HTML5 Responsive Design

- A Brief Review of the Box Model
- Laying Out the Experience Section with CSS Positioning
- Laying Out the Details Section with Table Display
- Laying Out the Photos Section with Flex Box
- Adding a Media Query for the Middle Width Photos Section
- Fluid Grids

Quiz 1 Project 1

Lesson 7: **Responsive Images**

- Problems with Images in Responsive Design
  - Inspecting Your Web Page Resources and Network Use
  - Strategies for Smart Image Loading
  - Using JavaScript to Load Image Content
- Solutions Coming in the Future with New HTML Support
  - The Srcset Attribute of the <img> Element
  - The <picture> Element
  - Use the data-src Option with JavaScript for Now
  - High Density Pixel (Retina) Displays
  - Setting breakpoints in a world full of different devices

Quiz 1 Project 1

Lesson 8: **Styling Choices for Mobile Devices (Narrow Screens)**

- Mobile
  - Mobile Devices are Optional
  - The Experience Section: Removing the <aside> Element
  - The Details Section
  - Photos
  - Creating Menu Buttons for the Narrow View

Quiz 1 Project 1

Lesson 9: **Responsive Design in the About Page**

- Adding a Page to the Dandelion Tours Site
  - The Structure and Content of the About Page
  - Styling the About Page
  - The Sidebar
  - Adding a Responsive Image Slide Show
  - How CSS Sprites Work

Quiz 1 Project 1

Lesson 10: **Responsive Forms**

- Designing with the User in Mind
  - Creating the Contact Us Page
  - Styling the Form
  - Handling the Style for Required Elements
  - Initializing the Form and Adding Handlers
  - Validating the Form

Quiz 1 Project 1 Project 2

Lesson 11: **Responsive Video**

- Video on the Web
  - Video

Introduction to Responsive Design

Welcome to the O'Reilly School of Technology (OST) HTML5 and Responsive Design course.

Course Objectives

When you complete this course, you will be able to:

- describe what makes a web page responsive.
- create HTML structure for your content appropriate for responsive design.
- use the viewport meta tag to set pages up for mobile.
- use media queries to create different styles for different properties, like browser width.
- load content dynamically based on brwowers properties, like width.
- lay out web pages so they look good and work well on a variety of screen sizes.
- use CSS Sprites to create small icon or button images to increase download efficiency.
- style fonts, images, forms, and video for responsive web pages.

Lesson Objectives

When you complete this lesson, you will be able to:

- use OST's Sandbox and learning tools.
- explain (at a basic level) what a responsive design is.
- read about what to expect in the Responsive Design course.
- use the Developer Tools in your browser.

Before we begin, you need to learn a little about the development environment you'll be using.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you

go completely off the rails.

- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

```
White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add looks like this.

If we want you to remove existing code, the code to remove will look like this.

We may also include instructive comments that you don't need to type.
```

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

```
The plain black text that we present in these INTERACTIVE boxes is
provided by the system (not for you to type). The commands we want you to type look lik
e this.
```

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

```
Gray "Observe" boxes like this contain information (usually code specifics) for you to
observe.
```

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

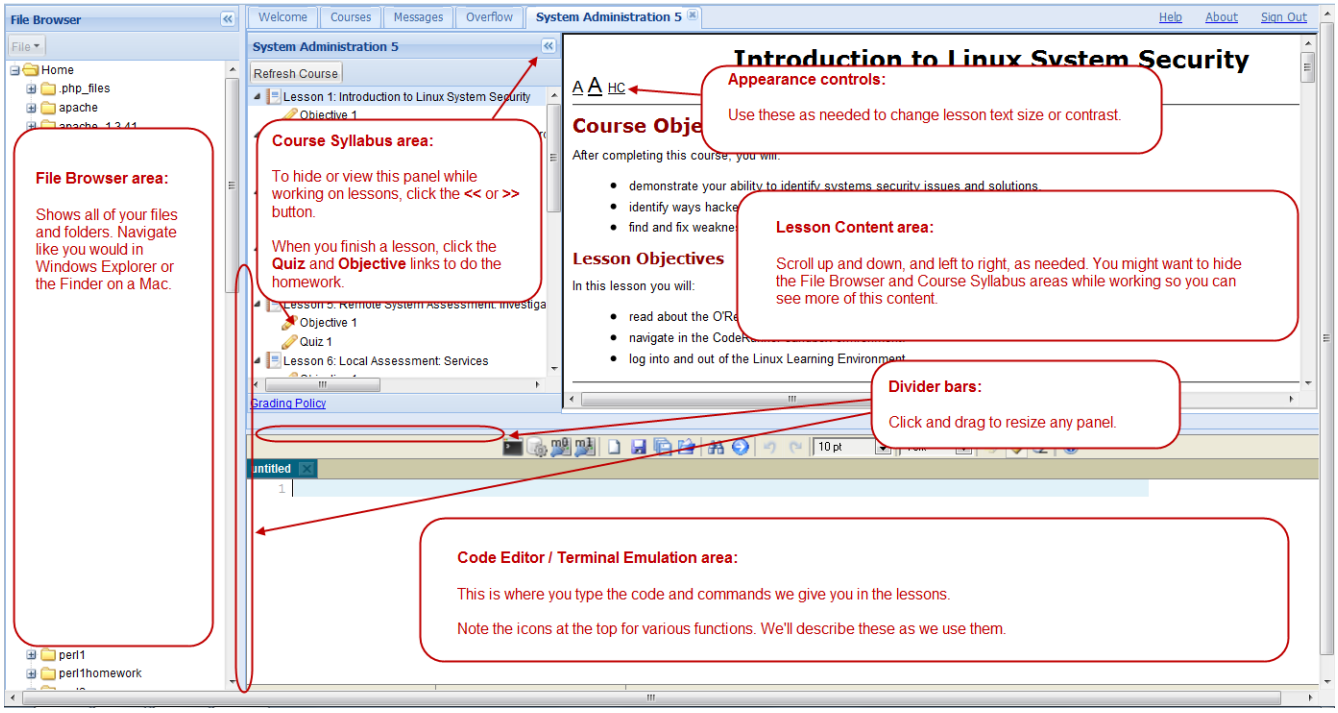
Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's a quick overview of the various areas of the screen:



These videos explain how to use CodeRunner:

File Management Demo

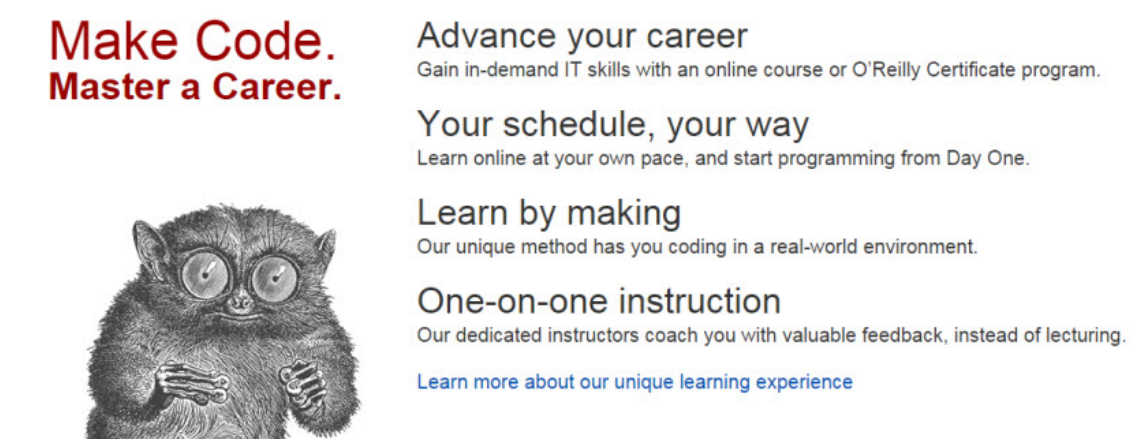
Code Editor Demo

Coursework Demo

Welcome to Responsive Design

A Responsive Site

Have you used a website on your desktop that seems to work equally well on on your big desktop screen, your laptop screen, and even your mobile phone? A site that works well, even if you increase your default font size? If you have, then you've experienced **responsive design**. To see a responsive web design in action, just take a look at the oreillyschool.com website's home page:



The size of the browser used to make this screen shot is 1300 pixels wide by 1275 pixels high (on my regular, non-retina screen). Many desktop users now have computers with large displays (for instance, mine is set to 2560 by 1600 pixels), and websites are beginning to take advantage of all that space by spreading out and filling browsers when they're opened wide. The standard for designing a web page used to be 800 by 600; then, it went to 960 x 800, and now, well, there is no "best" browser width to design for because people are viewing web pages on everything from huge screens to tiny mobile devices.

Go ahead and open a browser window and load the O'Reilly School of Technology home page in your own browser. It might look a bit different from what's shown here (as the page design changes fairly often). Take note of what you see on the screen, so when you follow along with the instructions below, you can see the changes in your own browser window.

Now, try reducing the width of your browser just a bit. Here's the O'Reilly School home page at 1100 pixels wide:

Make Code. Master a Career.

Advance your career
Gain in-demand IT skills with an online course or O'Reilly Certificate program.

Your schedule, your way
Learn online at your own pace, and start programming from Day One.

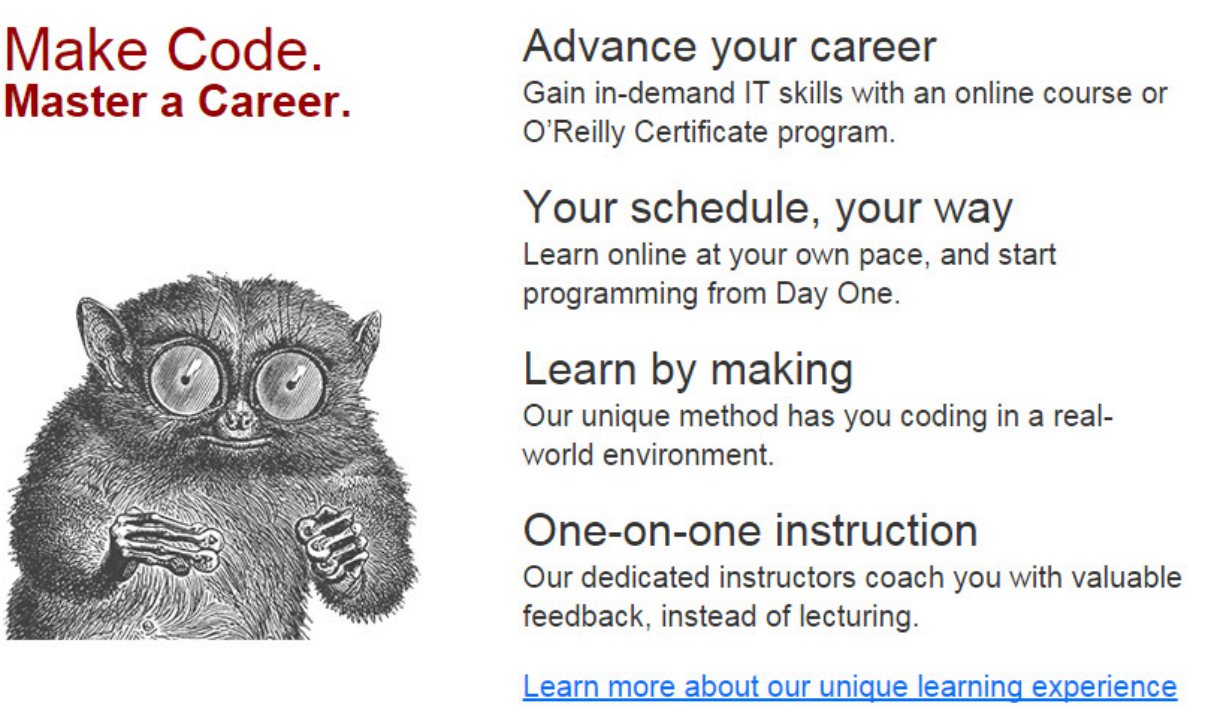
Learn by making
Our unique method has you coding in a real-world environment.

One-on-one instruction
Our dedicated instructors coach you with valuable feedback, instead of lecturing.

Learn more about our unique learning experience

Mostly it looks the same, but notice a couple of subtle differences. The pull quote box that appeared over the main image at the bottom right has disappeared, and the text to the left of the image is now displayed using a slightly smaller font size, so it doesn't overlap the main part of the image. Also, if you scroll all the way to the bottom of the page, you'll see that the footer has been changed from a two-column display with the black section next to the red section, to a one-column display with the black section appearing on top of the red section (not shown in the screen shot).

Reduce the width of your browser again, and you'll see more changes. This screen shot was taken with the browser at about 750 pixels wide:



Now, notice that the menu across the top of the page under the school's name has completely changed. The text links in black have been hidden under a pulldown menu, named (appropriately enough) "Menu," and the phone number has been made smaller and changed to a button, to match the "Log In" and "Live Chat" buttons. Also, the search area has been moved from the far right over next to the menu.

Below that, notice that the image has changed; we no longer see the picture of the student at all, but we also don't see a scroll bar at the bottom indicating there's more to see on the right. In other words, the image is being dynamically resized to fit the space for the image perfectly. In addition, the pull quote that appeared over the image has now completely disappeared, leaving more room for the rest of the text on top of the image.

Below that, notice also that the animal (tarsier) logo has moved under the "Make code, master a career" text. The text to the right of the tarsier is wrapping nicely so it still fits on the page. The elements on the page below also resize nicely to fit on the page, even at this narrower width.

Reduce the width of the browser once more, this time to about the width of a large mobile phone display—in the screen shot below, my browser was set to about 595 pixels wide:

from Day One.

Learn by making

Our unique method has you coding in a real-world environment.

One-on-one instruction

Our dedicated instructors coach you with valuable feedback, instead of lecturing.

Our unique method has you coding in a real-world environment.

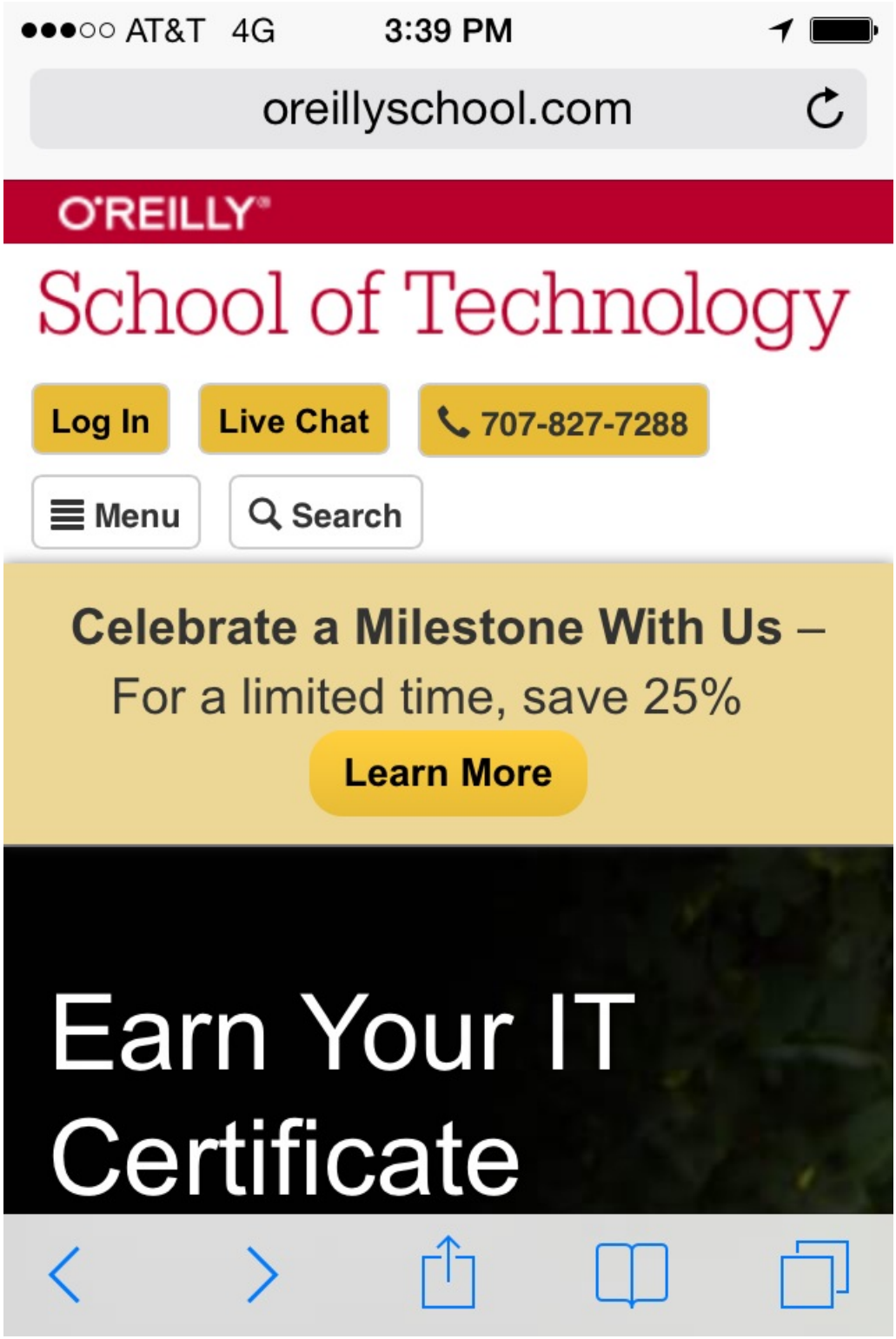
One-on-one instruction

Our dedicated instructors coach you with valuable feedback, instead of lecturing.

Learn more about our unique learning experience

Even more changes! The top part of the page looks about the same, except for a smaller font size. However, below that, the tarsier logo is much smaller and now appears to the right side of the page, and the text items that were on the right now appear below "Make code, master a career" on the left. If you keep scrolling down, you'll notice that some of the items that were appearing next to one another horizontally, now appear above each other. In addition, the footer has once again changed design.

This is, in fact, almost exactly what you'll see on a mobile device if you load the page on your phone. I've done that and included a screen shot below (on an iPhone 4s, with a retina screen):



On the phone, I see a 25% off promo between the menu and the top portion of the page, and the menu at the top now appears on two lines instead of one. That's because, even though my iPhone has a resolution of 640 pixels wide, it's being displayed as if it were 320 pixels wide. Understanding retina displays and screen densities are part of learning responsive design, so we'll be returning to this issue later in the course.

If you're still following along, keep scrolling down on the phone, and you'll see that the rest of the page looks very similar to what you saw when you used your desktop browser at a narrow width. That's because the browser is detecting when the screen size is a narrow width and assumes you're on a mobile device when it's narrower than a certain width.

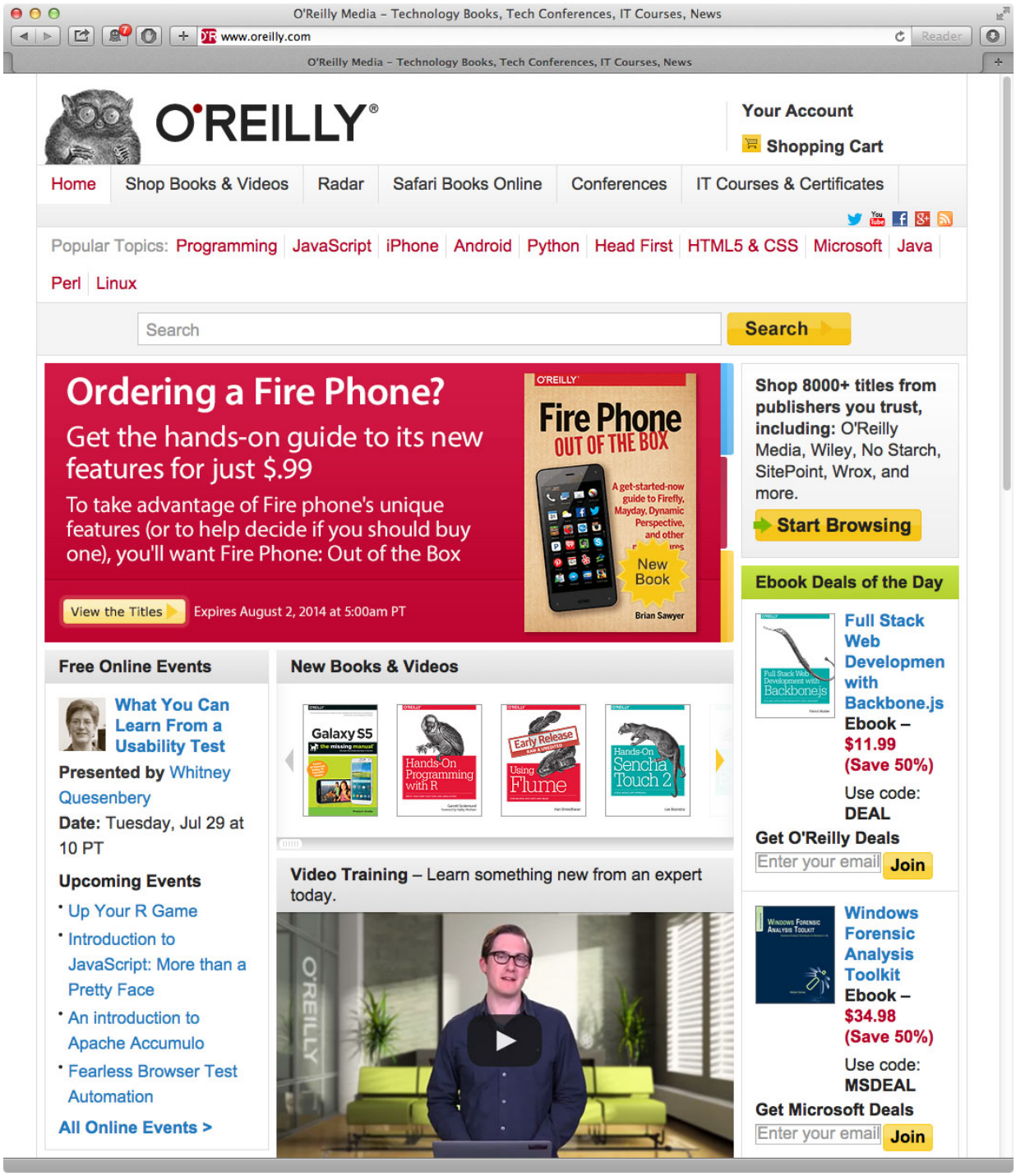
Note

Because not all of you may have access to a mobile device, we'll be testing our designs for mobile using desktop browsers, and for the most part, that will do just fine. However, in the "real world," when you work as a web developer, it's important to be able to test your designs on mobile devices, because there are some features that are different (we'll see a couple of these when we talk about responsive forms, screen densities, and screen orientation) that you just can't test on a desktop. If you do have an internet-connected mobile device handy as you go through this course, don't hesitate to give your designs a try! I'll provide some tips on how to do this later.

A Non-Responsive Site

The O'Reilly School of Technology website is a great example of a responsive web design. There are a few things we could potentially improve on (for example, the forms on another page in the site could be made a bit more mobile friendly), but on the whole, the site works well at a variety of screen sizes and on at least three devices: my desktop computer with a large screen, my laptop, and my iPhone with 4G.

So now let's take a look at another O'Reilly site that isn't quite so responsive: the main oreilly.com home page. At 1075 pixels wide, the page looks fine:



When I reduce the width of my browser to 600 pixels wide, the entire right side of the page is cut off, and I have to use the scroll bar at the bottom to see it:

Popular Topics: Programming | JavaScript | iPhone | Android | Python | Perl | Linux

Search

Free Online Events

What You Can Learn From a Usability Test

Presented by Whitney Quesenbery

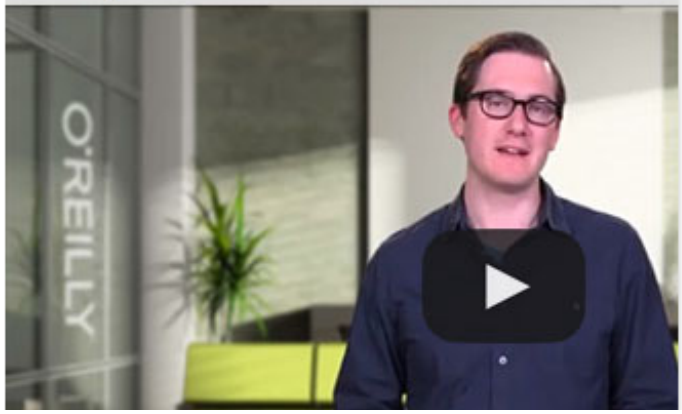
Date: Tuesday, Jul 29 at 10 PT

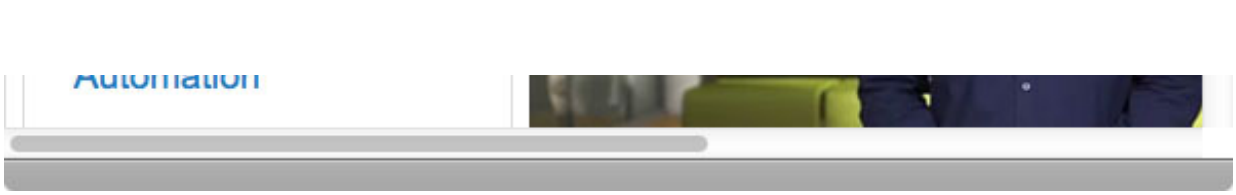
Upcoming Events

- Up Your R Game
- Introduction to JavaScript: More than a Pretty Face
- An introduction to Apache Accumulo
- Fearless Browser Test Automation

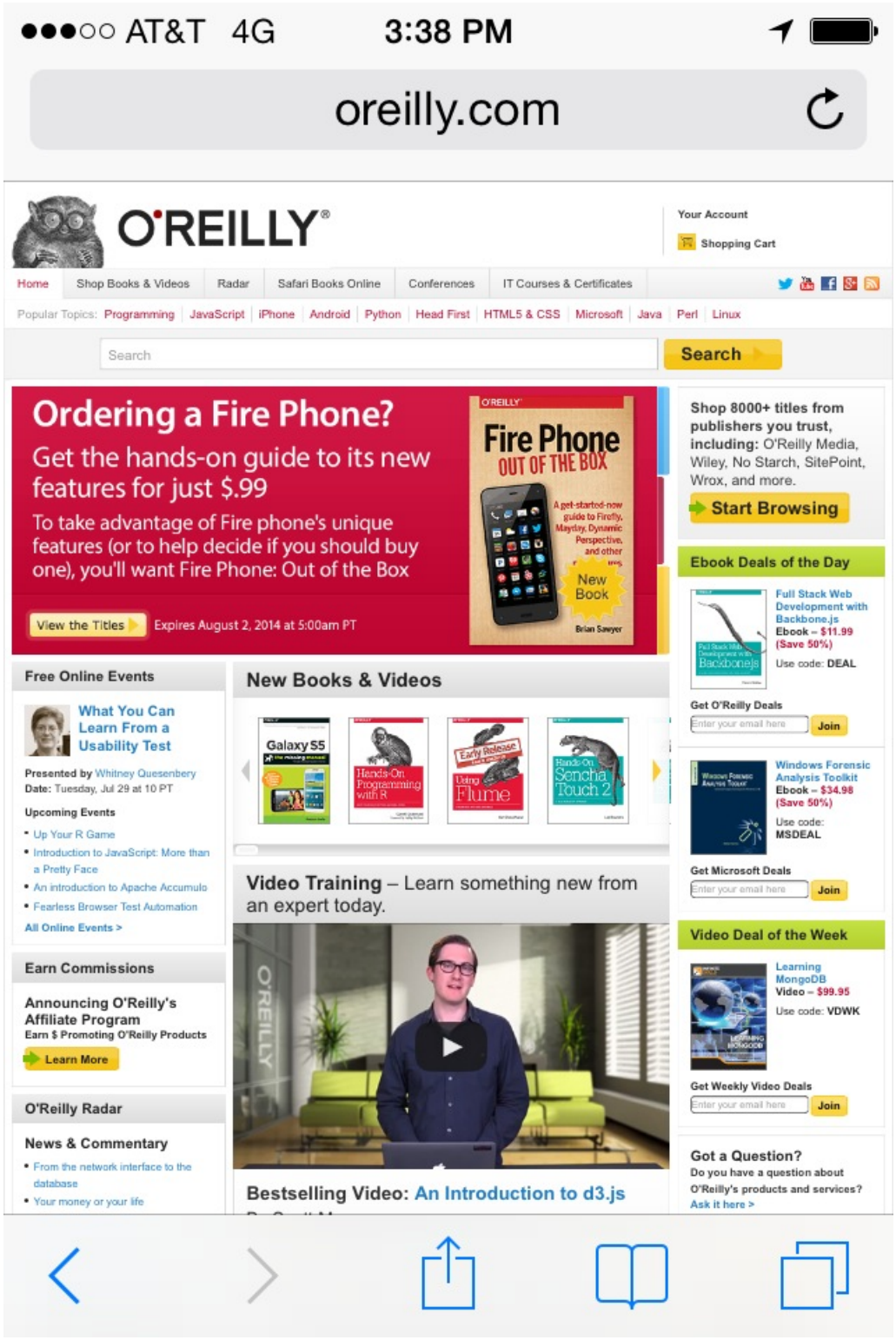
New Books & Videos

Video Training – Learn something new today.





Now, you might think that the page would be cut off in the mobile browser too, but check it out:



On the phone, the page is *scaled* to fit the mobile browser width. This is the default way the mobile browser handles a website on the iPhone. The upside is you can see the entire width of the page without having to scroll sideways. The downside is everything on the page is now tiny! So, if you want to use something on the page, like tap a link or enter text in the search bar, you need to use a gesture to zoom into the page first (unless, of course, you have amazing eyesight and tiny fingers).

The site is an example of a *fixed-width* design which became popular because it allows designers to have

more control over how the page looks on the wide variety of sizes of desktop screens. However, O'Reilly's home page does not "respond" well to a mobile device. The site's still perfectly usable but it's definitely not optimal when displayed on a small screen. That's the downside of a fixed-width design.

What is Responsive Design?

Now that you've seen examples of responsive and non-responsive web pages, let's talk a little more about what responsive design actually *is*, and how it came to be.

Back when the internet first got popular (in the late 1990s), most people just had desktop computers. These computers had small screens compared to today's massive displays, and so website designers got used to designing for a browser size of about 800 pixels in width by 600 pixels in height.

However, desktop displays got bigger, and then laptops came along. So now designers had to decide: do we optimize our website for big desktop displays, or for smaller laptop screens? Some chose to design fixed-width sites that looked the same on both desktops and laptops; others decided to make more *fluid* designs that stretched to fill wider browser windows on bigger screens.

Then, mobile devices exploded onto the market and mobile browsers came along with them. Users could tap links and pinch and zoom to zoom in and out of any area of the page. Even though websites weren't designed for mobile browsers, users could get along okay, but using websites designed for large computer screens, or even laptop screens, was not optimal.

At this point, designers decided they should probably make separate mobile sites for mobile users, designed to fit precisely on smaller screens. So, a designer might create one website for larger screens at 960 pixels wide, and another website for mobile screens at 320 pixels wide (the original width of the iPhone). When users accessed the normal website on a mobile phone, they were redirected to the mobile site automatically instead.

Do you currently use any websites that have a separate site for mobile? As of this writing, Twitter still has a separate site for mobile you can access at mobile.twitter.com. Try it in your browser to see how it compares with the regular twitter.com site.

Having two different sites to maintain wasn't too bad when everyone had an iPhone, but then of course, along came the iPad, the Android phone, tablet computers, and so on. Now, all of a sudden web designers were faced with far too many different-sized screens on too many different devices to have custom web sites for each and every one.

So, at this point it made more sense to start thinking about how to design *one* website that would work across *all* screens: from very wide to very small. That's when responsive design was born.

The term *responsive design* was coined in 2010 by Ethan Marcotte to describe websites that are flexible enough to be displayed on a variety of displays, and use a new (in 2010) feature of HTML and CSS called *CSS media queries* that allows you to select different CSS rules based on features of the display that are detectable by the browser (such as browser width).

So, a *responsive website* is a site that can be viewed and used in browsers on everything from a small mobile device to a large desktop computer, and that uses CSS media queries to select different rules for displaying and laying out content.

Now you might think...okay, but what's so hard about that? Well, it can be trickier than you think! We have to think carefully about how to structure the content for a web page so that it can flow into a large screen and a small screen and still make sense. We have to design our CSS so we can use one set of rules for laying out a page on a big screen, and another for a small screen. We have to think about how images are going to look: an image that's sized for a big screen is going to be way too big for a small screen. We have to think about bandwidth: while small screens on mobile devices can display beautiful images, we are often downloading those images on cellular networks that are much slower than a wifi or direct ethernet connection. There is a *lot* to consider when planning a responsive design. We are going to cover all of these topics, and more, in this course.

Accessing the Console

Before we get on with the rest of the course, you should know how to use the browser console to inspect your web pages. Every modern browser has developer tools that can help you understand how your page is laid out, see the CSS that is applied for a specific element, and compute the size of elements.

In this course, we'll show most examples using the Chrome and Firefox browser tools. Both these browsers have great tools and these are also the most popular browsers on the web, but you should become familiar with multiple browsers and their respective tools for testing and debugging your responsive web designs.

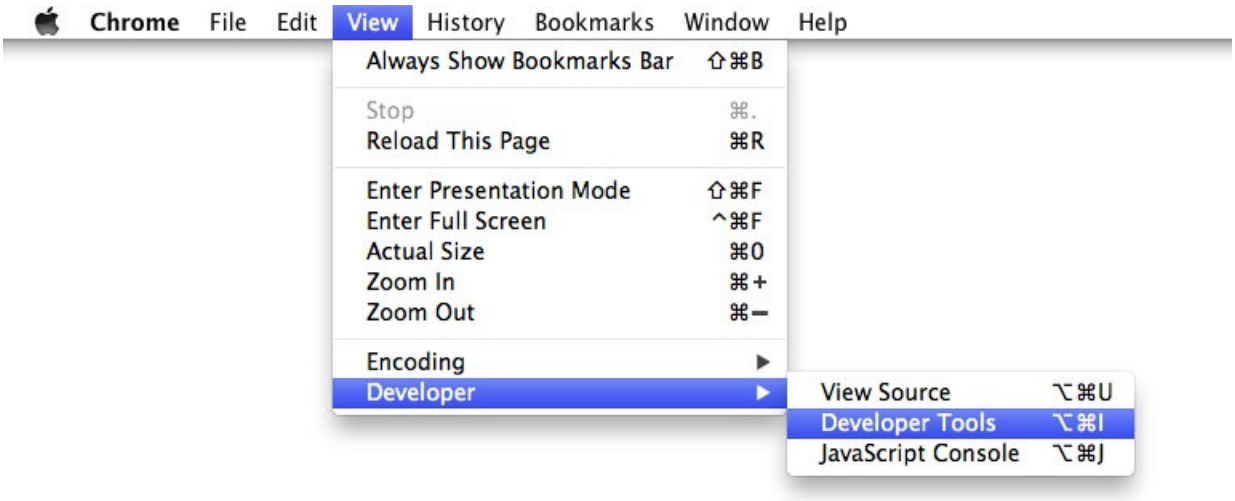
Note

Browsers are continually updated with new versions that include new versions of their tools. So, while the basic functionality of the tools will likely remain the same, your version may look slightly different from what you see in this course. As you become familiar with the different browser tools, you'll be able to figure out how to use each one. For this course, make sure you've updated your browser to the most recent version so you're working with the most up-to-date version of the developer tools.

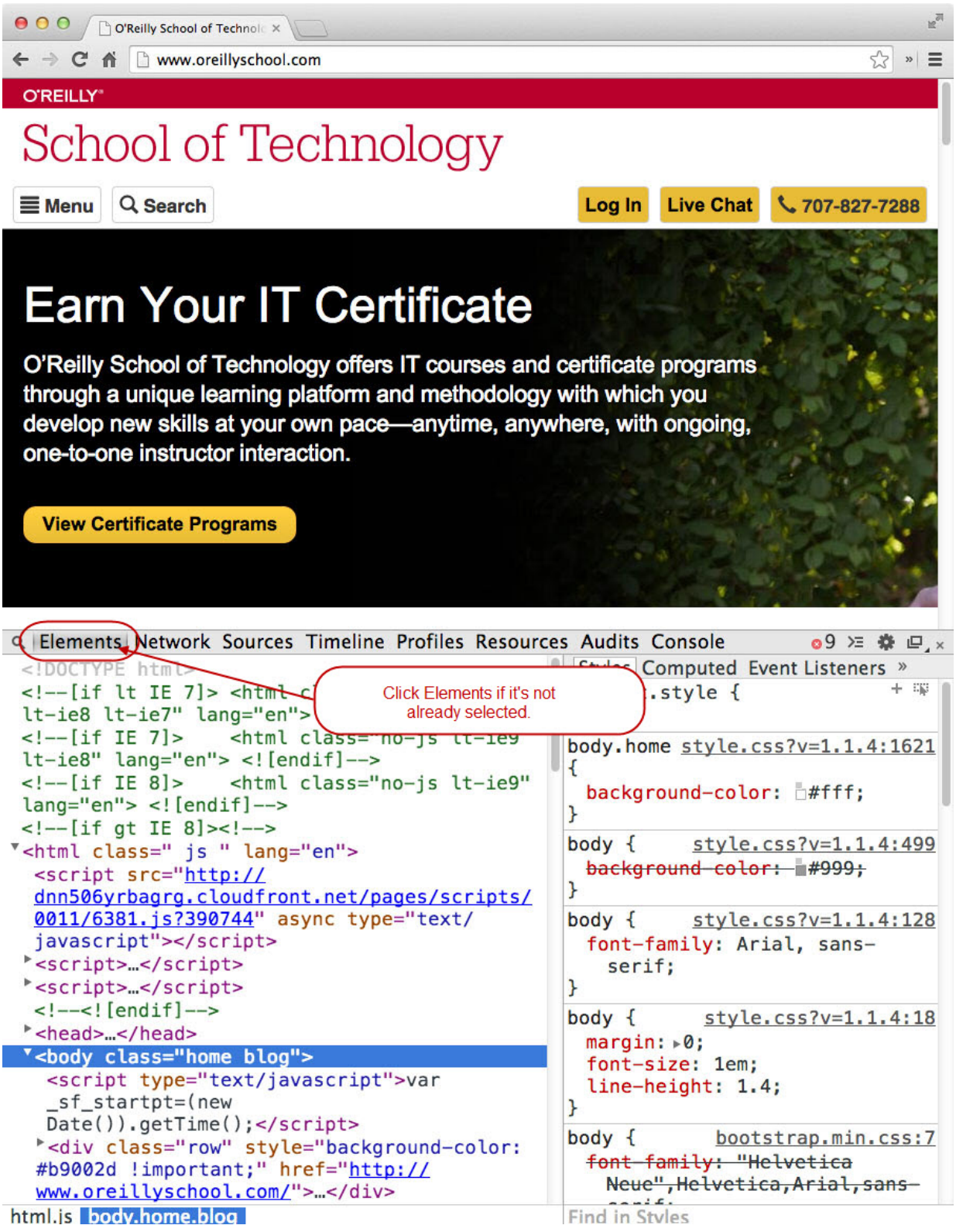
To get instructions to access your console, click on the link to the browser you're using:

- Chrome
- Safari
- Firefox
- Internet Explorer

To access the developer tools in Chrome, use the **View | Developer | Developer Tools** menu:



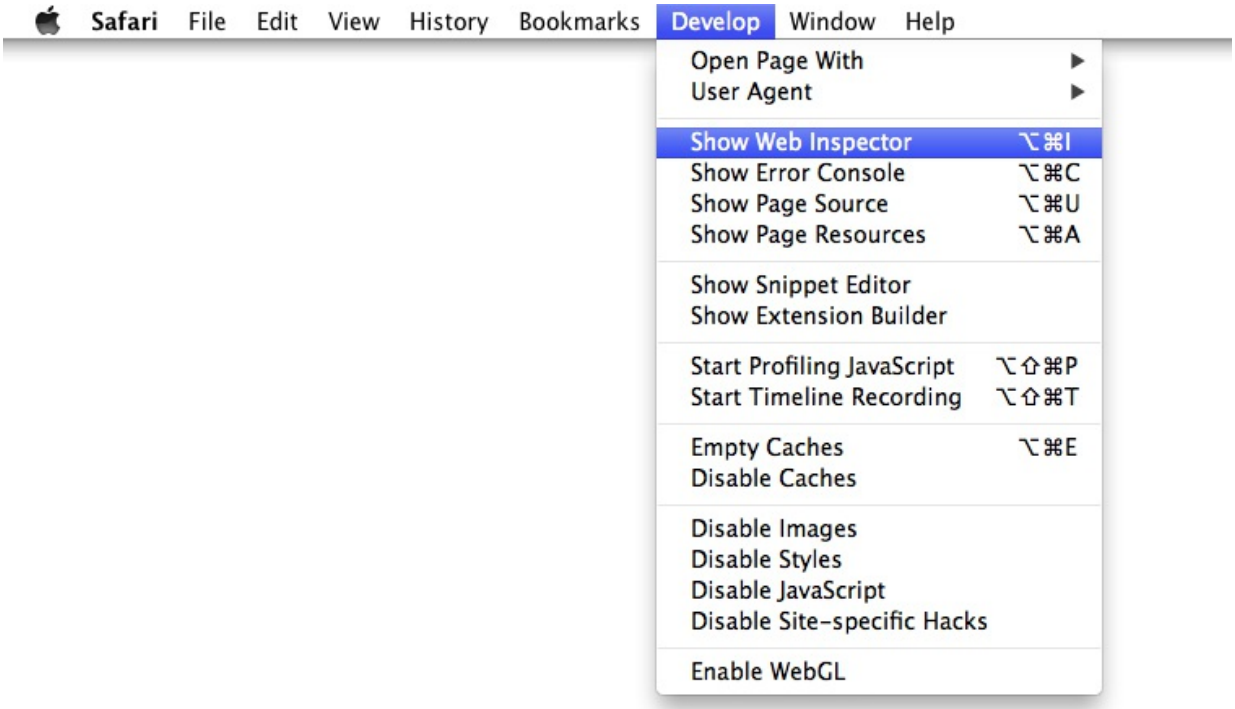
The developer tool will open in the bottom half of your browser window. If the **Elements** tab is not already selected, click it at the top left of the developer portion of the browser window. Once you do that, you'll see:



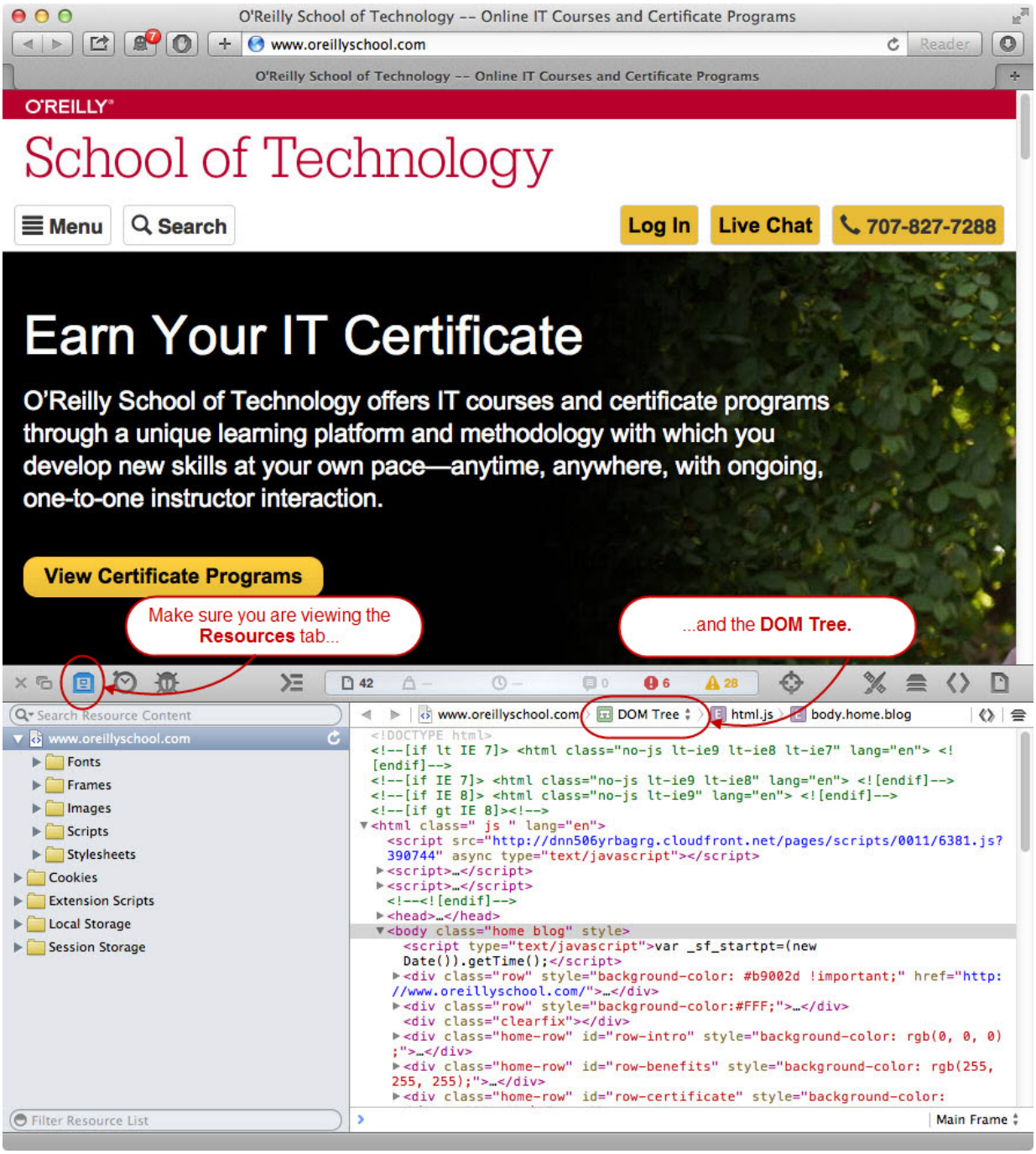
There are many parts to the Elements tool. You'll become more familiar with some of them as we work through the course.

Continue to the next step.

To access the developer tools in Safari, use the **Develop | Show Web Inspector** menu (if you don't have **Develop** enabled, you can enable it with **Safari | Preferences | Advanced | Show Develop menu in menu bar**):



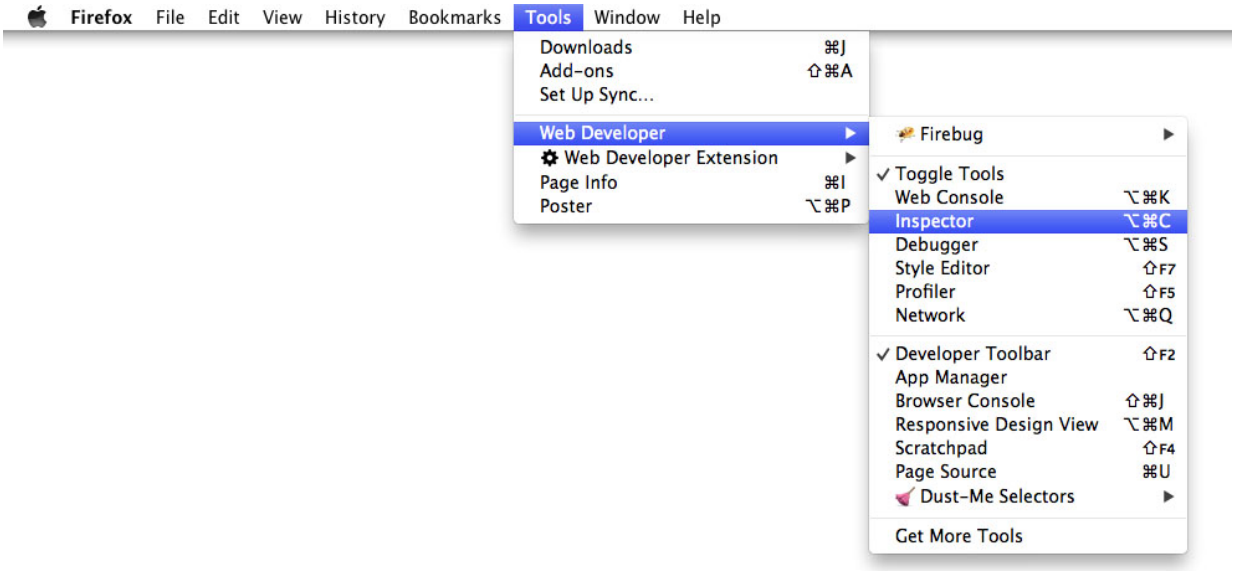
Make sure you've selected the Resources tab, and that you're viewing the page as a DOM Tree:



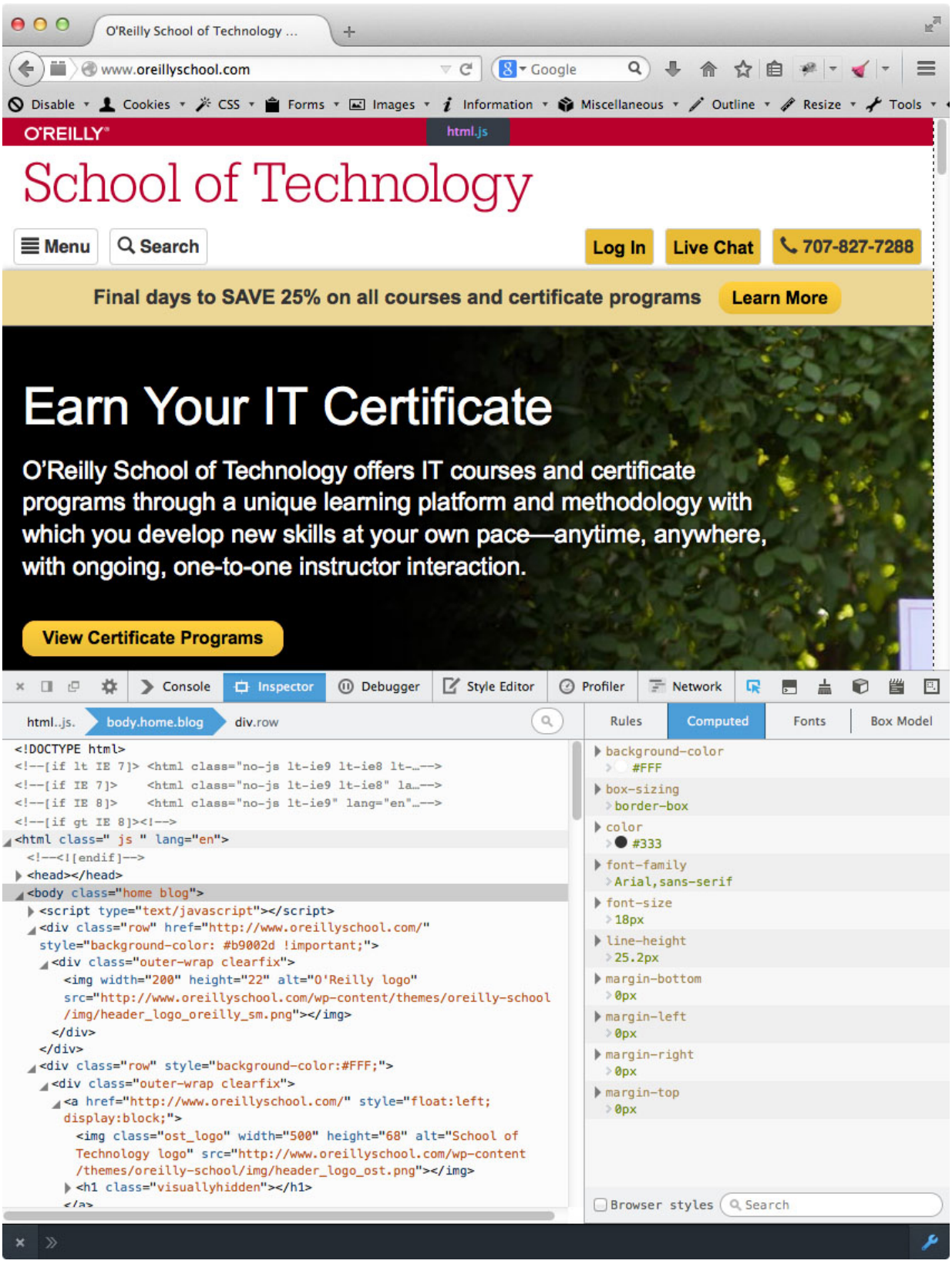
Just like with Chrome, there are many parts to the developer tools in Safari. We'll focus on using Chrome and Firefox in this course, but you should familiarize yourself with the Safari tools if you use that browser for development.

Continue to the next step.

To enable the Firefox developer tools, use the **Tools > Web Developer > Inspector** menu and select the Inspector tab:



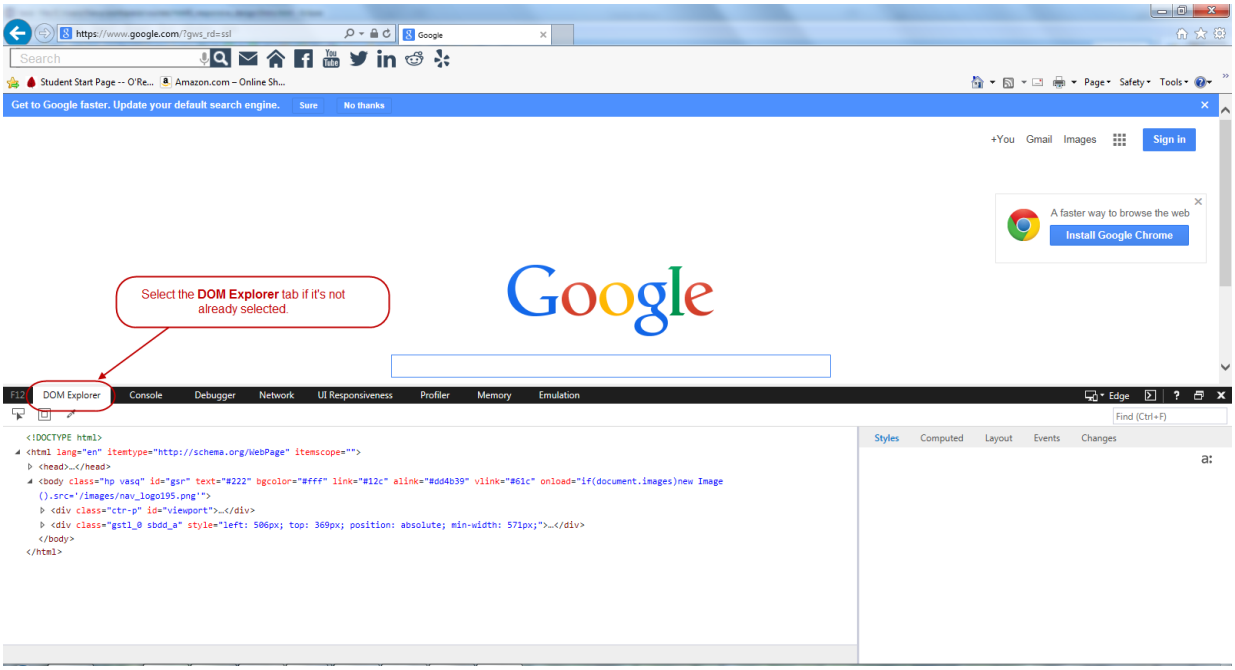
Once you do that, you'll see this:



Again, as you can see, there are many components in the Firefox developer tools, and we'll be taking a closer look at some of these throughout the course.

Continue to the next step.

To enable the Internet Explorer developer tabs, select **Tools | F12 Developer Tools** and then select the **DOM Explorer** tab if necessary.



About the Images Used in This Course

For this course, I've used several public domain images and videos. Some are from Flickr, some from Wikimedia Commons, some from the Internet Archive, and a couple of icon images are freely downloadable from findicons.com.

Most of the images from Flickr are shared under the Creative Commons share-alike with attribute license. Here is the information for those images:

OBSERVE:

```
Photo attributions

Link to license: https://creativecommons.org/licenses/by/2.0/

Changes to photos: cropping
Large size: 960 x 720
Medium size: 600 x 450
Small size: 300 x 225  (saved at JPEG 6)

Tour Guide: https://www.flickr.com/photos/davebloggs007/14096480067
Dandelion: https://www.flickr.com/photos/8047705@N02/5572197407/

Iceland:
Iceland_rainbowVolcano: https://www.flickr.com/photos/vicmontol/541610754/
Iceland_Svartifoss Cascade: https://www.flickr.com/photos/vicmontol/541691325
Iceland_Thingvellir: https://www.flickr.com/photos/pocius/4436760643/
Iceland_GullfossWaterfall ("Golden Falls"): https://www.flickr.com/photos/opalsson/3905929777
Iceland_Road: https://www.flickr.com/photos/ja-pix/13134517544

Brazil:
Brazil_Beach: https://www.flickr.com/photos/over_kind_man/3179806357
Brazil_Hill: https://www.flickr.com/photos/rhysasplundh/5313036883/
Brazil_Butterfly: https://www.flickr.com/photos/dany13/10762441335/

Alaska:
Alaska_Whales: https://www.flickr.com/photos/cmichel67/9313654297
Alaska_View: https://www.flickr.com/photos/sbetts/6366505353
Alaska_Fox: https://www.flickr.com/photos/slobirdr/14140441518
Alaska_Sunset: https://www.flickr.com/photos/noelzialee/327103368
```

We provide a linked list of the images in case you need to use them outside of the OST sandbox. Just right-

click the image name in the list and save it wherever you need it.

Now that you've been introduced to responsive web design, and you've seen a responsive web site, we'll dive right into the nitty gritty of how you plan and create a responsive website. Along the way, we'll explore these key aspects of responsive design:

- Content-oriented design strategies
- Semantic HTML
- CSS techniques for responsive design, including media queries
- Screen densities for different devices, and how to work with them
- Feature detection
- Web fonts
- Responsive images
- Responsive forms
- Loading content dynamically
- Performance issues

For this course, we assume you already know some HTML, CSS, and JavaScript (and jQuery). If you need a refresher on these technologies, talk to us about those courses.

Goals for a Responsive Site

Lesson Objectives

When you complete this lesson, you will be able to:

- describe the website we'll be building in the course.
- choose whether to create a responsive website or separate applications for mobile and desktop.
- organize your content and describe the goals of your potential audience, and your website.
- describe the kinds of tools that are available for creating responsive design applications.

Our Goals

With responsive web design, our goal is to create a website that can dynamically change depending on the screen size of the device being used to view it. It's one solution to the problem of designing *one* website (instead of many) for the many devices—from phones with small screens to large desktop monitors—now available to those using websites.

Throughout this course, we'll work on a website for a fictional travel company, Dandelion Tours. We want users of the site to be able to find information about travel tours, see photographs from previous tours, and contact the tour guides. And we want users to have a positive experience on the site no matter what device they're using to access it, and that's where responsive design comes in.

> **Note** Our Dandelion Tours website won't be completely functional as we're not going to hook it up to a backend server, but you'll get the full experience of building the front end of the site in this course.

Choices

Before we get started with the design using HTML, CSS, and a bit of JavaScript, we should mention that there are other options besides building one website for all devices. We'll quickly run through what some of these options are, and why you might choose one versus another.

Different devices, different sites

You could choose to develop separate websites for mobile and for larger screens. This practice is becoming less popular as support for responsive design has progressed in desktop and mobile browsers. However, as we discussed in the previous lesson, the advantage to using a separate site for mobile is you can create a design from scratch, one specific for smaller screens, without having to worry about making it work for all sizes of screen. And as we mentioned earlier, the big downside of this approach is then having to maintain two separate websites!

Website for desktop, mobile app for mobile devices

Another option would be to develop apps designed for specific devices. For instance, you could have a website for your desktop users, and then direct users to mobile apps if they try to access your website via mobile (prompting them to download an app for a premium experience). Again, the advantage of this approach is that you have a lot more control over the small screen experience. And the big disadvantage is that you're now supporting a website as well as one or more apps for devices. As the number of devices grows, this promises to become more of an issue.

