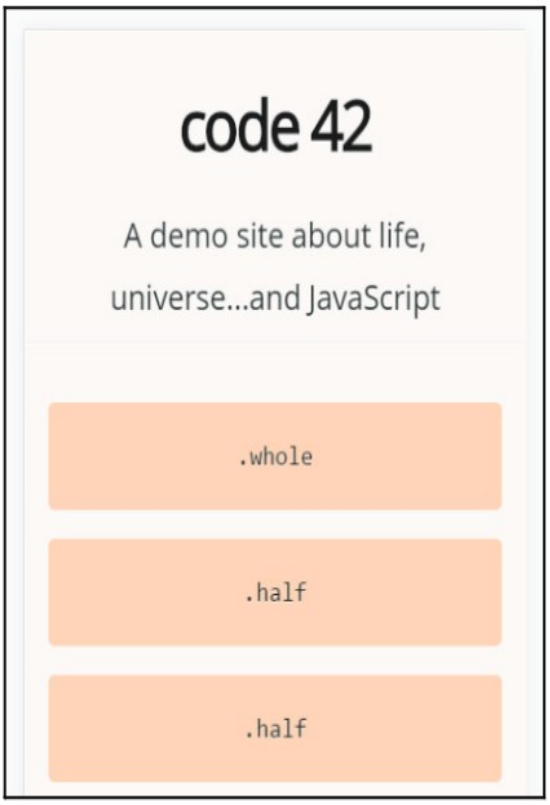
The demo will use the original example from the Gridism site, but the CSS has been reworked to bring it up to date and remove some unnecessary code. For ease of convenience, we will assume use of Google Chrome throughout this demo.

Let's make a start:

1. From the code download that accompanies this book, go ahead and download a copy of `gridism.html`, along with `normalize.css`, `gridism.css`, and `style.css`. Save the HTML markup at the root of our project area, and the two CSS files within the CSS subfolder.
2. Try running `gridism.html` in a browser, then enable its device or responsive mode (by pressing *Ctrl* + *Shift* + *I* then *Ctrl* + *Shift* + *M*). We should see something akin to the screenshot shown at the beginning of this exercise.
3. The screenshot at the start of this section was taken in Google Chrome, set to emulate an iPhone 6 Plus in landscape mode. Now use the orientation tool in Chrome:



4. To change the orientation to portrait:

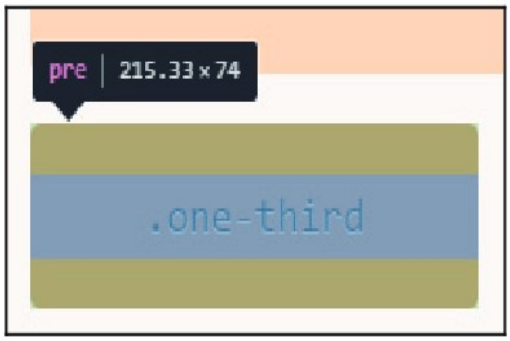


Notice how the grid has automatically realigned itself? The trick here is not in the `style.css` file, but within `gridism.css`; if we open it in a text editor and look for this block of code, on or around lines 50-61, it will look like this:

```
47  /* Width classes also have shorthand versions numbered as fractions
48   * For example: for a grid unit 1/3 (one third) of the parent width,
49   * simply apply class="w-1-3" to the element. */
50  .grid .whole,          .grid .w-1-1 { width: 100%; }
51  .grid .half,           .grid .w-1-2 { width: 50%; }
52  .grid .one-third,      .grid .w-1-3 { width: 33.3332%; }
53  .grid .two-thirds,     .grid .w-2-3 { width: 66.6665%; }
54  .grid .one-quarter,    .grid .w-1-4 { width: 25%; }
55  .grid .three-quarters, .grid .w-3-4 { width: 75%; }
56  .grid .one-fifth,      .grid .w-1-5 { width: 20%; }
57  .grid .two-fifths,     .grid .w-2-5 { width: 40%; }
58  .grid .three-fifths,   .grid .w-3-5 { width: 60%; }
59  .grid .four-fifths,    .grid .w-4-5 { width: 80%; }
60  .grid .golden-small,   .grid .w-g-s { width: 38.2716%; } /* Golden section: smaller piece */
61  .grid .golden-large,   .grid .w-g-l { width: 61.7283%; } /* Golden section: larger piece */
```

We can see that the library makes good use of percentage values to assign a width to each block. The real crux of this is not in the widths set, but the size of our container; for gridism, this is set to `978px` by default. So, for example, if we were to set a cell width of `.one-third`, we would want `33.3332%` of `736px`, or `245.33px`. We then ensure all grid cells have the right dimensions by applying the box-sizing attribute to each of our grid cells.

See how easy that was? In place of having to work out percentages, we simply specify the name of the column type we need, depending on how wide we need it to be:



Hold on a moment. How come the screenshot shows `215.33`, and not `245.33`, as the calculation indicated it should be?

Aha, this is just something we need to be mindful of; when working with a grid system like Gridism, the calculations are based on the full width of our viewport. Any padding required will be included within the width calculations of our column, so we may need a slightly larger column then we anticipate! It goes to show that even though our grid system doesn't have all of the mod-cons of current systems, we can still produce a useable grid, as long as we plan it carefully.

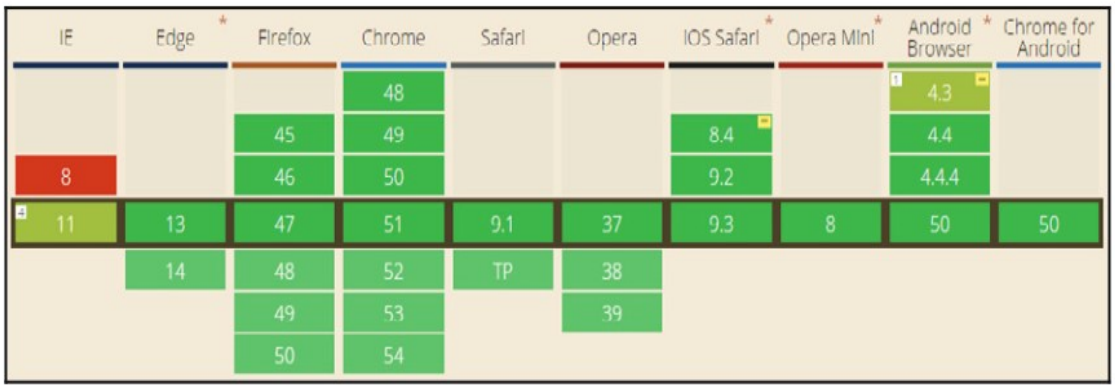
Okay, let's move on. We talked in passing about the fact that many grids use flexbox to help control their appearance; this is a great option to use, but can require setting a lot of additional properties that would otherwise be unnecessary for simple layouts. With careful planning, there is every possibility that we can avoid using it, but if we're working on a complex layout with lots of different elements, then there will be occasions when using it will avoid a lot of heartache! With this in mind, let's take a quick look at the basics of how it works in more detail.

Exploring the use of flexbox

So, what is flexbox?

It's a module that has been designed to provide a more efficient way to layout and distribute space around items in a container, particularly if their sizes are not yet known. We can set a number of properties to ensure that each item best uses the available space around it, even if its size changes.

At the time of writing, this is a W3C Candidate Recommendation; this means that it is effectively on the last call before becoming a browser standard in late 2016. This should be something of a formality though, as most browsers already support it as a standard:

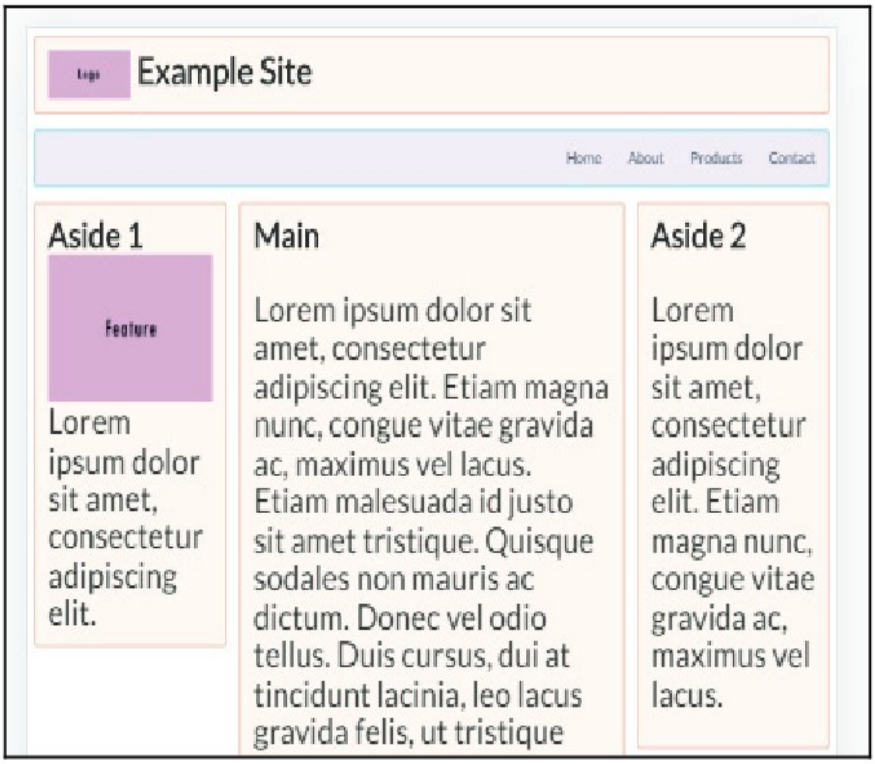


Source: `http://caniuse.com/#search=flexbox`

To fully understand how it all works is outside the scope of this book, but to help get started, we can run a quick demo, and explore some of the main features:

1. From the code download that accompanies this book, go ahead and extract copies of `flexbox.html` and `flexbox.css`; store the HTML markup at the root of our project area, and the CSS style sheet in the `css` subfolder of our project area.