One *responsive* website that works on desktop and mobile

Unless you want your users to have a very specific experience that would be difficult to provide in a website for mobile, I recommend a website that is accessible on all devices—one that is designed to be responsive—provides the best user experience. Many companies choose to create a responsive website, and then provide additional apps for devices that do something more specific to mobile, something that's not provided by the website at all. For instance, think about a company like Starbucks. You can access their website on any device, but if you want to be able to use your phone to pay for a drink you need to download the app.

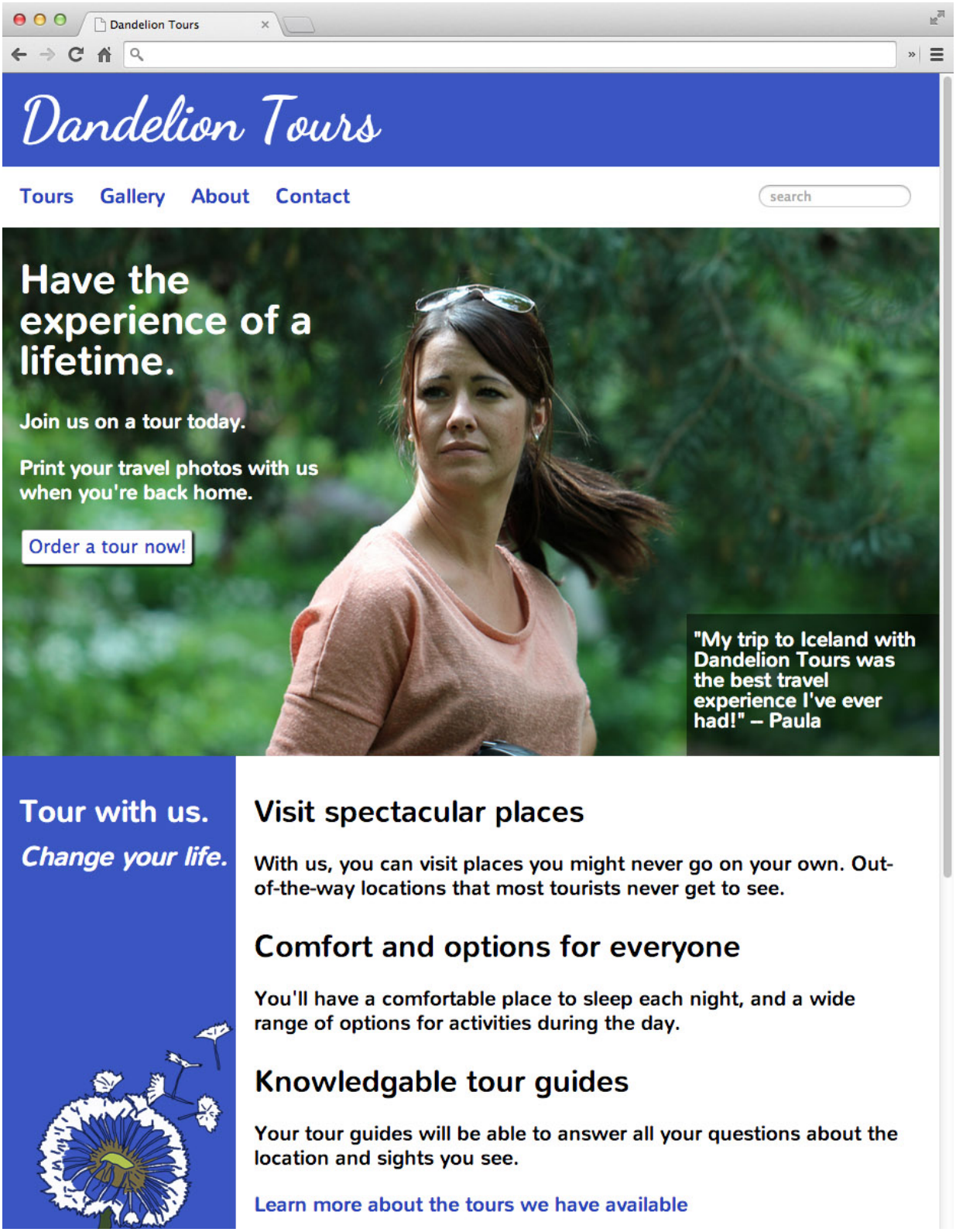
In this course, obviously, our choice is to build a responsive website for Dandelion Tours that will work on, and provide the user a good experience on, all (modern) devices that can access the web.

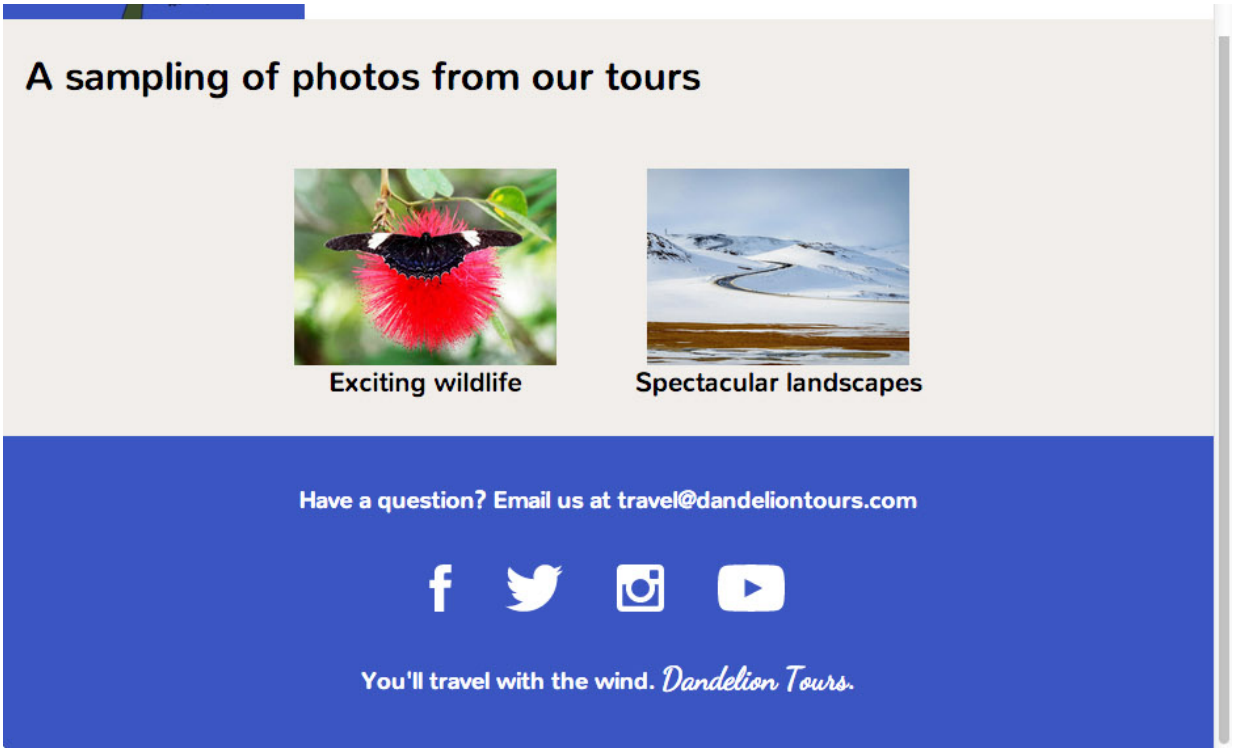
The End Result

The final website will consist of six pages:

- a Home page that provides an overview of what the site is about
- details about the Tours
- a Gallery of images and videos
- information About the company
- a way to Contact the company for more information
- a page for Ordering a tour

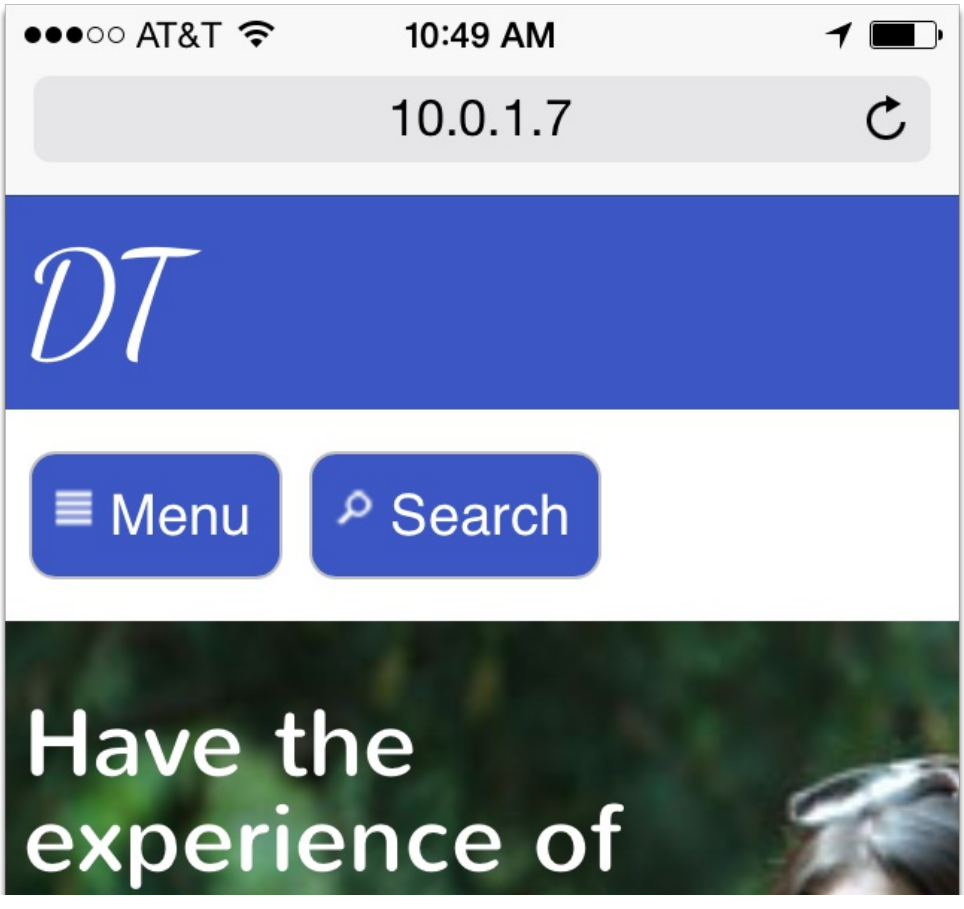
The home page of the finished site will look like this:

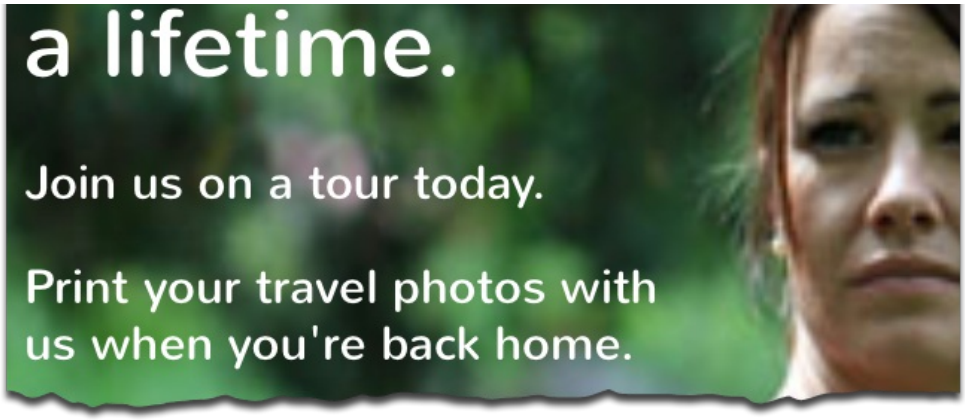




At the top of the page is the header, with the Dandelion Tours logo, the navigation bar, including a search input and links to other pages in the site. Following that is a call to action section for the user to purchase a tour, with a background image (presumably of a satisfied tour guest!) and a quote. Below that, we have more sales talk, including a link to a page that will provide more information about the tours that are available. Below that, we have a sampling of photos from previous tours, and finally, at the bottom, we have the footer of the page, including some links to the fictional tour company's social media sites.

The same page when the browser is narrow (either on the desktop or on a mobile device with low pixel density—which you'll learn about later), looks like this:





Tour with us.

Change your life.

Visit spectacular places

With us, you can visit places you might never go on your own. Out-of-the-way locations that most tourists never get to see.

Comfort and options for everyone

You'll have a comfortable place to sleep each night, and a wide range of options for activities during the day.

Knowledgable tour guides

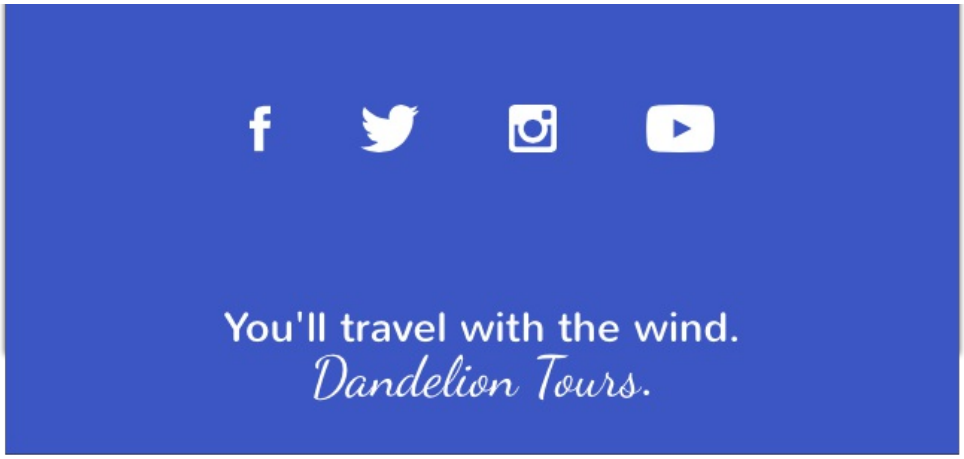
Your tour guides will be able to answer all your questions about the location and sights you see.

Learn more about the tours we have available

A sampling of photos from our tours

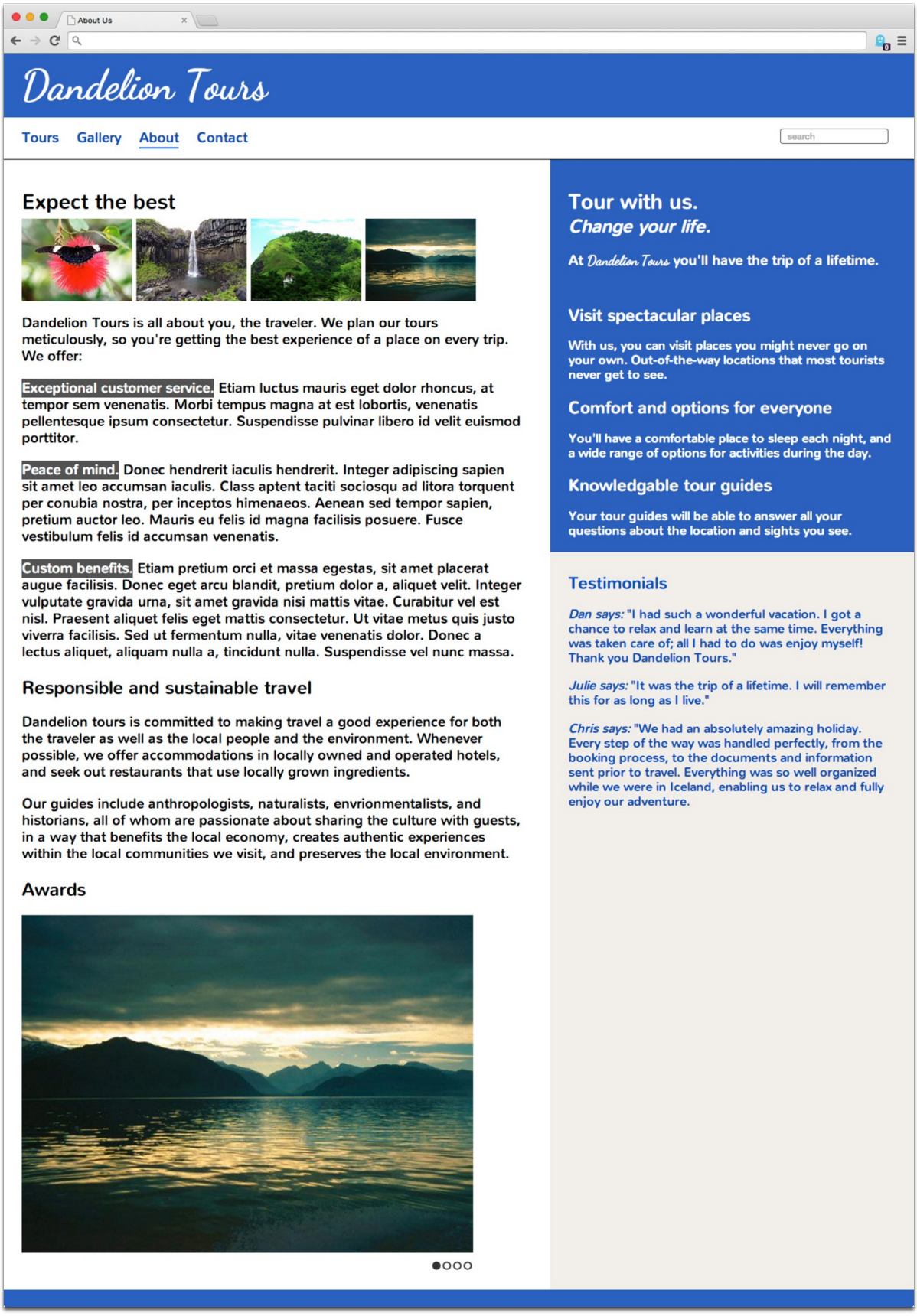
Incredible views

Have a question? Email us at travel@dandeliontours.com



The page layout changes to fit the narrower view. That means the page is a lot longer (you have to scroll more to see all parts of the page), but the content looks better and is easier to read in the narrow view using this layout than the layout we use for the wide screen view. Notice that the page fits the narrower screen perfectly so you don't need to scroll sideways. Changing the layout and style of a web page when the browser width is wide versus narrow is the crux of responsive design, although we'll look at many other related issues too.

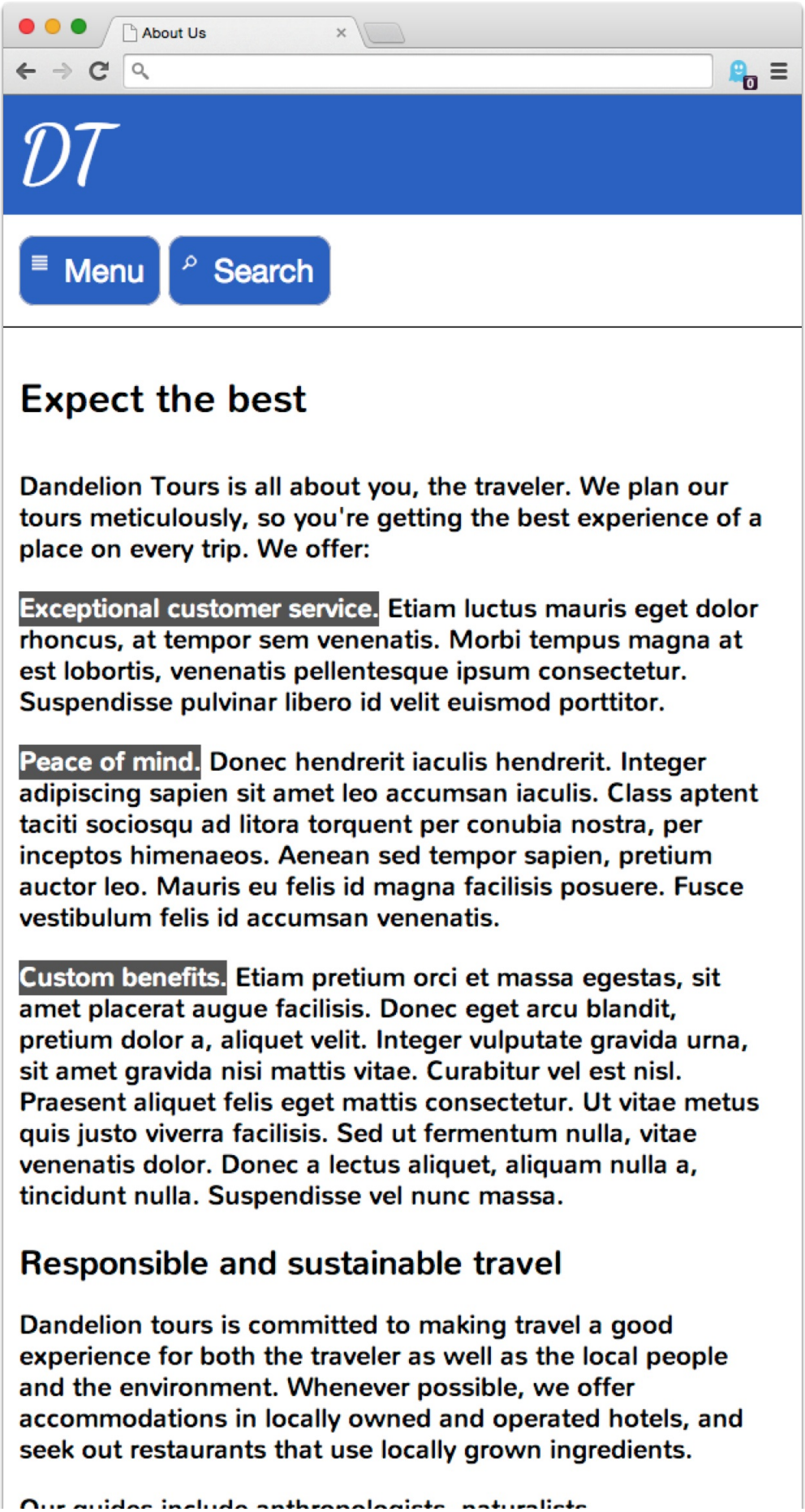
The home page of the site has one structure; the other pages in the site—the About page, the Tours page, the Gallery page, the Order page, and the Contact page—have a different structure:



For all the pages in the site other than the Home page, we've structured the pages so that in the wide view, there is a sidebar on the right side of the page, and the primary content of the page is on the left. In our example, the sidebar content is the same for each page, although it wouldn't have to be. The heading, the navigation and the footer are the same as with the Home page, however. Notice too that the page we're currently on is selected in the navigation links at the top, under the logo.

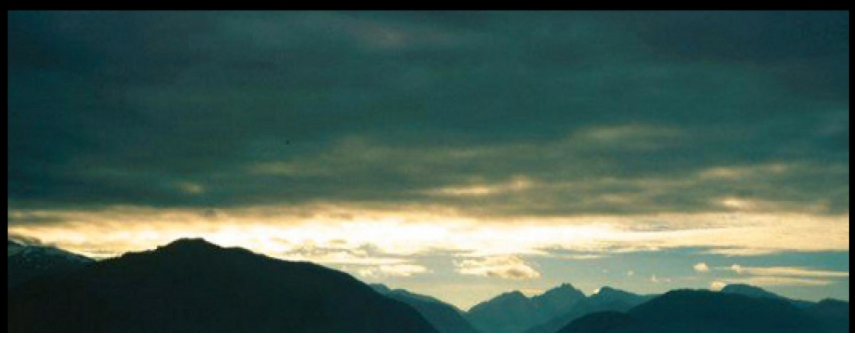
However, having the sidebar on the right side of the page doesn't work well in the narrow view because the

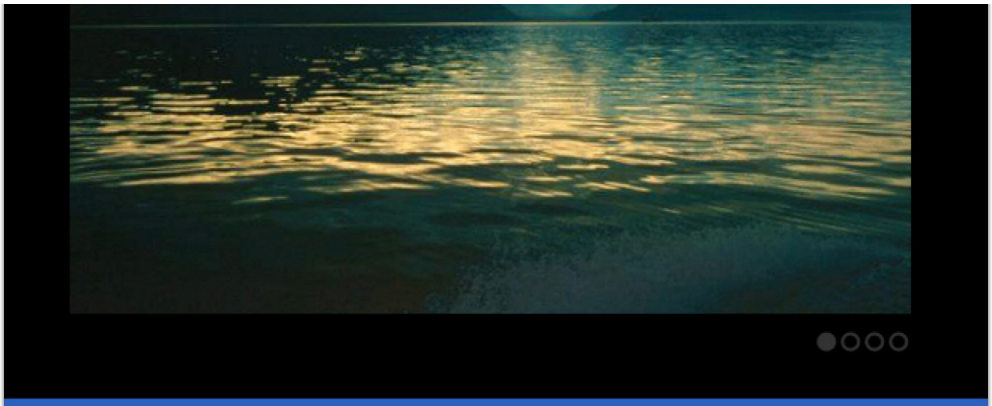
content in both the sidebar and the primary content area gets squeezed too much. So for the narrow view (and on mobile), we'll use a single column layout, moving the sidebar below the primary content:



Our guides include anthropologists, naturalists, envrionmentalists, and historians, all of whom are passionate about sharing the culture with guests, in a way that benefits the local economy, creates authentic experiences within the local communities we visit, and preserves the local environment.

Awards





TOUR WITH US.

Change your life.

At *Dandelion Tours* you'll have the trip of a lifetime.

VISIT SPECTACULAR PLACES

With us, you can visit places you might never go on your own. Out-of-the-way locations that most tourists never get to see.

COMFORT AND OPTIONS FOR EVERYONE

You'll have a comfortable place to sleep each night, and a wide range of options for activities during the day.

KNOWLEDGABLE TOUR GUIDES

Your tour guides will be able to answer all your questions about the location and sights you see.

TESTIMONIALS

Dan says: "I had such a wonderful vacation. I got a chance to relax and learn at the same time. Everything was taken care of; all I had to do was enjoy myself! Thank you Dandelion Tours."

Julie says: "It was the trip of a lifetime. I will remember this for as long as I live."

Chris says: "We had an absolutely amazing holiday. Every step of the way was handled perfectly, from the booking process, to the documents and information sent prior to travel. Everything was so well organized while we were in Iceland, enabling us to relax and fully enjoy our adventure.

We moved the sidebar below the primary content because the primary content is more important, so we want the user to see that first. We also chose to move the photos from the top of the page in the wide view to the bottom of the primary content, again because the photos are less important than the text (in this case—this

won't always be true). On this page, the goal is to share more information about the company, so the photos are secondary to that information. On a page like Gallery, we would want to put photos (or video) right at the top above any text content because that's why the user accessed the page: to see photos and videos from the tours.

You can probably see how it's important to think about the goals of the end user, as well as the goals of the company, when considering how to lay out content in both the wide and narrow views of the website. Considering these goals when creating layout and style is a vitally important part of responsive design.

Think 'Content First'

Users visit a website for content. In the case of Dandelion Tours, they are visiting the site to get information about the tours, the company, prices, and so on. So, while you might be tempted to start thinking about building a website in terms of the layout, the thing that should be primary in your mind when considering your website is not the HTML, not the CSS, not the JavaScript, not even the layout... but the *content*. For most of the pages on this site (with one exception, the About page), we've created realistic content for the site. Why? Because we need to think carefully about how to organize the content on the site, which content is most important so we can make sure it's easily accessible to the user in a variety of different layouts, what goals the user has when accessing the site and how the content fits into that, and so on.

Think about a site that's completely different from a Travel company site. A shopping site, or a site with video learning content, or a site for a small town City Hall... the content on these sites will be completely different from Dandelion Tours, and the goals of the user are also completely different.

Your final project is going to be implementing a responsive website for a site that's completely different from Dandelion Tours, so begin thinking *now* about the content you might want to use for the site, and keep that in mind as you go through the rest of the course, thinking about the goals of the user for your site, how you might prioritize the content differently, and (once you have that down), how you might lay it out (and handle wide and narrow views!) and style it differently from what we dohere in order to meet your own goals, as well as those of your potential audience of users. You'll probably find you'll want to make different design choices from what we do here for your web site. My goal for this course is that you learn responsive design techniques that you can apply to any kind of website.

When you're planning your responsive website, here are some good questions to keep in mind:

- What is the content's purpose?
- How does content relate?
- Which content is most important?
- Is content repeating across the site or specific to a page?
- What are the categories of content (and how many)?

We've talked about the purpose of our content, but what about how each piece of content for Dandelion Tours relates to the other content in the site, and how the content should be organized? We want the Home page to be about enticing potential customers to either purchase a tour, or explore the site to get more information about the tours. So we've included some "sales" language on that page, along with some photos to get users excited. We've split the rest of the site up into five pages, each corresponding to a goal of the user: one, get more detailed information about each tour (**travel.html**, the Tours page); two, get more detailed information about the company so hopefully they trust Dandelion Tours and understand how the company works (**about.html**, the About page); three, see examples of photos and videos from tours so they know what to expect on a tour (**gallery.html**, the Gallery page); a way to contact the company to provide their email and desired tour dates (**contact.html**, the Contact page); and one other page (not shown in the screenshots), an order page that allows the user to purchase a tour (**order.html**, the Order page that you'll implement as a project). Each page corresponds to one specific goal, and all five work together to support the Home page and the overall site, and goals of the company.

Which content is most important? Well, it depends on the page. On the home page, the content that's most important is right at the top under the company logo and navigation: an order button (in case the user is ready to purchase) and a strong message selling the tours that the company offers.

Notice that we put the navigation right at the top under the logo to make it easy for the user to find. We keep the navigation simple and clear so the user has no trouble finding their way around the site. Navigation shouldn't detract from the core message of the site, but it should be easy to find and easy to use.

On the other pages, we've put the primary content on the left, and the sidebar content on the right, which prioritizes the primary content slightly because most users tend to look at a page from left to right (this might be different in some other countries, so it's an important consideration when creating internationalized versions of web pages, a topic we don't cover in this course). On some pages, we consider photos equal to

or perhaps more important than text (for example, the Gallery page, and possibly the Tours page—another page you'll create), and on other pages, the text is more important (for example, the About page). Which content is most important on a page also determines how the layout changes when in the narrow view versus the wide view, as you saw above.

On the Dandelion Tours page, we have several repeating sections throughout the site: on every page, we use the same header (with small modifications to the navigation to keep the user oriented within the site), and the same footer. And on pages with the sidebar, we're repeating the same sidebar content. However, the primary content on each page is different.

As we discussed above, we've split up the pages on the site according to the goals of the user, which corresponds well to the categories of content we have: informational (for example, the About page), sales (the Home and Tours pages), inspirational (the Gallery page), and action pages (the Contact and Order pages).

Thinking about these questions as you plan your site is a great way to help you organize and prioritize the content you want to put on your site. And thinking about the content in terms of goals helps you to focus on making sure the site meets the goals of the end user: after all, that's really what a website is all about! And making a website responsive just makes the user experience for users better, because you know they are getting the same great content optimized for the view they are using when accessing your site.

File Organization, and Classes and IDs

In this course, we'll create four HTML files corresponding to pages in the site: **index.html** (the Home page), **about.html** (the About page), **contact.html** (the Contact page), and **gallery.html** (the Gallery page). These are the pages we create in the lessons. You will create an additional two pages: **travel.html** for the Travel page, and **order.html** for the Order page. We'll also create several CSS and JavaScript files, which we'll describe as we go. The way we'll organize these is to split them up into files that are shared across all the pages in the site, and files that contain CSS and JavaScript that are specific to individual pages.

Organizing files and organizing content are two important things to think about as you plan your website. Another thing to think about, especially as you start planning your HTML structure, and think about prioritizing content for wide views, narrow views, and mobile views, is the ids and classes you use on your HTML elements.

Think of ids and classes as performing two important functions in your code: first, ids and classes add semantic information for you, the developer (and anyone else who might be looking at or working on your code in the future). Using good names for ids and classes helps anyone to understand your intentions for that structure and content.

Second, ids and classes allow you to select elements in your CSS and your JavaScript code to style those elements or add behavior for those elements. Ids allow you to select one single element in your page easily, because, remember: ids must be unique within an HTML page. You can use the same id in two different pages, and in fact, we'll do this often; for instance, the sidebar section has the id "sidebar" in every page, but it is unique *within* each HTML page. Sometimes, however, you don't need to select an element uniquely, and in fact you may want to select multiple elements, and in that case, using a class is a better option. For instance, we want to style the Dandelion Tours text as the company logo when we use it both in the header and the footer. To use the same font family for the logo no matter where it appears in the page, we use the class "name" on every element that contains this logo. We can then use this class to style the logo using the same font family with one rule in the CSS. We could also, if we wanted to, select all elements containing the logo in our code using this same class name.

And, of course, if you need to, you can use both an id *and* a class on an element. We'll do that less often, but occasionally we'll need to.

So, it's a great idea to be thinking about classes and ids for your content structure as you're thinking about the content organization and the file organization. You don't need to make any final decisions at this point, and you may need to make changes to your plan as you actually build the HTML structure and begin adding CSS, but it's a good idea to have a plan in mind before you start building the pages. Jot these ideas down as you're planning out your content.

Responsive Design Tools

You may have heard of tools that do things like help you organize your CSS (for example, SASS) compress your Javascript (JSMin, Packer), and organize all your HTML and CSS code (Bootstrap). We won't cover these tools in this course, but they are great tools to look at once you've completed the course.

There are also tools that can help you create and work with responsive content, such as ResponseJS and Foundation, which are general-purpose JavaScript libraries for responsive design, and libraries like Galleria

(for responsive images), SocialCount (for responsive social media widgets), and FitVids (for responsive video). Another great resource with a ton of tools, from libraries for creating responsive web applications to widgets and much more, is the Filament Group of tools. We won't use any of the tools just mentioned; however, we will experiment with a tool called SlidesJS to implement a responsive slide show widget, so you'll get a sense of how to work with one of these tools in your responsive website.

We'll mention a tool called "Fluid Grids" later in the course; this is a tool quite a few people use to create responsive web pages. Again, we won't use any grid tools in this course, but you can explore this topic on your own once you've completed the course.

We will not do device detection in this course: our goal is to make pages that look good on every screen width, whether on desktop or mobile. However, this is getting more difficult because some mobile devices (like the most recent set of mobile phones, like iPhone 6 and 6+, and the Galaxy S5 LTE-A) have an incredible resolution (number of pixels), so using width as measured in pixels is no longer a reliable way to determine screen size. We do not recommend detecting devices (we prefer to detect features and screen size and make decisions about how to present content based on that information), but sometimes you might find it's just absolutely necessary to look for certain devices. In that case, use a library—either a server-side library (if you're serving content from back-end code), or a front-end library. It's not a good idea to write this code yourself, as devices are constantly being upgraded and new devices added. Check out the Mozilla Wiki for a list of device detection libraries (known as User Agent detection libraries).

Finally, there are many responsive templates you can choose from if you want to start with pre-built structure and CSS, along with some JavaScript, to jumpstart your designs. Some examples are HTML5 Up, Twenty, Tesselatte, and so many more! (Just Google responsive design templates and you'll find many options). We won't be using any templates in this course, but it's fun to look at these to get ideas for sample layouts. These are a lot easier to use and understand once you have a solid understanding of and experience with creating your own responsive designs from scratch, so while we encourage you to take a look at templates for ideas, don't try to use any of them until you've completed the course.

As you can see, there is no shortage of tools, libraries, and templates to help you create responsive designs. Once you've learned the fundamentals of responsive design, you'll be in a great position to evaluate all the various options you have available and choose the right ones to help augment your designs and make creating responsive designs more efficient.

Now that you understand where we're heading in this course, and you've had a chance to think about the goals, content, and organization of Dandelion Tours, we'll dive right into building the site in the next lesson.

Structuring Content for a Responsive Site

Lesson Objectives

When you complete this lesson, you will be able to:

- use HTML to structure content for web pages.
- properly set the size of the viewport for your page.

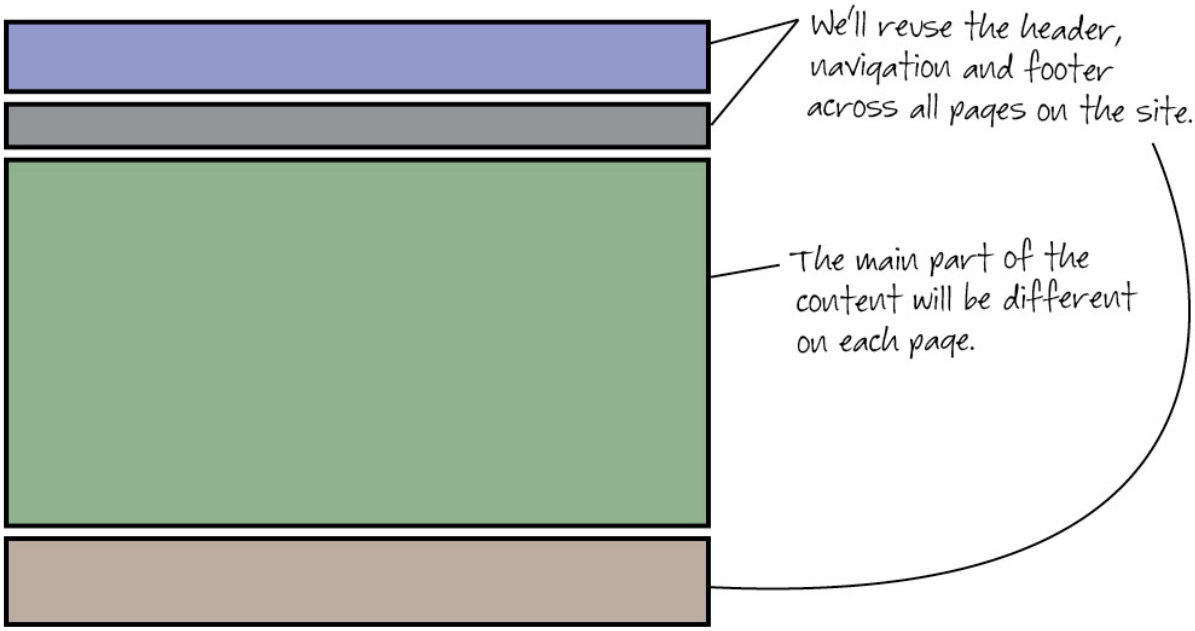
In this lesson we begin building Dandelion Tours, the website we'll be building throughout the course. Here, we focus on the content and structure of the web page and discuss how good structure is an important base for creating responsive web pages. We also look at a simple way to make your pages work better on mobile: the viewport <meta> element.

Creating Semantic Structure and Adding Content

In the previous lesson, we thought about the goals of a user who might access the Dandelion Tours site, and the content of the site that we need to provide to meet those goals. In this lesson, we'll start marking up that content with the HTML to create structure for the home page.

Planning the Structure of the Site

We've split the home page of Dandelion Tours into four main sections: the **header**, with the company name/logo at the top; the **navigation** that allows users to get to other pages on the site; the **main content** for the home page, which includes a testimonial, a little bit about the experience of traveling with Dandelion Tours, and some sample photos to get users excited about the tours; finally, we add a **footer** with an email address, social media icons/links, and the company tag line. To give our travel site a consistent look and feel, we'll reuse the header, the navigation, and the footer on all the pages. The main part of the content will be different on each page:



We are used to seeing branding and navigation at the top of the page, so we're going to continue to keep that information at the top. We typically read a page from the top down, looking for important information, so after the header and navigation, the next most important thing we'd like users to see on the home page is content to get them excited, a button to take them directly to the page where they can sign up for a tour (the same page as we'll link to from the "Tours" link in the navigation), and a testimonial from a satisfied customer.

Look again at the complete Dandelion Tours Home page:

Dandelion Tours

Tours Gallery About Contact

search

Tour with us.

Change your life.

Visit spectacular places

With us, you can visit places you might never go on your own. Out-of-the-way locations that most tourists never get to see.

Comfort and options for everyone

You'll have a comfortable place to sleep each night, and a wide range of options for activities during the day.

Knowledgable tour guides

Your tour guides will be able to answer all your questions about the location and sights you see.

Learn more about the tours we have available

A sampling of photos from our tours

Exciting wildlife

Spectacular landscapes

Have a question? Email us at travel@dandeliontours.com

As you read the page from the top down, notice that we use headings to split up the sections and help the user find the most important points on the page. This also helps them keep track of their place in the page as they look at the content and scroll down and then back up.

We also keep the content simple, concise, and to the point. We don't want to clutter up the page with unnecessary content; our goal is to get the user excited about taking the tour, and ultimately, have them order a tour.

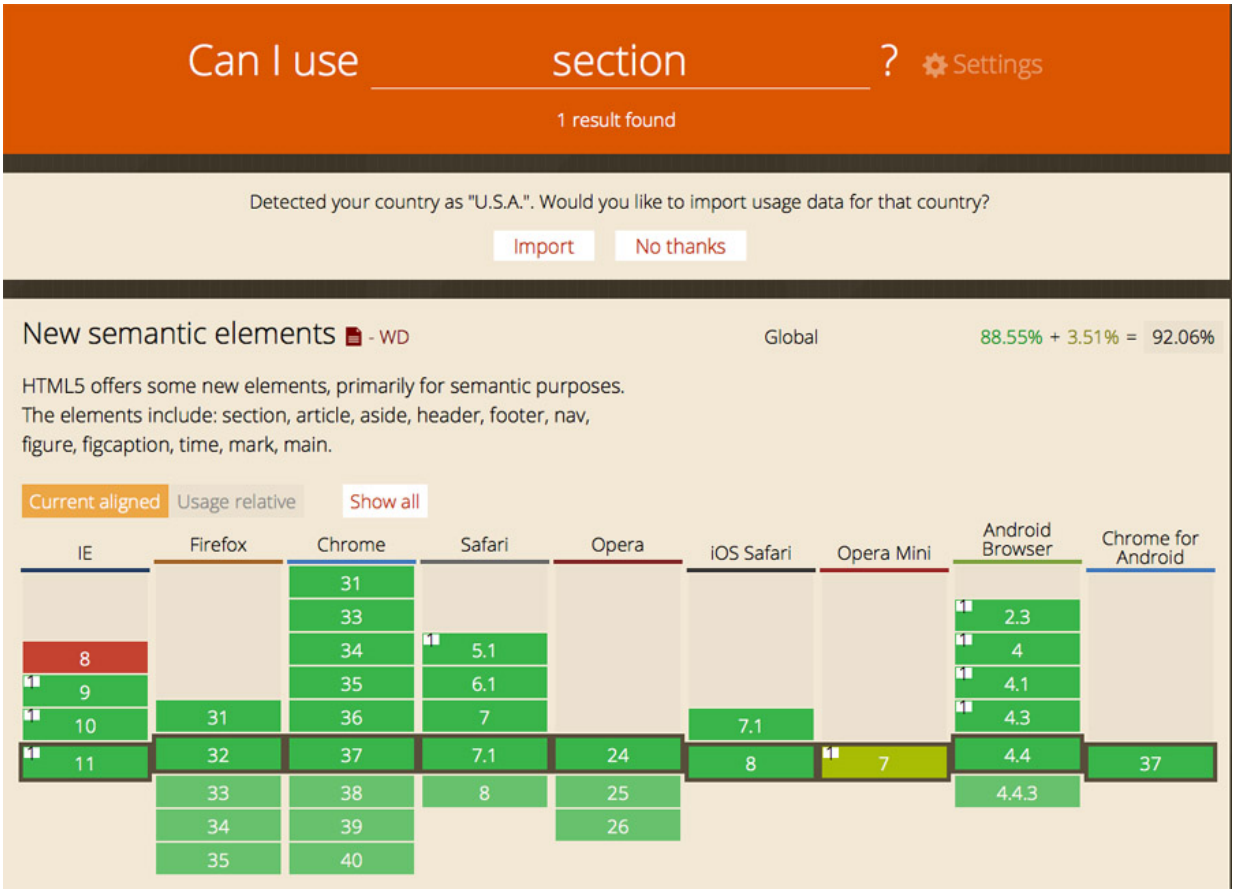
HTML Semantic Markup

As you know, using modern web technologies, we can completely separate content from presentation. We mark up our content using HTML, and we style the presentation of that content with CSS. (and later, we'll add some JavaScript for behavior).

We use HTML tags to structure content. HTML has been around since the mid-90s and while a lot of it has stayed the same, some new elements have been added and others have been removed over the years. Most recently, several *semantic* elements were added to better describe the meaning of content. These elements are part of the HTML5 standard, which is fully supported by all modern browsers, including most mobile browsers (all of those that you'd be concerned about).

HTML will continue to change over time, and new elements will be added to the language. While it's tempting to use these new elements right away, it's important to remember that not all users upgrade their browsers when new browser versions are released supporting new HTML elements (or new CSS and JavaScript versions). A great site to use to check to see whaich versions of browsers support which features is **caniuse.com**.

Let's check to see which browser versions support the HTML5 elements we'd like to use in our responsive page. Access http://caniuse.com, and type in "section" in the search bar at the top. You should see that, as we just said, all modern browsers support the <section> element:



Notice that IE8 and earlier did *not* support the new elements added in HTML5, so if you're concerned about

your users having this browser, you'll need to use a *polyfill*. A polyfill can add capability into some browsers to allow you to use an element (or CSS or JavaScript) that the browser doesn't support out of the box. So, for instance, if you want to make sure a web page that uses the <section> element works on IE8, you can add a polyfill to the page that will check to see if the browser supports an element, and if the browser doesn't, the polyfill will add support for that element (using JavaScript and some HTML tricks). A good polyfill to check out for adding HTML5 element support into older browsers is HTML5Shiv. There are many other polyfills available for all kinds of different web features; check out the Modernizr Github project for a great list of polyfills that are available. For this course, we're going to assume users have a modern browser, so we won't worry about backward compatibility for elements.

Let's go ahead and get the HTML for the Dandelion Tours home page added to a new HTML file. Once you've got that done, we'll come back and talk more about some of the elements and about the structure we chose for the content of the page. Create a new file and add the HTML:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Dandelion Tours</title>
</head>

<body>

<header>
    <div class="full">
        <a href="index.html" class="name">Dandelion Tours</a>
    </div>
    <div class="mobile">
        <a href="index.html" class="name">DT</a>
    </div>
</header>
<nav>
    <div class="menu mobile">
        <button id="menuButton">Menu</button>
    </div>
    <ul class="menu full">
        <li><a href="travel.html">Tours</a></li>
        <li><a href="gallery.html">Gallery</a></li>
        <li><a href="about.html">About</a></li>
        <li><a href="contact.html">Contact</a></li>
    </ul>
    <div class="search mobile">
        <button id="searchButton">Search</button>
    </div>
    <div class="search full">
        <form><input type="search" id="searchInput" placeholder="search"></form>
    </div>
</nav>
<section id="experience">
    <div>
        <h1> Have the experience of a lifetime. </h1>
        <p> Join us on a tour today. </p>
        <p> Print your travel photos with us when you're back home. </p>
        <p> <button class="order">Order a tour now!</button> </p>
    </div>
    <aside>
        "My trip to Iceland with Dandelion Tours was the best travel
        experience I've ever had!" -- Paula
    </aside>
</section>
<section id="details">
    <header>
        <h1>Tour with us.</h1>
        <h2>Change your life.</h2>
    </header>
    <div>
        <ul>
            <li>
                <h1>Visit spectacular places</h1>
                <p>
                    With us, you can visit places you might never go on your own
.
                    Out-of-the-way locations that most tourists never get to see
.
                </p>
            </li>
            <li>
                <h1>Comfort and options for everyone</h1>
                <p>
                    You'll have a comfortable place to sleep each night, and
```

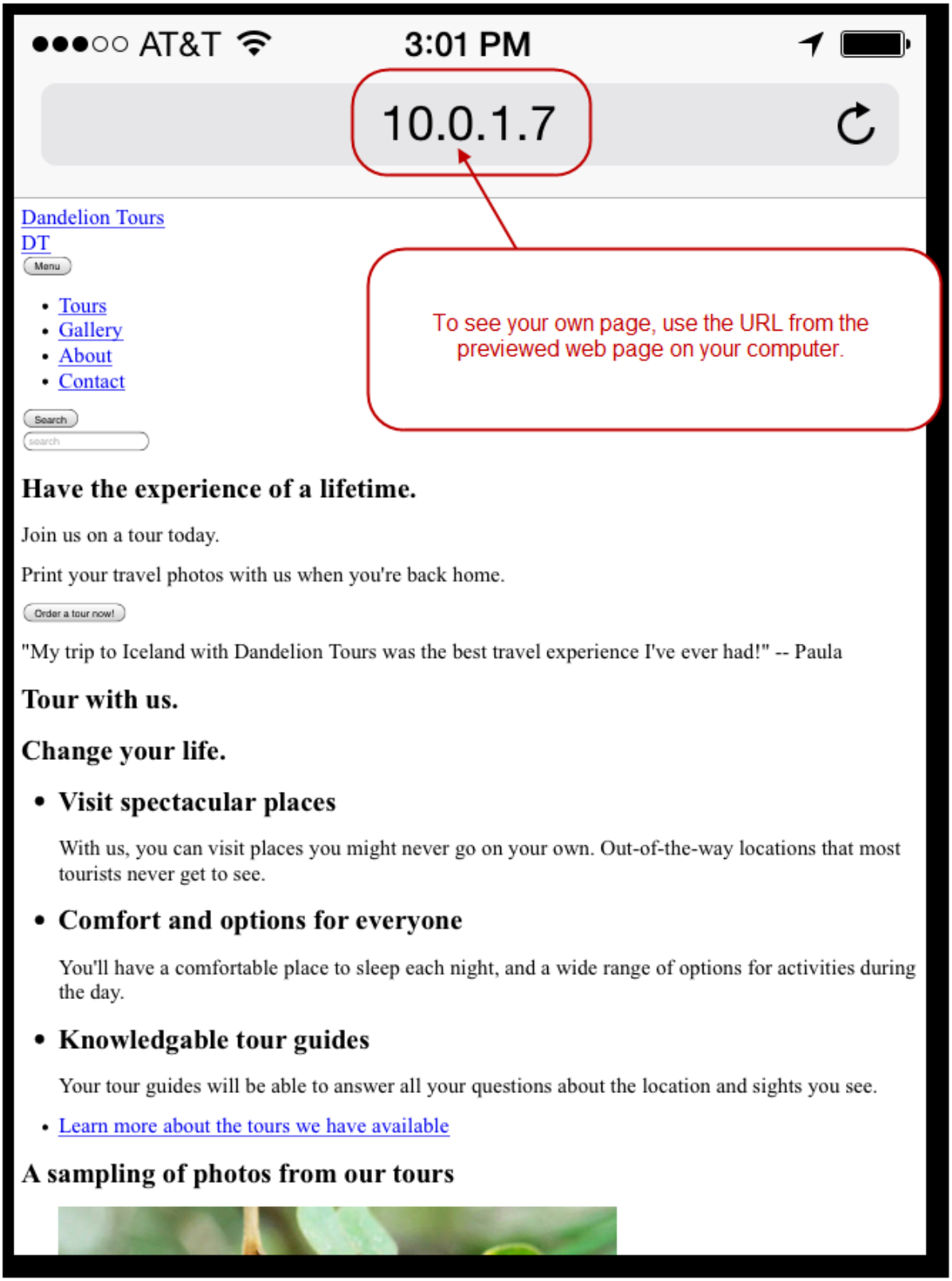
```
                    a wide range of options for activities during the day.
                </p>
            </li>
            <li>
                <h1>Knowledgable tour guides</h1>
                <p>
                    Your tour guides will be able to answer all your questions
                    about the location and sights you see.
                </p>
            </li>
            <li>
                <p>
                    <a href="travel.html">Learn more about the tours we have ava
ilable</a>
                </p>
            </li>
        </ul>
    </div>
</section>
<section id="photos">
    <header>
        <h1>A sampling of photos from our tours</h1>
    </header>
    <div class="photo-container">
        <div class="photo-with-caption">
            <figure>
                <img src="https://courses.oreillyschool.com/html5_responsive_des
ign/images/Brazil_Butterfly_m.jpg" alt="photo of mountains and rainbow">
                <figcaption>Exciting wildlife</figcaption>
            </figure>
        </div>
        <div class="photo-with-caption">
            <figure>
                <img src="https://courses.oreillyschool.com/html5_responsive_des
ign/images/Iceland_Road_m.jpg" alt="photo of road in snow">
                <figcaption>Spectacular landscapes</figcaption>
            </figure>
        </div>
        <div class="photo-with-caption">
            <figure>
                <img src="https://courses.oreillyschool.com/html5_responsive_des
ign/images/Alaska_View_m.jpg" alt="photo of rainbow and waterfall">
                <figcaption>Incredible views</figcaption>
            </figure>
        </div>
    </div>
</section>

<footer>
    <div>
        Have a question? Email us at
        <a href="mailto:example@example.com">travel@dandeliontours.com</a>
    </div>
    <div>
        <a class="social" href="#">Facebook</a>
        <a class="social" href="#">Twitter</a>
        <a class="social" href="#">Instagram</a>
        <a class="social" href="#">Youtube</a>
    </div>
    <div>
        You'll travel with the wind. <span class="name">Dandelion Tours</span>.
    </div>
</footer>

</body>
</html>
```

Save this in your **/ResponsiveDesign** folder as **index.html**. Now preview your **index.html** file. You see

a page that contains all of your content. Of course, it's styled with just the default style the browser uses for various elements we used, like the larger font size, bold for headings, and such.

Try resizing the browser window. Notice that by default most of the content on the page is responsive in your desktop browser, in that it adjusts to the width of the screen. The images don't scale, however, so as you make the width of the browser smaller than the images, a scrollbar appears along the bottom of the window.

If you have a mobile device, try loading the page into your mobile browser. You'll notice that the size of the web page is scaled to fit your device, so everything looks really tiny. This isn't so great, but it's the default way that web pages are displayed on mobile devices, as we saw earlier when we looked at O'Reilly's home page.



A Closer Look at the HTML

Let's now go through the HTML in more detail. We begin the page with the doctype:

OBSERVE:

```
<!doctype html>
```

This tells the browser the document type (in our case, HTML) and the version. As of HTML5, no version number is necessary because, going forward from HTML5, HTML will be backward-compatible with older versions (at least, that's the plan). If you were using an older version of HTML, you'd have a more complex doctype, but now that all modern browsers support HTML5, we don't need that.

The rest of the page is inside the <html> element and split into two main sections: the <head> and <body>. In the <head> element, we have the <meta> tag and <title> element. These are required for valid HTML5 documents. We're using the modern version of the <meta> tag, specifying the charset as UTF-8 in the charset attribute. This is typically what you'll use because it supports many languages (even languages with special characters), and because most text editors use this encoding when creating text files.

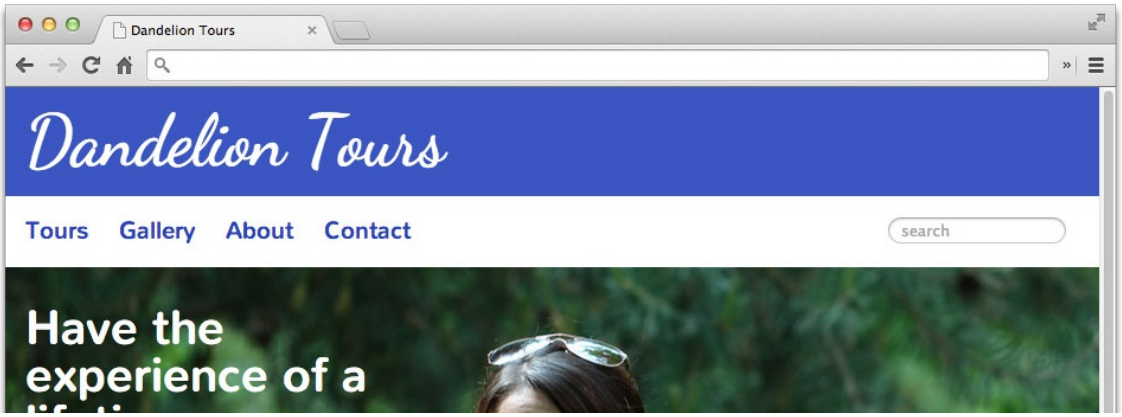
Now take a closer look at the content and elements in the body, which correspond to what you actually see in the browser window. As we sketched out in the wireframe above, we've split the page into sections using elements:

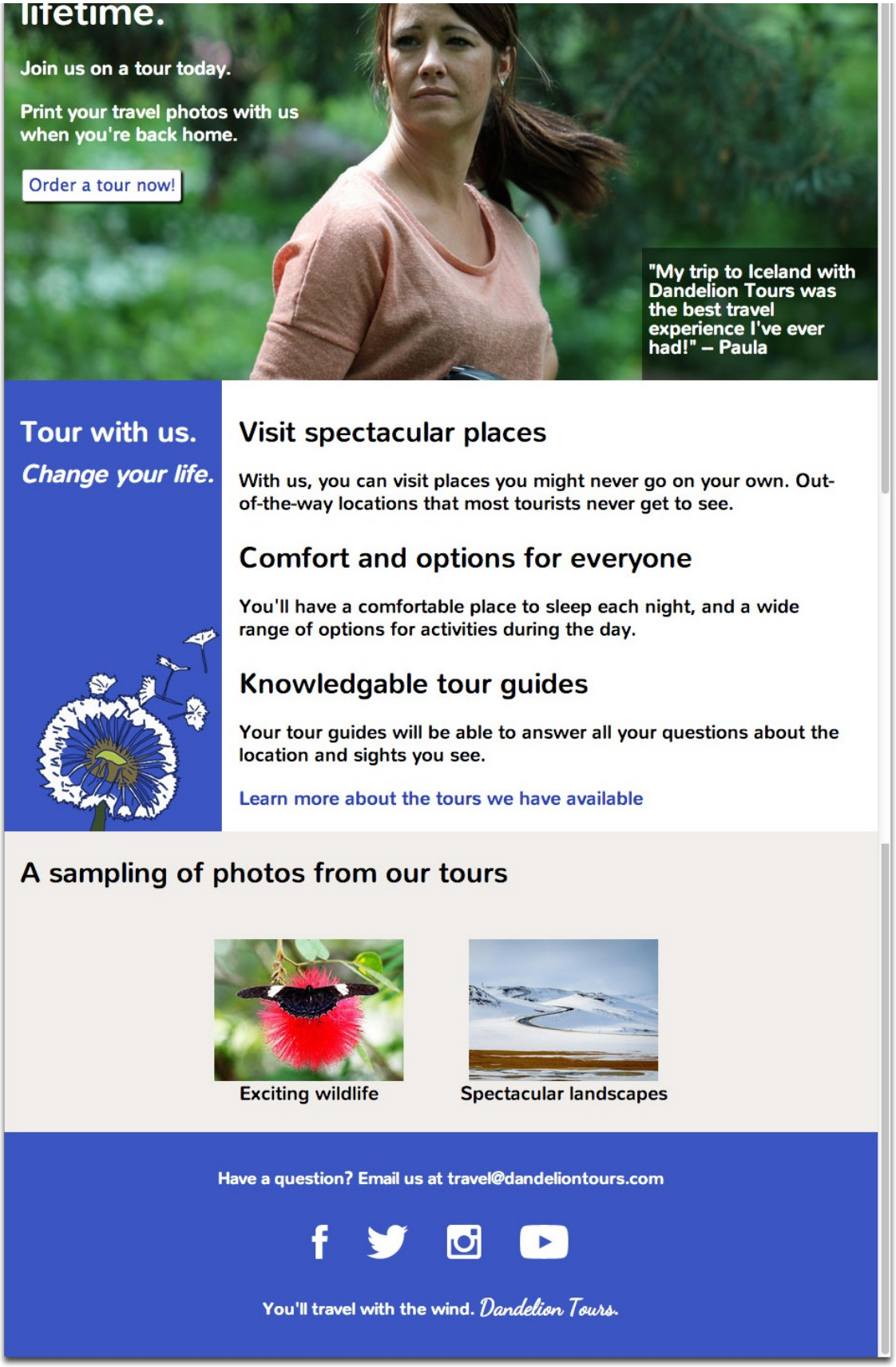
OBSERVE:

```
<header>
   ...
</header>
<nav>
   ...
</nav>
<section id="experience">
   ...
</section>
<section id="details">
   ...
</section>
<section id="photos">
   ...
</section>
<footer>
   ...
</footer>
```

The <header>, <nav>, and <footer> content and structure will be reused across all the pages in the website to give them consistency, but the main sections of content—on this page, we have three—will be different from page to page. Notice that we've given each section a different id so we can target each separately with CSS if necessary.

Recall the finished version of this page:





The three middle sections correspond to the sections of the page: Have the experience of a lifetime; Tour with us; and A sampling of photos from our tours.

The <header>, <nav>, <section>, and <footer> elements make the HTML easier to understand. These elements help us see the big-picture structure of the page when we look at the HTML.

We also use elements in the page (think of as a generic way to divide up the page contents without attaching any particular meaning to them). is helpful for identifying parts of content that will be styled with CSS later, and you'll see that we've already added several classes to many of the elements so we can add style.

We're using some standard HTML elements (<p>, <ul>), plus a few other newer HTML5 elements (<aside>) in the page. The goal is to use the best HTML elements to both structure the content and add meaning, like "this is a paragraph," or "this is a "header." At this point we're not worrying about style, because we can go in and completely change the style of any element using CSS. The point is to use the right element for the content, and worry about style later.

Structuring Content for Multiple Devices

You might have noticed is that in the <header> and <nav> we've got two of everything: two logo links for Dandelion Tours in the **<header>**, and two site menus and two search areas in the **<nav>**:

OBSERVE:

```
<header>
    <div class="full">
        ...
    </div>
    <div class="mobile">
        ...
    </div>
</header>
<nav>
    <div class="menu mobile">
        ...
    </div>
    <ul class="menu full">
        ...
    </ul>
    <div class="search mobile">
        ...
    </div>
    <div class="search full">
        ...
    </div>
</nav>
```

We're using two different classes on each of the pairs of elements: one is "**full**" and one is "**mobile**". Here, we're thinking ahead just a bit. We know we're going to need two different logos, two different menus, and two different search areas depending on whether the browser is wide (corresponding to the "**full**" class) or narrow (corresponding to the "**mobile**" class). If you look at the page as it's currently displayed in the browser, you'll see both the full and the mobile header and nav areas; but don't worry, we'll take care of that once we begin styling the page with CSS, and show just the one that makes sense for the browser width.

Try to keep the duplicated content in your website to a minimum to make maintenenace of the site easier. Although, sometimes you will need duplicate content in situations where you know you can't reuse the same content and structure for the both the wide screen and the narrow screen (mobile) versions of a page.

Setting the Viewport

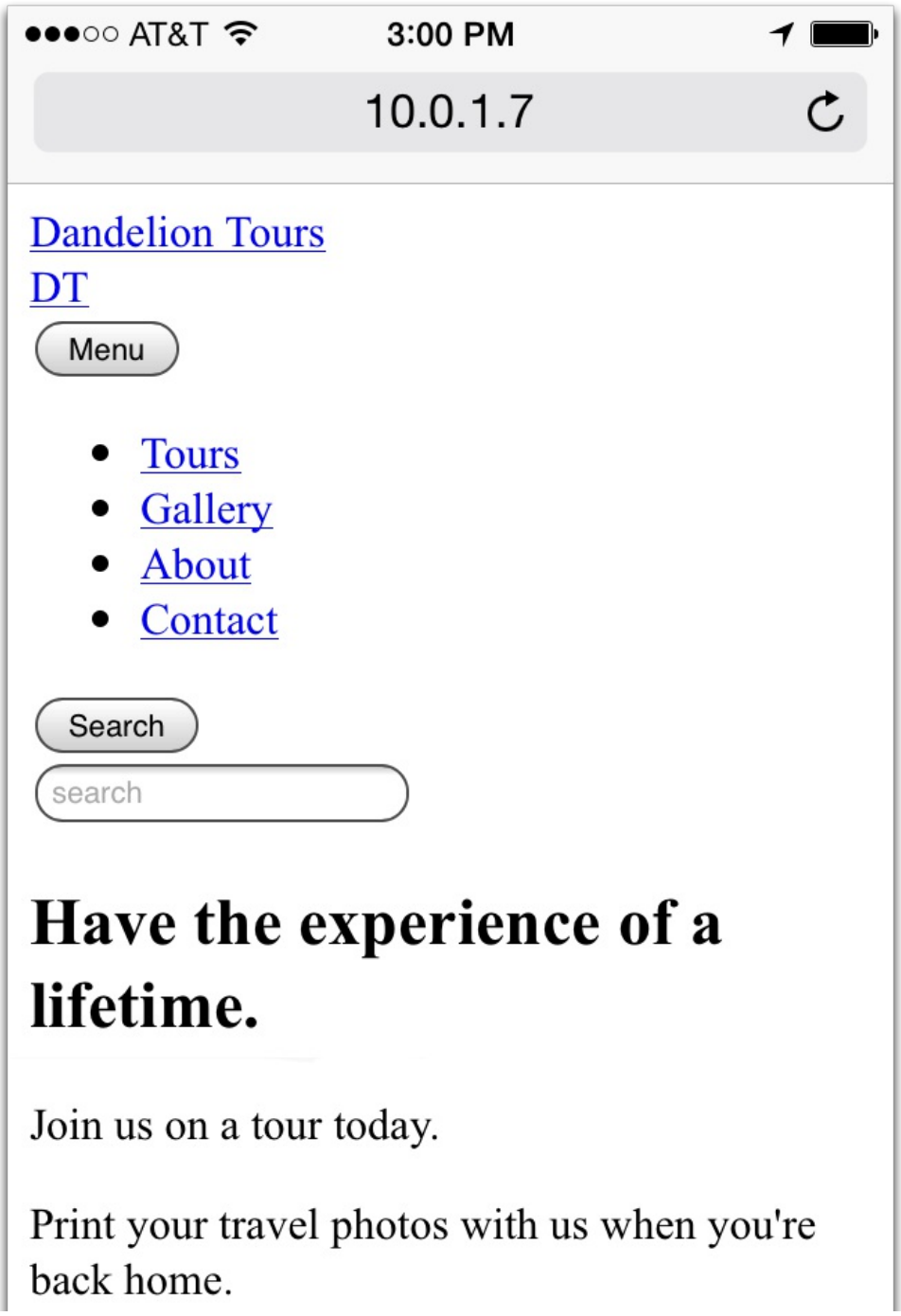
HTML is an important part of creating a responsive website: choosing meaningful elements to structure your content will help you to create a solid base upon which to build the rest of the design. As you've probably guessed, a lot of the work that will make a site look and behave responsively is going to happen in CSS and JavaScript.

Before we jump into CSS for the site though, we need to talk about a key element that makes your site responsive: the viewport <meta> element. Just like the <meta> charset attribute you saw earlier, the viewport is an attribute of the <meta> element, and we set all the various options of the viewport in this one element. Add the HTML below to your file:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Dandelion Tours</title>
</head>

<body>
...
</body>
</html>
```

□ and □. In your desktop browser you will see no change, but if you have a mobile device, you'll see a big change; now, the web page is no longer scaled down to fit the mobile screen. Instead, it looks more like this:



Order a tour now!