2. Try previewing `flexbox.html` in a browser. For this, we will need to enable the browser's responsive mode (or device mode, depending on browser); if all is well, we should see something akin to this screenshot:

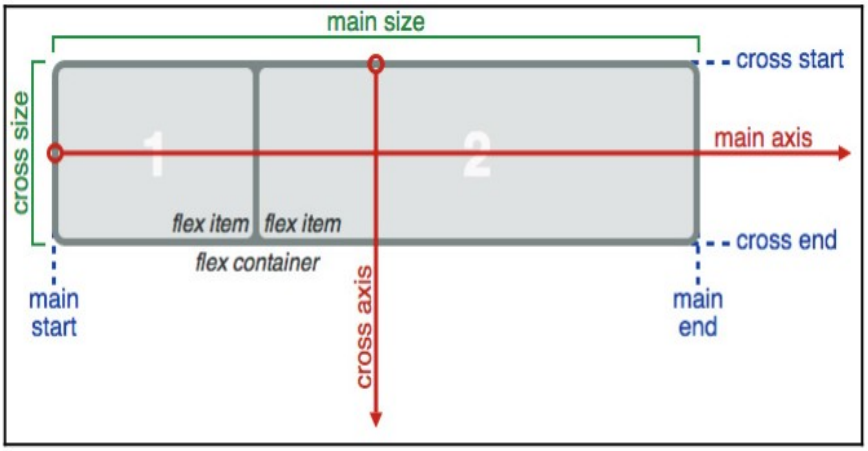


The demo is based on a pen created by Irina Kramer, which is available at `https://codepen.io/irinakramer/pen/jcLlp`; for the purposes of our demo, we focus on the example layout taken from that pen.

At first glance, this demo looks very straightforward. It could certainly use some help in the color department, but that's not what is of interest to us at the moment. If we dig deeper into the code, we can see that flexbox has been incorporated in various places; let's explore its use in more detail.

Taking a closer look

Taking a closer look at our code, we will find that a large part of it uses standard attributes, which we might find on any site. The code that is of interest to us starts on line 50; to understand its role, we first need to get our heads around the basic concept of flex layouts:



Source: W3C

In a nutshell, items are laid out following either the main axis (from main-start to main-end) or the cross axis (from cross-start to cross-end):

Property	Purpose
`main axis`	The primary axis along which flex items are laid out; this is dependent on the flex-direction property.
`main-start | main-end`	The start and end points of flex items that are placed within the container (horizontally).
`main size`	A flex item's width or height, whichever is in the main dimension, is the item's main size. The main size property can be the item's height or width size.
`cross axis`	The axis perpendicular to the main axis. Its direction depends on the main axis direction.
`cross-start | cross-end`	Start and end points for flex lines that are filled with items and placed into the container (vertically).
`cross size`	This is the width or height of a flex item, whichever is in the cross dimension.

With this in mind, let's explore some of the flexbox terms that have been used in our code; the initial few styles are standard rules that could apply to any site. The code of interest to us starts on line `29`.

If we scroll down to that line, we are met with this:

```
/*Basic Grid Styles*/
.Grid {
  display: flex;
  flex-flow: row;
  flex-wrap: wrap;
}

.Grid-cell {
  flex: 1;
}
```

Our first attribute, `display: flex`, defines the container which contains the flex items; here, we're setting it to show items in rows, and to wrap from left to right. If we had a number of columns in our layout, and by this I mean more than just two or three, then we might use `align-items` and `justify-content` to ensure that each column was evenly spread throughout the row, irrespective of the width of each column.

With the `.grid` defined, we need to style our grid-cells, or the containers where we host our content. There are several properties we can apply; the one we've used is `flex`, which is shorthand for `flex-grow`, `flex-shrink`, and `flex-basis`. In our case, it is recommended that the shorthand version be used, as this will set the other values automatically; we've set `flex-grow` to `1`, which indicates how much it should grow, in relation to other flexible items in the same container.

The next property of interest is in the `.nav` rule. Here, we've used flex-flow again, but this time we also justify-content; the latter controls how items are packed on each row (in this case, toward the end of the line):

```
.nav {
  list-style: none;
  background: rgba(102, 51, 255, 0.1);
  margin: 0 0 1em;
  border: 1px solid #33cccc;
  border-radius: 3px;
  display: flex;
  flex-flow: row wrap;
  justify-content: flex-end;
}
```

Our last block of code of particular interest is this section, within the large screen media query:

```
149   .Grid--Holy-grail .aside-1 {
150     order: 1;
151   }
152
153   .Grid--Holy-grail .main {
154     order: 2;
155   }
```

The order property simply specifies the order of each item in our flex container; in this case, we have `.aside-1` and `.aside-2` in position 1 and 2 respectively (not in shot), with the `.main` in the middle at position 2.

There are a lot more properties we can set, depending on our requirements. Take a look at the source code on the original pen. There are plenty of reference sources about flexbox available online—as a start, have a look at Chris Coyier's guide, available at `http://bit.ly/1xEYMhF`.

Let's move on. We've explored some examples of what is possible now, but there is at least one downside with using flexbox. The technology works very well, but can add a fair amount of code and complexity when implementing in a site.

It's time to look for something simpler to use, which doesn't require quite the same effort to implement; enter CSS grid templates! This is still an early technology, with minimal browser support, but is already easier to implement. Let's dive in and take a look in more detail.

Visiting the future

Imagine that we have flexbox as a technique for creating grid layouts, but its design is meant for simpler, one-dimensional layouts; it doesn't work so well if the layout is complicated! Is there an answer, something better, that is designed for the job?

Fortunately there is; I am of course referring to a relatively new technology, named CSS Grid Layout. Support for this is minimal for now, but this is likely to change. In a nutshell, it provides a simpler way to create grids in a browser, without the plethora of options we saw with flexbox.

The downside of using CSS Grid Layout as a technology is that support for it has yet to hit mainstream; it is supported in IE11/Edge, but only under the `-ms-` prefix. Opera, Firefox, and Chrome offer support, but all require a flag to be enabled to view the results:

Source: `CanIUse.com`

Leaving aside the concerns about support for a moment, it is easy to see why CSS Grid Layout will take off as a technique. The whole concept has been designed to simplify how we reference cells, rows, and columns; if we compare with flexbox, it is more straightforward to apply styles using CSS Grid Layout than with flexbox.

If you would like to learn more about CSS Grid Layout, then have a look online. This article by Chris House explains it well: `http://bit.ly/2bMGlDp`.

To see how it compares, let's dive in and build a simple demo to illustrate some images in a grid layout.

Implementing a basic gallery grid

For our next demo, we're going to make use of an example created by the developer Rachel Andrew, at `http://codepen.io/rachelandrew/full/LGpONE/`; we'll be replacing the images with ones from Flickr, depicting pictures of my favorite European town, Bruges. No, it's not to do with the lace, before you ask: good food, fine wine, great atmosphere, stunning chocolates for sale…what more could you ask for, I wonder?

But I digress. Before we get into creating our code, there are a couple of points we must bear in mind:

- This demo is cutting edge, it won't work in all browsers, and for some, it requires enabling support within the browser. Take care, it is perfectly okay to enable the flag, but make sure you get the right one:

- We have to restart Google Chrome in step 1, so make sure you only have the flags page displayed at the start of the demo.

Without further ado, let's make a start on our demo:

1. We'll begin by enabling support in Google Chrome for CSS Grid Layout. To do so, browse to `chrome://flags` and search for **Experimental Web Platform features**. Click on the enable button to activate it, then hit the blue **Relaunch Now** button at the bottom of the page to relaunch Google Chrome.
2. With support enabled, go ahead and extract a copy of `gridtemplate.html` from the code download that accompanies this book; save it to the root of our project area.
3. In a new text file, add the following styles. We'll go through them in blocks, beginning with some initial styling for our images and labels:

```
body {
  font-family: helvetica neue, sans-serif;
}

img {
  max-width: 100%;
  border-radius: 10px;
}
```

4. Next up comes the rules needed to set our container; note that the only style used that relates to our grid is `box-sizing`, which we set to `border-box`:

```
.wrapper {
  list-style: none;
  margin: 0;
  padding: 0;
```

```
}

.wrapper li {
  box-sizing: border-box;
  padding: 1em;
  min-width: 1%;
}
```

5. The real magic starts to happen in a set of media queries; we begin with assigning wrapper as our grid container, then set the column and row layout of our grid:

```
@media screen and (min-width: 500px) {
  .wrapper {
    display: grid;
    grid-template-columns: 1fr 1fr;
  }

  .wrapper li:nth-child(1) {
    grid-column: 1 / 3;
  }
}
```

6. In our second query, we set individual styles for our grid wrapper and list items, this time for 640px or greater:

```
@media screen and (min-width: 640px) {
  .wrapper {
    grid-template-columns: 1fr 1fr 1fr;
  }

  .wrapper li:nth-child(1) {
    grid-column: 2;
    grid-row: 1 / 3;
  }

  .wrapper li:nth-child(2) {
    grid-column: 3;
    grid-row: 2;
  }

  .wrapper li:nth-child(3) {
    grid-column: 1;
    grid-row: 1;
  }

  .wrapper li:nth-child(4) {
    grid-column: 3;
    grid-row: 1;
```

```
  }
  .wrapper li:nth-child(5) {
    grid-column: 1;
    grid-row: 2;
  }
}
```

7. Save the file as `gridtemplate.css`, within the `css` subfolder of our project area.
8. Try previewing the results in a browser; if all is well, we should see results similar to this screenshot:

Okay, granted. It's probably not what you might expect in terms of styling, but this demo isn't about making it look pretty, but the basic grid effect. There are nonetheless some important concepts that are good to understand, so let's dive in and explore what took place in our demo in more detail.

Exploring what happened

Earlier in this chapter, we explored how flexbox can be used to create a grid layout; if we were to compare CSS styling, it is easy to see that on balance, we need to provide more styling when using flexbox than using CSS Grid Layout.

The only styling attribute that we've used in our core styles is `box-sizing`, which we set to `border-box`. Nothing else has been used at this point—all of our CSS Grid Layout styles have been set within two media queries.

Our first media query sets the `.wrapper` class as our grid container. Note that we've only need to set it once, as it will cascade through to larger viewports that are `500px` or greater in size.

Once the grid container is assigned, we then specify the grid columns for our template – the `1fr` value assigned represents the fraction of free white space in the grid around each cell's content (hence the `fr` unit). We then finish up by specifying `grid-row` or `grid-column` in both media queries – these values define a grid item's location within the grid; these two terms are shorthand for `grid-row-start`, `grid-row-end`, `grid-column-start` and `grid-column-end` respectively.

For a more detailed explanation of how these terms are used in creating grids, refer to the Mozilla Developer Network articles available at `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout`.

Taking it further

Learning a new technology is like putting on new clothes; at some point, we will outgrow those clothes, or they no longer present the appeal that attracted us to them at the time of purchase.

It's at times like this we need to progress onto something more advanced or with additional functionality, otherwise our development will come to a standstill! Thankfully, there are literally dozens of options available online that we can explore—one might be forgiven for thinking that there are too many and where does one start?

A great starting point is a responsive framework such as Bootstrap or Unsemantic; these have been made to improve the usability and help speed up the process of development. These frameworks were introduced with the aim of providing a grid or foundation for rapid prototyping of the various mobile functionalities, layouts which allow the designers and developers to better make use of their development time.

This is just one part of what is available to help you along, let's briefly cover a few ideas that might serve as somewhere to start:

- **Bootstrap** is downloadable from `http://getbootstrap.com/`, this veteran grid system was first created by Facebook, before becoming a standalone product in its own right.
- If you want to explore something that is more than just a framework, then **Responsive Grid System** might be an option; it's available from `http://www.responsivegridsystem.com/`, with a SASS version available for those who use that CSS preprocessor.
- Instead of simply downloading a framework, how about generating one online? For this, try **Responsify.it** (`http://responsify.it`) and Gridpak.com (`http://gridpak.com`) as possible contenders.
- We used a minimal Grid system earlier in the form of Gridfy, there are others available, if this one is not to your liking. As an example, try Gridly, which can be downloaded from `http://ionicabizau.github.io/gridly/example/`.
- It's worth noting that not every Grid system is available as a standalone—some form part of a component library. A good example is **Formstone**; its grid system is available from `https://formstone.it/components/grid/`. For those of you who use the Less CSS preprocessor, this grid comes with a version that can be used with this tool.
- Staying with the theme of component libraries, why not have a look at **Metro UI**? This library, available from `http://metroui.org.ua/grid.html`, even has backing from Microsoft; it does require jQuery though!
- Some of you might have heard of the 960.gs grid system, which was available a few years ago – it has been replaced by **Unsemantic**, which is available from `http://unsemantic.com/`.
- We covered the use of flexbox as a technology for creating grid-based layouts; as a start point, why not have a look at the PureCSS library? This is available at `http://purecss.io`; it's a good example of using flexbox to produce clean layouts without too much fuss.

As developers, this is one area of responsive design where we are spoilt for choice; the great thing about open source software is that if a framework we choose isn't right, then we can always try another! To help us make the decision, there are a few questions we can ask ourselves:

- Do you need a version that works with a CSS preprocessor? Although preprocessed CSS is a superset of standard CSS, there are grid systems available that are specifically built from a preprocessing technology such as SASS or PostCSS. This is easier than trying to convert finished CSS into something that can be compiled by our processor.

- How complex is your site? Is it a single page *calling card* affair, or something substantially more complex? There is clearly no point in burdening a simple site with a complex grid arrangement; equally if we're building a complex site, then our chosen grid system must be up to par.
- Is browser support an issue? If we can forgo support for some of the older browsers (and particularly below IE8), then choosing a CSS-only option is preferable to one dependent on using jQuery. The same principle applies if we already have to use more than just the occasional external resource. There is no need to add in a plugin if using CSS is sufficient.
- Does your site need to make use of UI components which need to be styled using a themed library? If so, check the library; it may already have a grid system built in that we can use.

The key here is that we shouldn't simply choose the first available option to us, but carefully consider what is available and pick something that satisfies our requirements where possible. Any styling can of course be overridden—the trick here is to choose the right one, so that overriding is minimal or not required for our site.

Summary

Constructing the layout grid for any site is key critical to its success; traditionally, we may have done this first, but in the world of responsive design, content comes first! Throughout the course of this chapter, we've covered a few topics to help get you started, so let's take a moment to recap what we have learned.

We kicked off with an introduction to flexible grid layouts, with a mention that we may have to change our design process to facilitate creating responsive grids. We then moved onto to explore the different types of layout we can use, and how responsive layouts compare to these different types.

Next up, we began on the most important part of our layout—setting the available viewport; this controls how much is visible at any one point. We covered the need to set a viewport directive in our code, so that content is correctly displayed; we examined how not providing the directive can have a negative impact on the appearance of our content! In addition, we covered some of the additional properties and units of value we can use, along with balancing the viewport size against user experience.

We then moved onto exploring the benefits of flexible grid layouts, before taking a look at how they work in more detail; we then created a simple demo using a prebuilt grid system available from the Internet.

Moving on, we then took a look at using flexbox as a technology; we explored it through a simple demo, before dissecting some of the issues with using flexbox. We then saw how a replacement is in the works. We rounded out the chapter with a demo to explore how it can be activated today, and that it is simpler to develop solutions for it once it becomes a mainstream standard.

Now that we have our layout in place, it's time to move on. We need to start adding content! It's assumed that text would be added by default, but what about media? How do we make it responsive? We'll answer these questions, and more, in the next chapter, when we take a look at adding responsive media to our pages.

3

Adding Responsive Media

A picture paints a thousand words...

A key element of any website is a visual content; after all, text will become very repetitive and dull, without adding some form of color!

Adding media not only gives color to a site, but can serve as a vital tool to show potential customers what a product looks like or how we should use it. In fact, sales can go up based purely on being able to see a product being demonstrated. With the advent of mobile devices, it is more important that we not only add media, but also ensure it works well on a range of different devices.

Throughout the course of this chapter, we will explore different ways of adding media to our pages, and see how easy it is to make it respond to any changes in available screen size. In this chapter, we will cover the following topics:

- Understanding the basics of adding images using `<picture>`
- Exploring alternatives to adding images
- Making video and audio content responsive
- Adjusting text to fit automatically on the screen

Curious? Let's get cracking!

Making media responsive

Our journey through the basics of adding responsive capabilities to a site has so far touched on how we make our layouts respond automatically to changes; it's time for us to do the same to media!

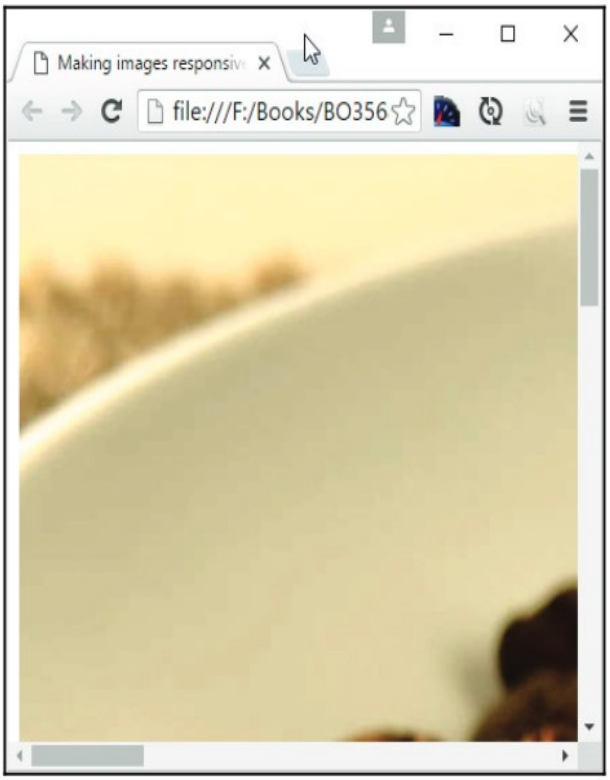
If your first thought is that we need lots of additional functionality to make media responsive, then I am sorry to disappoint; it's much easier, and requires zero additional software to do it! Yes, all we need is just a text editor and a browser; I'll be using my favorite editor, Sublime Text, but you can use whatever works for you.

Over the course of this chapter, we will take a look in turn at images, videos, audio, and text, and we'll see how with some simple changes, we can make each of them responsive. Let's kick off our journey, first with a look at making image content responsive.

Creating fluid images

It is often said that *images speak a thousand words*. We can express a lot more with media than we can using words. This is particularly true for websites selling products; a clear, crisp image clearly paints a better picture than a poor quality one!

When constructing responsive sites, we need our images to adjust in size automatically. To see why this is important, go ahead and extract `coffee.html` from a copy of the code download that accompanies this book and run it in a browser. Try resizing the window. We should see something akin to this:



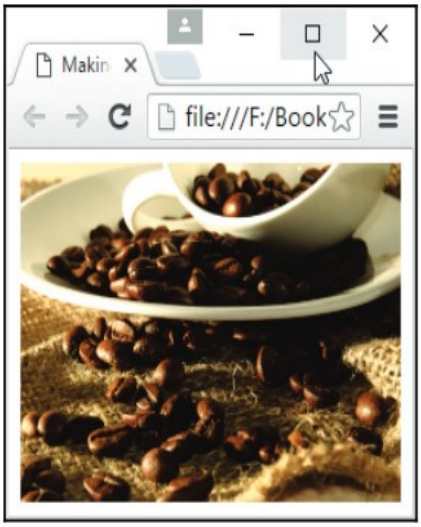
It doesn't look great, does it? Leaving aside my predilection for nature's finest bean drink, we can't have images that don't resize properly, so let's take a look at what is involved to make this happen:

1. Go ahead and extract a copy of `coffee.html` and save it to our project area.
2. We also need our image—this is in the `img` folder; save a copy to the `img` folder in our project area.
3. In a new text file, add the following code, saving it as `coffee.css`:

```
img {
   max-width: 100%;
   height: auto;
}
```

4. Revert to `coffee.html`. You will see line 6 is currently commented out; remove the comment tags.
5. Save the file, then preview it in a browser. If all is well, we will still see the same image as before, but this time try resizing it.