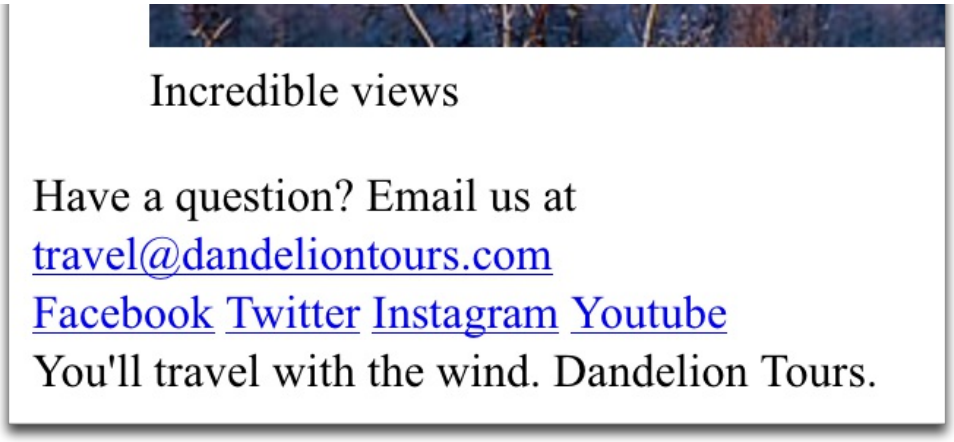
"My trip to Iceland with Dandelion Tours was the best travel experience I've ever had!" -- Paula

Tour with us.

Change your life.

- ## Visit spectacular places

  With us, you can visit places you might never go on your own. Out-of-the-way locations that most tourists never get to



We've included more of the screen so you can see how the content works on the mobile screen for the most part, except the images. They are now much too large, and to see them you have to tap and hold the screen, and then move it around. Not the best user experience for images, but for reading the text and tapping on links, the experience is much improved. Again, don't worry about the images—we'll take care of those later with CSS!

So, what is the viewport?

The viewport is the rectangular area in which your web page is drawn. On the desktop the viewport is resizeable. If you make your window bigger, the viewport gets bigger; if you make your window smaller the viewport gets smaller. If you make your window too small, the web page becomes larger than the viewport, scroll bars appear on the right and bottom, and you have to scroll to see the rest of the page.

On mobile devices like the iPhone, Android phone, iPad, and others, the viewport is the area that determines how your page is laid out, and where text wraps. The viewport on a device can be larger than the display, in which case you'll have content that's "off screen" and you'll have to scroll to see it by tapping, holding and then moving your finger.

On mobile devices like the iPhone 4 and 4s and some Android phones, the viewport is set to a default width: 980 pixels wide. Now this might seem odd since the width of an iPhone 4 is much less than that (an iPhone 4 has a width of 320 pixels). So the iPhone scales the web page so that the viewport is 980 pixels wide within the 320 actual pixels that the iPhone displays. That's why our web page initially looks tiny. If you try scaling it up, by using a gesture, you can get the web page to appear to be a normal size, but because the viewport is 980 pixels wide, you'll have to scroll around in the page to see all the content.

Note Of course with the most recent mobile devices, we have screens with much higher *pixel densities* so the scaling the device does to fit a page onto the screen is different depending on which device you have. We'll talk a lot more about pixel densities later.

With the viewport <meta> element, we can change what that initial viewport width is set to:

OBSERVE:

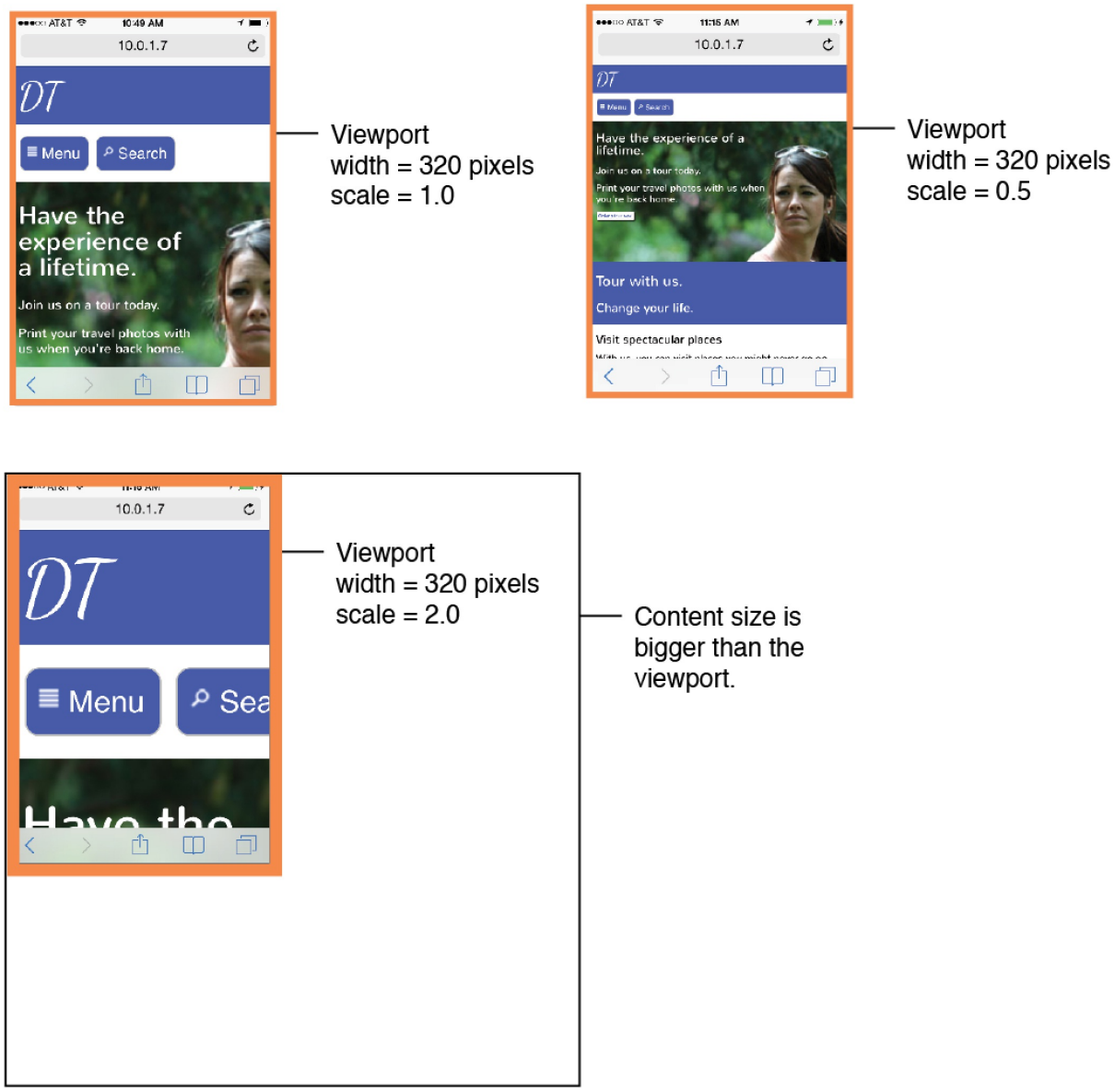
```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Here, we're saying, **set the width of the viewport equal to the width of the device**. So, in the case of my iPhone, the actual width of the device is 320 pixels, so the width of the viewport is set to that. We also want to make sure the web page is not scaled within that viewport width; in other words, we don't want to be "zoomed in" to the page. So, by setting **initial-scale=1**, we make sure that we're not initially zoomed in or out of the page; we see it at exactly the right scale for a 320 pixel width viewport. If we set initial-scale=2 instead, the page would load into the 320-pixel-wide viewport, but then appear zoomed in to twice the normal size. Typically, we want the initial scale to be set to 1, and this works on all mobile devices, regardless of the actual screen size and pixel density.

By setting the viewport to the device width, the browser now flows the content into the page as if the page is 320 pixels wide (instead of 980 pixels wide). So the text wraps nicely when it hits the edge of the viewport, and we see most of the web page looks readable and is sized appropriately. Again, the images aren't

working quite right now, but we'll fix that.

When you've set the viewport to the device width, here's how the scale affects how you see the page:



The viewport is 320 pixels wide (because we've set it to the device width), but the scale affects how much of the page you can see. If you are zoomed in (the scale is set to 2), you only see half the page and you have to pan around using your finger to see the rest of the page. If you are zoomed out, for instance to a scale of 0.5, the page content looks smaller, and reflows to fill the 320-pixel viewport.

Some people also like to set the user-scalable option in the <meta> element to prevent users from being able to scale a page up or down with a pinch/zoom gesture. That way the page stays nicely in its 320-pixel viewport at all times, at the scale you want the user to see the page:

OBSERVE:

```
<meta name="viewport" content="width=device-width, initial-scale=1, user-scalable=no">
```

This also helps to make sure that if the user changes the orientation of the device (for example, on a phone or tablet, from portrait to landscape), the page remains scaled appropriately for the new width.

Even if you don't have a mobile device to test on (which is perfectly fine), you can see from the screenshot how setting the viewport improves the user experience of viewing a website on mobile. Setting up the viewport correctly is an important first step in making your website responsive!

There is a lot more detail you can learn about the viewport on iPhone and Android devices, but what you've learned here will take you far, and is all you need to know for this course.

Now that we've got the content and structure of the Dandelion Tours page set up, and we've set up the viewport so the page will render properly in mobile browsers, we need to begin working on the style of the page. This is where the web page will start looking more like the finished website, and also where you'll learn how style can be used to create responsiveness in some really cool and exciting ways.

Media Queries

Lesson Objectives

When you complete this lesson, you will be able to:

- write media queries to create breakpoints in your page.
- create style for the page that changes at different widths of the browser window.
- use reset.css to create a baseline of CSS that is common across browsers.
- order your CSS file includes correctly.

Now that we have a solid HTML structure in place for our Dandelion Tours home page, it's time to think about CSS. This is where responsive design can really make a page work well on a variety of screen sizes. In this lesson, we'll plan how we're going to organize our CSS and start adding rules so that our pages begin to look more like the final design. You'll learn how to use *media queries*, one of the key features of responsive design, to create different rules for different screen widths.

Styling the Page

Organizing Style Sheets

The Dandelion Tours website has several web pages that are part of the site. Rather than putting all the CSS for the entire site into one big file, we're going to split it up across multiple files. The files we're going to create for the home page of the site are:

- dt.css: the CSS that applies to every page of the site.
- main.css: the CSS for the home (or "main") page of the site.
- reset.css: a file containing reset CSS to set the baseline of style for all elements.

Create empty files for each of those, and then add links to them in the head of your **index.html**:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Dandelion Tours</title>

<link rel="stylesheet" href="reset.css">
<link rel="stylesheet" href="dt.css">
<link rel="stylesheet" href="main.css">

</head>
```

if you reload the file in your browser you won't see any changes because we haven't added anything to the CSS files yet. We'll do that next.

Let's talk about the order of these files first. The **reset.css** file will contain CSS that "resets" the style of elements in the browser to remove all default styles. We do this because each browser adds its own default style to elements; we want to make sure we're starting from a known state, where our elements have no style yet.

Next is **dt.css**. These are the style rules that will be applied to *all* the pages in Dandelion Tours. This sets styles for elements that appear on all the pages (for example, the header and footer), as well as styles for common properties such as font families, font sizes, links, and so on. This builds on the "no style" that we created with **reset.css**, hence it comes next in the order.

Finally, we'll put all the style rules for the **index.html** page in **main.css**. We put this last in the list because we might need to override something in **dt.css**, and also because the styles in this file are the most specific to the page.

Here's a quick example of how this ordering will work:

- In **reset.css**, we'll set the font size of every element to be the same: 100% of the default size of text in the browser (which is typically set to 16 pixels). So, for instance, our headings and our body text will all look the same once we have **reset.css** in place.
- In **dt.css**, we might decide that we want all <h1> headings to have a font size that is 1.5 times the default size of body text in the page (16 pixels times 1.5 = 24 pixels). So, if every page in the site includes **dt.css**, then all <h1> headings will be 24 pixels. This rule builds on the rule in the reset file (which set <h1> headings to 100%), overriding it so our headings are bigger.
- In **main.css**, we might decide that we want a specific heading, say the heading for the "experience" section of the page, to be bigger than 24 pixels. We might want it to be 32 pixels. So, we'll add a rule to set the "experience" <h1> heading to be 2 times the default size of body text in the page. This rule will override the font size for headings in **dt.css**, but only for the heading in the specific section, "experience."

Remember, in CSS, if two rules set different style properties for an element, the rule that comes *last* in the ordering takes precedence. So we put the most specific rules last, as well as rules that might need to override or build on rules that come before.

Reset CSS

Using a reset CSS style sheet is common in web design. You can get a reset CSS style sheet from various sources; we'll use a popular one from Eric Myer, called appropriately enough, Reset CSS. You can learn more about Eric's Reset CSS at his website, http://meyerweb.com/eric/tools/css/reset/.

We're going to use a version of his code from http://www.cssreset.com, where you can download a variety of different reset CSS files. This is a "minified" version of the CSS, which removes the white space and comments from the file so it's fast to download. Remember that any files you link to from your HTML—CSS, JavaScript, etc.—must be downloaded by the browser before it can display your page, so making these files small and efficient can help make your page faster to load for end users.

Go ahead and download the code from http://www.cssreset.com, or copy and paste the code below into your own **reset.css** file:

CODE TO TYPE:

```
/* Eric Meyer's Reset CSS v2.0 - http://cssreset.com */
html,body,div,span,applet,object,iframe,h1,h2,h3,h4,h5,h6,p,blockquote,pre,a,abbr,acronym,address,big,cite,code,del,dfn,em,img,ins,kbd,q,s,samp,small,strike,strong,tt,var,b,u,i,center,dl,dt,dd,ol,ul,li,fieldset,form,label,legend,table,caption,tbody,tfoot,thead,tr,th,td,article,aside,canvas,details,embed,figure,figcaption,footer,header,hgroup,menu,nav,output,ruby,section,summary,time,mark,audio,video{border:0;font-size:100%;font:inherit;vertical-align:baseline;margin:0;padding:0}article,aside,details,figcaption,figure,footer,header,hgroup,menu,nav,section{display:block}body{line-height:1}ol,ul{list-style:none}blockquote,q{quotes:none}blockquote:before,blockquote:after,q:before,q:after{content:none}table{border-collapse:collapse;border-spacing:0}
```

reset.css and your **index.html** file, and you should see that the page looks different: all the default style that the browser applied to the various elements is gone, so now the text in the page all looks like body text. There are various other subtle things that the reset CSS does, like reset the line height and so on, but the most obvious difference will be the font size and font weight.

Now that our page has no style at all, let's build it back up with style rules that work for us!

Setting Up the Style for the Header

Let's begin with the header. We've designed the header so that when the page is greater than a certain width, we display the full name of the company, Dandelion Tours; when the page is less than a certain width, we want to display a short version, which you can think of as the company logo. In our case, we're keeping it simple and just using the letters "DT" (for Dandelion Tours) as our "logo" (in a real company, you'd have a much fancier logo). We'll be using a custom font later for the letters, but for now, we'll use a regular font.

We have the HTML for both the full name (with the class "full") and the logo (with the class "mobile") in the **index.html** file already. We need CSS to style the two headers and to determine when to display the full name and when to display the logo.

Since the header is the same for every page on the site, we'll add the CSS for the header to the **dt.css** file, so go ahead and edit that file, and add this CSS:

CODE TO TYPE:

```
html, body {
    width: 100%;
    margin: 0px;
    padding: 0px;
}
body {
    font-family: sans-serif;
}

/* header */
body > header > div {
    background-color: rgba(39, 69, 189, .9);
    color: white;
    padding: .8em 3% .8em 2%;
}
body > header > div > .name {
    font-size: 3em;
}

a {
    text-decoration: none;
    color: white;
}
a:hover {
    text-shadow: 0px 0px 2px white;
}
```

dt.css and your **index.html** file. The header of the page now has a blue background with white text, and the font size of the text is quite large. You'll also notice that the links in the <nav> part of the page have disappeared; that's because they are colored white so you can't see them.

First, we set both the <body> and <html> elements to be 100% the width of the page, with no padding or margin. This makes sure that all our top-level elements (like the header) will be flush against the sides of the browser window, and take up the full width of the page. We also set the default font-family of the body to a sans-serif font, which is a default font (determined by the browser) with no serifs.

Next, we use a **child selector** to select the elements that hold the full name (the with the class "full") and the logo (the with the class "mobile"). The child selector **>** between the elements says "select the that's directly nested inside the <header> element, which is directly nested inside the <body> element. We use this selector because we've got <header> elements inside our <section> elements lower in the page that might also contain elements. By using this child selector, we ensure that we select the top <header> only. However, notice we're selecting *two* elements with this selector, because they both need to get the same style to set the **background color**, **text color**, and **padding**.

OBSERVE:

```
body > header > div {
    background-color: rgba(39, 69, 189, .9);
    color: white;
    padding: .8em 3% .8em 2%;
}
```

We'll come back and talk more about using **em** as a measurement in CSS later; for now, notice that we use **%** to determine the left and right padding for the header. (Remember: you read the numbers for padding clockwise: top, right, bottom, left.) Try changing the width of your browser (while previewing **index.html)**. As the browser gets really wide, the padding on the left side of the header expands a bit, and when the browser is really narrow, the padding shrinks a bit. This allows us to make use of space a little better—more space looks better when the browser is wide, and less space looks (and works) better when the browser is narrow (and on mobile devices).

If you haven't seen **rgba** used before in CSS, this is simply another way of specifying color by using RGB values, along with a transparency, or **alpha**, value. In this case, 39 is the value for Red, 69 is the value for Green, and 189 is the value for Blue. Together, these RGB values make up the blue color for the header. The alpha is .9, which means it's mostly opaque. However, if we decide later to add an image to the background of our page, we'd be able to see that image—faintly—through the background, which can be a nice effect.

In the next rule, we also use a child selector to select the element with the class "name" that's nested in a in the header. Again, we select two elements with this selector: both of the <a> elements with the class name. We set the **font-size of the elements to 3em**, which equates to three times the font-size of the element in which they are nested, which, as we know from **reset.css**, is 100% the size of the default browser font-size (typically, 16px, unless you've changed it using preferences). Now the heading text will be nice and big, so it looks like a logo at the top of the page:

OBSERVE:

```
body > header > div > .name {
    font-size: 3em;
}
```

Finally, we set the color of the <a> elements in the page to white, remove the underlines, and add a nice shadow to the text when we hover over the links:

OBSERVE:

```
a {
    text-decoration: none;
    color: white;
}
a:hover {
    text-shadow: 0px 0px 2px white;
}
```

You read the **text-shadow** property like this: the shadow is 0px offset from the x-axis (left or right), 0px offset from the y-axis (up or down), it has a 2px blur, and is colored white. Try moving your cursor over the heading at the top of the page and you'll see this effect. It's just a subtle cue to the user that they have moved the mouse over a clickable item. This rule also causes the navigation items have disappeared, but we'll come back and fix that shortly.

Using Media Queries to Set Break Points

Now that we've got the style of the header right, we need to figure out how to display the "full" header when the browser is wide, and the "mobile" header when the browser is narrow or on a mobile device.

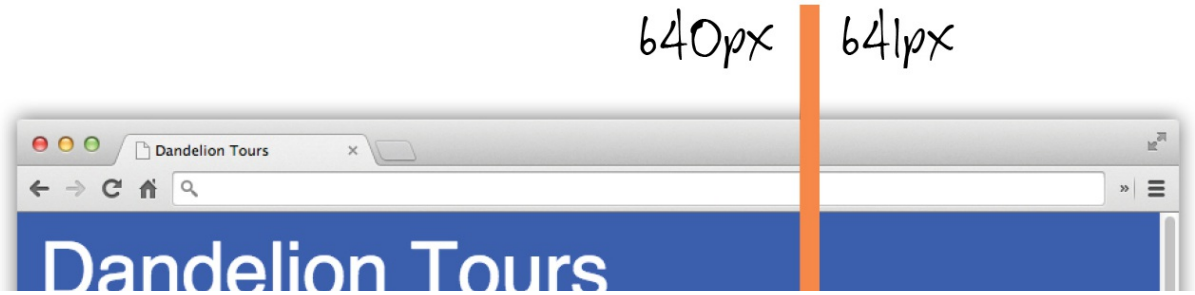
We can do this with *media queries*. With media queries, you can define styles that apply when the browser is a certain width or height, when you're displaying content in a browser with a certain resolution, or even if you're displaying content that is meant to be printed rather than displayed on a monitor.

Most of the time, however, you'll use media queries to change the style that's used when the width of the browser window changes. This is the most common use case for media queries, and it's key to creating responsive web designs.

In our design, we want to use one header ("full") when the browser width is wide, and another header ("mobile") when the browser width is narrow. But what is "wide" and what is "narrow"?

We can decide what is wide and narrow for our display by choosing a width, in pixels, where we switch from wide to narrow. In other words, if the window is wider than this specific width, it's considered "wide" and will use one style; if it's narrower than this specific width, then it's considered "narrow," and will use another style.

Let's look at a concrete example that will help all of this make more sense. In the image below, we've chosen a width of 640px as our *breakpoint*. A breakpoint is the width where we switch from using one style to another. If the browser window is greater than 640px, we use our "wide" style; if the browser window is 640px or less, we use our "narrow" style.



search
Have the experience of a lifetime.
Join us on a tour today.
Print your travel photos with us when you're back home.
Order a tour now!
"My trip to Iceland with Dandelion Tours was the best travel experien e I've ever had!" -- Paula
Tour with us.
Change your life.
Visit spectacular places
With us, you can visit places you might never go on your own. Out-of he-way locations that most tourists never get to see.
Comfort and options for everyone
You'll have a comfortable place to sleep each night, and a wide rang of options for activities during the day.
Knowledgable tour guides
Your tour guides will be able to answer all your questions about the l cation and sights you see.

A sampling of photos from our tours

Dandelion Tours

DT

Menu
Search
Have the experience of a lifetime.
Join us on a tour today.
Print your travel photos with us when you're back home.
Order a tour now!
"My trip to Iceland with Dandelion Tours was the best travel experience I've ever had!" -- Paula
Tour with us.
Change your life.
Visit spectacular places
With us, you can visit places you might never go on your own. Out-of-the-way locations that most tourists never get to see.
Comfort and options for everyone
You'll have a comfortable place to sleep each night, and a wide range of options for activities during the day.
Knowledgable tour guides
Your tour guides will be able to answer all your questions about the location and sights you see.

Right now, the only things that change in the image above when we cross the breakpoint are the header and the navigation, but eventually our entire page will change when we cross this breakpoint.

To use media queries in your CSS, you decide on a breakpoint and write a media query that defines style rules to use below that breakpoint; another media query defines style rules to use above that breakpoint. Here's how we'll set up the media queries in our CSS for a breakpoint at 640px. Add the following code to the end of your dt.css:

CODE TO TYPE:

```
@media only screen and (min-width: 641px) {
    .mobile {
        display: none;
    }
}
@media only screen and (max-width: 640px) {
    .full {
        display: none;
    }
}
```

the **dt.css**file and your **index.html** file. Now, the page should display only one of the headers ("full" or "mobile") rather than both. Make your browser window wide: you see the "full" header, with the full name of the company, Dandelion Tours. Now slowly narrow the width of your browser. As soon as the width of the browser is 640px or less, the "mobile" header replaces the "full" header, showing just the company logo, the letters DT.

Also notice that because we are using just the class names—mobile and full—in the media queries, you see some changes in the way the <nav> looks as the browser width crosses the breakpoint. When the browser is wide, the full list of links to other pages in the site appear under the header (remember, the text is now white so you won't be able to see it), and the search input is below that. When the browser is narrow, both the navigation menu and the search input are replaced by buttons. Take another look at the HTML for the <nav> so you can see why this is the case:

OBSERVE:

```
<nav>
    <div class="menu mobile">
        <button id="menuButton">Menu</button>
    </div>
    <ul class="menu full">
        <li><a href="travel.html">Tours</a></li>
        <li><a href="gallery.html">Gallery</a></li>
        <li><a href="about.html">About</a></li>
        <li><a href="contact.html">Contact</a></li>
    </ul>
    <div class="search mobile">
        <button id="searchButton">Search</button>
    </div>
    <div class="search full">
        <form><input type="search" id="searchInput" placeholder="search"></form>
    </div>
</nav>
```

The styles in the media queries also apply to the two different structures we have for the <nav>; one for wide and one for narrow. This is exactly what we want; we're using just the class names to select the various pieces of the <header> and <nav> to appear/disappear at the wide and narrow settings for the viewport.

How Media Queries Work

Let's take a closer look at these media queries. First, notice that a media query starts with the @media directive. A CSS directive is a "special" command for CSS (in other words, it's not a style rule). There are several directives in CSS (@font-face is another one we'll be using in this course), although as of this writing, not many are supported across all the major browsers. However, the @media directive is well supported.

In both our media queries, we specify that the rules we're defining should apply **only to devices with screens**:

OBSERVE:

```
@media only screen and (min-width: 641px) {
    .mobile {
        display: none;
    }
}
@media only screen and (max-width: 640px) {
    .full {
        display: none;
    }
}
```

The **screen** is one of many media types you can specify in a media query. Other options include print, tv, and even braille. You can also specify **all** if your style is intended for all media types. The media type is required for all media queries but if you don't specify one, the default is **all**. You'll probably use **all**, **screen**, and **print** most often.

Next, we use **and** to combine the media type with a media expression in parentheses. You can combine multiple media expressions with "and" and "or," although typically that's not necessary.

OBSERVE:

```
(media-feature: value)
```

So in our case, the media-feature in the **first media query** is min-width, and the value is 641px, and max-width and 640px in the **second media query**:

OBSERVE:

```
@media only screen and (min-width: 641px) {
    .mobile {
        display: none;
    }
}
@media only screen and (max-width: 640px) {
    .full {
        display: none;
    }
}
```

The first media query says: "Apply the following rules when on a device with a screen, and the viewport is 641 pixels wide or wider." The second media query says: "Apply the following rules when on a device with a screen, and the viewport is 640 pixels wide or narrower."

There are *many* options for media features to allow your media queries to become fairly complex if you need them to be. Some of the other media features are device-height, min- and max-device-width, min- and max-device-height, aspect ratio, color, color-index, resolution, and so on. You can target specific sizes of specific devices with media queries, although in practice, we don't often need to do that.

If we're writing pages that we want to reflow to fill the available width and height, so that the width of the page works in a variety of devices, and the height adjusts appropriately so that we scroll down to see all the content of the page, most of the time we'll use **max-width** and **min-width** as our media feature.

Finally, notice that the media query encloses all the rules that apply to that specific case inside curly brackets {}. Within those brackets, we write the CSS rules, as normal, that will apply only if the media query is applied. We usually indent those rules so we can more easily see that they're part of the enclosing media query. You can include more than one rule in a media query; we'll do that shortly when we add the rules for the <nav> element to these two media queries.

Finishing the CSS for the Nav

Let's go ahead and add the rest of the basic style for the <nav> so it looks a bit better. Add the following code to your dt.css:

CODE TO TYPE:

```
html, body {
    width: 100%;
    margin: 0px;
    padding: 0px;
}
body {
    font-family: sans-serif;
}

/* header */
body > header > div {
    background-color: rgba(39, 69, 189, .9);
    color: white;
    padding: .8em 3% .8em 2%;
}
body > header > div > .name {
    font-size: 3em;
}

a {
    text-decoration: none;
    color: white;
}
a:hover {
    text-shadow: 0px 0px 2px white;
}

/* nav */
nav {
    position: relative;
    padding: .8em 3% .8em 2%;
}

nav a {
    color: rgb(39, 69, 189);
}

@media only screen and (min-width: 641px) {
    .mobile {
        display: none;
    }
    nav > ul.menu > li {
        display: inline-block;
        padding: .2em 1em .2em 0;
    }
    nav > div.search.full {
        position: absolute;
        top: 0px;
        right: 0px;
        padding: .8em 3% .8em 2%;
    }
}

@media only screen and (max-width: 640px) {
    .full {
        display: none;
    }
}
```

Notice the order: we add the new rules that apply to <nav> in both the full and mobile browser widths above the media queries, and then we update the wide media query with new rules for <nav> specifically when the browser is wider than 640 pixels.

□ the **dt.css** file, and □ your **index.html** file.

Let's go through the CSS. We set a relative position for the <nav> because we're positioning the search bar in

the wide mode absolutely, relative to the <nav>. Remember that if you absolutely position an element, you're positioning it relative to the most closely positioned parent element, in this case, the <nav> element.

> **Note** If you have forgotten about how absolute and relative positioning work with CSS, don't worry; we'll come back to that a little later.

The rule that positions the search input is nested within a media query, because we only see the search input when we're in wide mode (using the "full" class on the search input). When we're in the narrow mode, the search <input> disappears, and the <button> we use for the narrow view appears. This happens because our two search options use the "full" and "mobile" classes, so the rules in the media queries apply to search just like they do to the header options.

Finally, we also changed the list of navigation links to be inline-block, which means they appear horizontally instead of vertically, and we now display the links using blue text (the blue matches the blue of the header), so we can see them again.

Multiple Break Points

It is possible to have more than one break point for your page. Especially today when we have such a wide variety of output devices (everything from a tiny mobile phone up to a wide screen monitor or TV screen that displays web pages), you might need more than one breakpoint and more than two styles for your page at different widths.

We'll be using just one breakpoint for Dandelion Tours, but in case you'd like to try a page with multiple breakpoints, here's how you'd do it. Suppose you've got a web page that needs different style rules for a breakpoint at 640 pixels and another breakpoint at 1920 pixels:

OBSERVE:

```
@media screen and (max-width: 640px) {
    /* CSS rules here */
}
@media screen and (min-width: 641px) and (max-width: 1920px) {
    /* CSS rules here */
}
@media screen and (min-width: 1921px) {
    /* CSS rules here */
}
```

Now we have three @media directives, each with their own CSS rules. The first @media directive specifies the style for pages up to and including 640 pixels wide; the second for web pages between 641 pixels and 1920 pixels; and the third for web pages 1921 pixels or wider.

Notice that if the max-width of the first media query goes up to 640 pixels, then the min-width of the second media query must begin at 641 pixels. Similarly, the second media query specifies styles for up to and including 1920 pixels, so the third media query must start its min-width at 1921 pixels. There should be no overlap in the pixel numbers you choose so that the browser knows which media query to use for a given size (with no ambiguity).

Media Queries in HTML

You can also use media queries with the **media** attribute of the <link> element in your HTML. The value of the **media** attribute is the same as the value you use to specify the @media directive in your CSS, except that you don't specify any CSS style rules directly; instead, you select which CSS file to load. For example, if you put your CSS rules for the "full" header in one CSS file, say **full.css**, and your CSS rules for the "mobile" header in another, say **mobile.css**, you'd change your HTML <link> elements like this:

OBSERVE:

```
<link rel="stylesheet" href="reset.css">
<link rel="stylesheet" href="dt.css">
<link rel="stylesheet" href="full.css" media="only screen and (min-width: 641px)
">
<link rel="stylesheet" href="mobile.css" media="only screen and (max-width: 640p
x)">
```

Just like before, you first load **reset.css** and then **dt.css**. Except now the rules for the "full" and "mobile" headers are in two separate files, **full.css** and **mobile.css**, so you load only *one* of these files, depending on the width of the viewport. If the width of the viewport is 641 pixels or wider, **full.css** is loaded. If it's 640 pixels or narrower, **mobile.css** is loaded.

As you can see, this basically accomplishes the same thing as using the @media directive, except that you end up with a lot more CSS files to manage. Now instead of just managing one file, **dt.css**, for the header rules, we have to manage three. This way of specifying media queries is handy if your style rules don't have a whole lot of overlap and are very large. If we put all our rules in one file using @media directives, then we're potentially loading a lot more CSS than we need. By putting the rules in separate files, we can reduce the amount of CSS that we need to load.

However, in our case, the number of rules we'll have in our media queries is fairly small, so it's more efficient for us to combine them into one file, **dt.css**, and use the @media directive to specify which rules to use depending on the width of the viewport.

Media queries are one of the main ways we create responsive web pages. With media queries, we can create different style rules for pages based on properties of the page, like the width of the browser. Media queries support a variety of features that you can use to create rules specific to devices, although we will not do that in this course.

Now that you know how to use media queries to set up rules for the Dandelion Tours page, we're ready to add more style to the page!

Fonts and Measurements

Lesson Objectives

When you complete this lesson, you will be able to:

- add web fonts to your page.
- style text in your page using web fonts.
- size text and spacing in your page using relative measurements.

In this lesson, we'll style the text in the Dandelion Tours page, and also add some spacing to the various sections of the page. To do this, we'll look at how to use web fonts, how to size text correctly, and the best units of measurement to use to create a responsive design.

Getting Your Measurements Right

It's time to get the text on the page set up like we want it. Recall that we're using a **reset.css** to remove any default formatting that browsers add to text (such as a bigger font size and bold for headings), so now it's up to us to add all that back in. In addition, we'd like to use two font families on the page: one for the Dandelion Tours logo, and one for all the other text. We'll focus on getting the font families set up, and then discuss font sizing and style.

Adding Web Fonts to the Page

We web developers used to be fairly limited in the fonts we could use for our web pages, because the fonts had to be available on the end user's computer, and there were only a few fonts we could rely on being there.

Times have changed! Now, browsers support **web fonts**: fonts that can be dynamically downloaded and loaded into the browser. There are plenty of sites where you can download fonts to include in your web page. You then make those fonts available online (on your web server), so users viewing your website can see those fonts. However, another option is to use hosted fonts, such as those hosted at Google Fonts. By using hosted fonts, you don't have to download and host the fonts yourself; you just link to the fonts hosted at a site like Google, and the user's browser downloads the fonts from there.

The two fonts we're going to use for the Dandelion Tours page are Nunito and Dancing Script. Nunito is a sans-serif font, while Dancing Script is a handwriting font. We'll using Dancing Script for the logo, and Nunito for the rest of the page. That way the logo will stand out from the rest of the page well.

Note

If you'd like to go through the process of selecting a font at Google Fonts yourself, you can. However, you may also skip over this note to where we update the code below; we provide the link for the two fonts we're going to use. To select a font using Google's font service, find the font you want to use from the list of fonts on the main page and click **Add to Collection**. The font appears in your collection at the bottom of the page. Then click **Use** (on the right, next to the font name). The next page will show a link to add to your HTML to link directly to the font you want (make sure you selected the **Standard** tab in section 3 on that page to see the link). We use these links below to add Nunito and Dancing Script to Dandelion Tours.

We selected both fonts at Google Fonts, and got the links to add to the Dandelion Tours page. Modify your index.html file as shown::

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Dandelion Tours</title>

<link href='http://fonts.googleapis.com/css?family=Nunito' rel='stylesheet' type
='text/css'>
<link href='http://fonts.googleapis.com/css?family=Dancing+Script' rel='styleshe
et' type='text/css'>

<link rel="stylesheet" href="reset.css">
<link rel="stylesheet" href="dt.css">
<link rel="stylesheet" href="main.css">

</head>
...
```

You added links to the web fonts in your HTML, but that doesn't mean your page will actually use them! To use the web fonts, we need to modify the CSS. These fonts will be used in all the pages on the Dandelion Tours website, so we want to add the CSS to the **dt.css** file. Modify your dt.css file as shown:

CODE TO TYPE:

```
html, body {
    width: 100%;
    margin: 0px;
    padding: 0px;
}
body {
    font-family: Nunito, sans-serif;
}
.name {
    font-family: "Dancing Script";
}
...
```

dt.css and your **index.html** file. Now, the web page changes! The text on the page is in the Nunito font-family, and the company logo is in the Dancing Script font-family.

To apply the Nunito font to the page, we added that font name to the font family property in the rule for the **body** HTML tag, with a fallback to whatever the default sans-serif font is on the user's computer if, for some reason, the browser fails to load the Nunito font from Google. All the other elements in the page will inherit this font from <body>, so this is the only place we need to specify it. (This is the *cascading* in cascading style sheets—CSS!).

To apply the Dancing Script font to the logo, we use the **name** class. Anywhere we want to see text using Dancing Script, we simply add the **name** class to the element. For instance, in the <header>, we're already using the **name** class in both the full and mobile versions of the header:

OBSERVE:

```
<header>
    <div class="full">
        <a href="index.html" class="name">Dandelion Tours</a>
    </div>
    <div class="mobile">
        <a href="index.html" class="name">DT</a>
    </div>
</header>
```

So both the full and mobile versions of the logo now display in the Dancing Script font.

We used a <link> to link to the font families we want to include in the Dandelion Tours web pages. Another

way you can include font families in your pages is to use the **@font-face** CSS directive, adding rules to specify font options in your CSS file instead of your HTML. Using **@font-face** gives you more options over the various font features you can include, but they are also quite a bit more complicated to use. For more on using **@font-face** to specify fonts, check out the MDN page on @font-face.

There are many other useful web font-related resources you might want to investigate. Adobe Typekit is a great resource for finding fonts, creating libraries of fonts, comparing fonts, and so on. If you need a specific font that's not available for free, you can also purchase fonts there. Another popular resource for finding and purchasing fonts is Font Squirrel. Good browser support for web fonts has opened up a whole new avenue for design expression in web pages.

Font Sizes

We've added a few font sizes to our CSS so far, but we haven't set the main font sizes (or other sizing-related properties, like padding) for the basic components of the web page (headings, paragraphs, and such.) and we need to talk about how to size for responsive design.

You might be used to setting your font sizes using pixels:

OBSERVE:

```
font-size: 16px;
```

Using pixels to define your font sizes gives you more control over the text in your web pages and how that text will look in your users' browsers; however, it's not a good idea. When you specify font sizes using pixels, you are making it more difficult for users to change the default font size in their browser, because fonts specified in pixels don't scale well when users change that setting. This is an important setting for accessibility. If you have a hard time reading small text, you might change your browser preferences to use a "minimum font size." We want to make sure our web pages don't break that.

Another reason not to use pixels is because we're using media queries to create a responsive design that works in a variety of screen widths. It's a lot easier to specify font sizes as a relative measurement of a baseline size than it is to figure out different font sizes in pixels for each of the various breakpoints you define in your media queries. In other words, when the page is really wide, we want our program to change the heading to be 1.5 times the standard size, and when the web page is really narrow, change the heading to be .8 times the standard size. That's much easier than figuring out what the exact pixel measurements would be.

When it comes to specifying font sizes—and specifying all kinds of measurements in web pages, like for padding, widths, and so on—you'll find almost as many opinions as there are web developers. However, there are some best practices that are widely agreed upon for solid responsive web designs, so we're going to use those in this course. The first of these is to stay away from pixels for sizing fonts.

So what are the alternatives? Well, other units of measurement include: **%**, **em**, and **rem**. We'll be using all three of these in our design.

In the **reset.css** file we're using, there is a rule that sets the font size for all elements to 100%:

OBSERVE:

```
html,body,div,span, ... {
    font-size: 100%;
    ...
}
```

This sets the baseline font for our pages to 100% of whatever the default font size is set to for the user's browser. That means, if a user has increased her minimum font size, our web page will use that increased size as the default font size. In many browsers, if users do not change the default font size, the default is set to 16 pixels. Let's assume that's the case for our discussion. So, unless we've changed a font size in our CSS, the font sized being used in all our elements is currently 16 pixels.

Now, let's say we want to make the <h1> headings in our page 1.5 times bigger than the body text. Here's where the **em** unit comes in handy:

OBSERVE:

```
h1 {
    font-size: 1.5em;
}
```

The **em** unit is relative to the font-size of the *parent* element of the element to which we're applying it. If the <h1> heading is nested inside the <body>, which has a font size of 100% (typically 16 pixels), 1.5em is 1.5 * 16 = 24 pixels. However, if the user has increased her default font size to 20 pixels, her headings will be 1.5 * 20 = 30 pixels. So using **em** in CSS is useful because you can specify fonts in sizes relative to a baseline size, and those sizes will adapt to any sizes that a user has specified in her browser. We'll use **em** to specify our font sizes, as well as a few other measurements (with one exception that we'll get to shortly). Let's update the CSS to set the font sizes for our headings and paragraphs. We want these sizes to apply to all our web pages (unless we override them specifically), so we'll make these changes in the **dt.css** file. Update the CSS in your dt.css file::

CODE TO TYPE:

```
html, body {
    width: 100%;
    margin: 0px;
    padding: 0px;
    font-size: 100%;
}
body {
    font-family: Nunito, sans-serif;
}
.name {
    font-family: "Dancing Script";
}
p {
    font-size: 1em;
    line-height: 1.2em;
}
h1 {
    font-size: 1.5em;
}
h2 {
    font-size: 1.3em;
}
h3 {
    font-size: 1.1em;
}

/* header */
body > header > div {
    background-color: rgba(39, 69, 189, .9);
    color: white;
    padding: .8em 3% .8em 2%;
}
body > header > div > .name {
    font-size: 3em;
}
...
```

and your **index.html** file. Your headings now look bigger than the body text, and there's a little more spacing between the lines in the paragraphs. By using the **em** measurement for the **line-height** property, we ensure that the line-height stays relative to the font size, so if we later decide to set the baseline font size to 120% instead of 100%, the line-height in the paragraphs will scale appropriately. Remember when we set the font-size of the logo to 3em in the <header> in the previous lesson? Now, you can see that this sets the font size of the logo to three times the baseline font size of the page.

A bit earlier, we said that the **em** font size measurement is relative to the font size of the parent element. What if we haven't set a specific font size for a parent element? For example, if we have an <h1> heading nested in a <header>, and we haven't set a font size for the <header>, how will we know the font size of the <h1> element? Again, this is where the *cascading* comes in. In CSS, some properties of parent elements are *inherited* by child elements; font-size is one of them (the font-size *cascades* down to all the child elements).

So if you set a font-size for the <body>, that will affect all the elements that are nested within the <body>. Unless those elements *override* the font-size, either by specifying the font-size as an absolute measurement using pixels, or with a relative measurement using **em** or **%**, the font-size of the <body> is used for every nested element in the page.

Let's take another look at the rule we used to add padding to the in the <header>. We'll also add another, similar rule to add padding to our headings and paragraphs to give them a little breathing room (right now, they have no padding or margins—thanks to the **reset.css**). Update the CSS in your dt.css file:

CODE TO TYPE:

```
html, body {
    width: 100%;
    margin: 0px;
    padding: 0px;
    font-size: 100%;
}
body {
    font-family: Nunito, sans-serif;
}
.name {
    font-family: "Dancing Script";
}
p {
    font-size: 1em;
    line-height: 1.2em;
}
h1 {
    font-size: 1.5em;
}
h2 {
    font-size: 1.3em;
}
h3 {
    font-size: 1.1em;
}
h1, h2, p {
    padding: .8em 0 .3em 0;
}
/* header */
body > header > div {
    background-color: rgba(39, 69, 189, .9);
    color: white;
    padding: .8em 3% .8em 2%;
}
body > header > div > .name {
    font-size: 3em;
}
...
```

and your **index.html** file. The headings and paragraphs now have a bit more spacing at the top and bottom. We're not setting the left and right padding in this general rule, because we'll be setting that in more specific rules where necessary, for each individual section of the page.

Notice that we use the **em** unit to measure the padding on the top and bottom of our <h1>, <h2>, <p>, and elements, and using 0 (specified by default in pixels, but because it's 0, it doesn't matter), and a **%** measurement for the left and right padding.

We want the spacing on the top and bottom of our elements to be relative to the font size of the parent element (which is going to typically be a , a <section> or a <header>). If the font size of the parent element is increased, then its height increases, and the padding will increase. This allows our designs to stay balanced: if you change the font size of an element, your padding will still look good. We want the spacing on the left and right to be relative to the overall width of the page or the containing element. We could specify this in **em**, but it's a little easier to see and understand what the padding is going to be if we specify it in terms of **%** rather than **em**. Both units of measurement are *relative* to the size of the containing element or page, which is important for responsive design.

This isn't necessarily the choice you'd make for your web pages; you might decide that a fixed width padding

works better for you, in which case you'd want to specify padding in pixels. If you do use fixed width padding, keep in mind that you may need to set different padding sizes for different screen widths using media queries so that the design responds well, and you don't end up with too much or too little space.

Using Rem Instead of Em

Lastly, let's set the style for the <footer> of the page. Font sizes in page footers are often a bit smaller than the rest of the page, so we're going to use a font size of .85 times the default font size for the page. However, we'll run into one small problem with this--we'll be using **rem** units to fix it. Add the following CSS to the bottom of your dt.css file:

CODE TO TYPE:

```
...
body > footer {
    margin: 0px;
    padding: 1.25em;
    background-color: rgba(39, 69, 189, .9);
    color: white;
    font-size: .85em;
}
body > footer div {
    text-align: center;
    padding: 1.25em 0;
}
body > footer .social {
    padding: 0em 1em;
}
```

dt.css and your **index.html** file. Your footer now has a blue background color, and the text in the footer is white and centered. The <footer> element takes up the entire width of the page, so even if you change the width of the browser window, the text should always remain centered. We set the padding for the containing the text to 1.25em at the top and bottom (again, so the top and bottom padding size is relative to the font size), and the left and right padding to 0. For the social media links, which we select with the **.social** class, we use a left and right padding of 1em to add space between the links. Because we're specifying the spacing using **em**, if we change the font size of the text in the footer later, this padding will remain proportional to the size of the text.

Notice, however, that the Dandelion Tours logo in the footer is now a tad small. The Dancing Script letters are a bit smaller compared with Nunito letters of the same font size. We've nested the logo in a element with the class **name**, so we can select this and increase the font size of the logo in the footer to adjust for this. We could use **em** to specify the font size of the logo in the footer. Update your dt.css file like this:

CODE TO TYPE:

```
body > footer .social {
    padding: 0em 1em;
}
body > footer .name {
    font-size: 1.5em;
}
```

Here, we set the font size of the element in the footer with the class "name" to 1.5 times the font size of the parent element. Well, the parent element is the footer, and the font size of the footer is .85 times the font size of *its* parent, which is the body, whose font size is the default font size of the browser. So 1.5 * .85 = 1.275, so the font size of the logo in the footer will be 1.275 times the default font size of the browser.

This will work fine, but we might decide that we want to set the font size of the logo in the footer relative to the font size of the body instead, rather than the font size of the footer. That will make it just a little easier to figure out what that font size will actually be (which mens we'll have one less multiplication to do!). How do we do that? With **rem**. Update your dt.css file::

CODE TO TYPE:

```
body > footer .name {
    font-size: 1.5em1.2rem;
}
```

dt.css and your **index.html** file. This specifies that the element with the class **name** in the <footer> should have a font size of 1.2 times the font size of the **root element**; in other words, the body. The "r" in **rem** means "root." The result is that if we come along later and change the font size of the footer, the font size of the logo will *not* change. You might decide that's not what you want, in which case you should use **em** instead of **rem**, but for this example, we'll stick with **rem**.

Font Formats

In this lesson, we used the Google Web Fonts service to provide fonts for the page. But what exactly is the URL that we're linking to at Google Web Fonts doing? Well, it's generating some CSS that uses the **@font-face** CSS directive, which is how the font gets loaded, and how we can refer to it in our CSS. For instance, if you put the URL of the link to the Nunito font in your browser, you'll see something like this:

OBSERVE:

```
@font-face {
    font-family: 'Nunito';
    font-style: normal;
    font-weight: 400;
    src: local('Nunito-Regular'),
         url(http://fonts.gstatic.com/s/nunito/v7/mUangSEE-PUZJVLGz-Lt6-vvDin1pK
8aKteLpeZ5c0A.ttf) format('truetype');
}
```

Linking to the font at Google is easier than adding the **@font-face** to your CSS directly, but behind the scenes, we actually are using **@font-face**.

Included in the rule is a property to tell our CSS where to get the font (in the **src** property, with the **url** property value), and this links to a font in the **TrueType** format.

There are several different formats that are used to create web fonts, including TrueType, Web Open Font Format (WOFF), and Open Type (OTF). TrueType was created by Microsoft, but oddly enough, Internet Explorer didn't support TrueType until recently. Fortunately, Internet Explorer 9 and later now supports TrueType, so you can use TrueType safely for all modern browsers. However, if you know that you want to support users who may be using older versions of Internet Explorer, you'll need to write your own @font-face rules and provide backup font formats for those people.

Take a look at caniuse.com to see which browsers support which font formats:

- TrueType
- WOFF
- SVG
- EOT

Many people who use special fonts in their web pages are transitioning to WOFF because that format is well-supported across all browsers, both desktop and mobile. WOFF is also standardized by the W3C. If you are concerned that a font you're using may not work in some browsers, make sure you provide a backup. Update your dt.css file, like this:

CODE TO TYPE:

```
...
body {
    font-family: Nunito, sans-serif;
}
.name {
    font-family: "Dancing Script", cursive;
}
...
```

- **dt.css** and your **index.html** file. Nothing changes if your browser supports the two web fonts, but for users who may be using much older browsers, the body will be displayed in a sans-serif font (rather than the default serif font), and the logo will now use a cursive font (whichever cursive font is the default in their browser).

For a much deeper study of fonts and typography in responsive design, I recommend the book Responsive Typography (O'Reilly, 2014). Typography in general, and web fonts specifically, are deep topics that can keep you busy for a long long time.

Page Layout for Responsive Design

Lesson Objectives

When you complete this lesson, you will be able to:

- use CSS positioning, table display, and flex box to create responsive layouts.
- use CSS positioning to position elements precisely within a container element.
- use CSS table display to create even columns of content.
- use CSS flex box to create flexible columns of content.
- create backgrounds for elements that flexibly resize as the page resizes.

Creating a Responsive Layout

Taking Stock

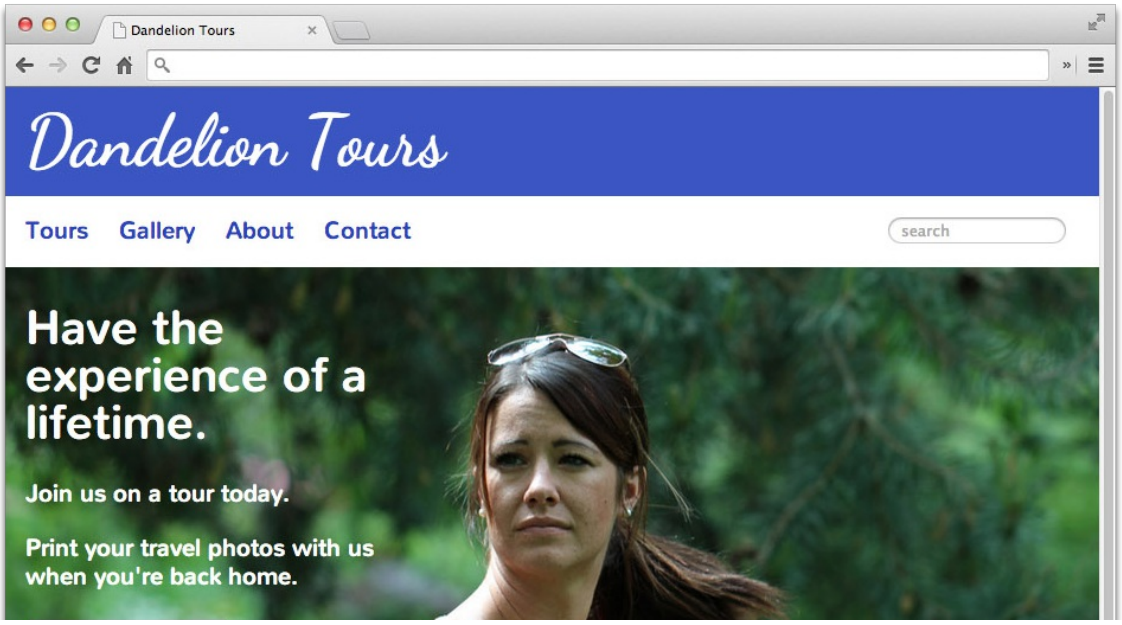
Let's take stock of where we are with the Dandelion Tours page. We've got the header laid out (although we'll be revisiting the header later when we fix the buttons for the mobile version of the site), and we have the fonts and font sizes sorted out, so now it's time to tackle the main part of the page (the part below the header and above the footer).

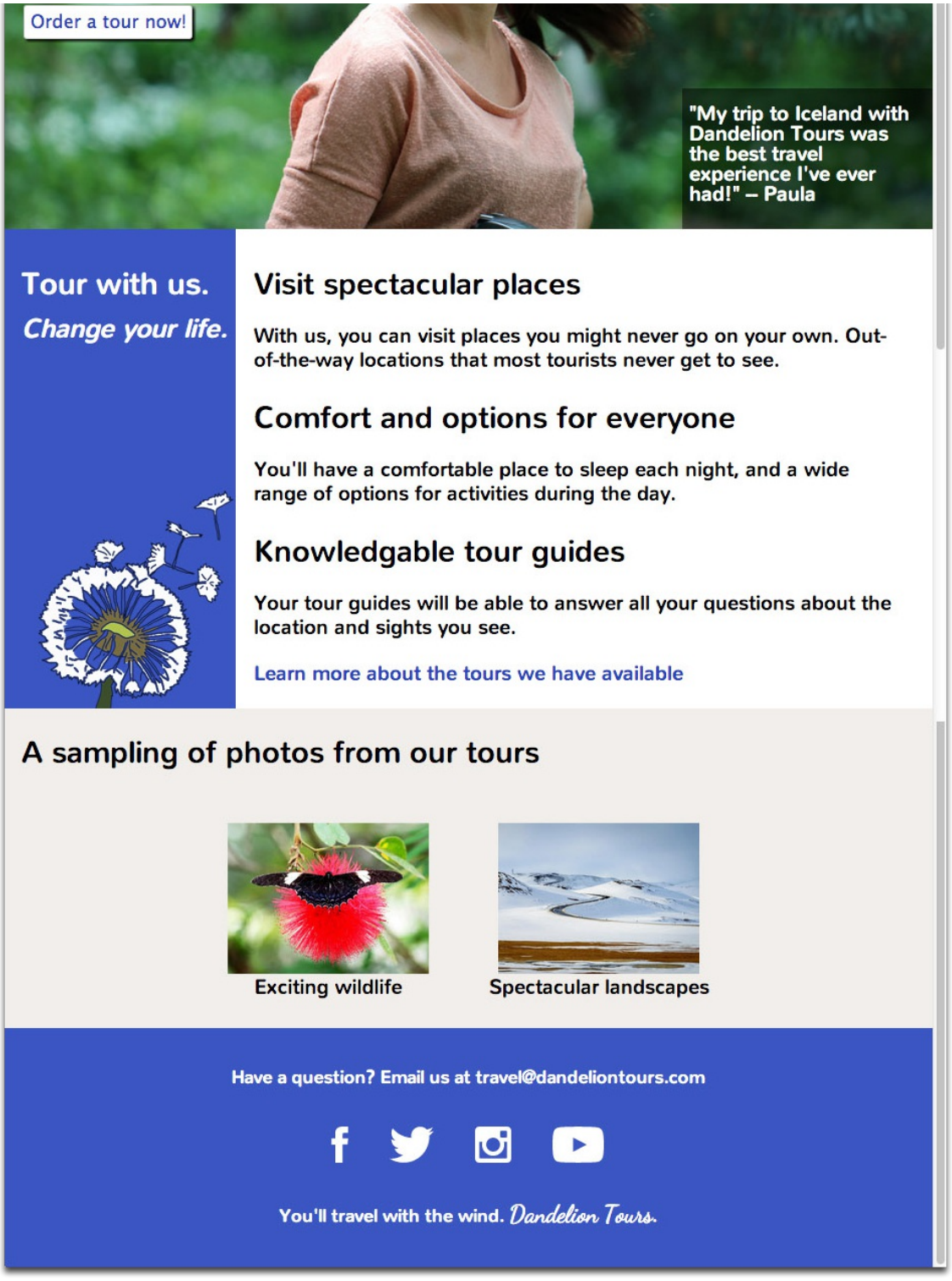
Recall that the main part of the page is split up into three sections: experience, details, and photos.

OBSERVE:

```
...
<section id="experience">
   ...
</section>
<section id="details">
   ...
</section>
<section id="photos">
   ...
</section>
...
```

Eventually, we'd like the main part of the page to look like this:





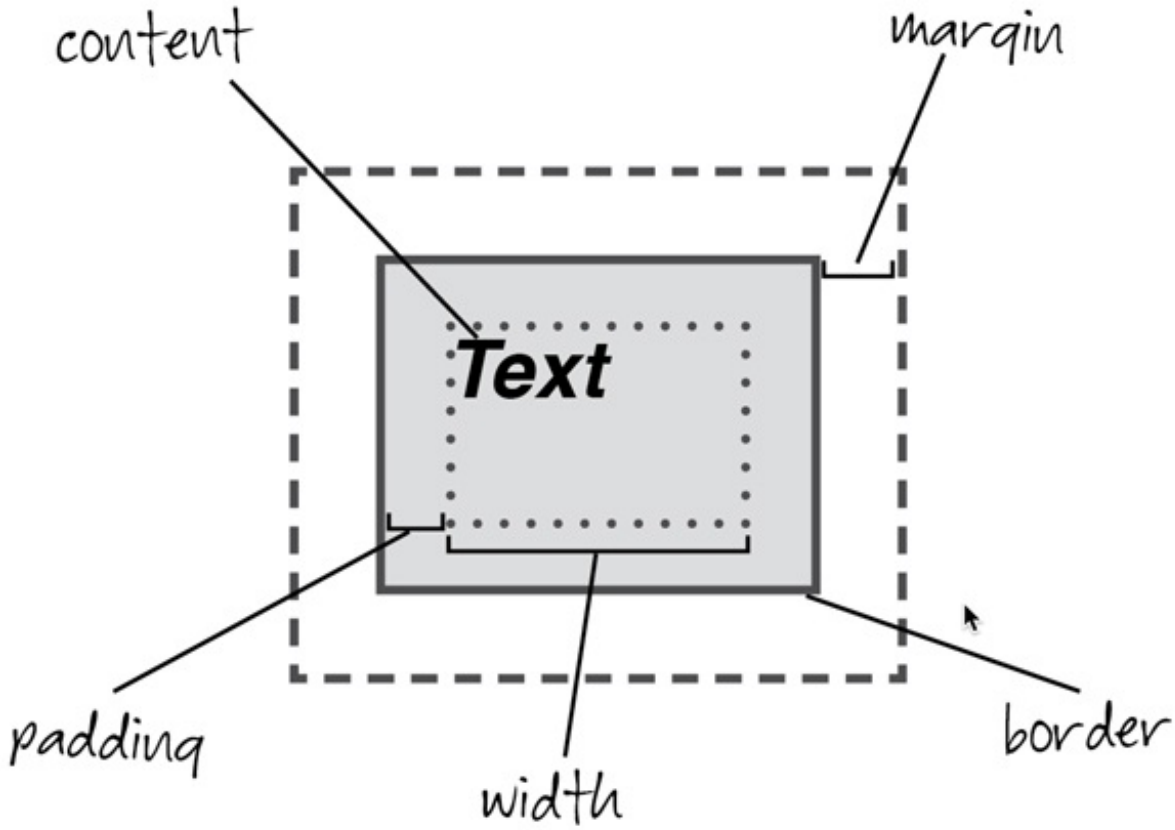
So, we need to lay out the content of the page. Right now, the content is just flowing into the page (using the basic default rules for block and inline elements). We're going to change that by positioning various elements in the page.

You may remember that there are a variety of techniques for using CSS to lay out content in a page. We're going to look at three techniques, all of which we'll end up using in our design for laying out the "wide" view of the page. Then we'll come back in a later lesson and change the layout for the "mobile" view.

Of course, our goal is to create a layout that works well with a variety of different sizes of browser windows. We want to make sure the page looks good when it's wide and when it's narrow, and everywhere in between. That's what responsive design is all about.

A Brief Review of the Box Model

Let's do a brief recap of the box model that CSS uses to determine how elements are presented in the page. It's a good idea to have this in mind while creating a layout for a web page.



Every block element (like <section>, , and <p>.) is composed of the content itself (usually text), padding (between the content at the border), the border, and margin (between the border and the edge of the element, acting as spacing between elements). Not all elements have padding, border, and margin, but you can set values for all these parts of the box around the content using CSS.

There are several different units of measurement we can use to specify the size of the padding, border, and margin. If you want precise measurements in pixels (which you'd want if you were creating a *fixed width* page design), you use the "px" measurement. If you want the size to be relative to the width and height of the element's container, you can use %. For instance, if you have a nested in the <body>, and you set the padding to 10%, the left and right spacing will be 10% of the width of the page, and the top and bottom spacing will be 10% of the height of the page.

Also, remember that when setting the width of an element, you are setting the width of the content area. Padding, border, and margin are extra space that get added onto the total width of the element once it's laid out in the page. That means when calculating total pixels or percentages that your element occupies in the page, you must include the size of the padding, border, and margin.

Okay, with that out of the way, let's get back to creating the layout for the Dandelion Tours page.

Laying Out the Experience Section with CSS Positioning

Let's begin with the experience section; here's all the HTML for this section.

OBSERVE:

```
<section id="experience">
    <div>
        <h1> Have the experience of a lifetime. </h1>
        <p> Join us on a tour today. </p>
        <p> Print your travel photos with us when you're back home. </p>
        <p> <button class="order">Order a tour now!</button> </p>
    </div>
    <aside>
        "My trip to Iceland with Dandelion Tours was the best travel
        experience I've ever had!" -- Paula
    </aside>
</section>
```

Not only do we need to style the text in this section properly, we also need to add a background image to it, and move the <aside> over to the right side of the page. That <aside> is going to need to move if we resize the page, so we'll need to take that into consideration as we build the CSS. In addition, we'd really like the image to always be sized correctly so that it fills the width of the page (so we don't get any extra white space to the right of the image if we make the page wider, or a horizontal scroll bar along the bottom of the page if we make the page narrower).

You've probably noticed that we're not using an <img> element for the background image for this section. Instead, we'll define a background image for the "experience" <section> element using CSS. The **background-image** property has been included in CSS for a long while now, but more recently (with CSS3), browsers are supporting a variety of properties that allow you to position and size background images more easily, which makes using the **background-image** property more appealing in responsive designs.

So, first, let's get the background image working for the experience section. We'll be adding the CSS to the file **main.css**, because this CSS applies only to the home page. Go ahead and open up your empty **main.css** and add the following CSS to your main.css file:

CODE TO TYPE:

```
section#experience {
    position: relative;
    color: white;
    padding: 0em 60% 2em 2%;
    background-image: url("https://courses.oreillyschool.com/html5_responsive_de
sign/images/TourGuide.jpg");
}

section#experience h1 {
    font-size: 2em;
}
```

main.css and your **index.html** file. You now see the image in the background of the experience section, and now the text of the experience section is white, so it's easy to read on top of the fairly dark image. We also increase the font size of the heading to 2em (overriding the rule in **dt.css** that we added in the previous lesson to set the font size of <h1> headings to 1.5em). Notice that we're adding a 60% right padding and a 2% left padding to the experience section. Using a % to measure means the padding expands and shrinks just a bit as we widen or narrow the page (just like we did with the header). The 2% deliberately matches the 2% left padding we added previously to both the <header> and the <nav> so that the text in the header, navigation and experience sections of the page all line up on the left.

Note Background images of elements fill the content area of elements, as well as the padding (but not the border or margin areas). So we use padding to add space around the text, while making sure the background image fills the entire width of the page.

When you stretch your browser quite wide, you'll see that the bakground image repeats. That's because the default value for background-repeat is "repeat," so if we don't specify something different, the background image will repeat if the image doesn't fill the element entirely. In addition, as you stretch the browser wide, the text in the experience section stretches with it, to fill the available space within its part of the page (which is 38% of the browser window width, because of the 2% left padding and the 60% right padding), and we lose some of the vertical space for the section.

It would be a lot nicer if we could make the image stretch to fill the available space, and also make sure that we have a minimum height for the section so it looks better when the browser is wide. Update the CSS in your main.css file:

CODE TO TYPE:

```
section#experience {
    position: relative;
    color: white;
    padding: 0em 60% 2em 2%;
    background-image: url("../images/TourGuide.jpg");
    background-size: cover;
    min-height: 400px;
}

section#experience h1 {
    font-size: 2em;
}
```

main.css and your **index.html** file. Widen, and then narrow, your browser window. The background image will stretch to match the width of the browser, but the section won't get shorter than 400 pixels tall, which keeps the section looking good.

When you set **background-size** to "cover," you're telling the browser to scale the image down as small as possible, but making sure that it still covers the background of the element you've selected. And of course, **min-height** makes sure the height of an element doesn't go below the value specified. This height works well for desktop browsers in the "wide" view but we might want to reconsider this property for the "mobile" view later.

So, what about the <aside> element? Here's where the *absolute positioning* comes in, the first positioning technique we're using in this lesson. We want the aside content to hug the bottom-right corner of the experience section. Notice that we added the rule **position: relative;** to the experience section. That means, if we set the position of the aside to "absolute," we can position it wherever we want *relative* to the experience section—in other words, at the bottom right of the section element, which is where we want it. Here's how. Update the CSS in your main.css file:

CODE TO TYPE:

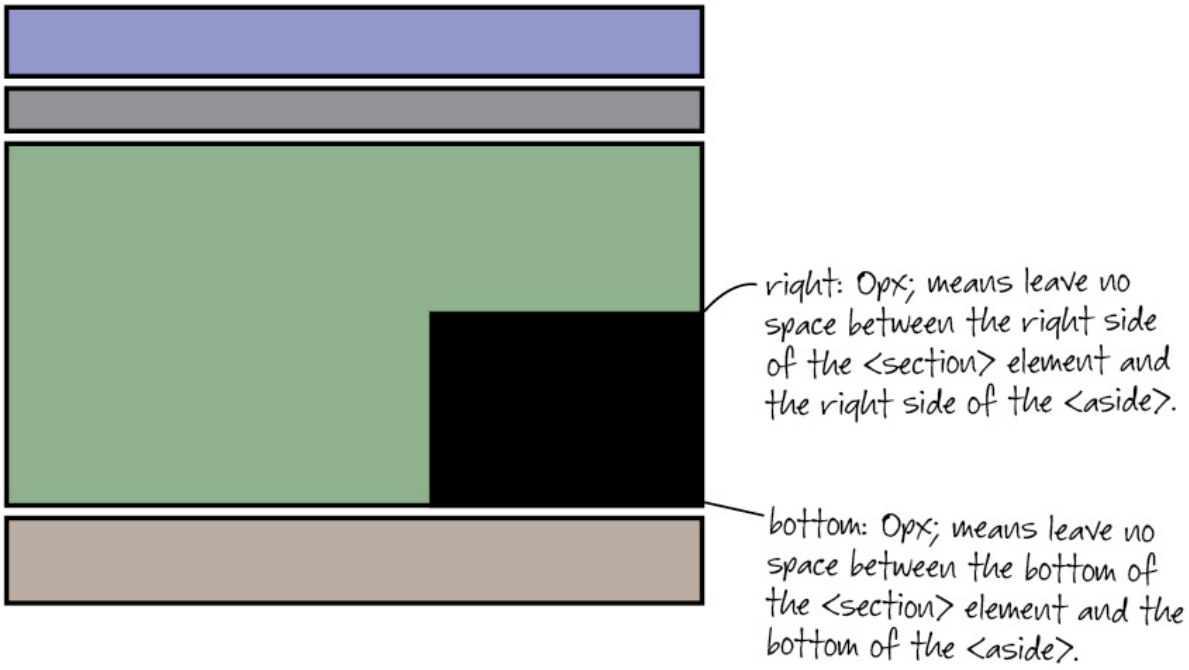
```
section#experience {
    position: relative;
    color: white;
    padding: 0em 60% 2em 2%;
    background-image: url("https://courses.oreillyschool.com/html5_responsive_de
sign/images/TourGuide.jpg");
    background-size: cover;
    min-height: 400px;
}

section#experience h1 {
    font-size: 2em;
}

section#experience aside {
    position: absolute;
    bottom: 0px;
    right: 0px;
    width: 25%;
    padding: .8em 3% 1.2em 2%;
    background-color: rgba(0, 0, 0, .5);
}
```

main.css and your **index.html** file. You now see the aside content at the bottom right of the experience section. Make your browser a lot wider, and then a lot narrower. Notice that it sticks to the bottom right of the section, and also changes width just a bit depending on the width of the page—we do this so that the width fits better with the overall width of the page, changing as the width of the page changes.

Let's take a closer look at how we positioned the <aside> element. First, notice that the experience section is

positioned relative, but has no offsets specified (no left, right, top, or bottom properties that would shift the element out of its normal position in the flow of the page), so it sits where it normally would in the flow of the page. This is exactly what we want.

Even though setting the position of the <section> to relative doesn't change the experience section at all, it does allow us to position the <aside> element that's directly nested within the experience <section>. When you position something absolutely, you take it completely out of the flow of the page: it's like the element is sitting on top of the rest of the page. When you provide an offset for the absolutely positioned element, such as with top, bottom, left, or right, that shifts the element by the amount you specify, but shifts it from where? Well, if the absolutely positioned element is nested within another positioned element (like the experience section), the measurements you specify with top, bottom, left, or right shift the element with respect to that parent element. In our case, we're specifying that we want the <aside> to appear 0 pixels from the bottom of the experience section, and 0 pixels from the right of the section. So, the <aside> sits snugly at the bottom right corner of the experience section:



Notice that we've given the <aside> some padding: 2% on the right means that the space between the right side of the text in the <aside> and the right side of the browser window will match the space between the text in the experience section and the left side of the browser. That helps to keep things matched up and balanced looking. We've also set the background color of the <aside> to black, with a .5 alpha value (meaning, the <aside> is 50% transparent) so we can see the background image partially through the background.

Before we leave the experience section, we'll add just a bit of CSS for the order button. This style isn't specifically related to responsive design, so we won't go through it in detail. Update the CSS in your main.css file:

CODE TO TYPE:

```
section#experience {
    position: relative;
    color: white;
    padding: 0em 60% 2em 2%;
    background-image: url("https://courses.oreillyschool.com/html5_responsive_de
sign/images/TourGuide.jpg");
    background-size: cover;
    min-height: 400px;
}

section#experience h1 {
    font-size: 2em;
}
section#experience button.order {
    margin-bottom: 8em;
    padding: .3em;
    background-color: white;
    border: 1px solid silver;
    border-radius: .2em;
    box-shadow: 2px 2px 3px black;
    color: rgb(39, 69, 189);
}
section#experience aside {
    position: absolute;
    bottom: 0px;
    right: 0px;
    width: 25%;
    padding: .8em 3% 1.2em 2%;
    background-color: rgba(0, 0, 0, .5);
}
```

main.css and your **index.html** file. The order button in the experience section should now look a bit better, with rounded corners (**border-radius**), and better colors to match the rest of the page.

Laying Out the Details Section with Table Display

Next up is the details section. Here's a reminder of the structure and content of that section:

OBSERVE:

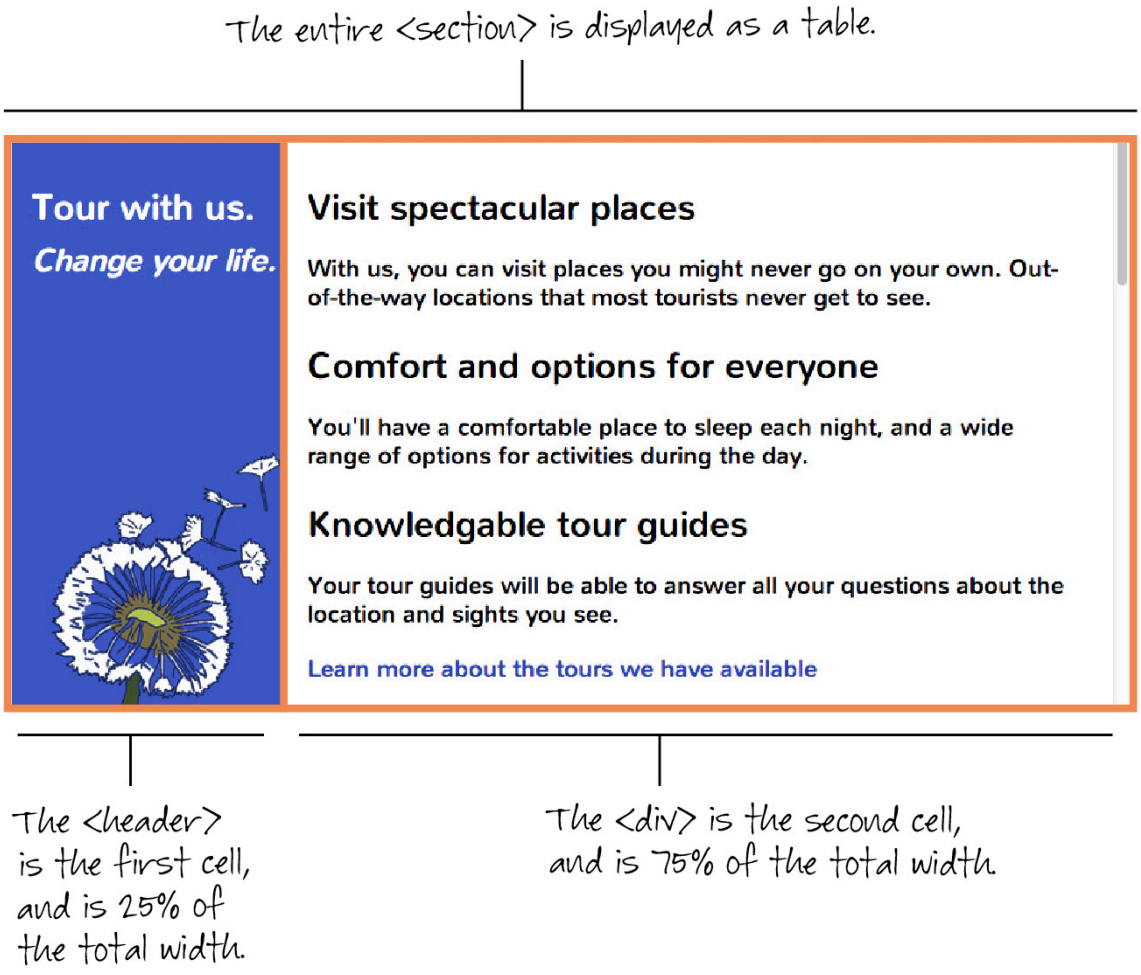
```
<section id="details">
    <header>
        <h1>Tour with us.</h1>
        <h2>Change your life.</h2>
    </header>
    <div>
        <ul>
            <li>
                <h1>Visit spectacular places</h1>
                <p>
                    With us, you can visit places you might never go on your own
.
                    Out-of-the-way locations that most tourists never get to see
.
                </p>
            </li>
            <li>
                <h1>Comfort and options for everyone</h1>
                <p>
                    You'll have a comfortable place to sleep each night, and
                    a wide range of options for activities during the day.
                </p>
            </li>
            <li>
                <h1>Knowledgable tour guides</h1>
                <p>
                    Your tour guides will be able to answer all your questions
                    about the location and sights you see.
                </p>
            </li>
            <li>
                <p>
                    <a href="travel.html">Learn more about the tours we have ava
ilable</a>
                </p>
            </li>
        </ul>
    </div>
</section>
```

With this part of the page, we'd like to lay out the section so that the <header> of the section is on the left, and the containing the rest of the content of the section is on the right. We could try to float the <header> to the left, or absolutely position the two parts, but to get a solid layout with even columns, we have a better option: the table display.

Now, I know what you're thinking: we don't want to use tables to display content that isn't tabular. That's true. Back in the day, people had to use the HTML <table> element (along with the <tr> and <td> elements) to make even columns for content, because there was no way to create that presentation with CSS. That technique went by the wayside as we learned to separate content and structure cleanly from presentation. So today, we use tables only to display content that truly is tabular (like scores from a hot dog eating contest, for example).

However, around the same time that we stopped using tables to present content, we realized that we still needed a way to make even columns for our content, and thus the table display was born in CSS. This technique for styling and presenting content has been supported by browsers for a while now, so it's safe to use table display for all the major desktop and mobile browsers.

Just like with HTML tables, we will display our content using table cells, but we'll specify which elements are table cells using CSS instead of HTML. For our example, we need only one row in our table, with two table cells:



We can use any existing block elements for the table itself, and the table cells nested inside the table. Our existing structure for the details section works perfectly: we will use the <section> element for the table, and the <header> and elements as the two table cells. Add the following CSS to your main.css file, below the CSS you've already added for the experience section:

CODE TO TYPE:

```
...
section#details {
    display: table;
}
section#details header {
    display: table-cell;
    background-color: rgba(39, 69, 189, .9);
    background-image: url("https://courses.oreillyschool.com/html5_responsive_de
sign/images/dlogo_small.png");
    background-position: right bottom;
    background-repeat: no-repeat;
    width: 25%;
    color: white;
    padding: .2em .2em .2em 2%;
}
section#details header h1 {
    padding-top: 1em;
}
section#details header h2 {
    font-style: italic;
    padding: .3em 0;
}
section#details div {
    display: table-cell;
    padding: .8em 3% .8em 2%;
}
section#details a {
    color: rgb(39, 69, 189);
}
```

main.css and your **index.html** file. Now your details section looks completely different. The <header> appears on the left side of the page, while the appears next to it on the right side of the page. If you change the width of the browser, the two columns for these two parts of the details section remain even on the bottom.

Here's what we did to get the table layout for the parts of the details section:

OBSERVE:

```
section#details {
    display: table;
}
section#details header {
    display: table-cell;
    ...
}
section#details div {
    display: table-cell;
    ...
}
```

Compare this to the diagram above, and you can see that we created the **table** from details <section> element, and the two **table cells** using the <header> and elements. We could have added another element to act as a row, but CSS is smart enough to figure out what we want without that, so why add more structure? The great thing about using table display is that we get two even columns with very little effort, and no changes to our HTML.

Let's take a quick look at the CSS we used to create the background of the header on the left side of the details section:

OBSERVE:

```
section#details header {
    ...
    background-color: rgba(39, 69, 189, .9);
    background-image: url("https://courses.oreillyschool.com/html5_responsive_design/images/dlogo_small.png");
    background-position: right bottom;
    background-repeat: no-repeat;
}
```

We've added both a **background color** and a **background image** to the <header> element in the details section. The background image is a PNG, and has a transparent background, so that's why the color shows through. Notice that we position the background image at the bottom right of the header, and set the repeat to "no-repeat." That means the background image will stick to the bottom right of the header. We set the width of the <header> to 25%, and leave the rest of the width for the :

OBSERVE:

```
section#details header {
    ...
    width: 25%;
}
```

That means as you expand or shrink the web page, the <header> and will expand or shrink too, but the background image for the <header> should always be at the bottom right. If you shrink the browser so that it's very narrow, the background image falls off the side, but we'll figure out how best to deal with that later, when we talk about layout for the narrow/mobile version of the site.

Laying Out the Photos Section with Flex Box

The final piece of the main part of the page is the photos section. Here's what that content and structure looks like:

OBSERVE:

```
<section id="photos">
    <header>
        <h1>A sampling of photos from our tours</h1>
    </header>
    <div class="photo-container">
        <div class="photo-with-caption">
            <figure>
                <img src="https://courses.oreillyschool.com/html5_responsive_design/images/Brazil_Butterfly_m.jpg" alt="photo of mountains and rainbow">
                <figcaption>Exciting wildlife</figcaption>
            </figure>
        </div>
        <div class="photo-with-caption">
            <figure>
                <img src="https://courses.oreillyschool.com/html5_responsive_design/images/Iceland_Road_m.jpg" alt="photo of road in snow">
                <figcaption>Spectacular landscapes</figcaption>
            </figure>
        </div>
        <div class="photo-with-caption">
            <figure>
                <img src="https://courses.oreillyschool.com/html5_responsive_design/images/Alaska_View_m.jpg" alt="photo of rainbow and waterfall">
                <figcaption>Incredible views</figcaption>
            </figure>
        </div>
    </div>
</section>
```

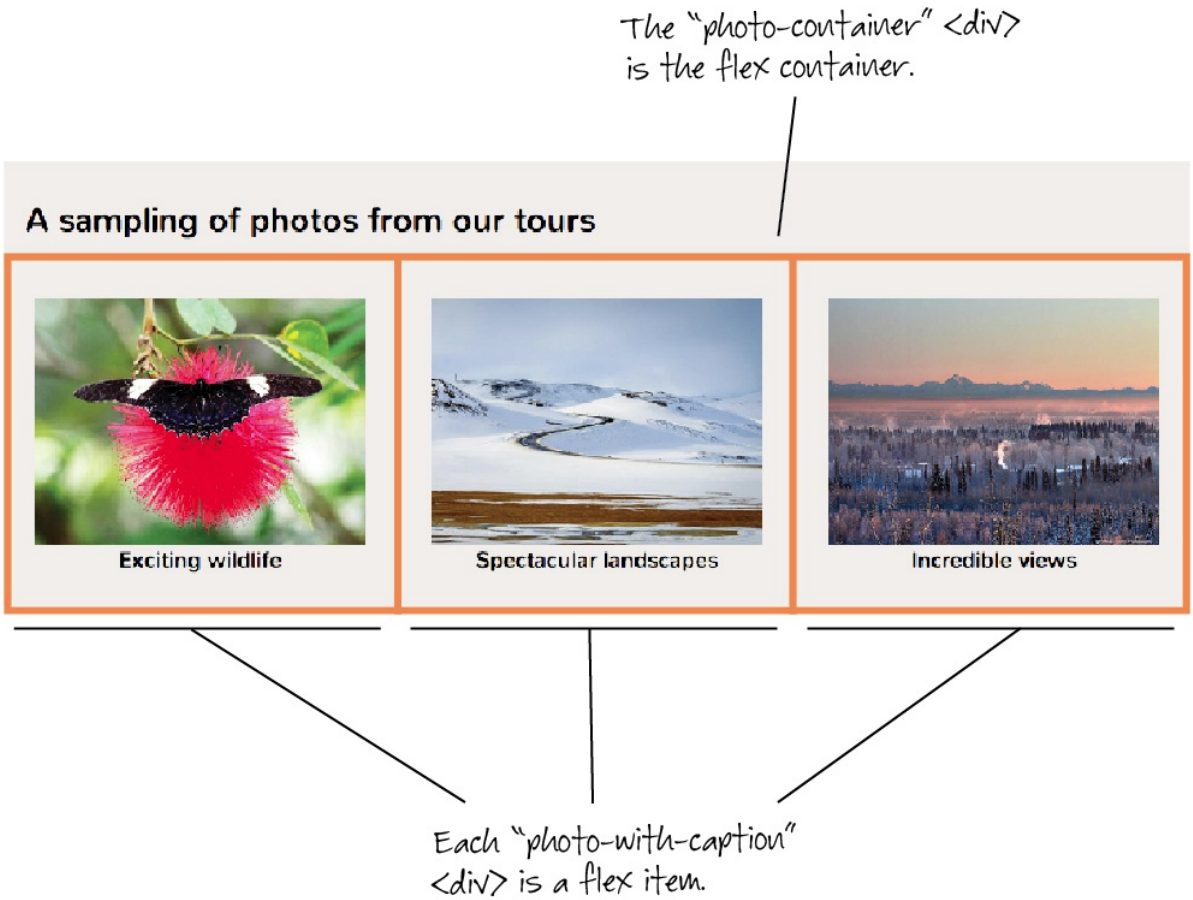
Take a close look at the structure of this part of the page. The photos section is defined with a **<section>** element, like the other sections of the page, and within that we have a **<header>**, and a ** ** that contains the photos, with the class **"photo-container"**. Each photo consists of another ** **, with the class **"photo-with-caption"**, containing a **<figure>**, which in turn contains an **<img>** and **<figcaption>**. Using <figure> and <figcaption> with your images is good practice because it's meaningful information the browser can use to know that your content includes an image (for instance, this information can be used in accessible browsers, like screen readers, to make the content more meaningful to listeners).

Our goal for the photos section is to display the three images side by side in the section when the browser is in the wide mode, and vertically (on top of one another) in narrow/mobile mode. We'll also add a new middle-width mode specifically for this section, to display two of the three photos, side by side, when the browser is between 641px and 1000px wide (as shown above in the screen shot).

To do the layout for the middle-wide and wide modes, we'll use a CSS feature called *flex box*. Similar to table display, flex box has a container with items that you can lay out horizontally and vertically, except that flex box gives you even more flexibility than table display. Flex box is a more recent addition to CSS, and went through a couple of variations before being nailed down, but all the major browsers now support it in their most recent versions, although IE 10, and (as of this writing), Safari, and iOS still require a browser prefix on the CSS property. You'll see how to use that in just a moment.

Note IE 9 and earlier do not support flex box, so make sure you're using the latest version of IE. If you're concerned about users on older browsers when implementing your site, keep this in mind if you want to use flex box.

To use CSS flex box, we need a flex box container, and flex box items that will get laid out within that container (in this way, it's similar to table display). Looking at our HTML, we're going to use the "photo-container" for the container, and each "photo-with-caption" will be an item:



Here's the initial CSS. Add the following CSS to your main.css file, below the CSS you've already added for the experience section:

CODE TO TYPE:

```
...
section#photos div.photo-container {
    display: -ms-flex;
    display: -webkit-flex;
    display: flex;
    -webkit-justify-content: center;
    -ms-justify-content: center;
    justify-content: center;
    align-items: flex-start;
    margin: auto;
}
```

main.css and your **index.html** file. The three images in the photos section now appear in a horizontal arrangement in the page, however, they are far too big and overlap quite a bit. We'll fix that in a moment, but first, let's step through some of the flex box properties.

To set up the "photo-container" as the flex box container, we set the **display** property to "flex" (similar to how we set up the table display earlier by setting the **display** property to **table**). However, we don't have to identify the "photo-with-caption" s within the flex box container specifically as flex box items; any block elements nested directly within the container will be used automatically as the items for the flex box container.

Notice that we are setting **display** *three* times. What's going on here? Recall that, as of this writing, IE10 and Safari for desktop and iOS require that we use the **vendor prefix** on the **flex** property value. Browsers use these vendor prefixes—"-ms-" for IE and "-webkit-" for Safari—when the browser is first implementing features that aren't yet set in stone in the W3C CSS specification. They are an indication to developers that these CSS features may change and that if not all browsers implement the feature yet, you might have to find a workaround for that feature. In the case of flex box, all the browsers implement the feature, but IE10 and Safari still require the vendor prefixes in order for the feature to work (Firefox and Chrome do not).

We use a specific ordering for the three **display** properties:

OBSERVE:

```
display: -ms-flex;
display: -webkit-flex;
display: flex;
```

We put the two versions with the vendor prefix first. Browsers read the CSS top down, so Safari will ignore "-ms-flex" and apply "-webkit-flex" because Safari recognizes that property value. Then it will ignore the last "flex" because it doesn't recognize that value. The same thing will happen with IE10, except it will use "-ms-flex". Chrome and Firefox will ignore both the previous vendor-prefixed property values and just use "flex." We put "flex" at the bottom so that when Safari and IE (which actually does use "flex" in IE11) eventually *do* recognize "flex," that's the last property in the list.

The **justify-content** property tells the flex box how to justify the items in the flex box container. We want the images to stay centered in the container, so we use the value **center**:

OBSERVE:

```
-webkit-justify-content: center;
-ms-flex-pack: center;
justify-content: center;
```

For this property, we use the browser extensions on the properties, rather than the property values. And notice that IE10 and Safari used two different properties to justify content before the name of the property was finalized. However, the agreed upon name is now **justify-content**, so we define this property last in the list so browsers that don't recognize **-webkit-justify-content** and **-ms-flex-pack** use this property.

There are lots of other options for this property—as we said earlier, flex box is designed to be flexible in the varities of layouts you can create with it. For instance, we could make sure the images are evenly spaced within the container by using the property value **space-around**, and right-justified or left-justified using the property values **flex-end** or **flex-start** respectively.

The last flex box property we specify is the **align-items** property:

OBSERVE:

```
align-items: flex-start;
```

We use **flex-start** as the value for this property, which aligns all the items in the container at the top. We know that our images are all the same aspect ratio, so aligning them at the top means that all columns of the container will be equal height. **flex-start** is actually the default value for **align-items** if you don't specify a value. However, we want to show you an example of a flex box property that was added later to the specification. This property doesn't have an equivalent vendor-prefixed version. So, browsers that don't support the property (IE10 and Safari), will just ignore it. That's okay though, because we get the right behavior anyway, since all our images are the same width and height.

Okay, now we need to fix the problems we still have with our flex box lay out: we need to add some spacing between the images, and also size the images correctly. Add the following CSS to your main.css file:

CODE TO TYPE:

```
...
section#photos div.photo-container {
    display: -ms-flex;
    display: -webkit-flex;
    display: flex;
    -webkit-justify-content: center;
    -ms-justify-content: center;
    justify-content: center;
    align-items: flex-start;
    margin: auto;
}
section#photos div.photo-with-caption figure {
    padding: 1.5em;
    text-align: center;
}
section#photos div.photo-with-caption img {
    width: 300px;
}
```

main.css and your **index.html** file. Now the photos look *much* better. You see three smaller images, across the width of the page. If you expand the page, the images will stay evenly spaced, centered in the container, with their container taking up the full width of the page. It looks great! We also aligned the images and image captions to the center of the <figure> elements, by adding the property **text-align** with the value **center** to the rule for the **figure** elements. That means all inline elements (including text and images) within any element nested inside the **figure** will be aligned center, which is what we want.

We can still improve our images. We made the images a specific width: 300 pixels. This width works fine when the web page is wider than 980 pixels or so, but below that the images start to jam together and overlap. We could use a media query to change the pixel width of images at different page widths, but there's a much easier way to accomplish what we want. Try this instead. Update the CSS in your main.css file:

CODE TO TYPE:

```
...
section#photos div.photo-with-caption img {
    width: 300px;
    width: 100%;
}
```

main.css and your **index.html** file. Now change the width of your browser to very wide and very narrow. The images resize so that they stay evenly spaced, within their respective flex box items, without any overlap.

In responsive design, by structuring your image content and laying out your pages in the ways we've described here, you can take advantage of the 100% width on your images. In this case, that means that the image is taking up 100% of its container (minus the padding on the <figure>); and since the <figure> elements are nested within elements that are acting as flex box items, that means that our images will dynamically resize as we widen or narrow the browser, because the flex box items are getting wider and narrower. Note that the image will shrink down a lot if you narrow the browser, but will expand only up to the full width of the image itself (in pixels). This will keep the images from getting too large for the screen and looking pixelated.

Using flex box for the images and setting the width to 100% is a great way to create a responsive design: we've used source images that are good quality and size (we'll talk more about image quality and load-time trade offs later), so as the page widens, and the images grow, they still look good. You can widen the browser as wide as your big desktop screen and they get large (up to their full width), allowing the user to see more details in the image. You can make the browser very narrow, and still see all of the (small) images.

For a more complete guide to flex box, check out the excellent resource, CSS Tricks: A Complete Guide to Flex Box. There are lots of options which give you a ton of flexibility in how you lay out your content. This website also explains very well—visually—all the options for the container and the items, so it's easy to see which options you need.

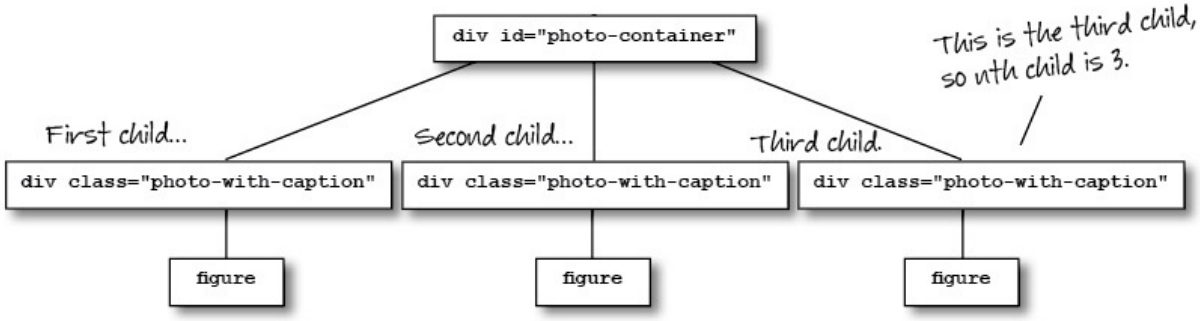
We've just touched on the ways you can use flex box, and you'll probably find this is a CSS feature you'll use often in responsive designs, especially as users upgrade to the latest browsers with support for this feature.

Adding a Media Query for the Middle Width Photos Section

There's one more thing we want to do before we wrap up this lesson on page layout, and that is to add a media query to switch from three images to two images in the photos section if the width of the page is less than 1000 pixels.

In practice, you might decide you don't need to do this. After all, our images are resizing beautifully no matter how wide the browser window is. However, it's a good exercise to see how to dynamically remove elements from the display in a situation like this.

Now you might think we need to add an id to the "photo-with-caption" element we want to hide if the browser width goes below 1000 pixels in order to select that element in the rule in the media query. We actually don't have to though; there is a CSS selector we can use to select a specific **child element**. Remember that the DOM is structured as a tree—here's a snippet of the DOM tree for our page:



Look at the HTML again to see how the DOM structure matches the HTML structure:

OBSERVE:

```
<section id="photos">
    <header>
        <h1>A sampling of photos from our tours</h1>
    </header>
    <div class="photo-container">
        <div class="photo-with-caption">
            <figure>
                ...
            </figure>
        </div>
        <div class="photo-with-caption">
            <figure>
                ...
            </figure>
        </div>
        <div class="photo-with-caption">
            <figure>
                ...
            </figure>
        </div>
    </div>
</section>
```

The **"photo-container" ** contains three child elements: the three "photo-with-caption" elements. Let's say we want to hide the last photo if the width of the browser goes below 1000 pixels. We can select that and hide it. Update the CSS in your main.css file:

CODE TO TYPE:

```
...
@media only screen and (max-width: 1000px) {
    section#photos div.photo-with-caption:nth-child(3) {
        display: none;
    }
}
```

main.css and your **index.html** file. Now, when you decrease the width of the browser below 1000 pixels, the third image disappears! The other two images will get a bit bigger; this allows us to show the images at a slightly larger size for browser widths less than 1000 pixels. In a later lesson, we'll look at how to switch the display completely when we go below 641 pixels for "mobile" mode.

Notice the selection we're using to select the third "photo-with-caption" element:

OBSERVE:

```
section#photos div.photo-with-caption:nth-child(3)
```

The first part of this says: find the elements with the class "photo-with-caption" that are nested in the <section> with the id "photos". This should all be familiar. And, as you know, this will select all three elements.

The **next part** is called a **pseudo-class**. A pseudo-class represents the *state* of an element. You're probably very familiar with the **:hover** pseudo-class, for instance, which lets you specify CSS rules for when you're hovering over an element like a link. In this case, the state we're selecting for is the number of the child in a set of siblings. **nth-child** is often used to select every other child, to add striping to every other element in a list, to make the items easy to read. However, in this case, we just want to select the child that is the third of the siblings (all the "photo-with-caption" elements). You could also decide to remove the first or second image instead, by using 1 or 2 in the parentheses, respectively.

Let's quickly add the finishing touches to the photos section, and we'll be done! Add the following CSS to your main.css file:

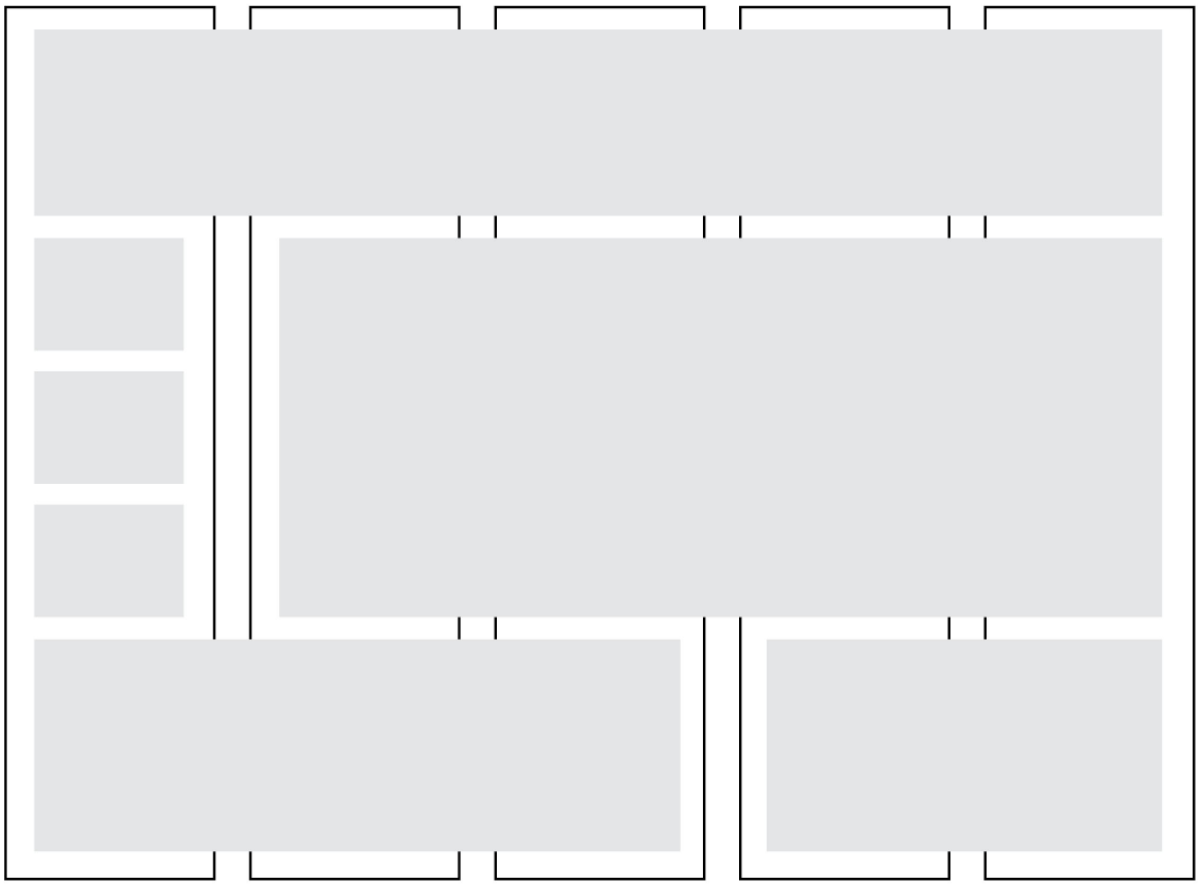
CODE TO TYPE:

```
...
section#photos {
    background-color: rgb(241, 238, 234);
}

section#photos header {
    padding: .8em 3% .8em 2%;
}
section#photos div.photo-container {
    display: -ms-flex;
    display: -webkit-flex;
    display: flex;
    -webkit-justify-content: center;
    -ms-justify-content: center;
    justify-content: center;
    align-items: flex-start;
    margin: auto;
}
section#photos div.photo-with-caption figure {
    padding: 1.5em;
    text-align: center;
}
section#photos div.photo-with-caption img {
    width: 100%;
}
@media only screen and (max-width: 1000px) {
    section#photos div.photo-with-caption:nth-child(3) {
        display: none;
    }
}
```

main.css and your **index.html** file. Check out your handiwork—it's looking pretty good! Now the photos section is complete.

Fluid Grids

Some designers take responsive layout a lot further with *fluid grids*. With a grid design, you divide up your page into columns, and every element you add to your page is aligned with these columns. Some elements take up multiple columns, but every element is aligned on the columns in some way:



If you're strict about aligning elements with columns, then your page will look beautifully balanced and aligned when you're done.

Initially, designers created grids for the standard 960px desktop sized browser. However, as we discussed in an earlier lesson, we now see a huge variety in the sizes of screens we use, so grids need to be more flexible and responsive. Developers have created grid systems you can download and use as a starting point to build responsive grids: these usually include HTML templates, as well as CSS to lay everything out, with media queries to allow for responsive grids. These grid systems are helpful for getting started with grids if you want to build your page that way. For instance, check out grid systems like Bootstrap, Profound Grid, Golden Grid, and Skeleton as just a few of the many options you have (as of this writing).

However, as you've seen, the CSS flex box features now available in browsers can let you build your own grid system. You could design your entire page in a flex box container, being careful to align all content within the page so that everything lines up according to a grid. We aren't going to do that in this course, so we'll leave it as something for you to explore on your own.

Next, we'll take a deeper look at responsive images. In this lesson, you learned some techniques for laying out content, including images, but there's a lot more to images than this.

In this lesson, we talked about three different ways to lay out content in a page so that the content adjusts well to different screen widths:

- CSS positioning (using relative and absolute positioning)
- CSS table display
- CSS flex box

In a later lesson, we'll look at additional ways we might want to adjust the content for mobile, or very narrow, screen widths.

Responsive Images

Lesson Objectives

When you complete this lesson, you will be able to:

- use the Network tab in your browser to inspect image sizes and asset load times.
- use Media Queries to load different background images depending on screen width.
- use JavaScript to load different images into the <img> tags in your page.
- use the **data-src** attribute to specify custom data in your HTML elements.
- use the **srcset** attribute of the <img> element to specify a set of images the browser can choose from when rendering the web page.
- use the <picture> element to specify multiple images, and media queries to select the correct image to display based on the width of the screen.
- implement high density pixel displays.

Problems with Images in Responsive Design

In our design and implementation of the Dandelion Tours web page, we've used several images, including:

- the background image for the experience <section>, TourGuide.jpg.
- the background image for the <header> in the details <section>, dlogo_small.png.
- three content images for the photos <section>.
    - Brazil_Butterfly_m.jpg
    - Iceland_Road_m.jpg
    - Alaska_View_m.jpg

Most websites include lots of images; in fact, downloading images for web pages is typically the most time consuming part of loading a web page in the browser. Fortunately, browsers are implemented so that images are downloaded separately from the HTML and CSS of the page so your page can begin loading and even displaying without waiting for all the images to load.

Many of us have high-speed internet connections and use modern browsers, so even image-heavy websites load fairly quickly. However, more and more users are accessing the web via mobile, and despite all their incredible powers, mobile devices are limited by the speed of their network connections. Even the fastest mobile network is still far slower than ethernet network connections. For instance, in 2014, average download speed on 3G networks is 2.5 Mbps on AT&T in the United States. Average download speed on 4G/LTE AT&T networks is 6.2Mbps. Compare these numbers to the average download speed of 32.1Mbps for ethernet connections in the United States, and 26.5Mbps in Europe. In South Korea, the average download speed is a mind-blowing 88.68Mbps; and while their mobile speeds are also much higher than the United States, their mobile speeds are still only a quarter of their ethernet speeds at 14.31Mbps.

Note To explore download speeds across the world, check out http://explorer.netindex.com/maps; my source for the data cited above.

With higher internet speeds, as well as computers with retina screens (screens with higher pixel densities), we're tempted to put bigger images into our pages because they look great on big, modern screens. However, if users with slower internet connections or mobile connections try to download those images, your website is going to be a frustrating user experience.

So, image size is a major consideration when developing a responsive website that you intend to be viewed on both mobile devices and desktop computers. We want to make sure that the images we provide for download by a web page to your browser are appropriate for the screen size and device you're using. Even though mobile screens are often high-density screens (with more pixels so the content looks sharp), we don't necessarily want to use large images for mobile because they will take too long to download. So we need to find ways to make sure that the right images are being downloaded depending on the user's screen and device.

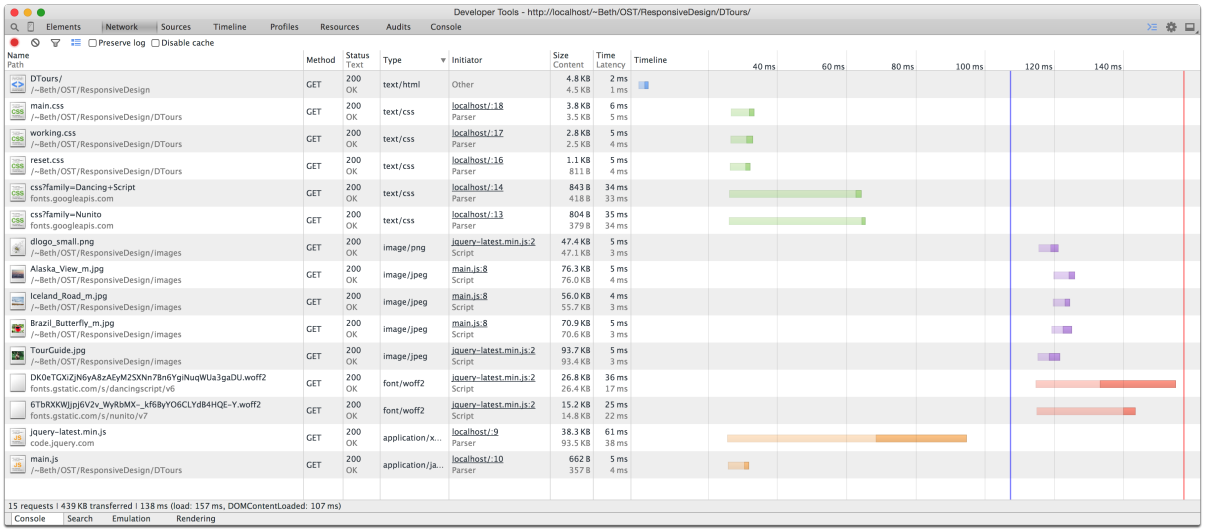
First, we'll take a look at how to use the browser developer tools to inspect the resources that our web page is downloading. We'll then look at a couple of different strategies for making sure we're downloading the right images to

the user. Finally, we'll talk about what high density screens (retina screens) actually are, and what they mean for your web page design.

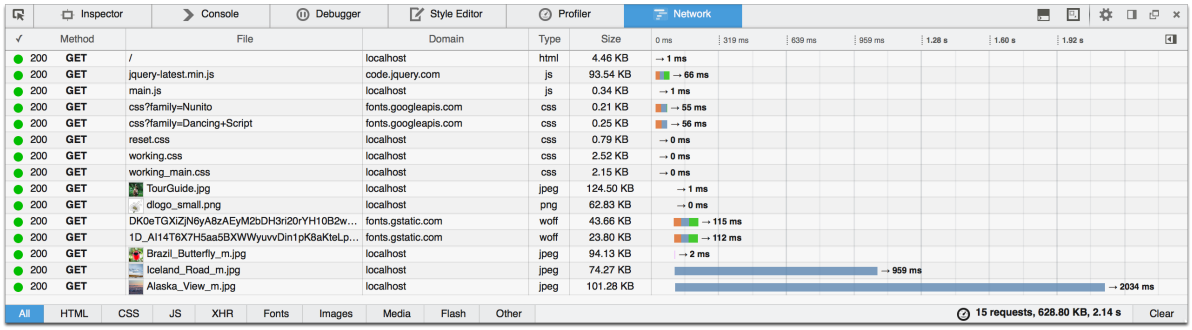
Inspecting Your Web Page Resources and Network Use

All the major browsers come with a developer tool called **Network** to let you inspect the resources your page is downloading (including the HTML, CSS, JavaScript, images, and fonts), and how much time the download is taking. I've included screenshots from the Chrome and Firefox Network tools below. If you access the Network tool in your own browser, it might look a bit different from these screenshots, but the basic idea should be the same.

Here's Chrome:



Here's Firefox:



Try to access the **Network** tool yourself in the browser. Open a new browser window, and load the Dandelion Tours page you've been working on. Open up the developer tools, and select the **Network** tab. Then **Shift+Reload** the page (hold down **Shift** while you click the **Reload** button). This will sometimes force the browser to download the resources for the page again rather than using versions cached on your local machine.

> **Note** The URLs for the resources in your page will likely be different from mine, but the names of the resources should mostly be the same.

In either Chrome or Firefox, you can tell which resources your page is downloading when you load the page: the HTML (shown as just **/** in Firefox and **DTours/** in Chrome), the CSS, the Google-hosted fonts, the images, and in my case, some JavaScript that I've added to the page (you'll be adding your own later).

Take note of which resources take the longest to download, and which are the biggest. You can look under the **Size** column to see the size of each resource, and time columns give you an indication of how long the page takes to load. Whether the browser has cached a resource greatly impacts the load time.

You can see from the screenshots above that the largest resources being downloaded by my Dandelion Tours page are the images and the jQuery library. We have four fairly large images on the page, so if anything

is going to slow down the load time of the page, it will be those images. These images are local to my computer so they don't require a network download, but if you were hosting the web page online, they would take quite a bit longer for the user to download the first time (before they are cached). On my desktop computer (where I took the screenshots), the load time is not that noticeable, but on my iPhone (a 4s iPhone with 4G network, but usually with only 2 or 3 bars of connectivity here at home), the page load definitely takes longer.

If I'm on a big screen desktop computer and reduce the width of the browser below 640 pixels, it doesn't make any a difference on the load times of the resources, because I'm still using a computer with a fast connection. I may not get the most out of the big images I'm using (because of the narrow width of the browser), but the page isn't going to take any longer to load. However, if you that narrow browser window is on a mobile device connected via mobile network (not wifi), those big images will make a real difference in load time. So let's take a look at a couple of different ways we can change which images are loaded if we're accessing the web page from a mobile device.

Strategies for Smart Image Loading

To reduce the wait time for users when downloading a web page, particularly on mobile devices, we want to make sure they aren't downloading images they don't need, or images that are the wrong size for the device.

One strategy we can use is with media queries. You've already used media queries to change the layout of a page depending on the width of the user's screen; we can also use media queries to determine which images to download into the background of an element. However, this is a useful strategy for background images only; if you specify an image in your HTML using the <img> tag, that image will be downloaded regardless of what media queries you have in your CSS. We'll discuss a strategy for handling images loaded with the <img> tag next. For now, we'll focus on the background images we're loading in the CSS.

So far, we've styled the experience <section> of the site like this (in **main.css**):

OBSERVE:

```
section#experience {
    position: relative;
    color: white;
    padding: 0em 60% 2em 2%;
    background-size: cover;
    background-image: url("https://courses.oreillyschool.com/html5_responsive_de
sign/images/TourGuide.jpg");
    min-height: 400px;
}
```

The TourGuide.jpg image file is about 100Kb, and 960 pixels wide. That's not too bad, but far more than we need for a mobile page that is 640 pixels wide. So, we created a version of the image that is 640 pixels wide and only about 50Kb that we can use if the screen is less than or equal to 640 pixels wide. Update your file, main.css:

CODE TO TYPE:

```
section#experience {
    position: relative;
    color: white;
    padding: 0em 60% 2em 2%;
    background-size: cover;
    background-image: url("https://courses.oreillyschool.com/html5_responsive_design/images/TourGuide.jpg");
    min-height: 400px;
}

@media only screen and (min-width: 641px) {
    section#experience {
        background-image: url("https://courses.oreillyschool.com/html5_responsive_design/images/TourGuide.jpg");
        min-height: 400px;
    }
}
@media only screen and (max-width: 640px) {
    section#experience {
        background-image: url("https://courses.oreillyschool.com/html5_responsive_design/images/TourGuide_sm.jpg");
        background-position: center top;
        padding: 0em 30% .8em 2%;
    }
}
```

main.css and your **index.html** file. Narrow your browser to 640 pixels or less, and you should see the smaller image appear in the browser window. It's cropped a bit differently so you should be able to tell when the image loads.

Now, when you do this in your browser (that is, start with a wider browser window, and then narrow the window so that the browser loads the smaller image), you are actually requesting and loading *both* images, so you haven't saved any network downtime at all. In fact, you are increasing the number of requests you're making because you end up loading both images. However, on mobile devices, you'll only request *one* of these images: if the screen width is 640 pixels or less, you'll request and download the smaller image; otherwise, you'll request and download the larger image (same as if you don't resize your browser window). This makes your web page "responsive" in a new way: for smaller, narrower screens, your web page will take less time to load, because you are downloading almost 50Kb less data!

Let's take a closer look at a couple of other things we did in this new CSS code:

OBSERVE:

```
@media only screen and (min-width: 641px) {
    section#experience {
        background-image: url("https://courses.oreillyschool.com/html5_responsive_design/images/TourGuide.jpg");
        min-height: 400px;
    }
}
@media only screen and (max-width: 640px) {
    section#experience {
        background-image: url("https://courses.oreillyschool.com/html5_responsive_design/images/TourGuide_sm.jpg");
        background-position: center top;
        padding: 0em 30% .8em 2%;
    }
}
```

First, notice that we moved the **min-height** property into the wide media query (that is, the media query that specifies properties for when the page is greater than 640 pixels wide). For the narrow version of the page, we don't need to set a min-height for the background image, because the text will never get wide enough that we lose too much height in the element to see the image properly.

Also notice that for the narrow media query, we specify a **position for the image at the center and top of the element**. That means that if any part of the image needs to be cropped to fit into the element, it will be cropped from the sides and bottom. That keeps the image of the woman more centered in the element, and generally looking better. We've also changed the **padding** a bit for the mobile version, to give the text a bit more room on the right hand side so it stays balanced on the page.

If you make your browser narrow (less than 640 pixels wide) and **Shift+Reload** the page again, and look at the **Network** tab in the developer tools, you see that the smaller version of the Tour Guide photo has been loaded instead of the larger version.

The other background image we use in the page is the image for the <header> of the details <section>. This image is already fairly small, around 50Kb, so we're not going to bother creating a separate media query for it (although that will make a good exercise for you).

You might be tempted at this point to try to make *all* your images background images: don't. As we discussed in an earlier lesson, there is a significant semantic difference between background images and content images. For instance, the two images we're using in our web page for the experience section and the header of the details section truly are background images. If they weren't there, there would be very little difference to the end user in terms of understanding the page and the message we're trying to convey. However, the photos, which are specified with the <img> tag, truly are content images. There, we want the user to get a sense of the kinds of things they'll be seeing on the tour, so if those images disappeared, the user experience would be downgraded. Also, keep in mind that for accessibility, background images are often ignored by screen readers, whereas screen readers will usually read the content of the **alt** property of an <img> tag to users so they get a sense of what the image is about. So choosing the right kind of image is also important for accessibility.

Using JavaScript to Load Image Content

When the browser sees an <img> tag, it sends a request to the server asking for the image. So, if you specify an image in the <img> tag, the browser will always request it—even if that image is hidden with CSS or replaced later with JavaScript or a CSS media query.

If we want to make sure the browser loads the correct image based on the width of the screen (or even the screen's pixel density, which we'll look at later), we have to request and load the images using JavaScript instead. We're going to do that for the three photos in the details <section> using jQuery. First, we'll modify the **index.html** file to link to the jQuery library, as well as a new JavaScript file that you're going to write, **main.js**. We'll modify the <img> elements in the photos section so we're not directly linking to the images with the **src** attribute. Finally, we'll add our own code to load the images for the photos section. Edit index.html and add the following two script links in the <head>:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Dandelion Tours</title>

<script src="http://code.jquery.com/jquery-latest.min.js"></script>
<script src="main.js"></script>

<link href='http://fonts.googleapis.com/css?family=Nunito' rel='stylesheet' type
='text/css'>
<link href='http://fonts.googleapis.com/css?family=Dancing+Script' rel='styleshe
et' type='text/css'>

<link rel="stylesheet" href="reset.css">
<link rel="stylesheet" href="dt.css">
<link rel="stylesheet" href="main.css">

</head>
...
```

We haven't created **main.js**, so don't reload the page just yet. Instead, continue editing **index.html** to modify the <img> tags in the photos <section> where we are linking to the three images. Change the **src** attribute of each image into a **data-src** attribute and modify the image URLs. Edit index.html and modify the images in the photos <section>:

CODE TO TYPE:

```
...
<section id="photos">
    <header>
        <h1>A sampling of photos from our tours</h1>
    </header>
    <div class="photo-container">
        <div class="photo-with-caption">
            <figure>
                <img data-src="https://courses.oreillyschool.com/html5_responsiv
e_design/images/Brazil_Butterfly_m.jpg" alt="photo of mountains and rainbow">
                <figcaption>Exciting wildlife</figcaption>
            </figure>
        </div>
        <div class="photo-with-caption">
            <figure>
                <img data-src="https://courses.oreillyschool.com/html5_responsiv
e_design/images/Iceland_Road_m.jpg" alt="photo of road in snow">
                <figcaption>Spectacular landscapes</figcaption>
            </figure>
        </div>
        <div class="photo-with-caption">
            <figure>
                <img data-src="https://courses.oreillyschool.com/html5_responsiv
e_design/images/Alaska_View_m.jpg" alt="photo of rainbow and waterfall">
                <figcaption>Incredible views</figcaption>
            </figure>
        </div>
    </div>
</section>
...
```

and your **index.html** file. The images disappear for now; we'll fix that in just a moment by writing code in the **main.js** file to load the images. But first, let's take a closer look at what we did to modify the <img> tags.