This time around, our image grows or shrinks automatically, depending on the size of our browser window:



Although our image does indeed fit better, there are a couple of points we should be aware of when using this method:

- Sometimes you might see `!important` set as a property against the `height` attribute when working with responsive images; this isn't necessary, unless you're setting sizes in a site where image sizes may be overridden at a later date
- We've set `max-width` to `100%` as a minimum; you may need to set a `width` value too, to be sure that your images do not become too big and break your layout

This is an easy technique to use, although there is a downside that can trip us up—spot what it is? If we use a high-quality image, its file size will be hefty—we can't expect users of mobile devices to download it, can we?

Don't worry though – there is a great alternative that has quickly gained popularity among browsers; we can use the `<picture>` element to control what is displayed, depending on the size of the available window. Let's dive in and take a look.

Implementing the <picture> element

In a nutshell, responsive images are images that are displayed their optimal form on a page, depending on the device your website is being viewed from. This can mean several things:

- You want to show a separate image asset based on the user's physical screen size. This might be a 13.5-inch laptop or a 5-inch mobile phone screen.
- You want to show a separate image based on the resolution of the device or using the device-pixel ratio (which is the ratio of device pixels to CSS pixels).
- You want to show an image in a specified image format (WebP, for example) if the browser supports it.

Traditionally, we might have used simple scripting to achieve this, but it is at the risk of potentially downloading multiple images or none at all, if the script loads after images have loaded, or if we don't specify any image in our HTML and want the script to take care of loading images.

We clearly need a better way to manage responsive images! A relatively new tag for HTML5 is perfect for this job: `<picture>`. We can use this in one of three different ways, depending on whether we want to resize an existing image, display a larger one, or show a high-resolution version of the image. The recommended way to approach this is:

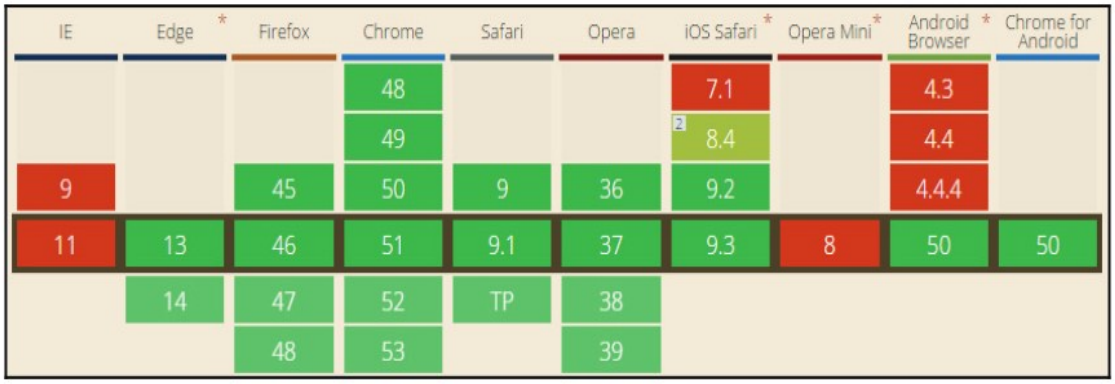
1. `srcset` attribute
2. `sizes` attribute
3. `picture` element

We'll explore all three in detail; let's start with implementing the `srcset` attribute in a standard `<img>` tag, before moving on to using the `<picture>` element.

Using the srcset attribute

A key benefit of using the `<picture>` element is using the srcset attribute to select any one of several images, based on whether we want to display higher resolution versions or different sizes of the same image in different viewports.

Support for this in browsers is very good, with only Opera Mini and up to IE11 not wanting to join the party:



Source: `http://caniuse.com/#search=srcset`

To make use of this `srcset` attribute, we need to avail ourselves of sufficient different images, then specify what should be displayed at the appropriate trigger point, as shown in this example, based on defining the `device-pixel` ratio:

```
<img src="small-image.png" srcset="small-img.png 1x, med-img.png 2x, lrg-
img.png 3x">
```

Here, the `src` attribute acts as fallback image for those browsers that do not support `srcset` (although support is now very good!). The `srcset` attribute allows us to specify different images to use, either based on the device-pixel ratio or the available viewport:

```
<img src="default.png" srcset="small.png 256w, med.jpg 511w, large.jpg
640w">
```

In this case, our fallback is `default.png`; however, if the browser being used supports `srcset`, then it will display `small.png` at `256w` or `med.png` at `511w`. If, however, we wanted to change the size and use different images based on the available viewport, then we would have to add an extra attribute—`sizes`. It's easy enough to configure, so let's pause for a moment to see what this means in practice.

Exploring the sizes attribute

When adding pictures as part of the content on a responsive site, the images may each take up 100% of the element width, but the content itself doesn't always take 100% of the width of the window! For example, we might set each image element to 100% width (so they fill their parent containers), but that the overall content on screen only fills 50% of the available screen width.

To overcome this, we need to know the URLs for the various images to use, along with the width of each image resource; we can't get this from standard markup in the page layout, as images start downloading before CSS is applied.

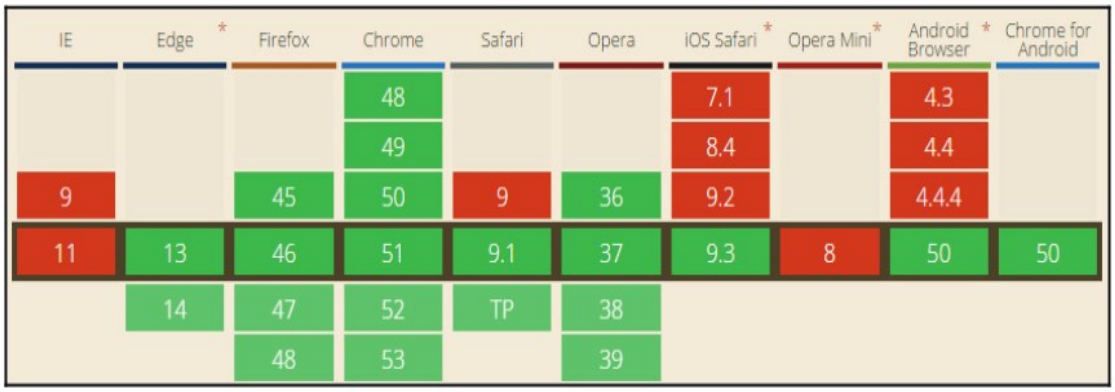
Instead, we can simply set suitable widths within our HTML code using the `srcset` attribute and suitable width descriptors. This is a little controversial for some, as it starts to blur the divide between HTML markup and the CSS presentation layer. This aside, let's take a look at an example of how we can set up the code:

```
<img src="img/orchid.jpg" sizes="50vw" srcset="img/orchid.jpg 200w,
img/orchid-2x.jpg 400w, img/orchid-hd.jpg 600w">
```

In this excerpt, we set a default of 50% or half of the viewport width; the browser can then select the appropriate image to display, depending on the available width.

Manipulating the HTML5 <picture> element

We've covered two key parts of making images responsive, but to bring it all together, we can use the HTML5`<picture>` element, which has garnered good support in most browsers:



Source: `http://caniuse.com/#search=picture`

The `<picture>` element uses this syntax:

```
<picture>
   <source media="(min-width: 60rem)" sizes="(max-width: 500px) 50vw, 10vw"
   src="high-res-image-2x.png 145w ">
   <source media="(min-width: 35rem)" src="med-res-image.png">
   <source src="low-res-image.png">
   <img src="fallback-image.png" alt="Non-supported browser.">
   <p>Text to display</p>
</picture>
```

In this extract, we've tied together all of the various attributes we can use with the `<picture>` element; in this case, we've specified media queries (one of `60rem` and another of `35rem`), and that if our viewport is only 50% or less (indicated by the `50vw` value in the code), we display the normal images; if it is higher, then we display the high-definition images (as specified by using the `100vw` value).

We will explore how this works in more detail, in *Exploring what happened*, later in this chapter.

Putting it all together

Now that we've seen all three elements in use, let's pull them together and create a simple demo that automatically adjusts which image to use, based on the available viewport. For simplicity, we will concentrate just on the image, but there is nothing stopping us from developing this further into a full-sized page!

Let's make a start. For this demo, I would strongly recommend using Google Chrome if you have it installed; its device mode is perfect for this task!

1. From a copy of the code download that accompanies this book, go ahead and extract copies of the four landscape images, and save them to the `img` folder at the root of our project area.

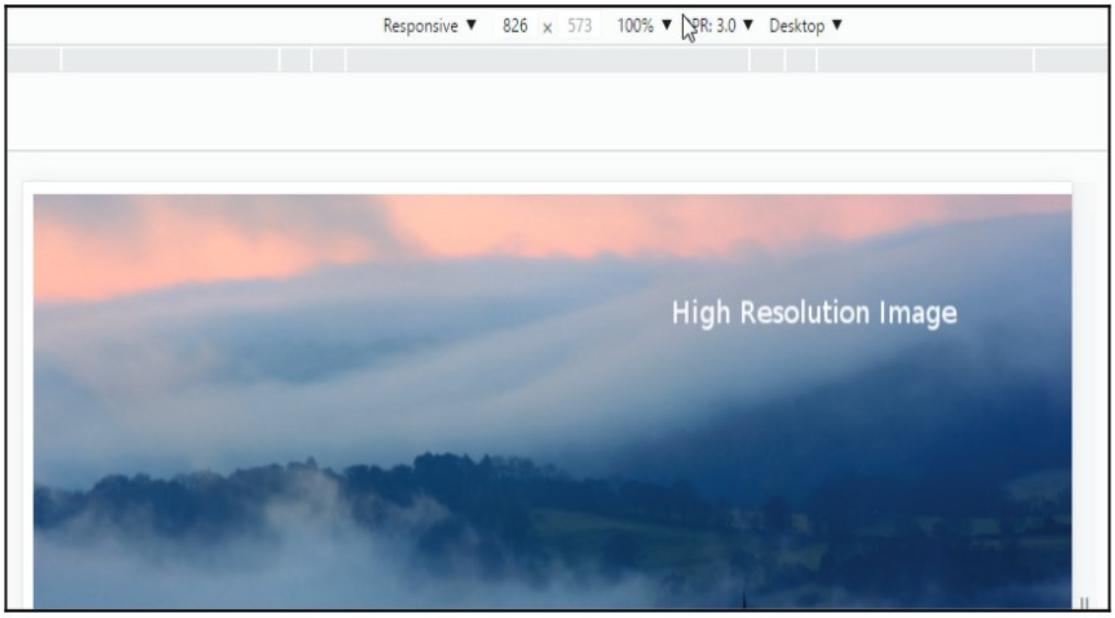
2. Next, fire up your text editor, and add the following code:

```
<!DOCTYPE html>
<html>
<head>
<title>Adding responsive images with the <picture> element</title>
</head>
<body>
```