What exactly is **data-src**, or more generally, **data-***? **data-*** is a way for you to specify a custom attribute in your HTML elements. HTML5 added support for these arbitrary attributes in elements. As long as the attribute you add begins with "data-" you can create any new attributes you want and add them to the opening tags of any HTML element. You don't want to use custom attributes to do the work that an existing attribute would do; however, if you need to add custom data to your HTML elements that doesn't fit into an existing attribute, this is a really handy feature of HTML. Adding a custom attribute doesn't change how the element is rendered or styled in any way, so these attributes are useful primarily when you are using them with JavaScript.

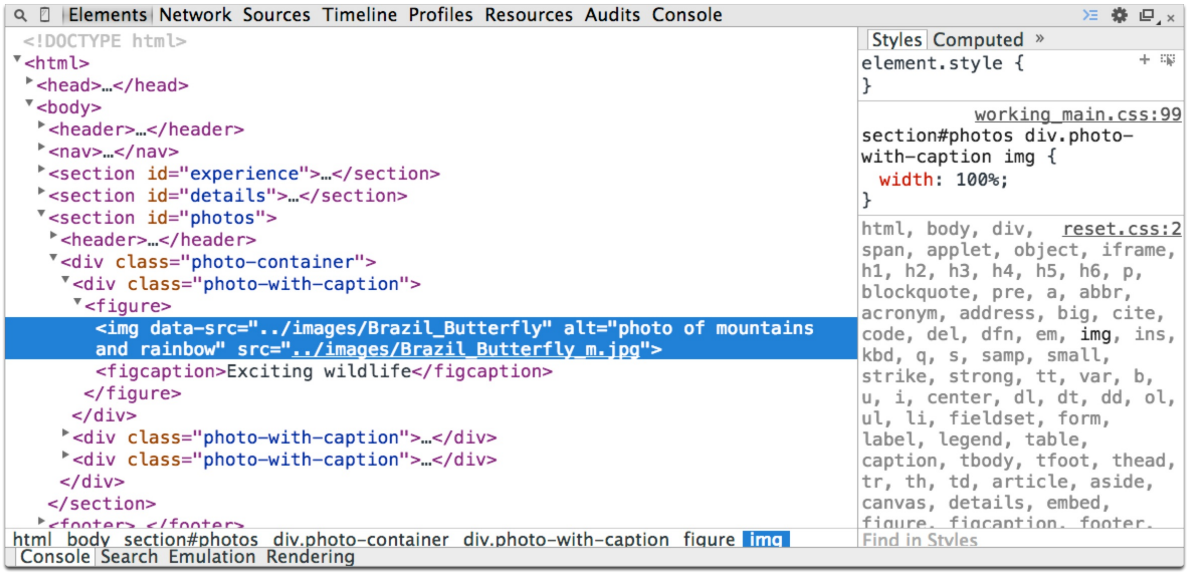
In our case, we want to store the image URL to use in the page in the <img> elements, but we don't want to use the **src** attribute, because using the **src** attribute will cause the image to be loaded by the browser. By moving the image URLs to the **data-src** attribute, we can store this data in the <img> element without the browser loading the image (it will only load images specified in the **src** attribute), and then use JavaScript to load the images using the URLs in the **data-src** attributes. Notice that we *removed* the extension "_m.jpg" from each of the images in the code above. We did this because we want to get the name of the image, and then decide, based on the width of the screen, whether to load the medium sized images (with the "_m.jpg" extension) or the small images (with the "_s.jpg" extension). We've created both medium and small versions of each of the images we're using in the page, so we can load either one.

Let's take a look at the code, so you can see what all this means. Create or open up your **main.js** file, and add the following JavaScript (jQuery):

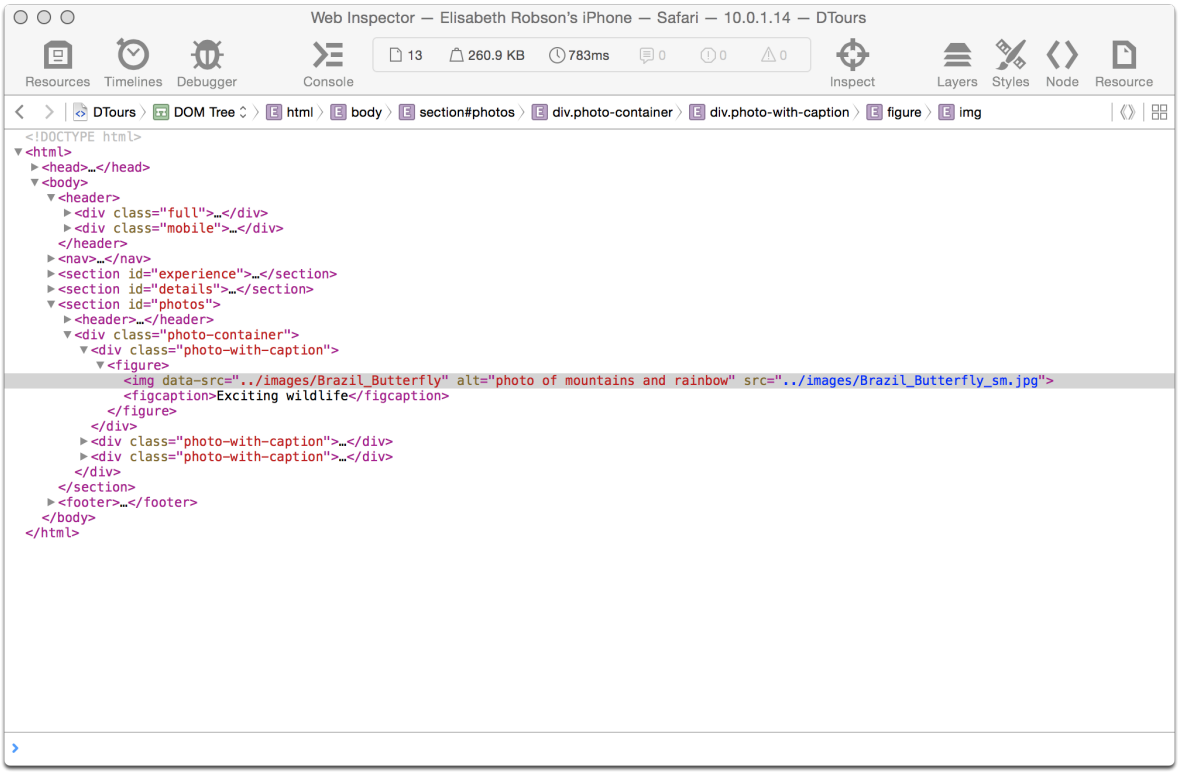
CODE TO TYPE:

```
$(document).ready(function () {

    // returns a list of images in the photos section
    var $imgs = $("#photos .photo-with-caption figure img");

    if ($(window).width() > 640) {
        $imgs.each(function(i, theImage) {
            theImage.src = $(theImage).data("src") + "_m.jpg";
        });
    } else {
        $imgs.each(function(i, theImage) {
            theImage.src = $(theImage).data("src") + "_sm.jpg";
        });
    }
});
```

main.js and your **index.html** file. Now the images appear in the page. If you load the page when the browser is wide, the medium-sized images load (check out the network tab to see which images are loading). If you narrow your browser, and **Shift+Reload**, the small images load. You can also use the Elements tab to inspect your HTML. After loading the page, you see both the **data-src** and **src** attributes in the images, and you can check to see the value of the **src** attribute directly. For instance, after reloading the page with the window opened wide, I see:



Here, the <img> tag I'm inspecting has the **src** attribute set to a medium size image. After loading the page on my iPhone, I see:



And here, the <img> **src** attribute is set to a small image because the width of the iPhone screen is less than or equal to 640 pixels.

Let's step through the jQuery code in detail. First, we have defined this code to execute once the page is fully loaded, by specifying a **ready** function. This is a jQuery method that is equivalent to specifying a load handler on the window in plain JavaScript. It says, once the page has completely loaded and the DOM is ready, then execute the function we're passing into the **ready** method.

The first thing we do is get all the images that are in the photos section using a selector:

OBSERVE:

```
var $imgs = $("#photos .photo-with-caption figure img");
```

This says: get all the images that are nested within a <figure> element which is nested in an element with the class ".photo-with-caption" that is, in turn, nested within an element with the id "#photos." The resulting **$imgs** object is a jQuery list of all the images.

We then test to see the **width of the browser window**, using the jQuery method, **width()**.

OBSERVE:

```
if ($(window).width() > 640) {
    $imgs.each(function(i, theImage) {
        theImage.src = $(theImage).data("src") + "_m.jpg";
    });
} else {
    $imgs.each(function(i, theImage) {
        theImage.src = $(theImage).data("src") + "_sm.jpg";
    });
}
```

If the width is greater than 640 pixels, we **loop through all the images**, get the value of the image's **data-src** attribute using the jQuery **data()** method, **append the string "_m.jpg"** to the URL, and update the **src** attribute to contain the URL for the medium size images. Similarly **if the width is equal to or less than 640 pixels**, we **loop through all the images**, get the value of the image's **data-src** attribute using the jQuery **data** method, **append the string "_sm.jpg"** to the URL, and update the **src** attribute of the images. As soon as the **src** attribute of an image is added or changed, the browser will load that image. This code

runs as soon as the page is loaded, and you see these images load.

When you use the jQuery **data** method to get the value of the **data-src** attribute, notice that you don't have to specify the "data-" part of the attribute name, only the part that is appended to it, in this case "src." The **data** method is specifically for getting **data-*** attributes from your HTML.

Note The **data-src** attribute we added to the <img> tags (replacing the **src** attribute) is a feature of HTML5, and has solid support in all browsers, going back a few versions—except for IE. IE11 fully supports the new **data-*** attributes, as they are called, but previous versions of IE did not. However, because we're using the jQuery **data** method to access the **data-src** attribute, the code should work in IE going back to IE8 (jQuery handles the browser support issues for us).

Solutions Coming in the Future with New HTML Support

The Srcset Attribute of the <img> Element

We've solved the problem of deciding which images to load into the <img> elements in our page using JavaScript. However, not everyone is happy with this solution because it requires writing code. It would be better if we could handle this "image problem" using HTML, without the JavaScript coding.

Now that mobile devices are so popular, the "image problem" is high on the list of problems to solve at the W3C and with responsive designers. There are two proposals that are in development at the W3C to create HTML elements that will allow the browsers to make a smart decision about which images to download and display, depending on the characteristics of the user's display and network connection. Unfortunately, these proposals are still in the development stage, so you won't be able to actually use them right now for public web pages.

Note There are a couple of browsers that do already support the two solutions we're going to discuss: Chrome (version 39) and Opera (version 25). So if you have either of these browsers installed, you can run this code! However, because the majority of users are probably not be using these browsers, it's not a good idea to use either of these solutions yet for pages you plan to post on the internet. Until there is widespread support for the solutions in all browsers, we are stuck playing with them on our local computers only.

The first solution we'll discuss is a new attribute proposed for the <img> element: the **srcset** attribute. To use this, we'll first remove the jQuery code we just wrote, and then update the <img> elements in the photos section once again. Edit index.html and modify your HTML:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Dandelion Tours</title>

<script src="http://code.jquery.com/jquery-latest.min.js"></script>
<script src="main.js"></script>

<link href='http://fonts.googleapis.com/css?family=Nunito' rel='stylesheet' type
='text/css'>
<link href='http://fonts.googleapis.com/css?family=Dancing+Script' rel='styleshe
et' type='text/css'>

<link rel="stylesheet" href="reset.css">
<link rel="stylesheet" href="dt.css">
<link rel="stylesheet" href="main.css">

</head>

...

<section id="photos">
    <header>
        <h1>A sampling of photos from our tours</h1>
    </header>
    <div class="photo-container">
        <div class="photo-with-caption">
            <figure>
                <img data-src="https://courses.oreillyschool.com/html5_responsiv
e_design/images/Brazil_Butterfly" alt="photo of mountains and rainbow">
                <img srcset="https://courses.oreillyschool.com/html5_responsive_
design/images/Brazil_Butterfly_m.jpg 1400w,
                            https://courses.oreillyschool.com/html5_responsive_d
esign/images/Brazil_Butterfly_sm.jpg 640w">
                <figcaption>Exciting wildlife</figcaption>
            </figure>
        </div>
        <div class="photo-with-caption">
            <figure>
                <img data-src="https://courses.oreillyschool.com/html5_responsiv
e_design/images/Iceland_Road" alt="photo of road in snow">
                <img srcset="https://courses.oreillyschool.com/html5_responsive_
design/images/Iceland_Road_m.jpg 1400w,
                            https://courses.oreillyschool.com/html5_responsive_d
esign/images/Iceland_Road_sm.jpg 640w">
                <figcaption>Spectacular landscapes</figcaption>
            </figure>
        </div>
        <div class="photo-with-caption">
            <figure>
                <img data-src="https://courses.oreillyschool.com/html5_responsiv
e_design/images/Alaska_View" alt="photo of rainbow and waterfall">
                <img srcset="https://courses.oreillyschool.com/html5_responsive_
design/images/Alaska_View_m.jpg 1400w,
                            https://courses.oreillyschool.com/html5_responsive_d
esign/images/Alaska_View_sm.jpg 640w">
                <figcaption>Incredible views</figcaption>
            </figure>
        </div>
    </div>
</section>
...
```

and your **index.html** file using either Chrome (version 38 or greater) or Opera (version 25 or greater). The images load just fine in either one of these browsers. If you try another browser, you will not see the images! Notice that this solution requires no JavaScript at all, so when (or if) this solution is implemented in all browsers, it will be easier than our previous jQuery-based solution. Right now, this is a proposed solution, so it remains to be seen if it's adopted across the board by browsers.

The nice thing about this solution is that it makes use of the existing <img> element, so no new element needs to be added to HTML. However, some web designers have complained that the syntax is a little hard to understand. The value of the **srcset** attribute is a comma-delimited list of image URLs, each followed by a number with the "suggested" width for using the image. It's important to note that the **srcset** is a "suggestion" for the browser, but under some circumstances the browser may choose to load whichever image it wants. Also, note that the widths that follow the images are suggestions; in our case, the smaller image will probably be loaded up to 640 pixels wide, and the medium image will probably be loaded for any widths above 640 pixels. However, we *must* specify a width to follow the medium size image. We used 1400, however, if you open your browser wider than that and reload the page, the medium images are still used. Again, **srcset** is considered a suggestion to the browser.

The <picture> Element

A second proposed solution to the "image problem" is a whole new element for HTML: the <picture> element. The <picture> element is more flexible than the **srcset** attribute on the <img> element, and rather than being a suggestion like the <img> **srcset**, it clearly defines what browsers should do in a variety of situations. Let's change the code to use the <picture> element, and then we'll step through how it works. Modify index.html as shown:

CODE TO TYPE:

```
...
<section id="photos">
    <header>
        <h1>A sampling of photos from our tours</h1>
    </header>
    <div class="photo-container">
        <div class="photo-with-caption">
            <figure>
                <img srcset="https://courses.oreillyschool.com/html5_responsive_
design/images/Brazil_Butterfly_m.jpg 1400w,
                             https://courses.oreillyschool.com/html5_responsive_d
esign/images/Brazil_Butterfly_sm.jpg 640w">
                <picture>
                    <source media="(min-width: 641px)"
                            srcset="https://courses.oreillyschool.com/html5_resp
onsive_design/images/Brazil_Butterfly_m.jpg">
                    <source media="(max-width: 640px)"
                            srcset="https://courses.oreillyschool.com/html5_resp
onsive_design/images/Brazil_Butterfly_sm.jpg">
                    <img src="https://courses.oreillyschool.com/html5_responsive
_design/images/Brazil_Butterfly_sm.jpg"
                         alt="photo of mountains and rainbow">
                </picture>
                <figcaption>Exciting wildlife</figcaption>
            </figure>
        </div>
        <div class="photo-with-caption">
            <figure>
                <img srcset="https://courses.oreillyschool.com/html5_responsive_
design/images/Iceland_Road_m.jpg 1400w,
                             https://courses.oreillyschool.com/html5_responsive_d
esign/images/Iceland_Road_sm.jpg 640w">
                <picture>
                    <source media="(min-width: 641px)"
                            srcset="https://courses.oreillyschool.com/html5_resp
onsive_design/images/Iceland_Road_m.jpg">
                    <source media="(max-width: 640px)"
                            srcset="https://courses.oreillyschool.com/html5_resp
onsive_design/images/Iceland_Road_sm.jpg">
                    <img src="https://courses.oreillyschool.com/html5_responsive
_design/images/Iceland_Road_sm.jpg"
                         alt="photo of mountains and rainbow">
                </picture>
                <figcaption>Spectacular landscapes</figcaption>
            </figure>
        </div>
        <div class="photo-with-caption">
            <figure>
                <img srcset="https://courses.oreillyschool.com/html5_responsive_
design/images/Alaska_View_m.jpg 1400w,
                             https://courses.oreillyschool.com/html5_responsive_d
esign/images/Alaska_View_sm.jpg 640w">
                <picture>
                    <source media="(min-width: 641px)"
                            srcset="https://courses.oreillyschool.com/html5_resp
onsive_design/images/Alaska_View_m.jpg">
                    <source media="(max-width: 640px)"
                            srcset="https://courses.oreillyschool.com/html5_resp
onsive_design/images/Alaska_View_sm.jpg">
                    <img src="https://courses.oreillyschool.com/html5_responsive
_design/images/Alaska_View_sm.jpg"
                         alt="photo of mountains and rainbow">
                </picture>
                <figcaption>Incredible views</figcaption>
            </figure>
        </div>
```

```
    </div>
</section>
...
```

Also make a small change to your CSS in **main.css**:

CODE TO TYPE:

```
section#photos div.photo-with-caption img {
    width: 100%;
}
section#photos div.photo-with-caption picture {
    width: 100%;
}
```

and your **index.html** file using either Chrome (version 38 or greater) or Opera (version 25 or greater). Again, you see either the medium images if your browser is open wide, or the small images if your browser is narrow or you are on your mobile device with a screen width of less than 640 pixels.

The <picture> element contains one or more <source> elements that you use to specify the images, only one of which will be used. Each <source> element has a **media attribute** that specifies a media query—just like the media queries we're using in our CSS—that determines under which conditions the image will be loaded and used in the page. Each <source> element also has a **srcset attribute** that you use to specify the image to use when the media query's feature is true. So, in this example:

OBSERVE:

```
<picture>
    <source media="(min-width: 641px)"
            srcset="https://courses.oreillyschool.com/html5_responsive_design
/images/Brazil_Butterfly_m.jpg">
    <source media="(max-width: 640px)"
            srcset="https://courses.oreillyschool.com/html5_responsive_design
/images/Brazil_Butterfly_sm.jpg">
    <img src="https://courses.oreillyschool.com/html5_responsive_design/images/B
razil_Butterfly_sm.jpg"
         alt="photo of mountains and rainbow">
</picture>
```

...if the width of the browser is 641 pixels or greater, the medium image will be used, and if the width of the browser is 640 pixels or less, the small image will be used.

We also provide an <img> element inside the <picture> element that can be used as a fallback for older browsers (although I've noticed this fallback doesn't seem to work in Firefox).

Both the **srcset** attribute of the <img> element and the **picture** element are much-anticipated additions to HTML to give us a lot more flexibility and control over how images are loaded by the browsers. Unfortunately, for now, these are still experimental and can't be used in public web pages on the internet because of lack of support in browsers.

Note Check the http://caniuse.com/ website when you take this course as more browsers may be supporting the srcset attribute and the picture element by the time you take the course.

Use the data-src Option with JavaScript for Now

Because we can't use either the **srcset** attribute on the <img> element, or the <picture> element quite yet, let's switch our code back to using the <img> element with the **data-src** attribute, plus a little JavaScript to load the correct images (as we did above). In this version, we'll take a quick look at how to rewrite the code using plain JavaScript instead of jQuery in case you want to try that (and also reduce the total download size by eliminating jQuery). Edit index.html and modify the images in the photos <section>:

CODE TO TYPE:

```
...
<section id="photos">
    <header>
        <h1>A sampling of photos from our tours</h1>
    </header>
    <div class="photo-container">
        <div class="photo-with-caption">
            <figure>
                <picture>
                    <source media="(min width: 641px)"
                            srcset="https://courses.oreillyschool.com/html5_responsive_design/images/Brazil_Butterfly_m.jpg">
                    <source media="(max width: 640px)"
                            srcset="https://courses.oreillyschool.com/html5_responsive_design/images/Brazil_Butterfly_sm.jpg">
                    <img src="https://courses.oreillyschool.com/html5_responsive_design/images/Brazil_Butterfly_sm.jpg"
                         alt="photo of mountains and rainbow">
                </picture>
                <img data-src="https://courses.oreillyschool.com/html5_responsive_design/images/Brazil_Butterfly" alt="photo of mountains and rainbow">
                <figcaption>Exciting wildlife</figcaption>
            </figure>
        </div>
        <div class="photo-with-caption">
            <figure>
                <picture>
                    <source media="(min width: 641px)"
                            srcset="https://courses.oreillyschool.com/html5_responsive_design/images/Iceland_Road_m.jpg">
                    <source media="(max width: 640px)"
                            srcset="https://courses.oreillyschool.com/html5_responsive_design/images/Iceland_Road_sm.jpg">
                    <img src="https://courses.oreillyschool.com/html5_responsive_design/images/Iceland_Road_sm.jpg"
                         alt="photo of mountains and rainbow">
                </picture>
                <img data-src="https://courses.oreillyschool.com/html5_responsive_design/images/Iceland_Road" alt="photo of road in snow">
                <figcaption>Spectacular landscapes</figcaption>
            </figure>
        </div>
        <div class="photo-with-caption">
            <figure>
                <picture>
                    <source media="(min width: 641px)"
                            srcset="https://courses.oreillyschool.com/html5_responsive_design/images/Alaska_View_m.jpg">
                    <source media="(max width: 640px)"
                            srcset="https://courses.oreillyschool.com/html5_responsive_design/images/Alaska_View_sm.jpg">
                    <img src="https://courses.oreillyschool.com/html5_responsive_design/images/Alaska_View_sm.jpg"
                         alt="photo of mountains and rainbow">
                </picture>
                <img data-src="https://courses.oreillyschool.com/html5_responsive_design/images/Alaska_View" alt="photo of rainbow and waterfall">
                <figcaption>Incredible views</figcaption>
            </figure>
        </div>
    </div>
</section>
...
```

Change your CSS back too:

Update the CSS in your main.css file:

```
section#photos div.photo-with-caption img {
    width: 100%;
}
section#photos div.photo-with-caption picture {
    width: 100%;
}
```

Now, update the JavaScript in **main.js** to use plain JavaScript. (We'll be using more jQuery later, so it's up to you if you want to change your code at this point; if you don't, just take a look at the code below so you know how you can implement this feature with either jQuery or plain JavaScript). Edit your main.js file and modify your JavaScript code:

CODE TO TYPE:

```
// We haven't deleted the jQuery code below; rather we've just commented it out
// in case you want to go back to using it later.
/*
$(document).ready(function () {

    // returns a list of images
    var $imgs = $("#photos .photo-with-caption figure img");

    if ($(window).width() > 640) {
        $imgs.each(function(i, theImage) {
            theImage.src = $(theImage).data("src") + "_m.jpg";
        });
    } else {
        $imgs.each(function(i, theImage) {
            theImage.src = $(theImage).data("src") + "_sm.jpg";
        });
    }
});
*/

window.onload = function() {
    var images = document.querySelectorAll("#photos .photo-with-caption figure i
mg");
    if (!images) {
        return;
    }

    for (var i = 0; i < images.length; i++) {
        var theImage = images[i];
        var url = theImage.getAttribute("data-src");
        if (window.innerWidth > 640) {
            url = url + "_m.jpg";
        } else {
            url = url + "_sm.jpg";
        }
        theImage.src = url;
    }
};
```

If you're not going to use jQuery, then you can delete or comment out the line in your **index.html** that links to jQuery. Modify your index.html file again as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Dandelion Tours</title>

<!--
<script src="http://code.jquery.com/jquery-latest.min.js"></script>
-->
<script src="main.js"></script>
```

and your **index.html**. Your images load as before, but now, you are once again using the **data-src** attribute and some JavaScript to load the images. Notice we just commented out the link to jQuery for now because we'll be using it again later. Removing the link to jQuery will reduce the total amount of data to download from the server by a bit (check the Network tab in the console to see the difference). However, notice that jQuery is a fairly common library on the internet. Chances are your browser may already have this file cached! Even if it doesn't, the browser will cache it after the first download, so unlike images which are different from page to page in your site, the jQuery library will only need to be downloaded once per session (this is true for both desktop and mobile browsers).

High Density Pixel (Retina) Displays

You may have heard about "retina displays," "high DPI screens," or "high density pixel displays" recently. All these terms basically mean the same thing: screens with higher pixel density than was standard for a long time on computers (in particular, CRT desktop monitors).

What is "pixel density?" It's the number of pixels that can fit into a distance. For instance, the iPhone 4s has a screen resolution of 640 pixels by 960 pixels. The phone screen is about 2 inches across, so those 640 pixels across fit into 2 inches. Now imagine an old SVGA CRT monitor from the 1990s. Those screens typically had a resolution of 800 pixels across, and the monitors were about 9-10 inches across. A pixel on the SVGA display is much bigger than a pixel on the iPhone display. We compute "pixels per inch" or "dots per inch" (DPI) by measuring the number of pixels that fit into a square inch of display. For the iPhone 4s, the pixels per inch is 326. For that old SVGA monitor, the pixels per inch is only about 72. We say that a high density display (or "high DPI" display) is a device with a pixel display density of 200 pixels per inch or more.

Note We write "pixels per inch" as "ppi" and "dots per inch" as "dpi." This comes from the old days of graphic design when we printed our designs, and we measured resolution in terms of the number of ink dots a printer could put onto a square inch of paper.

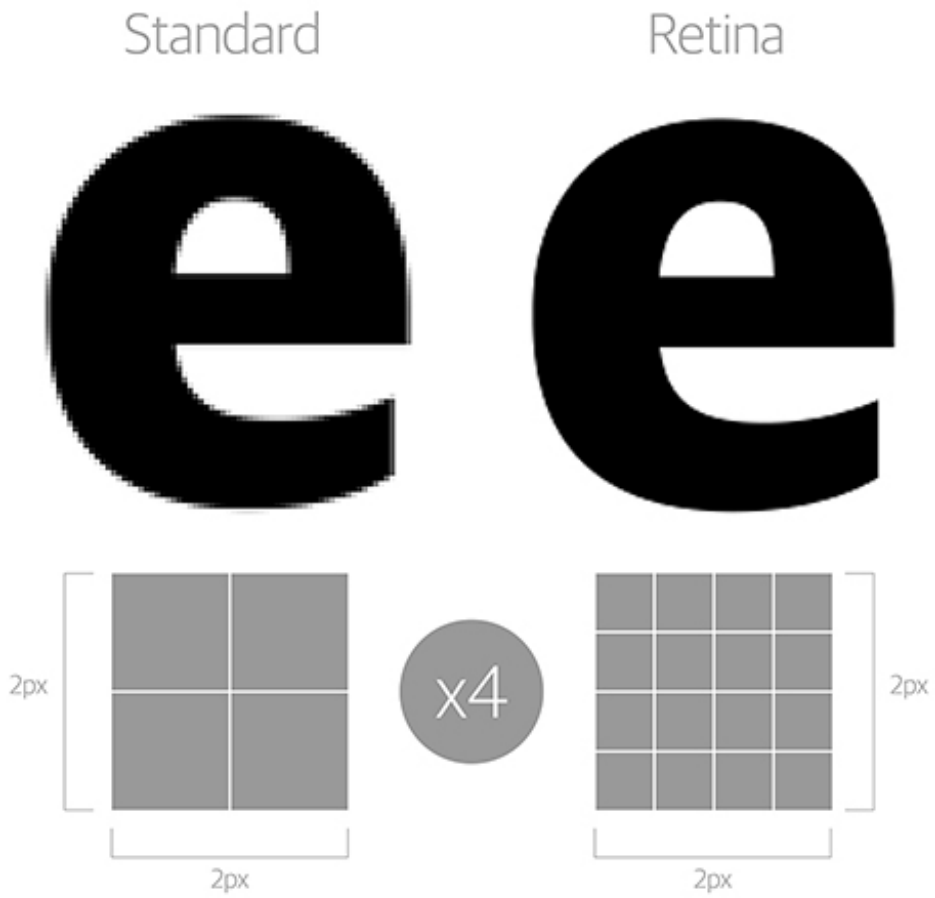
Note The term "retina display" is actually an Apple-specific term used initially for the iPhone, and now used on other Apple devices with high-density displays. Other companies tend not to use the term "retina display" to avoid infringing on Apple's trademarks.

So how do high-density displays affect your design? The first thing to remember is that users are going to be viewing your web page on vastly different displays with vastly different capabilities. An 800 x 600 image in your page will look much bigger on an older monitor, or a monitor that has been set to a lower screen resolution, than it will on a high-resolution screen. As you've seen, we can address some issues of varying screen sizes using media queries. However, on mobile, these issues are a little more complex because the high pixel densities of modern phones and tablets mean that you have to choose between a crisp but big image, and a less-crisp (potentially slightly blurry) but smaller and faster-to-download image. The varying sizes of mobile device displays also mean that you should rarely if ever design your page for a specific resolution or a specific pixel density. We picked a break point of 640 pixels for our design to switch from the wide view to the narrow view. Why 640 pixels? Because until recently that is a fairly common resolution width for some of the devices on the market. However, that is changing rapidly, especially as displays get better and better with more resolution, so using 640 pixels simply because it matches the width of a specific device isn't necessarily the best choice.

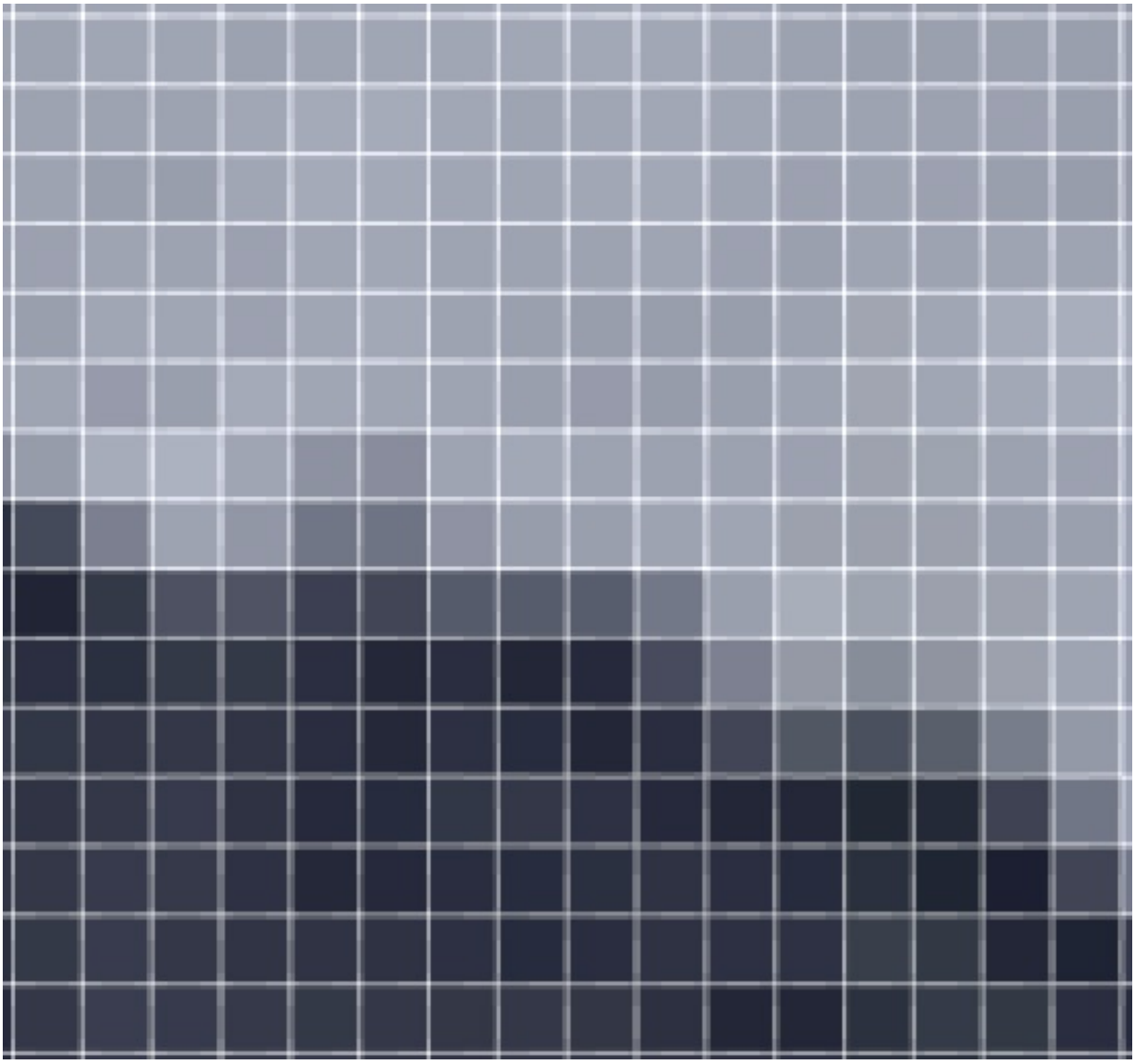
One thing to know about mobile devices is that some, like Apple's, use a trick called the "device pixel ratio" to make sure that the text of web pages doesn't become too small to see. Remember when we added the **viewport** <meta> tag to the **index.html** file?

OBSERVE:
`<meta name="viewport" content="width=device-width, initial-scale=1, user-scalable=no">`

This tag ensures that devices will scale their displays using the device pixel ratio assigned to that device. In the case of the iPhone 4s, the device pixel ratio is 2. That means that, when the **viewport** width is set to "device-width", the page is displayed as if it is 320 pixels wide instead of 640 pixels (640 divided by the device pixel ratio of 2 is 320). When you display text on a screen that has twice the pixels in width and height, you're actually *quadrupling* the number of pixels used to display the text, so letters looks crisp and sharp. Here's how it works:



Text scales well in this situation because of how the browser renders text in the display. The letters remain the same size visually (because they are displaying *as if* the screen was 320 pixels wide), but they actually have four times the pixels for that display. However, if you have a 320-pixel image that you scale up by 2 (to make the image *look* like it's 320 pixels, but on a 640-pixel display), that image is going to look blurry. Why? Because the rendering engine has to double every pixel for both the width and the height of the image, essentially quadrupling every pixel in the image:



So does that mean you need to create separate images for all the difference devices, and all their different device pixel ratios? If you want your images to look perfect on every device, yes! But that's unrealistic. At some point, we have to compromise and choose a design that is going to work well across a variety of devices, using media queries and JavaScript to help load images that are appropriate for a group of devices that are similar in size and resolution.

As you can probably see, that can get incredibly complex, depending on how far you go. We're going to keep our design fairly simple, but there are always tradeoffs. For instance, we chose to create images that are 640 pixels wide for the narrow design for Dandelion Tours, instead of 320 pixels, so the images will be crisp on a device like the iPhone 4s.

We chose to use 640 pixels as our breakpoint. However, newer devices like the iPhone 6 and iPhone 6+, along with various other newer Android phones, have larger resolutions, so the width is greater than 640 pixels across. For instance, the iPhone 6 is 750w x 1334h with a device pixel ratio of 2, and iPhone 6+ is 1242w x 2208h with a device pixel ratio of 3, so images that are 640 pixels across won't look crisp on these phones. Also, notice that if you turn your iPhone 6 or iPhone 6+ into landscape mode, you will switch into the wide view from the narrow view (can you see why?), which you might not want.

The Width We Use in Media Queries and the Width of a Device

It's important to notice here that the width we specify in the media queries to select style is based on the width of a device *and* the device pixel ratio. We've been using 640 pixels as our breakpoint so that all current iPhone and Android mobile *phones* use the narrow view, even including the iPhone 6+ in portrait mode (1242/3 = 414, so pages on the iPhone 6+ display *as if* the screen was 414 pixels wide, well below our 640-pixel breakpoint). Using 640 pixels means the narrow view will be used for landscape mode for the iPhone 4, 4s, 5, and 5s, but not the 6 and 6s. Similarly, some Android devices will be wider than 640 pixels in landscape mode. However, if you load Dandelion Tours into your iPad, even an older model iPad, like the

iPad 2, you'll see the wide view of the page, because the iPad 2 has a width of 768 pixels and a device pixel ratio of 1 (so the device width and the apparent width are the *same*).

Until recently there was no way for you to detect the device pixel ratio or resolution of a display using JavaScript or CSS media queries. However, these features are being added, which means we'll have more flexibility in our designs in the future. For instance, the <picture> element allows you to specify more than one image in the **srcset** attribute of the <source> element, with an optional device pixel ratio for each image:

OBSERVE:

```
<picture>
    <source media="(max-width: 640px)"
        srcset="image1.jpg, image2.jpg 2x">
    ...
</picture>
```

This says "load image2.jpg if the device has an device pixel ratio of 2; otherwise load image1.jpg."

Unfortunately, various devices (for example, Apple and Android) don't yet have a common way of specifying device pixel ratio or resolution in CSS and JavaScript, so we're still very much in the development stages of the technology for dealing with the wide variety of displays that are out there. However, you can probably imagine we'd like to be able to write media queries based on both the resolution of the display and the width and device pixel ratio of the device. Support for these features is spotty and a bit inconsistent at the moment, but we're getting there.

If you're interested in delving more deeply into this incredibly complex and rapidly evolving topic, check out the links below:

- Designer's Guide to DPI
- iOS Resolution Quick Reference
- iPhone 6 Screens Demystified
- Supporting Multiple Screens on Android
- More about Device Pixel Ratio

Setting breakpoints in a world full of different devices

Now that there are so many devices on the market with such a huge variety of displays and sizes, it doesn't really make much sense to set breakpoints based on the width of a specific phone. We've been using 640 pixels because that size corresponds to a set of devices that were on the market for a while. Picking a position for a breakpoint (or perhaps more than one) these days is a lot more complex.

Some developers advocate for choosing a breakpoint based purely on how your design looks. Create a design, say a narrow design, and expand your browser until that design stops looking nice and is no longer easy to use or easy to read. That's where a breakpoint should go. You can do this one or more times depending on how many breakpoints you think you need for your design.

With images specifically, it's important to keep in mind the tradeoff between an image size that will look good on high density displays and a size that will be quick to download on mobile devices. Some images are less important than others, so you can get away with either eliminating the download for the mobile version of a site (using media queries for background images, or using JavaScript to load only the important images), or using lower-resolution images. Ultimately, it's up to you as the page designer and implementor to make these decisions.

Next lesson we'll look at more design choices we might make for mobile devices (narrower screens). Before you go on, get some practice thinking about responsive images by doing the quizzes and the project.

Styling Choices for Mobile Devices (Narrow Screens)

Lesson Objectives

When you complete this lesson, you will be able to:

- design the layout and style for your page so it works well in a narrow view (and mobile).
- hide or move elements into a different layout depending on the width of the view.
- describe some of the issues to consider when planning a mobile view.
- use media queries to create different layouts for different views.
- use jQuery to hide and show elements, and use animations to provide a good user experience.

In this lesson, we'll continue working on the style and layout for Dandelion Tours, focusing specifically on the narrow view and how it will work on mobile devices.

Mobile

Let's tackle a few more layout issues for the narrow view of Dandelion Tours. First, we're going to change the layout of the narrow view so that the main part of the page (the part of the page that's styled with **main.css**) is a *single column* of information. This is a common layout for mobile devices, and allows the content to use the maximum amount of width possible on the mobile display. Second, we'll implement drop-down menus for the **Menu** and **Search** buttons that are visible in the narrow view. In this lesson, you'll see how it's possible to change how your content is presented quite a bit without having to make any changes to your HTML.

Mobile Devices are Optional

Although we discuss how the Dandelion Tours page should be structured and styled for mobile devices in this lesson, you don't require a mobile device to complete the exercises. Most of the features we discuss are the same in the "narrow" view of the web page on desktop computers. If you do have a mobile device, however, we encourage you to try your work in the browser on your device. Use the caniuse.com page to verify that the features we're using to create the page will work on your particular device and browser. The URL at the top of the page when you preview should work fine on your device. Also, use the links we provided earlier in the course to check the screen resolution of your device so you know whether to expect the narrow view or the wide view, based on the width and device pixel ratio of your device.

The Experience Section: Removing the <aside> Element

Alright, let's get to work on the experience section. We've already got a media query for this section that specifies a breakpoint at 640 pixels, using a smaller image for the background of the section in the narrow view, and a larger image for the background in the wide view. In addition, we might want to remove the <aside> element from the narrow view. The aside works well in the wider view, but not in the narrow view—it gets too narrow as the width of the browser gets smaller, and the experience section ends up looking a bit cluttered.

We can remove the <aside> section entirely for the narrow view, using the same strategy we used to remove the "full" logo in the narrow view. Let's do that now. Modify main.css as shown:

CODE TO TYPE:

```
...
@media only screen and (min-width: 641px) {
    section#experience {
        background-image: url("https://courses.oreillyschool.com/html5_responsive_design/images/TourGuide.jpg");
        min-height: 400px;
    }
    section#experience aside {
        position: absolute;
        bottom: 0px; right: 0px;
        width: 25%;
        padding: .8em 3% 1.2em 2%;
        background-color: rgba(0, 0, 0, .5);
    }
}
@media only screen and (max-width: 640px) {
    section#experience {
        background-image: url("https://courses.oreillyschool.com/html5_responsive_design/images/TourGuide_sm.jpg");
        background-position: center top;
        padding: 0em 30% .8em 2%;
    }
    section#experience aside {
        display: none;
    }
}
...
section#experience aside {
    position: absolute;
    bottom: 0px; right: 0px;
    width: 25%;
    padding: .8em 3% 1.2em 2%;
    background-color: rgba(0, 0, 0, .5);
}
```

main.css and your **index.html** file. Now, when your browser is wide (greater than 640 pixels), you'll see the <aside> as normal; but when your browser is narrow, that <aside> will disappear. This makes the mobile version of the site more streamlined and easier to read.

To accomplish this, all we did was move the rule for the <aside> element from the experience section into the media query for a browser width of greater than 640 pixels, and then add a new rule in the media query for a maximum width of 640 pixels to hide the <aside> by setting the **display** property to **none**. Note that the <aside> is still downloaded by the mobile version of the page because the HTML hasn't gone away. However, in this case, the amount of data is very small so this won't slow down the mobile site.

The Details Section

The details section consists of a <header> with two headings and a with more details about the tours. In our current layout, the <header> and the are displayed in two columns, with the <header> in a column on the left, and the in a slightly wider column on the right. This layout works well for wide views where we have plenty of room to display the two columns side by side.

However, in a narrow view such as a mobile device, it would be much better to show the <header> above the so they appear in a single column. We don't have to do anything complicated to make this happen because this would be the normal display of the two parts of the section if we weren't using the table display. So, we need a media query to specify the table display only for views wider than 640 pixels; for views narrower than that, we just default to the normal display ordering for the section. We'll also move the background image we're using for the <header> in the wide view to the bottom of the <section> in the narrow view, where it will have more room. Modify main.css as shown:

CODE TO TYPE:

```
...
section#details a {
    color: rgb(39, 69, 189);
}

@media only screen and (min-width: 641px) {
    section#details {
        display: table;
    }
    section#details header {
        display: table-cell;
        background-color: rgba(39, 69, 189, .9);
        background-image: url("https://courses.oreillyschool.com/html5_responsiv
e_design/images/dlogo_small.png");
        background-position: right bottom;
        background-repeat: no-repeat;
        width: 25%;
        color: white;
        padding: .2em .2em .2em 2%;
    }
    section#details header h1 {
        padding-top: 1em;
    }
    section#details header h2 {
        font-style: italic;
        padding: .3em 0;
    }
    section#details div {
        display: table-cell;
        width: 75%;
        padding: .8em 3% .8em 2%;
    }
}

@media only screen and (max-width: 640px) {
    section#details {
        background-image: url("https://courses.oreillyschool.com/html5_responsiv
e_design/images/dlogo_small.png");
        background-position: right bottom;
        background-repeat: no-repeat;
    }
    section#details header {
        color: white;
        background-color: rgba(39, 69, 189, .9);
        padding: .2em 3% .8em 2%;
        font-size: 1.5em;
    }
    section#details div {
        background-color: rgba(255, 255, 255, .7);
        padding: .2em 3% .2em 2%;
    }
}

section#photos {
...
```

main.css and your **index.html** file. Your wide view looks the same, but your narrow view looks quite a bit different. The <header> appears above the , and you now see the dandelion background image at the bottom of the <section> instead of the <header>.

First, we moved most of the existing rules for the details section into a new media query for the wide view. We then added several new rules for the narrow view into a new media query:

OBSERVE:

```
section#details {
    background-image: url("../images/dlogo_small.png");
    background-position: right bottom;
    background-repeat: no-repeat;
}
```

The first rule **sets the background image** of the <section> to the same dandelion image we use for the <header> in the wide view. Remember, this image is a transparent png, so the background color of the element into which it's placed will show through. In this case, the background color of the <section> is white (by default). **We'll put the image at the bottom right corner** like we did before, so it doesn't overlap the text in the <section> too much. However, we also need to reduce the opacity of the image just a bit the text isn't difficult to read. We take care of that by **setting the background color** of the in the section with this rule:

OBSERVE:

```
section#details div {
    background-color: rgba(255, 255, 255, .7);
    padding: .2em 3% .2em 2%;
}
```

Here, we **set the background color of the to white, with an alpha value (opacity) of .7 (70%)**. That means that the dandelion image in the background of the <section>, which we set above, will show through the background of the just a bit. By setting the alpha to .7, we effectively reduce the opacity of that image so it doesn't make the text too difficult read.

The <header> properties are basically the same as they are for the wide view, except for the background image.

Photos

We've already done quite a bit of work in the photos section. We show three images in the wide view, and we added a middle-width view in which we show two images (up to 1000 pixels wide). We also allow the images to expand to fill their flex boxes up to the full size of the image so in the wide view, the user can expand the width of the browser and see the image at its largest size (in case they want to see details in the photos).

In the narrow view, however, it might be better to display the images vertically so the user can see the photos at the largest size possible on their devices.

Right now we have one media query for the photos section, which removes the third image if the view is narrower than 1000 pixels:

OBSERVE:

```
@media only screen and (max-width: 1000px) {
    ...
}
```

We need two more media queries: one to display the photos using the flex box layout we created earlier if the width of the display is greater than 640 pixels, and one to display the photos in a vertical column if the width is less than 640 pixels. We'll also keep the middle view we have for widths between 640 pixels and 1000 pixels to remove the third image from the flex box display in that case. Here's how it looks with these changes. Modify main.css as shown:

CODE TO TYPE:

```
...
@media only screen and (min-width: 641px) {
    section#photos div.photo-container {
        display: -ms-flex;
        display: -webkit-flex;
        display: flex;
        -webkit-justify-content: center;
        -ms-flex-pack: center;
        justify-content: center;
        align-items: flex-start;
        margin: auto;
    }

    section#photos div.photo-with-caption figure {
        padding: 1.5em;
        text-align: center;
    }

    section#photos div.photo-with-caption img {
        width: 100%;
    }
}

@media only screen and (min-width: 641px) and (max-width: 1000px) {
    section#photos div.photo-with-caption:nth-child(3) {
        display: none;
    }
}

@media only screen and (max-width: 640px) {
    section#photos div.photo-with-caption figure {
        position: relative;
    }
    section#photos div.photo-with-caption figure img {
        width: 100%;
        display: block;
    }
    section#photos div.photo-with-caption figure figcaption {
        position: absolute;
        bottom: 0;
        background-color: rgba(255, 255, 255, .7);
        padding: 1em 1em 1em 2%;
        box-sizing: border-box;
        width: 100%;
    }
}
```

main.css and your **index.html** file. At the wide and middle width views, you'll see no change, but if you make your browser narrower than 640 pixels the page changes so that the images are now in a single column, with each caption sitting at the bottom of the image with a partially transparent white background.

Each image is set to take up 100% of the width of the page (up to the width of the image itself, which is also 640 pixels). We do that with this rule:

OBSERVE:

```
section#photos div.photo-with-caption figure img {
    width: 100%;
    display: block;
}
```

We **set the width property to 100%**, but we also have to **set the display property to block**. This ensures there's no extra space beside the image, and prevents the browser from adding a horizontal scroll bar (if you leave the images as inline elements, which is what they are by default, the horizontal scroll bar appears).

The other two rules style each <figcaption> element at the bottom of each corresponding photo. We first **set the position of the <figure> elements to relative**:

OBSERVE:

```
section#photos div.photo-with-caption figure {
    position: relative;
}
```

Now we can position the <figcaption> element, which is a child of the <figure> element, relative to its parent <figure> element. We do that by **setting the position of the <figcaption> element to absolute** and its **bottom offset to 0 pixels**:

OBSERVE:

```
section#photos div.photo-with-caption figure figcaption {
    position: absolute;
    bottom: 0;
    background-color: rgba(255, 255, 255, .7);
    padding: 1em 1em 1em 2%;
    box-sizing: border-box;
    width: 100%;
}
```

This rule ensures the <figcaption> sits flush with the bottom of the <figure> (and thus the image the <figure> contains). We **set the background color of the caption to white, with an alpha of .7**, and **add some padding**.

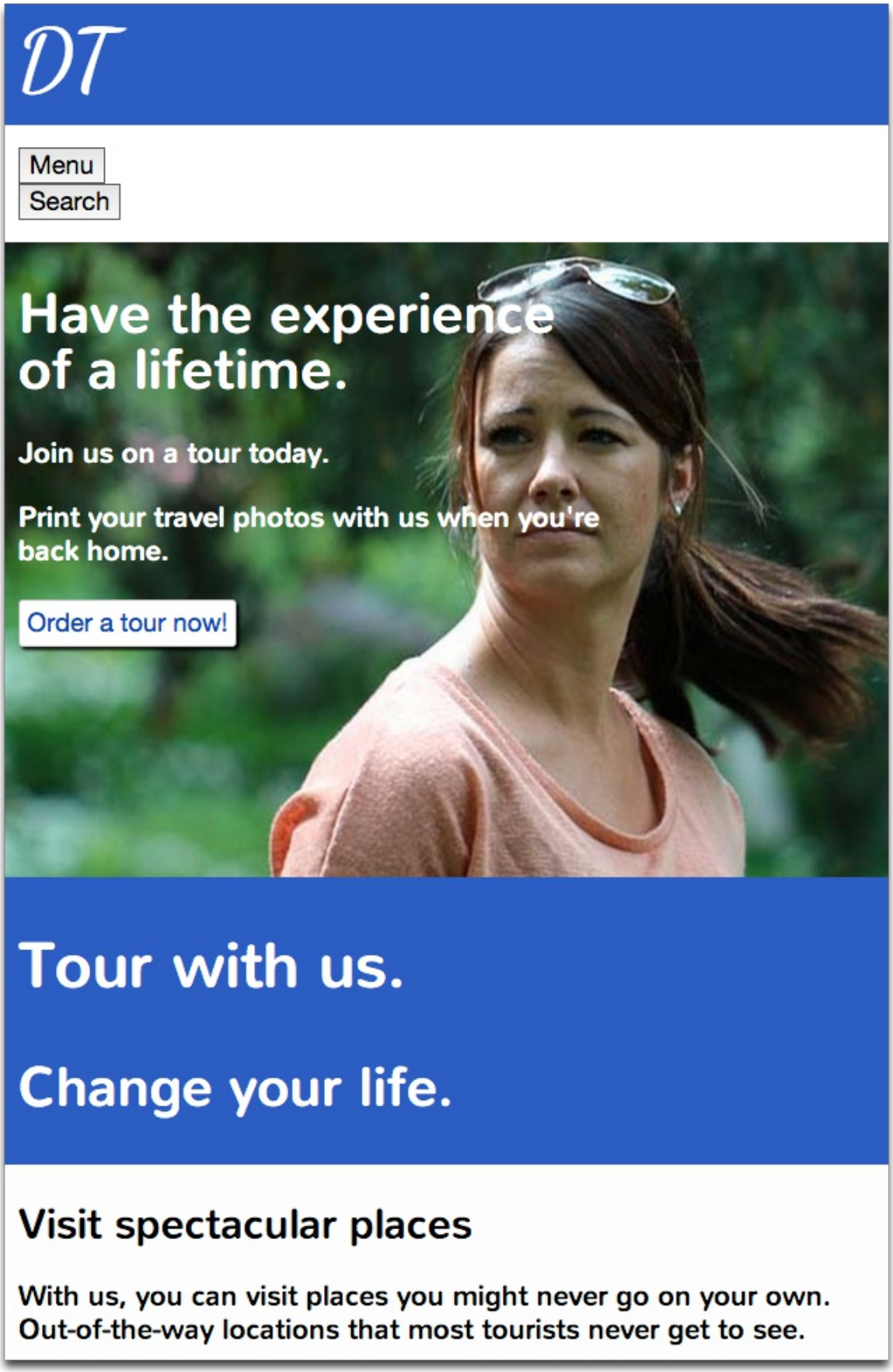
We want the <figcaption> element to take up the full width of the page, but because <figcaption> is positioned absolutely, it sits out of the flow of the page, so setting the width to 100% will actually cause the <figcaption> to expand in size beyond the total width of the page. This happens because the width does not include the padding, only the content. A width of 100% sets the width of the element relative to its container element. Well, since the <figcaption> is positioned absolutely, the container element is the <body>, so the width is set to 100% of the <body>. This would work great if the width included the padding but, by default, it doesn't.

We can fix this by using the (new) **box-sizing** property, and setting the value to **border-box**. This tells CSS to use the padding and border when computing the width of the <figcaption> element. Our <figcaption> does not have a margin, so using **border-box** means we're specifying the width of the entire element. So now, setting a width of 100% ensures the <figcaption> fits entirely in the width of the page, and prevents the browser from creating a horizontal scrollbar.

Avoiding a horizontal scrollbar is especially important on mobile devices, so users don't have to swipe left to see the right part of the page. Our goal is to get the content to fit precisely in the width of the mobile page.

Creating Menu Buttons for the Narrow View

We've got a great looking vertical display for the narrow view of our web page, which is going to work well on mobile devices, but we still need to fix the navigation part of the page for the narrow view. Currently, the top part of the page looks like this:



We've got two buttons in the navigation bar, but we haven't styled these buttons or implemented the menus the user should see when they tap on the buttons. Both buttons have the "mobile" class, so we're going to style them using that class. Take a look at the HTML for the <nav> element:

OBSERVE:

```
<nav>
    <div class="menu mobile">
        <button id="menuButton">Menu</button>
    </div>
    <ul class="menu full">
        <li><a href="travel.html">Tours</a></li>
        <li><a href="gallery.html">Gallery</a></li>
        <li><a href="about.html">About</a></li>
        <li><a href="contact.html">Contact</a></li>
    </ul>
    <div class="search mobile">
        <button id="searchButton">Search</button>
    </div>
    <div class="search full">
        <form><input type="search" id="searchInput" placeholder="search"></form>
    </div>
</nav>
```

We want to style the buttons so they fit well into a page designed for the mobile screen. In addition to style, we need to write some JavaScript code that will show a menu of links (in the case of the "Menu" button) or a search input (in the case of the "Search" button). We'll tackle the style first, and then come back to implement the JavaScript.