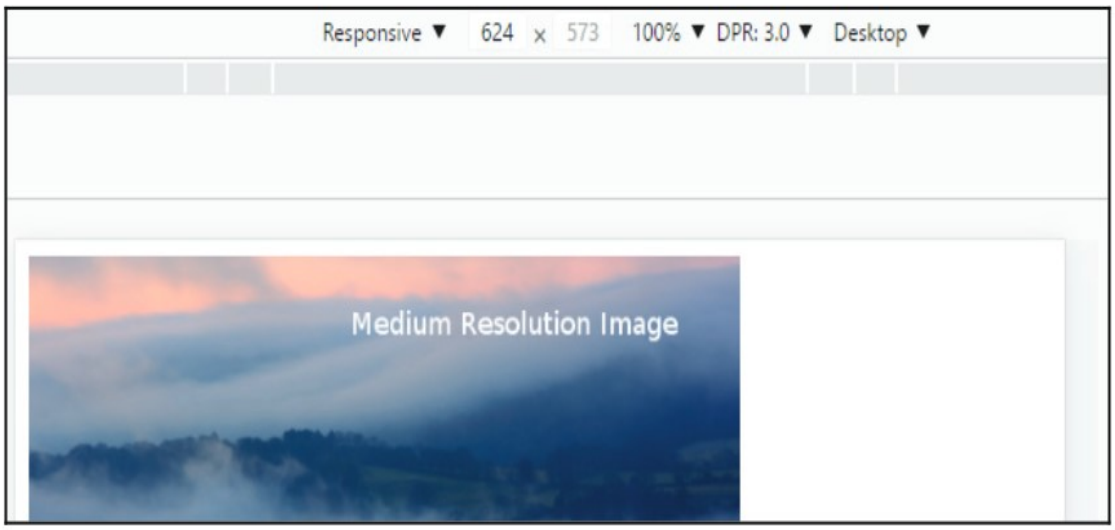
```
    <picture>
      <source media="(min-width: 800px)" sizes="(max-width: 1000px)
      100vw" srcset="img/high-res-image.png 738w">
      <source media="(max-width: 799px)" sizes="(max-width: 600px)
      100vw" srcset="img/med-res-image.png 738w">
      <img src="img/fallback-image.png" alt="Human">
    </picture>
  </body>
  </html>
```

3. Save this as `pictureelement.html` at the root of our project folder.
4. Go ahead and preview the results of the file in Google Chrome (or another browser if preferred). Make sure you switch on that browser's device/responsive mode.

If all is well, we should see the image flip between two similar versions; to identify which is which, I've added the words **High Resolution Image** on one, and **Medium Resolution Image** on the other image used:



This is the same image, but this time using the medium resolution version:



Although this demo may look simple at face value, it is deceptive. We have the power to construct some complex statements, which can automatically select an image based on a number of criteria! It's important to understand how this demo works, as a basis for using it for complex examples. Let's take a moment to explore it in more detail.

Exploring what happened

If we take a look through our picture element demo, the code used may initially look complex, but is simpler than it looks! The key to it is understanding each part the `<source>` statements and how they interact with each other. Let's tackle the first one:

```
<picture>
  <source media="(min-width: 800px)" sizes="(max-width: 1000px) 100vw"
  srcset="img/high-res-image.png 738w">
  ...
</picture>
```

In this one, we're specifying `high-res-image.png` as our source image; this will only be displayed when our browser window is showing a minimum width of `800px`. The size of the image will either go to a maximum of `1000px` or `100vw`—the latter equivalent to 100% width of the available viewport space. The `738w` against the image is just the width of the image specified in the code (1w unit is equal to 1px, so our image is 738px wide).

Moving onto the second source statement, we find it shows a similar set up, but this time the media query is limited to a maximum width of `799px`, and that the size of the image will go to `600px` or the full width of the viewport, depending on its current size:

```
<picture>
  ...
    <source media="(max-width: 799px)" sizes="(max-width: 600px) 100vw"
    srcset="img/med-res-image.png 738w">
    <img src="img/fallback-image.png" alt="Human">
  </picture>
```

To finish off the `<picture>` element, we specify `fallback-image.png` as our fallback for those browsers that have yet to support this element in HTML5.

We've only scratched the surface of what is possible with the `<picture>` element; for more details, take a look at the site maintained by the Responsive Images Community Group, hosted at `https://responsiveimages.org/`.

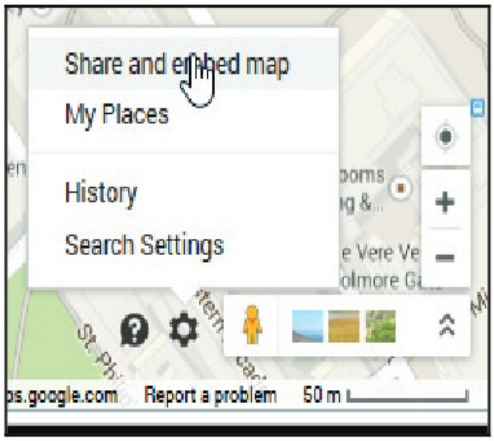
Creating a real-world example

We've explored the theory behind making images responsive with a couple of useful techniques; it's time we got practical! The basis for our next demo is going to look at making a responsive map using Google Maps.

Responsive maps, I hear you ask? Surely this should come automatically, right? Well no, it doesn't, which makes its use a little awkward on mobile devices. Fortunately, we can easily fix this; the great thing about it is that it only requires a little additional CSS:

1. Let's make a start by browsing to `http://maps.google.com`, then entering the zip code of our chosen location; in this instance, I will use Packt's UK office, which is B3 2PB.

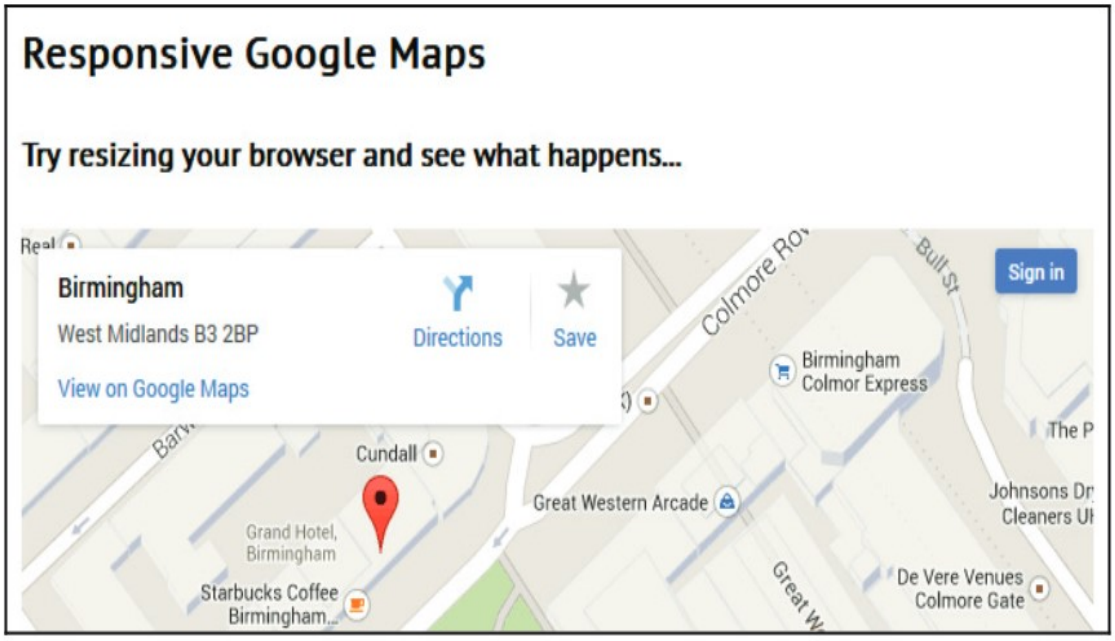
2. Click on the cog, then select **Share and embed** map, as shown in this screenshot:



3. In the dialog box that appears, switch to the Embed map tab, then copy the contents of the text field starting with `<iframe src=...`
4. In a copy of the code download that accompanies this book, extract a copy of `googlemaps.html` in your favorite text editor, and add the `<iframe>` code in between the google-maps div tags.
5. Next, add the following CSS styling to a new file, saving it as `googlemaps.css`:

```
#container { margin: 0 auto; padding: 5px; max-width: 40rem; }
.google-maps { position: relative; padding-bottom: 60%;overflow:
hidden; }
.google-maps iframe { position: absolute; top: 0; left: 0; width:
100% !important; height: 100% !important; }
```

If all is well, we will see a Google Maps image of Birmingham, with Packt's office marked accordingly:



At this point, try resizing the browser window. You will see that the map resizes automatically; the CSS styling that we've added has overridden the standard styles used within Google Maps to make our map responsive and accessible from any device we care to use.

Taking things further

Throughout the course of this chapter, we've followed the principle of using just a browser and text editor to construct our code. This, of course, included not downloading anything that was core to creating our examples (save for media and content).