Take a look at the file **dt.css**. Remember that the rules for the <nav> are in this file because they will be included in every file for the site, and aren't specific to the main page. So far, we've got a couple of general rules for styling the navigation, plus two media queries: one for the wide view, and one for the narrow view with just one rule that hides the "full" versions of the navigation bar in the narrow view:

OBSERVE:

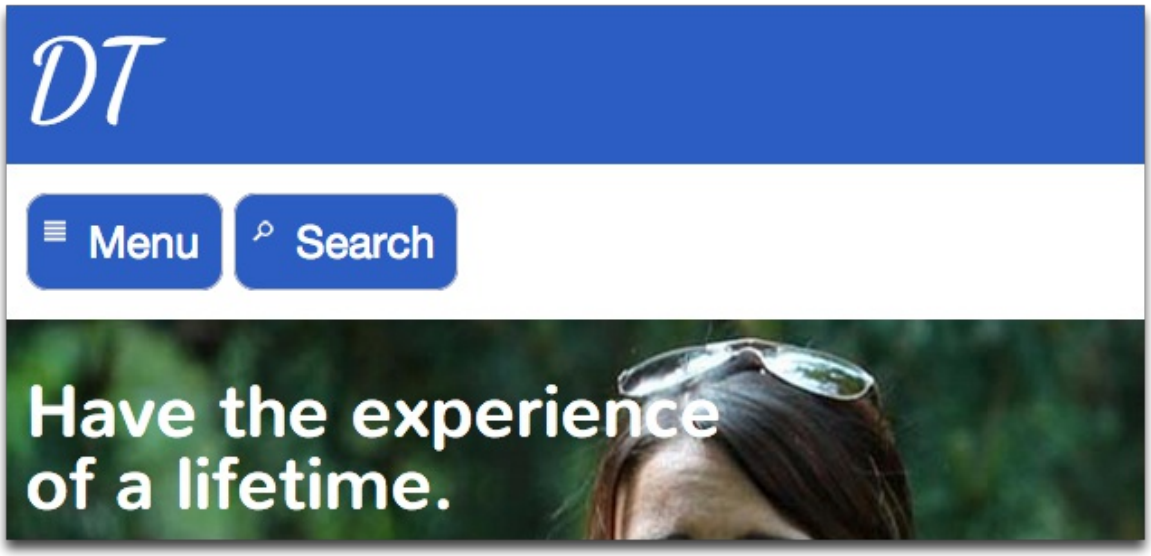
```
@media only screen and (min-width: 641px) {
    .mobile {
        display: none;
    }
    nav > ul.menu > li {
        display: inline-block;
        padding: .2em 1em .2em 0;
    }
    nav > div.search.full {
        position: absolute;
        top: 0px;
        right: 0px;
        padding: .8em 3% .8em 2%;
    }
}

@media only screen and (max-width: 640px) {
    .full {
        display: none;
    }
}
```

We're going to add several rules to the media query for the narrow view to position and style the two buttons. Edit your file **dt.css** and add the CSS below:

CODE TO TYPE:

```
...
@media only screen and (max-width: 640px) {
    .full {
        display: none;
    }

    nav > div.mobile {
        display: inline;
    }
    nav > div.mobile > button {
        background-color: rgba(39, 69, 189, .9);
        background-origin: border-box;
        background-position: .2em .4em;
        background-repeat: no-repeat;
        color: white;
        border-radius: .5em;
        border: 1px solid silver;
        padding: .5em .5em .5em 1.4em;
        font-size: 1.2em;
        outline: none;
    }
    #menuButton {
        background-image: url(https://courses.oreillyschool.com/html5_responsive
_design/images/appbar.lines.horizontal.4_24.png);
    }
    #searchButton {
        background-image: url(https://courses.oreillyschool.com/html5_responsive
_design/images/appbar.magnify_24.png);
    }
}
...
```

Save **dt.css** and your **index.html** file. Now the navigation buttons look quite different:



Let's go over the rules we used to style the buttons. First, we set the **display** of both the elements that contain the buttons to **inline** so that the buttons sit next to each other in the page. We then set various properties of the <button> elements inside the s to style the buttons to fit in with the look and feel of the Dandelion Tours page, with a blue background, rounded corners and white text. We use a background image for both buttons, defined in the bottom two rules, so we also go ahead and set some rules for the background that apply to both buttons. Then we create two rules using the IDs of the two buttons to set the background image for each. Both of these images are transparent png images, so the background color shows through. The icons we use for two buttons—the horizontal lines for the links menu button, and the magnifying glass for the search button— are familiar to users and help them quickly identify what they'll get when they use these

buttons.

Notice the **outline** property, which is set to **none**. This prevents the blue outline that the browser adds to a button that's been clicked (or tapped) from appearing, which improves the look of the buttons. We don't need that outline to give feedback to the user that the button's been clicked, because we're going to display the menu associated with the button they just clicked.

Okay, we've got the buttons styled, but they don't do anything yet. To get the buttons to display content, we need to add some JavaScript code. Our goal is to display content from the "full" element corresponding to the button the user clicks on. So if they click on the button with the classes "menu mobile," we'll display a menu of links from the "menu full" element. If they click on the button with the classes "search mobile," we'll display the search input from the "search full" . That way, we're repurposing the content from the wide view for the pop-up menus in the narrow view.

We'll use jQuery for this code, so if you commented out the link to jQuery earlier, edit your **index.html** file, and add that back in. We'll also add in a new link to the new file **dt.js** that you'll create in just a moment. Edit your index.html file and update your HTML:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Dandelion Tours</title>

<script src="http://code.jquery.com/jquery-latest.min.js"></script>
<script src="dt.js"></script>
<script src="main.js"></script>
...
```

Don't bother previewing yet, because we still need to add the jQuery code before anything new will happen. We'll use jQuery here to make use of the jQuery **slideDown** and **slideUp** animations. When the user clicks on (or taps) one of the buttons, the menu below the button will be revealed using this animation so it appears smoothly over a short time and not suddenly, which is a bit jarring. This is a nice effect, and fairly straightforward to implement using jQuery. If you need a refresher on these two animations, check out the jQuery documentation for slideDown and slideUp. Create a new file and add the following JavaScript:

CODE TO TYPE:

```
$(document).ready(function() {
    $("#menuButton").click(function() {
        var $searchMenu = $("nav div.search.full");
        if ($searchMenu.hasClass("menu-display")) {
            menuUpDown($searchMenu);
        }
        var $menu = $("nav ul.menu.full");
        menuUpDown($menu);
    });
    $("#searchButton").click(function() {
        var $menu = $("nav ul.menu.full");
        if ($menu.hasClass("menu-display")) {
            menuUpDown($menu);
        }
        var $searchMenu = $("nav div.search.full");
        menuUpDown($searchMenu);
    });
});

function menuUpDown($menu) {
    if ($menu.hasClass("menu-display")) {
        $menu.slideUp(500, function() {
            $(this).removeClass("menu-display");
        });
    } else {
        $menu.addClass("menu-display").slideDown(500);
    }
}
```

Save it as **dt.js** and your **index.html** file. Try clicking on the Menu button and then the Search button. When you click on the Menu button, the links appear below the two buttons. Similarly, when you click on the Search button, the search input appears below the two buttons.

Right now the style of the "full" items we're displaying when you click on the two buttons doesn't work well, but we can fix that in a moment. Let's go over the JavaScript code. Note that we're using the class "**menu-display**" to indicate which menu we're displaying: the menu of links or the search input menu. We want only one item with the "menu-display" class to be shown at a time. So, when you click on the menu button, we first **check to see if the search button menu is visible**. If it is, we **hide the search button menu** before we show the links menu:

OBSERVE:

```
$("#menuButton").click(function() {
    var $searchMenu = $("nav div.search.full");
    if ($searchMenu.hasClass("menu-display")) {
        menuUpDown($searchMenu);
    }
    var $menu = $("nav ul.menu.full");
    menuUpDown($menu);
});
```

Similarly, **if you click on the search button**, we first **check the links menu to see if it is being displayed**. If it is, we **hide that before we show the search menu**:

OBSERVE:

```
$("#searchButton").click(function() {
    var $menu = $("nav ul.menu.full");
    if ($menu.hasClass("menu-display")) {
        menuUpDown($menu);
    }
    var $searchMenu = $("nav div.search.full");
    menuUpDown($searchMenu);
});
```

We set up these two click handlers in the jQuery ready function. Notice that jQuery allows you to define multiple ready functions, so this function will not conflict with the ready function we're using to set up the images in the file **main.js**.

We're using a helper function **menuUpDown** to show and hide the menus:

OBSERVE:

```
function menuUpDown($menu) {
    if ($menu.hasClass("menu-display")) {
        $menu.slideUp(500, function() {
            $(this).removeClass("menu-display");
        });
    } else {
        $menu.addClass("menu-display").slideDown(500);
    }
}
```

This function **checks to see whether the menu object we pass in is already being displayed**. We know it is if that menu has the "menu-display" class. If it is being displayed, that means we should hide the menu (that is, you clicked once on the button to show it, and now you've clicked again to hide it). That gives users an easy way to hide a menu. We use the jQuery **slideUp** method to hide the menu, and then **remove the "menu-display" class** once that animation is complete. It's important to wait until the animation is done before removing the class, otherwise, the menu will change mid-animation.

If the menu is not already being displayed when you click on the button, **we display it, using the jQuery method slideDown**.

Let's work on the style for the menus using the "menu-display" class. Remember, only one menu at a time will have this class, so we can use the same class to target both the links menu and the search menu. Modify dt.css as shown:

CODE TO TYPE:

```
...
@media only screen and (max-width: 640px) {
    ...
    #searchButton {
        background-image: url(https://courses.oreillyschool.com/html5_responsive
_design/images/appbar.magnify_24.png);
    }
    .menu-display {
        position: absolute;
        top: 3.6em;
        left: .8em;
        background-color: rgba(39, 69, 189, .9);
        padding: .8em;
        z-index: 1;
        border-radius: 0em 1em 1em 1em;
    }
    .menu-display > li, .menu-display > form {
        display: block;
        padding: .8em
    }
    .menu-display a {
        color: white;
    }
}
...
```

dt.css and **index.html**. Now, when you click on the Menu button, a much nicer menu of links appears below the buttons; it's the same for the Search button. Try clicking on the menu items in varying orders to verify that the code works as you expect: if you click on Menu, the links menu appears; click on it again, and the links menu disappears. If you click on Menu again, the links menu appears; now if you click on Search, the search menu appears and the links menu disappears. At this stage, it appears the code is working well.

These click handlers also handle taps on touch devices, so if you try this web page on your touch phone you'll see that the buttons work the same way.

Earlier we added the **position** property to the <nav> and set it to **relative**. So, when we position the links and search menus using **absolute** positioning, we are positioning them relative to the <nav> element.

This is looking pretty good, but there's a small bug we need to fix. To see the bug, click on the links menu button and then again, making sure that the links menu is closed. Then expand the browser window wide enough that you see the wide view. Notice that the links do not appear in the navigation bar where they should. Now, reduce the width of the browser again. Click on the menu button to show the links menu. Expand the browser to the wide view, and then back to the narrow view. You will probably see the search button *below* the menu button instead of beside it.

So what's causing this bug? It's actually a side effect of the jQuery slideUp and slideDown animations. To see the problem, close the links menu again, and expand the browser to the wide view. The links should not show up in the navigation bar. Now, use the browser's developer tools to inspect the HTML (for example, in Chrome, use the Elements tab in the developer tools).

Open the <nav> element and look at the <ul> element; you can see that a **style** attribute has been added, with a property setting the **display** to **none**. This is added by the jQuery slideUp animation to hide the element you're sliding up. Unfortunately, this **style** attribute doesn't go away when we switch to the wide view, so even though the "full" menus are set to display in the wide view, this **style** attribute on the HTML element overrides the CSS in the **dt.css** file that we've created, so you don't see the links menu. (The same thing happens with the search menu).

To fix this bug, we're going to remove this **style** attribute, along with our "menu-display" class, from both the elements with the "full" class whenever the user resizes the window wider than our breakpoint of 640 pixels. By removing the **style** attribute on a resize, the only CSS that applies to that element is the CSS in our **dt.css** file, which is what we want. We don't need the **style** attribute or the "menu-display" class on our menu items when we're in the wide view because in that view, we always want those "full" menu items to show.

Edit your file **dt.js** and add a resize handler to your code, like this:

CODE TO TYPE:

```
$(document).ready(function() {
    $(window).resize(function() {
        if ($(window).width() > 640) {
            $("nav ul.menu.full").removeClass("menu-display").removeAttr("style"
);
            $("nav div.search.full").removeClass("menu-display").removeAttr("sty
le");
        }
    });
    $("#menuButton").click(function() {
        var $searchMenu = $("nav div.search.full");
        if ($searchMenu.hasClass("menu-display")) {
            menuUpDown($searchMenu);
        }
        var $menu = $("nav ul.menu.full");
        menuUpDown($menu);
    });
    $("#searchButton").click(function() {
        var $menu = $("nav ul.menu.full");
        if ($menu.hasClass("menu-display")) {
            menuUpDown($menu);
        }
        var $searchMenu = $("nav div.search.full");
        menuUpDown($searchMenu);
    });
});
...
```

dt.js and your **index.html** file. Your menus now behave themselves properly as you widen or narrow your browser window. That is, you see the two buttons, with correct behavior in the narrow view, and the full links menu and search input in the wide view.

On mobile, the buttons work just like in the narrow view. Using buttons like this in mobile helps to save space on the small screen. Another option that many use on mobile is to move the navigation to the bottom of the screen and "stick" it there (so it's always visible, even as you scroll).

Of course, the links to the other pages in the site don't work (because we don't have any other pages yet), and the search doesn't work (we haven't implemented that either), but you can see how you'd use these menus in either the narrow view or the wide view. This is just one of many ways we could have implemented the two buttons and their menus for the narrow, or mobile view; you might choose a completely different way to display content like this for the two different views for your own site. However, we hope this lesson has given you a taste of how you can change the way content works between two different views using media queries, CSS and a little bit of JavaScript (and no changes to the HTML).

In this lesson, you modified the narrow, or mobile, view of the page quite a bit, using media queries, CSS, and JavaScript. We removed the <aside> in the experience section of the page entirely for the narrow view, and also changed how we display the content in the details and photos section to use the full width of the page for each content item, making it easier for the user to read and see the images. We also styled the two buttons in the navigation bar for the mobile view, and reused the HTML from the elements with the "full" class to create two menu items for each button that slide open and closed when you click or tap on the buttons.

You might have noticed that there's no button on the search form input in the mobile display. Unlike on a desktop browser, where you can implement a "return" keypress handler to begin the search, in mobile there's no such action as pressing a return key. So think about how you might modify the search drop-down in the narrow view to add a feature that allows the user to easily submit the search. (You'll be implementing this as an exercise for this lesson).

Responsive Design in the About Page

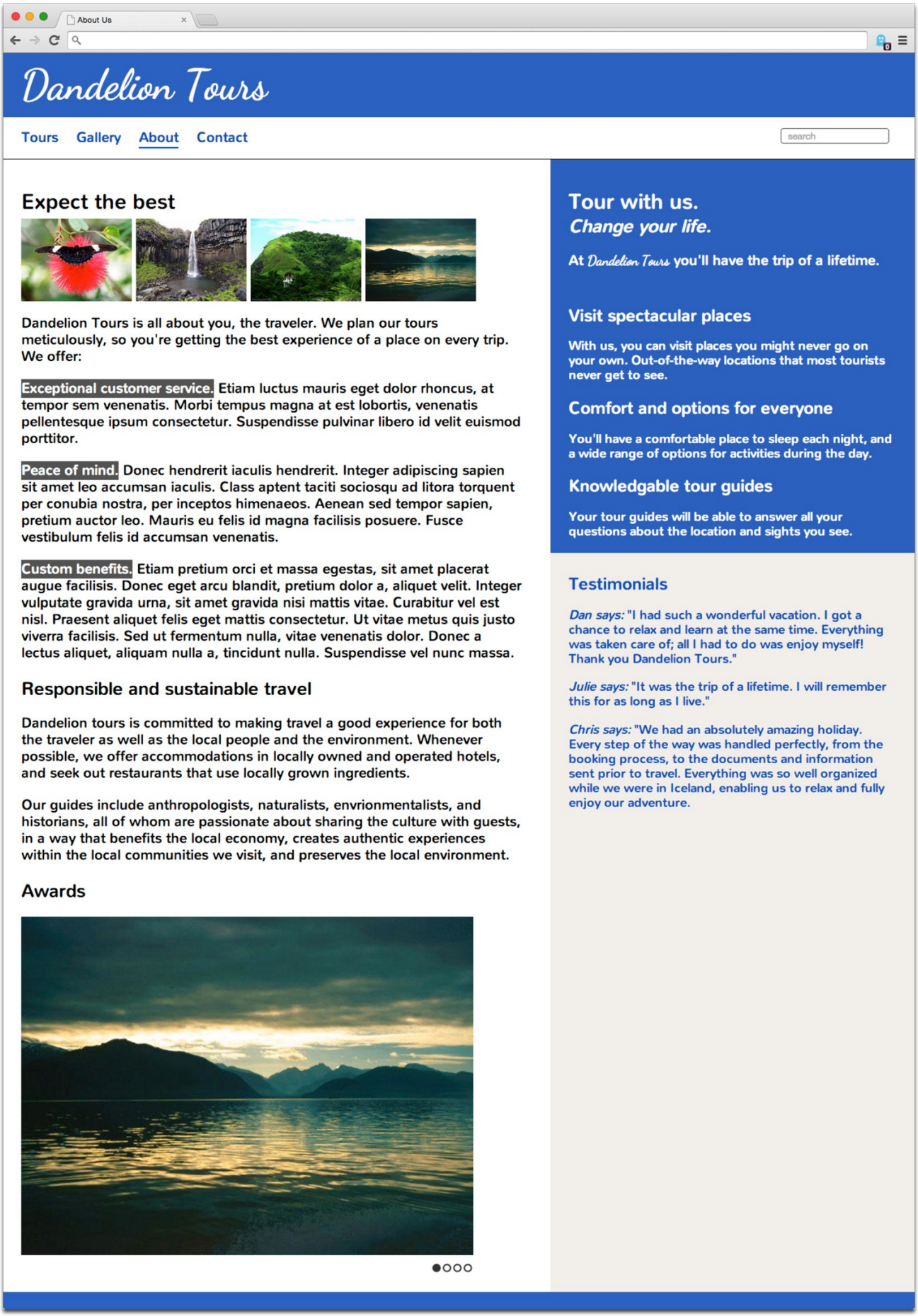
Lesson Objectives

When you complete this lesson, you will be able to:

- use table display to create a two-column page.
- use flex box to reorder elements in the page.
- create an image slide show with the SlideJS JavaScript library.
- use CSS sprites to make your web page more efficient.

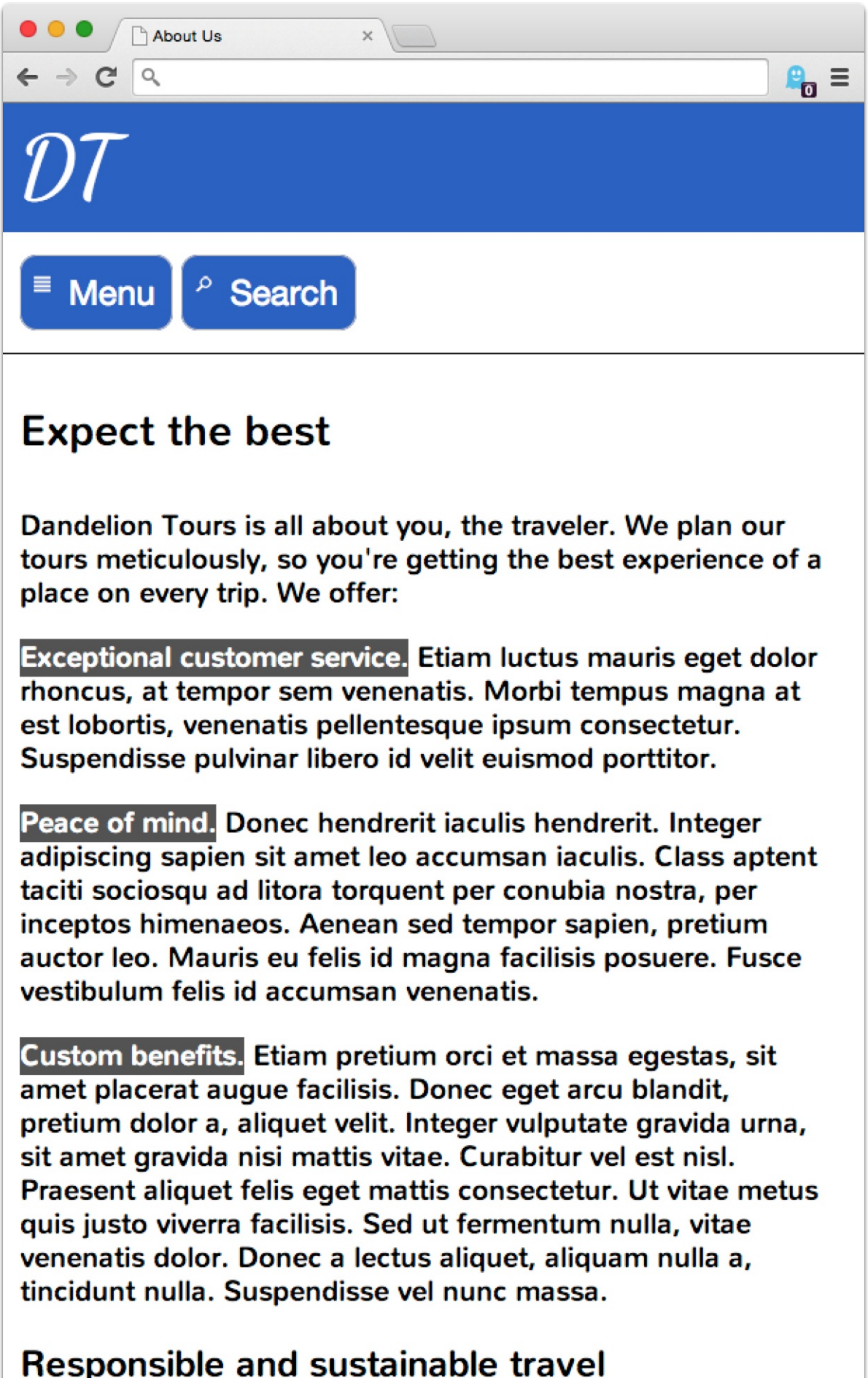
Adding a Page to the Dandelion Tours Site

The next challenge for us is to add another page to Dandelion Tours: the About page. This page is going to be a bit different from the home page in that it will use the structure we talked about at the very beginning of the course: a section with the primary content for the page, and a section that will be used as a sidebar. In the wide view, these two sections will be side by side, with the primary content on the left, and the sidebar on the right:



In the mobile view, we'll move the images in the primary content below the text (because they are slightly less

important than the text) and move the sidebar below the primary content:



Responsible and Sustainable Travel

Dandelion tours is committed to making travel a good experience for both the traveler as well as the local people and the environment. Whenever possible, we offer accommodations in locally owned and operated hotels, and seek out restaurants that use locally grown ingredients.

Our guides include anthropologists, naturalists, envrionmentalists, and historians, all of whom are passionate about sharing the culture with guests, in a way that benefits the local economy, creates authentic experiences within the local communities we visit, and preserves the local environment.

Awards

TOUR WITH US.

Change your life.

At *Dandelion Tours* you'll have the trip of a lifetime.

VISIT SPECTACULAR PLACES

With us, you can visit places you might never go on your own. Out-of-the-way locations that most tourists never get to see.

COMFORT AND OPTIONS FOR EVERYONE

You'll have a comfortable place to sleep each night, and a wide

However, for all the pages in the site, we're using the same header, navigation, and footer, so we'll copy that structure over from the **index.html**, and we'll reuse all the style in the **dt.css** file.

To create the layout for the About page, we'll use **table display** and **flex box** again, like we did for part of the home page, and explore a feature of flex box that allows you to change the ordering of the content in the page. We'll use this feature to move the photos that appear at the top of the About page in the wide view below the primary content text when we're in the narrow view.

We'll also use a JavaScript library, SlidesJS, to create an image slideshow for the Awards section of the About page. We chose this library because it's responsive: that is, it works well at a variety of different window widths, as well as on mobile devices. Along the way, we'll look at a way to use jQuery to load snippets of the page, and explore CSS sprites.

Let's get started!

The Structure and Content of the About Page

The About page is going to use the same header, navigation, and footer as the Home page (in **index.html**), but the rest of the structure will be a bit different. For the About page, and all the other pages that aren't the Home page, we'll split the page into two parts: the primary content, and a sidebar. The sidebar will be the same across all the pages that have it.

To create the About page, you can copy your **index.html** page to a new file, **about.html**, and make the changes shown below, or you can copy and paste the HTML below into **about.html**. We've highlighted what's different in the HTML below from what is in **index.html**. Create a new file and add the HTML:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>About Us</title>

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

<link href='http://fonts.googleapis.com/css?family=Nunito' rel='stylesheet' type
='text/css'>
<link href='http://fonts.googleapis.com/css?family=Dancing+Script' rel='styleshe
et' type='text/css'>

<link rel="stylesheet" href="reset.css">
<link rel="stylesheet" href="dt.css">
<link rel="stylesheet" href="about.css">
<link rel="stylesheet" href="sidebar.css">

<script src="http://code.jquery.com/jquery-latest.min.js"></script>
<script src="dt.js"></script>

</head>

<body>

<header>
    <div class="full">
        <a href="index.html" class="name">Dandelion Tours</a>
    </div>
    <div class="mobile">
        <a href="index.html" class="name">DT</a>
    </div>
</header>
<nav>
    <div class="menu mobile">
        <button id="menuButton">Menu</button>
    </div>
    <ul class="menu full">
        <li><a href="travel.html">Tours</a></li>
        <li><a href="gallery.html">Gallery</a></li>
        <li><a href="about.html" class="selected">About</a></li>
        <li><a href="contact.html">Contact</a></li>
    </ul>
    <div class="search mobile">
        <button id="searchButton">Search</button>
    </div>
    <div class="search full">
        <form><input type="search" id="searchInput" placeholder="search"></form>
    </div>
</nav>
<section id="about">
    <header>
        <h1>Expect the best</h1>
    </header>
    <div class="about-photos">
        <img src="https://courses.oreillyschool.com/html5_responsive_design/imag
es/Brazil_Butterfly_sm.jpg" alt="Photo of butterfly in Brazil">
        <img src="https://courses.oreillyschool.com/html5_responsive_design/imag
es/Iceland_SvartifossCascade_sm.jpg" alt="Photo of waterfall in Iceland">
        <img src="https://courses.oreillyschool.com/html5_responsive_design/imag
es/Brazil_Hill_sm.jpg" alt="Photo of green hill in Brazil">
        <img src="https://courses.oreillyschool.com/html5_responsive_design/imag
es/Alaska_Sunset_sm.jpg" alt="Photo of sunset, mountains, lake in Alaska">
    </div>
    <div class="about-benefits">
        <p>
```

```
            Dandelion Tours is all about you, the traveler. We plan our tours me
ticulously,
            so you're getting the best experience of a place on every trip. We o
ffer:
        </p>
        <ul>
            <li>
                <p><span>Exceptional customer service.</span> Etiam luctus
mauris eget dolor rhoncus, at tempor sem venenatis. Morbi tempus magna
at est lobortis, venenatis pellentesque ipsum consectetur. Suspendisse pulvinar
libero id velit euismod porttitor. </p>
            </li>
            <li>
                <p><span>Peace of mind.</span> Donec hendrerit iaculis hendrerit
.
Integer adipiscing sapien sit amet leo accumsan iaculis.
Class aptent taciti sociosqu ad litora torquent per
conubia nostra, per inceptos himenaeos. Aenean sed tempor sapien, pretium auctor

leo. Mauris eu felis id magna facilisis posuere. Fusce vestibulum felis id
accumsan venenatis. </p>
            </li>
            <li>
                <p><span>Custom benefits.</span> Etiam pretium orci et massa ege
stas, sit amet placerat augue facilisis.
Donec eget arcu blandit, pretium dolor a, aliquet velit. Integer vulputate gravi
da urna,
sit amet gravida nisi mattis vitae. Curabitur vel est nisl. Praesent aliquet fel
is eget
mattis consectetur. Ut vitae metus quis justo viverra facilisis. Sed ut fermentu
m nulla,
vitae venenatis dolor. Donec a lectus aliquet, aliquam nulla a, tincidunt nulla.

Suspendisse vel nunc massa.</p>
            </li>
        </ul>
        <h2>Responsible and sustainable travel</h2>
        <p>
            Dandelion tours is committed to making travel a good experience for
both the
            traveler as well as the local people and the environment. Whenever p
ossible,
            we offer accommodations in locally owned and operated hotels, and se
ek out
            restaurants that use locally grown ingredients.
        </p>
        <p>
            Our guides include anthropologists, naturalists, environmentalists,
and historians,
            all of whom are passionate about sharing the culture with guests, in
 a way
            that benefits the local economy, creates authentic experiences withi
n the local
            communities we visit, and preserves the local environment.
        </p>
    </div>
    <div id="about-awards">
        <!-- load with jQuery -->
    </div>
</section>
<section id="sidebar">
    <header>
        <h1>Tour with us.</h1>
        <h2>Change your life.</h2>
        <p>
            At <span class="name">Dandelion Tours</span> you'll have the trip of
 a lifetime.
        </p>
```

```
    </header>
    <div>
        <ul>
            <li>
                <h1>Visit spectacular places</h1>
                <p>
                    With us, you can visit places you might never go on your own
.
                    Out-of-the-way locations that most tourists never get to see
.
                </p>
            </li>
            <li>
                <h1>Comfort and options for everyone</h1>
                <p>
                    You'll have a comfortable place to sleep each night, and
                    a wide range of options for activities during the day.
                </p>
            </li>
            <li>
                <h1>Knowledgable tour guides</h1>
                <p>
                    Your tour guides will be able to answer all your questions
                    about the location and sights you see.
                </p>
            </li>
        </ul>
    </div>
    <footer>
        <h1>Testimonials</h1>
        <p><span>Dan says:</span> "I had such a wonderful vacation. I got a chan
ce
to relax and learn at the same time. Everything was taken care of; all I had to
do was enjoy myself! Thank you Dandelion Tours."</p>
        <p><span>Julie says:</span> "It was the trip of a lifetime. I will
remember this for as long as I live."</p>
        <p><span>Chris says:</span> "We had an absolutely amazing holiday. Every

step of the way was handled perfectly, from the booking process, to the document
s
and information sent prior to travel. Everything was so well organized while we
were
in Iceland, enabling us to relax and fully enjoy our adventure. </p>
    </footer>
</section>
<footer>
    <div>
        Have a question? Email us at
        <a href="mailto:example@example.com">travel@dandeliontours.com</a>
    </div>
    <div>
        <a class="social" href="#">Facebook</a>
        <a class="social" href="#">Twitter</a>
        <a class="social" href="#">Instagram</a>
        <a class="social" href="#">Youtube</a>
    </div>
    <div>
        You'll travel with the wind. <span class="name">Dandelion Tours</span>.
    </div>
</footer>

</body>
</html>
```

□ Save it as **about.html** and □. Notice we've linked to two new CSS files that you haven't created yet, **about.css** and **sidebar.css**, and we'll get to those in a moment. However, even without those files, you can see the the content of the page and get a rough idea of what it's going to be like. The header, navigation, and footer all work as they did in the Home page because all the CSS for those parts of the file is in **dt.css**, which

we've already created and we're linking to, along with **reset.css**.

> **Note** We included some filler text (the text in Latin) to fill out this page. Feel free to replace this text with your own if you like, or just imagine the text is all about travel.

We did make one small change to the HTML for the navigation: we added a "**selected**" class to the link for the About page:

OBSERVE:

```
<li><a href="about.html" class="selected">About</a></li>
```

We'll use this class to add an underline to the link corresponding to the page we're on (in this case, the **about.html** link), so it's easy for the user to tell which page they're on when they're looking at the navigation. Let's go ahead and add the **selected** class to **dt.css** so we can see that change in the About page. Modify dt.css as shown:

CODE TO TYPE:

```
...
nav a.selected {
    border-bottom: 2px solid rgb(39, 69, 189);
}
...
```

dt.css and your **about.html** file again. Now you see an underline on the link for the About page, indicating that we are on that page. Try clicking on the Dandelion Tours logo. That will take you to the Home page (because it's linked to **index.html**), and then click on the About link to get back to **about.html**. Try this in both the wide and narrow views to make sure the navigation is working properly.

You'll probably notice that there aren't any images in the Awards section of the page yet. We'll get to those shortly.

Styling the About Page

Now that we've got the content into the page, let's style it. We'll create two new style files: **about.css** and **sidebar.css**. The style rules in **about.css** will be for the primary content on the page, that's unique to the About page, and the rules in **sidebar.css** will be for the sidebar content that can be reused in the other pages in the site. Create a new file, about.css, and add the following CSS:

CODE TO TYPE:

```
section {
    box-sizing: border-box;
    border-top: 1px solid black;
}
section#about div.about-benefits span {
    background-color: #535353;
    color: white;
}
@media only screen and (min-width: 641px) {
    section#about {
        display: table-cell;
        width: 60%;
        padding: .8em 3% .8em 2%;
    }
    section#about div.about-photos img {
        width: 22%;
    }
}
@media only screen and (max-width: 640px) {
    section#about div.about-benefits, section#about div#about-awards {
        padding: .8em 3% .8em 2%;
    }
    section#about header {
        padding: .8em 3% 0 2%;
    }
    section#about div.about-photos {
        padding: 0;
    }

    section#about {
        display: -webkit-flex;
        display: flex;
        -webkit-flex-direction: column;
        flex-direction: column;
    }
    section#about div.about-photos {
        -webkit-order: 2;
        order: 2;
    }
    section#about div#about-awards {
        -webkit-order: 3;
        order: 3;
    }
    section#about div.about-photos img {
        width: 100%;
        display: block;
    }
}
```

about.css and your **about.html** file. In the wide view, the "about" <section> appears at the top of the page (below the <header> and <nav>), and the "sidebar" <section> below that. That's because while we've added a rule for the "about" <section> to use table-display, we haven't yet added a rule to set table display for the sidebar, so the sidebar appears where it normally would in the flow of the page, below the "about" <section>. Within the "about" <section>, you'll see four small images at the top, followed by the benefits, followed by awards (again, with nothing in it yet).

Go ahead and change the width of the browser to below 640 pixels so you can see what happens in the narrow view. Those four small images at the top of the page disappear; instead, they are now *below* the benefits section. We moved an entire element, the "about-photos" , to somewhere else in the page, just by using CSS. Let's go over all the CSS to see how this works.

We'll start with the wide view media query since that's similar to what we've been doing so far.

OBSERVE:

```
@media only screen and (min-width: 641px) {
    section#about {
        display: table-cell;
        width: 60%;
        padding: .8em 3% .8em 2%;
    }
    section#about div.about-photos img {
        width: 22%;
    }
}
```

This says, **for views wider than 640 pixels**, display the **"about" <section>** as a **table-cell**. Shortly, we'll add a similar rule for the "sidebar" <section>, which will cause the two sections to sit side by side in the page. We've **set the width of the "about" <section> to 60%** of the width of the page, which means that the sidebar will take up the rest of the space (40%). We've also **set the width of the images within the "about-photos" to 22%**. Since the "about-photos" takes up 100% of the width of the "about" <section>, and the "about" <section> takes up 60% of the width of the page, the images appear to take up almost, but not quite, the entire width of the "about" <section>. Remember, when you specify the width of an element, that width is relative to the width of the containing element.

What about the narrow view? Take another look at the media query for that:

OBSERVE:

```
@media only screen and (max-width: 640px) {
    section#about div.about-benefits, section#about div#about-awards {
        padding: .8em 3% .8em 2%;
    }
    section#about header {
        padding: .8em 3% 0 2%;
    }
    section#about div.about-photos {
        padding: 0;
    }
    section#about {
        display: -webkit-flex;
        display: flex;
        -webkit-flex-direction: column;
        flex-direction: column;
    }
    section#about div.about-photos {
        -webkit-order: 1;
        order: 1;
    }
    section#about div#about-awards {
       -webkit-order: 2;
        order: 2;
    }
    section#about div.about-photos img {
        width: 100%;
        display: block;
    }
}
```

We use flex box layout in the narrow view. The **"about" <section> is the flex box container**, which means that all the elements nested within this section are the flex box items (the <header>, and the "about-photos", "about-benefits" and "about-awards" s). By default the ordering of the items in the flex box container is row, not column, which means that by default, the items would display horizontally rather than vertically in the container. We want the items to appear vertically, so we set the **flex-direction** to **column**.

In the narrow view, we want the photos to appear below the benefits text. That's because when on a mobile device, we want the user to see the most important information on the page near the top of the page. In the wide view, having the photos up at the top didn't take much room, and in any normal width browser, the benefits text will be easy to see. Not so in the narrow view, where the user would have to scroll past all the photos to get to the text.

With flex box, we can change the ordering of the flex box items in the container. We use the **order** property to do that. By default the location of an item is set to the order it appears in the flow of the page (that is, in your HTML), so normally, the "about-photos" would appear at the top of the flex box container. We change the order of the items we want by specifying a number for those items. It doesn't really matter which numbers you choose, as long as the number of the item you want to appear toward the bottom is greater than the number of the item you want to appear toward the top. Here, we set the **order of photos to 1, and awards to 2**. By default, the benefits will stay at the top (because unless an order is specified, by default, benefits will appear in the order of the flow of the page). Once we've changed the order of the photos to 1, we then need to specify an order of 2 for awards so that the awards will continue to appear at the bottom of the flex box container.

We've set the **padding** on the "about-photos" section to 0, the **width** of the images in that section to 100% and the **display** to block. That means the images will fill up the entire width of the page (they have no border or margin either), and will appear in a vertical column (because they are now block elements instead of inline elements). Notice that a width of 100% also ensures they stretch and shrink to fill the width even as you change the width of the browser a little, within the narrow view.

Note

Here are the tools you'll use to support IE10, iOS6, and older versions of Firefox and Safari.

For the display property you'll use:
display: -webkit-box;
display: -moz-box;
display: -ms-flexbox;

For the order property you'll use:
-webkit-box-ordinal-group: 2;
-moz-box-ordinal-group: 2;
-ms-flex-order: 2;

The Sidebar

Now let's style the sidebar. Right now the sidebar appears below the primary content in both the wide view and the narrow view (because of its natural ordering in the page for the wide view). We want it to appear next to the primary content in the wide view, and we want to give it some good style to fit in with the rest of the Dandelion Tours style. Go ahead and make these changes to sidebar.css:

CODE TO TYPE:

```
section#sidebar {
    background-color: rgba(39, 69, 189, .9);
    font-size: .8em;
    color: white;
}
section#sidebar header {
    font-size: 1rem;
}

section#sidebar header h1 {
    padding: 1em 0em 0em 0em;
}
section#sidebar header h2 {
    font-style: italic;
    padding: .3em 0;
}

section#sidebar header, section#sidebar div, section#sidebar footer {
    padding: .8em 5%;
}
section#sidebar footer p span {
    font-style: italic;
}

@media only screen and (min-width: 641px) {
    section#sidebar {
        display: table-cell;
        width: 40%;
        background-color: rgb(241, 238, 234);
        color: rgb(39, 69, 189);
    }
    section#sidebar header, section#sidebar div {
        background-color: rgba(39, 69, 189, .9);
        color: white;
    }
}

@media only screen and (max-width: 640px) {
    section#sidebar h1 {
        text-transform: uppercase;
    }
    section#sidebar footer {
        background-color: rgb(241, 238, 234);
        color: rgb(39, 69, 189);
    }
}
```

it as **sidebar.css** and your **about.html** file. Now the sidebar appears next to the primary content in the wide view. Change the width of your browser to the narrow view, and the sidebar moves down to below the primary content again. This is exactly what we want because the sidebar content is not nearly as important as the primary content and for users on mobile, we'd like to make sure all the primary content comes first.

At this point, the CSS for the sidebar is familiar to you: we've got two media queries, one each for the wide view and narrow view at the same breakpoint we've been using, 640 pixels. In the wide view we set the **display** to **table-cell**, which creates the side-by-side display of the primary content and the sidebar in the wide view; notice that the two columns are equal in length—a big benefit of the table display!

For the narrow view, we take away the **table-cell** display, so that the sidebar appears in the page in its normal location determined by the flow of the content (based on the HTML structure of the page). One change we make to the style of the sidebar in the narrow view is to change the headings to uppercase text. This helps to identify the sections of the sidebar in the narrow view. This is just a stylistic choice to show you yet another way you can change the look of your content in two different views.

Adding a Responsive Image Slide Show

For the awards section of the site, we're going to display images as a slide show, so you can click or swipe to move to the next slide. For this example, we'll reuse the images we've created for the Dandelion Tours photos sections, but in a real site, you could show images of the various awards that the company has won.

We'll use a JavaScript library called SlidesJS to implement the slide show for the awards. You can download your own copy of SlidesJS if you like, or you can just link to the version we've already downloaded and put here: https://courses.oreillyschool.com/html5_responsive_design/software/jquery.slides.min.js.

Add the link to SlidesJS to the top of your HTML in **about.html**. While you're at it, go ahead and add two other links: one to a CSS file we'll create shortly to hold the CSS for the awards section, and the other to the JavaScript we need to get the slide show working. Update the file about.html as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Dandelion Tours</title>

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

<link href='http://fonts.googleapis.com/css?family=Nunito' rel='stylesheet' type
='text/css'>
<link href='http://fonts.googleapis.com/css?family=Dancing+Script' rel='styleshe
et' type='text/css'>

<link rel="stylesheet" href="reset.css">
<link rel="stylesheet" href="dt.css">
<link rel="stylesheet" href="about.css">
<link rel="stylesheet" href="sidebar.css">
<link rel="stylesheet" href="awards.css">

<script src="http://code.jquery.com/jquery-latest.min.js"></script>
<script src="https://courses.oreillyschool.com/html5_responsive_design/software/
jquery.slides.min.js"></script>
<script src="dt.js"></script>
<script src="awards.js"></script>

</head>
```

Don't bother previewing yet, because we still need to add the JavaScript and CSS to make the slide show work.

To use SlidesJS, add some images to an element,then call the **slidesjs** method on that element. The SlidesJS library takes all those images and turns them into a slide show. We still haven't actually added any images to the "about-awards" element of the page yet; that's because we're going to load two different snippets of HTML depending on the width of the browser when the page is loaded. One snippet of HTML will link to large images meant for the wide view, while the other snippet will link to smaller images meant for the narrow view. This way, we load the images that work best for the view we're acccessing. As we discussed earlier, one of the issues with mobile devices is that even though they have high pixel density screens capable of displaying beautiful, large images, we need to be concerned about download speeds. Because of that, loading smaller images is often a better choice so users don't have to wait too long for their pages to load on mobile devices when using a slow network.

We need to create two HTML files, one for the large images, **slides.html** and one for the small images, **slides_640.html**. Create a new file slides.html and add the following HTML:

CODE TO TYPE:

```
<h2>Awards</h2>
<p id="slides">
    <img src="http://courses.oreillyschool.com/html5_responsive_design/images/Al
aska_Sunset.jpg" alt="Alaska Sunset">
    <img src="http://courses.oreillyschool.com/html5_responsive_design/images/Ic
eland_Ponies.jpg" alt="Iceland Ponies">
    <img src="http://courses.oreillyschool.com/html5_responsive_design/images/Ic
eland_GullfossWaterfall.jpg" alt="Waterfall in Iceland">
    <img src="http://courses.oreillyschool.com/html5_responsive_design/images/Al
aska_Whales.jpg" alt="Whales in Alaska">
</p>
```

- **slides.html**.

Create a new file slides_640.html and add the following HTML:

CODE TO TYPE:

```
<h2>Awards</h2>
<p id="slides">
    <img src="http://courses.oreillyschool.com/html5_responsive_design/images/Al
aska_Sunset_640.jpg" alt="Alaska Sunset">
    <img src="http://courses.oreillyschool.com/html5_responsive_design/images/Ic
eland_Ponies_640.jpg" alt="Iceland Ponies">
    <img src="http://courses.oreillyschool.com/html5_responsive_design/images/Ic
eland_GullfossWaterfall_640.jpg" alt="Waterfall in Iceland">
    <img src="http://courses.oreillyschool.com/html5_responsive_design/images/Al
aska_Whales_640.jpg" alt="Whales in Alaska">
</p>
```

- **slides_640.html**. The only difference in these two files is that one loads the large set of images and the other loads the small set of images.

Now that we have the HTML for the "about-awards" set up in two files, we need a bit of code to tell the page which snippet of HTML to load. Again, we'll use jQuery for this, as it has a simple **load()** method we can use to load HTML into an element. You've already created a link to **awards.js** in your HTML, so now you'll create that file and add the jQuery to load the HTML snippet based on the width of the page. Create a new file awards.js and add the following JavaScript:

CODE TO TYPE:

```
$(document).ready(function() {
    console.log($(window).width());
    if ($(window).width() > 960) {
        $("#about-awards").load("slides.html", function() {
            console.log("loading big slides");
            slidesLoaded();
        });
    } else {
        $("#about-awards").load("slides_640.html", function() {
            console.log("loading 640 slides");
            slidesLoaded();
        });
    }
});

function slidesLoaded() {
    $("#slides").slidesjs({
        width: 640, height: 480,
        navigation: false
    });
}
```

- **awards.js**. Don't preview yet; we still need the CSS! Let's go over the code real quick first. We're adding

another ready function (remember, jQuery allows you to specify multiple ready functions, so this is just fine). We check to see if the width of the page is greater than 960 pixels; if it is, we load the HTML snippet from **slides.html** which will cause the big images to load. If the width of the page is less than or equal to 960 pixels we load the HTML snippet from **slide_640.html**, which will cause the small images to load.

But why are we using 960 pixels? Don't we want to use the bigger images unless the browser is in the narrow view, at 640 pixels or less? Well, remember that the awards slide show appears in the primary content of the page which is 60% the width of the entire page. We're also going to add CSS in a new file **awards.css** to set the maximum width of the slide show element (the with the id "slides") to 90% (which is 90% of the width of the primary content, which is 60% the width of the page). In other words, we don't need the big images to load unless the page is wider than 960 pixels. If we load the bigger images and the page is, say, 800 pixels, then the primary content will be 480 pixels (60% of 800), and the slide show element will be 432 pixels. So the big images will need to be scaled down to fit into that size anyway. We might as well load the smaller images to reduce download time if we aren't going to display the images at a bigger size.

Back to the JavaScript: notice that we're using the jQuery **load()** method to load the snippet directly into the "about-awards" . The first argument to the **load()** method is the name of the file containing the HTML we want to load, and the second argument is the function we call once that load has completed. Once the HTML is loaded from the file into the "about-awards" , we call the function **slidesLoaded()**. This is where the SlidesJS magic happens. We select the element with the id "slides" (the <p> element we loaded from the file), and call the **slidesjs()** method, passing an object with three properties: the width, the height, and whether we want navigation (we don't; we're going to use the SlidesJS pagination instead, which is automatically set up for us, as you'll see in a moment). So that's it! SlidesJS does the rest for us, inserting the right elements with the right classes into our HTML. The width and the height we pass in are just the initial values; we'll be creating CSS so that the slide show will stretch and shrink as you expand and narrow your browser.

Next we need to add some style so that the slides and the pagination elements look good and are responsive in our page. First, take a look at the HTML and CSS that's generated by SlidesJS and inserted into the page with the **load()** function (this is something you can look at using the **Elements** tab in your Chrome browser console, or analogous tabs in other browsers—try it!):

OBSERVE:

```
<div id="about-awards">
<h2>Awards</h2>
<p id="slides" style="overflow: hidden; display: block;">
  <div class="slidesjs-container" style="overflow: hidden; position: relative; w
idth: 540px; height: 405px;">
    <div class="slidesjs-control" style="position: relative; left: 0px; width: 5
40px; height: 405px;">
      <img src="https://courses.oreillyschool.com/html5_responsive_design/images
/Alaska_Sunset_640.jpg" alt="Alaska Sunset" class="slidesjs-slide" slidesjs-inde
x="0" style="position: absolute; top: 0px; left: 0px; width: 100%; z-index: 0; -
webkit-backface-visibility: hidden; display: none;">
      <img src="https://courses.oreillyschool.com/html5_responsive_design/images
/Iceland_Ponies_640.jpg" alt="Iceland Ponies" class="slidesjs-slide" slidesjs-in
dex="1" style="position: absolute; top: 0px; left: -540px; width: 100%; z-index:
 0; display: block; -webkit-backface-visibility: hidden;">
      <img src="https://courses.oreillyschool.com/html5_responsive_design/images
/Iceland_GullfossWaterfall_640.jpg" alt="Waterfall in Iceland" class="slidesjs-s
lide" slidesjs-index="2" style="position: absolute; top: 0px; left: 0px; width:
100%; z-index: 10; display: block; -webkit-backface-visibility: hidden;">
      <img src="https://courses.oreillyschool.com/html5_responsive_design/images
/Alaska_Whales_640.jpg" alt="Whales in Alaska" class="slidesjs-slide" slidesjs-i
ndex="3" style="position: absolute; top: 0px; left: 540px; width: 100%; z-index:
 0; display: block; -webkit-backface-visibility: hidden;">
    </div>
  </div>
  <ul class="slidesjs-pagination">
    <li class="slidesjs-pagination-item"><a href="#" data-slidesjs-item="0" clas
s="">1</a></li>
    <li class="slidesjs-pagination-item"><a href="#" data-slidesjs-item="1" clas
s="">2</a></li>
    <li class="slidesjs-pagination-item"><a href="#" data-slidesjs-item="2" clas
s="active">3</a></li>
    <li class="slidesjs-pagination-item"><a href="#" data-slidesjs-item="3" clas
s="">4</a></li>
  </ul>
</p>
</div>
```

Notice that SlidesJS has moved the images in the snippet file inside two elements, and added a bunch of inline CSS. The SlidesJS JavaScript will use this extra structure as well as the CSS to implement the slide show functionality. SlidesJS also added a whole new <ul> element, containing the pagination for the slides: these will be displayed as little dots underneath the award image so you can navigate the images in the slide show. We're keeping the navigation minimal for the awards since most users will not care about the details of the awards beyond seeing that the company has won awards (which hopefully makes the customer feel good about going on a tour with Dandelion Tours!).

Now that you've seen the resulting HTML, the CSS you're going to add next will make a whole lot more sense. Create a new file awards.css and add the following CSS:

CODE TO TYPE:

```
#slides {
    display: none;
}

.slidesjs-container {
    overflow: hidden;
    position: relative;
}

.slidesjs-pagination {
    margin: 6px 0px 0px 0px;
    float: right;
    list-style: none;
}

.slidesjs-pagination li {
    display: inline;
    margin: 0px 1px;
}

.slidesjs-pagination li a {
    display: inline-block;
    width: 13px;
    height: 0px;
    padding-top: 13px;
    background-image: url("https://courses.oreillyschool.com/html5_responsive_de
sign/images/pagination.png");
    background-position: 0px 0px;
    overflow: hidden;
}

.slidesjs-pagination li a.active, .slidesjs-pagination li a:hover.active {
    background-position: 0px -13px
}

.slidesjs-pagination li a:hover {
    background-position: 0px -26px
}

#slides {
    margin: 0 auto 0 0;
    max-width: 90%;
}
#slides img {
    width: 100%;
}
@media only screen and (max-width: 640px) {
    #about-awards {
        background-color: black;
        color: white;
    }
    #slides {
        margin: 0 auto;
    }
}
```

awards.css. Now that all the pieces are there, go ahead and your **about.html** file to check out your handiwork! Scroll down to the awards section and take a look at the slide show. You should see this in the wide view:

Awards

Change the width of your browser to the narrow view and see how the awards section changes. If you have a mobile device, try loading the page and using the slide show. You can swipe left and right to get to the next and previous slides.

Let's go over the CSS so you can see how it works. First, and this might seem a bit odd, but we set the display of the "slides" to none:

OBSERVE:

```
#slides {
    display: none;
}
```

This is recommended by SlidesJS so that while the images are loading the slides element doesn't "flicker." The SlidesJS JavaScript later shows the element, so this works fine. The next part of the CSS is all taken from the SlidesJS example file, with just a few minor modifications:

OBSERVE:

```
.slidesjs-container {
    overflow: hidden;
    position: relative;
}
.slidesjs-pagination {
    margin: 6px 0px 0px 0px;
    float: right;
    list-style: none;
}
.slidesjs-pagination li {
    display: inline;
    margin: 0px 1px;
}
.slidesjs-pagination li a {
    display: inline-block;
    width: 13px;
    height: 0px;
    padding-top: 13px;
    background-image: url("https://courses.oreillyschool.com/html5_responsive_de
sign/images/pagination.png");
    background-position: 0px 0px;
    overflow: hidden;
}
.slidesjs-pagination li a.active, .slidesjs-pagination li a:hover.active {
    background-position: 0px -13px
}
.slidesjs-pagination li a:hover {
    background-position: 0px -26px
}
```

Pay attention to the rules for the **slidesjs-pagination** class. These rules are for the <ul> element with the class **slidesjs-pagination** that SlidesJS adds to the HTML. Look back at the HTML. The list has four items in it: one for each image that we loaded into the slide show, and in each list item is an <a> element that we use to create a **link to each image** loaded into the slide show.:

OBSERVE:

```
  <ul class="slidesjs-pagination">
    <li class="slidesjs-pagination-item"><a href="#" data-slidesjs-item="0" clas
s="">1</a></li>
    <li class="slidesjs-pagination-item"><a href="#" data-slidesjs-item="1" clas
s="">2</a></li>
    <li class="slidesjs-pagination-item"><a href="#" data-slidesjs-item="2" clas
s="active">3</a></li>
    <li class="slidesjs-pagination-item"><a href="#" data-slidesjs-item="3" clas
s="">4</a></li>
  </ul>
```

Look at the rule for the links in the list items within the **slidesjs-pagination** :

OBSERVE:

```
.slidesjs-pagination li a {
    display: inline-block;
    width: 13px;
    height: 0px;
    padding-top: 13px;
    background-image: url("https://courses.oreillyschool.com/html5_responsive_de
sign/images/pagination.png");
    background-position: 0px 0px;
    overflow: hidden;
}
```

Look at the CSS; we load **one png image** for the pagination items: **pagination.png**. Go ahead and load

that image into your browser window, with this URL: https://courses.oreillyschool.com/html5_responsive_design/images/pagination.png. You'll see this image:

If you go to your page, you'll see that the pagination item corresponding with the currently loaded image is black, while the pagination items for the other images are white. Hover your cursor over another pagination item and you'll see it turn a dark red color. These are the three parts of the png image shown above shown in different states of the links in the list.

In this example (which is taken straight from the SlidesJS examples, with only a few minor modifications), we use a single image for all three modes of the pagination items. This is called a **CSS Sprite**. We'll come back and cover this in more detail in just a moment. For now, just note that each <li> element in the "slidesjs-pagination" element corresponds to one pagination item below the slide show.

Finally, in the awards CSS, we've added three of our own rules to make sure the slide show is fully responsive:

OBSERVE:

```
#slides {
    margin: 0 auto 0 0;
    max-width: 90%;
}
#slides img {
    width: 100%;
}
@media only screen and (max-width: 640px) {
    #about-awards {
        background-color: black;
        color: white;
    }
    #slides {
        margin: 0 auto;
    }
}
```

The first rule, for the element with the id "slides," sets the max-width to 90%, and the right margin to auto. This ensures that the element can grow and shrink with the size of the browser, but will never exceed 90% of the width of the "about-awards" in the primary content area: the "about" <section>. In the next rule we make sure **all the images within the slide show are displayed at a width of 100%**, again ensuring that the images can shrink or grow with the width of the "slides" element. Finally, we **set up a media query to make a few changes when the browser is in the narrow view**. We change the background color of the "about-awards" element to black and the text color to white, and set both the left and right margin of the "slides" element to auto, meaning that the slide show will appear in the center of the page, rather than aligned to the left as it is in the wide view. And that's it!

You might notice that the pagination items in the narrow view are a bit harder to see than in the wide view, because they are displayed on a black background. We could have made another image to use (similar to **pagination.png**, only with lighter colors), however, because it's likely that this will be seen primarily on mobile devices when users don't need to use the pagination items (because they can swipe instead), we decided to leave it as is.

You might have noticed that if you're in wide mode and switch to narrow mode in your browser, the browser doesn't load the small images instead of the big ones, and if you load the page in the narrow view and then expand the width of your browser to the wide view, the browser doesn't load the bigger images. This is actually a *good* thing. Why? Well, if you start out in the wide view, and switch to the narrow view, it doesn't hurt to use the bigger images in the narrow view. There's no real reason to load the smaller images in this case. The big images are already there, so let's use them! The other scenario, where we start out in the narrow view with the smaller images and then expand the width of the browser, is a lot less likely to happen because the user who opens the page in the narrow view is probably on a mobile device and doesn't have the option to expand the browser to a wide view. If you did want the browser to load the big images when the user expands the page, how you might do that?

How CSS Sprites Work

Let's come back to that CSS Sprite. A CSS Sprite describes the technique of combining multiple small images into one image, and then using CSS to determine which part of the image is displayed on the page at any given time. You might wonder if it might be better to have a few small images rather than one big one. It isn't. For small images, like the ones we're using to implement the pagination items in the slide show, the request to retrieve the image from the server is larger and takes longer than downloading the image itself. So making three requests to get three tiny images is less efficient than making one request for a bigger image.

In our CSS for the <a> element in each of the <li> elements within the "slidesjs-pagination" , we're setting the background image of the link to the **pagination.png** image. The browser downloads the image *once* and reuses that same downloaded image for each of the links (so the browser doesn't need to download the image four different times).

For the default state of the link, the image is placed so that the top of the image is flush with the top of the element, and the element is 13 pixels in height, which is the height of one circle. The image is the background image, so we position it with the **background-position** property. The overflow is hidden, so the part of the image that extends beyond the size of the element (which is 13 pixels by 13 pixels) does not show.

When you click on a pagination item, the class "active" is added to the link (you can see this if you look back up at the HTML above for the SlidesJS code: you'll see the third image, which is the one I selected, has the class "active" in the corresponding third pagination item). That means the rule:

OBSERVE:

```
.slidesjs-pagination li a.active, .slidesjs-pagination li a:hover.active {
    background-position: 0px -13px
}
```

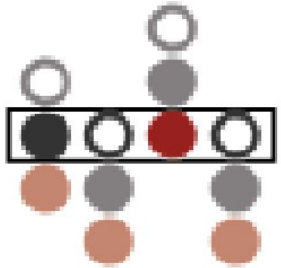
...is used to style the link as well. This changes the background-position of the image to -13px on the y-axis, and moves the image up 13 pixels. The link is still 13 pixels by 13 pixels, so we see only the 13 pixels in the middle of the image, which is the black circle.

If you hover over a pagination item, then the <a> element is in the hover state, and so this rule is used:

OBSERVE:

```
.slidesjs-pagination li a:hover {
    background-position: 0px -26px
}
```

This changes the background position to -26px on the y-axis, moving the image up another 13 pixels, or 26 pixels from its original position. It shows the bottom 13 pixels of the image, which is the red circle. So, if I've selected the first image, and I'm hovering over the pagination item for the third image, the pagination items will look like this:



Here, I'm showing the position of the **pagination.png** image for each pagination item, and outlining the parts of the images you see in each of the links in the list. The semi-opaque parts of the images are still there in the page, but they are hidden (because of the **overflow: hidden** property we used).

This is a CSS Sprite! It's simply a small image that gets shifted around using CSS (or sometimes JavaScript and CSS) depending on what state your elements are in and which part of the image you need to display. CSS Sprites are commonly used in responsive web design, because by combining a few (or many!) small images into one bigger image, we can make the download of the assets required for the page more efficient, which makes the experience of using the web page better.

In this lesson, we accomplished a lot: we added a whole new page to the site; created two different layouts for the page —one for the wide view and one for the narrow view; we added a responsive slide show element to the page, and we

learned how to use CSS Sprites. Nice work!

Responsive Forms

Lesson Objectives

When you complete this lesson, you will be able to:

- create forms for responsive websites.
- use HTML5 input **types** to improve the user experience.
- style forms to work on both wide and narrow (mobile) screens.
- use CSS to provide visual feedback to the user about required form fields and invalid input data.
- use JavaScript to validate form data.

Designing with the User in Mind

Earlier in the course, we said that we use responsive web design to create a website that can dynamically change depending on the screen size of the device being used to view it. In addition to size, there are other elements we can alter to make a website more usable on both the small and large screens that will make certain actions easier for the user, and also make sure the website provides clear feedback when the user makes a mistake.

From Wikipedia: "User understanding of the content of a website often depends on user understanding of how the website works. This is part of the user experience design."

This course isn't focused on user experience design, but in this lesson, we'll look at how user experience design often overlaps with responsive design. In general, you'll always want to consider the usability of your website along with how responsive it is. There's no point in making a responsive website if it makes no sense to people!

In this lesson, we're going to build the "Contact Us" page for Dandelion Tours. This page has a form that users can use to request more information about tours for a destination. We'll look at how we can structure form elements to work responsively, and how to communicate clearly with users so they enter the correct information in the form before submitting it.

Creating the Contact Us Page

Let's begin building the Contact Us page and form with the HTML. As always, we want to get the content and structure created correctly first, and then style it and add the behavior we want. The Contact Us page will consist of a form on the primary part of the page, on the left, and the sidebar on the right (following the design of the About Us page we did previously). You can start with the **about.html** file and make edits if you want; otherwise you can copy and paste the code below. We highlighted the changes from the About Us page. Create **contact.html** as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Contact Us</title>

<meta name="viewport" content="width=device-width, initial-scale=1, user-scalabl
e=no">

<link href='http://fonts.googleapis.com/css?family=Nunito' rel='stylesheet' type
='text/css'>
<link href='http://fonts.googleapis.com/css?family=Dancing+Script' rel='styleshe
et' type='text/css'>

<link rel="stylesheet" href="reset.css">
<link rel="stylesheet" href="dt.css">
<link rel="stylesheet" href="contact.css">
<link rel="stylesheet" href="sidebar.css">

<script src="http://code.jquery.com/jquery-latest.min.js"></script>
<script src="dt.js"></script>
<script src="contact.js"></script>

</head>

<body>

<header>
    <div class="full">
        <a href="index.html" class="name">Dandelion Tours</a>
    </div>
    <div class="mobile">
        <a href="index.html" class="name">DT</a>
    </div>
</header>
<nav>
    <div class="menu mobile">
        <button id="menuButton">Menu</button>
    </div>
    <ul class="menu full">
        <li><a href="travel.html">Tours</a></li>
        <li><a href="gallery.html">Gallery</a></li>
        <li><a href="about.html">About</a></li>
        <li><a href="contact.html" class="selected">Contact</a></li>
    </ul>
    <div class="search mobile">
        <button id="searchButton">Search</button>
    </div>
    <div class="search full">
        <form><input type="search" id="searchInput" placeholder="search"></form>
    </div>
</nav>
<section id="contact">
    <header>
        <h1>Contact Us!</h1>
    </header>
    <div class="contact-welcome">
        <p>
            Thank you for choosing Dandelion Tours!
        </p>
    </div>
    <div class="contact-form">
    <form>
        <div class="section" id="trips">
            <h2>Which tour are you interested in?</h2>
            <div id="trips-buttons">
```

```
            <button data-img="https://courses.oreillyschool.com/html5_responsive
_design/images/Alaska_Sunset_op.jpg">Alaska</button>
            <button data-img="https://courses.oreillyschool.com/html5_responsive
_design/images/Iceland_Road_op.jpg">Iceland</button>
            <button data-img="https://courses.oreillyschool.com/html5_responsive
_design/images/Brazil_Butterfly_op.jpg">Brazil</button>
            </div>
        </div>
        <div class="section" id="dates">
            <h2>When are you interested in going?</h2>
            <p>Between:</p>
            <input id="start-date" type="date" min="2015-03-01" max="2016-03-01"
>
            <input id="end-date" type="date" min="2016-03-01" max="2017-12-31">
        </div>
        <div class="section" id="info">
            <h2>Your information</h2>
            <input id="first" type="text" placeholder="First name" required>
            <input id="last" type="text" placeholder="Last name" required>
            <input id="email" type="email" placeholder="Email address" required>
            <input id="phone" type="tel" placeholder="Telephone number" required
>
        </div>
        <div id="message"></div>
        <button id="submit">Submit</button>
    </form>
    </div>
</section>
<section id="sidebar">
    <header>
        <h1>Tour with us.</h1>
        <h2>Change your life.</h2>
        <p>
            At <span class="name">Dandelion Tours</span> you'll have the trip of
 a lifetime.
        </p>
    </header>
    <div>
        <ul>
            <li>
                <h1>Visit spectacular places</h1>
                <p>
                    With us, you can visit places you might never go on your own
.
                    Out-of-the-way locations that most tourists never get to see
.
                </p>
            </li>
            <li>
                <h1>Comfort and options for everyone</h1>
                <p>
                    You'll have a comfortable place to sleep each night, and
                    a wide range of options for activities during the day.
                </p>
            </li>
            <li>
                <h1>Knowledgable tour guides</h1>
                <p>
                    Your tour guides will be able to answer all your questions
                    about the location and sights you see.
                </p>
            </li>
        </ul>
    </div>
    <footer>
        <h1>Testimonials</h1>
        <p><span>Dan says:</span> "I had such a wonderful vacation. I got a chan
ce
```

```
to relax and learn at the same time. Everything was taken care of; all I had to
do was enjoy myself! Thank you Dandelion Tours."</p>
        <p><span>Julie says:</span> "It was the trip of a lifetime. I will
remember this for as long as I live."</p>
        <p><span>Chris says:</span> "We had an absolutely amazing holiday. Every

step of the way was handled perfectly, from the booking process, to the document
s
and information sent prior to travel. Everything was so well organized while we
were
in Iceland, enabling us to relax and fully enjoy our adventure. </p>
    </footer>
</section>
<footer>
    <div>
        Have a question? Email us at
        <a href="mailto:example@example.com">travel@dandeliontours.com</a>
    </div>
    <div>
        <a class="social" href="#">Facebook</a>
        <a class="social" href="#">Twitter</a>
        <a class="social" href="#">Instagram</a>
        <a class="social" href="#">Youtube</a>
    </div>
    <div>
        You'll travel with the wind. <span class="name">Dandelion Tours</span>.
    </div>
</footer>

</body>
</html>
```

and . The form appears in the page with sections to choose a trip, specify preferred dates for the trip, enter personal information, and of course, a **Submit** button to submit the form to the server (although we won't implement the server side for this course).