There will be times though when this approach is not sufficient, we may find we need to avail ourselves of additional support to get a job done. Our overriding question should always be to check that we really need it, and that we're not just being lazy! If when answering that question, we do find that need additional help is needed, then there are a number of sources you can try out, to help take things further:

- It goes without saying, but there will come a time when we need to resort to using jQuery (`http://www.jquery.com`) to help within our development. The state of responsive design is such that we should only need jQuery to make it easier to reference elements in the DOM, and not to make images or content responsive!
- The Responsive Images site hosted at `https://responsiveimages.org/`. We covered it briefly at the end of the `<picture>` demo, but it's worth pointing it out again. It's a useful compendium of material to help understand and use the `<picture>` element.
- The developer Scott Jehl created a polyfill for `<picture>`, to extend support to those browsers that do not support it natively; you can download it from `https://scottjehl.github.io/picturefill/`.
- Are you in need of a responsive carousel? There are plenty available online, but one which I've found to work well, is ResponsiveSlides, available from `http://responsiveslides.com/`. Granted, the project is a few years old, but this particular plugin keeps things nice and simple, which is very much in keeping with the theme for this book!
- A good example of where responsive capabilities are already present is in using the SVG image format. These are effectively vector-based images that we can manipulate using CSS; the key benefit though is that SVG images can automatically grow or shrink, with no loss of quality. Browser support for the format is excellent, although IE (and Edge) both have a couple of quirks that require attention when using these browsers (for more details, see `http://caniuse.com/#feat=svg`).
- Another idea to try is with responsive icons. A good example that is worth a look is the FontAwesome library, available from `http://fontawesome.io/`. These will resize equally as well. In this instance, they would be perfect for smaller images, such as credit card icons or shopping baskets on e-commerce sites.

- Taking things even further afield, how about support for the WebP image format? Yes, this is one that hasn't gained huge support yet, with it being limited to Chrome and Opera at the time of writing. However, when used with the `<picture>` element, it shows off a nice trick:

```
<picture>
    <source type="image/webp" srcset="retina-image.webp 2x,
    image.webp 1x" />
    <img srcset="retina-image.jpg 2x" src="image.jpg"
    alt="an image" />
</picture>
```

- In our example, the browser will check for WebP support, if it can support it, it will display the appropriate image in WebP format, depending on what device-pixel-ratio is supported on the device being used. If WebP isn't supported, then it will fall back to using JPEG (although this could equally have been a different format such as PNG).

There are certainly things we can do, once we've become accustomed to working with responsive images, and want to graduate away from just using HTML5 and CSS3. It is important to note though, that there are a number of projects operating online that aren't listed here.

The main reason for this is age—support for responsive images was patchy for a while, which meant a number of projects appeared to help provide support for responsive images. Support for the `<picture>` and associated elements is getting better all of the time, which reduces some of the attraction of these older projects; it is worth considering whether it is sensible to use them, or if the impact of not using them can be mitigated by changes to the user experience.

Okay, let's move on; time to get a little animated, I think! Alright, that was a terrible lead in to our next topic, given that we're going to explore making videos responsive. Over the next few pages, we'll see that although some of the same principles apply here, there are some bumps along the way, which might impact our journey.

Making video responsive

Flexible videos are somewhat more complex than images. The HTML5 `<video>` maintains its aspect ratio just like images and therefore we can apply the same CSS principle to make it responsive:

```
video {
   max-width: 100%;
   height: auto !important;
}
```

Until relatively recently, there have been issues with HTML5 video—this is mainly due to split support for the codecs required to run HTML video. The CSS required to make a HTML5 video is very straightforward, but using it directly presents a few challenges:

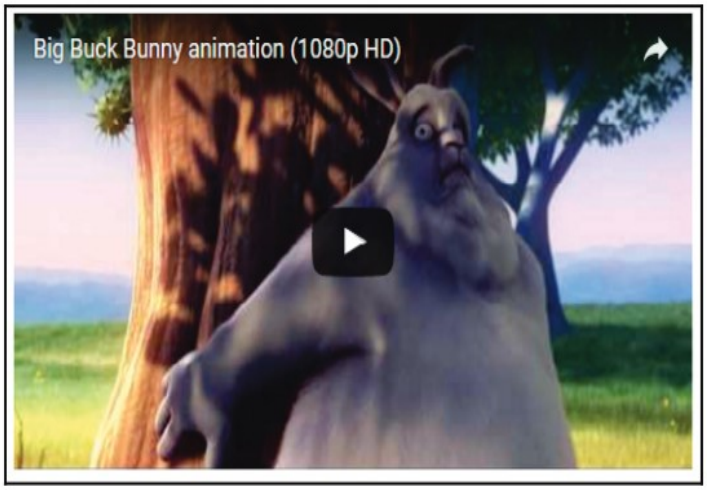
- Hosting video is bandwidth intensive and expensive
- Streaming requires complex hardware support in addition to video
- It is not easy to maintain a consistent look and feel across different formats and platforms

For many, a better alternative is to host the video through a third-party service such as YouTube. There is a caveat that they would be in control of your video content; if this isn't an issue, we can let them worry about bandwidth issues and providing a consistent look and feel; we just have to make it fit on the page! This requires a little more CSS styling to make it work, so let's explore what is involved.

Embedding externally hosted videos

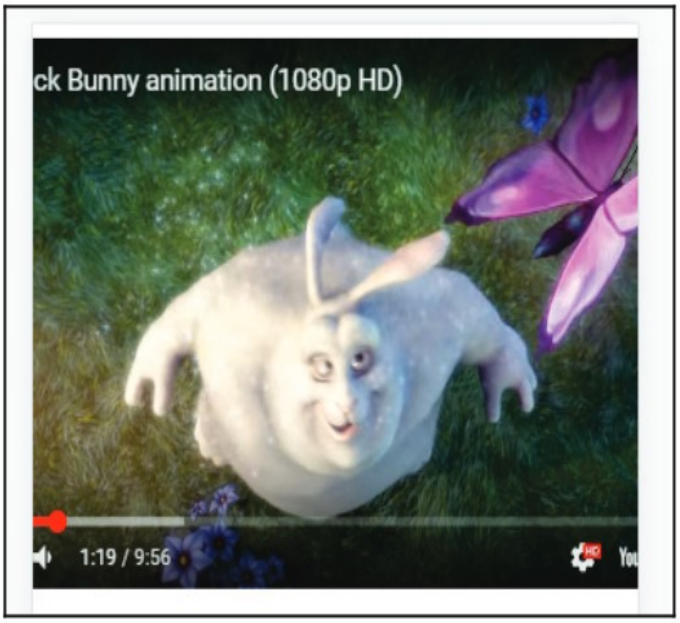
To embed those videos, we need to use iframes, which unfortunately do not maintain aspect ratio by default; we can work around this with a CSS solution by Thierry Koblentz.

Let's for argument's sake say that we have a YouTube video, such as this one, titled *The Big Buck Bunny*, by the Blender Foundation:



(c) Blender Foundation | www.bigbuckbunny.org

Looks okay, doesn't it? Granted, we can't immediately tell it is a video from YouTube, but this next screenshot clearly shows it is:



Hold on; that doesn't look right, does it? The screenshot was taken in Google, but set to emulate the screen estate of a Galaxy S5 mobile phone, but it clearly shows that the video is not responsive.

To see this in action, extract a copy of `youtube.html` from the code download that accompanies this book to our project area, then run it in a browser. Activate your browser's responsive mode (or device mode, depending on browser) and resize the screen to 360px by 640px. You will soon see how it doesn't resize well!

How do we fix this?

The trick is to create a box with a proper aspect ratio, say 4:3 or 16:9 (through zero height and bottom padding in %), and then fit the video and stretch it inside the box up to the box dimensions by positioning it absolutely with respect to the box. The bottom padding acts as the width that helps to maintain the aspect ratio. Let's alter our code to fix this issue:

1. In `youtube.html`, add this link within the `<head>` section:

```
<link rel="stylesheet" type="text/css" href="css/youtube.css">
```

2. Further down, alter the code as shown:

```
<div class="video-box-wrapper">
  <iframe width="560" height="315"
  src="https://www.youtube.com/embed/XSGBVzeBUbk" frameborder="0"
  allowfullscreen></iframe>
</div>
```

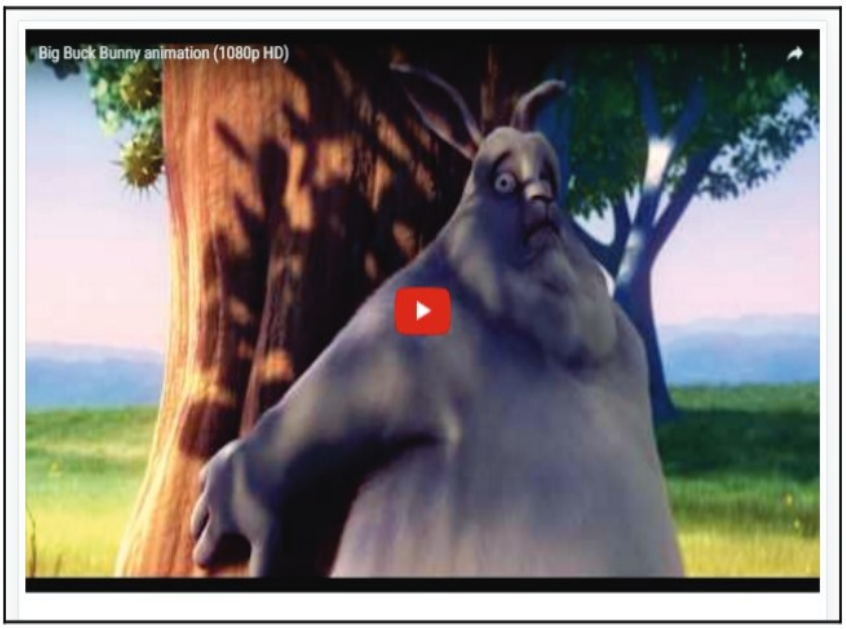
3. Save the file. Switch to a new file, then add the following code and save it as `youtube.css` within the `css` subfolder of our project area:

```
.video-box-wrapper {
  padding-bottom: 5.25%; height: 0; position: relative;padding-top:
  1.875rem; overflow: hidden; }

.video-box-wrapper iframe,
.video-box-wrapper object,
.video-box-wrapper embed { position: absolute; left: 0; top: 0;
width: 100%; height: 100%; }
```

A word of note—setting `height: 0` ensures the element is present within the DOM so that older browsers can format the inner box properly.

4. Save the file, revert back to your browser, and re-enable its responsive (or device) mode if it is not already switched on.
5. Try previewing the results now; if all is well, we should see something akin to this. It uses the same Galaxy S5 size settings, but this time zoomed in to 150% for clarity:



This looks much better! With some simple styling, we have the best of both worlds; we can let YouTube do all the heavy lifting while we concentrate on making our video available from our site on multiple devices. The CSS we used forces all of the video content to the full width of the `.video-box-wrapper` container, which in turn is positioned relative to its normal position. We then add `56.25%` to the bottom to maintain the classic 16:9 aspect ratio and provide a little extra padding at the top so it doesn't appear to go off screen!

Question: How did we arrive at 56.25%? This is simply 9 divided by 16 (the aspect ratio), which is 0.5625 or 56.25%.

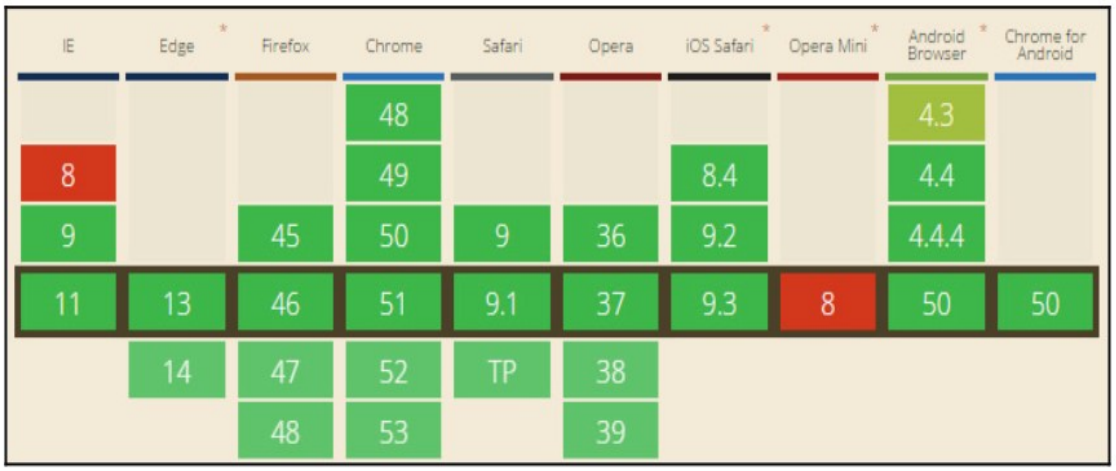
There will be occasions, though, when we have to host our own videos; this might be for controlling visibility or preventing adverts from being played, if we were to host it externally. To achieve this, we can use the now current HTML5 `<video>` element to render content on a page; let's take a look and see how this works in action.

Introducing the new HTML5 video element

If hosting videos on an external source is not possible, then we must host locally; for this, we can use the native HTML5 video tag, which looks something like this:

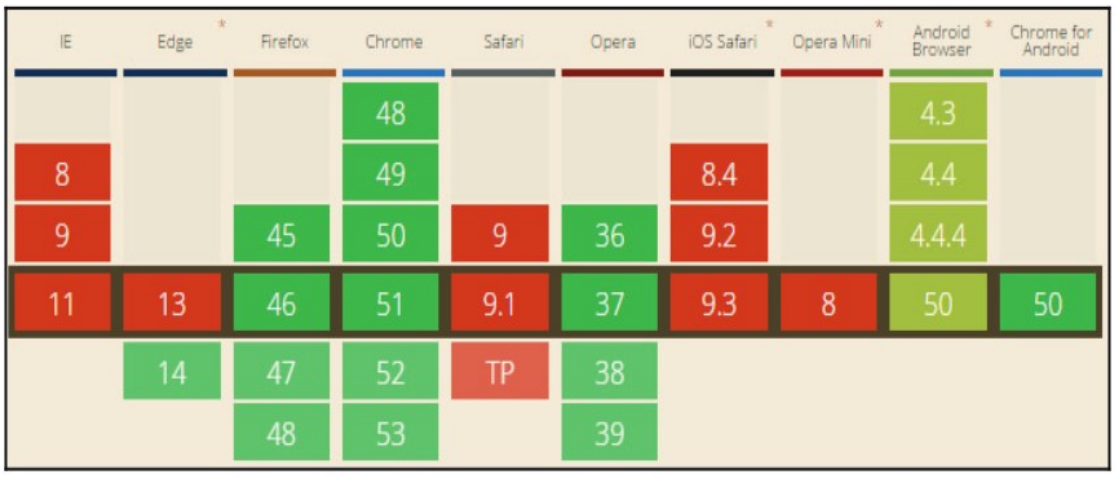
```
<video controls>
  <source src="video/bigbuckbunny.webm" type="video/webm">
  <source src="video/bigbuckbunny.mp4" type="video/mp4">
</video>
```

In the past, codec support for the HTML5 element has been split across each platform; in 2015, Firefox added support for H.264 to its browsers across all platforms, which goes a long way to rationalize support for HTML5 video. At present, support for the two formats (MP4 and WebM) is good, but not 100% across all browsers – this screenshot indicates the current state of play for desktop and mobile browsers for the MP4 format:



Source: CanIuse.com

In contrast, support for the WebM format is not quite so complete:



Source: CanIuse.com

In reality, the only format we need to worry about using is MP4; we can use WebM format if desired. If we do so, then it must come first in the `<source>` list; otherwise, the browser will pick the first available supported format (in this case, MP4) and not use WebM!

Before continuing, I would strongly recommend making sure you have Google Chrome or Firefox installed – WebM video will work in IE9 or above, but not without adding codec support for the format!

Now that we've been introduced, let's move on and put it into practice, with a simple demo to illustrate how the `<video>` element works in action.

Embedding HTML5 video content

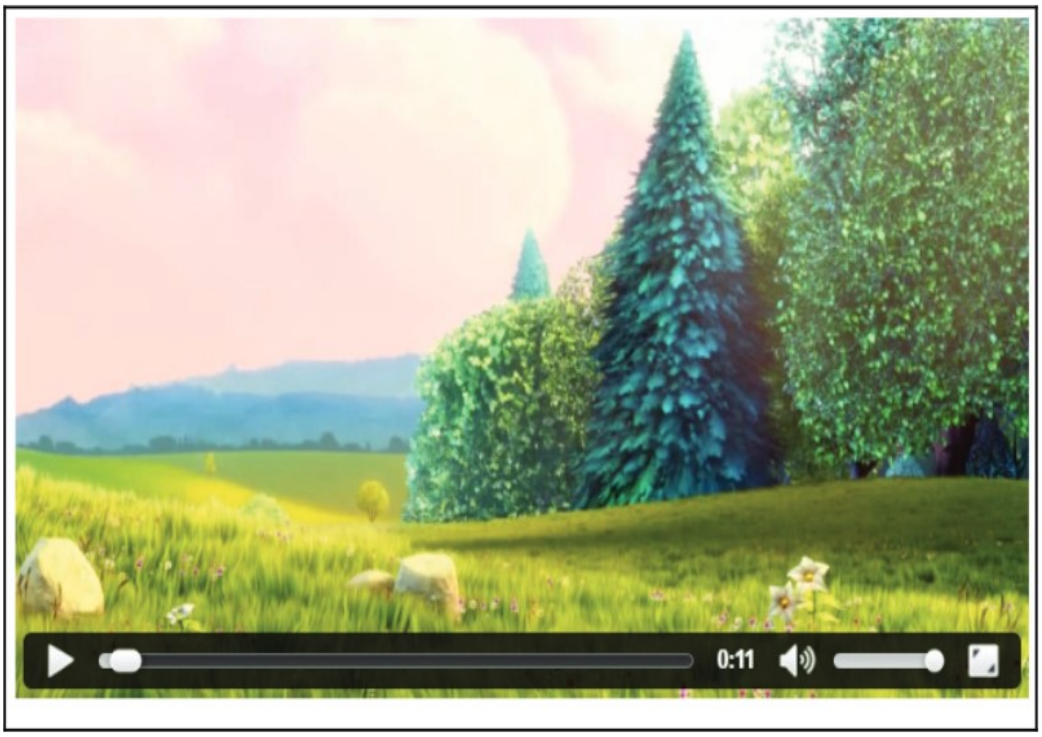
If our requirements are such that we have to host a video ourselves, then implementing it using the HTML5 standard tags is very easy; it consists of setting any number of different sources within the `<video>` tags so that we can play the same video using the supported format for that browser. Let's dive in and take a look at how we do it:

1. We'll start by extracting copies of the following, from the code download that accompanies this book – the `video` folder and `html5video.html`. Save them to the root of our project folder.

2. In a new file, go ahead and add these styles; save the file as `html5video.css` in the `css` subfolder of our project area:

```
video {
   max-width: 100%;
   /* just in case, to force correct aspect ratio */
   height: auto !important;
}
```

3. Try previewing the results in a browser. If all is well, we should see something akin to this (screenshot taken from Chrome):



The result looks perfect—the question is, which version of our video is being used? One way to find out is to right-click on the video, while it is still playing, then click on **Save video as...**. If all is well, we should see a **Save As** dialog box open, ready to save the WebM format if we're using FireFox, Chrome, or Opera; otherwise it will be MP4 (if using IE).

Exploring what happened

The real question, though, is not so much how does it all work, but if it is responsive?

The answer is yes; our use of the HTML5`<video>` tags mean that we can select any number of different video formats to use; the browser will simply choose the first available that it is able to play. The order is critical though:

```
<video controls>
    <source src="video/bigbuckbunny.webm" type="video/webm">
    <source src="video/bigbuckbunny.mp4" type="video/mp4">
</video>
```

If we were to swap it around so that MP4 is first, then WebM will be ignored for almost all browsers, as MP4 can be played in almost all of the browsers!