Take a closer look at the form. At the top part of the form, where the user will select a tour, we're using three **buttons**, one for each tour destination. We use a **data-*** attribute to specify the image to use in the background of the button, which we'll set programmatically. For now the buttons look just like regular form buttons, but we'll change the style of these buttons using CSS.

OBSERVE:

```
    <form>
        <div class="section" id="trips">
            <h2>Which trip are you interested in?</h2>
            <div id="trips-buttons">
            <button data-img="https://courses.oreillyschool.com/html5_responsive
_design/images/Alaska_Sunset_op.jpg">Alaska</button>
            <button data-img="https://courses.oreillyschool.com/html5_responsive
_design/images/Iceland_Road_op.jpg">Iceland</button>
            <button data-img="https://courses.oreillyschool.com/html5_responsive
_design/images/Brazil_Butterfly_op.jpg">Brazil</button>
            </div>
        </div>
```

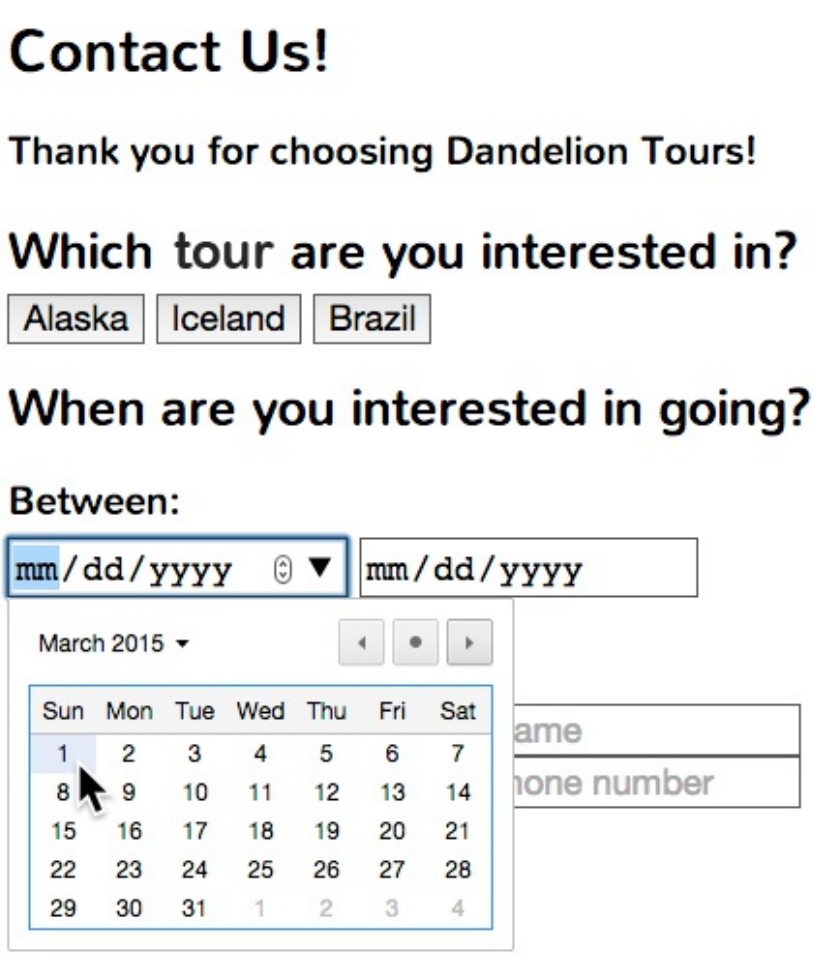
For the part of the form where the user will select dates for their trip, we use two <input> elements, both of **type date**. Using the correct **type** for <input> elements can help create a better user experience, and is especially important on mobile. Before HTML5, this type did not exist, but now we can use this type to create a pop-up calendar in the browser, which makes choosing a date a lot easier!

Note The pop-up calendar works in most modern browsers. If it's not supported, the field works like a text field.

OBSERVE:

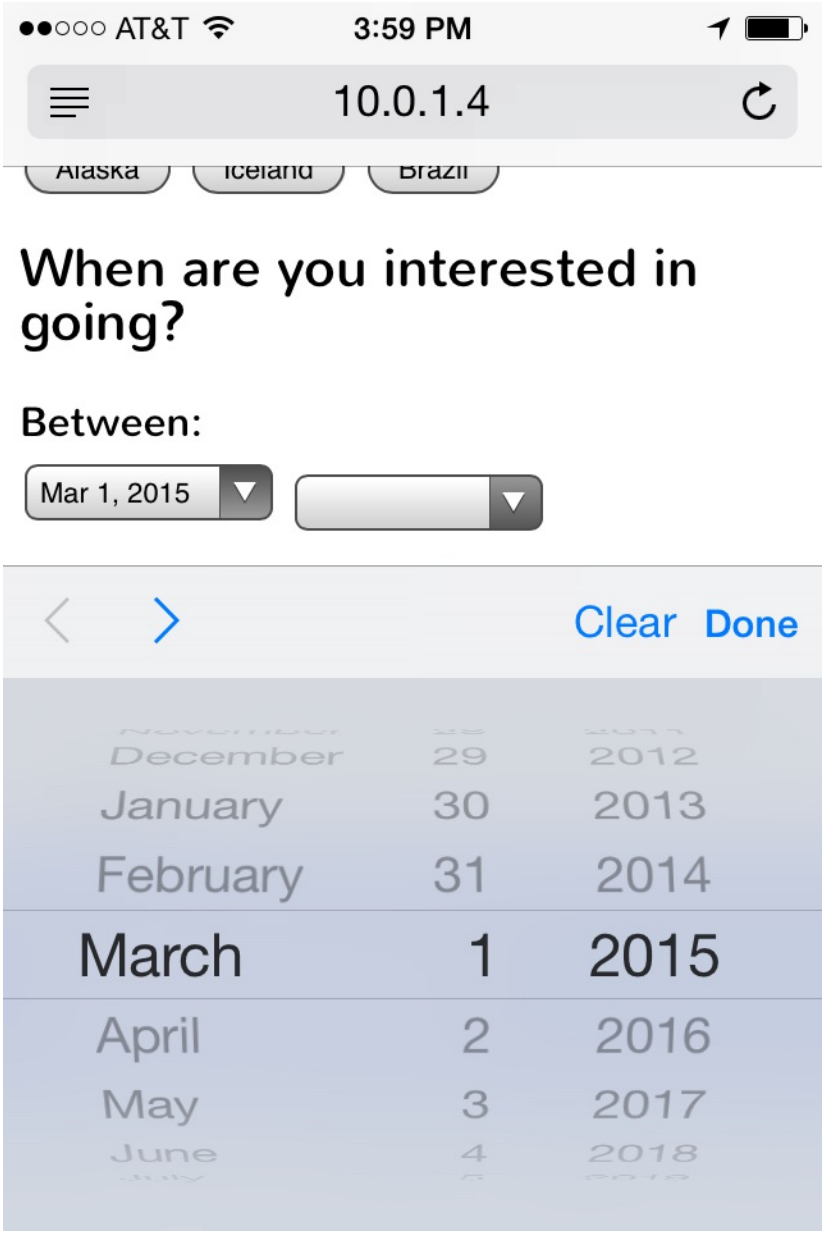
```
        <div class="section" id="dates">
            <h2>When are you interested in going?</h2>>
            <p>Between:</p>
            <input id="start-date" type="date" min="2015-03-01" max="2016-03-01"
>
            <input id="end-date" type="date" min="2016-03-01" max="2017-12-31">
        </div>
```

This is the pop-up calendar you'll see in Chrome:



Note As of this writing, Internet Explorer does not supply a pop-up calendar for a **date** input type. However, you can edit the dates by modifying the text in the input fields; as long as the format of each date is valid, you'll be able to submit the form without an error message (for more on the valid format for dates, see the validation we're doing below).

On mobile devices, the **date** type creates a date rolodex, again making choosing date a lot easier on the user. Here's the date rolodex on the iPhone 4s:



Without the **date** type (for instance, if we used type **text** as we would have done in the past before HTML5), the user would be responsible for entering the date as text and using the correct format for that date. So the **date** type makes validating forms a lot easier on us developers, too.

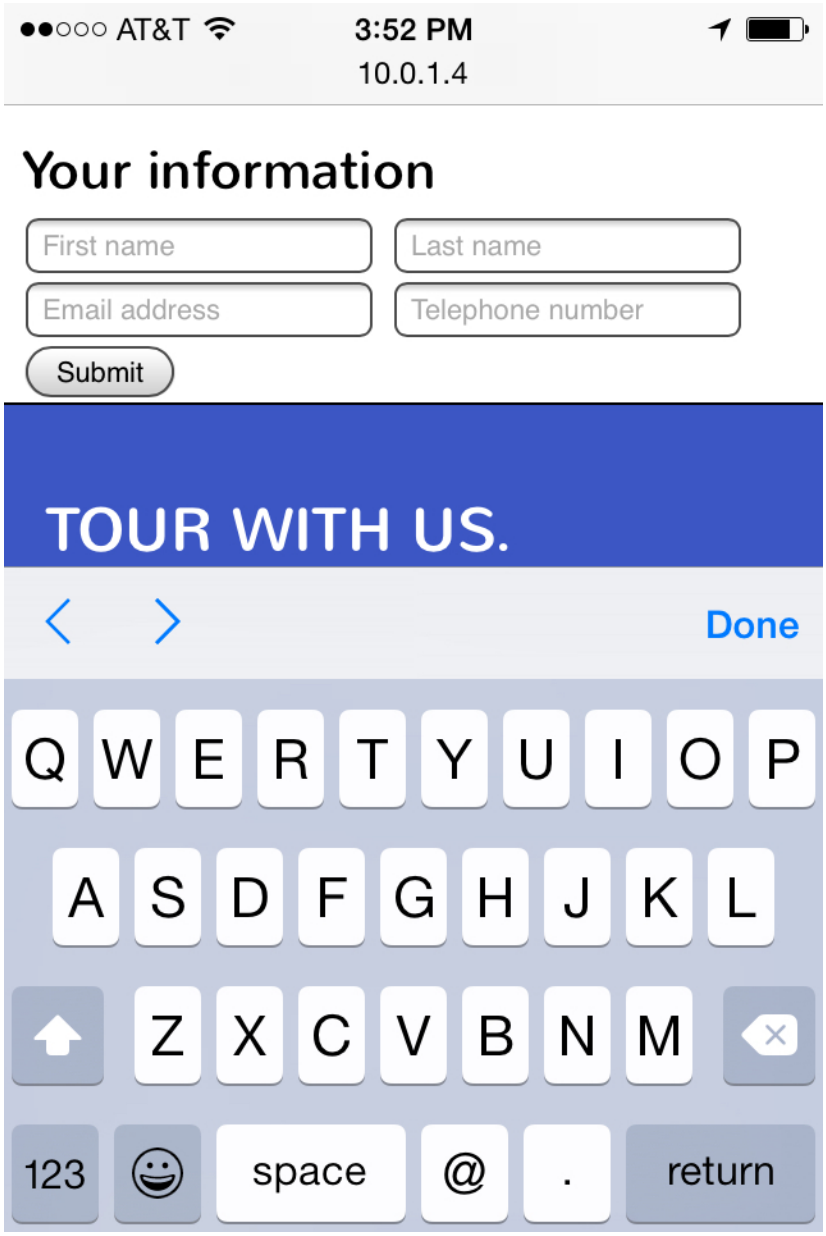
Lastly, for the part of the form where users enter their personal information, we've got four input fields, one each for the first name, last name, email, and phone. For the name fields, we just use the **text type**, but notice that for the **email** field, we use type **email**, and for the **phone** field, we use type **tel**:

OBSERVE:

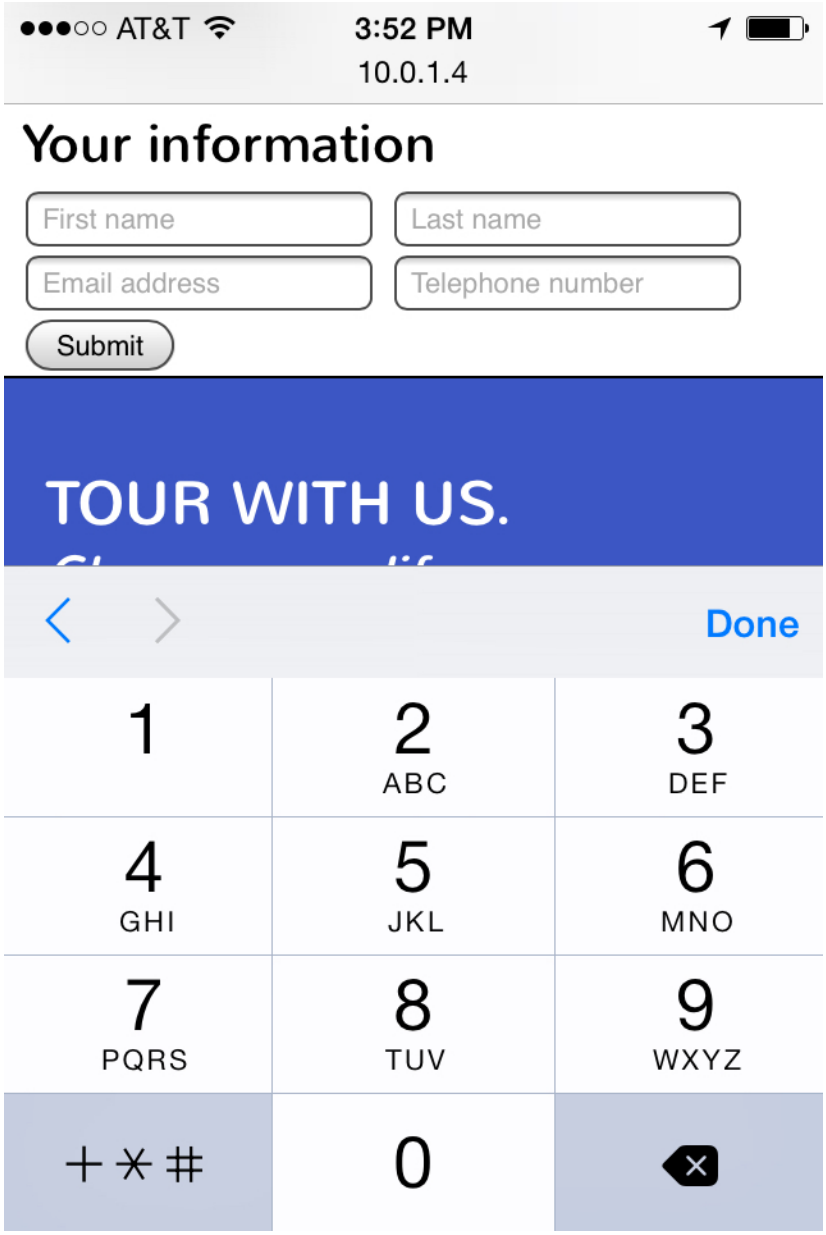
```
        <div class="section" id="info">
            <h2>Your information</h2>
            <input id="first" type="text" placeholder="First name" required>
            <input id="last" type="text" placeholder="Last name" required>
            <input id="email" type="email" placeholder="Email address" required>
            <input id="phone" type="tel" placeholder="Telephone number" required
>
        </div>
        <div id="message"></div>
        <button id="submit">Submit</button>
    </form>
```

On desktop computers, using these types doesn't make much difference, but on mobile, using these types makes a huge difference in terms of usability. Using the type **email** causes the keyboard to include an "@"

key (and sometimes a ".com" key, too), which can save a lot of time when entering an email address, since typically, the "@" key is one or two keyboards away from the main one:



Similarly, using the **tel** type causes the keyboard to change to a numeric keyboard for typing phone numbers:



Users definitely appreciate these special keyboards for both email addresses and phone numbers when entering data on a form.

We're using two other HTML5 features of forms that can make the user experience better: the **placeholder** and **required** attributes. The **placeholder** attribute allows you to specify text in an input field that the user sees until they click in the field and start typing. This is usually a hint to the user about what's expected in the field. We're using these placeholders to indicate that we want the user's first and last names in the first two fields, and their email address and telephone number in the third and fourth fields, respectively. We're not using labels for these fields (which may be an easier way to create the form), but with placeholders the user knows what to enter in these fields even without the labels.

The **required** attribute is a boolean attribute, with no value. Just putting the attribute name there indicates that the field is required in order for the user to submit the form. By default, the browser will tell the user if they've forgotten to enter a value in a required field when they click the Submit button:

This is handy but it's not a substitute for validating form data. We're going to go beyond this very basic validation for our page, but we'll still use the **required** attribute to determine the state of an input element. With the **required** attribute, using CSS only, we can specify a style to use if the user hasn't yet entered a value into a field. We'll do our own validation of the content the user enters into the field using JavaScript.

Finally, we've got a "message" that we'll use to display validation error messages to the user, and a <button> to submit the form.

Before we leave the form, take a quick look at the <meta> tag at the very top of the document, where we specify **user-scalable=no** in the content:

OBSERVE:

```
<meta name="viewport" content="width=device-width, initial-scale=1, user-scalable=no">
```

That code is there to ensure that when the user clicks on a form input element in the mobile version of the form, the page doesn't scale up. By default, if the font-size isn't big enough in the form, the page will scale up to make it easier for the user to see the form. This is okay, except that the page doesn't scale back down once the user has finished working with a given form element. We're going to increase the font-size of our form elements for the narrow view of the page, which means no scaling will be necessary: the user should be able to see the form just fine at the font-size we'll be using.

Styling the Form

First, let's get the basic style in so the form works with the sidebar in the page (just like we did with the primary content and sidebar in the About Us page). Remember, the primary content and sidebar are displayed using **table display** in the wide view of the page, so we'll use a rule similar to the one we used in **about.html** to use **table-cell** display, and set a **width** of **60%** (we're re-using **sidebar.css** for the sidebar so we don't need to write any rules for that section). And again, just like we did with the About Us page in the narrow view, for the Contact Us page in the narrow view, we don't use **table-cell** display and instead use the default order of the two sections ("contact" and "sidebar") in the structure of the page. We also add a little padding to space everything a bit. Create a new file, contact.css, and add the following CSS:

CODE TO TYPE:

```
section {
    box-sizing: border-box;
    border-top: 1px solid black;
}
@media only screen and (min-width: 641px) {
    section#contact {
        display: table-cell;
        width: 60%;
        padding: .8em 3% .8em 2%;
    }
}
@media only screen and (max-width: 640px) {
    section#contact {
        padding: .8em 3% 0 2%;
    }
}
```

it as **contact.css** and your **contact.html** file. You now see the sidebar beside the primary content in the wide view, and below the primary content in the narrow view.

Now let's style the buttons in the first part of the form. Eventually, we're going to add images in the background, but we'll do that with code. For now, we'll modify the text and corners, add some spacing, and add rules for how the buttons should look when you hover your cursor over the button, or tap on a button with your finger (on a mobile device). Add the following CSS to contact.css:

CODE TO TYPE:

```
...
button {
    outline: none;
    border-radius: 5px;
}
button#submit {
    margin-top: 5px;
}
div#trips-buttons button {
    border: 2px solid transparent;
    background-size: cover;
    width: 30%;
    text-shadow: 0px 0px 3px gray;
    cursor: pointer;
    padding: 15px;
}
div#trips-buttons button:hover, div#trips-buttons button:active {
    color: white;
    text-shadow: 0px 0px 5px black;
}
```

contact.css and your **contact.html** file. Check out the buttons in the top part of the form. They're bigger now, with nice rounded corners, and the text has a text shadow behind it, which is going to make the text easier to see when we put images in the background of the buttons. If you hover over a button, you'll see the text turn white. By setting the **outline** property to **none**, we ensure that when you click on a button, it doesn't get the blue border that the browser adds by default.

Next, we're going to add some style for the input fields, and also add a media query with a rule to increase the size of the input fields and buttons when we're in the narrow view. Update contact.css and add the following CSS:

CODE TO TYPE:

```
...
input, button {
    outline: none;
    border-radius: 5px;
}
...
div#info input {
    margin-bottom: 5px;
    padding: 5px;
    width: 90%;
}
div#info input:focus {
    background-color: rgba(157, 206, 255, .5);
}
@media only screen and (min-width: 1000px) {
    div#info input {
        width: 50%;
    }
}
@media only screen and (max-width: 640px) {
    input, button {
        font-size: 1.2em;
        margin-bottom: 12px;
    }
}
```

contact.css and your **contact.html** file. Change the width of the browser from wide to very wide, and then to narrow. Pay attention to what happens to the input fields. When the browser is in the wide view, the input fields take up about 90% of the width of the left part of the page. When the browser is very wide, then the width of the input fields switches to being 50% as wide as the left part of the page. This is so the input fields don't get too long (which doesn't look very good). Finally, when the browser is narrow, the sidebar moves under the form, and the input fields take up 90% of the width of the page.

With the pseudo element **:focus**, we can style an <input> element when it's in focus (that is, you've tabbed into it, or clicked the mouse in it, or tapped on it). We've set the background color to light blue so the user knows which input element is the current one.

We've also added a rule to increase the font-size and bottom margin of both the <input> and <button> elements when in the narrow view. This increases the size of the inputs and buttons and makes them much easier for users to tap on with their fingers.

Handling the Style for Required Elements

It's usually a good idea to indicate visually which field values in a form are required in order to submit that form. In our case, *all* the fields are required. We're going to be initializing the dates in the two date fields programmatically, so if the user doesn't change those values to something else, we'll still have default values for the dates. However, for the tour selection, and the personal information required from the user, we need to show the user that they need to enter values in order to submit the form. We'll use a red asterisk (*) to indicate required values—this is something many users are already familiar with because red asterisks have been used on the web for many years to indicate a field value in a form is required.

Thanks to recent additions to CSS, we can add the red asterisks using CSS! Using CSS instead of adding an element to the HTML to indicate required fields is preferred because the asterisk for this use is considered style, not content, so it's best if we can avoid adding it to the HTML. We're going to use two different tricks, one for the heading above the tour selection buttons, and one for the four <input> elements in the personal information part of the form. Update contact.css as shown:

CODE TO TYPE:

```
...
div#trips > h2::after {
    content: url("https://courses.oreillyschool.com/html5_responsive_design/images/required_sm.png");
}
div#trips > h2.done::after {
    content: "";
}
div#info input:required:invalid, div#info input:focus:invalid {
    background-image: url("https://courses.oreillyschool.com/html5_responsive_design/images/required_sm.png");
    background-position: right top;
    background-repeat: no-repeat;
    background-size: 15px;
}
```

contact.css and your **contact.html** file. You see a red asterisk beside the <h2> heading, "Which tour are you interested in?" as well as in the four <input> elements under "Your information." For the heading, we're using the **::after** pseudo-element (CSS also has a **::before** pseudo-element).

The **::after** pseudo-element is typically used to add "cosmetic content" to an element (see MDN), which is exactly what we want to do here. We select the <h2> element in the with the id "trips," and specify some **content** to add after the content that's already in the <h2> element. The content we want to add is an image—a red asterisk (PNG file)—so we specify the URL of the image, and the image is added to the <h2>. If you use the Elements inspector tool in the browser, you'll see **::after** in the HTML, corresponding to the PNG that appears after the heading.

We want to remove this image once the user has successfully selected a tour, and we'll do that by adding the class **done** to the heading (with code). When we do that, the rule:

OBSERVE:

```
div#trips > h2.done::after {
    content: "";
}
```

...will match the heading, and sets the **::after** content of the heading back to nothing, removing the image.

That functionality doesn't work yet, because we don't have a way to select the tour, or add the **done** class to the heading, but we'll get there shortly when we get to the code, and revisit these two style rules so you can

see how they work.

The second trick we use to add the red asterisk to the page with CSS is a rule to add the asterisk to the background of the <input> elements if the state of those elements is invalid:

OBSERVE:

```
div#info input:required:invalid, div#info input:focus:invalid {
    background-image: url("https://courses.oreillyschool.com/html5_responsive_design/images/required_sm.png");
    background-position: right top;
    background-repeat: no-repeat;
    background-size: 15px;
}
```

The **:required** pseudo-class selects an element with the **required** attribute (all four of the <input> elements in the "info" have this attribute); the **:focus** pseudo-class selects an element that has received focus. The **:invalid** pseudo-class selects an element with content that fails to validate based on the type attribute of that element. The only input element of the four that helps determine "validity" in this case is the input with the type attribute **email**. We'll do additional validation in code and display a message to the user in the "message" if the any of the values they enter is invalid. However, using the red asterisk in this way is a good indicator to the user which fields are required.

Try the form now: you can enter any text in the first and last name fields, and the phone field, and the red asterisk will go away. But the red asterisk will not go away for the email field unless you type something like "a@b". Obviously this isn't actually a valid email address, but it's one more level of validation that the browser offers us now.

Initializing the Form and Adding Handlers

We've got the content, structure, and style of the page in place, so now it's time to write some JavaScript. First, we'll write the code to initialize the form and handle the tour buttons; then we'll deal with validating the form data when the user clicks (or taps) the Submit button.

There are two steps we need to take to initialize the form: set dates in the two date fields and add images to the background of each of the tour buttons.

To initialize the dates in the date fields, we'll use the **min** and **max** values of the "start-date" and "end-date" <input> elements, respectively. Look at the HTML for these two elements; we defined the **min** and **max** values for both date fields. This limits how far backward and forward the user can go in the calendar. Take a look at contact.html:

OBSERVE:

```
<input id="start-date" type="date" min="2015-03-01" max="2016-03-01">
<input id="end-date" type="date" min="2016-03-01" max="2017-12-31">
```

We'll use the **min** attribute value from the "start-date" to initialize the starting date field, and use the **max** attribute value from the "end-date" to initialize the ending date field.

Then we'll use the **data-img** attribute of the tour buttons to initialize the background image for each button. Before we do that though, we're going to *preload* all the images we need for these buttons. For each button, we need two images: one for the unselected state and one for the selected state. The unselected state uses a slightly opaque version of the image, while the selected state uses a regular version of the image. By preloading the images, we make the response time of the buttons a little quicker when the user goes to select a button (it doesn't really help with the unselected state images since we use those right away, but by preloading the regular state images, the buttons should switch images faster when the user selects a tour). To preload images, all we have to do is create an **Image** object for each image we want to preload.

We've already added a link to a JavaScript file **contact.js** in the HTML, so go ahead and create that new file now, and add the JavaScript below:

CODE TO TYPE:

```
$(function() {

    preloadButtonImages();

    var $startDate = $("input#start-date");
    $startDate.val($startDate.attr("min"));

    var $endDate = $("input#end-date");
    $endDate.val($endDate.attr("max"));

    $("div#trips-buttons button").each(function() {
        var img = $(this).data("img");
        $(this).css("background-image", "url(" + img + ")");
    });
    $("div#trips-buttons button").click(function(e) {
        $("div#trips > h2").addClass("done");
        $("div#trips-buttons button").not(this).each(function() {
            var img = $(this).data("img");
            var isSelected = $(this).hasClass("selected");
            if (isSelected) {
                $(this).removeClass("selected").css("background-image", "url(" +
 img.replace("_sm", "_op") + ")");
            }
        });
        var img = $(this).data("img");
        var isSelected = $(this).hasClass("selected");
        if (isSelected) {
            $(this).removeClass("selected").css("background-image", "url(" + img
.replace("_sm", "_op") + ")");
        } else {
            $(this).addClass("selected").css("background-image", "url(" + img.re
place("_op", "_sm") + ")");
        }
        e.preventDefault();
    });

});

function preloadButtonImages() {
    $("div#trips-buttons button").each(function() {
        var img = $(this).data("img");
        (new Image()).src = img;
        (new Image()).src = img.replace("_sm", "_op");
    });
}
```

contact.js and your **contact.html** file. You should see that the date fields now have initial values. Try changing the preferred start or end date for your tour. When you pop up the calendar, the date that we initialized in the date field is selected in the calendar, which makes it easier for you to choose dates.

You should also see that each button in the tour section of the form has a background image now, and that both of the date fields are initialized to dates. Try selecting a tour by clicking on a button. Now, select another tour. When one button changes to the selected mode, the other button changes back to the unselected mode, but the text is not staying white in the button when it's selected. We can fix that by adding a bit more CSS for the **selected** class we're using to track whether or not a button is selected. Update contact.css and add the following CSS:

CODE TO TYPE:

```
...
div#trips-buttons button.selected {
    border: 2px inset gray;
    color: white;
    text-shadow: 0px 0px 5px black;
}
```

contact.css and your **contact.html** file. Now the buttons look a little better when they're selected: the text on the selected button stays white, and the inset border makes the button look depressed into the page, which accentuates the selected look.

Also notice that the red asterisk on the heading about the tour buttons disappears as soon as you select a tour! Let's take a closer look at the click handler for the buttons so we can see how that's done, and also how we're using the **selected** class on the buttons to determine which button is selected:

OBSERVE:

```
$(function() {
    ...
    $("div#trips-buttons button").click(function(e) {
        $("div#trips > h2").addClass("done");
        ...
    });
});
```

To remove the asterisk on the heading, all we have to do is **add the "done" class to the heading**. The CSS takes care of the rest. Remember that we **wrote a rule** that matches the <h2> heading with the "done" class and removes the **content** (the red asterisk image) that we added to the heading using the **::after** pseudo-element in the **previous rule**:

OBSERVE:

```
div#trips > h2::after {
    content: url("../images/required_sm.png");
}
div#trips > h2.done::after {
    content: "";
}
```

What about the buttons? Take a look at the code again:

OBSERVE:

```
$(function() {
    ...
    $("div#trips-buttons button").click(function(e) {
        $("div#trips > h2").addClass("done");
        $("div#trips-buttons button").not(this).each(function() {
            var img = $(this).data("img");
            var isSelected = $(this).hasClass("selected");
            if (isSelected) {
                $(this).removeClass("selected").css("background-image", "url(" +
 img.replace("_sm", "_op") + ")");
            }
        });
        var img = $(this).data("img");
        var isSelected = $(this).hasClass("selected");
        if (isSelected) {
            $(this).removeClass("selected").css("background-image", "url(" + img
.replace("_sm", "_op") + ")");
        } else {
            $(this).addClass("selected").css("background-image", "url(" + img.re
place("_op", "_sm") + ")");
        }
        e.preventDefault();
    });
});
```

Here, we first **look at every button that is *not* the one we just clicked on** and if a button is selected, we remove the "selected" class and change the background image from the selected version to the opaque version of the image. The selected images have the string "_sm" in them, while the opaque images have the string "_op" in them. For instance, the selected Alaska image is Alaska_Sunset_sm.jpg, while the unselected

Alaska image is Alaska_Sunset_op.jpg. We're using the **replace** method to simply replace the "_sm" part of the string with "_op" to make this switch.

Then, we **look at the button we clicked on** to select a tour. If the button is currently selected, and we click on it, we unselect the button. If it's not currently selected, then we select it. We do this by adding or removing the "selected" class and switching out the background image.

Finally, we call **e.preventDefault**. The **e** object is the event object that is passed into the click handler. We call this to prevent the browser from displaying the popup indicating that the user hasn't filled out the "Your information" fields yet (since we're using the red asterisk to indicate that). Try commenting out this line of code, reload the page, and see what happens when you click on a tour button. What do you think of the browser's default behavior? I decided I found it annoying (especially since the tour buttons are not submit buttons!), which is why I added the call to the **preventDefault** method. However, you might choose to leave this line commented out if you like the additional reminder to the user that they still need to fill in the rest of the form.

Do you notice any bugs? What if you select a tour and then deselect it? Look at the red asterisk on the heading. What do you think the behavior should be here? (You'll get to fix this bug as an exercise!)

Validating the Form

We've got the basic behavior of the form in place, so now it's time to add some validation code. We'll write a separate **validateForm** function to handle this, and add this as the click handler for the "submit" button. Modify contact.js as shown:

CODE TO TYPE:

```
$(function() {
    ...
    $("div#trips-buttons button").each(function() {
        ...
    });

    $("button#submit").click(validateForm);
});
...
```

The validation code in **validateForm** will create messages to the user indicating whether they've missed a field in the form or entered a value that doesn't validate (for instance, a date in the wrong format, or an invalid phone number). Of course, there's only so much we can do with JavaScript to ensure that the user entered correct data, but doing some rudimentary checking on the client side (that is, in the browser) before the form gets submitted to the server is helpful to the user because submitting a form usually means waiting for the server to return a new page. If the user's made an error, this can be frustrating and time consuming. So, doing as much checking as we can in the browser makes the process seem faster and more responsive, and provides a better overall user experience.

There's nothing particularly special about the code below, so we'll go through it fairly quickly. Go ahead and add it to your **contact.js** file and give it a try. Edit contact.js and add the following JavaScript:

CODE TO TYPE:

```
function validateForm(e) {
    e.preventDefault();

    // validate a trip button has been selected
    var $trip = $("div#trips-buttons button.selected");
    if ($trip.length === 0) {
        $("#message").html("Please select a trip");
        return;
    }

    // validate date
    var startDate = $("input#start-date").val();
    var endDate = $("input#end-date").val();

    if (!startDate) {
        $("#message").html("Please enter valid start date");
        return;
    }
    if (!endDate) {
        $("#message").html("Please enter valid end date");
        return;
    }

    if (startDate && endDate) {
        // regular expression for date
        var dateReg = /^\d{4}-\d{2}-\d{2}$/;
        if (!(startDate.match(dateReg) && endDate.match(dateReg))) {
            $("#message").html("Please enter valid start and end dates");
            return;
        }
    }

    // validate names
    var first = $("input#first").val();
    var last = $("input#last").val();
    if (!first) {
        $("#message").html("Please enter a first name");
        return;
    }
    if (!last) {
        $("#message").html("Please enter a last name");
        return;
    }

    // validate email
    var email = $("input#email").val();
    if (!email) {
        $("#message").html("Please enter an email address");
        return;
    } else {
        var filter = /^\s*[\w\-\+_]+(\.[\w\-\+_]+)*\@[\w\-\+_]+\.[\w\-\+_]+(\.[\
w\-\+_]+)*\s*$/;
        if (!email.match(filter)) {
            $("#message").html("Please enter a valid email address");
            return;
        }
    }

    // validate telephone number
    var phone = $("input#phone").val();
    if (!phone) {
        $("#message").html("Please enter a phone number");
        return;
    } else {
        var filter = /^(\+\d{1,2}\s)?\(?\d{3}\)?[\s.-]?\d{3}[\s.-]?\d{4}$/;
        if (!phone.match(filter)) {
```

```
            $("#message").html("Please enter a valid phone number with area code
");
            return;
        }
    }

    // all is well, empty the message!
    $("#message").html("");
}
```

Before you try the code, also make a quick update to your **contact.css** file to add a style for the "message" , to show the text in red. This will make it easier for the user to see if they've made an error. Update contact.css and add the following CSS:

CODE TO TYPE:

```
...
div#message {
    color: red;
}
```

contact.css and your **contact.html** file. Try entering both correct and incorrect values in the form fields and clicking the **Submit** button. Try leaving some fields blank. Try not selecting a tour button. Make sure your form is behaving as you'd expect.

We go through each part of the form and make sure it has a value, and that the value is "valid" (at least, as far as we can tell). For the tour buttons, that's not too hard: we just make sure that one of the buttons is selected. If none are selected, we display a message in the "message" and then **return** so that none of the other validation runs (we'll present error messages to the user one at a time, starting with the top of the form).

If the user has correctly selected a tour, then we continue validating the rest of the form, using a similar strategy. To check the dates, we first check to see if there's a value in each field, and if there is, we use a regular expression to match the date using the expected format. Since we're using a **date** type for the <input> elements, this will be a lot easier on the user; unless they make an effort to go in and change the format of the date, the date *should* be in the correct format.

Likewise, for the name, email, and phone fields, we check for values, and in the case of email and phone, attempt to validate the format. In each case, if the user forgets to enter a value, or uses an invalid format, we update the "message" and return. We don't have to worry about removing the red asterisk on these fields; as soon as the user types a value into the field, the red asterisk goes away. That doesn't mean the value they used is "valid" though, so we need the separate "message" field to give them more information about what's wrong.

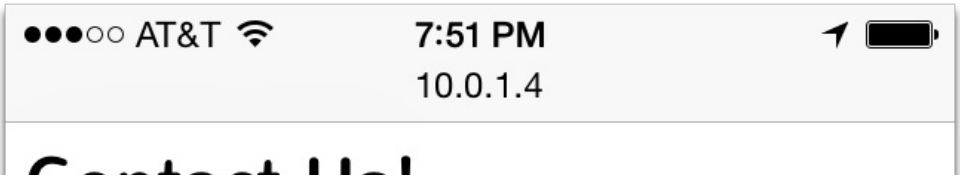
If we get through the entire form with no problems, then we update the "message" to display nothing (to erase any previous error message that may have been displayed). At this point, if you had a server-side program to submit the form to, and you'd specified that in the **action** attribute of the form, you'd call:

OBSERVE:

```
$("div.contact-form form").submit();
```

We don't though, so we leave this line off. However, we are again calling **e.preventDefault** (at the top of the function). Not only does this prevent default browser popups on empty fields in the "Your information" section, it also prevents the browser from submitting the form to *nowhere* (since we haven't specified an action on the form) and emptying out the form. This allows us to implement the validation before *we* submit the form by calling the **submit** method, as shown above, which we'd do in a real-world situation.

If you have a mobile device, test the form on that too, and see what you think. Here's what it looks like on my iPhone 4s:



Contact Us!

Thank you for choosing Dandelion Tours! More here.

Which tour are you interested in?*

When are you interested in going?

Between:

Mar 1, 2015

Dec 31, 2017

Your information

First name *

Last name *

Email address *

Telephone number *

Submit

In this lesson, we created a responsive form, and learned how to use the correct input types for various kinds of fields, like date, email, and phone. Using the correct input types can make for a better user experience, especially on mobile. We also learned how to style the form, validate the form data after the user submits the form, and provide helpful feedback to the user through visual queues and a validation message.

Take some time to practice by doing the project, and then we'll move on to add some video to Dandelion Tours!

Responsive Video

Lesson Objectives

When you complete this lesson, you will be able to:

- add a video to your web page using the <video> HTML element.
- add a video to your web page from a service like Vimeo using an <iframe>.
- style video in your web page using CSS.
- use attributes to control various options in your video player.
- describe formats used to encode video for the web.

Video on the Web

Video has been part of the web for a long time now, but up until fairly recently, most of that video was served as Flash video, requiring a Flash plugin. Now we have what we call *native* video, which really just means video that the browser knows how to play without a plugin like Flash. Unfortunately, just because we have native video doesn't mean that video is easy. It's still a pretty complex topic. However, it has become a bit easier to add short videos to your web pages, and it's become a whole lot easier to embed videos, from sites like YouTube and Vimeo (among many others), in your web pages.

In this lesson, we'll take a brief look at video for the web. We'll talk about some of the issues you need to think about when adding video to your site, and we'll take a look at the <video> HTML element as well as how to embed videos from a third-party site (in this example, Vimeo). Of course, we'll make sure the web page containing the videos is responsive and works well on both the large and small screen.

Video

The <video> element was added to HTML to work in a similar way to the <img> element: that is, you provide a source URL for the video, which is downloaded from that URL and played. Videos are much more complex than images, so the <video> element provides lots of options for doing things like showing video controls (play, pause, volume, and such.), muting and looping the video, and controlling how video is loaded and played.

Let's dive right in and get the <video> element added to a web page, along with three source URLs for a video in three different video formats. For this part of the course, we'll create a new file, **gallery.html**, and link this page into the rest of the Dandelion Tours website at the Gallery link.

Note

We'll use public-domain, copyright-free and royalty-free videos obtained from archive.org. For your own pages for this course, you may use videos downloaded from archive.org, or use videos you create yourself, in one of the three main formats we'll discuss: mpeg, ogg, or webm. You'll need to know how to create and format videos correctly if you use your own, so if you don't know how, don't worry about it; just use videos you find on archive.org, which are all public domain, free to use videos. Create a new file as shown below:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Gallery</title>

<meta name="viewport" content="width=device-width, initial-scale=1, user-scalabl
e=no">

<link href='http://fonts.googleapis.com/css?family=Nunito' rel='stylesheet' type
='text/css'>
<link href='http://fonts.googleapis.com/css?family=Dancing+Script' rel='styleshe
et' type='text/css'>

<link rel="stylesheet" href="reset.css">
<link rel="stylesheet" href="dt.css">
<link rel="stylesheet" href="gallery.css">
<link rel="stylesheet" href="sidebar.css">

<script src="http://code.jquery.com/jquery-latest.min.js"></script>
<script src="dt.js"></script>

</head>

<body>

<header>
    <div class="full">
        <a href="index.html" class="name">Dandelion Tours</a>
    </div>
    <div class="mobile">
        <a href="index.html" class="name">DT</a>
    </div>
</header>
<nav>
    <div class="menu mobile">
        <button id="menuButton">Menu</button>
    </div>
    <ul class="menu full">
        <li><a href="travel.html">Tours</a></li>
        <li><a href="gallery.html" class="selected">Gallery</a></li>
        <li><a href="about.html">About</a></li>
        <li><a href="contact.html">Contact</a></li>
    </ul>
    <div class="search mobile">
        <button id="searchButton">Search</button>
    </div>
    <div class="search full">
        <form><input type="search" id="searchInput" placeholder="search"></form>
    </div>
</nav>
<section id="gallery">
    <header>
        <h1>Preview your tour</h1>
    </header>
    <h2>Video from our last Alaska tour: Moose encounter!</h2>
    <div class="gallery-video">
        <video controls>
            <source src="https://courses.oreillyschool.com/html5_responsive_desi
gn/videos/moose.mp4" type="video/mp4">
            <source src="https://courses.oreillyschool.com/html5_responsive_desi
gn/videos/moose.ogg" type="video/ogg">
            <source src="https://courses.oreillyschool.com/html5_responsive_desi
gn/videos/moose.webm" type="video/webm">
            Your browser does not support the <code>video</code> element.
            <img src="https://courses.oreillyschool.com/html5_responsive_design/
```

```
images/moose.jpg">
        </video>
    </div>
</section>
<section id="sidebar">
    <header>
        <h1>Tour with us.</h1>
        <h2>Change your life.</h2>
        <p>
            At <span class="name">Dandelion Tours</span> you'll have the trip of
 a lifetime.
        </p>
    </header>
    <div>
        <ul>
            <li>
                <h1>Visit spectacular places</h1>
                <p>
                    With us, you can visit places you might never go on your own
.
                    Out-of-the-way locations that most tourists never get to see
.
                </p>
            </li>
            <li>
                <h1>Comfort and options for everyone</h1>
                <p>
                    You'll have a comfortable place to sleep each night, and
                    a wide range of options for activities during the day.
                </p>
            </li>
            <li>
                <h1>Knowledgable tour guides</h1>
                <p>
                    Your tour guides will be able to answer all your questions
                    about the location and sights you see.
                </p>
            </li>
        </ul>
    </div>
    <footer>
        <h1>Testimonials</h1>
        <p><span>Dan says:</span> "I had such a wonderful vacation. I got a chan
ce
to relax and learn at the same time. Everything was taken care of; all I had to
do was enjoy myself! Thank you Dandelion Tours."</p>
        <p><span>Julie says:</span> "It was the trip of a lifetime. I will
remember this for as long as I live."</p>
        <p><span>Chris says:</span> "We had an absolutely amazing holiday. Every

step of the way was handled perfectly, from the booking process, to the document
s
and information sent prior to travel. Everything was so well organized while we
were
in Iceland, enabling us to relax and fully enjoy our adventure. </p>
    </footer>
</section>
<footer>
    <div>
        Have a question? Email us at
        <a href="mailto:example@example.com">travel@dandeliontours.com</a>
    </div>
    <div>
        <a class="social" href="#">Facebook</a>
        <a class="social" href="#">Twitter</a>
        <a class="social" href="#">Instagram</a>
        <a class="social" href="#">Youtube</a>
    </div>
```

```
    <div>
        You'll travel with the wind. <span class="name">Dandelion Tours</span>.
    </div>
</footer>

</body>
</html>
```

it as **gallery.html** and . A video appears in your page, with controls at the bottom for playing and pausing, scrubbing (moving forward and backward within the video), controlling the audio level, and opening the video fullscreen. Try clicking play on the video and make sure you can see the video play and hear the audio.

Note If you're having problems with the video, make sure you're using the most recent version of the browser of your choice or try a different browser. Each browser supports different video formats.

Let's take a close look at the <video> element we used to insert the video into the page:

OBSERVE:

```
<video controls>
    <source src="https://courses.oreillyschool.com/html5_responsive_design/videos/moose.mp4" type="video/mp4">
    <source src="https://courses.oreillyschool.com/html5_responsive_design/videos/moose.ogg" type="video/ogg">
    <source src="https://courses.oreillyschool.com/html5_responsive_design/videos/moose.webm" type="video/webm">
    Your browser does not support the <code>video</code> element.
    <img src="https://courses.oreillyschool.com/html5_responsive_design/images/moose.jpg">
</video>
```

Notice that the <video> element as used here is more similar to the <picture> element we explored earlier than it is to the <img> element. The people who designed this element recognized that we'd have similar problems as we have with the <img> element if we allow only one source file for the video, so they designed the <video> element to allow multiple sources from the beginning. This was a smart move, as each browser supports different video formats, and there are many competing video formats out there, so it's unlikely that we'll converge on a single format anytime in the near future (if ever). So, rather than specifying the video file we want with one **src** attribute, we specify the various video file options using the <source> element, nested within the <video> element. Only one of the video files will actually play; the browser tries each one, top down, until it reaches a format it knows how to play.

Note The <video> element also supports the **src** attribute, so if you have just one video file, you can specify it using this attribute. However, we don't recommend that unless you're targeting just one kind of browser (for example, if you're writing an application for just the iPhone Safari browser).

In our example, we have **three different video files** that could potentially be used as the source file to load the video into the browser: an mp4 file (which is the MPEG format), an ogg file (which is the Ogg Theora format), and a webm file (which is the Webm format). So how does the browser know which one to use? It checks the **type** of the file, and uses the first file (top down) that it supports. The type specified in the <source>'s **type** attribute is a mime type, similar to the mime types you may have used when specifying a link to a JavaScript file ("text/javascript") or CSS file ("text/css"). These mime types are standard across the internet, and you can get a comprehensive list of the types you can use for various kinds of files on Wikipedia.

If the browser you're using does not support the <video> element, as would be the case if you're using an older browser (older than say, 2011 or so), or a browser on some mobile devices, what you'll see is the **fallback text and an image of moose** instead. At this point, almost every browser supports the <video> element (to verify, see caniuse.com), so if you're still not seeing the video, chances are you're using a browser that doesn't know how to play any of the types we've provided (mp4, ogg, or webm), and again, if that's the case, try a different browser. There are other ways to provide a fallback that allows the user (in many cases) to see a video; for instance, if you need to, you can provide a Flash video fallback. We won't go into details on how to do this, however, since the <video> element is now widely supported.

Look at the opening tag of the <video> element, and notice that we specified a *boolean attribute*, **controls**. A boolean attribute is one that is true or false: if it's there, the element considers the attribute value to be true. In

this case, the **controls** attribute being present (and thus true) indicates that the video should be displayed with controls, like play/pause, volume, and so on. You should see those controls at the bottom of the video. Each browser will display these controls a little differently.

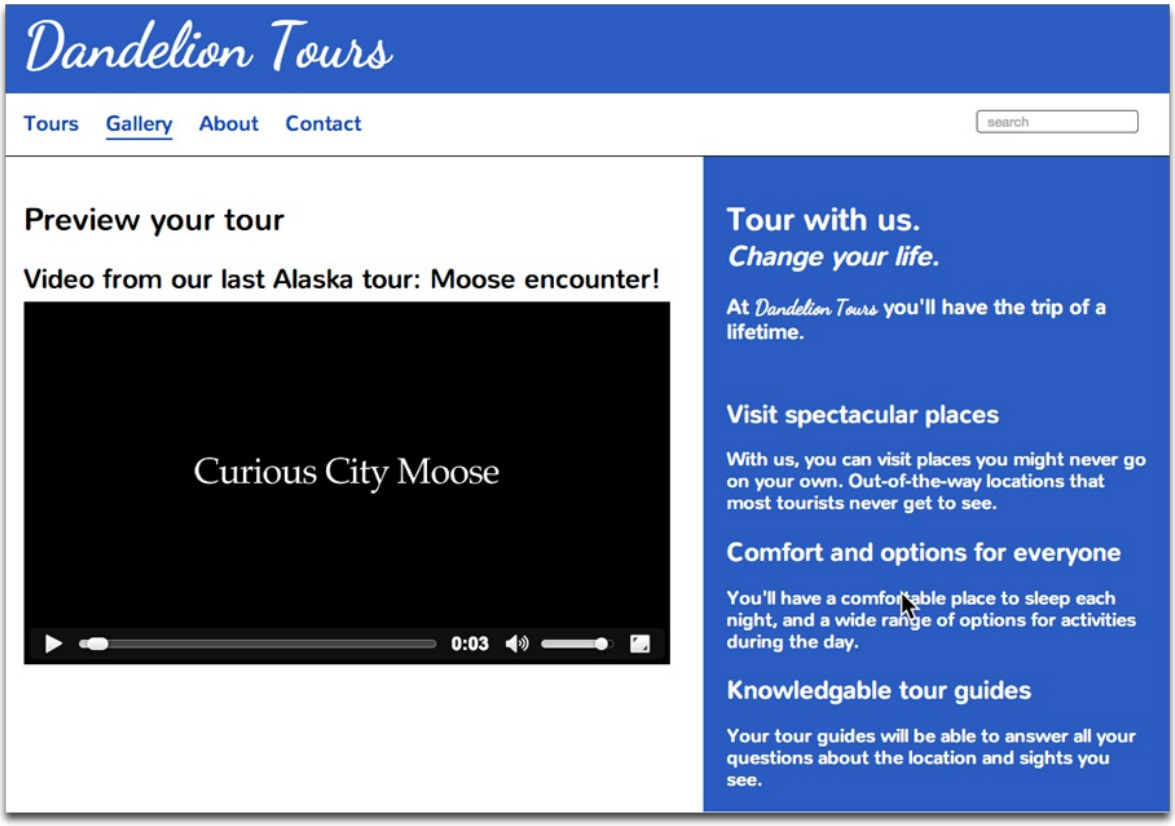
Typically, you'll want to provide the user with controls for video using the **controls** attribute, but there are a couple of cases when you won't. First, you may have a short video that you want to play in the background of a page when the page first loads. This is becoming a more common style (although it's probably a fad. Remember web pages that started playing music when they loaded?) in web pages, so you may have seen examples of this. Personally, I find it annoying, but clearly current web designers like using it for some designs. For us this is a use case for showing video without controls. (If you do this, just remember to mute the sound on the video, which we'll see how to do shortly.) Another use case for leaving off the **controls** attribute is if you're going to build your own controls, using HTML, CSS, and JavaScript.

You probably still have questions about the video formats, but before we dive into that, let's get the CSS for the page set up so it looks better. We'll use the same layout we've been using for the other pages (Tours, About, and Contact), so most of this will be familiar. Create a new CSS file and type the code shown below:

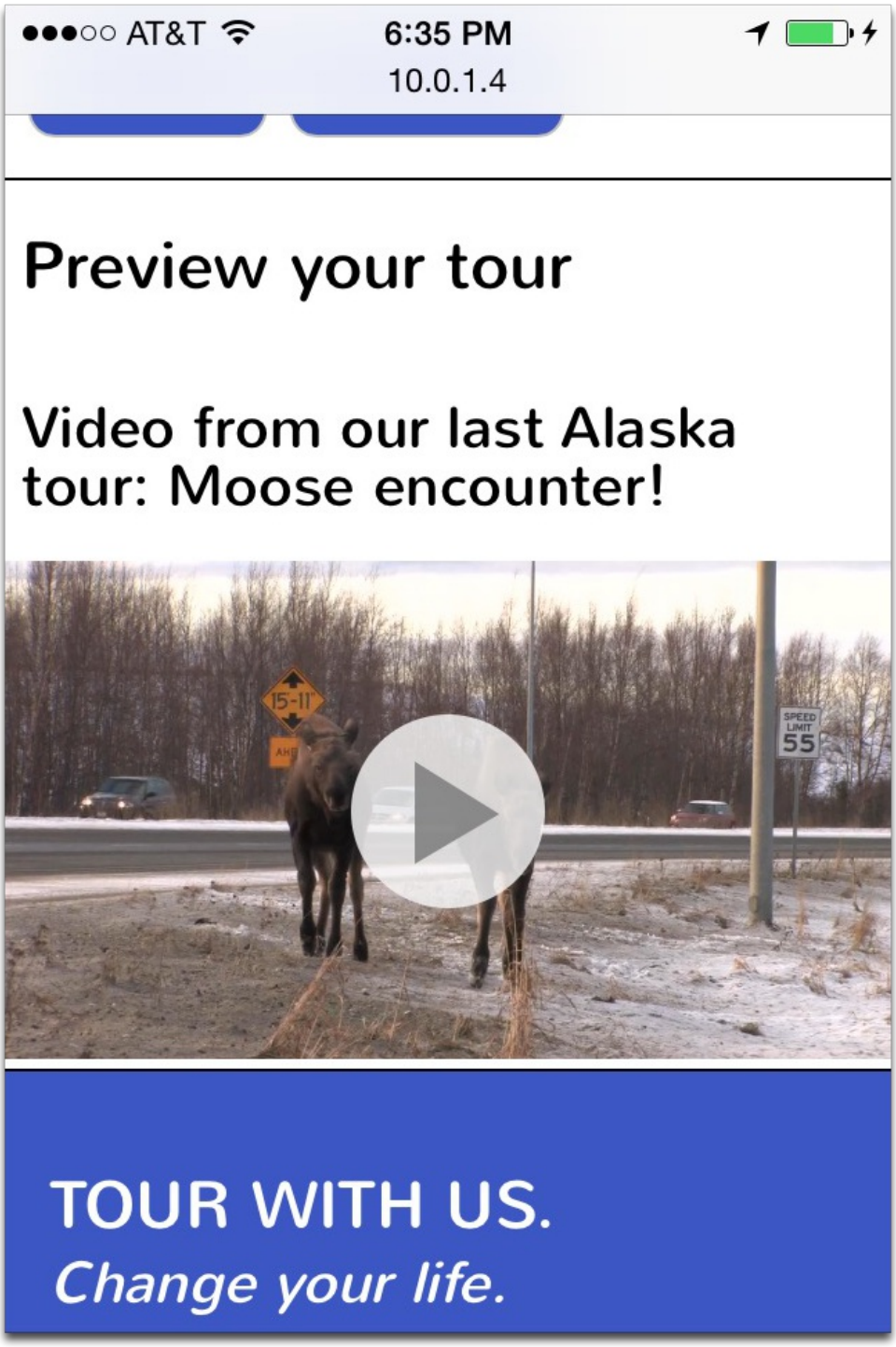
CODE TO TYPE:

```
section {
    box-sizing: border-box;
    border-top: 1px solid black;
}
@media only screen and (min-width: 641px) {
    section#gallery {
        display: table-cell;
        width: 60%;
        padding: .8em 3% .8em 2%;
    }
}
@media only screen and (max-width: 640px) {
    section#gallery h1, section#gallery h2, section#gallery p {
        padding: .8em 3% .8em 2%;
    }
}
div.gallery-video video, div.gallery-video img {
    width: 100%;
    height: auto;
}
```

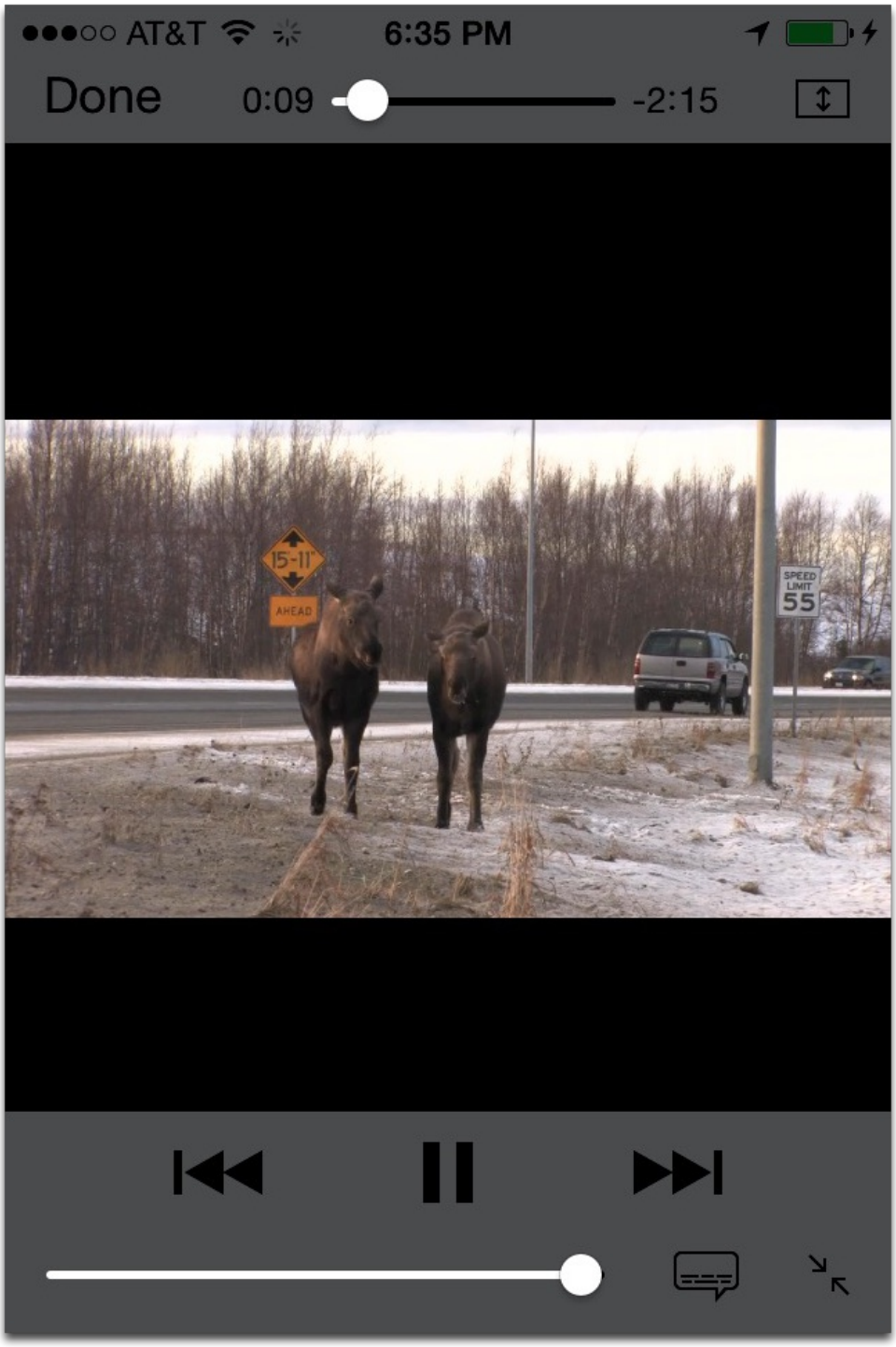
□ it as **gallery.css** and □ your **gallery.html** file. You've already linked the **gallery.css** file into your HTML in the previous step, so you see the page in two columns in the wide view, with the sidebar on the right, and the video in the column on the left (notice the controls on the bottom of the video):



If you make the browser more narrow, the video gets smaller (resizing with the browser width), until we switch into the narrow view at 640 pixels wide, when the video will take up the full width of the page, but the video works in the same way. On mobile, the video should look like the narrow view (depending on the screen size of your mobile device):



However, when you click to play the video, it will most likely open into full screen mode as it does on my iPhone:



We use table-cell display in the wide view (just like we've been doing), and simply allow the default layout in the narrow view (again, just like we've been doing). However, we **set the width of both the <video> element and the fallback <img> element** to make sure they expand to the width of the table-cell (which is 60% of the width of the page). We also **set the height to auto**, so the width and height stay proportional as the page expands and shrinks.

OBSERVE:

```
div.gallery-video video, div.gallery-video img {
    width: 100%;
    height: auto;
}
```

Video Formats

So, you probably noticed that we're using three different video formats. The video content for each of the URLs we provided is exactly the same, but the format of the video is different.

When we talk about video file formats, like mpeg or ogg or webm, we're actually referring to the *container* file. A container file for video is similar to a zip file: it's a file that wraps other files, grouping them together. For video, a container file wraps a video track (without audio), one or more audio tracks (without video), and possibly some additional tracks with metadata associated with the video.

When you load a video file into a player, the player is opening up the container file (if it knows how), and *decoding* the video and audio tracks inside in order to play them. The tracks are created with a particular *codec* (*codec* means "COde / DECode") which is the algorithm that was used to package up the video and audio information into tracks (series of images and sounds). So when the player decodes the video and audio tracks to play them, it's converting those tracks back into a series of images and sounds. A given video format (container) can often support a variety of different codecs—usually, different versions of one base algorithm. There are a huge number of codecs in use in the video world; for the web, the most popular codecs are H.264, VP8, and Theora. Different containers can contain tracks encoded with various codecs, but typically, for instance, an OGG container will contain tracks encoded using the Theora codec, and so on.

In addition to codecs for the video tracks, there are different codecs for audio tracks. As you can see, video and audio can get extremely complicated, and an in-depth treatment would require a whole course unto itself. Fortunately, to use video on the web, you don't usually need to know all those details (unless you're also *creating* the audio and/or video, in which case you'll need to dive into those details).

If you create your own audio and video, and you know which codecs are used to encode the tracks, you can specify this information in the **type** attribute of the <source> element in the <video> element:

OBSERVE:

```
<video controls>
    <source src="https://courses.oreillyschool.com/html5_responsive_design/videos/moose.mp4" type="video/mp4">
    <source src="https://courses.oreillyschool.com/html5_responsive_design/videos/moose.ogg" type="video/ogg; codecs=theora,vorbis">
    <source src="https://courses.oreillyschool.com/html5_responsive_design/videos/moose.webm" type="video/webm">
    Your browser does not support the <code>video</code> element.
    <img src="https://courses.oreillyschool.com/html5_responsive_design/images/moose.jpg">
</video>
```

The more information you provide to the browser, the easier it will be for the browser to determine which video file it can use to display the video. If you provide *no* information (for example, if you supply the URL for the filename only), the browser has to start downloading all three files and try to play them, which is a waste of bandwidth, particularly in a mobile environment where bandwidth is limited. So at the very least, provide the mime type in the **type** attribute to help the browser determine the type of the file.

For a slightly dated but terrific, more in-depth, look at video and audio, check this free online resource from **Dive into HTML5**: Video, by Mark Pilgrim. To see which browsers support which video formats, go to http://caniuse.com/#search=video, and scroll down: the page shows browser versions and which video formats are supported for each of the formats most commonly used on the web.

Getting More Control of Your Video

There are a variety of attributes you can provide on the <video> element. For a comprehensive list and discussion, check out the MDN page on <video>. We're going to use two other attributes in the Gallery page: one to start the moose video playing automatically after the page is loaded (**autoplay**) and the other to mute the audio volume, which is the polite thing to do when autoplaying video on a web page (**muted**). Edit gallery.html as shown:

CODE TO TYPE:

```
...
<video controls autoplay muted>
    <source src="https://courses.oreillyschool.com/html5_responsive_design/videos/moose.mp4" type="video/mp4">
    <source src="https://courses.oreillyschool.com/html5_responsive_design/videos/moose.ogg" type="video/ogg">
    <source src="https://courses.oreillyschool.com/html5_responsive_design/videos/moose.webm" type="video/webm">
    Your browser does not support the <code>video</code> element.
    <img src="https://courses.oreillyschool.com/html5_responsive_design/images/moose.jpg">
</video>
...
```

and . Both **autoplay** and **muted** are boolean attributes, so you don't need to provide values for them (like "muted=true"). Just supplying the attribute names is enough to cause them to take effect. So, when you load the **gallery.html** page now, the moose video plays automatically, but with no sound. The audio icon on the player shows that it's muted. If you click on this icon, the audio begins to play. If the autoplay drives you nuts (it might!), feel free to remove this attribute once you've tried it out.

Note If you turn off **autoplay**, you probably also want to turn off **muted**, since most users would expect the sound to be on when they click Play on an embedded video.

Using the <video> element attributes is one way to customize the video player a little bit; however, if you really want to customize it completely, you can, by providing your own controls and programming them yourself using JavaScript.

However, it's difficult to create responsive video with custom controls. On most mobile devices (as of this writing), the default video player takes over the screen and supercedes any controls you may have provided. So, in responsive design, it's best to rely on the built-in controls provided by the browser, so that's what we'll do here.

Embedding Video

If you want to add video to your web pages without having to worry about different file formats and codecs and all that gnarly stuff, it's now fairly easy to embed video from services like YouTube and Vimeo (and many others). For short videos, these services are usually free (unless you want professional-level options for lengthy videos and/or specific encoding parameters). The upside for using services like these is they handle storing, encoding, and streaming the video to your page, and provide players along with the video. If you choose a well-known video service like YouTube or Vimeo, chances are the delivery will be speedy, providing a good user experience. The downside is that these players are a lot less customizable than if you use the <video> element. For most use cases, however, these services work just fine.

One issue with many of these services is that you'll typically need to embed the video in your page using an <iframe> element (rather than using the <video> element). The <iframe> element is a container for HTML from another website, so it's like having a tiny web page inside your web page! That makes the video a little bit more difficult to style with CSS, but there are some tricks we can use, as you'll see shortly.

Note For more on the <iframe> element, check out the MDN iframe page.

For now, let's go ahead and add another video to the Gallery page using an <iframe>. For this part of the lesson, we've selected a video from Vimeo that has no audio, so as not to compete with our Moose video. Modify gallery.html as shown:

CODE TO TYPE:

```
<section id="gallery">
    <header>
        <h1>Preview your tour</h1>
    </header>
    <h2>Video from our last Alaska tour: Moose encounter!</h2>
    <div class="gallery-video">
        <video controls>
            <source src="https://courses.oreillyschool.com/html5_responsive_desi
gn/videos/moose.mp4" type="video/mp4">
            <source src="https://courses.oreillyschool.com/html5_responsive_desi
gn/videos/moose.ogg" type="video/ogg">
            <source src="https://courses.oreillyschool.com/html5_responsive_desi
gn/videos/moose.webm" type="video/webm">
            Your browser does not support the <code>video</code> element.
            <img src="https://courses.oreillyschool.com/html5_responsive_design/
images/moose.jpg">
        </video>
    </div>
    <h2>Video from Iceland: Clouds!</h2>
    <div class="gallery-vimeo">
        <iframe src="//player.vimeo.com/video/113707290"
                width="500" height="281" frameborder="0"
                webkitallowfullscreen mozallowfullscreen allowfullscreen>
        </iframe>
    </div>
</section>
```

□ and □. You now see another video in the page. It looks pretty good, but we'd like the size to match the video above and also to resize with the width of the browser window. If you resize the window now, you'll see that the Vimeo video doesn't resize with the window, like our moose video does.

Now, you might think that setting the width and height of the Vimeo video like we did the moose video would work. It doesn't! Remember that the Vimeo video is nested within an <iframe> element, so we really don't have any control over the video itself. Instead we only have control over the <iframe> element that wraps the video, and the elements within which that <iframe> is nested. So, to style the video properly for the page, we need to nest the <iframe> inside another element—which we've done, in the element with the class "gallery-vimeo"—and then style both that containing element and the <iframe>. Modify gallery.css as shown:

CODE TO TYPE:

```
section {
    box-sizing: border-box;
    border-top: 1px solid black;
}
@media only screen and (min-width: 641px) {
    section#gallery {
        display: table-cell;
        width: 60%;
        padding: .8em 3% .8em 2%;
    }
}
@media only screen and (max-width: 640px) {
    section#gallery h1, section#gallery h2, section#gallery p {
        padding: .8em 3% .8em 2%;
    }
}
div.gallery-video video, div.gallery-video img {
    width: 100%;
    height: auto;
}
/* This will work for youtube and vimeo videos that are 16:9 */
div.gallery-vimeo {
    position: relative;
    padding-bottom: 56.25%; /* 281:500 */
    height: 0px;
}

div.gallery-vimeo iframe {
    position: absolute;
    top: 0px;
    left: 0px;
    width: 100%;
    height: 100%;
}
```

gallery.css and your **gallery.html** file. Now, the width of the Vimeo video matches the width of the moose video, and when you resize the browser, the Vimeo video expands and shrinks with the window, just like we wanted.

To make this work, we're using a bit of a trick, based on the size of the Vimeo video. If you look at the **width** and **height** properties on the <iframe> element, this tells you the aspect ratio of the video (it's set by Vimeo you when you generate the embed code; we'll look at this in more detail shortly):

OBSERVE:

```
<iframe src="//player.vimeo.com/video/113707290"
          width="500" height="281" frameborder="0"
          webkitallowfullscreen mozallowfullscreen allowfullscreen>
</iframe>
```

We use that size to compute how much space (padding) we need to allow for the height of the video to fit the <iframe> into the containing precisely.

First, we create the style for the containing element, using the class "**gallery-vimeo**." We **position the relative** so that we can later position the <iframe> inside it **absolute** (remember that absolutely positioned elements are positioned relative to their most closely positioned parent). Then we **set the padding-bottom to 56.25%** and the **height to 0px**. Huh?

OBSERVE:

```
div.gallery-vimeo {
    position: relative;
    padding-bottom: 56.25%; /* 281:500 */
    height: 0px;
}
```

We do this because we want to control the height of the and make sure it's the exact height we need for the <iframe>, no matter what the width of the browser page. So, we use **%** for the height of the padding so that the padding height is set relative to the *width of the containing element*. Yes! It sounds bizarre, but we set the height of the padding relative to the width of the "gallery" <section>. Why would we do this?

We want the video to take up the full width of the <section> element (that is, we want it to stretch and shrink with the browser window), and remember, the **width** of the <section> is the width of the *content* (not including margins, borders, and padding). The width of the <section> is determined by the width of the page: remember, we set the <section> in the wide view to 60% the width of the page, and in the narrow view, it's 100% the width of the page by default. We also applied some padding to the <section>—3% on the right and 2% on the left. Suppose you open up your browser window so that the width of the <section> is 500 pixels, *not including* the padding. In other words, the width of the content area of the <section> is 500 pixels. If the video is going to take up 100% of the content area of the <section>, then the width of the video will also be 500 pixels. So what do we want the height to be, if the width is 500 pixels? We want the height to be 281 pixels! (We know this because that's the actual width and height of the video). If you then widen the browser so that the content width of the <section> is 600 pixels, we want the height to be 337.2 pixels. Why? Because we maintain the same *aspect ratio* so the video doesn't look stretched. Let's take a closer look at how we compute that:

OBSERVE:

```
Aspect ratio of original video = (281 * 100) / 500 = 281/5 = 56.25

Height of the video at 600 pixels wide, using the same aspect ratio:
(56.25 * 600) / 100 = 56.25 * 6 = 337.2
```

If the width is 500 pixels, the height is 281 pixels, which is 56.25% of 500. If the width is 600 pixels, the height is 337.2 pixels, which is 56.25% of 600. As long as the height of the video is always 56.25% of the width, we know that the video will have the correct aspect ratio. The width of the video will equal the width of the containing <section>.

So why are we setting the **padding-bottom** property of the rather than the **height** property? Because height is calculated based on the height of the containing element, which is *not* related to the width of the browser window. By using the **padding-bottom** property to add height to the , we can make the size of the based on the *width* of the containing element.

That number, 56.25, may change if the aspect ratio of the video you're embedding is not the same as the one we're using here. So always check the width and height of the video you're embedding (by checking the attribute values that Vimeo or YouTube add to the <iframe>), and compute that number yourself. Remember, that number will be:

OBSERVE:

```
Height of padding in % = (height in pixels * 100) / width in pixels
```

The entire height of the element is determined by the padding, so we set the **height** property to **0px** so there is no other height added to that (so it will fit the <iframe> perfectly). Now, are you wondering about how the <iframe> will go into an element with no height for the *content*? That's where the absolute positioning comes into play.

Okay, that takes care of the containing element; what about the <iframe>? We position the <iframe> with **absolute positioning**:

OBSERVE:

```
div.gallery-vimeo iframe {
    position: absolute;
    top: 0px;
    left: 0px;
    width: 100%;
    height: 100%;
}
```

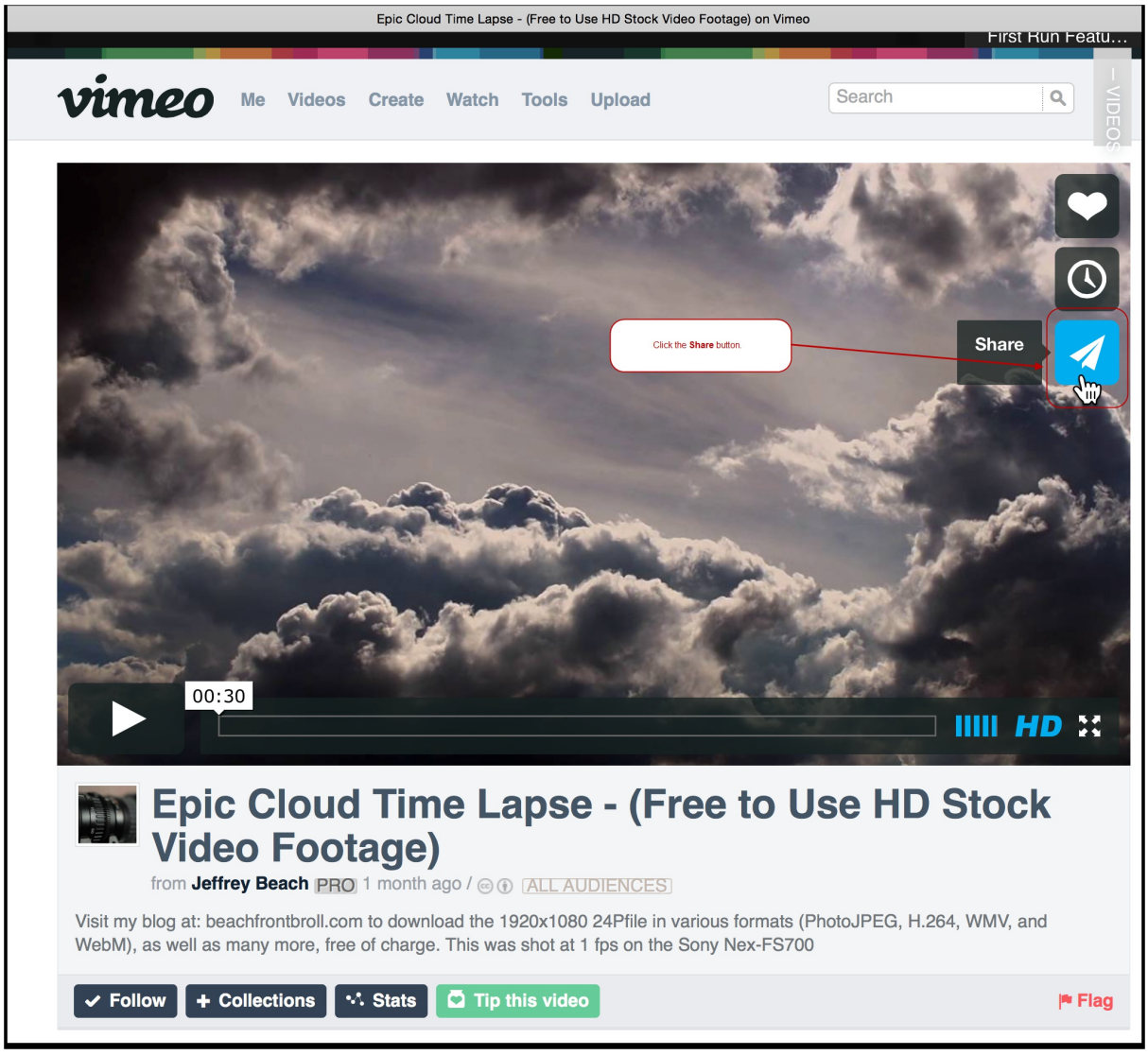
By positioning the <iframe> absolutely, we can specify that we want the **top left** corner of the <iframe> to be at the top left corner of the containing . We **set the width and height to 100%** to make sure that the <iframe> is the full width and height of the content (so we see the whole video) and fills the containing . Because we're positioning the <iframe> absolutely, this element is sitting outside the flow of the rest of the page. In other words, it is sitting *on top of* the padding we gave to the containing . So, the height of the that *is* sitting in the normal flow of the page allows the height of the page to work correctly, and gives us an element to place the <iframe> on top of, with the same exact width and height.

Whew! That was complicated to explain, but actually really straightforward to code, once you know how to compute the % used to create the **padding-bottom** element of the containing element.

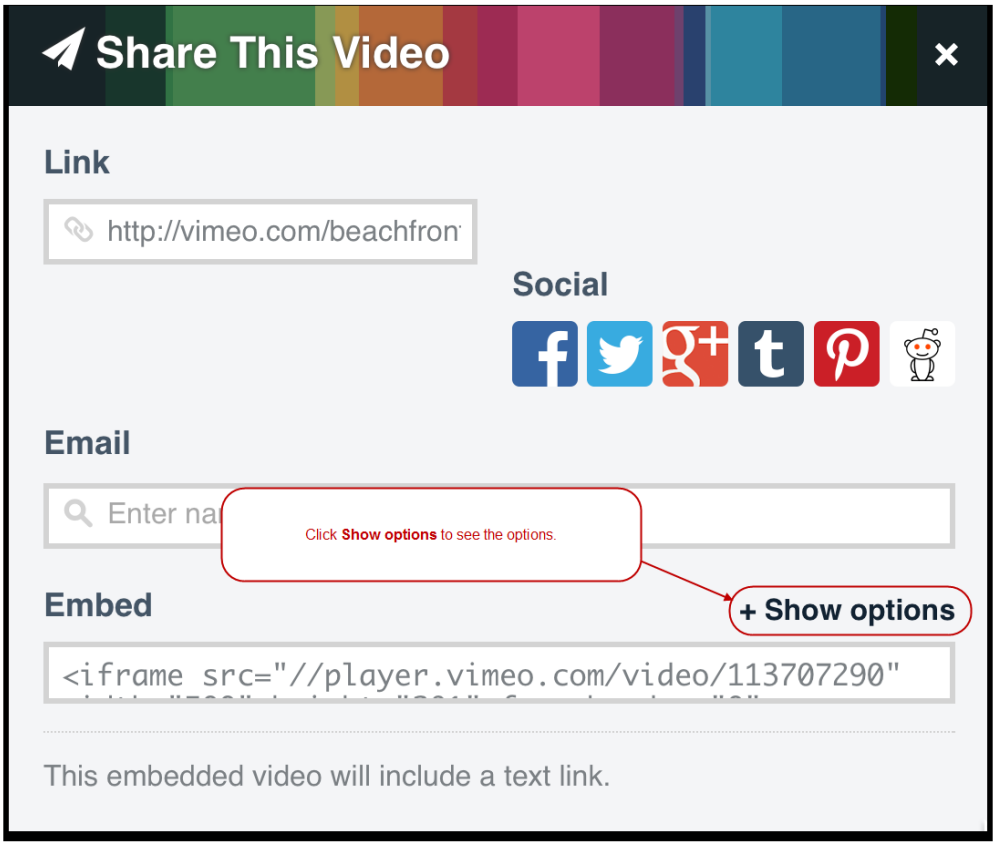
Before we finish off this section of the lesson, let's take a quick look at how to generate the <iframe> you'll use from Vimeo. The process is similar for YouTube and many of the other services you can use to embed video.

Note Because these services change fairly frequently, the web pages may look a little different from these screenshots, but the process should be about the same.

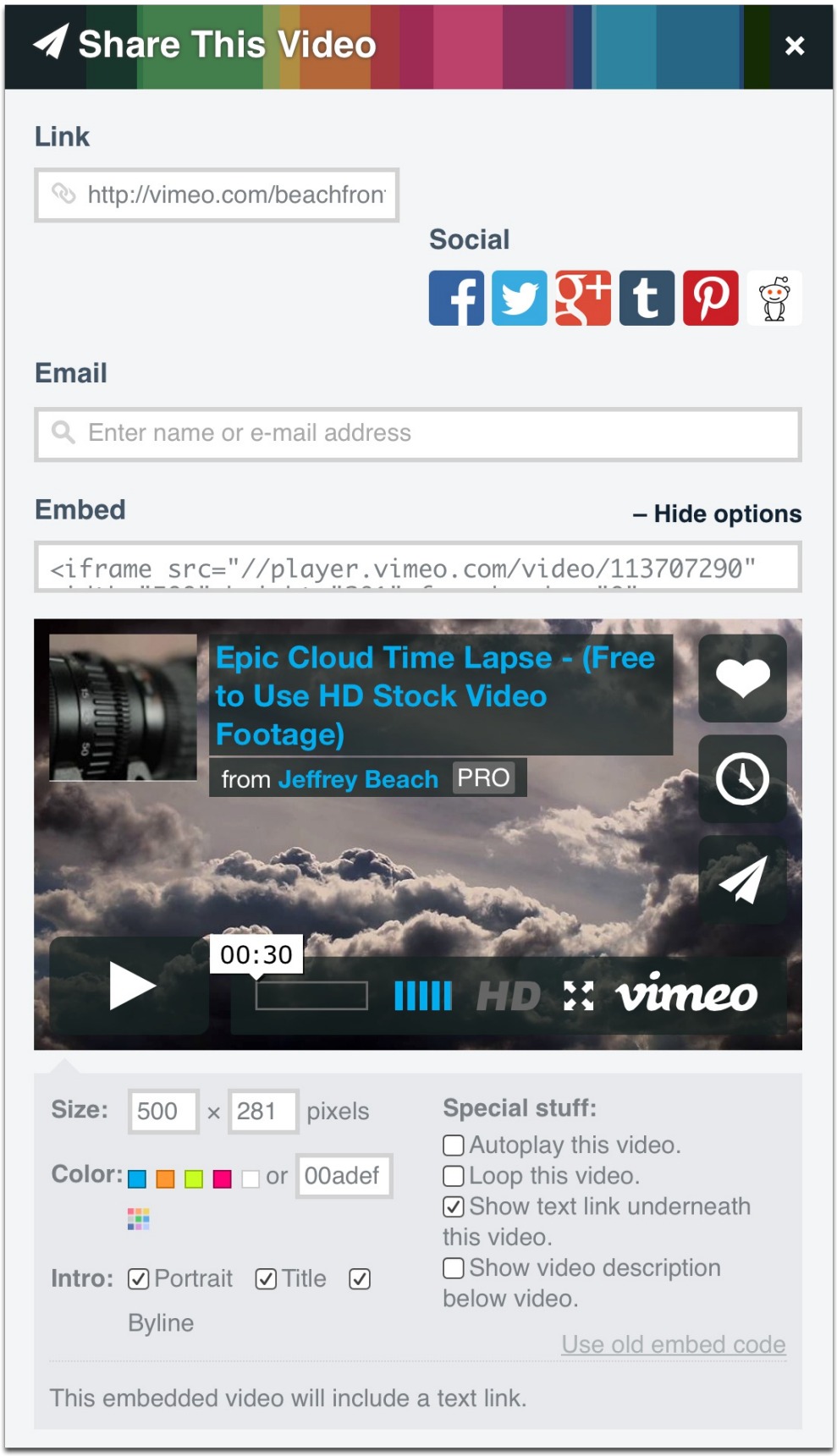
The video we used at Vimeo is at https://vimeo.com/113707290. To generate the embed code, first, click the **Share** button:



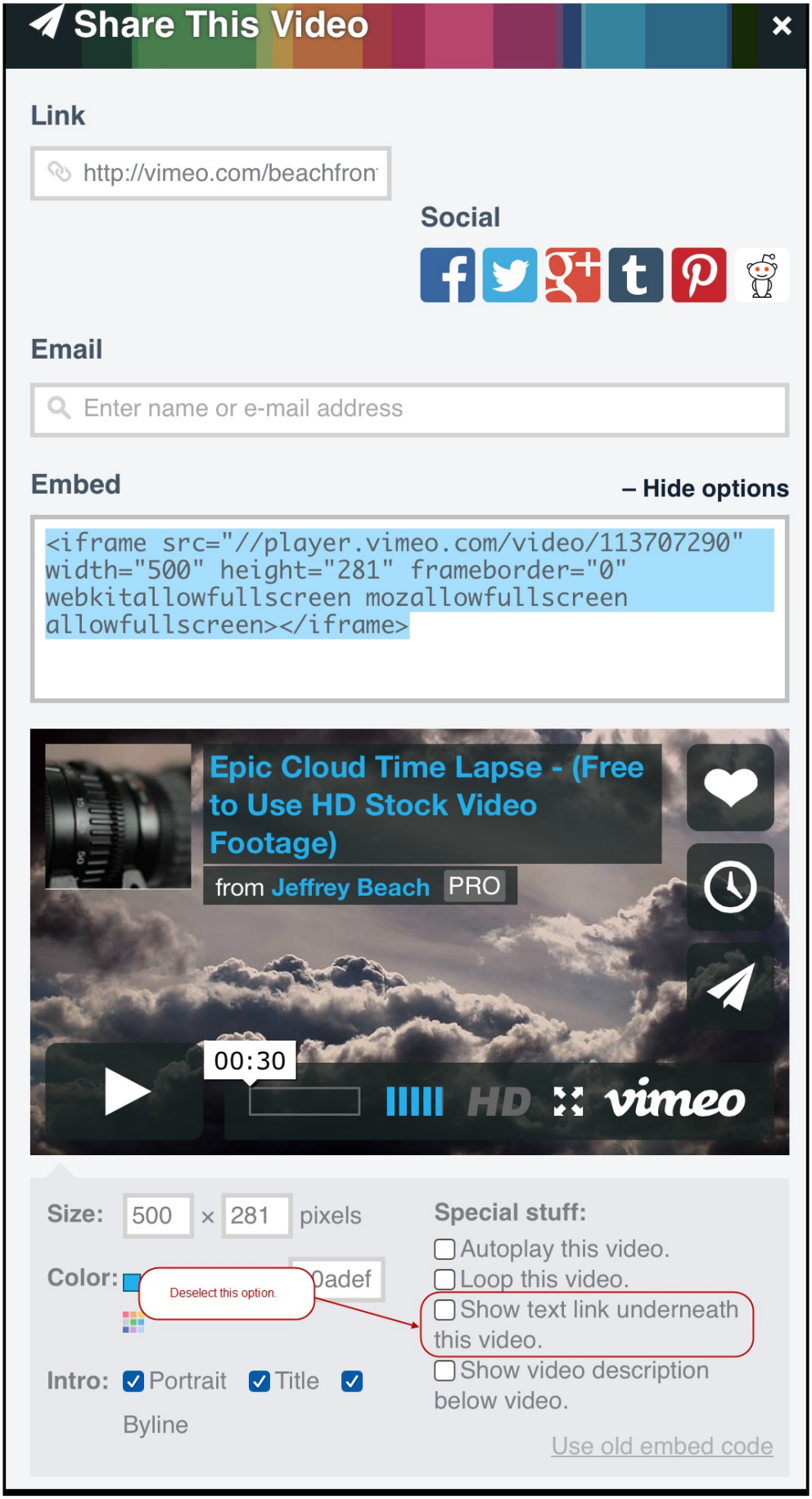
A dialog box pops up:



Click **Show Options** to see more options for customizing the embed code:



Here you can customize a few things to determine the text associated with the video, the colors of the various pieces of the video player, and even set things like whether the video should autoplay or loop. We don't want any text under the video, so uncheck the "Show text link underneath the video" option. Then look at the HTML generated above the video. This is the code you want to copy and paste into your web page. Just select this text, and copy:



And that's it! You're ready to go with an embedded video. Just paste it into your HTML as we did above:

OBSERVE:

```
<iframe src="//player.vimeo.com/video/113707290"
          width="500" height="281" frameborder="0"
          webkitallowfullscreen mozallowfullscreen allowfullscreen>
</iframe>
```

Progressive vs. Streaming Video

The distinction between how a video is loaded and played by the <video> element and how a video is loaded and played by a site like Vimeo is important. Typically, video served from a site like Vimeo is *streamed*. This means that the video is sent from a special streaming server to the video player and is not stored locally on your computer. It also means that you can click forward in the video and the streaming will pick up there, usually pretty quickly, because the data rate of the stream is set specifically to match the bandwidth of your access point, and the streaming player on the server can stop streaming where it was before, and start streaming where you clicked, without having to stream the part of the video in between.

In contrast, a video that's loaded and played *progressively* is downloaded to your machine and stored locally (it's cached by the browser). The data rate of the video you're downloading is set when the video is created so it's not adjusted based on your bandwidth. If you click forward in the video, you have to wait until the video has been downloaded to that point before it begins playing again. One big advantage of progressively downloaded video is that once it's downloaded and cached on your local machine, it's very fast to play it again (as long as it remains in the cache).

In both cases, the video can start playing right away. However, the progressively loaded video may stop and start depending on whether the download is keeping up with the speed at which it's playing.

It's a fairly straightforward process to offer a progressive video download on your website: you simply store the video files and link to them in the <video> element. That's it. It's a lot trickier to offer streaming video with your own streaming server because you need a server that's specifically designed to stream video. Many small businesses end up using a service, like YouTube, Vimeo, Brightcove, or one of many others, for hosting and serving streaming video.

SVG and Responsive Design

Lesson Objectives

When you complete this lesson, you will be able to:

- describe the difference between raster image formats, like JPEG, GIF, and PNG, and vector image formats, like SVG.
- add a SVG file to your page and style it.
- describe when and why to use SVG images.

SVG in Responsive Design

In this lesson, we'll finish up the footer of our Dandelion Tours website, and replace the text linksto the various social media sites with images. But rather than using icons in the PNG, JPEG, or GIF formats (the typical image formats you see on the web), we're going to use a format called called **Scalable Vector Graphics**, or **SVG**. Why use SVG? For some uses, SVG is more efficient and responsive than these other formats. So we'll take a look at how SVG is different and why and when you might want to use it in your responsive designs. You'll then use those same SVG skills to add a map to the Gallery page.

Adding SVG Images to the footer

Let's get started by replacing the social media text links with images in the footer. Edit your **index.html** file, and make the changes below:

Modify index.html as shown:

```
...
<footer>
    <div>
        Have a question? Email us at
        <a href="mailto:example@example.com">travel@dandeliontours.com</a>
    </div>
    <div>
        <a class="social" href="#">Facebook</a>
        <a class="social" href="#">Twitter</a>
        <a class="social" href="#">Instagram</a>
        <a class="social" href="#">Youtube</a>
    </div>
    <div>
        <a class="social" href="#"><img src="https://courses.oreillyschool.com/html5_responsive_design/images/facebook.svg"></a>
        <a class="social" href="#"><img src="https://courses.oreillyschool.com/html5_responsive_design/images/twitter.svg"></a>
        <a class="social" href="#"><img src="https://courses.oreillyschool.com/html5_responsive_design/images/instagram.svg"></a>
        <a class="social" href="#"><img src="https://courses.oreillyschool.com/html5_responsive_design/images/youtube.svg"></a>
    </div>
    <div>
        You'll travel with the wind. <span class="name">Dandelion Tours</span>.
    </div>
</footer>
</body>
</html>
```

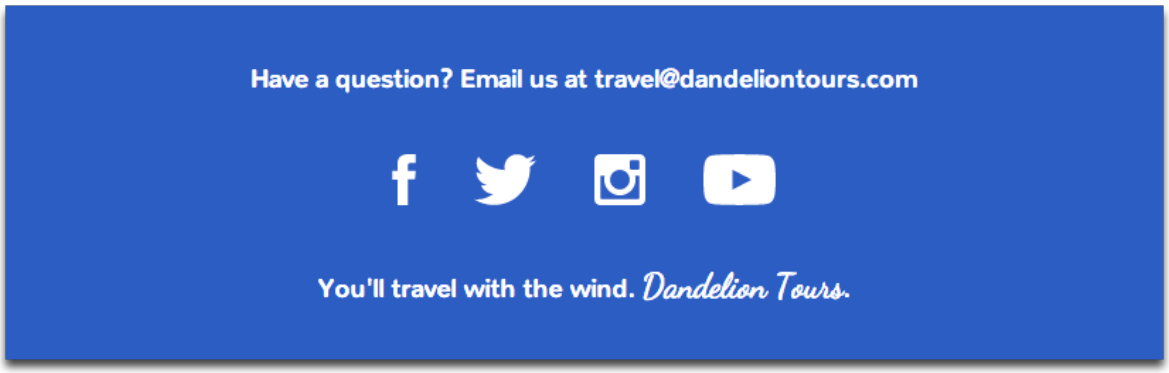
Make this same change to your other HTML files too: **about.html**, **travel.html**, **contact.html**, and **gallery.html**. your **index.html** file. The social media images you just added are HUGE! You might be wondering why we're using such large images. We'll get to that shortly... but for now, let's get the CSS for the footer updated so the images fit the page a bit better.

Open **dt.css** and make the following changes (remember that the CSS for the footer is in **dt.css** because it's shared across all the HTML files for each page in the site):

Modify dt.css as shown:

```
...
body > footer {
    margin: 0px;
    padding: 1.25em;
    background-color: rgba(39, 69, 189, .9);
    color: white;
    font-size: .85em;
}
body > footer div {
    margin: auto;
    width: 70%;
    text-align: center;
    padding: 1.25em 0px;
}
body > footer .social {
    padding: 0em 1em;
}
body > footer .social img {
    vertical-align: middle;
    height: 2em;
}
body > footer .name {
    font-size: 1.2rem;
}
```

dt.css and **index.html**. Now the images in the footer look a *lot* better:



In the HTML, each of our SVG images is nested within a link with the class "social":

OBSERVE:

```
<a class="social" href="#"><img src="http://courses.oreillyschool.com/html5_responsive_design/images/facebook.svg"></a>
```

So we select all the images nested within elements with the class "social," and in the CSS, we align them to the middle (so they look centered with each other), and set the **height** to **2em**. By using **2em** for the height, we size the images based on the font size of the footer (the closest parent element with a font size set). We set the **font-size** of the footer to **.85em**; which you'll remember means the font size of the footer is .85 the size of the body, which is 100% (16px in most browsers by default, unless the user has changed it in their specific browser). So we set the height of the social images relative to the body font size; what about the width? We don't specify a width, so the browser, by default, just maintains the aspect ratio of the image, so it increases the width of the image appropriately, based on the original height/width ratio of the image.

The end result is that if you decide later to increase the font size of the body to, say, 20px, you'll see that the font size of the footer increases, and the size of the social images increases, because both are sized relative to the font size of the body.

Note We haven't added actual links to the <a> elements with the class "social" because Dandelion Tours isn't real and so doesn't have social media presence on any of those sites; in the real world, these icons would link to the pages for Dandelion Tours on each service. Feel free to add your own links here!

What is SVG?

Okay, so we have SVG images added to our footer, but we haven't talked about what SVG is, or why we're using it here. Let's start by looking at what SVG is. As we said earlier, SVG stands for "Scalable Vector Graphics". However, even though an SVG file displays as an image in the browser, the content of the file itself is actually text: it's a variant of XML (as is HTML), so you can open up an SVG file in your editor and look at it!

Let's take a look at the SVG file **twitter.svg**:

OBSERVE:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1
.1/DTD/svg11.dtd">
<svg version="1.1" id="Layer_1" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="
http://www.w3.org/1999/xlink" x="0px" y="0px"
     width="412px" height="334.824px" viewBox="0 0 412 334.824" enable-backgroun
d="new 0 0 412 334.824" xml:space="preserve">
<path id="twitter-icon" fill="#FFFFFF" d="M412,39.633c-15.158,6.724-31.449,11.26
9-48.547,13.31
    c17.449-10.461,30.854-27.025,37.164-46.764c-16.333,9.687-34.422,16.721-53.67
6,20.511c-15.418-16.428-37.388-26.691-61.698-26.691
    c-54.56,0-94.668,50.916-82.337,103.787c-70.25-3.524-132.534-37.177-174.223-8
8.314c-22.142,37.983-11.485,87.691,26.158,112.85
    c-13.854-0.438-26.891-4.241-38.285-10.574c-0.917,39.162,27.146,75.781,67.795
,83.949c-11.896,3.236-24.926,3.979-38.17,1.447
    c10.754,33.58,41.972,58.018,78.96,58.699C89.604,289.692,44.846,302.131,0,296
.846c37.406,23.982,81.837,37.979,129.571,37.979
    c156.932,0,245.595-132.553,240.251-251.436C386.339,71.471,400.668,56.584,412
,39.633z"/>
</svg>
```

You can look at the file in your own editor too. Right-click the image in the footer of the Dandelion Tours page, click **Save as...**, and save the image on your computer. Then, you can open it in any text editor (like Notepad or Text Edit).

At the top of the file is the XML directive, which tells the browser this file is XML (rather than HTML, which is the default type the browser expects), followed by the doctype. This is analogous to the HTML doctype you put at the top of your HTML files, only a lot more complicated.

Below the doctype is the opening <svg> tag (you'll see the matching closing tag at the bottom of the file), which is analogous to the <html> element. Again, it's much more complicated than the <html> element. Finally, the <path> element is nested in the <svg> element. This <path> element is where the information for the image itself is provided: the path spells out the shape of the image you're seeing in the page, in this case, for the Twitter icon (the one that looks like a bird).

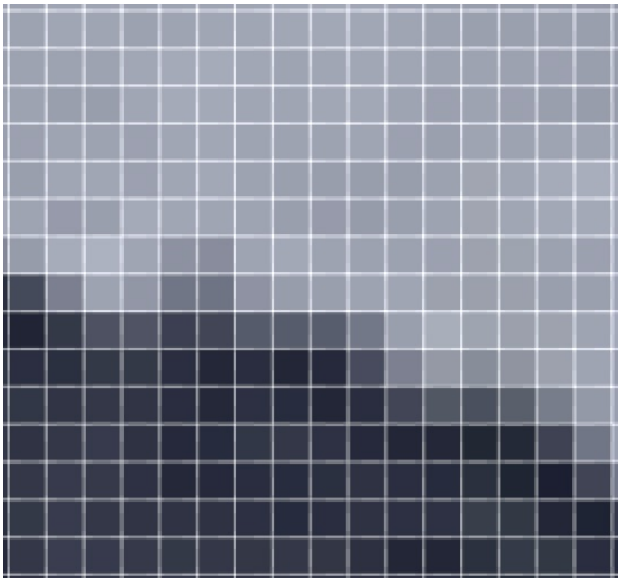
Now, clearly, you can't write SVG image paths easily yourself using text. However, many graphics manipulation programs, like Illustrator, GIMP, and many others, allow you to draw an image and save it as SVG, meaning the SVG file is generated for you, so you don't have to worry about any of these complexities.

So what's the difference between SVG and the other image formats we're used to using on the web, and when and why would you use SVG?

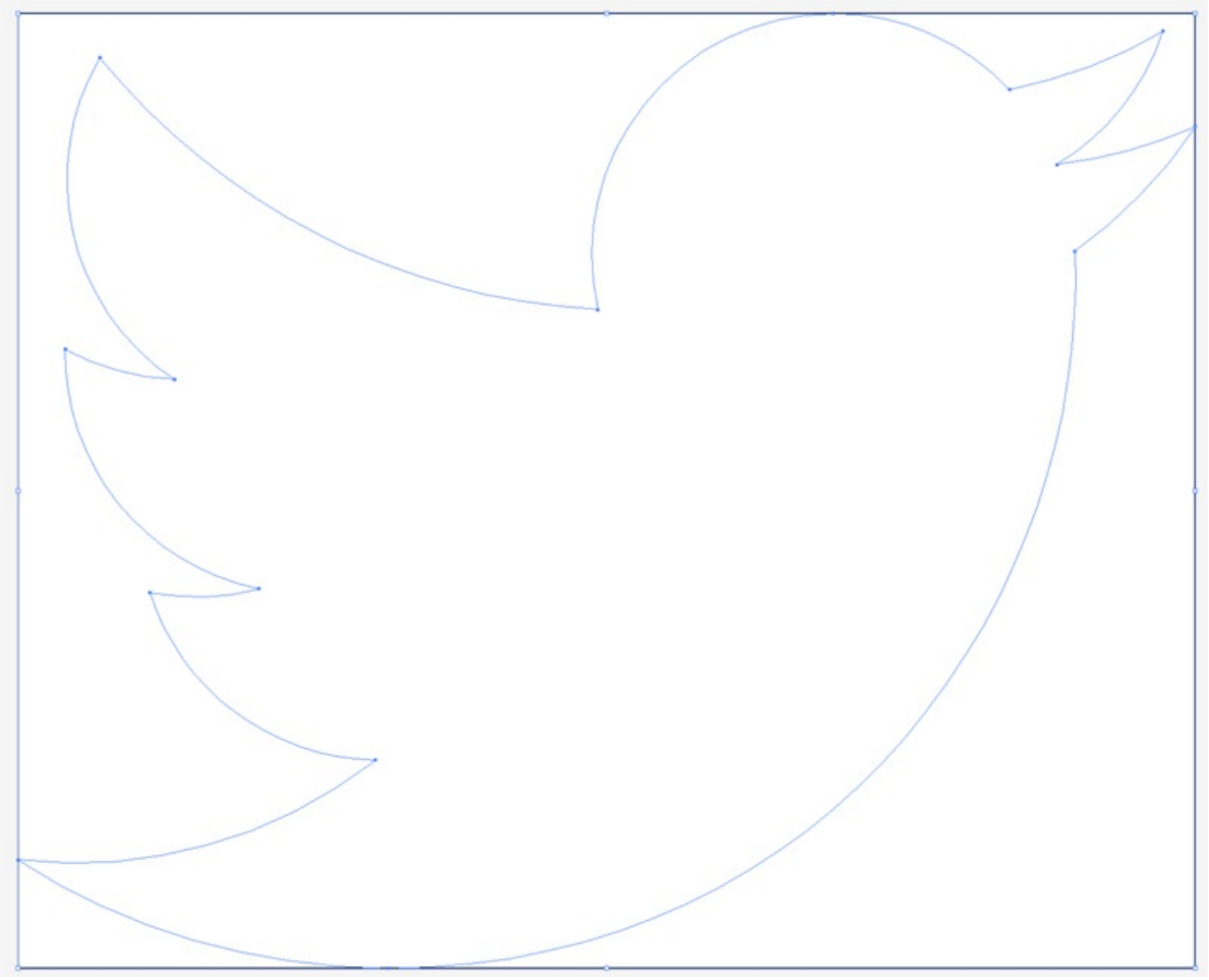
As you've already seen by looking at the SVG file, the SVG graphics format doesn't specify an image in terms of *pixels*; but in terms of one or more *paths*. These paths can get incredibly complex, but ultimately, you're telling the computer how to draw lines to make shapes, and how to fill those shapes, rather than specifying the individual pixels that make up an image. This is why it's called a *vector* image format; we use vectors (paths) to represent the image instead of pixels (although ultimately, the path is used to generate pixels in the browser to display on your monitor).

Other formats like GIF, JPEG, and PNG are all **raster** image formats that specify an image in terms of its **pixels**. If you open one of the JPEG images we use in Dandelion Tours and zoom way in, you'll see

something like this:



The contents of a **raster** image file are individual pixels. If you open an SVG file in a program like Adobe Illustrator, you'll see this:



You can zoom into this image and you'll never see pixels; you'll always see lines making a shape. You can scale the shape up so it's bigger, or down so it's smaller, but the end result is the same: you have lines (*paths*) that outline the shape of the icon (in this case, the Twitter icon). That's why it's called *scalable* vector graphics; because we're specifying an image using paths (vectors), we can *scale* the image up or down as much as we want without losing quality.

That scaling is why, sometimes, SVG is a good choice for graphics in a page. Unlike a photo that's stored in

the JPEG format, you can scale an SVG image way up or way down and the quality of that image stays *exactly the same*. As you saw previously, that's not the case with a raster format like JPEG, PNG, or GIF. If you scale a JPEG image down, then some of the pixels must be thrown away to fit the image into a smaller space. Likewise, if you scale a JPEG image up beyond its actual size, then the computer has to fill in the image with more pixels by guessing the appropriate colors to use to keep the image looking about the same. Neither of these processes is perfect, so scaling a raster image always results in small imperfections (usually not visible to end users of the web except when scaling a small image up very large though). It also requires processing power for the browser to do the scaling.

Another reason to use SVG for some images is that they tend to be smaller files, and so require less downloading time for the browser, which is especially important in mobile applications. For our use case, the images are small anyway (and only one color), so the image size would not be a problem in any format. However, if we wanted to scale one of these icons up very large, using SVG would make a huge difference in terms of download speed. Can you see why? To represent a very large Twitter icon, say 800 pixels by 800 pixels, using JPEG would require a file that can store 640000 pixels (800 times 800). Compare that to the size of the same image using JPEG if we want to display the image at 100 pixels by 100 pixels, for a total of 10000 pixels. Clearly, the 640000-pixel image file will be much larger than the 10000-pixel image. In SVG, the file size is exactly the same for the image no matter what size we want it to be displayed, because the SVG file contains the instructions for how to draw the shape and fill it. Those instructions are the *same*, no matter what size the image is displayed.

To perfect the footer of the page, let's add a media query to reduce the size of the social media icons when the width of the browser is narrower than 640px. That way the icons will fit better on the page when the window is small, or you're using a mobile browser. Add the following code to the bottom of dt.css:

CODE TO TYPE:

```
@media only screen and (max-width: 640px) {
    body > footer .social img {
        height: 1.2em;
    }
}
```

dt.css and **index.html**. All we did here is decrease the **height** of the SVG images from 2em to 1.2em if the width is less than 640 pixels. Change the width of your browser to switch into the narrow view. It will be difficult to see the icons' size change in the footer because the footer will jump down when the page layout changes. So once you're in the narrow view, scroll down to the bottom of the page, and slowly widen the page again. When it switches to the wide view, the icons will increase in size.

Adding an SVG Map to the Gallery

With the advantages we've listed above for SVG, you might wonder why we wouldn't want to use SVG for *all* our images. For photos, SVG isn't appropriate. Photos are originally created as pixels, and there's no good way to represent photographs as vectors. For icons or silhouettes, it makes more sense to use SVG, especially if you think you'll need to scale the image. Another good example of when to use SVG is with maps.

A good source for SVG maps is Wikipedia. Here's an example of a map of Iceland we downloaded from Wikipedia:

As you can see, just like the Twitter icon, the map of Iceland consists of lines (in the shape of the outline of Iceland, the regions within Iceland and islands off the coast), and a fill (the color of the ocean outside the coastline, and the color of the island inside the coast line). There are four more areas inside the island representing national parks. If you edit the file **iceland.svg**, you'll see that in this case, the file is made up of paths and polygons (a polygon is just a compound path). It's a far more complex file than the **twitter.svg** file, but it's the same idea. The size of the file is 211,722Kb, which is about what a high quality JPEG of the image would be at about 800px by 600px. The advantage of using an SVG here is that we can scale the map as large or small as we want without any loss of quality. This would be particularly important in a map-based application (think of zooming into a Google map and how you don't lose quality; you'd want the same zooming ability in your own map application if you were to make one).

Let's add this map of Iceland to the Gallery page, and overlay a point showing the location of Reykjavik, the capital of Iceland, where the video of the clouds on the Gallery page was shot. Modify gallery.html as shown:

CODE TO TYPE:

```
<section id="gallery">
    <header>
        <h1>Preview your tour</h1>
    </header>
    ...
    <h2>Map of Iceland: here's where we took the video!</h2>
    <div class="gallery-svg">
        <img src="https://courses.oreillyschool.com/html5_responsive_design/imag
es/Iceland.svg">
        <div id="overlay">
            <div id="Reykjavik" class="point" data-color="#DD3626"
                data-x="25" data-y="70">
            </div>
        </div>
    </div>
</section>
```

□ and □. The map of Iceland appears, but it's the wrong size and the overlay point is not in the correct location. To fix this, add the follwing code to the end of gallery.css:

CODE TO TYPE:

```
div.gallery-svg {
    position: relative;
}
div.gallery-svg img {
    width: 100%;
    height: auto;
    display: block;
}
div.gallery-svg div.overlay {
    position: absolute;
    top: 0px;
    left: 0px;
}
div.point {
    position: absolute;
    width: 20px;
    height: 20px;
    border-radius: 10px;
    background-color: blue;
    top: 70%;
    left: 25%;
}
```

gallery.css and **gallery.html**. Now, the map fits in the page perfectly in the page, and you see a dot on the map where Reykjavik is located. If you change the width of the browser, the map will change sizes too. If you make the browser narrower than 640px, you'll see it fits in the narrow view just like the videos do, taking up the full width of the page. If you try the page on your mobile device, the map will work there too (SVG is supported in every browser: http://caniuse.com/#search=svg). Finally, if you have a large monitor and you expand the web page to take up the full width of your monitor, the map expands and looks just as good at a very large size as it does at a smaller size. That's the advantage of SVG!

Let's go over the HTML and CSS for the map with the overlay. In the HTML, we **use the "gallery-svg" ** to wrap the map SVG image and the s we're using to create the point. This is positioned **relative** so we can position other elements relative to it. The image is nested inside this and given a **width** of **100%** and a **height** of **auto** so it fills the . We also set the **display** to **block** to eliminate the space we get below an inline element otherwise.

OBSERVE:

```
<div class="gallery-svg">
    <img src="https://courses.oreillyschool.com/html5_responsive_design/images/Iceland.svg">
    <div id="overlay">
        <div id="Reykjavik" class="point" data-color="#DD3626"
            data-x="25" data-y="70">
        </div>
    </div>
</div>
```

We added **three custom data-* attributes** to the with the id "Reykjavik." We won't use these in this example, but you could potentially use them to position the "point" on the map and give it a color using JavaScript. If you had a map with several points you wanted to add, writing JavaScript code to add them to the map would be a good solution. Another solution would be to add the points to the map yourself, but to do that you'd need to edit the SVG file using a program like Illustrator or GIMP.

The "overlay" is positioned **absolute**, relative to the "gallery-svg" so that the "point" sits on top of the map. The "point" is positioned **absolute**, 70% from the top of the map and 25% from the left of the map. By using % to position the point, the point stays in the same position on the map as you change the width of the browser. We set the **border-radius** of the "point" to half the size of its **width** and **height**, so the is shown as a circle. Finally, we set the **background-color** to **blue** so it shows up well on the map.

Scalable vector graphics, or SVG, is another tool in your responsive design toolbox. Now you know when you might want to use SVG images, and how to add them to your website.

Congratulations! You've almost completed the course! You're well on your way to becoming a responsive designer for web applications. As you've seen, the keys to responsive website design are making it look good, make sense, and offer a good user experience at a variety of screen sizes and on a variety of devices. We can't solve every potential web design problem with responsive design, but we can do a lot. In this course, you've learned:

- how to create solid HTML structure for the content in your pages that will work well with responsive design.
- how to use media queries, in your HTML and in your CSS to provide different experiences at different screen widths.
- how to use the viewport meta tag to set up your pages so they work well on different devices.
- how to layout your pages at different screen widths, and in such a way that the content makes sense at different screen widths.
- how fonts and images work in responsive design.
- how to include web fonts in your page.
- what pixel density means and how it affects your design.
- how to create responsive forms in your page that work at different screen widths and use input types that allow validation and visual feeback for the user.
- how to add video to your page so that it works in a variety of different settings and is sized correctly for the page.
- how to use SVG in responsive design.

We've covered a lot of ground and now you're ready to tackle the final project. Good luck!

Final Project

Final Project

In this course, you have built a responsive front-end for the Dandelion Tours website. Well done!

Now it's time for you to build a responsive website for yourself. You can choose to use the topic you selected way back at the beginning of the course, or you can choose a completely new topic—it's up to you.

Look back at your solution for the project for the **Goals** lesson where you considered the goals for your final project. Revise these if necessary (or create new ones if you're selecting a completely new topic).

Your final project website should include at least three or four pages, with navigation between the pages, and at least two different layouts for pages for different browser widths. Consider how your site might be used by mobile users (even if you don't have a mobile device, you should be able to think through the issues based on what you've learned in this course). Make sure you incorporate the following features in your website:

- Appropriate HTML structure for the content you're creating for your site, including the proper elements for content (like headers, navigation, headings, and so on.).
- Navigation that will work on both desktop and mobile browsers, and be clear and easy for users to understand.
- Layout designs and structure that work at all browser widths. Use your best judgement about what looks good at the various browser widths. Feel free to take a look at some responsive templates and other responsive websites you find online as examples of design, although obviously you need to submit your own work for the project (in other words, borrow ideas, but don't steal code!).
- Include at least one web font in your site.
- Include a few images in your site—use your own, or public-domain, royalty- and copyright-free images. If you use images with a share-alike-with-attribute Creative Commons license, make sure you include the attribution in comments in the HTML and/or CSS you submit. The images can be of any format that works for each use case (JPEG, GIF, PNG, or SVG).
- Include a video in your site—either your own, or a public-domain, royalty- and copyright-free video you've downloaded or included with Vimeo or YouTube.
- Include a form in your site.

Be creative! We look forward to seeing your amazing responsive website!