The real magic lies not in the use of a specific video format, but in the CSS rule we've created:

```
video {
   max-width: 100%;
   /* just in case, to force correct aspect ratio */
   height: auto !important;
}
```

Our use of percentage values means that it will automatically scale up or down when our window is resized; the maximum size of the video is constrained by the video's dimensions, not other elements on screen. Of course, we may decide to host the video within a parent container; the video will fill that container, but the parent may only stretch over part of the site.

Building a practical example

If you spend any time on the Internet, it's possible you've come across sites where the developer hasn't used images as a background, but *video*.

This isn't entirely new as a concept; it's been around for a couple of years now. If done with care, it can work very well. It's a perfect candidate for making full-size video as a responsive background. The great thing about it is that we can make it responsive using pure CSS. That's right, no need for any JavaScript.

For our next demo, we'll take a break from creating content. This time around, we'll run the demo from the code download that accompanies this book, and take a look at the code in more detail later in the demo. We'll be using videos from the Big Buck Bunny project, created by the Blender Foundation as our background; over this, we'll overlay a simple block of sample text, generated using the Lorem Ipsum generator.

To see this in action, go ahead and run the `fullscreen .html` demo from a copy of the code download that accompanies this book. If all is well, you will see the video play behind a simple `<div>` with text:

If we take a look at our code in more detail, we can see the video element in use; it's been set to autoplay, with sound muted and a poster (or placeholder) image set. The real magic, though, lies in the CSS styling, so let's explore this in more detail.

Exploring what happened

The trick that makes our video work is in this code. We need to set two media queries with 16:9 aspect ratio (one as a min-aspect-ratio, another as the max) so that our video displays correctly on the screen:

```
60  @media (max-width: 47.9375rem) {
61    .fullscreen {
62      background: url('../img/videoframe.jpg') center center / cover no-repeat;
63    }
64
65    .fullvideo {
66      display: none;
67    }
68  }
69
```

When resizing it though, it will show white space. We fix that by setting negative margins, which makes the viewport much wider, and allows us to center the content on screen:

```
51  @media (min-aspect-ratio: 16/9) {
52    .fullvideo {
53      height: 300%;
54      top: -100%;
55      width: 300%;
56      left: -100%;
57    }
58  }
```

A key point to note is the values used for `height`, `top`, `left`, and `width`; although these seem extreme, they are required to help center the video on screen when viewing the content with a `16/9` aspect ratio set.

Perfect! Our video plays well. We can see the content without too much difficulty. Everything should be good, surely? Well, yes and no; concepts such as background video are not without their risks; it's important to understand where things might fall over if we're not careful. Let's pause for a moment and consider some of the potential traps that might upset the proverbial apple cart, if we're not careful with our video.

Exploring the risks

In our previous example, we explored the concept of adding video as background content. It's a fashion that has taken off within the last couple of years, and provides an interesting effect, that is different to seeing the standard images we might otherwise see!

It's not without a certain element of risk though; there are a few pointers we must consider, when adding video as the background content:

- It's possible to add video, but we shouldn't just add it because we can—any video we add using this method must amplify the site's overall message.
- Any video added will likely be set to autoplay, but the sound must be muted by default—if possible, it shouldn't have any sound at all.
- Does our video fit with the site brand, tone, color palette, and so on? There is no point building a killer site, only to ruin it with a rubbish video.
- Costs are something we must consider; it can be expensive to host video content, so it must be compressed as much as possible to keep file sizes down, and in a suitable format that works on multiple devices, including mobile.
- Our video should not be too long; we must strike a balance between making it too long and not long enough so that it does not feel too repetitive.
- Accessibility is a key pointer; it must be of sufficiently high contrast so as to make the text overlay legible.
- Our video may look good, but what about performance? Your customers will not thank you if you produce a lightning fast site, but slow it down with a large, poorly optimized video as a background; they will very likely vote with their feet!
- The compatibility technique we've used doesn't work on IE8, so a static placeholder must be included as a fallback; in the event the browser we use doesn't support HTML5 video or its attributes (such as autoplay, for mobiles).

Even though we have some clear pointers that should be considered, it should not stop us from using this effect; I'm one for pushing out the boundaries of what is possible, provided we do it well!

Making audio responsive

Question—we've worked on making videos responsive, but what about audio content?

Well, we can apply similar principles to the HTML5 `<audio>` element; instead of setting a specific width, we can use max-width and set a percentage figure to control how wide it displays on the screen.

The code to achieve this is very simple, and should by now be familiar—let's take a look at what is involved:

1. For this demo, we need to avail ourselves of suitable files; for licensing reasons, you won't find any in the code download that accompanies this book, unfortunately! One way to achieve this is to take a copy of an iTunes file (normally in `.m4a` format), then use an online service such as Media.io (`http://media.io/`) to convert it to the right formats. You will need to convert to both MP3 and OGG formats, before continuing with this demo.
2. Assuming we now have the right files, go ahead and extract a copy of `audioelement.html` from the code download that accompanies this book, and save it to the root of our project area.
3. Next, at the root of our project area, go ahead and create a new folder called `audio`; into it, save copies of the audio files you either have or created from step 1.
4. In a new file go ahead and add the following code, saving it as `audioelement.css` in the `css` subfolder at the root of our project area:

```
audio {
  max-width: 100%;
  width: 800px;
}
```

5. Try previewing the results of our work in a browser—if all is well, we should see something akin to this screenshot:

At first glance, it may not look special, but then the `<audio>` element isn't meant to look anything out of the ordinary! The key here though is when we resize the browser window; we've set a max width value of `100%`, but have constrained this by setting an upper limit of `50rem` in the width attribute. No matter how many times we resize our window, the audio player will fill the full width, but not go any wider than `50rem`.

Unlike the `<video>` element, we can't resize the height using just CSS; to do this requires overriding the `<audio>` element with jQuery, which is out of the scope of this book.

Let's move on and put our new-found knowledge to the test to create a practical example—how about making a video fullscreen, and responding to changes in the browser viewport automatically? Setting up video using this technique is always fraught with controversy, but I'm not one to shy away from a challenge, so without further ado, let's dive in and see why we must step carefully when using video at fullscreen.

Taking things further

Throughout the course of this book, we've concentrated on using the core technologies of HTML5 and CSS3; in many cases, this is all we need, but there will come a time when we have to use other technologies to help fulfill a task, as we've outgrown the art of possible with plain CSS and HTML code.

Thankfully, there are lots of options available online to help with making videos responsive, and to take our skills. It goes without saying though that we should always ask ourselves if our need for another library is because the realities of life mean that we can't achieve our task without using it or if we've simply become too lazy!

If indeed we do need to download and use an additional library, there are a few good options to try out, which include:

- **FluidVids**: It is available from `http://toddmotto.com/labs/fluidvids`; the library is a couple of years old, but may be worth a look.
- **responsiveVideo**: It is downloadable from `http://cbavota.bitbucket.org/responsive-video/`. This has been around for a couple of years, so may not work so well.
- **Embed Responsively**: It is hosted at `http://embedresponsively.com/`, and will return appropriate embed code for any of the major video hosting companies, such as YouTube; it's also responsive to boot!

- **FitVids.js**: This plugin, available from `http://fitvidsjs.com` and built by Chris Coyier of CSS Tricks' fame, may be worth a look, although it hasn't been updated for at least 2-3 years.
- **MediaElement.js**: It is available from `http://mediaelementjs.com`, and is a great library that works with both the `<video>` and `<audio>` elements; it allows us to override the standard element and customize it to our requirements using jQuery and CSS. There are plenty of examples of doing this online, along with suitable tutorials on how to achieve a custom look and feel to any player we skin using jQuery.

A small word of caution—a number of the video plugin libraries for jQuery haven't been updated for some time; you may well find that they no longer work properly with more recent versions of jQuery. This isn't necessarily a bad thing, as support for the HTML5 `<video>` and `<audio>` elements is now excellent; this renders many of these libraries surplus to requirements!

Some of you may ask why we need to use jQuery to skin either HTML5 audio or video players; many of the individual elements are not accessible using plain CSS, and need JavaScript to expose those elements before styling them with CSS.

Phew! We're almost through this part of the journey, but before we move onto taking a look at using media queries in the next chapter, there is one more part of making responsive content; how about the text we have on our sites? It might not immediately strike you as being one we would associate with videos and text (at least in the context of making content responsive), but all will shortly become clear.

Making text fit on screen

When building sites, it goes without saying but our designs clearly must start somewhere—this is usually with adding text. It's therefore essential that we allow for this in our responsive designs at the same time.

Now is a perfect opportunity to explore how to make our text fluid and fill the available space. Although text is not media in the same way as images or video, it is still content that has to be added at some point to our pages! With this in mind, let's dive in and explore how we can make our text responsive.

Sizing with em units

When working on non-responsive sites, it's likely that sizes will be quoted in pixel values; it's a perfectly acceptable way of working. However, if we begin to make our sites responsive, then content won't resize well using pixel values; we have to use something else.

There are two alternatives: em or rem units. The former is based on setting a base font size that in most browsers defaults to 16px; in this example, the equivalent pixel sizes are given in the comments that follow each rule:

```
h1 { font-size: 2.4em; }      /* 38px */
p  { line-height: 1.4em; }    /* 22px */
```

Unfortunately, there is an inherent problem with using em units; if we nest elements, then font sizes will be compounded, as em units are calculated relative to its parent. For example, if the font size of a list element is set at 1.4em (22px), then the font size of a list item within a list becomes 30.8em (1.4 x 22px).

To work around these issues, we can use rem values as a replacement, these are calculated from the root element, in place of the parent element. If you look carefully throughout many of the demos created for this book, you will see rem units being used to define the sizes of elements in the demos.

Using rem units as a replacement

The rem (or root em) unit is set to be relative to the root, instead of the parent; it means that we eliminate any issues with compounding at a stroke, as our reference point remains constant, and is not affected by other elements on the page.

The downside of this is support—rem units are not supported in IE7 or 8, so if we still have to support these browsers, then we must fall back to using pixel or em values instead. This of course raises the question: should we still support these browsers, or is their usage of our site so small as to not be worth the effort required to update our code?